Skip to content

Quickstart

This API is generally stable but some areas are still being extended and improved.

Blender Python API features:

- Edit any data the user interface can (Scenes, Meshes, Particles etc.).
- Modify user preferences, keymaps and themes.
- Run tools with own settings.
- Create user interface elements such as menus, headers and panels.
- Create new tools.
- Create interactive tools.
- Create new rendering engines that integrate with Blender.
- Subscribe to changes to data and it's properties.
- Define new settings in existing Blender data.
- Draw in the 3D Viewport using Python.

(Still) missing features:

- Create new space types.
- Assign custom properties to every type.

Before Starting

This document is intended to familiarize you with Blender Python API but not to fully cover each topic.

A quick list of helpful things to know before starting:

- Enable Developer Extra and Python Tooltips.
- The Python Console is great for testing one-liners; it has autocompletion so you can inspect the API quickly.
- Button tooltips show Python attributes and operator names (when enabled see above).
- The context menu of buttons directly links to this API documentation (when enabled see above).
- Many python examples can be found in the text editor's template menu.
- To examine further scripts distributed with Blender, see:
  - `scripts/startup/bl_ui` for the user interface.
  - `scripts/startup/bl_operators` for operators.

  Exact location depends on platform, see: directory layout docs.

Running Scripts

The two most common ways to execute Python scripts are using the built-in text editor or entering commands in the Python console. Both the *Text Edit* and *Python Console* are space types you can select from the header. Rather than manually configuring your spaces for Python development, you can us the *Scripting* workspace accessible from the Topbar tabs.

From the text editor you can open `.py` files or paste them from the clipboard, then test using *Run Script*. The Python Console is typically used for typi in snippets and for testing to get immediate feedback, but can also have entire scripts pasted into it. Scripts can also run from the command line with Blender but to learn scripting in Blender this isn't essential.

Key Concepts

Data Access

Accessing Data-Blocks

You can access Blender's data with the Python API in the same way as the animation system or user interface; this implies that any setting that can be changed via a button can also be changed with Python. Accessing data from the currently loaded blend-file is done with the module `bpy.data`. It giv access to library data, for example:

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

```
>>> bpy.data.scenes
<bpy_collection[1], BlendDataScenes>
```

```
>>> bpy.data.materials
<bpy_collection[1], BlendDataMaterials>
```

Accessing Collections

You will notice that an index as well as a string can be used to access members of the collection. Unlike Python dictionaries, both methods are available; however, the index of a member may change while running Blender.

```
>>> list(bpy.data.objects)
[bpy.data.objects["Cube"], bpy.data.objects["Plane"]]
```

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

```
>>> bpy.data.objects[0]
bpy.data.objects["Cube"]
```

Accessing Attributes

Once you have a data-block, such as a material, object, collection, etc., its attributes can be accessed much like you would change a setting using the graphical interface. In fact, the tooltip for each button also displays the Python attribute which can help in finding what settings to change in a script.

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

```
>>> bpy.data.scenes["Scene"]
bpy.data.scenes['Scene']
```

```
>>> bpy.data.materials.new("MyMaterial")
bpy.data.materials['MyMaterial']
```

For testing what data to access it's useful to use the Python Console, which is its own space type. This supports auto-complete, giving you a fast way to explore the data in your file.

Example of a data path that can be quickly found via the console:

```
>>> bpy.data.scenes[0].render.resolution_percentage
100
>>> bpy.data.scenes[0].objects["Torus"].data.vertices[0].co.x
1.0
```

Data Creation/Removal

When you are familiar with other Python APIs you may be surprised that new data-blocks in the bpy API cannot be created by calling the class:

```
>>> bpy.types.Mesh()
Traceback (most recent call last):
  File "<blender_console>", line 1, in <module>
TypeError: bpy_struct.__new__(type): expected a single argument
```

This is an intentional part of the API design. The Blender Python API can't create Blender data that exists outside the main Blender database (accessed through `bpy.data`), because this data is managed by Blender (save, load, undo, append, etc).

Data is added and removed via methods on the collections in `bpy.data`, e.g:

```
>>> mesh = bpy.data.meshes.new(name="MyMesh")
>>> print(mesh)
<bpy_struct, Mesh("MyMesh.001")>
```

```
>>> bpy.data.meshes.remove(mesh)
```

Custom Properties

Python can access properties on any data-block that has an ID (data that can be linked in and accessed from `bpy.data`). When assigning a property you can pick your own names, these will be created when needed or overwritten if they already exist.

This data is saved with the blend-file and copied with objects, for example:

```
bpy.context.object["MyOwnProperty"] = 42

if "SomeProp" in bpy.context.object:
    print("Property found")

# Use the get function like a Python dictionary
# which can have a fallback value.
value = bpy.data.scenes["Scene"].get("test_prop", "fallback value")

# dictionaries can be assigned as long as they only use basic types.
collection = bpy.data.collections.new("MyTestCollection")
collection["MySettings"] = {"foo": 10, "bar": "spam", "baz": {}}

del collection["MySettings"]
```

Note that these properties can only be assigned basic Python types:

- int, float, string
- array of ints or floats
- dictionary (only string keys are supported, values must be basic types too)

These properties are valid outside of Python. They can be animated by curves or used in driver paths.

Context

While it's useful to be able to access data directly by name or as a list, it's more common to operate on the user's selection. The context is always available from `bpy.context` and can be used to get the active object, scene, tool settings along with many other attributes.

Some common use cases are:

```
>>> bpy.context.object
>>> bpy.context.selected_objects
>>> bpy.context.visible_bones
```

```
>>> bpy.context.visible_bones
```

Note that the context is read-only, which means that these values cannot be modified directly. But they can be changed by running API functions or by using the data API.

So `bpy.context.active_object = obj` will raise an error. But `bpy.context.view_layer.objects.active = obj` works as expected.

The context attributes change depending on where they are accessed. The 3D Viewport has different context members than the Python Console, so take care when accessing context attributes that the user state is known.

See `bpy.context` API reference.

Operators (Tools)

Operators are tools generally accessed by the user from buttons, menu items or key shortcuts. From the user perspective they are a tool but Python can run these with its own settings through the `bpy.ops` module.

Examples:

```
>>> bpy.ops.mesh.flip_normals()
{'FINISHED'}
>>> bpy.ops.mesh.hide(unselected=False)
{'FINISHED'}
>>> bpy.ops.object.transform_apply()
{'FINISHED'}
```

> Tip
>
> The Operator Cheat Sheet gives a list of all operators and their default values in Python syntax, along with the generated docs. This is a good way to get an overview of all Blender's operators.

Operator Poll()

Many operators have a "poll" function which checks if the cursor is in a valid area or if the object is in the correct mode (Edit Mode, Weight Paint Mode etc). When an operator's poll function fails within Python, an exception is raised.

For example, calling `bpy.ops.view3d.render_border()` from the console raises the following error:

```
RuntimeError: Operator bpy.ops.view3d.render_border.poll() failed, context is incorrect
```

In this case the context must be the 3D Viewport with an active camera.

To avoid using try-except clauses wherever operators are called, you can call the operators own `poll()` function to check if it can run the operator in the current context.

```
if bpy.ops.view3d.render_border.poll():
    bpy.ops.view3d.render_border()
```

Integration

Python scripts can integrate with Blender in the following ways:

- By defining a render engine.
- By defining operators.
- By defining menus, headers and panels.
- By inserting new buttons into existing menus, headers and panels.

In Python, this is done by defining a class, which is a subclass of an existing type.

Example Operator

```
import bpy


def main(context):
    for ob in context.scene.objects:
        print(ob)


class SimpleOperator(bpy.types.Operator):
    """Tooltip"""
    bl_idname = "object.simple_operator"
    bl_label = "Simple Object Operator"

    @classmethod
    def poll(cls, context):
        return context.active_object is not None

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


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


# Register and add to the "object" menu (required to also use F3 search "Simple Object Ope
def register():
    bpy.utils.register_class(SimpleOperator)
    bpy.types.VIEW3D_MT_object.append(menu_func)


def unregister():
    bpy.utils.unregister_class(SimpleOperator)
    bpy.types.VIEW3D_MT_object.remove(menu_func)


if __name__ == "__main__":
    register()

    # test call
    bpy.ops.object.simple_operator()
```

Once this script runs, `SimpleOperator` is registered with Blender and can be called from Operator Search or added to the toolbar.

To run the script:

1. Start Blender and switch to the Scripting workspace.
2. Click the *New* button in the text editor to create a new text data-block.
3. Copy the code from above and paste it into the text editor.
4. Click on the *Run Script* button.
5. Move your cursor into the 3D Viewport, open the Operator Search menu, and type “Simple”.
6. Click on the “Simple Operator” item found in search.

6. Click on the "Simple Operator" item found in search.

See also

The class members with the `bl_` prefix are documented in the API reference `bpy.types.Operator`.

Note

The output from the `main` function is sent to the terminal; in order to see this, be sure to use the terminal.

Example Panel

Panels are registered as a class, like an operator. Notice the extra `bl_` variables used to set the context they display in.

```
import bpy


class HelloWorldPanel(bpy.types.Panel):
    """Creates a Panel in the Object properties window"""
    bl_label = "Hello World Panel"
    bl_idname = "OBJECT_PT_hello"
    bl_space_type = 'PROPERTIES'
    bl_region_type = 'WINDOW'
    bl_context = "object"

    def draw(self, context):
        layout = self.layout

        obj = context.object

        row = layout.row()
        row.label(text="Hello world!", icon='WORLD_DATA')

        row = layout.row()
        row.label(text="Active object is: " + obj.name)
        row = layout.row()
        row.prop(obj, "name")

        row = layout.row()
        row.operator("mesh.primitive_cube_add")


def register():
    bpy.utils.register_class(HelloWorldPanel)


def unregister():
    bpy.utils.unregister_class(HelloWorldPanel)


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

To run the script:

1. Start Blender and switch to the Scripting workspace.
2. Click the *New* button in the text editor to create a new text data-block.

2. Click the *New* button in the text editor to create a new text data-block.
3. Copy the code from above and paste it into the text editor.
4. Click on the *Run Script* button.

To view the results:

1. Select the default cube.
2. Click on the Object properties icon in the buttons panel (far right; appears as a tiny cube).
3. Scroll down to see a panel named "Hello World Panel".
4. Changing the object name also updates *Hello World Panel's* name: field.

Note the row distribution and the label and properties that are defined through the code.

See also

`bpy.types.Panel`

Types

Blender defines a number of Python types but also uses Python native types. Blender's Python API can be split up into three categories.

Native Types

In simple cases returning a number or a string as a custom type would be cumbersome, so these are accessed as normal Python types.

- Blender float, int, boolean -> float, int, boolean
- Blender enumerator -> string

```
>>> C.object.rotation_mode = 'AXIS_ANGLE'
```

- Blender enumerator (multiple) -> set of strings

```
# setting multiple camera overlay guides
bpy.context.scene.camera.data.show_guide = {'GOLDEN', 'CENTER'}

# passing as an operator argument for report types
self.report({'WARNING', 'INFO'}, "Some message!")
```

Internal Types

`bpy.types.bpy_struct` is used for Blender data-blocks and collections. Also for data that contains its own attributes: collections, meshes, bones, scenes, etc.

There are two main types that wrap Blender's data, one for data-blocks (known internally as `bpy_struct`), another for properties.

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

```
>>> C.scene.objects
bpy.data.scenes['Scene'].objects
```

Note that these types reference Blender's data so modifying them is visible immediately.

Mathutils Types

Accessible from `mathutils` are vectors, quaternions, Euler angles, matrix and color types. Some attributes such as `bpy.types.Object.location`, `bpy.types.PoseBone.rotation_euler` and

bpy.types.Object.location, bpy.types.PoseBone.location_euler and bpy.types.Scene.cursor_location can be accessed as special math types which can be used together and manipulated in various useful ways.

Example of a matrix, vector multiplication:

```
bpy.context.object.matrix_world @ bpy.context.object.data.verts[0].co
```

Note

mathutils types keep a reference to Blender's internal data so changes can be applied back.

Example:

```
# modifies the Z axis in place.
bpy.context.object.location.z += 2.0

# location variable holds a reference to the object too.
location = bpy.context.object.location
location *= 2.0

# Copying the value drops the reference so the value can be passed to
# functions and modified without unwanted side effects.
location = bpy.context.object.location.copy()
```

Animation

There are two ways to add keyframes through Python.

The first is through key properties directly, which is like inserting a keyframe from the button as a user. You can also manually create the curves and keyframe data, then set the path to the property. Here are examples of both methods. Both insert a keyframe on the active object's Z axis.

Simple example:

```
obj = bpy.context.object
obj.location[2] = 0.0
obj.keyframe_insert(data_path="location", frame=10.0, index=2)
obj.location[2] = 1.0
obj.keyframe_insert(data_path="location", frame=20.0, index=2)
```

Using low-level functions:

```
obj = bpy.context.object
obj.animation_data_create()
obj.animation_data.action = bpy.data.actions.new(name="MyAction")
fcu_z = obj.animation_data.action.fcurves.new(data_path="location", index=2)
fcu_z.keyframe_points.add(2)
fcu_z.keyframe_points[0].co = 10.0, 0.0
fcu_z.keyframe_points[1].co = 20.0, 1.0
```