Skip to content

API Reference Usage

Blender has many interlinking data types which have an auto-generated reference API which often has the information you need to write a script, but can be difficult to use. This document is designed to help you understand how to use the reference API.

Reference API Scope

The reference API covers `bpy.types`, which stores types accessed via `bpy.context` – *the user context* or `bpy.data` – *blend-file data.*

Other modules such as `bmesh` and `aud` are not using Blender's data API so this document doesn't apply to those modules.

Data Access

The most common case for using the reference API is to find out how to access data in the blend-file. Before going any further it's best to be aware of II data-blocks in Blender since you will often find properties relative to them.

ID Data

ID data-blocks are used in Blender as top-level data containers. From the user interface this isn't so obvious, but when developing you need to know about ID data-blocks. ID data types include Scene, Group, Object, Mesh, Workspace, World, Armature, Image and Texture. For a full list see the subclasses of `bpy.types.ID`.

Here are some characteristics ID data-blocks share:

- IDs are blend-file data, so loading a new blend-file reloads an entire new set of data-blocks.
- IDs can be accessed in Python from `bpy.data.*`.
- Each data-block has a unique `.name` attribute, displayed in the interface.
- Animation data is stored in IDs `.animation_data`.
- IDs are the only data types that can be linked between blend-files.
- IDs can be added/copied and removed via Python.
- IDs have their own garbage-collection system which frees unused IDs when saving.
- When a data-block has a reference to some external data, this is typically an ID data-block.

Simple Data Access

In this simple case a Python script is used to adjust the object's location. Start by collecting the information where the data is located.

First find this setting in the interface `Properties editor -> Object -> Transform -> Location`. From the button context menu select *Online Python Reference*, this will link you to: `bpy.types.Object.location`. Being an API reference, this link often gives little more information than the tooltip, though some of the pages include examples (normally at the top of the page). But you now know that you have to use `.location` and that it's an array of three floats.

So the next step is to find out where to access objects, go down to the bottom of the page to the references section, for objects there are many referenc but one of the most common places to access objects is via the context. It's easy to be overwhelmed at this point since there `Object` get referenced i so many places: modifiers, functions, textures and constraints. But if you want to access any data the user has selected you typically only need to check t `bpy.context` references.

Even then, in this case there are quite a few though if you read over these you'll notice that most are mode specific. If you happen to be writing a tool tha only runs in Weight Paint Mode, then using `weight_paint_object` would be appropriate. However, to access an item the user last selected, look for the `active` members, Having access to a single active member the user selects is a convention in Blender: e.g. `active_bone`, `active_pose_bone`, `active_node`, etc. and in this case you can use `active_object`.

So now you have enough information to find the location of the active object.

```
bpy.context.active_object.location
```

You can type this into the Python console to see the result. The other common place to access objects in the reference is bpy.types.BlendData.objects.

Note

This is **not** listed as bpy.data.objects, this is because bpy.data is an instance of the bpy.types.BlendData class, so the documentation points there.

With bpy.data.objects, this is a collection of objects so you need to access one of its members:

```
bpy.data.objects["Cube"].location
```

Nested Properties

The previous example is quite straightforward because location is a property of Object which can be accessed from the context directly.

Here are some more complex examples:

```
# Access the number of samples for the Cycles render engine.
bpy.context.scene.cycles.samples

# Access to the current weight paint brush size.
bpy.context.tool_settings.weight_paint.brush.size

# Check if the window is full-screen.
bpy.context.window.screen.show_fullscreen
```

As you can see there are times when you want to access data which is nested in a way that causes you to go through a few indirections. The properties a arranged to match how data is stored internally (in Blender's C code) which is often logical but not always quite what you would expect from using Blender. So this takes some time to learn, it helps you understand how data fits together in Blender which is important to know when writing scripts.

When starting out scripting you will often run into the problem where you're not sure how to access the data you want. There are a few ways to do this:

- Use the Python console's auto-complete to inspect properties. *This can be hit-and-miss but has the advantage that you can easily see the value of properties and assign them to interactively see the results.*
- Copy the data path from the user interface. *Explained further in* Copy Data Path.
- Using the documentation to follow references. *Explained further in* Indirect Data Access.

Copy Data Path

Blender can compute the Python string to a property which is shown in the tooltip, on the line below Python: .... This saves having to open the API references to find where data is accessed from. In the context menu is a copy data-path tool which gives the path from an bpy.types.ID data block, to its property.

To see how this works you'll get the path to the Subdivision Surface modifiers *Levels* setting. Start with the default scene and select the Modifiers tab, then add a Subdivision Surface modifier to the cube. Now hover your mouse over the button labeled *Levels Viewport*, The tooltip includes bpy.types.SubsurfModifier.levels but you want the path from the object to this property.

Note that the text copied won't include the bpy.data.collection["name"]. component since its assumed that you won't be doing collection look-ups on every access and typically you'll want to use the context rather than access each bpy.types.ID instance by name.

Type in the ID path into a Python console bpy.context.active_object. Include the trailing dot and don't execute the code, yet.

Now in the button's context menu select *Copy Data Path*, then paste the result into the console:

```
bpy.context.active_object.modifiers["Subdivision"].levels
```

Press Return and you'll get the current value of 1. Now try changing the value to 2:

```
bpy.context.active_object.modifiers["Subdivision"].levels = 2
```

You can see the value update in the Subdivision Surface modifier's UI as well as the cube.

Indirect Data Access

This more advanced example shows the steps to access the active sculpt brushes texture. For example, if you want to access the texture of a brush via Python to adjust its `contrast`.

1. Start in the default scene and enable Sculpt Mode from the 3D Viewport header.
2. From the Sidebar expand the Brush Settings panel's *Texture* subpanel and add a new texture. *Notice the texture data-block menu itself doesn't have very useful links (you can check the tooltips).*
3. The contrast setting isn't exposed in the Sidebar, so view the texture in the Properties Editor.
4. Open the context menu of the contrast field and select *Online Python Reference*. This takes you to `bpy.types.Texture.contrast`. No you can see that `contrast` is a property of texture.
5. To find out how to access the texture from the brush check on the references at the bottom of the page. Sometimes there are many references, and it may take some guesswork to find the right one, but in this case it's `tool_settings.sculpt.brush.texture`.
6. Now you know that the texture can be accessed from `bpy.data.brushes["BrushName"].texture` but normally you *won't* want to access the brush by name, instead you want to access the active brush. So the next step is to check on where brushes are accessed from via the references.

Now you can use the Python console to form the nested properties needed to access brush textures contrast: Context ‣ Tool Settings ‣ Sculpt ‣ Brush ‣ Texture ‣ Contrast.

Since the attribute for each is given along the way you can compose the data path in the Python console:

```
bpy.context.tool_settings.sculpt.brush.texture.contrast
```

Or access the brush directly:

```
bpy.data.textures["Texture"].contrast
```

If you are writing a user tool normally you want to use the `bpy.context` since the user normally expects the tool to operate on what they have selected. For automation you are more likely to use `bpy.data` since you want to be able to access specific data and manipulate it, no matter what th user currently has the view set at.

Operators

Most hotkeys and buttons in Blender call an operator which is also exposed to Python via `bpy.ops`.

To see the Python equivalent hover your mouse over the button and see the tooltip, e.g `Python: bpy.ops.render.render()`, If there is r tooltip or the `Python:` line is missing then this button is not using an operator and can't be accessed from Python.

If you want to use this in a script you can press `Ctrl - C` while your mouse is over the button to copy it to the clipboard. You can also use button's context menu and view the *Online Python Reference*, this mainly shows arguments and their defaults, however, operators written in Python show their f and line number which may be useful if you are interested to check on the source code.

Note

Not all operators can be called usefully from Python, for more on this see using operators.

Info Editor

Blender records operators you run and displays them in the Info editor. Select the Scripting workspace that comes default with Blender to see its output. You can perform some actions and see them show up – delete a vertex for example.

Each entry can be selected, then copied `Ctrl - C`, usually to paste in the text editor or Python console.

Note

Not all operators get registered for display, zooming the view for example isn't so useful to repeat so it's excluded from the output.

To display *every* operator that runs see Show All Operators.