01 - Python Programming in Blender, An Introduction

Pushpak Karnick, pushpak.karnick@digipen.edu

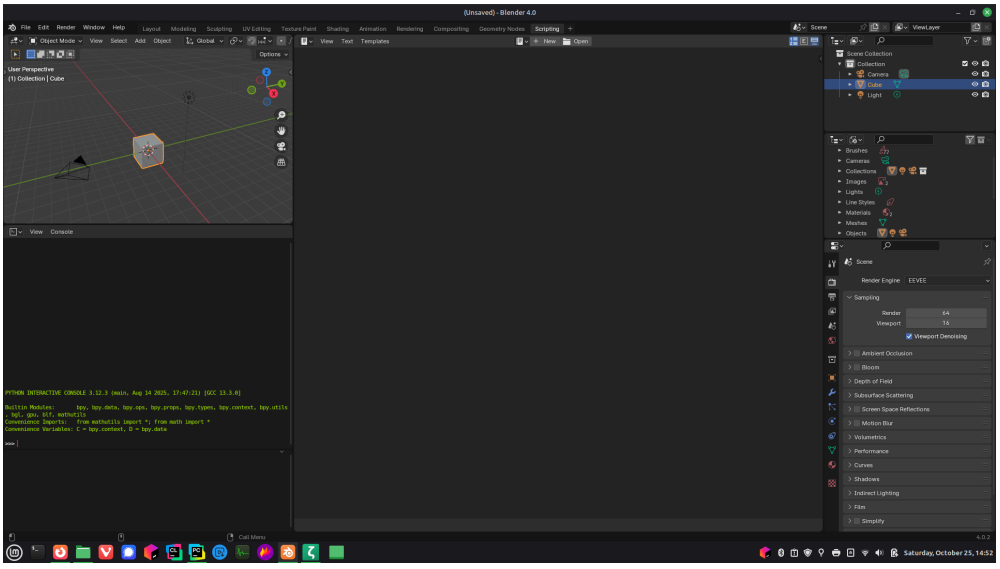
Python programming in Blender uses the `bpy` module to automate tasks, extend functionality, and create custom tools. Blender has a built-in Python interpreter, so you can write and execute scripts directly inside the application.

Getting started with scripting

Use the Scripting workspace

Blender's user interface includes a dedicated scripting workspace that provides - a text editor for writing code, - an interactive Python console for testing, and - an Info editor that shows the Python code for actions you perform.

1. Open Blender and click the **Scripting** tab at the top of the interface.
2. In the text editor, click **New** to create a new script.
3. Write your Python code in this new text block.



1-Blender-Scripting-Interface.png

Write your first script

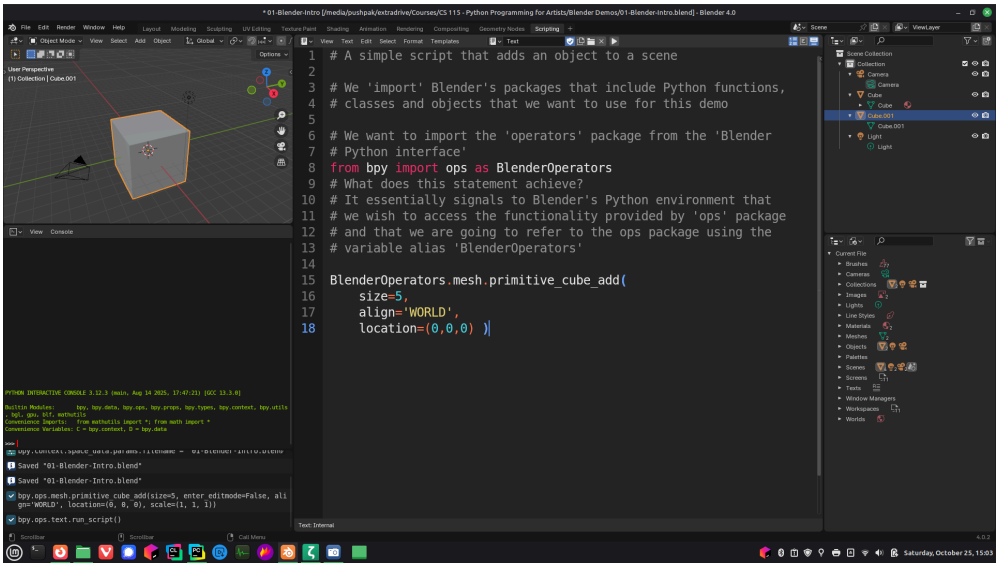
To create a simple script that adds a cube, we use the `bpy.ops` module, which contains functions for executing Blender's operators (tools).

```
# A simple script that adds an object to a scene

# We 'import' Blender's packages that include Python functions,
# classes and objects that we want to use for this demo

# We want to import the 'operators' package from the 'Blender
# Python interface'
```

```
from bpy import ops as BlenderOperators
# What does this statement achieve?
# It essentially signals to Blender's Python environment that
# we wish to access the functionality provided by 'ops' package
# and that we are going to refer to the ops package using the
# variable alias 'BlenderOperators'

BlenderOperators.mesh.primitive_cube_add(
    size=5,
    align='WORLD',
    location=(0,0,0) )
```

Run the script

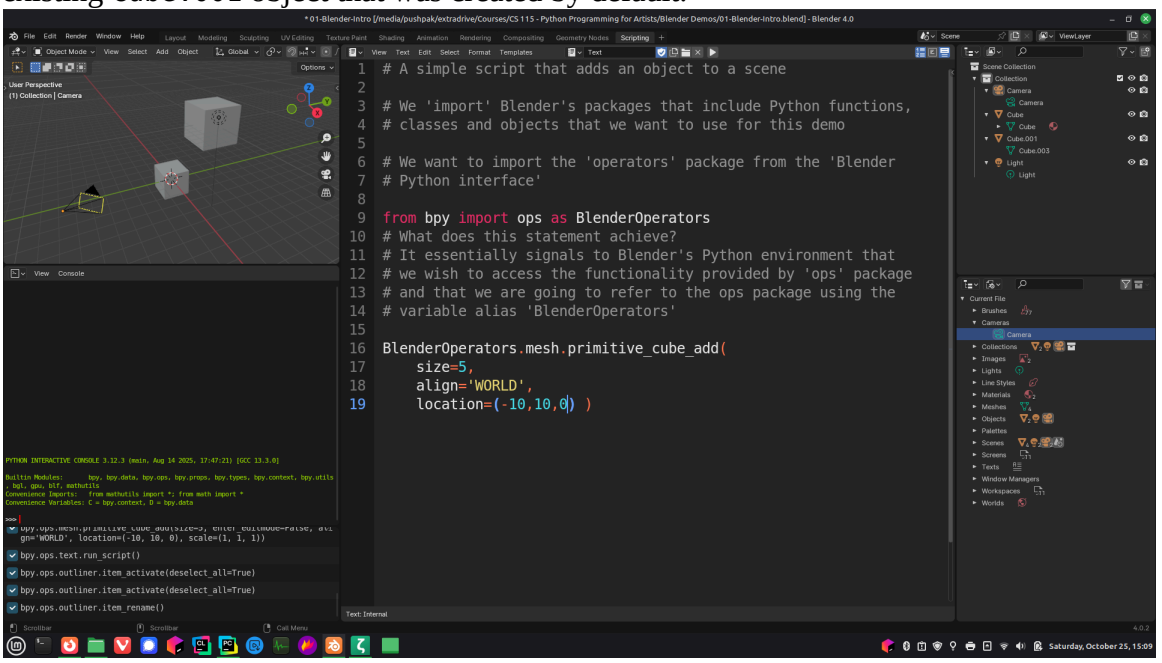
With your code written in the text editor, click the **Run Script** button (a play icon) in the header of the text editor. When you run the script, a new cube will appear in your scene.



01-Blender-Python-Add-Cube.png

Unfortunately, the new cube has been added on top of our existing cube. We should provide a suitable value for the `location=` argument to the `primitive_cube_add` function such that we can see both the cubes.

Let us change the code to add the cube at location (-10, 10, 0). Don’t forget to delete the existing `Cube.001` object that was created by default.

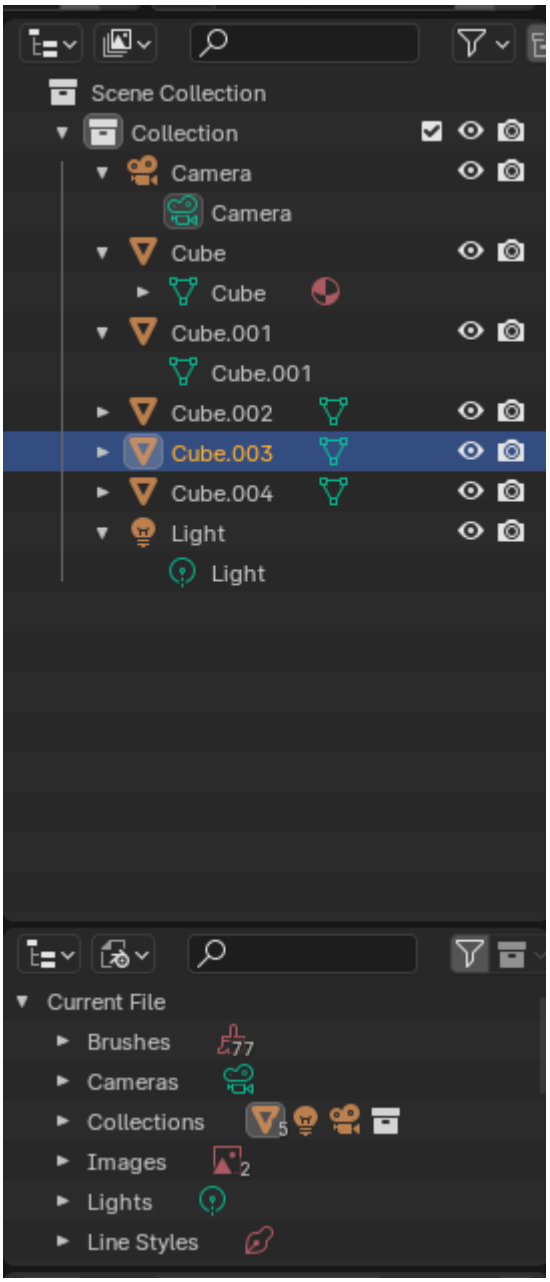


Data Organization within Blender

We can see that Blender has chosen to give some default values for the cube e.g. the name of the object is `Cube.NNN` where `NNN` is a numeric value.

Data within blender is organized into multiple categories, or `collections`

We observe that Blender has added new data on top of the existing data in our scene.



Error-New-Objects.png

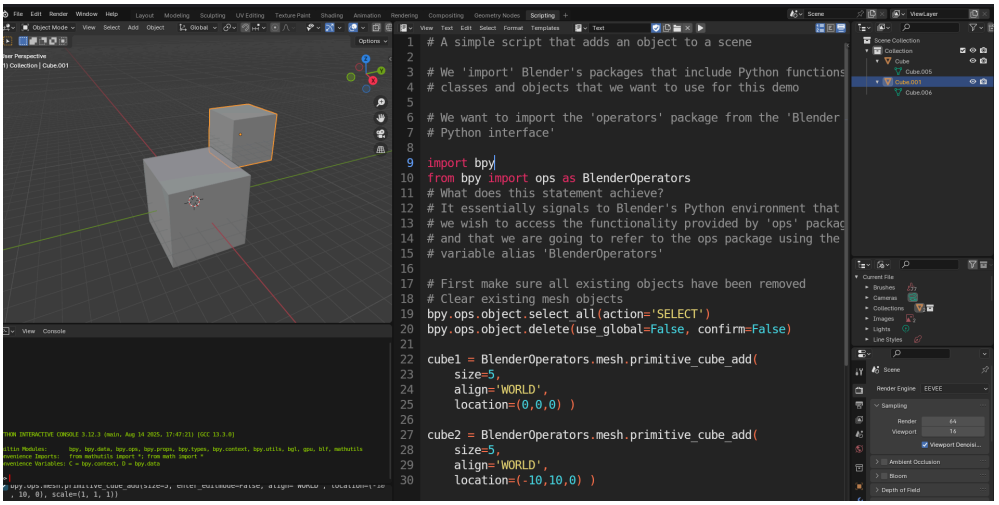
What we would like is to start with a clean scene each time we run our code. What we would have to do to achieve this is to go through Blender's collection of objects and delete them from the scene before we add any new cubes.

We utilize a two-step approach in removing the objects from the scene:

- First we 'select' all the objects via python (as if the user pressed Ctrl-A on the object list)
- Then we execute the `delete` command

```
# Clear existing mesh objects
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete(use_global=False, confirm=False)
bpy.ops.outliner.orphans_purge()
```

We add this code to the top of our python file and run it again.



2-Cubes-Scene.png

Success!! Now our scene has exactly 2 objects, and each of them is a cube.

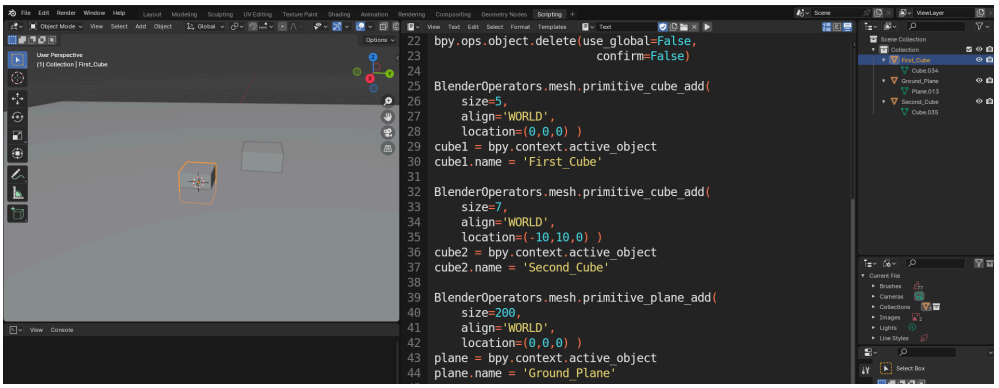
Finally, let us add a plane beneath the cubes that is about 200 x 200 units in dimensions. The code looks very similar to how we added the cubes, but the function to call is `primitive_plane_add`

```
BlenderOperators.mesh.primitive_plane_add(
    size=200,
    align='WORLD',
    location=(0,0,0) )
```

The plane seems to cut through our cube objects. This is because when we insert an object in the scene, the `location=` parameter actually specifies the location of the *center of the object*. In order for our cubes to be completely visible above the plane, we would have to move them up explicitly as below:

- Change the cube addition code to explicitly give the inserted object a user-friendly name
- Use this name to access the object
- Apply a suitable translation to the object

```
BlenderOperators.mesh.primitive_cube_add(
    size=5,
    align='WORLD',
    location=(0,0,0) )
cube1 = bpy.context.active_object
cube1.name = 'First_Cube'

BlenderOperators.mesh.primitive_cube_add(
    size=7,
    align='WORLD',
    location=(-10,10,0) )
cube2 = bpy.context.active_object
cube2.name = 'Second_Cube'

BlenderOperators.mesh.primitive_plane_add(
    size=200,
    align='WORLD',
    location=(0,0,0) )
plane = bpy.context.active_object
plane.name = 'Ground_Plane'
```



Objects-Custom-Names.png

Now, move the `First_Cube` object in the 'up' direction by half of its total size. Blender uses the Z-up convention for the world space, so if we want to move this object 'higher', we have to apply a translation in Z of amount 2.5 units.

```
# Get the underlying mesh
cubeMesh = bpy.data.objects['First_Cube'].data
translationVector = mathutils.Vector((0,0,2.5))
transformMatrix = mathutils.Matrix.Translation(translationVector)
cubeMesh.transform(transformMatrix)
```

Applying this code moves the first cube above the plane. Do the same for the second cube, but move it a bit higher so that it is floating on the plane.

```
# Now do the same for the second cube
# Get the underlying mesh
cubeMesh = bpy.data.objects['Second_Cube'].data
translationVector = mathutils.Vector((0,0,5))
transformMatrix = mathutils.Matrix.Translation(translationVector)
cubeMesh.transform(transformMatrix)
```

Adding materials to the objects

We follow the same playbook when adding materials to objects

- First, get the object using its name
- Then create a new material object
- Finally, link the mesh with the material object

```
# Plane
planeMesh = bpy.data.objects['Ground_Plane'].data
newMaterial = bpy.data.materials.new(name="Plane_Material")
newMaterial.use_nodes = True
newMaterial.diffuse_color = ( 255, 0.0, 0.0, 1.0 )
newMaterial.metallic = 1.0
planeMesh.materials.append( newMaterial )

# Cube 1
cubeMesh = bpy.data.objects['First_Cube'].data
newMaterial = bpy.data.materials.new(name="Cube1_Material")
newMaterial.use_nodes = True
newMaterial.diffuse_color = ( 0.04847, 0.487, 0.8847, 1.0 )
newMaterial.metallic = 0.15
cubeMesh.materials.append( newMaterial )
```

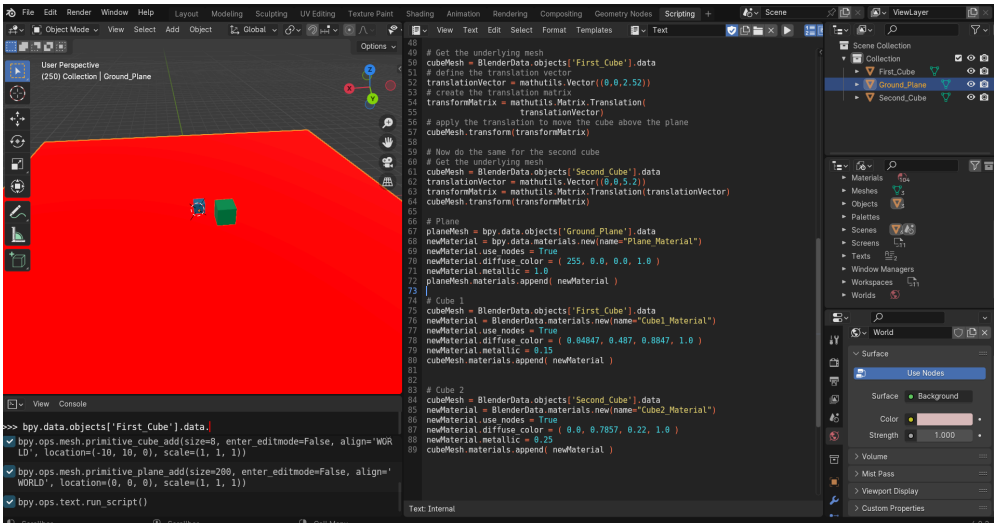
```
# Cube 2
cubeMesh = bpy.data.objects['Second_Cube'].data
newMaterial = bpy.data.materials.new(name="Cube2_Material")
newMaterial.use_nodes = True
newMaterial.diffuse_color = ( 0.0, 0.7857, 0.22, 1.0 )
newMaterial.metallic = 0.25
cubeMesh.materials.append( newMaterial )
```

Final TODO:

Change all references to `bpy.data` to `BlenderData` for code readability.

```
# Plane
planeMesh = BlenderData.objects['Ground_Plane'].data
newMaterial = BlenderData.materials.new(name="Plane_Material")
newMaterial.use_nodes = True
newMaterial.diffuse_color = ( 255, 0.0, 0.0, 1.0 )
newMaterial.metallic = 1.0
planeMesh.materials.append( newMaterial )

# Cube 1
cubeMesh = BlenderData.objects['First_Cube'].data
newMaterial = BlenderData.materials.new(name="Cube1_Material")
newMaterial.use_nodes = True
newMaterial.diffuse_color = ( 0.04847, 0.487, 0.8847, 1.0 )
newMaterial.metallic = 0.15
cubeMesh.materials.append( newMaterial )


# Cube 2
cubeMesh = BlenderData.objects['Second_Cube'].data
newMaterial = BlenderData.materials.new(name="Cube2_Material")
newMaterial.use_nodes = True
newMaterial.diffuse_color = ( 0.0, 0.7857, 0.22, 1.0 )
newMaterial.metallic = 0.25
cubeMesh.materials.append( newMaterial )
```

Done! Our scene should now look as below:



Scene-Render-with-Materials.png

This should provide a good introduction to the basic functionality of the python-based interface to operations, data, and scripting within Blender.

Appendix - Further forays into Blender

Data Types and Data Hierarchy in Blender

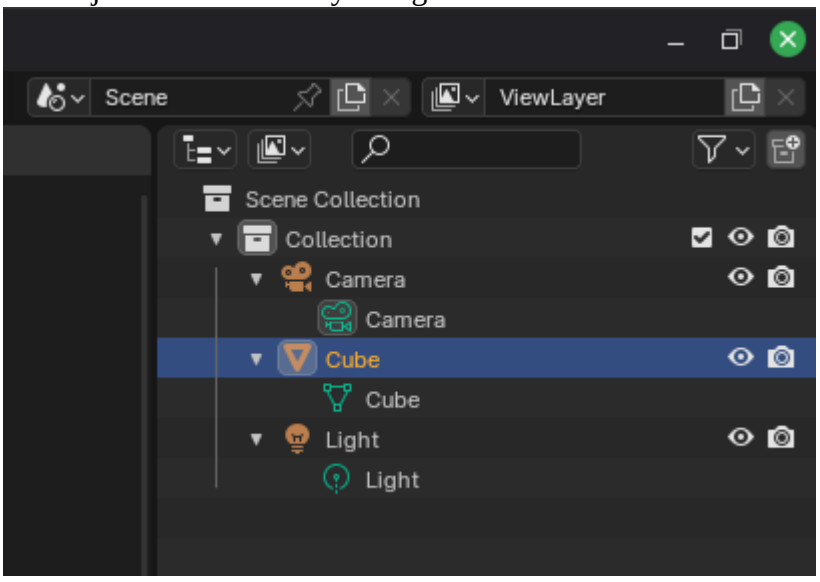
E.g. typing the following code in the Blender python console shows us the types of data stored in its memory.

```
>>> bpy.data.objects
<bpy_collection[3], BlendDataObjects>
```

Each data object can be accessed by its name, or its index.

```
>>> bpy.data.objects[0]
bpy.data.objects['Camera']

>>> bpy.data.objects['Light']
bpy.data.objects['Light']
```

The objects are denoted by orange icons in Blender's Scene view

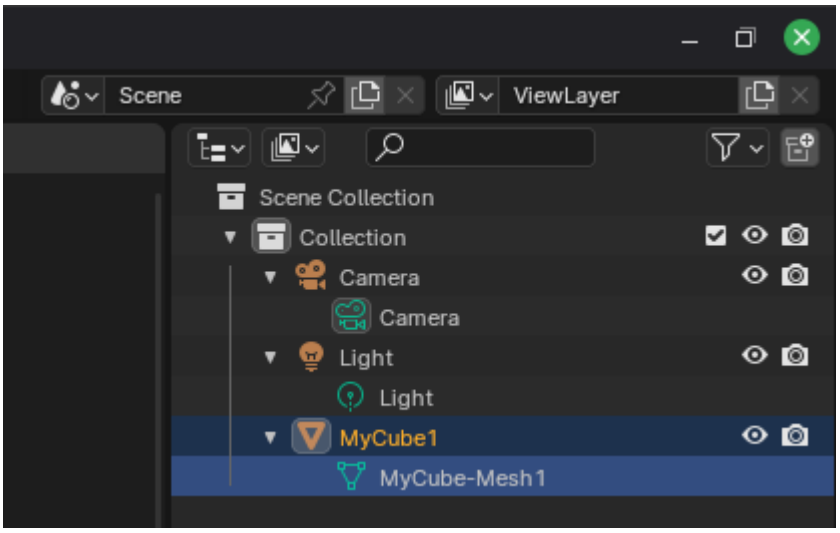


While the items highlighted in orange denote the Blender Data Objects (BlendDataObject), each underlying data is of a very different 'type'. These are stored in their respective type collections in Blender.

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

>>> bpy.data.meshes['Cube']
bpy.data.meshes['Cube']

>>> bpy.data.objects['Cube'].name
'Cube'

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

```
>>> bpy.data.objects['Cube'].name = "MyCube1"
>>> bpy.data.meshes['Cube'].name = "MyCube-Mesh1"
>>>
```

The above code changes the names of the Cube object, and the specific Cube mesh according to our instructions.



01-Blender-Rename-Objects.png

Modifying object properties

Let us use python to modify the properties of this object.

Example: Deleting all the materials

Blender allows the user access to individual materials applied to the objects. However, it does not support a `removeall` functionality that deletes all the materials in the scene. In order to remove the materials, we would have to loop through and remove them individually:

```
# Function to remove all material slots and materials from objects
def remove_materials_and_slots():
    # Loop through all objects in the scene
    for obj in bpy.data.objects:
        if obj.type == 'MESH' and obj.material_slots:
            # Select the object
            bpy.context.view_layer.objects.active = obj
            obj.select_set(True)
            # Remove all material slots
            while obj.material_slots:
                bpy.ops.object.material_slot_remove()

    # Remove any unused materials from the file
    bpy.ops.outliner.orphans_purge(do_local_ids=True,
        do_linked_ids=True, do_recursive=True)
    print("All materials and slots have been removed, and unused
        materials have been purged.")

# Run the function
remove_materials_and_slots()
```

Using the current 'context' of operations

`bpy.context`: Operating on the current selection

This module provides access to the “context” of your current Blender session, including the active object, selected objects, and the current mode.

Example: Moving the active object

```
import bpy
import mathutils

# Get the active object
active_object = bpy.context.active_object

# Move the object 2 units along the Y-axis
active_object.location += mathutils.Vector((0, 2, 0))
```

Performing an ‘operation’ without using Blender GUI

`bpy.ops`: Executing tools and operators

This module is used to call Blender’s built-in tools, just as if you were clicking a button in the user interface.

Example: Deleting all selected objects

```
from bpy import ops as BlenderOps
from bpy import data as BlenderData

# Deselect everything first
BlenderOps.object.select_all(action='DESELECT')

# Select all objects of type 'MESH'
for obj in BlenderData.objects:
    if obj.type == 'MESH':
        obj.select_set(True)

# Delete the selected objects
BlenderOps.object.delete()
```

Performing editable operations on individual primitives in a mesh

`bmesh`: Editing mesh data

For more complex mesh manipulation in Edit Mode, use the `bmesh` module. You create a `bmesh` object, populate it with the mesh’s data, make changes, and then write the changes back to the mesh.

Example: Moving all vertices of a mesh

```
from bpy import context as BlenderContext
from bpy import ops as BlenderOps
import bmesh

# Get the active mesh object and ensure we are in 'Object' Mode
obj = BlenderContext.active_object
BlenderOps.object.mode_set(mode='OBJECT')
```

```
# Create a new BMesh object from the mesh data
bm = bmesh.new()
bm.from_mesh(obj.data)

# Iterate through and modify each vertex
for vert in bm.verts:
    vert.co.x += 1.0

# Write the changes back to the mesh and free the BMesh
bm.to_mesh(obj.data)
bm.free()

# Revert to 'Edit' Mode
BlenderOps.object.mode_set(mode='EDIT')
```