Skip to content

Internal Data & Their Python Objects

The Python objects wrapping Blender internal data have some limitations and constraints, compared to ‘pure Python’ data. The most common things to keep in mind are documented here.

Life-Time of Python Objects Wrapping Blender Data

Typically, Python objects representing (wrapping) Blender data have a limited life-time. They are created on-demand, and deleted as soon as they are n used in Python anymore.

This means that storing python-only data in these objects should not be done for anything that requires some form of persistance.

There are some exceptions to this rule. For example, IDs do store their Python instance, once created, and re-use it instead of re-creating a new Python object every time they are accessed from Python. And modal operators will keep their instance as long as the operator is running. However, this is done for performances purpose and is considered an internal implementation detail. Relying on this behavior from Python code side for any purpose is not recommended.

Further more, Blender may free its internal data, in which case it will try to invalidate a known Python object wrapping it. But this is not always possible, which can lead to invalid memory access and is another good reason to never store these in Python code in any persistent way. See also the troubleshooting crashes documentation.

Data Names

Naming Limitations

A common mistake is to assume newly created data is given the requested name. This can cause bugs when you add data (normally imported) then reference it later by name:

```
bpy.data.meshes.new(name=meshid)

# normally some code, function calls...
bpy.data.meshes[meshid]
```

Or with name assignment:

```
obj.name = objname

# normally some code, function calls...
obj = bpy.data.meshes[objname]
```

Data names may not match the assigned values if they exceed the maximum length, are already used or an empty string.

It’s better practice not to reference objects by names at all, once created you can store the data in a list, dictionary, on a class, etc; there is rarely a reas to have to keep searching for the same data by name.

If you do need to use name references, it’s best to use a dictionary to maintain a mapping between the names of the imported assets and the newly creat data, this way you don’t run this risk of referencing existing data from the blend-file, or worse modifying it.

```
# typically declared in the main body of the function.
mesh_name_mapping = {}

mesh = bpy.data.meshes.new(name=meshid)
mesh_name_mapping[meshid] = mesh

# normally some code, or function calls...
# use own dictionary rather than bpy.data
mesh = mesh_name_mapping[meshid]
```

Library Collisions

Blender keeps data names unique (`bpy.types.ID.name`) so you can't name two objects, meshes, scenes, etc., the same by accident. However, when linking in library data from another blend-file naming collisions can occur, so it's best to avoid referencing data by name at all.

This can be tricky at times and not even Blender handles this correctly in some cases (when selecting the modifier object for e.g. you can't select betwee multiple objects with the same name), but it's still good to try avoiding these problems in this area. If you need to select between local and library data, there is a feature in `bpy.data` members to allow for this.

```
# typical name lookup, could be local or library.
obj = bpy.data.objects["my_obj"]

# library object name look up using a pair
# where the second argument is the library path matching bpy.types.Library.filepath
obj = bpy.data.objects["my_obj", "//my_lib.blend"]

# local object name look up using a pair
# where the second argument excludes library data from being returned.
obj = bpy.data.objects["my_obj", None]

# both the examples above also works for 'get'
obj = bpy.data.objects.get(("my_obj", None))
```

Stale Data

No updates after setting values

Sometimes you want to modify values from Python and immediately access the updated values, e.g: Once changing the objects `bpy.types.Object.location` you may want to access its transformation right after from `bpy.types.Object.matrix_world`, but this doesn't work as you might expect. There are similar issues with changes to the UI, that are covered in the next section.

Consider the calculations that might contribute to the object's final transformation, this includes:

- Animation function curves.
- Drivers and their Python expressions.
- Constraints
- Parent objects and all of their F-Curves, constraints, etc.

To avoid expensive recalculations every time a property is modified, Blender defers the evaluation until the results are needed. However, while the script runs you may want to access the updated values. In this case you need to call `bpy.types.ViewLayer.update` after modifying values, for example:

```
bpy.context.object.location = 1, 2, 3
bpy.context.view_layer.update()
```

Now all dependent data (child objects, modifiers, drivers, etc.) have been recalculated and are available to the script within the active view layer.

No updates after changing UI context

Similar to the previous issue, some changes to the UI may also not have an immediate effect. For example, setting `bpy.types.Window.workspace` doesn't seem to cause an observable effect in the immediately following code (`bpy.types.Window.workspace` is still the same), but the UI will in fact reflect the change. Some of the properties that behave that way are:

- `bpy.types.Window.workspace`
- `bpy.types.Window.screen`
- `bpy.types.Window.scene`
- `bpy.types.Area.type`
- `bpy.types.Area.uitype`

Such changes impact the UI, and with that the context (`bpy.context`) quite drastically. This can break Blender's context management. So Blender delays this change until after operators have run and just before the UI is redrawn, making sure that context can be changed safely.

If you rely on executing code with an updated context this can be worked around by executing the code in a delayed fashion as well. Possible options include:

- Modal Operator.
- `bpy.app.handlers`.
- `bpy.app.timer`.

It's also possible to depend on drawing callbacks although these should generally be avoided as failure to draw a hidden panel, region, cursor, etc. could cause your script to be unreliable

Can I redraw during script execution?

The official answer to this is no, or… *"You don't want to do that"*. To give some background on the topic:

While a script executes, Blender waits for it to finish and is effectively locked until it's done; while in this state Blender won't redraw or respond to user input. Normally this is not such a problem because scripts distributed with Blender tend not to run for an extended period of time, nevertheless scripts *ca* take a long time to complete and it would be nice to see progress in the viewport.

Tools that lock Blender in a loop redraw are highly discouraged since they conflict with Blender's ability to run multiple operators at once and update different parts of the interface as the tool runs.

So the solution here is to write a **modal** operator, which is an operator that defines a `modal()` function, See the modal operator template in the text editor. Modal operators execute on user input or setup their own timers to run frequently, they can handle the events or pass through to be handled by th keymap or other modal operators. Examples of modal operators are Transform, Painting, Fly Navigation and File Select.

Writing modal operators takes more effort than a simple `for` loop that contains draw calls but is more flexible and integrates better with Blender's design.

Ok, Ok! I still want to draw from Python

If you insist – yes it's possible, but scripts that use this hack will not be considered for inclusion in Blender and any issue with using it will not be consider a bug, there is also no guaranteed compatibility in future releases.

```
bpy.ops.wm.redraw_timer(type='DRAW_WIN_SWAP', iterations=1)
```