



# Blender - Python API

#14



Serdar ARITAN

Department of Computer Graphics  
Hacettepe University, Ankara, Turkey



### Operators (`bpy.types.Operators`)

The **Operator** class allows calling functions from the graphic interface. They are, essentially, commands that can be run in Blender.).

- A static string named `bl_idname` that contains a **unique name** by which the operator goes internally
- A static string named `bl_label` that contains the **displayed name** of the operator
- A `poll()` class method that verifies that the conditions for executing the operator are met and return either **True** or **False**
- An `execute()` method that runs when the operator is executed, returning a set of possible running states
- Optionally, a **docstring** that Blender will display as additional information



### Operators (`bpy.types.Operators`)

Following the Blender guidelines, the name starts with `OBJECT_OT`. Soon after the (optional) `docstring` comes `bl_idname` and `bl_label`, the two attributes that Blender uses respectively as an identifier and description of the operator:

```
class OBJECT_OT_collector_types(bpy.types.Operator):  
    """Create collections based on objects types"""  
    bl_idname = "object.pckt_type_collector"  
    bl_label = "Create Type Collections"  
    @classmethod  
    def poll(cls, context):  
        return False  
    def execute(self, context):  
        # our code goes here  
        return {'FINISHED'}
```

The `poll()` and `execute()` methods, at this stage, neither allow nor perform any action



## Blender/Python API Blender Add-on

### Operators (`bpy.types.Operators`)

`poll()` verifies that the conditions for running the operator are met. This method is marked with a `@classmethod` decorator that allows us to validate the conditions before the operator is run. `@classmethod` is a method that is bound to a class rather than its object. It doesn't require creation of a class instance, much like `staticmethod`.

Since this operator collects objects in the scene, it should not be used if the scene is empty:

```
@classmethod
def poll(cls, context):
    return len(context.scene.objects) > 0
```



```
from datetime import date

class Person:
    def __init__(self, age):
        self.age = age

    @classmethod
    def fromYear(cls, year):
        return cls(date.today().year - year)

# create fromYear class method
personFromNormal = Person(25)
print(personFromNormal.age)

personFromClassmethod = Person.printAge(1966)
print(personFromClassmethod.age)
```



### Operators (`bpy.types.Operators`)

After an operator is invoked, Blender runs its `execute()` method.

```
def execute(self, context):  
    main(context)  
    return {'FINISHED'}
```



### Operators (`bpy.types.Operators`)

```
def main(context):  
    mesh_cl = bpy.data.collections.new("Mesh")  
    light_cl = bpy.data.collections.new("Light")  
    cam_cl = bpy.data.collections.new("Camera")  
    context.scene.collection.children.link(mesh_cl)  
    context.scene.collection.children.link(light_cl)  
    context.scene.collection.children.link(cam_cl)  
    for ob in context.scene.objects:  
        if ob.type == 'MESH':  
            mesh_cl.objects.link(ob)  
        elif ob.type == 'LIGHT':  
            light_cl.objects.link(ob)  
        elif ob.type == 'CAMERA':  
            cam_cl.objects.link(ob)
```



### Operators (`bpy.types.Operators`)

# Required to use F3 search "OBJECT\_OT\_collector\_types" for quick access.

```
def menu_func(self, context):  
    self.layout.operator(OBJECT_OT_collector_types.bl_idname,  
                        text=OBJECT_OT_collector_types.bl_label)
```



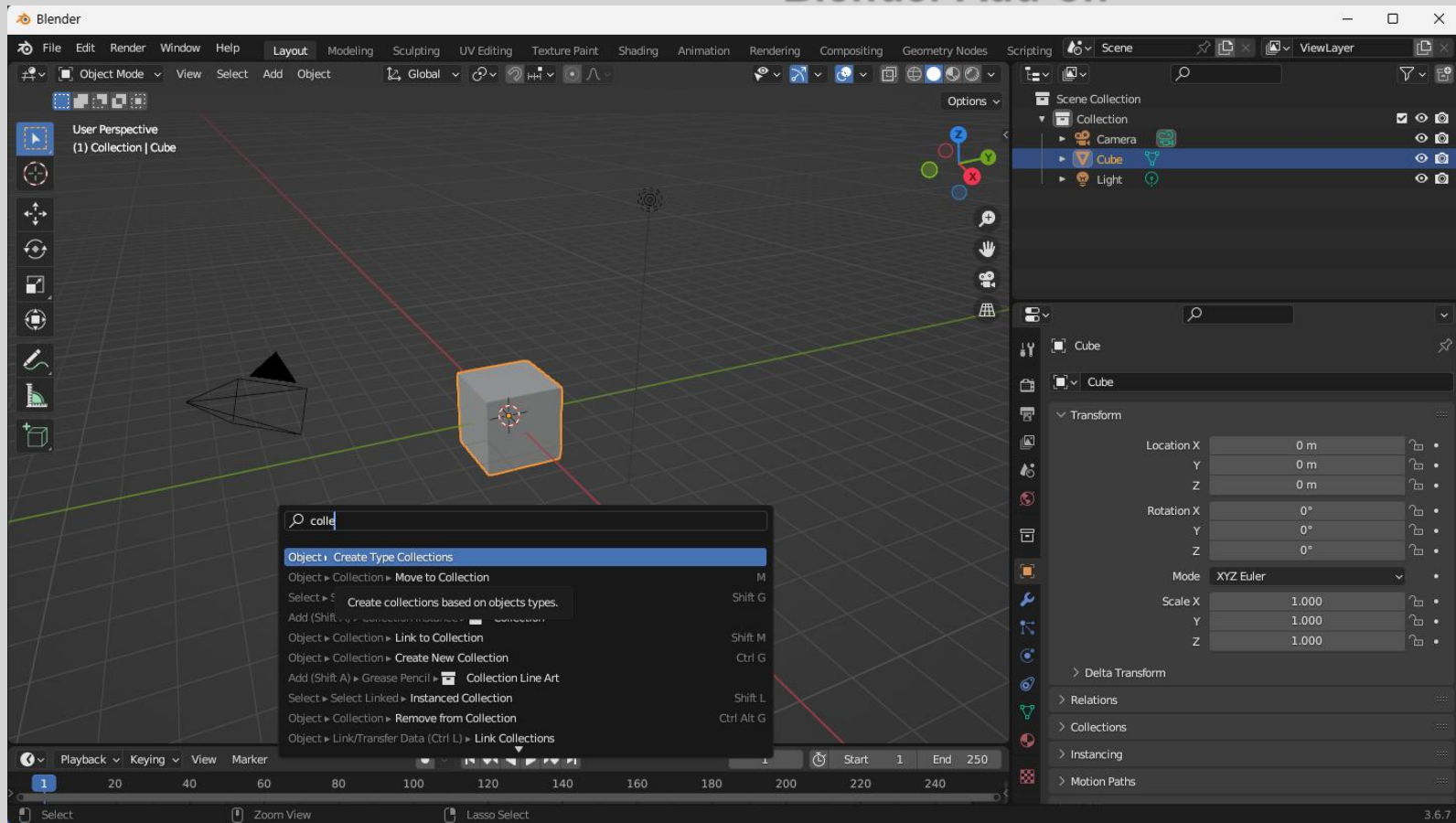


### Operators (`bpy.types.Operators`)

Add-on adds an operator to Blender when enabled and removes it when it is disabled. This is done via the `bpy.utils.register_class()` and `bpy.utils.unregister_class()` functions is invoked.

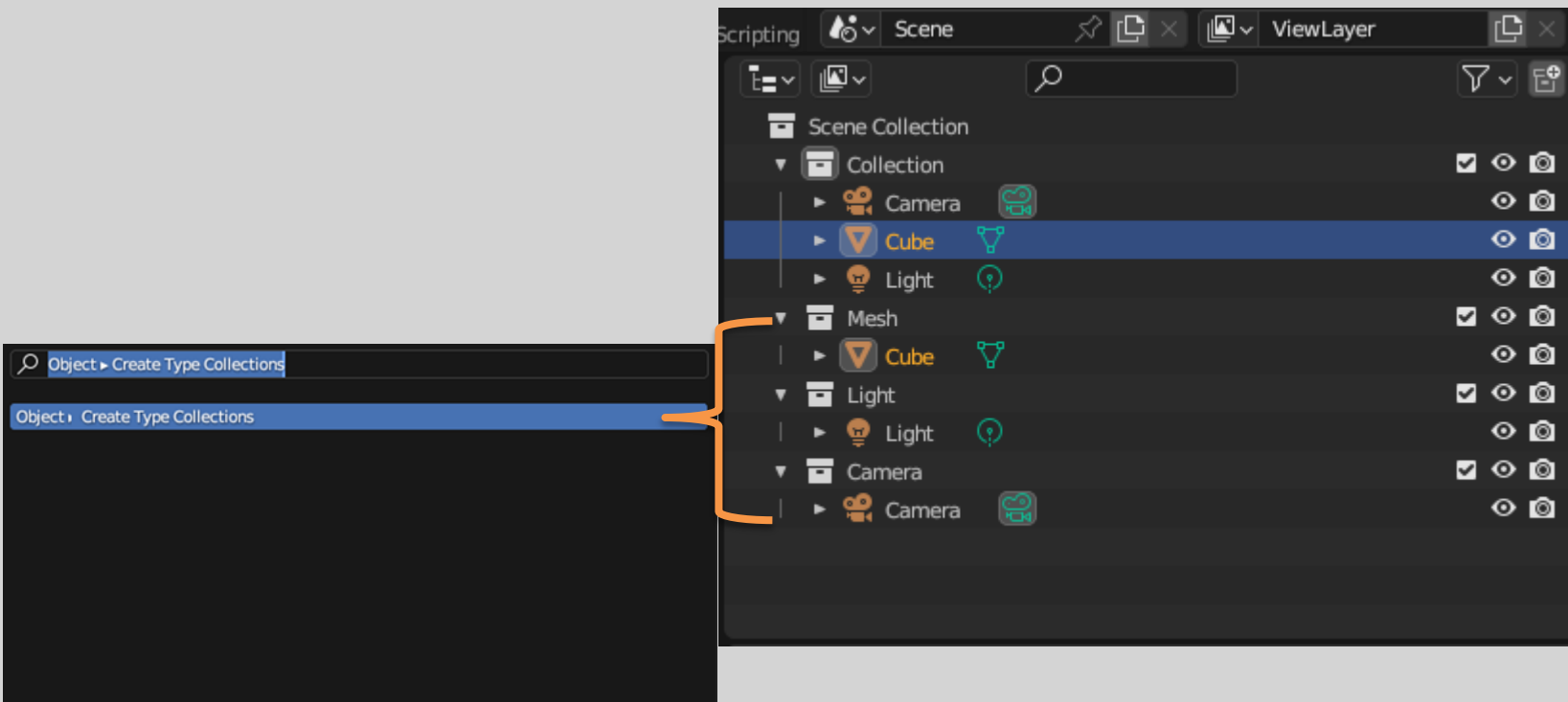
```
def register():  
    bpy.utils.register_class(OBJECT_OT_collector_types)  
    bpy.types.VIEW3D_MT_object.append(menu_func)  
  
def unregister():  
    bpy.utils.unregister_class(OBJECT_OT_collector_types)  
    bpy.types.VIEW3D_MT_object.remove(menu_func)
```

## Blender/Python API Blender Add-on



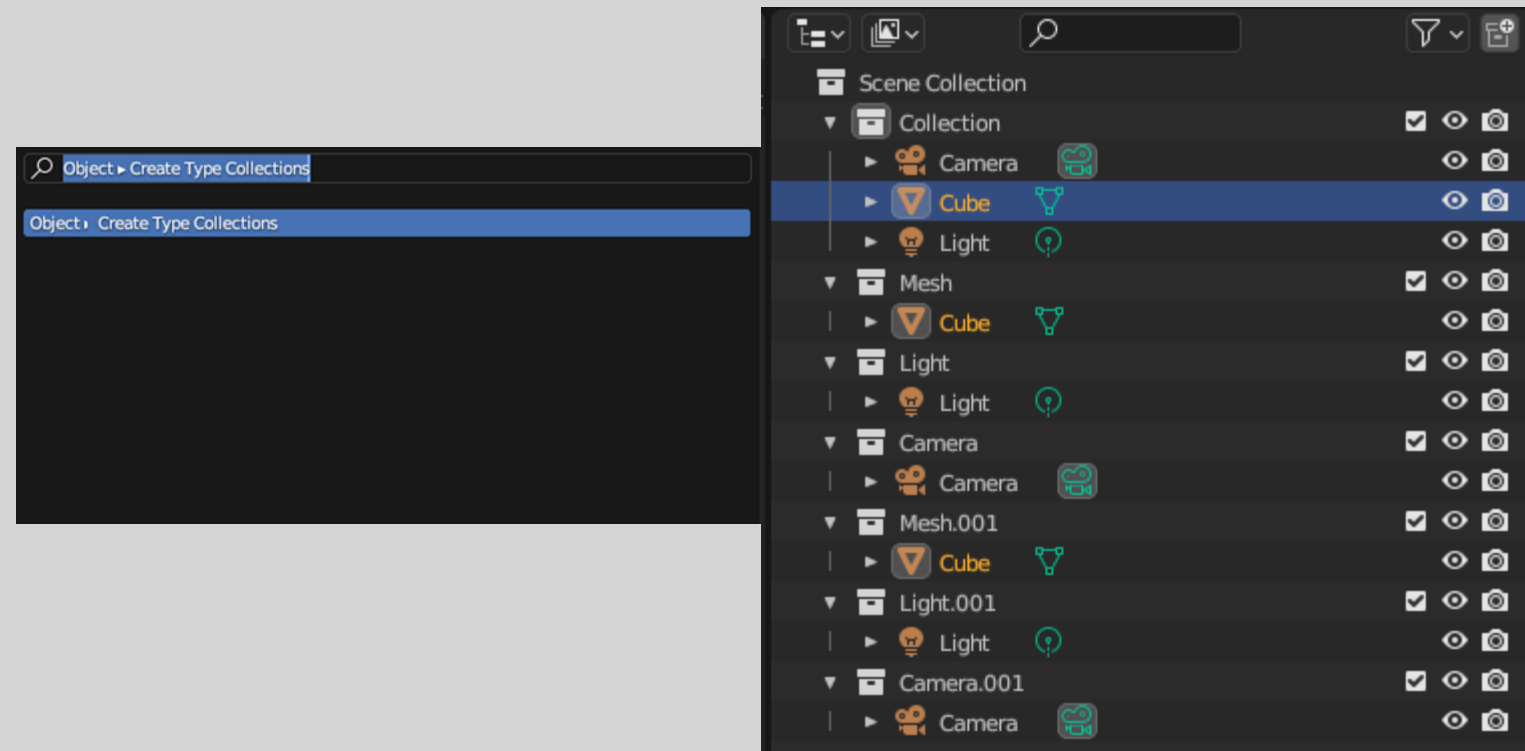
# Blender/Python API

## Blender Add-on



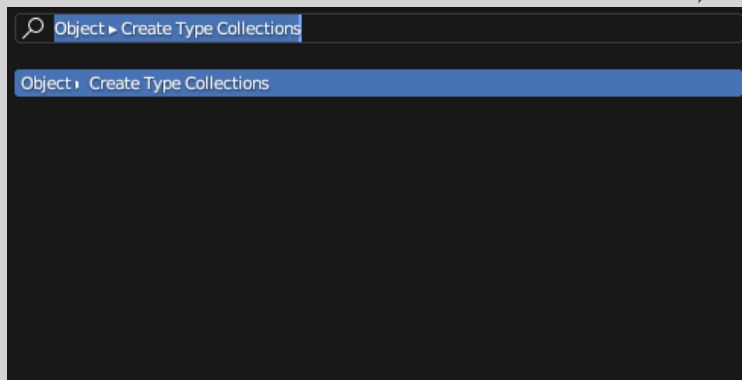
# Blender/Python API

## Blender Add-on



## Blender/Python API Blender Add-on

Add empty into collection



No  
empty





## Blender/Python API Blender Add-on

```
@staticmethod
def get_collection(name):
    """Returns the collection named after the given
    argument. If it doesn't exist, a new collection
    is created and linked to the scene
    """

    try:
        return bpy.data.collections[name]
    except KeyError:
        cl = bpy.data.collections.new(name)
        bpy.context.scene.collection.children.link(cl)
        return cl
```

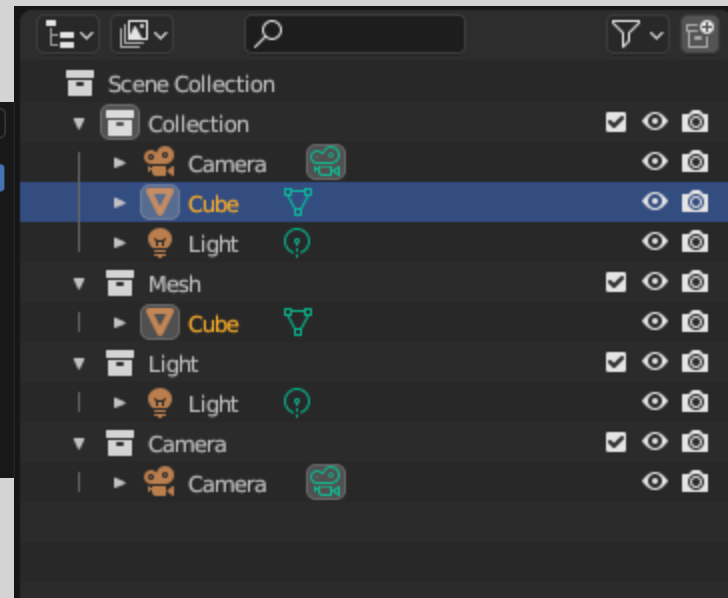
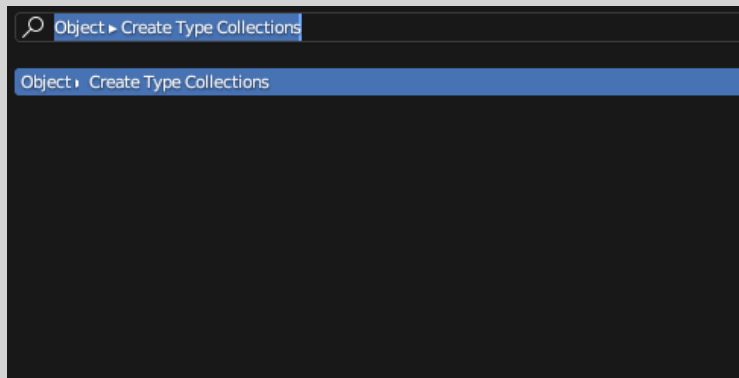


## Blender/Python API Blender Add-on

```
def execute(self, context):  
    for ob in context.scene.objects:  
        cl = self.get_collection(ob.type.title())  
        cl.objects.link(ob)  
  
    return {'FINISHED'}
```

# Blender/Python API

## Blender Add-on







## Blender/Python API Blender Add-on

Object ► Create Type Collections

Object ► Create Type Collections

User Perspective  
(1) Collection | Cube

Report: Error

Python: Traceback (most recent call last):  
File "iaddon\_operator\_2.py", line 44, in execute  
RuntimeError: Error: Object 'Cube' already in collection 'Mesh'

We need to enclose object linking in a **try/except** statement to avoid that:

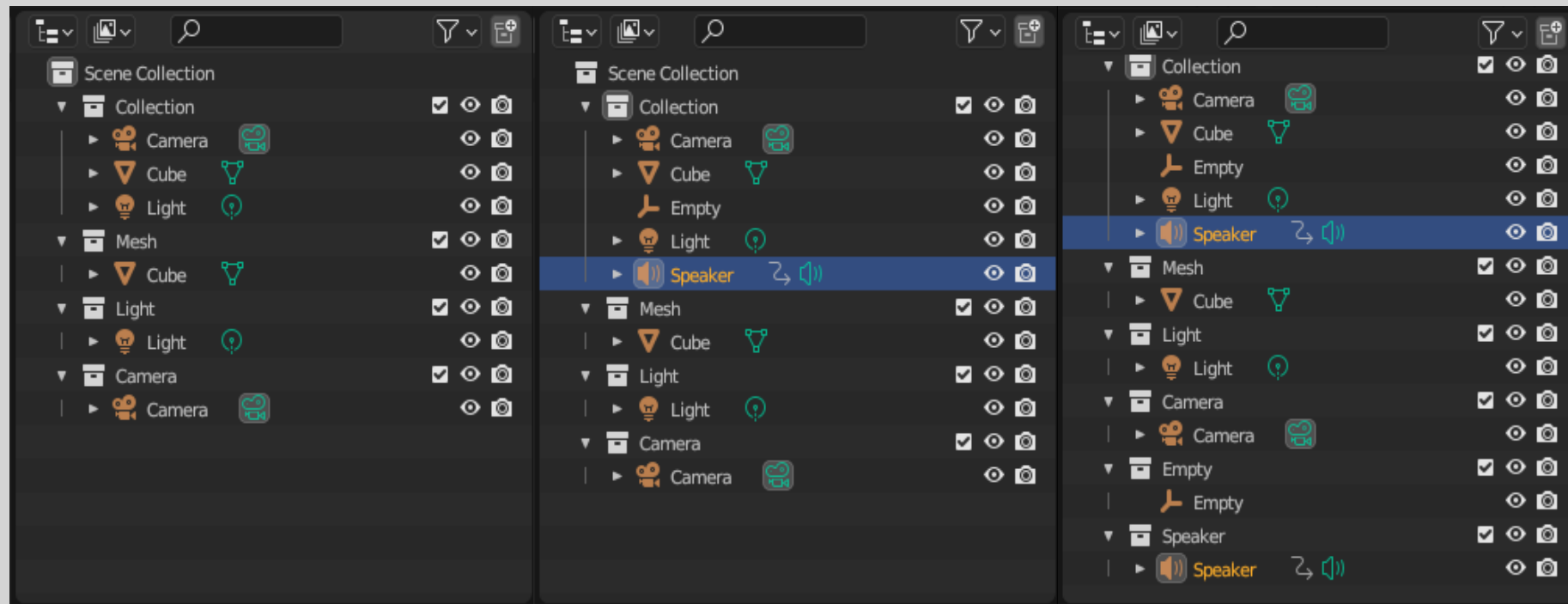
```
cl.objects.link(ob)
```



```
def execute(self, context):  
    for ob in context.scene.objects:  
        cl = self.get_collection(ob.type.title())  
        try:  
            cl.objects.link(ob)  
        except RuntimeError:  
            continue  
  
    return {'FINISHED'}
```

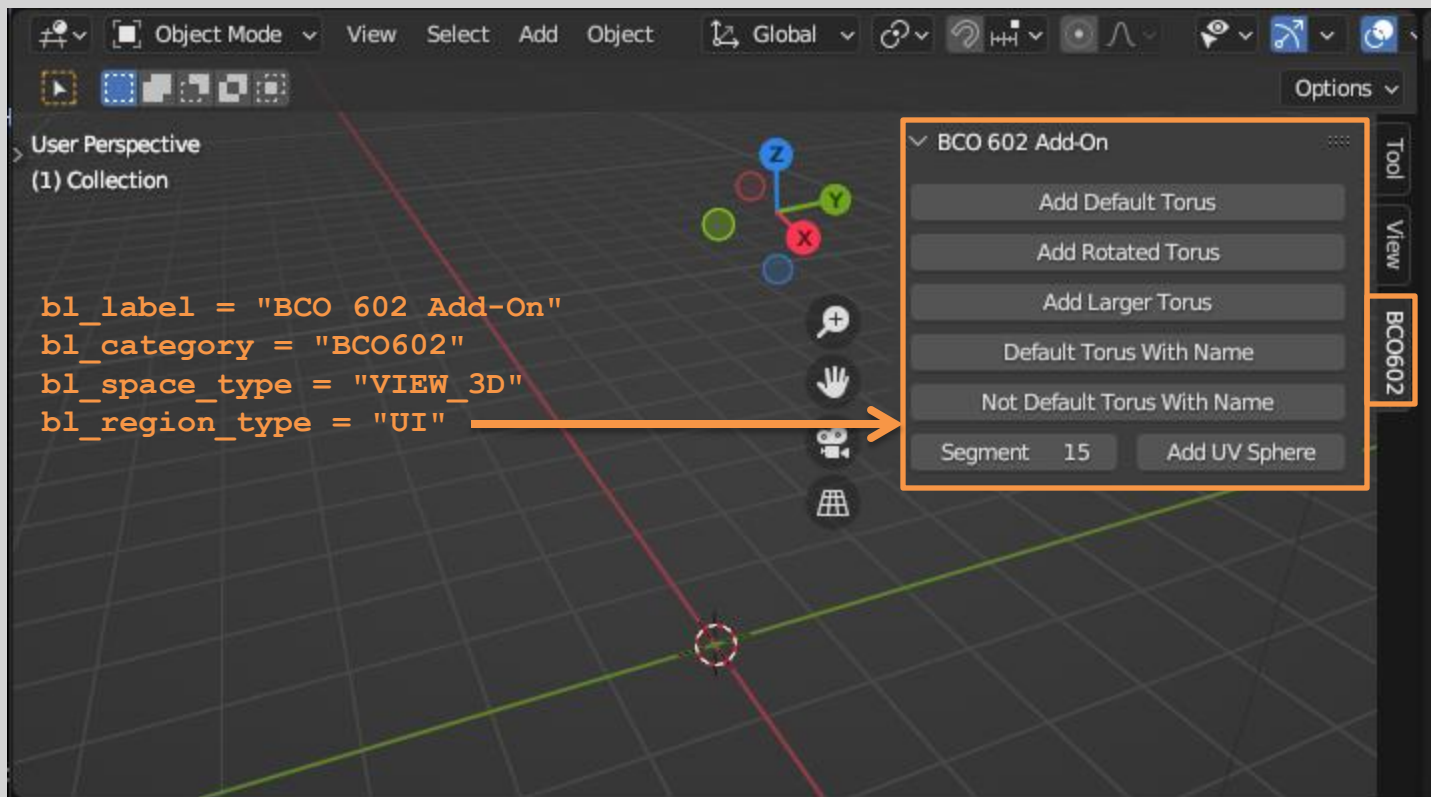
# Blender/Python API

## Blender Add-on



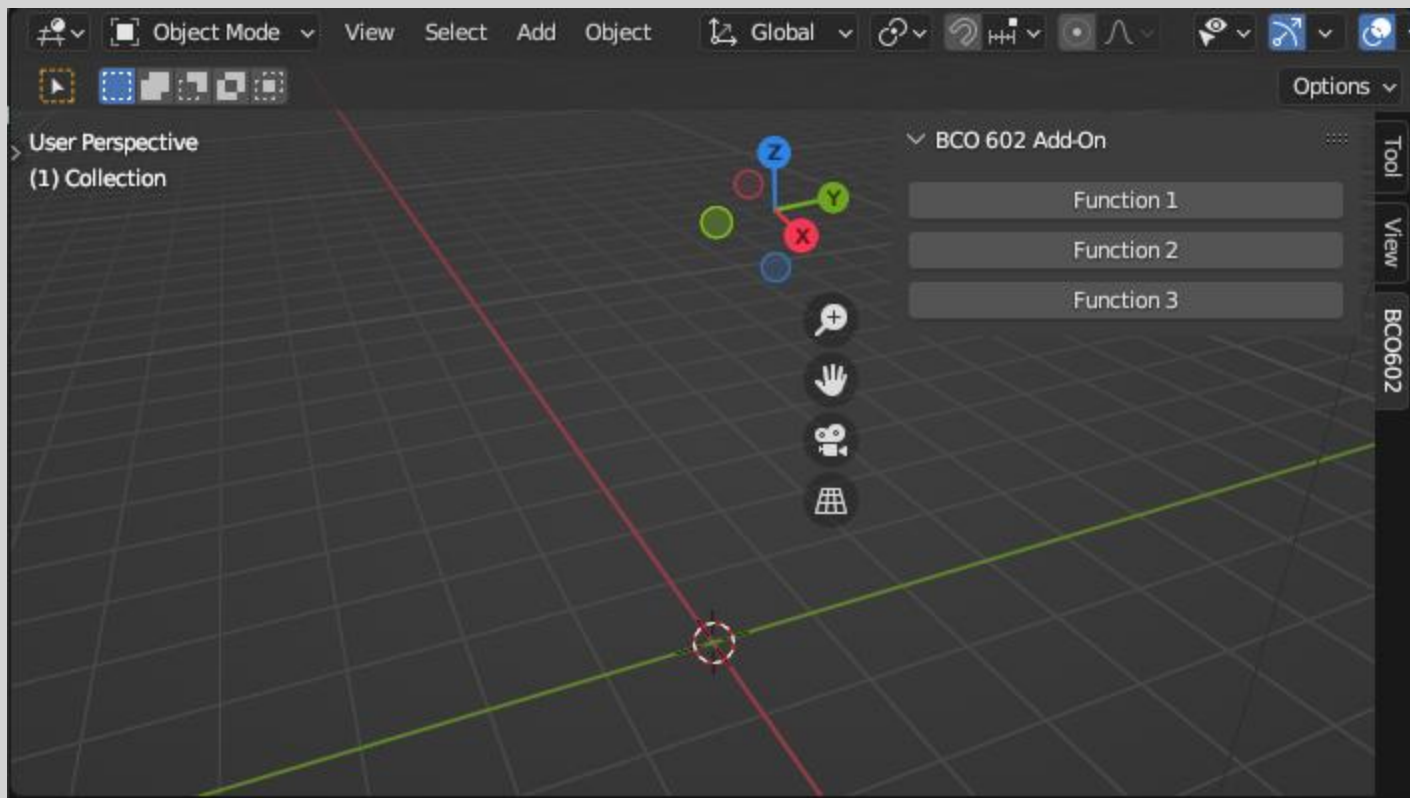


## Blender/Python API Blender Add-on



# Blender/Python API

## Blender Add-on





```
import bpy

class EXAMPLE_OT_func_1(bpy.types.Operator):
    bl_idname = "example.func_1"
    bl_label = "Function 1"

    def execute(self, context):
        # Implement your first function here
        self.report({'INFO'}, f"This is {self.bl_idname}")
        return {'FINISHED'}

class EXAMPLE_OT_func_2(bpy.types.Operator):
```



## Blender/Python API Blender Add-on

```
class EXAMPLE_PT_panel(bpy.types.Panel):
    bl_label = "BCO 602 Add-On"
    bl_category = "BCO602"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

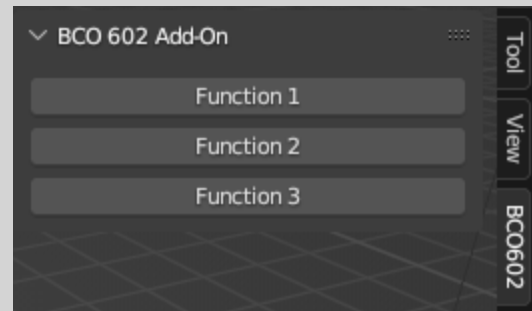
    def draw(self, context):
        layout = self.layout
        layout.operator(EXAMPLE_OT_func_1.bl_idname)
        layout.operator(EXAMPLE_OT_func_2.bl_idname)
        layout.operator(EXAMPLE_OT_func_3.bl_idname)

classes = (EXAMPLE_OT_func_1, EXAMPLE_OT_func_2, EXAMPLE_OT_func_3, EXAMPLE_PT_panel)

def register():
    for cls in classes:
        bpy.utils.register_class(cls)

def unregister():
    for cls in classes:
        bpy.utils.unregister_class(cls)

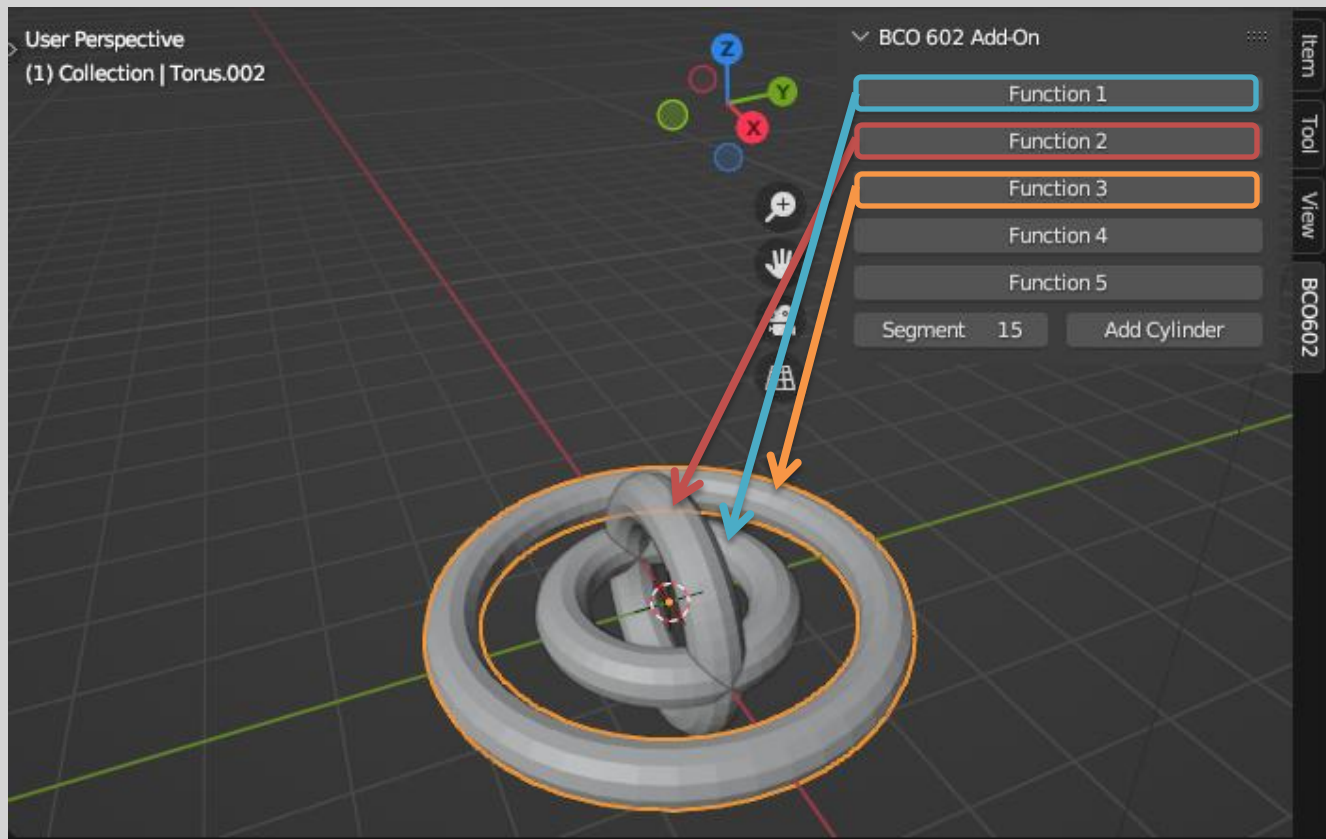
if __name__ == "__main__":
    register()
```







## Blender/Python API Blender Add-on







# SCRIPT LANGUAGES FOR ANIMATION

## Blender/Python API Blender Add-on

```
class EXAMPLE_OT_func_1(bpy.types.Operator):
    bl_idname = "example.func_1"
    bl_label = "Function 1"

    def execute(self, context):
        # Implement your first function here
        # Add default torus
        bpy.ops.mesh.primitive_torus_add(location=(0, 0, 0), rotation=(0, 0, 0), major_radius=1, minor_radius=0.25, abso_major_rad=1.25, abso_minor_rad=0.75)
        self.report({'INFO'}, f"This is {self.bl_idname}")
        return {'FINISHED'}

class EXAMPLE_OT_func_2(bpy.types.Operator):
    bl_idname = "example.func_2"
    bl_label = "Function 2"

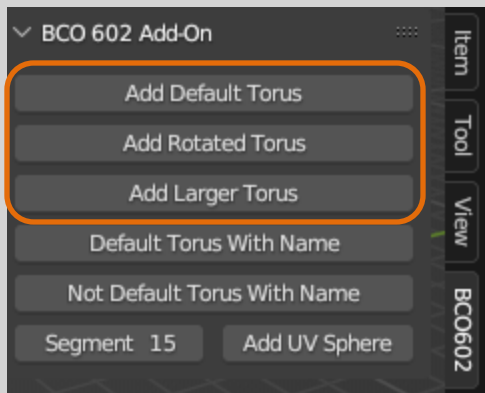
    def execute(self, context):
        # Implement your second function here
        # Add default 90 degree rotated torus
        bpy.ops.mesh.primitive_torus_add(location=(0, 0, 0), rotation=((90/57.30), 0, 0))
        self.report({'INFO'}, f"This is {self.bl_idname}")
        return {'FINISHED'}

class EXAMPLE_OT_func_3(bpy.types.Operator):
    bl_idname = "example.func_3"
    bl_label = "Function 3"

    def execute(self, context):
        # Implement your third function here
        bpy.ops.mesh.primitive_torus_add(major_radius=2)
        self.report({'INFO'}, f"This is {self.bl_idname}")
        return {'FINISHED'}
```



## Blender/Python API Blender Add-on



```
class EXAMPLE_OT_func_1(bpy.types.Operator):  
    bl_idname = "example.func_1"  
    bl_label = "Add Default Torus"
```

```
class EXAMPLE_OT_func_2(bpy.types.Operator):  
    bl_idname = "example.func_2"  
    bl_label = "Add Rotated Torus"
```

```
class EXAMPLE_OT_func_3(bpy.types.Operator):  
    bl_idname = "example.func_3"  
    bl_label = "Add Larger Torus"
```

```
class EXAMPLE_OT_func_4(bpy.types.Operator):  
    bl_idname = "example.func_4"  
    bl_label = "Function using variables"
```

```
custom_prop: bpy.props.StringProperty(default="Default Value")  
custom_float : bpy.props.FloatProperty(name="My Floating Point", default= 1.0)
```



## Blender/Python API Blender Add-on

```
class EXAMPLE_PT_panel(bpy.types.Panel):
```

```
    bl_label = "BCO 602 Add-On"
```

```
    bl_category = "BCO602"
```

```
    bl_space_type = "VIEW_3D"
```

```
    bl_region_type = "UI"
```

```
def draw(self, context):
```

```
    layout = self.layout
```

```
    layout.operator(EXAMPLE_OT_func_1.bl_idname)
```

```
    layout.operator(EXAMPLE_OT_func_2.bl_idname)
```

```
    layout.operator(EXAMPLE_OT_func_3.bl_idname)
```

```
op = layout.operator(EXAMPLE_OT_func_4.bl_idname, text="Default Torus With Name")
```

```
op.custom_prop = "Default Torus"
```

```
op = layout.operator(EXAMPLE_OT_func_4.bl_idname, text="Not Default Torus With Name")
```

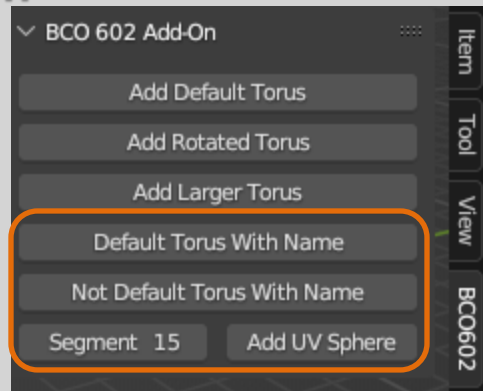
```
op.custom_prop = "Not Default Torus"
```

```
op.custom_float = 3
```

```
row = layout.row()
```

```
row.prop(context.window_manager, "int_p")
```

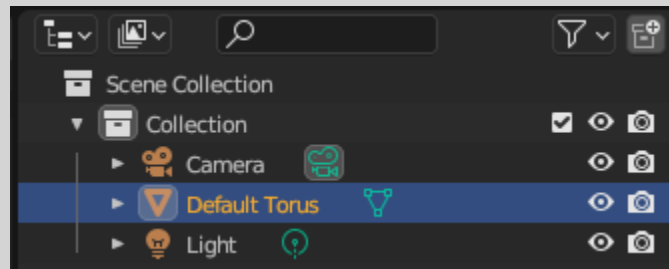
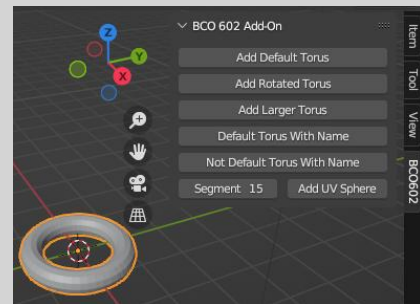
```
row.operator("mesh.primitive_uv_sphere_add").segments = context.window_manager.int_p
```





## Blender/Python API Blender Add-on

```
class EXAMPLE_OT_func_4(bpy.types.Operator):  
    bl_idname = "example.func_4"  
    bl_label = "Function using variables"  
  
    custom_prop: bpy.props.StringProperty(default="Default Value")  
    custom_float : bpy.props.FloatProperty(name="My Floating Point", default= 1.0)  
  
    def execute(self, context):  
        # Implement your fourth function here  
        # Add custom torus  
        v = self.custom_prop  
        f = self.custom_float  
        bpy.ops.mesh.primitive_torus_add(major_radius=f)  
        bpy.context.object.name = v  
        self.report({'INFO'}, f"String Prop is '{v}', Float Prop is '{f}'")  
        return {'FINISHED'}
```





## Blender/Python API Blender Add-on

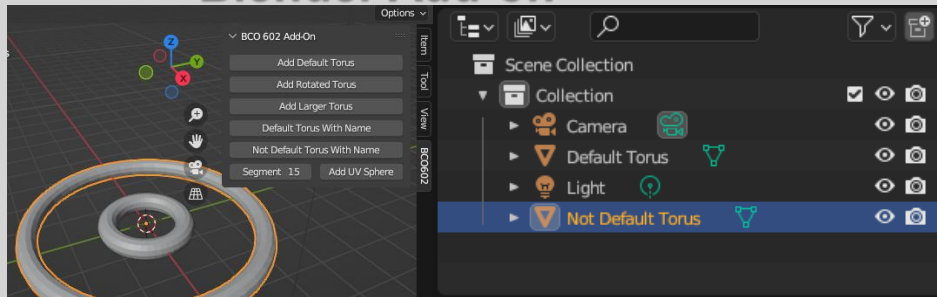
```
class EXAMPLE_PT_panel(bpy.types.Panel):  
    bl_label = "BCO 602 Add-On"  
    bl_category = "BCO602"  
    bl_space_type = "VIEW_3D"  
    bl_region_type = "UI"
```

```
def draw(self, context):  
    layout = self.layout  
    layout.operator(EXAMPLE_OT_func_1.bl_idname)  
    layout.operator(EXAMPLE_OT_func_2.bl_idname)  
    layout.operator(EXAMPLE_OT_func_3.bl_idname)
```

```
op = layout.operator(EXAMPLE_OT_func_4.bl_idname, text="Default Torus With Name")  
op.custom_prop = "Default Torus"
```

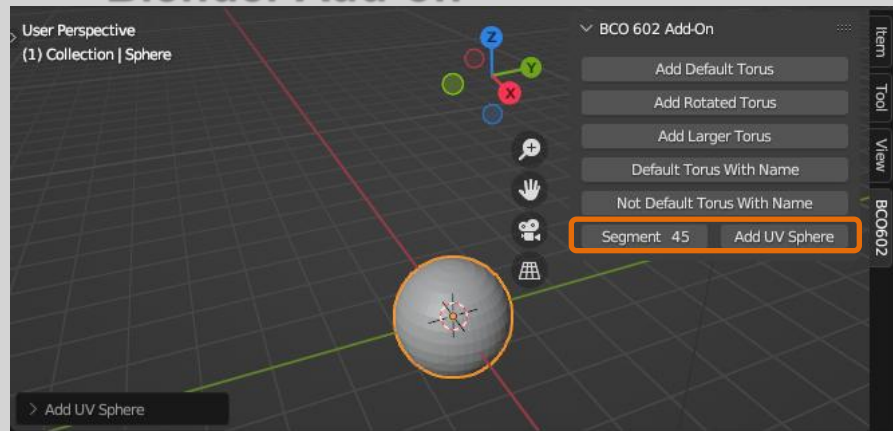
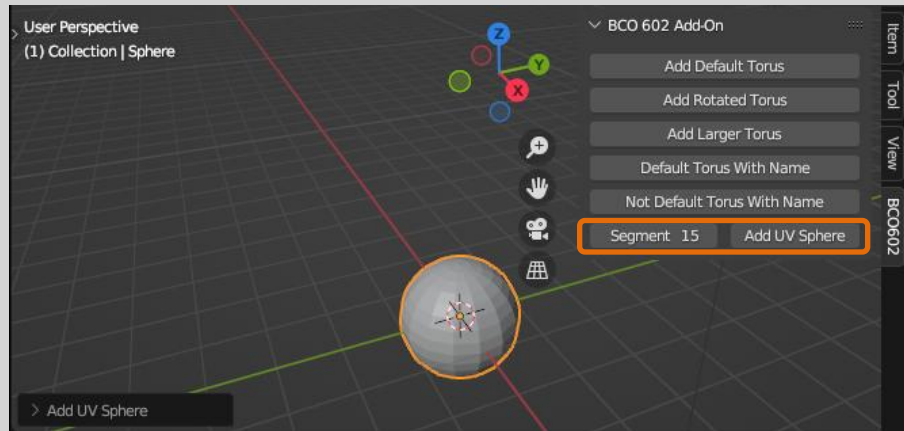
```
op = layout.operator(EXAMPLE_OT_func_4.bl_idname, text="Not Default Torus With Name")  
op.custom_prop = "Not Default Torus"  
op.custom_float = 3
```

```
row = layout.row()  
row.prop(context.window_manager, "int_p")  
row.operator("mesh.primitive_uv_sphere_add").segments = context.window_manager.int_p
```





## Blender/Python API Blender Add-on



```
import bpy
```

```
bpy.types.WindowManager.int_p = bpy.props.IntProperty(name = "Segment", default=15)
```

```
class EXAMPLE_PT_panel(bpy.types.Panel):
```

```
.....
```

```
    row = layout.row()
```

```
    row.prop(context.window_manager, "int_p")
```

```
    row.operator("mesh.primitive_uv_sphere_add").segments = context.window_manager.int_p
```