

# Blender - Python API

#4



Serdar ARITAN

Department of Computer Graphics  
Hacettepe University, Ankara, Turkey



Understanding how meshes are defined and created is essential for scripting geometry within Blender. The process is fairly straightforward and requires the user to define the following mesh attributes:

- Vertices (Points defined by X, Y, and Z)
- Edges (Wireframe curves defined by vertex indices)
- Faces (3D surfaces defined by vertex indices)



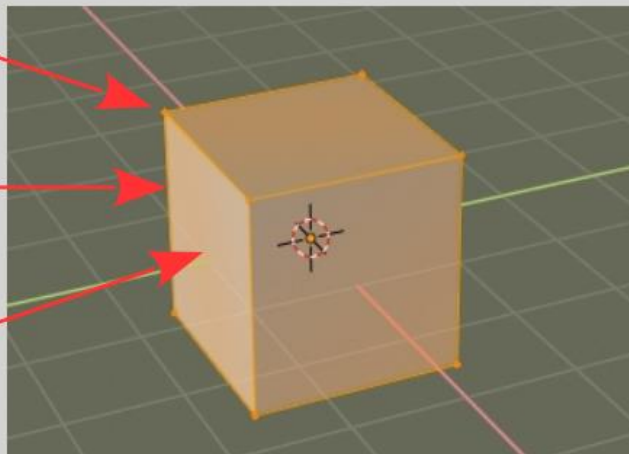
## Blender Mesh Definition

The default Blender Scene contains a Cube Mesh Object which, by default, is selected in Object Mode. With the Cube selected (orange outline) press the Tab key to enter Edit Mode to see the basic components of the Cube.

**Vertices** – corner intersection points.

**Edges** – joining the Vertices.

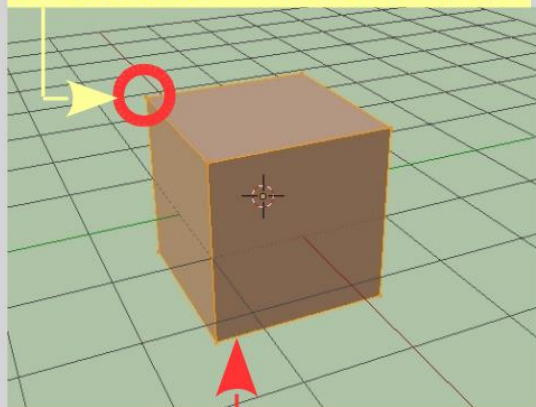
**Faces** – the area surrounded by the Edges.



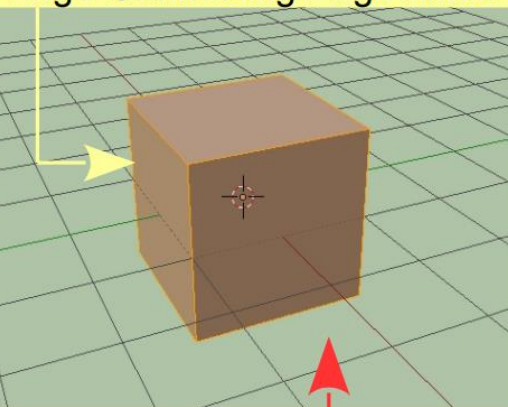


## Blender Mesh Definition

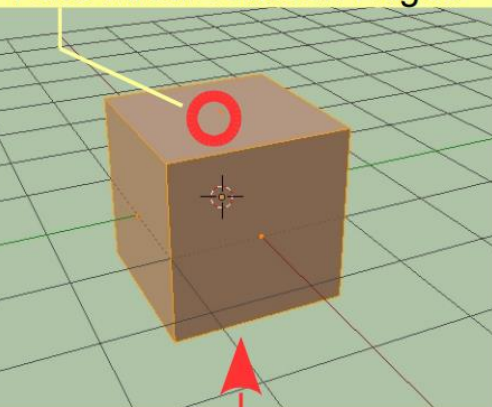
Vertex- Mesh Intersection Point



Edge-Connecting Edge Lines

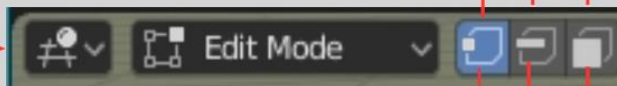


Face-Area Between Edges



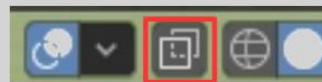
3D Viewport  
Editor Header

Selection Buttons



Vertex  
Edge  
Face

Show X-Ray  
(Show whole scene  
transparent(Toggle On/Off))

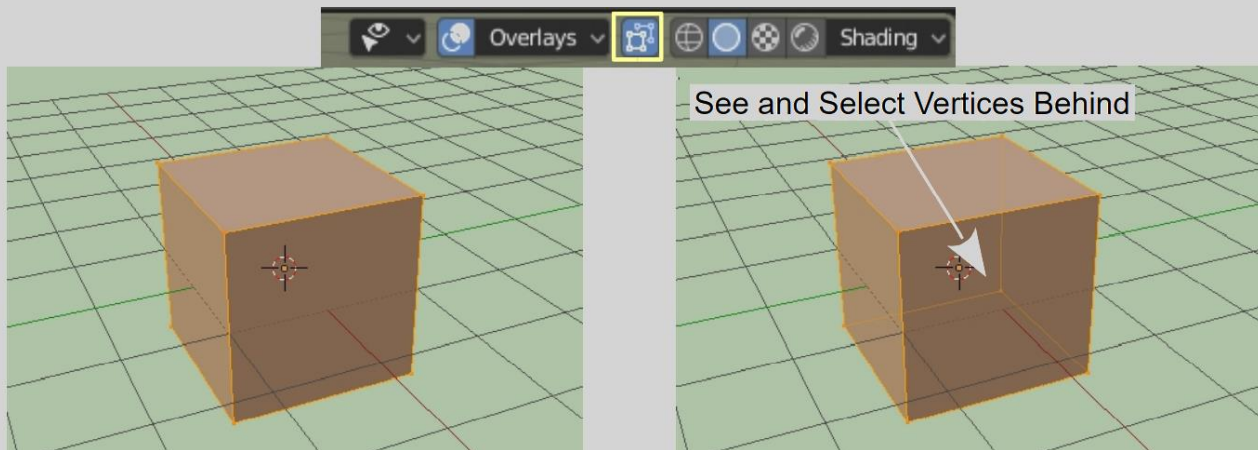






## Blender Mesh Definition

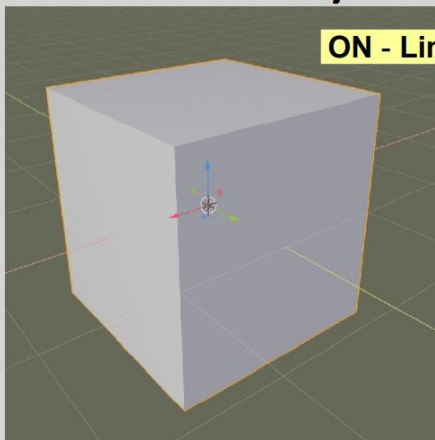
Only visible Vertices, Edges or Faces are available for selection. This means that you can only select the Vertices, Edges or Faces that you actually see in the Editor. Blender has a Show X-Ray function, This function is toggled on and off in the 3D View Editor Header by clicking the X-Ray button.





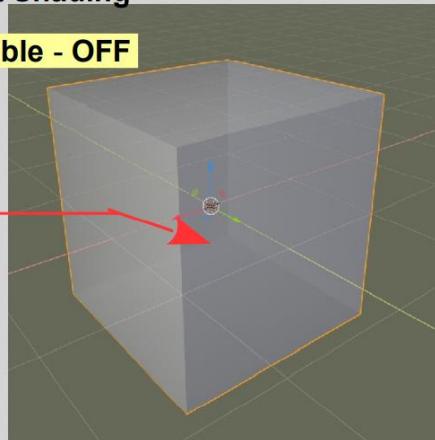
## Blender Mesh Definition

You can not select Vertices in Object Mode but there are occasions when you may wish to see hidden geometry.



ON - Limit Selection to Visible - OFF

Interior Geometry Visible





## Blender Mesh Definition

### Simple Mesh Definition 4-Corner Plane

```
import bpy
```

```
#Define vertices, faces
```

```
#The vertex array contains 4 items with X, Y, and Z definitions
```

```
verts = [(0,0,0), (0,5,0), (5,5,0), (5,0,0)]
```

```
# the faces array contains 1 item.
```

```
# The number sequence refers to the vertex array items.
```

```
# The order will determine how the face is constructed.
```

```
faces = [(0,1,2,3)]
```

```
#Define mesh and object
```

```
me = bpy.data.meshes.new('Plane')
```

```
#the mesh variable is then referenced by the object variable
```

```
ob = bpy.data.objects.new('Plane', me)
```

```
#Set location and scene of object
```

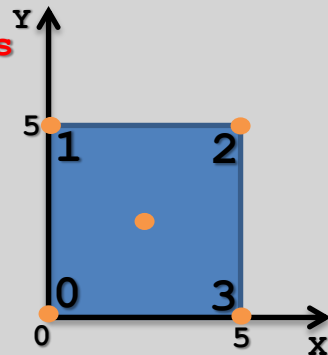
```
ob.location = bpy.context.scene.cursor.location # the cursor location
```

```
bpy.context.collection.objects.link(ob) # linking the object to the collection
```

```
#Create mesh
```

```
me.from_pydata(verts, [], faces)
```

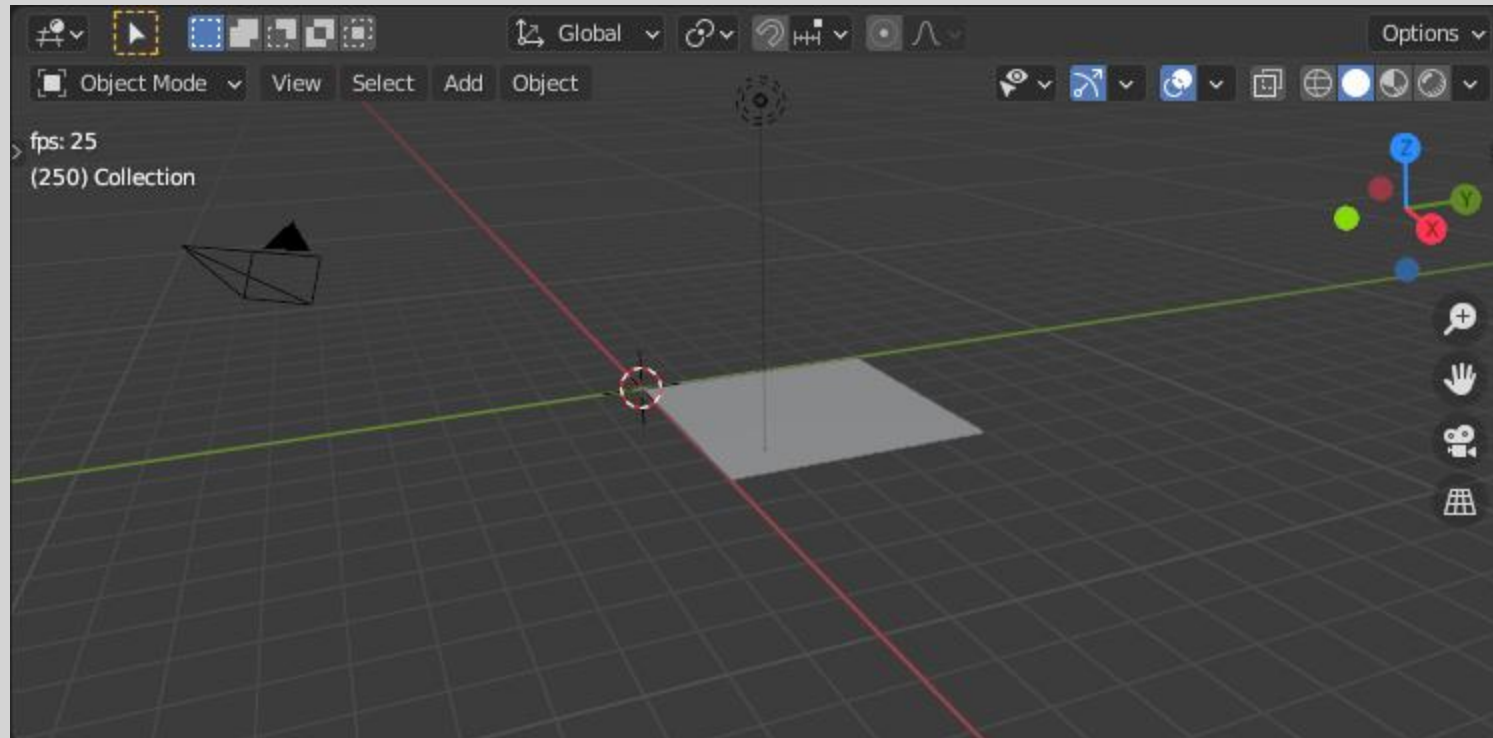
```
me.update(calc_edges = True) # Calculate edges
```



No Edge Data

# Blender Mesh Definition

## Simple Mesh Definition 4-Corner Plane





## Blender Mesh Definition Cube

```
import bpy
```

```
#Define vertices, faces
```

```
verts = [(0,0,0), (0,5,0), (5,5,0), (5,0,0), (0,0,5), (0,5,5), (5,5,5), (5,0,5)]
```

```
faces = [(0,1,2,3), (4,5,6,7), (0,4,5,1), (1,5,6,2), (2,6,7,3), (3,7,4,0)]
```

```
#Define mesh and object
```

```
me = bpy.data.meshes.new('Cube')
```

```
#the mesh variable is then referenced by the object variable
```

```
ob = bpy.data.objects.new('Cube', me)
```

```
#Set location and scene of object
```

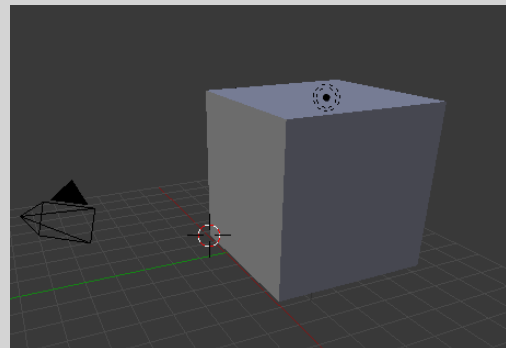
```
ob.location = bpy.context.scene.cursor.location # the cursor location
```

```
bpy.context.collection.objects.link(ob) # linking the object to the scene
```

```
#Create mesh
```

```
me.from_pydata(verts, [], faces)
```

```
me.update(calc_edges = True)
```



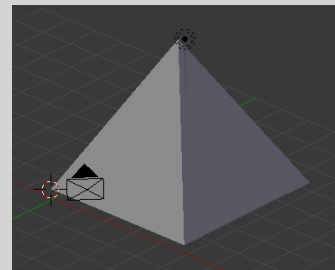




## Blender Mesh Definition Pyramid

```
import bpy
```

```
# Pyramid demonstrates how to create faces using 3 indices instead of 4
# Define vertices, faces
verts = [(0,0,0), (0,5,0), (5,5,0), (5,0,0), (2.5,2.5,4.5)]
faces = [(0,1,2,3), (0,4,1), (1,4,2), (2,4,3), (3,4,0)]
# Define mesh and object
me = bpy.data.meshes.new('Pyramid')
# the mesh variable is then referenced by the object variable
ob = bpy.data.objects.new('Pyramid', me)
# Set location and scene of object
ob.location = bpy.context.scene.cursor.location # the cursor location
bpy.context.collection.objects.link(ob) # linking the object to the scene
# Create mesh
me.from_pydata(verts, [], faces)
me.update(calc_edges = True)
```

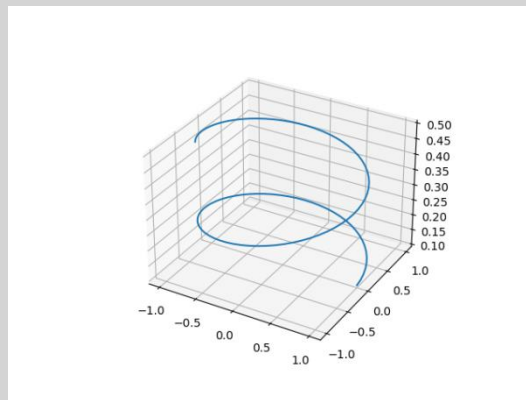




## NumPy & matplotlib Helix

```
import numpy as np
from math import sin, cos
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
n = 100 # Number of vertices
vertices = np.zeros((n, 3))
edges = np.zeros((n-1, 2))
# Generate 100 y values ranging from 0 to 10
yVals = np.linspace( 0, 10, 100 )
zVals = np.linspace( 0.1, 0.5, 100 )
# Iterate over y values and generate 3D vertex coordinates
for i, y in enumerate( yVals ):
    vertices[i] = ( cos( y ), sin( y ), zVals[i]) # Set (x,y,z) vertex coordinate
    if i < n - 1:
        # Set edge vertices => this vertex and the next
        edges[i] = ( i, i+1 )

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(vertices[:,0], vertices[:,1], vertices[:,2])
plt.show()
```





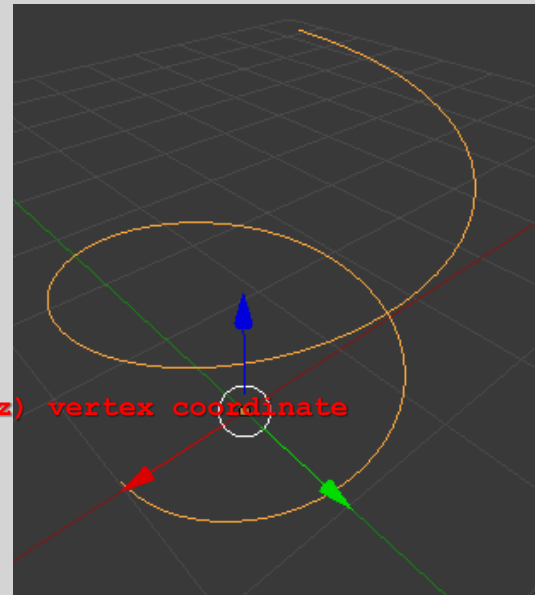
## Blender Helix

```
import bpy
from math import sin, cos

# Generate 100 y values
n = 100                                # Number of vertices
verts = []                             # n vertices
edges = []                             # n-1 edges
yVals = [y*0.18 for y in range(0, n)]
zVals = [z*0.01 for z in range(0, n)]

# Iterate over y values and generate 3D vertex coordinates
for i, y in enumerate( yVals ):
    verts.append(( cos( y ), sin( y ), zVals[i]))    # Set (x,y,z) vertex coordinate
    if i < n - 1:
        # Set edge vertices => this vertex and the next
        edges.append(( i, i+1 ))

# Generate a new mesh
mesh = bpy.data.meshes.new( 'helix' )
# Create mesh
mesh.from_pydata( verts, edges, ( ) )
# Generate an object to contain an instance of this mesh 'mesh'
ob = bpy.data.objects.new( 'helix', mesh )
# Link this virtual object to an actual scene (the active or 'context' scene)
bpy.context.collection.objects.link(ob)
ob.select_set(True)
```





## Blender/Python API

### The `bmesh` Module

The `bmesh` module provides access to blender's mesh data structures.

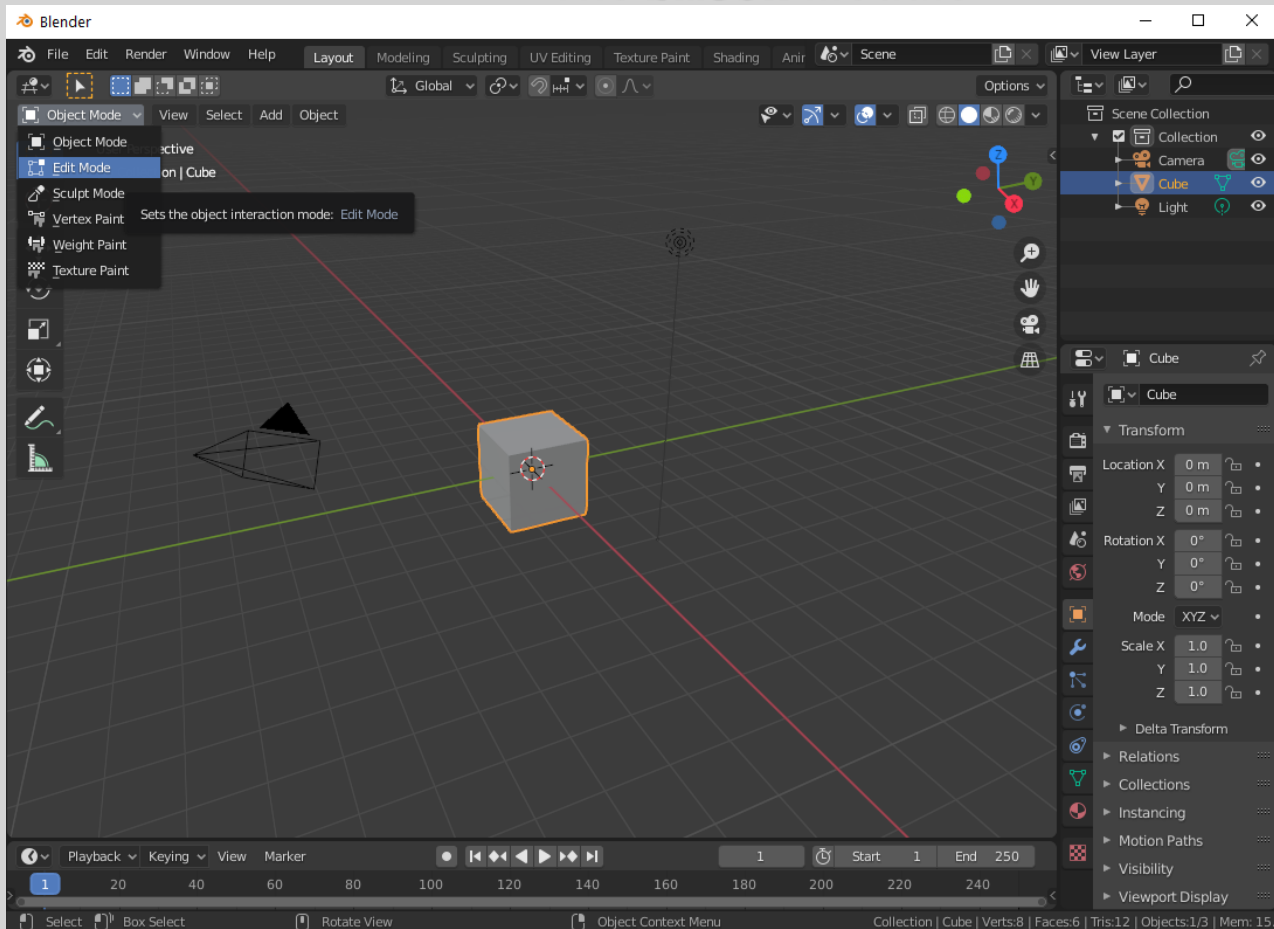
- Blender's default mode is **Object Mode**, which allows us to select and manipulate one or many objects, typically with transformations that can be appropriately applied to groups of disparate objects, such as rotation and translation.
- **Edit Mode** allows us to select one or many vertices of a single object to perform advanced and detailed transformations. The `bmesh` module deals almost exclusively in **Edit Mode** operations.

## Blender/Python API The bmesh Module

When switching into **Edit Mode**, the **activated object** at that time will be the **only** object the user can edit for that session of **Edit Mode**.

If the user want to manipulate a different object in **Edit Mode**, it must be switched back to **Object Mode** to **activate** the **desired object** first.

Only then, after switching back into **Edit Mode** with the desired object activated.







## Blender/Python API The bmesh Module

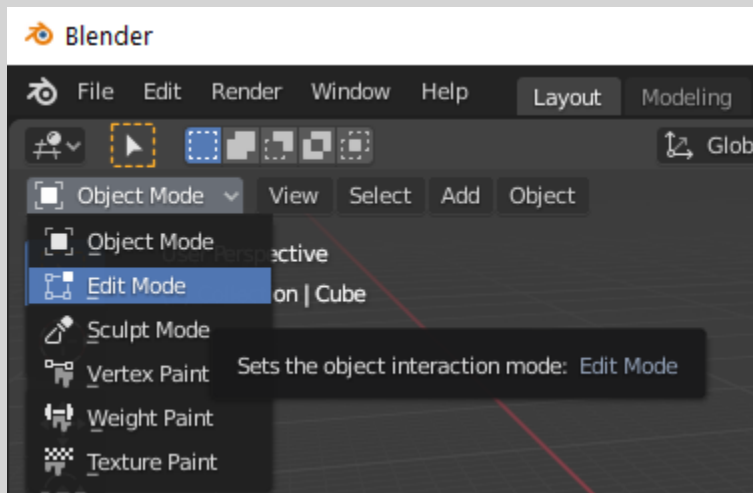
# To programmatically switch between Object Mode and Edit Mode

# Set mode to Edit Mode

```
bpy.ops.object.mode_set(mode="EDIT")
```

# Set mode to Object Mode

```
bpy.ops.object.mode_set(mode="OBJECT")
```





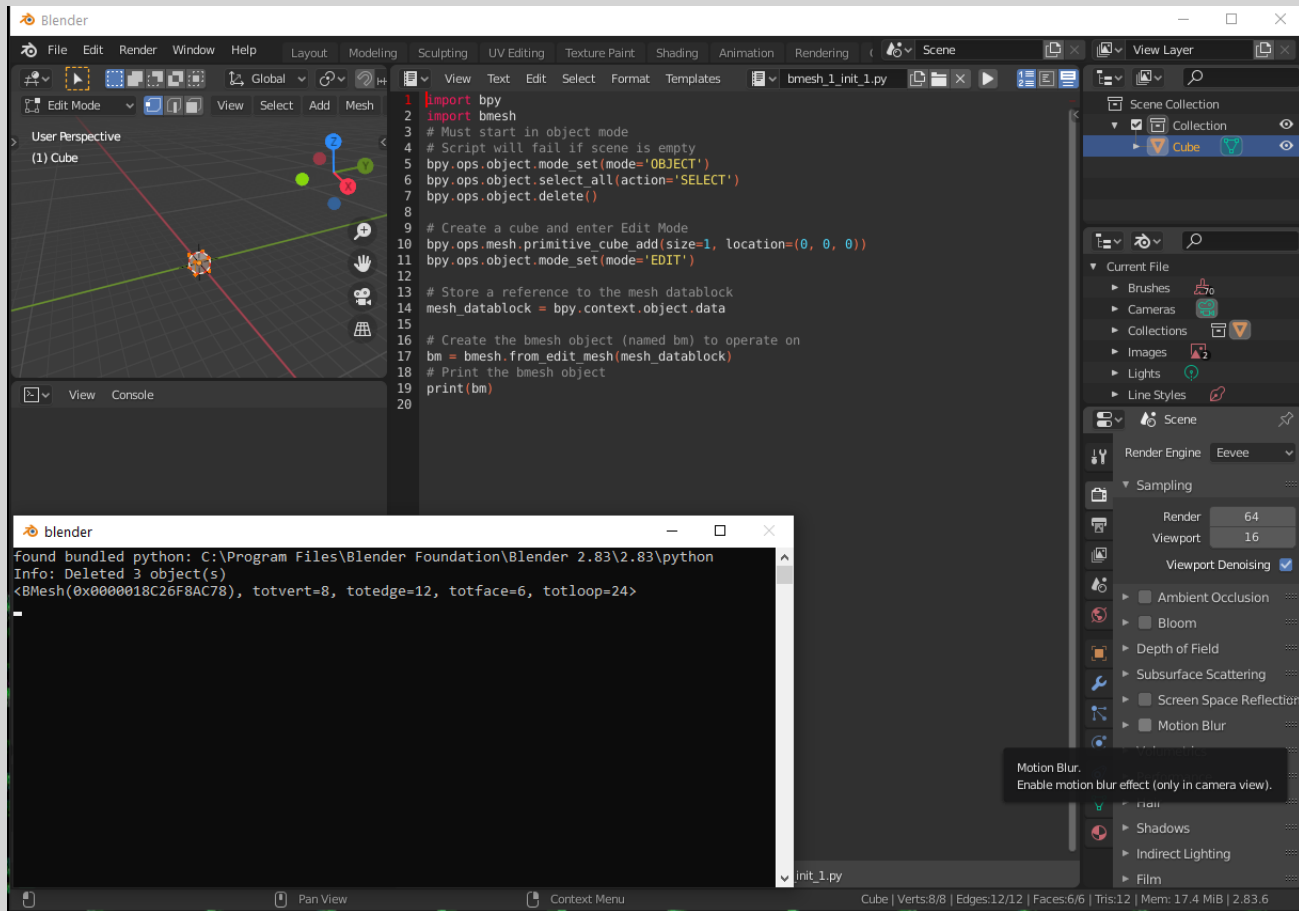
## Blender/Python API Instantiating a bmesh Object

```
import bpy
import bmesh
# Must start in object mode
# Script will fail if scene is empty
bpy.ops.object.mode_set(mode='OBJECT')
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete()
# Create a cube and enter Edit Mode
bpy.ops.mesh.primitive_cube_add(size=1, location=(0, 0, 0))
bpy.ops.object.mode_set(mode='EDIT')
# Store a reference to the mesh datablock
mesh_datablock = bpy.context.object.data

# Create the bmesh object (named bm) to operate on
bm = bmesh.from_edit_mesh(mesh_datablock)
# Print the bmesh object
print(bm)
```



## Blender/Python API Instantiating a bmesh Object



The screenshot displays the Blender 2.83 interface with the following components:

- Text Editor:** A Python script titled `bmesh_1_init_1.py` is shown. The script performs the following actions:
  - Imports `bpy` and `bmesh`.
  - Comments: `# Must start in object mode` and `# Script will fail if scene is empty`.
  - Sets the object mode: `bpy.ops.object.mode_set(mode='OBJECT')`.
  - Selects all objects: `bpy.ops.object.select_all(action='SELECT')`.
  - Deletes the selected objects: `bpy.ops.object.delete()`.
  - Comments: `# Create a cube and enter Edit Mode`.
  - Adds a cube: `bpy.ops.mesh.primitive_cube_add(size=1, location=(0, 0, 0))`.
  - Sets the object mode back to Edit: `bpy.ops.object.mode_set(mode='EDIT')`.
  - Comments: `# Store a reference to the mesh datablock`.
  - Gets the mesh datablock: `mesh_datablock = bpy.context.object.data`.
  - Comments: `# Create the bmesh object (named bm) to operate on`.
  - Creates a bmesh object: `bm = bmesh.from_edit_mesh(mesh_datablock)`.
  - Comments: `# Print the bmesh object`.
  - Prints the bmesh object: `print(bm)`.
- Console:** A terminal window titled `blender` shows the output of the script:
 

```
found bundled python: C:\Program Files\Blender Foundation\Blender 2.83\2.83\python
Info: Deleted 3 object(s)
<BMesh(0x0000018C26F8AC78), totvert=8, totedge=12, totface=6, totloop=24>
```
- Outliner:** The 'Scene Collection' is expanded, showing a 'Collection' containing a 'Cube'.
- Properties Panel:** The 'Scene' properties are visible, including the 'Render Engine' set to 'Eevee' and various rendering options like 'Ambient Occlusion', 'Bloom', 'Depth of Field', etc.



## Blender/Python API Instantiating a bmesh Object

```
<BMesh(0x000001D5CB6A1608), totvert=8, totedge=12, totface=6, totloop=24>
```

At the most basic level, BMesh stores topology in four main element structures:

Faces

Loops (stores per-face-vertex data, uvs, vcols, etc)

Edges

Verts

**Loops define the boundary loop of a face.** Each loop logically corresponds to an edge, though the loop is local to a single face so there will usually be more than one loop per edge (except at boundary edges of the surface).



## Blender/Python API

### Selecting Parts of a 3D Object

To select parts of a bmesh object, we manipulate the select Booleans of each `BMesh.verts`, `BMesh.edges`, and `BMesh.faces` object.

We use `ensure_lookup_table()` functions to remind Blender to keep certain parts of the `BMesh` object from being garbage-collected between operations. These functions take up minimal processing power, so we can call them liberally without much consequence. It is better to over-call them than to under-call them, because debugging this error:

**ReferenceError: BMesh data of type BMesh has been removed**

Can be nightmarish in large codebases with no protocol for `ensure_lookup_table()`.





## Blender/Python API

### Selecting Parts of a 3D Object

```
import bpy
import bmesh

# Must start in object mode
bpy.ops.object.mode_set(mode='OBJECT')
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete()

# Create a cube and enter Edit Mode
bpy.ops.mesh.primitive_cube_add(size=1, location=(0, 0, 0))
bpy.ops.object.mode_set(mode='EDIT')

# Set to "Face Mode" for easier visualization
bpy.ops.mesh.select_mode(type = "FACE")

# Register bmesh object and select various parts
bm = bmesh.from_edit_mesh(bpy.context.object.data)
# Deselect all verts, edges, faces
bpy.ops.mesh.select_all(action="DESELECT")
```



## Blender/Python API Selecting Parts of a 3D Object

**# Select a face**

```
bm.faces.ensure_lookup_table()  
bm.faces[0].select = True
```



**# Select an edge**

```
bm.edges.ensure_lookup_table()  
bm.edges[7].select = True
```

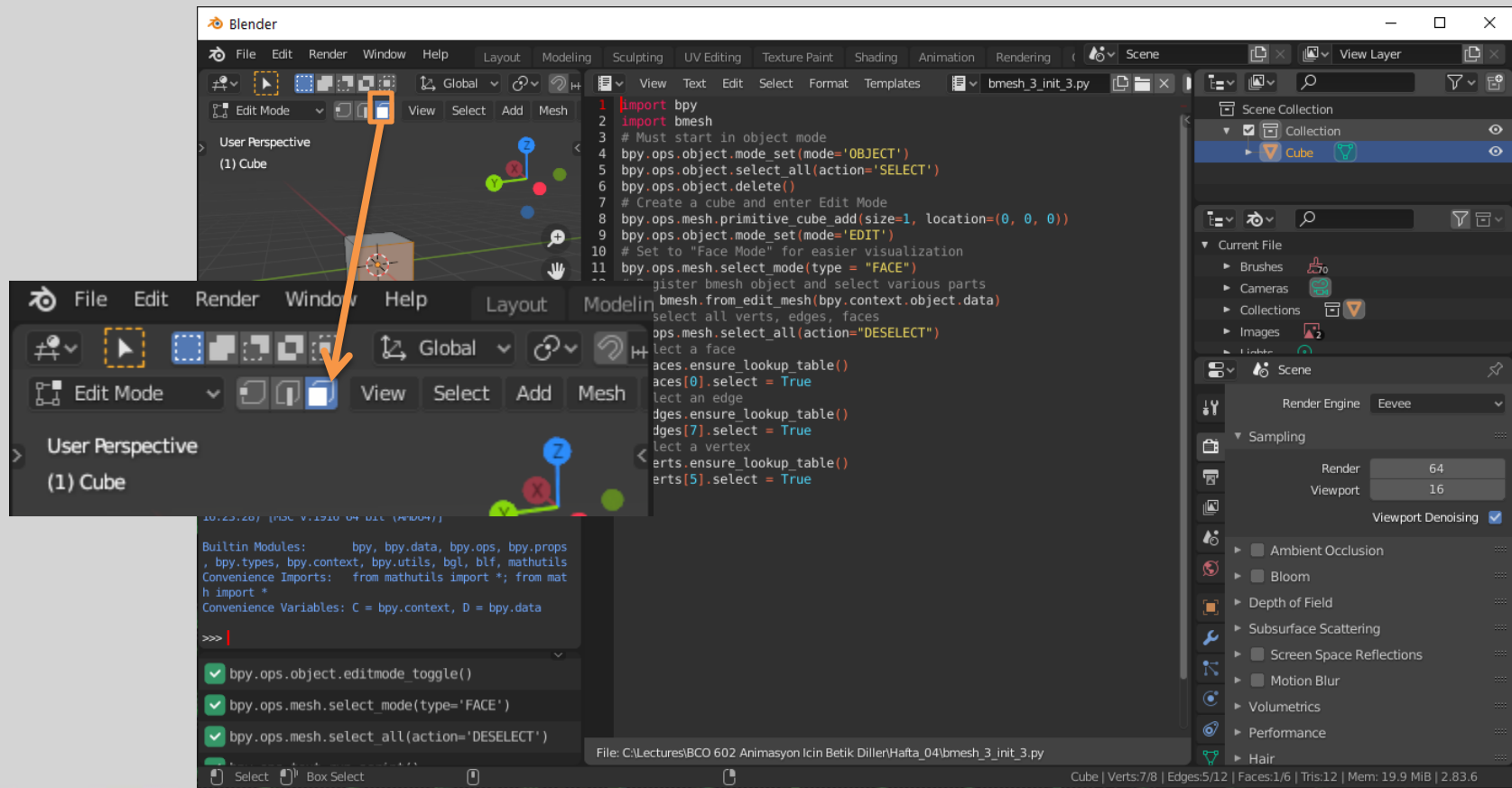


**# Select a vertex**

```
bm.verts.ensure_lookup_table()  
bm.verts[5].select = True
```



## Blender/Python API Instantiating a bmesh Object



Blender 2.80.0

File Edit Render Window Help Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Scene View Layer

View Text Edit Select Format Templates bmesh\_3\_init\_3.py

1 import bpy  
2 import bmesh  
3 # Must start in object mode  
4 bpy.ops.object.mode\_set(mode='OBJECT')  
5 bpy.ops.object.select\_all(action='SELECT')  
6 bpy.ops.object.delete()  
7 # Create a cube and enter Edit Mode  
8 bpy.ops.mesh.primitive\_cube\_add(size=1, location=(0, 0, 0))  
9 bpy.ops.object.mode\_set(mode='EDIT')  
10 # Set to "Face Mode" for easier visualization  
11 bpy.ops.mesh.select\_mode(type="FACE")  
12 # Register bmesh object and select various parts  
13 bmesh.from\_edit\_mesh(bpy.context.object.data)  
14 # select all verts, edges, faces  
15 bpy.ops.mesh.select\_all(action="DESELECT")  
16 # select a face  
17 faces.ensure\_lookup\_table()  
18 faces[0].select = True  
19 # select an edge  
20 edges.ensure\_lookup\_table()  
21 edges[7].select = True  
22 # select a vertex  
23 verts.ensure\_lookup\_table()  
24 verts[5].select = True

User Perspective (1) Cube

File Edit Render Window Help Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Scene View Layer

Edit Mode View Select Add Mesh

User Perspective (1) Cube

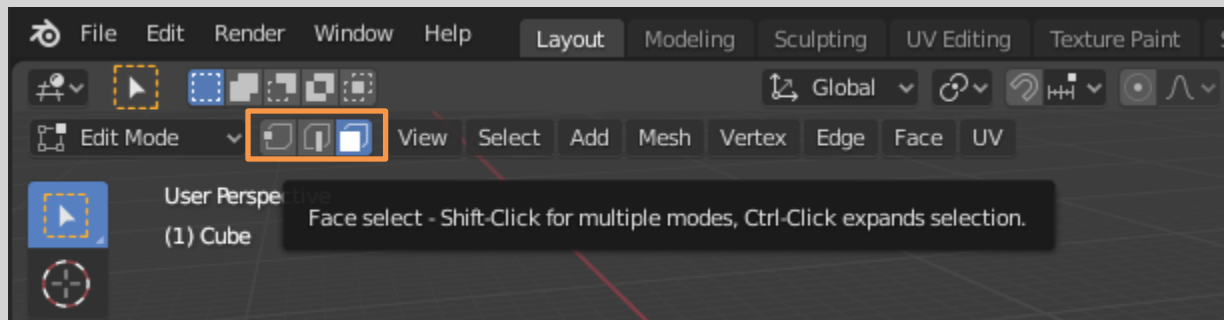
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, blf, mathutils  
Convenience Imports: from mathutils import \*; from math import \*  
Convenience Variables: C = bpy.context, D = bpy.data

>>> |

✓ bpy.ops.object.editmode.toggle()  
✓ bpy.ops.mesh.select\_mode(type='FACE')  
✓ bpy.ops.mesh.select\_all(action='DESELECT')

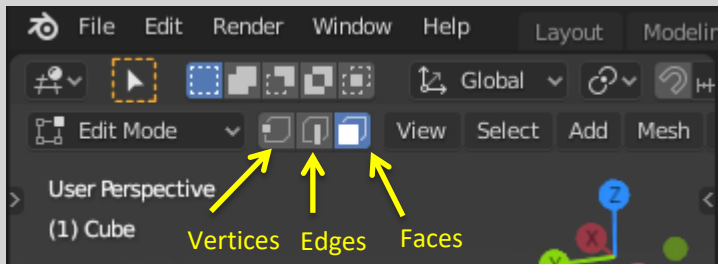
File: C:\Lectures\BCO 602 Animasyon İçin Betik Diller\Hafta\_04\bmesh\_3\_init\_3.py

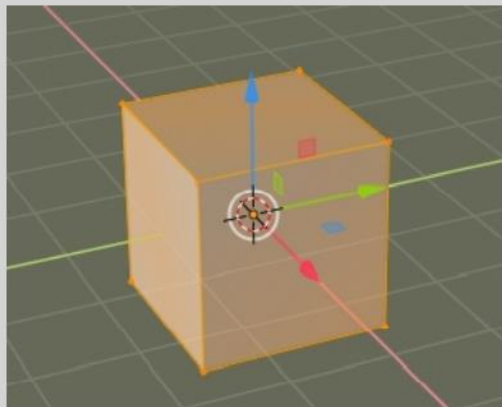
Cube | Verts:7/8 | Edges:5/12 | Faces:1/6 | Tris:12 | Mem: 19.9 MiB | 2.83.6



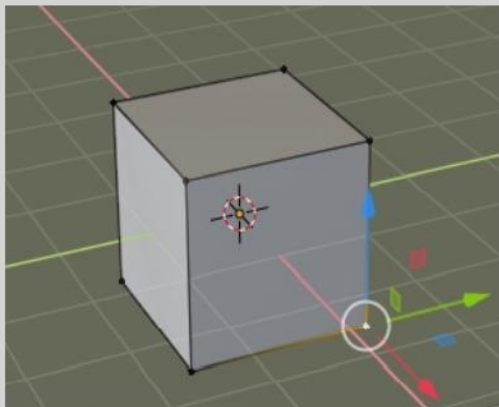
We run `bpy.ops.mesh.select_mode(type = "FACE")`.

The buttons in the figure correspond to the `VERT`, `EDGE`, and `FACE` arguments in `bpy.ops.mesh.select_mode()`.

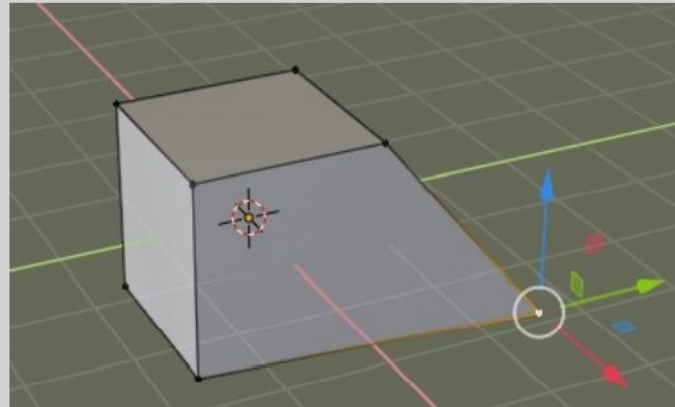




Default Cube – All Vertices  
Selected



Single Vertex Selected  
**EDIT MODE**



Single Vertex Moved (Translated)



## Blender/Python API Basic Transformations in Edit Mode

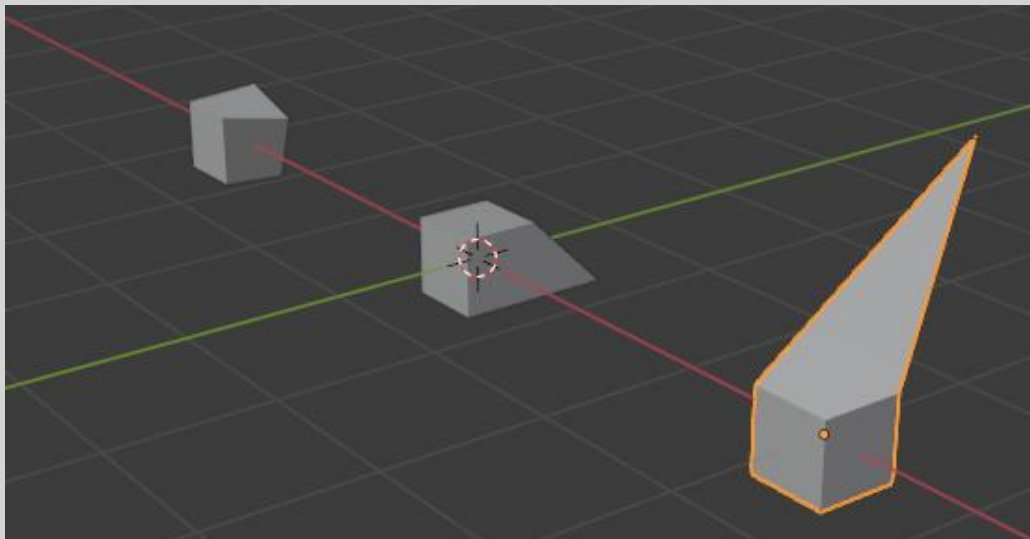
```
import bpy  
import bmesh
```

```
# Must start in object mode
```

```
bpy.ops.object.mode_set(mode = 'OBJECT')
```

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

```
bpy.ops.object.delete()
```







## Blender/Python API Basic Transformations in Edit Mode

```
# Create a cube and rotate a face around the y-axis
```

```
bpy.ops.mesh.primitive_cube_add(size = 0.5, location = (-3, 0, 0))
```

```
bpy.ops.object.mode_set(mode = 'EDIT')
```

```
bpy.ops.mesh.select_all(action = "DESELECT")
```

```
# Set to face mode for transformations
```

```
bpy.ops.mesh.select_mode(type = "FACE")
```

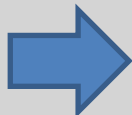
```
bm = bmesh.from_edit_mesh(bpy.context.object.data)
```

```
bm.faces.ensure_lookup_table()
```

```
bm.faces[1].select = True
```

```
bpy.ops.transform.rotate(value = 0.3, orient_axis = 'Y')
```

```
bpy.ops.object.mode_set(mode = 'OBJECT')
```





## Blender/Python API Basic Transformations in Edit Mode

# Create a cube and pull an edge along the y-axis

```
bpy.ops.mesh.primitive_cube_add(size = 0.5, location = (0, 0, 0))  
bpy.ops.object.mode_set(mode = 'EDIT')  
bpy.ops.mesh.select_all(action = 'DESELECT')
```



```
bm = bmesh.from_edit_mesh(bpy.context.object.data)  
bm.edges.ensure_lookup_table()  
bm.edges[4].select = True  
bpy.ops.transform.translate(value = (0, 0.5, 0))  
  
bpy.ops.object.mode_set(mode = 'OBJECT')
```



## Blender/Python API

### Basic Transformations in Edit Mode

```
# Create a cube and pull a vertex 1 unit  
# along the y and z axes  
# Create a cube and pull an edge along the y-axis
```

```
bpy.ops.mesh.primitive_cube_add(size=0.5, location=(3, 0, 0))  
bpy.ops.object.mode_set(mode = 'EDIT')  
bpy.ops.mesh.select_all(action = "DESELECT")
```



```
bm = bmesh.from_edit_mesh(bpy.context.object.data)  
bm.verts.ensure_lookup_table()  
bm.verts[3].select = True  
bpy.ops.transform.translate(value = (0, 1, 1))  
  
bpy.ops.object.mode_set(mode = 'OBJECT')
```



## Blender/Python API Extrude, Subdivide, and Randomize Operators

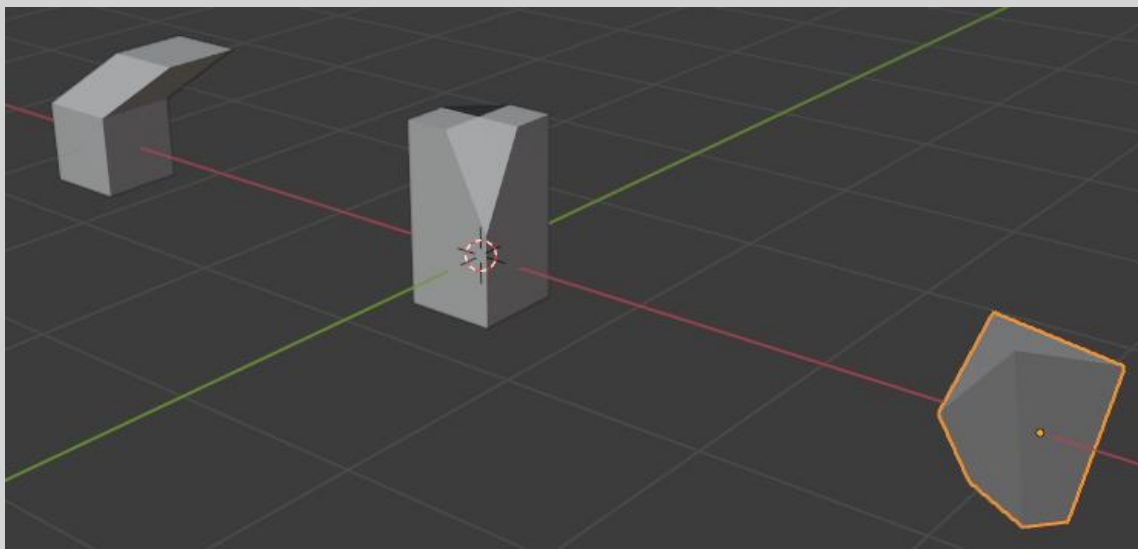
```
import bpy  
import bmesh
```

```
# Must start in object mode
```

```
bpy.ops.object.mode_set(mode = 'OBJECT')
```

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

```
bpy.ops.object.delete()
```





### Extrude, Subdivide, and Randomize Operators

```
# Create a cube and extrude the top face away from it
```

```
bpy.ops.mesh.primitive_cube_add(size=0.5, location=(-3, 0, 0))
```

```
bpy.ops.object.mode_set(mode = 'EDIT')
```

```
bpy.ops.mesh.select_all(action = "DESELECT")
```

```
# Set to face mode for transformations
```

```
bpy.ops.mesh.select_mode(type = "FACE")
```

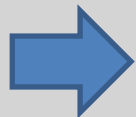
```
bm = bmesh.from_edit_mesh(bpy.context.object.data)
```

```
bm.faces.ensure_lookup_table()
```

```
bm.faces[5].select = True
```

```
bpy.ops.mesh.extrude_region_move(TRANSFORM_OT_translate =  
    {"value": (0.3, 0.3, 0.3),  
     "constraint_axis": (True, True, True),  
     "constraint_orientation": 'NORMAL'})
```

```
bpy.ops.object.mode_set(mode = 'OBJECT')
```





### Extrude, Subdivide, and Randomize Operators

**# Create a cube and subdivide the top face**

```
bpy.ops.mesh.primitive_cube_add(size = 0.5, location = (0, 0, 0))
```

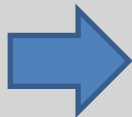
```
bpy.ops.object.mode_set(mode = 'EDIT')
```

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

```
bm = bmesh.from_edit_mesh(bpy.context.object.data)
```

```
bm.faces.ensure_lookup_table()
```

```
bm.faces[5].select = True
```



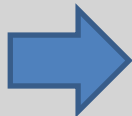
```
bpy.ops.mesh.subdivide(number_cuts = 1)
```

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

```
bm.faces.ensure_lookup_table()
```

```
bm.faces[5].select = True
```

```
bm.faces[7].select = True
```



```
bpy.ops.transform.translate(value = (0, 0, 0.5))
```

```
bpy.ops.object.mode_set(mode = 'OBJECT')
```





## Blender/Python API

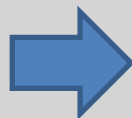
### Extrude, Subdivide, and Randomize Operators

# Create a cube and add a random offset to each vertex

```
bpy.ops.mesh.primitive_cube_add(size=0.5, location = (3, 0, 0))
```

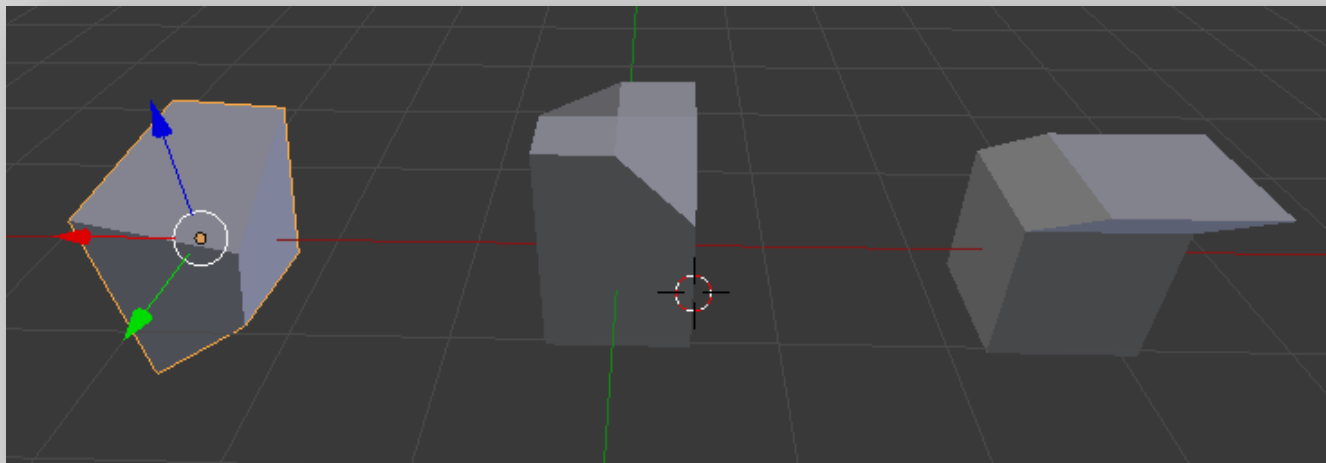
```
bpy.ops.object.mode_set(mode = 'EDIT')
```

```
bpy.ops.mesh.select_all(action = "SELECT")
```



```
bpy.ops.transform.vertex_random(offset = 0.5)
```

```
bpy.ops.object.mode_set(mode = 'OBJECT')
```





You may have noticed that the indices of vertices, edges, and faces in 3D objects are arranged in no particular order. In the example scripts thus far, we had manually located the indices in advance rather than discover them programmatically. Using trial-and-error tests to discover the index number of a part of an object is an acceptable practice in general, but suffers from a number of disadvantages.

- Default indices of objects vary wildly across different versions of Blender.
- Behavior of indexing after certain transformations is badly organized.
- Add-ons using hardcoded indices are very limited in user-interaction possibilities.



## Blender/Python API

### Note on Indexing and Cross-Compatibility

The workaround to this issue is selection by ***characteristic***. To select a **vertex** by a characteristic, we loop through each **vertex** in the object and run `bm.verts[i].select = True` on vertices that meet a criteria. The same holds for **edges** and **faces**. On paper, this method looks very computationally expensive and algorithmically complex, but you will find it is surprisingly fast and modular. Plugins that use pure selection by characteristic can often run successfully on many versions of Blender simultaneously.



## Blender/Python API

### Global and Local Coordinates

Blender stores many sets of coordinate data for each part of each object. In most cases, we will only be concerned with two sets of coordinates: ***global coordinates*** **G** and ***local coordinates*** **L**. When we perform transformations on objects, Blender stores these transformations as part of a **transformation matrix**, **T**. Blender will, at some point, apply the transformation matrix to the local coordinates. After Blender applies the transformation matrix, the local coordinates will be equal to the global coordinates, and the transformation matrix will be the **identity matrix** **I**.

Within the 3D Viewport, we view global coordinates  $\mathbf{G} = \mathbf{T} * \mathbf{L}$  always.



## Blender/Python API

### Global and Local Coordinates

We can control when Blender applies transformations with `bpy.ops.object.transform_apply()`. This will not change the appearance of the objects, rather it will set **L** equal to **G** and set **T** equal to the **I**. We can use this to our advantage to easily select specific parts of objects.

If we delay execution of `bpy.ops.object.transform_apply()` by **not** running it and not exiting **Edit Mode**, we can maintain two data sets **G** and **L**. In practice, **G** is very useful for positioning objects relative to others, and **L** is very easy to loop through to fetch indices.



# Fetching Global and Local Coordinates

```
def coords(objName, space='GLOBAL'):  
    # Store reference to the bpy.data.objects datablock  
    obj = bpy.data.objects[objName]  
    # Store reference to bpy.data.objects[].meshes datablock  
    if obj.mode == 'EDIT':  
        v = bmesh.from_edit_mesh(obj.data).verts  
    elif obj.mode == 'OBJECT':  
        v = obj.data.vertices  
    if space == 'GLOBAL':  
        # Return T * L as list of tuples  
        return [(obj.matrix_world @ v.co).to_tuple() for v in v]  
    elif space == 'LOCAL':  
        # Return L as list of tuples  
        return [v.co.to_tuple() for v in v]
```





## Blender/Python API The bmesh Module

```
# Place in bco602tk.py
# Function for entering Edit Mode with no vertices selected,
# or entering Object Mode with no additional processes

def mode(mode_name):
    bpy.ops.object.mode_set(mode=mode_name)
    if mode_name == "EDIT":
        bpy.ops.mesh.select_all(action = "DESELECT")

def selection_mode(type):
    bpy.ops.mesh.select_mode(type = type)
```



## Blender/Python API Global and Local Coordinates

```
class sel:
    # Add this to the ut.sel class, for use in object mode
    def transform_apply():
        bpy.ops.object.transform_apply(
            location=True, rotation=True, scale=True)
```

---

```
import bpy
tk = bpy.data.texts["bco602tk.py"].as_module()

# Will fail if scene is empty
tk.mode("OBJECT")
tk.delete_all()
tk.create.cube('Cube-1')
```



### Behavior of Global and Local Coordinates and Transform

```
# Check global and local coordinates
print('\nBefore transform:')
print('Global:', tk.coords('Cube-1', 'GLOBAL')[0:2])
print('Local: ', tk.coords('Cube-1', 'LOCAL')[0:2])
# Translate it along x = y = z
# See the cube move in the 3D viewport
tk.sel.translate((3, 3, 3))
# Check global and local coordinates
print('\nAfter transform, unapplied:')
print('Global: ', tk.coords('Cube-1', 'GLOBAL')[0:2])
print('Local: ', tk.coords('Cube-1', 'LOCAL')[0:2])
# Apply transformation
# Nothing changes in 3D viewport
tk.sel.transform_apply()
# Check global and local coordinates
print('\nAfter transform, applied:')
print('Global: ', tk.coords('Cube-1', 'GLOBAL')[0:2])
print('Local: ', tk.coords('Cube-1', 'LOCAL')[0:2])
```



```
blender
Before transform:
Global: [(-0.25, -0.25, -0.25), (-0.25, -0.25, 0.25)]
Local:  [(-0.25, -0.25, -0.25), (-0.25, -0.25, 0.25)]

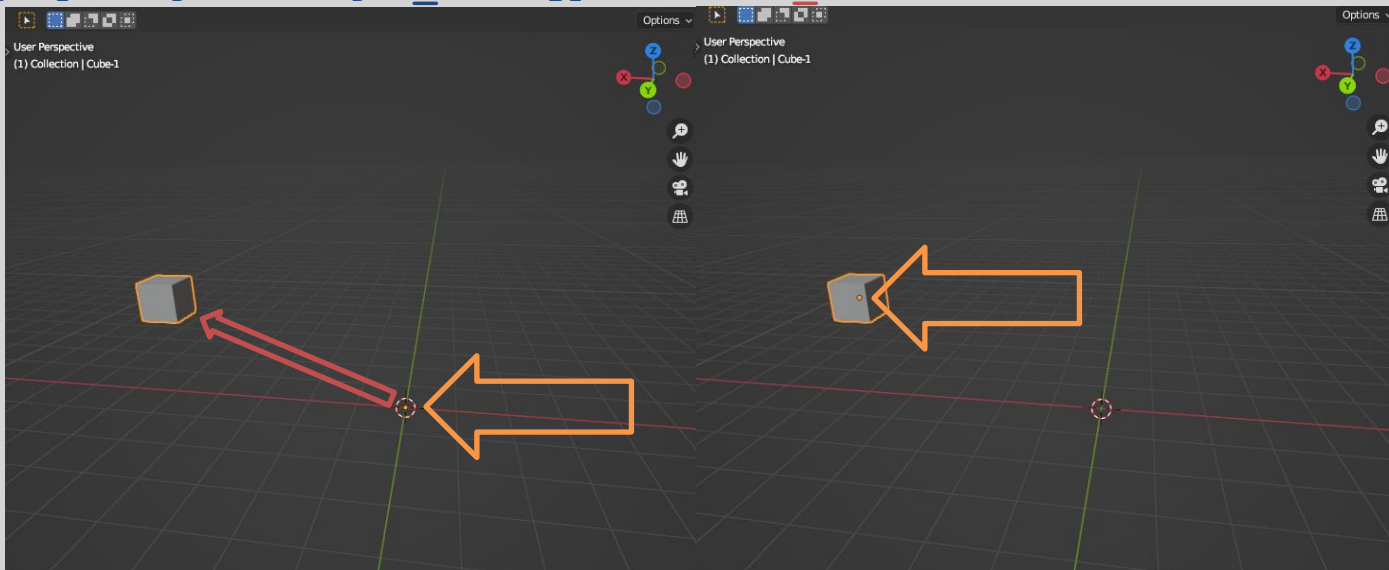
After transform, unapplied:
Global: [(2.75, 2.75, 2.75), (2.75, 2.75, 3.25)]
Local:  [(-0.25, -0.25, -0.25), (-0.25, -0.25, 0.25)]

After transform, applied:
Global: [(2.75, 2.75, 2.75), (2.75, 2.75, 3.25)]
Local:  [(2.75, 2.75, 2.75), (2.75, 2.75, 3.25)]
```

$$\leftarrow G = T \times L$$
$$G = L$$



```
bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```



```
C:\Program Files\Blender Foundation\Blender 3.6\ble...  
After transform, applied:  
Global: [(2.75, 2.75, 2.75), (2.75, 2.75, 3.25)]  
Local: [(2.75, 2.75, 2.75), (2.75, 2.75, 3.25)]  
  
After Origin set to geometry:  
Global: [(2.75, 2.75, 2.75), (2.75, 2.75, 3.25)]  
Local: [(-0.25, -0.25, -0.25), (-0.25, -0.25, 0.25)]
```



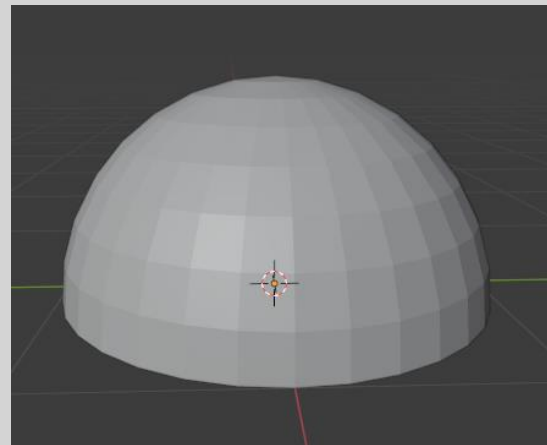
## Blender/Python API How to Make a Semi Sphere

```
import bpy
import bmesh

bpy.ops.mesh.primitive_uv_sphere_add(radius=1, location=(0, 0, 0))
bpy.ops.object.mode_set(mode = 'EDIT')
bpy.ops.mesh.select_all(action = "DESELECT")

bm = bmesh.from_edit_mesh(bpy.context.object.data)
bm.verts.ensure_lookup_table()

for i, v in enumerate(bm.verts):
    if v.co[2] < 0:
        bm.verts[i].select = True
bpy.ops.mesh.delete(type='VERT')
bpy.ops.object.mode_set(mode='OBJECT')
```

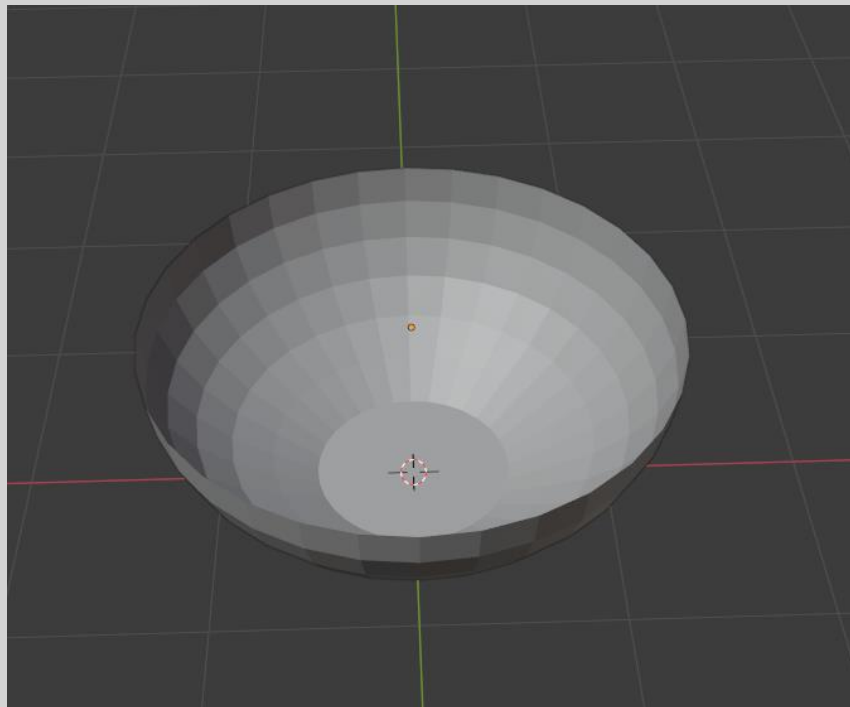






## Blender/Python API

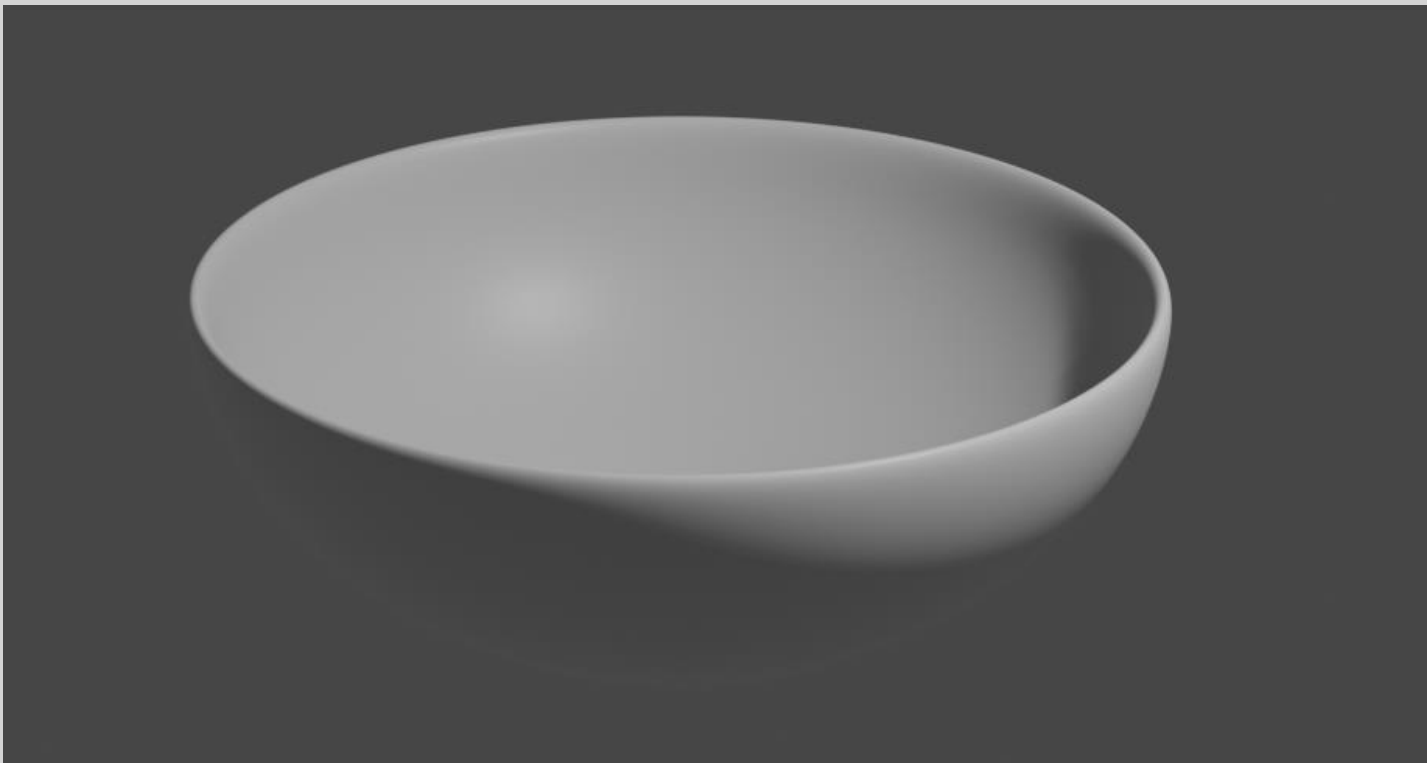
### Classwork: Make a Bowl





## Blender/Python API

If you know Blender well; apply modifiers which we will discuss later





## Blender/Python API

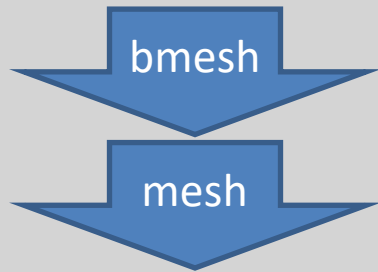
### The `bmesh` Module

Note that unlike `bpy`, a `BMesh` does not necessarily correspond to data in the currently open blend-file, a `BMesh` can be created, edited and freed without the user ever seeing or having access to it.

Unlike Edit-Mode, the `BMesh` module can use multiple `BMesh` instances at once.



## Blender/Python API The bmesh Module



```
# This example assumes we have a mesh object selected
import bpy
import bmesh

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

# 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
```



## Blender/Python API The bmesh Module

**bmesh.new**(*use\_operators=True*)

**Parameters:** *use\_operators* (*bool*) – Support calling operators in `bmesh.ops` (uses some extra memory per vert/edge/face).

**Returns:** Return a new, empty BMesh.

**Return type:** `bmesh.types.BMesh`

**from\_mesh**(*mesh*, *face\_normals=True*, *vertex\_normals=True*, *use\_shape\_key=False*, *shape\_key\_index=0*)

Initialize this bmesh from existing mesh datablock.

- Parameters:**
- *mesh* (`Mesh`) – The mesh data to load.
  - *use\_shape\_key* (*boolean*) – Use the locations from a shape key.
  - *shape\_key\_index* (*int*) – The shape key index to use.

### ! Note

Multiple calls can be used to join multiple meshes.

**free()**

Explicitly free the BMesh data from memory, causing exceptions on further access.

### ! Note

The BMesh is freed automatically, typically when the script finishes executing. However in some cases its hard to predict when this will be and its useful to explicitly free the data.



## Blender/Python API The bmesh Module

```
import bpy
import bmesh

# Make a new BMesh
bm = bmesh.new()

# Add a circle, should return all geometry created, not just verts.
bmesh.ops.create_circle(
    bm,
    cap_ends=False,
    radius=0.2,
    segments=8)

# Finish up, write the bmesh into a new mesh
me = bpy.data.meshes.new("Mesh")
bm.to_mesh(me)
bm.free()

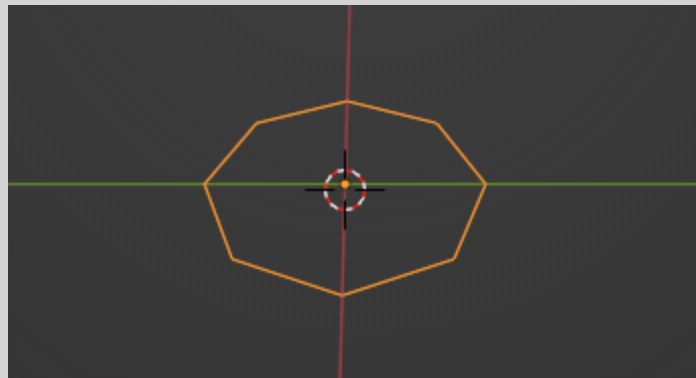
# Add the mesh to the scene
obj = bpy.data.objects.new("myCircle", me)
bpy.context.collection.objects.link(obj)

# Select and make active
bpy.context.view_layer.objects.active = obj
obj.select_set(True)
```

bmesh

mesh

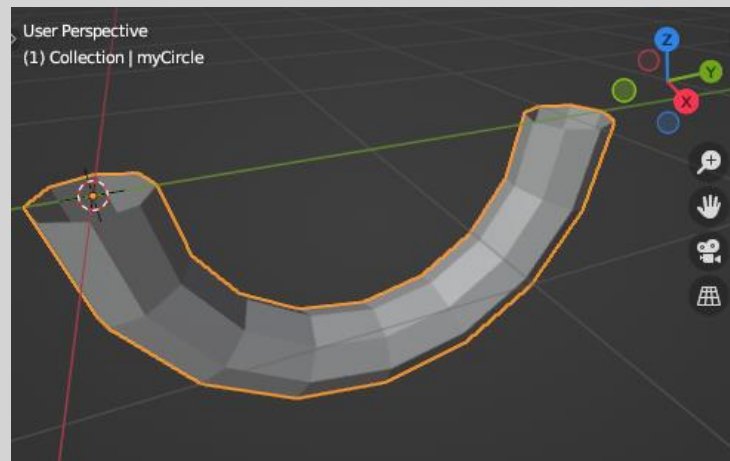
obj





```
import bpy
import bmesh

# Make a new BMesh
bm = bmesh.new()
# Add a circle, should return all geometry created, not just verts.
..... *
# Finish up, write the bmesh into a new mesh
..... *
# Spin and deal with geometry on side 'a'
edges_start_a = bm.edges[:]
geom_start_a = bm.verts[:] + edges_start_a
ret = bmesh.ops.spin(
    bm,
    geom=geom_start_a,
    angle=math.radians(180.0),
    steps=8,
    axis=(1.0, 0.0, 0.0),
    cent=(0.0, 1.0, 0.0))
edges_end_a = [ele for ele in ret["geom_last"]
                if isinstance(ele, bmesh.types.BMEdge)]
del ret
# Add the mesh to the scene
```







## Blender/Python API

### `bmesh.ops.spin`

```
bmesh.ops.spin(bm, geom, cent, axis, dvec, angle, space, steps, use_merge, use_normal_flip, use_duplicate)
```

Spin.

Extrude or duplicate geometry a number of times, rotating and possibly translating after each step

Parameters:

- `bm` (`bmesh.types.BMesh`) – The bmesh to operate on.
- `geom` (list of (`bmesh.types.BMVert`, `bmesh.types.BMEdge`, `bmesh.types.BMFace`)) – input geometry
- `cent` (`mathutils.Vector` or any sequence of 3 floats) – rotation center
- `axis` (`mathutils.Vector` or any sequence of 3 floats) – rotation axis
- `dvec` (`mathutils.Vector` or any sequence of 3 floats) – translation delta per step
- `angle` (`float`) – total rotation angle (radians)
- `space` (`mathutils.Matrix`) – matrix to define the space (typically object matrix)
- `steps` (`int`) – number of steps
- `use_merge` (`bool`) – Merge first/last when the angle is a full revolution.
- `use_normal_flip` (`bool`) – Create faces with reversed direction.
- `use_duplicate` (`bool`) – duplicate or extrude?

Returns:

- `geom_last`: result of last step
- type list of (`bmesh.types.BMVert`, `bmesh.types.BMEdge`, `bmesh.types.BMFace`)

Return type: dict with string keys

```
ret = bmesh.ops.spin(  
    bm,  
    geom=geom_start_a,  
    angle=math.radians(180.0),  
    steps=8,  
    axis=(1.0, 0.0, 0.0),  
    cent=(0.0, 1.0, 0.0))
```



## Blender/Python API Materials and Textures

Materials and Textures add color and define the surfaces of Objects. The application of Materials and Textures in Blender utilizes a graphical display called a **Node System** which represents the computer code generating the effect of Material and Texture.

Since the subject is extensive this lecture will be an introduction only.



## Blender/Python API

### Materials and Textures

In Computer Graphics, how the surface of an Object displays is determined by three factors;

Material, Texture and Lighting.

**Material:** The Base Color of the surface.

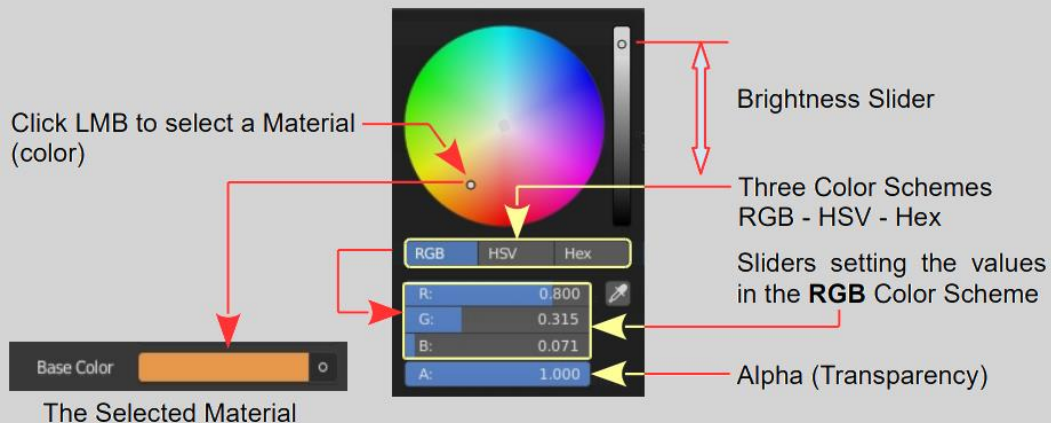
**Texture:** The physical characteristics of the surface.

**Lighting:** The background illumination or light emitting from Lights (Lamps).

## Materials

In computer graphics, a Material is the color of an Object which is how the visible spectrum of light reflects from the Object's surface. A Material also defines whether the surface appears dull (matt) or shiny (metallic).

In practice, Material (color) is selected in a Color Picker Circle.





### The Application of Materials and Material Slots

To simplify the understanding of the application of materials in Blender, it may be easier to think of materials as colors. The color of an object in a Blender scene is set by values entered using a color picker, typing RGB values, or moving sliders.

A Blender scene may have multiple materials, and these materials are stored in a cache for use by any of the objects that are introduced to the scene or any object in any scene in the Blender file. A Blender file may contain multiple scenes.

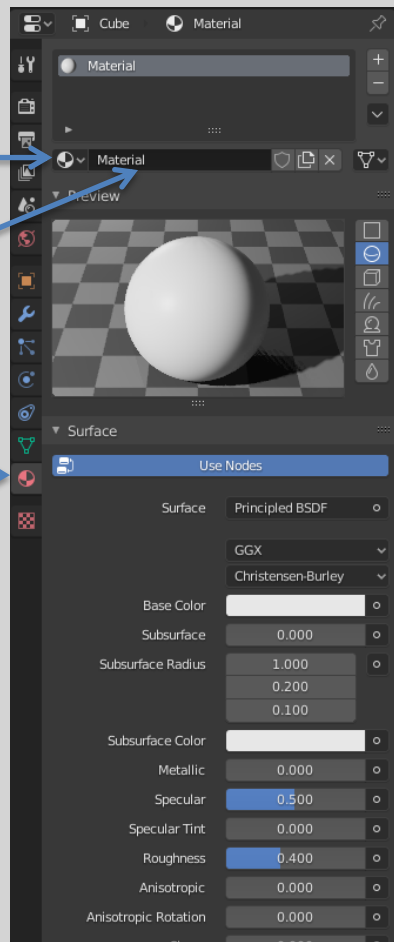
# Blender/Python API

## Materials

Browse material

Unique data block ID name

“Material” Button



To demonstrate the application of materials in Blender, set the screen to include the 3D view and the properties window with the “Material” tab active.

- a preview pane shows a dull gray color on a sphere,
- there is a material named “Material” in the unique data block ID name, and
- there is one material slot active showing the material named “Material.”



## Blender/Python API Materials

```
import bpy

def makeMaterial(name, diffuse = (1,1,1,1) , metallic = 0.0, specular = 0.8, roughness = 0.2):
    """
    This function defines a new material with a given name.

    If the diffuse value is not provided, it defaults to diffuse = (1, 1, 1, 1) (R, G, B, alpha)
    If the specular value is not provided, it defaults to 0.8
    If the metallic value is not provided, it defaults to 0.0
    If the roughness value is not provided, it defaults to 0.2
    """
    mat = bpy.data.materials.new(name)

    mat.diffuse_color = diffuse
    mat.specular_intensity = specular
    mat.metallic = metallic
    mat.roughness = roughness

    return mat

def setMaterial(ob, mat):
    me = ob.data
    me.materials.append(mat)
```



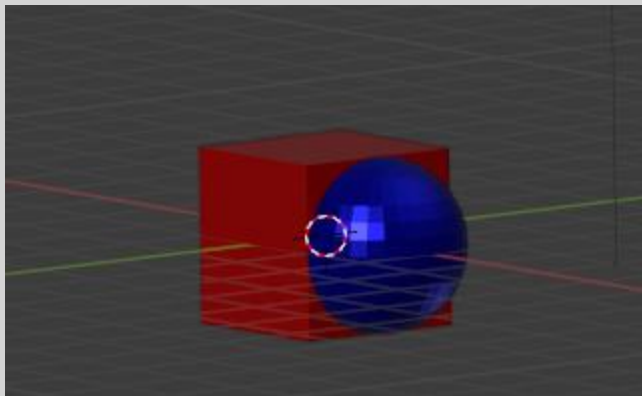
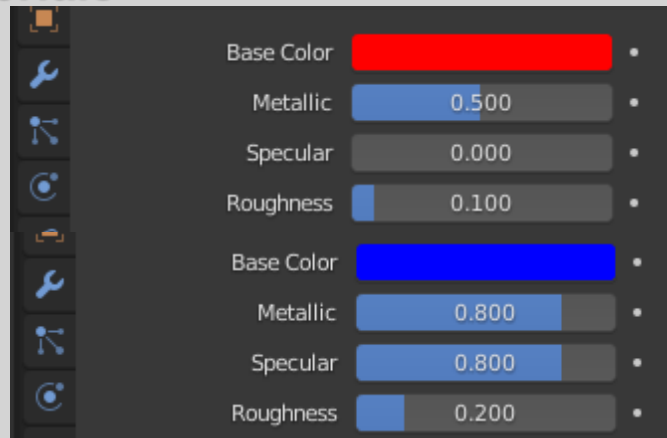


## Blender/Python API Materials

```
# Create two materials
red = makeMaterial('RedSemi', (1, 0, 0, 1), 0.5, 0, 0.1)
blue = makeMaterial('BlueSemi', (0, 0, 1, 1), 0.8, 0.8, 0.2)

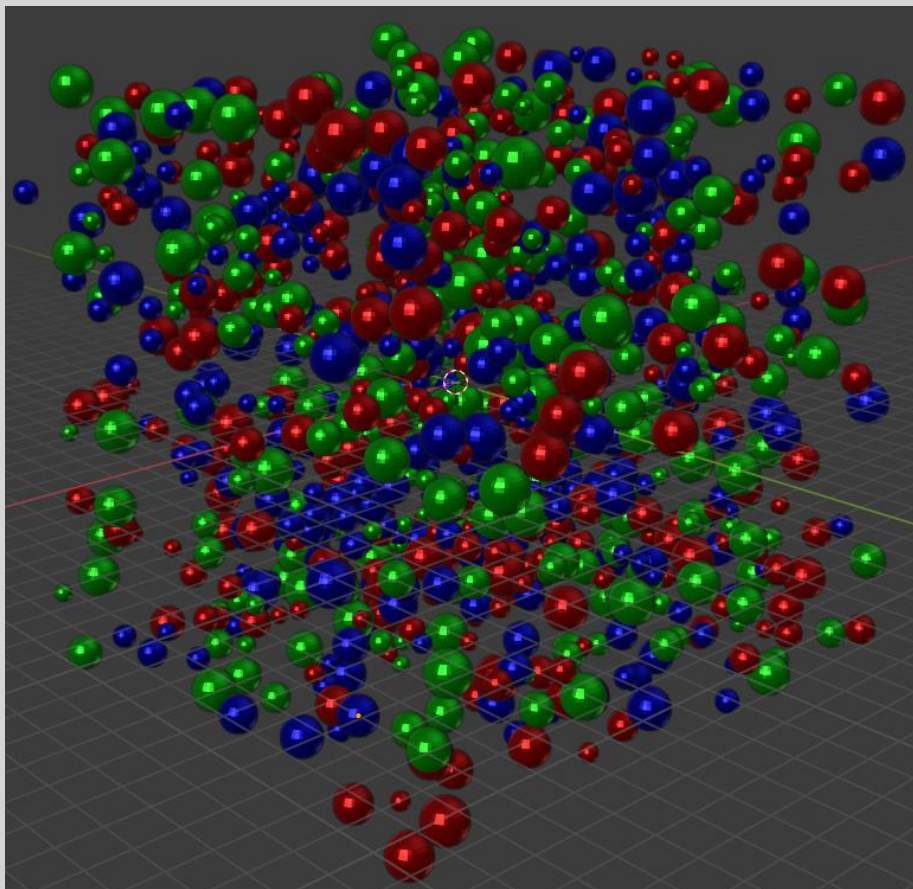
# Create red cube
bpy.ops.mesh.primitive_cube_add(location = (0,0,0))
setMaterial(bpy.context.object, red)

# and blue sphere
bpy.ops.mesh.primitive_uv_sphere_add(location = origin)
bpy.ops.transform.translate(value = (1,0,0))
setMaterial(bpy.context.object, blue)
```





## Blender/Python API Materials



Homework  
Colorize the random sphere  
work you have done before.



## Blender/Python API Materials

