Skip to content

API Overview

The purpose of this document is to explain how Python and Blender fit together, covering some of the functionality that may not be obvious from reading the API references and example scripts.

Python in Blender

Blender has an embedded Python interpreter which is loaded when Blender is started and stays active while Blender is running. This interpreter runs scri to draw the user interface and is used for some of Blender's internal tools as well.

Blender's embedded interpreter provides a typical Python environment, so code from tutorials on how to write Python scripts can also be run with Blender's interpreter. Blender provides its Python modules, such as `bpy` and `mathutils`, to the embedded interpreter so they can be imported int a script and give access to Blender's data, classes, and functions. Scripts that deal with Blender data will need to import the modules to work.

Here is a simple example which moves a vertex attached to an object named "Cube":

```
import bpy
bpy.data.objects["Cube"].data.vertices[0].co.x += 1.0
```

This modifies Blender's internal data directly. When you run this in the interactive console you will see the 3D Viewport update.

The Default Environment

When developing your own scripts it may help to understand how Blender sets up its Python environment. Many Python scripts come bundled with Blender and can be used as a reference because they use the same API that script authors write tools in. Typical usage for scripts include: user interface, import/export, scene manipulation, automation, defining your own tool set and customization.

On startup Blender scans the `scripts/startup/` directory for Python modules and imports them. The exact location of this directory depends o your installation. See the directory layout docs.

Script Loading

This may seem obvious, but it is important to note the difference between executing a script directly and importing a script as a module.

Extending Blender by executing a script directly means the classes that the script defines remain available inside Blender after the script finishes execution Using scripts this way makes future access to their classes (to unregister them for example) more difficult compared to importing the scripts as modules. When a script is imported as a module, its class instances will remain inside the module and can be accessed later on by importing that module again.

For this reason it is preferable to avoid directly executing scripts that extend Blender by registering classes.

Here are some ways to run scripts directly in Blender:

- Loaded in the text editor and press *Run Script*.
- Typed or pasted into the interactive console.
- Execute a Python file from the command line with Blender, e.g:

```
blender --python /home/me/my_script.py
```

To run as modules:

- The obvious way, `import some_module` command from the text editor or interactive console.
- Open as a text data-block and check the *Register* option, this will load with the blend-file.
- Copy into one of the directories `scripts/startup`, where they will be automatically imported on startup.
- Define as an add-on, enabling the add-on will load it as a Python module.

Add-ons

Some of Blender's functionality is best kept optional, alongside scripts loaded at startup there are add-ons which are kept in their own directory `scripts/addons`, They are only loaded on startup if selected from the user preferences.

The only difference between add-ons and built-in Python modules is that add-ons must contain a `bl_info` variable which Blender uses to read metadata such as name, author, category and project link. The User Preferences add-on listing uses `bl_info` to display information about each add-on. See Add-ons for details on the `bl_info` dictionary.

Integration through Classes

Running Python scripts in the text editor is useful for testing but you'll want to extend Blender to make tools accessible like other built-in functionality.

The Blender Python API allows integration for:

- `bpy.types.Panel`
- `bpy.types.Menu`
- `bpy.types.Operator`
- `bpy.types.PropertyGroup`
- `bpy.types.KeyingSet`
- `bpy.types.RenderEngine`

This is intentionally limited. Currently, for more advanced features such as mesh modifiers, object types, or shader nodes, C/C++ must be used.

For Python integration Blender defines methods which are common to all types. This works by creating a Python subclass of a Blender class which contains variables and functions specified by the parent class which are predefined to interface with Blender.

For example:

```
import bpy
class SimpleOperator(bpy.types.Operator):
    bl_idname = "object.simple_operator"
    bl_label = "Tool Name"

    def execute(self, context):
        print("Hello World")
        return {'FINISHED'}

bpy.utils.register_class(SimpleOperator)
```

First note that it defines a subclass as a member of `bpy.types`, this is common for all classes which can be integrated with Blender and is used to distinguish an Operator from a Panel when registering.

Both class properties start with a `bl_` prefix. This is a convention used to distinguish Blender properties from those you add yourself. Next see the execute function, which takes an instance of the operator and the current context. A common prefix is not used for functions. Lastly the register function i called, this takes the class and loads it into Blender. See Class Registration.

Regarding inheritance, Blender doesn't impose restrictions on the kinds of class inheritance used, the registration checks will use attributes and functions defined in parent classes.

Class mix-in example:

```
import bpy
class BaseOperator:
    def execute(self, context):
        print("Hello World BaseClass")
        return {'FINISHED'}

class SimpleOperator(bpy.types.Operator, BaseOperator):
    bl_idname = "object.simple_operator"
    bl_label = "Tool Name"
```

```
bpy.utils.register_class(SimpleOperator)
```

Note

Modal operators are an exception, keeping their instance variable as Blender runs, see modal operator template.

So once the class is registered with Blender, instancing the class and calling the functions is left up to Blender. In fact you cannot instance these classes from the script as you would expect with most Python API's. To run operators you can call them through the operator API, e.g:

```
import bpy
bpy.ops.object.simple_operator()
```

User interface classes are given a context in which to draw, buttons, window, file header, toolbar, etc., then they are drawn when that area is displayed s they are never called by Python scripts directly.

Construction & Destruction

In the examples above, the classes don't define an `__init__(self)` function. In general, defining custom constructors or destructors should not b needed, and is not recommended.

The lifetime of class instances is usually very short (also see the dedicated section), a panel for example will have a new instance for every redraw. Some other types, like `bpy.types.Operator`, have an even more complex internal handling, which can lead to several instantiations for a single operat execution.

There are a few cases where defining `__init__()` does make sense, e.g. when sub-classing a `bpy.types.RenderEngine`. When doing s the parent matching function must always be called, otherwise Blender's internal initialization won't happen properly:

```
import bpy
class AwesomeRaytracer(bpy.types.RenderEngine):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.my_var = 42
        ...
```

Warning

The Blender-defined parent constructor must be called before any data access to the object, including from other potential parent types `__init__()` functions.

Warning

Calling the parent's `__init__()` function is a hard requirement since Blender 4.4. The 'generic' signature is the recommended one here, as Blender internal BPY code is typically the only caller of these functions. The actual arguments passed to the constructor are fully internal data, and may change depending on the implementation.

Unfortunately, the error message, generated in case the expected constructor is not called, can be fairly cryptic and unhelping. Generally they should be about failure to create a (python) object:

MemoryError: couldn't create bpy_struct object_

With Operators, it might be something like that:

RuntimeError: could not create instance of <OPERATOR_OT_identifier> to call callback function execute

Note

In case you are using complex/multi-inheritance, `super()` may not work (as the Blender-defined parent may not be the first type in the MRO). It is best then to first explicitly invoke the Blender-defined parent class constructor, before any other. For example:

```
import bpy
class FancyRaytracer(AwesomeRaytracer, bpy.types.RenderEngine):
   def __init__(self, *args, **kwargs):
      bpy.types.RenderEngine.__init__(self, *args, **kwargs)
      AwesomeRaytracer.__init__(self, *args, **kwargs)
      self.my_var = 42
      ...
```

Note

Defining a custom `__new__()` function is strongly discouraged, not tested, and not considered as supported currently. Doing so presents a very hig risk of crashes or otherwise corruption of Blender internal data. But if defined, it must take the same two generic positional and keyword arguments, an call the parent's `__new__()` with them if actually creating a new object.

Note

Due to internal CPython implementation details, C++-defined Blender types do not define or use a `__del__()` (aka `tp_finalize()`) destructor currently. As this function does not exist if not explicitly defined, that means that calling `super().__del__()` in the `__del__()` function of a sub-class will fail with the following error: `AttributeError: 'super' object has no attribute '__del__'` If a call to the MRO 'parent' destructor is needed for some reason, the caller code must ensure that the destructor does exist, e.g. using something like that: `getattr(super(), "__del__", lambda self: None)(self)`

Registration

Module Registration

Blender modules loaded at startup require `register()` and `unregister()` functions. These are the *only* functions that Blender calls from you code, which is otherwise a regular Python module.

A simple Blender Python module can look like this:

```
import bpy

class SimpleOperator(bpy.types.Operator):
    """ See example above """

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

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

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

These functions usually appear at the bottom of the script containing class registration sometimes adding menu items. You can also use them for internal purposes setting up data for your own tools but take care since register won't re-run when a new blend-file is loaded.

The register/unregister calls are used so it's possible to toggle add-ons and reload scripts while Blender runs. If the register calls were placed in the body of the script, registration would be called on import, meaning there would be no distinction between importing a module or loading its classes into Blende This becomes problematic when a script imports classes from another module making it difficult to manage which classes are being loaded and when.

The last two lines are only for testing:

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

This allows the script to be run directly in the text editor to test changes. This `register()` call won't run when the script is imported as a module since `__main__` is reserved for direct execution.

Class Registration

Registering a class with Blender results in the class definition being loaded into Blender, where it becomes available alongside existing functionality. Once this class is loaded you can access it from `bpy.types`, using the `bl_idname` rather than the classes original name.

> Note
>
> There are some exceptions to this for class names which aren't guarantee to be unique. In this case use: `bpy.types.Struct.bl_rna_get_subclass_py()`.

When loading a class, Blender performs sanity checks making sure all required properties and functions are found, that properties have the correct type, and that functions have the right number of arguments.

Mostly you will not need concern yourself with this but if there is a problem with the class definition it will be raised on registering:

Using the function arguments `def execute(self, context, spam)`, will raise an exception:

```
ValueError: expected Operator, SimpleOperator class "execute" function to have 2 args, foun
3
```

Using `bl_idname = 1` will raise:

```
TypeError: validating class error: Operator.bl_idname expected a string type, not int
```

Inter-Class Dependencies

When customizing Blender you may want to group your own settings together, after all, they will likely have to co-exist with other scripts. To group these properties classes need to be defined, for groups within groups or collections within groups you can't avoid having to deal with the order of registration/unregistration.

Custom properties groups are themselves classes which need to be registered.

For example, if you want to store material settings for a custom engine:

```
# Create new property
# bpy.data.materials[0].my_custom_props.my_float
import bpy

class MyMaterialProps(bpy.types.PropertyGroup):
    my_float: bpy.props.FloatProperty()

def register():
    bpy.utils.register_class(MyMaterialProps)
    bpy.types.Material.my_custom_props: bpy.props.PointerProperty(type=MyMaterialProps)

def unregister():
    del bpy.types.Material.my_custom_props
    bpy.utils.unregister_class(MyMaterialProps)

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

> Note

The class **must be** registered before being used in a property, failing to do so will raise an error:

```
ValueError: bpy_struct "Material" registration error: my_custom_props could not register
```

```
# Create new property group with a sub property
# bpy.data.materials[0].my_custom_props.sub_group.my_float
import bpy

class MyMaterialSubProps(bpy.types.PropertyGroup):
    my_float: bpy.props.FloatProperty()

class MyMaterialGroupProps(bpy.types.PropertyGroup):
    sub_group: bpy.props.PointerProperty(type=MyMaterialSubProps)

def register():
    bpy.utils.register_class(MyMaterialSubProps)
    bpy.utils.register_class(MyMaterialGroupProps)
    bpy.types.Material.my_custom_props: bpy.props.PointerProperty(type=MyMaterialGroupProp

def unregister():
    del bpy.types.Material.my_custom_props
    bpy.utils.unregister_class(MyMaterialGroupProps)
    bpy.utils.unregister_class(MyMaterialSubProps)

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

Important

The lower most class needs to be registered first and that `unregister()` is a mirror of `register()`.

Manipulating Classes

Properties can be added and removed as Blender runs, normally done on register or unregister but for some special cases it may be useful to modify type as the script runs.

For example:

```
# add a new property to an existing type
bpy.types.Object.my_float: bpy.props.FloatProperty()
# remove
del bpy.types.Object.my_float
```

This works just as well for `PropertyGroup` subclasses you define yourself.

```
class MyPropGroup(bpy.types.PropertyGroup):
    pass
MyPropGroup.my_float: bpy.props.FloatProperty()
```

This is equivalent to:

```
class MyPropGroup(bpy.types.PropertyGroup):
    my_float: bpy.props.FloatProperty()
```

Dynamic Class Definition (Advanced)

In some cases the specifier for data may not be in Blender, for example a external render engines shader definitions, and it may be useful to define them a types and remove them on the fly.

```
for i in range(10):
    idname = "object.operator_{:d}".format(i)

    def func(self, context):
        print("Hello World", self.bl_idname)
        return {'FINISHED'}

    op_class = type(
        "DynOp{:d}".format(i),
        (bpy.types.Operator, ),
        {"bl_idname": idname, "bl_label": "Test", "execute": func},
    )
    bpy.utils.register_class(op_class)
```

Note

`type()` is called to define the class. This is an alternative syntax for class creation in Python, better suited to constructing classes dynamically.

To call the operators from the previous example:

```
>>> bpy.ops.object.operator_1()
Hello World OBJECT_OT_operator_1
{'FINISHED'}
```

```
>>> bpy.ops.object.operator_2()
Hello World OBJECT_OT_operator_2
{'FINISHED'}
```