Skip to content

Troubleshooting Errors & Crashes

Strange Errors when Using the 'Threading' Module

Python threading with Blender only works properly when the threads finish up before the script does, for example by using `threading.join()`.

Here is an example of threading supported by Blender:

```
import threading
import time

def prod():
    print(threading.current_thread().name, "Starting")

    # do something vaguely useful
    import bpy
    from mathutils import Vector
    from random import random

    prod_vec = Vector((random() - 0.5, random() - 0.5, random() - 0.5))
    print("Prodding", prod_vec)
    bpy.data.objects["Cube"].location += prod_vec
    time.sleep(random() + 1.0)
    # finish

    print(threading.current_thread().name, "Exiting")

threads = [threading.Thread(name="Prod %d" % i, target=prod) for i in range(10)]


print("Starting threads...")

for t in threads:
    t.start()

print("Waiting for threads to finish...")

for t in threads:
    t.join()
```

This an example of a timer which runs many times a second and moves the default cube continuously while Blender runs **(Unsupported)**.

```
def func():
    print("Running...")
    import bpy
    bpy.data.objects['Cube'].location.x += 0.05

def my_timer():
    from threading import Timer
    t = Timer(0.1, my_timer)
    t.start()
    func()

my_timer()
```

```
my_timer()
```

Use cases like the one above which leave the thread running once the script finishes may seem to work for a while but end up causing random crashes or errors in Blender's own drawing code.

So far, no work has been done to make Blender's Python integration thread safe, so until it's properly supported, it's best not make use of this.

Note

Python threads only allow concurrency and won't speed up your scripts on multiprocessor systems, the `subprocess` and `multiprocess` modules can be used with Blender to make use of multiple CPUs too.

Help! My script crashes Blender

TL;DR Do not keep direct references to Blender data (of any kind) when modifying the container of that data, and/or when some undo/redo may happen (e.g. during modal operators execution…). Instead, use indices (or other data always stored by value in Python, like string keys…), that allow you to get access to the desired data.

Ideally it would be impossible to crash Blender from Python, however, there are some problems with the API where it can be made to crash. Strictly speaking this is a bug in the API but fixing it would mean adding memory verification on every access since most crashes are caused by the Python objects referencing Blender's memory directly, whenever the memory is freed or re-allocated, further Python access to it can crash the script. But fixing this would make the scripts run very slow, or writing a very different kind of API which doesn't reference the memory directly.

Here are some general hints to avoid running into these problems:

- Be aware of memory limits, especially when working with large lists since Blender can crash simply by running out of memory.
- Many hard to fix crashes end up being because of referencing freed data, when removing data be sure not to hold any references to it.
- Re-allocation can lead to the same issues (e.g. if you add a lot of items to some Collection, this can lead to re-allocating the underlying container's memory, invalidating all previous references to existing items).
- Modules or classes that remain active while Blender is used, should not hold references to data the user may remove, instead, fetch data from the context each time the script is activated.
- Crashes may not happen every time, they may happen more on some configurations or operating systems.
- Be careful with recursive patterns, those are very efficient at hiding the issues described here.
- See last subsection about Unfortunate Corner Cases for some known breaking exceptions.

Note

To find the line of your script that crashes you can use the `faulthandler` module. See the Faulthandler docs.

While the crash may be in Blender's C/C++ code, this can help a lot to track down the area of the script that causes the crash.

Note

Some container modifications are actually safe, because they will never re-allocate existing data (e.g. linked lists containers will never re-allocate existing items when adding or removing others).

But knowing which cases are safe and which aren't implies a deep understanding of Blender's internals. That's why, unless you are willing to dive into the RNA C implementation, it's simpler to always assume that data references will become invalid when modifying their containers, in any possible way.

Do not:

```
class TestItems(bpy.types.PropertyGroup):
    name: bpy.props.StringProperty()

bpy.utils.register_class(TestItems)
bpy.types.Scene.test_items = bpy.props.CollectionProperty(type=TestItems)

first_item = bpy.context.scene.test_items.add()
for i in range(100):
    bpy.context.scene.test_items.add()
```

```
    bpy.context.scene.test_items.add()

# This is likely to crash, as internal code may re-allocate
# the whole container (the collection) memory at some point.
first_item.name = "foobar"
```

Do:

```
class TestItems(bpy.types.PropertyGroup):
    name: bpy.props.StringProperty()

bpy.utils.register_class(TestItems)
bpy.types.Scene.test_items = bpy.props.CollectionProperty(type=TestItems)

first_item = bpy.context.scene.test_items.add()
for i in range(100):
    bpy.context.scene.test_items.add()

# This is safe, we are getting again desired data *after*
# all modifications to its container are done.
first_item = bpy.context.scene.test_items[0]
first_item.name = "foobar"
```

Undo/Redo

For safety, you should assume that undo and redo always invalidates all `bpy.types.ID` instances (Object, Scene, Mesh, Light, etc.), as well obviously as all of their sub-data.

This example shows how you can tell undo changes the memory locations:

```
>>> hash(bpy.context.object)
-9223372036849950810
>>> hash(bpy.context.object)
-9223372036849950810
```

Delete the active object, then undo:

```
>>> hash(bpy.context.object)
-9223372036849951740
```

As suggested above, simply not holding references to data when Blender is used interactively by the user is the only way to make sure that the script doesn't become unstable.

Note

Modern undo/redo system does not systematically invalidate all pointers anymore. Some data (in fact, most data, in typical cases), which were detected as unchanged for a particular history step, may remain unchanged and hence their pointers may remain valid.

Be aware that if you want to take advantage of this behavior for some reason, there is no guarantee of any kind that it will be safe and consistent. Use it at your own risk.

Modifying Blender Data & Undo

In general, when Blender data is modified, there should always be an undo step created for it. Otherwise, there will be issues, ranging from invalid/broken undo stack, to crashes on undo/redo.

This is especially true when modifying Blender data in operators.

Undo & Library Data

One of the advantages with Blender's library linking system that undo can skip checking changes in library data since it is assumed to be static. Tools in Blender are not allowed to modify library data. But Python does not enforce this restriction.

This can be useful in some cases, using a script to adjust material values for example. But it's also possible to use a script to make library data point to newly created local data, which is not supported since a call to undo will remove the local data but leave the library referencing it and likely crash.

So it's best to consider modifying library data an advanced usage of the API and only to use it when you know what you're doing.

Abusing RNA property callbacks

Python-defined RNA properties can have custom callbacks. Trying to perform complex operations from there, like calling an operator, may work, but is not officially recommended nor supported.

Main reason is that those callback should be very fast, but additionally, it may for example create issues with undo/redo system (most operators store an history step, and editing an RNA property does so as well), trigger infinite update loops, and so on.

Edit-Mode / Memory Access

Switching mode `bpy.ops.object.mode_set(mode='EDIT')` or `bpy.ops.object.mode_set(mode='OBJECT')` will re allocate objects data, any references to a meshes vertices/polygons/UVs, armatures bones, curves points, etc. cannot be accessed after switching mode.

Only the reference to the data itself can be re-accessed, the following example will crash.

```
mesh = bpy.context.active_object.data
polygons = mesh.polygons
bpy.ops.object.mode_set(mode='EDIT')
bpy.ops.object.mode_set(mode='OBJECT')

# this will crash
print(polygons)
```

So after switching mode you need to re-access any object data variables, the following example shows how to avoid the crash above.

```
mesh = bpy.context.active_object.data
polygons = mesh.polygons
bpy.ops.object.mode_set(mode='EDIT')
bpy.ops.object.mode_set(mode='OBJECT')

# polygons have been re-allocated
polygons = mesh.polygons
print(polygons)
```

These kinds of problems can happen for any functions which re-allocate the object data but are most common when switching mode.

Array Re-Allocation

When adding new points to a curve or vertices/edges/polygons to a mesh, internally the array which stores this data is re-allocated.

```
bpy.ops.curve.primitive_bezier_curve_add()
point = bpy.context.object.data.splines[0].bezier_points[0]
bpy.context.object.data.splines[0].bezier_points.add()

# this will crash!
point.co = 1.0, 2.0, 3.0
```

This can be avoided by re-assigning the point variables after adding the new one or by storing indices to the points rather than the points themselves.

The best way is to sidestep the problem altogether by adding all the points to the curve at once. This means you don't have to worry about array re-allocation and it's faster too since re-allocating the entire array for every added point is inefficient.

Removing Data

Any data that you remove shouldn't be modified or accessed afterwards, this includes: F-Curves, drivers, render layers, timeline markers, modifiers, constraints along with objects, scenes, collections, bones, etc.

The `remove()` API calls will invalidate the data they free to prevent common mistakes. The following example shows how this precaution works:

```
mesh = bpy.data.meshes.new(name="MyMesh")
# normally the script would use the mesh here...
bpy.data.meshes.remove(mesh)
print(mesh.name)  # <- give an exception rather than crashing:

# ReferenceError: StructRNA of type Mesh has been removed
```

But take care because this is limited to scripts accessing the variable which is removed, the next example will still crash:

```
mesh = bpy.data.meshes.new(name="MyMesh")
vertices = mesh.vertices
bpy.data.meshes.remove(mesh)
print(vertices)  # <- this may crash
```

Unfortunate Corner Cases

Besides all expected cases listed above, there are a few others that should not be an issue but, due to internal implementation details, currently are:

Collection Objects

Changing: `Object.hide_viewport`, `Object.hide_select` or `Object.hide_render` will trigger a rebuild of Collection cache thus breaking any current iteration over `Collection.all_objects`.

Do not:

```
# `all_objects` is an iterator. Using it directly while performing operations on its mem
# the memory accessed by the `all_objects` iterator will lead to invalid memory accesses
for object in bpy.data.collections["Collection"].all_objects:
    object.hide_viewport = True
```

Do:

```
# `all_objects[:]` is an independent list generated from the iterator. As long as no obj
# its content will remain valid even if the data accessed by the `all_objects` iterator
for object in bpy.data.collections["Collection"].all_objects[:]:
    object.hide_viewport = True
```

Data-Blocks Renaming During Iteration

Data-blocks accessed from `bpy.data` are sorted when their name is set. Any loop that iterates of a data such as `bpy.data.objects` for example, and sets the objects `name` must get all items from the iterator first (typically by converting to a list or tuple) to avoid missing some objects and iterating over others multiple times.

sys.exit

sys.exit

Some Python modules will call `sys.exit()` themselves when an error occurs, while not common behavior this is something to watch out for becaus it may seem as if Blender is crashing since `sys.exit()` will close Blender immediately.

For example, the `argparse` module will print an error and exit if the arguments are invalid.

An dirty way of troubleshooting this is to set `sys.exit = None` and see what line of Python code is quitting, you could of course replace `sys.exit` with your own function but manipulating Python in this way is bad practice.

Copyright © Blender Authors
Made with Furo