Table of Contents

Index – E

- eyedropper_driver() (in module bpy.ops.ui)
- eyedropper_grease_pencil_color() (in module bpy.ops.ui)
- eyedropper_id() (in module bpy.ops.ui)

Skip to content

Blender 4.4 Python API Documentation

Welcome to the Python API documentation for Blender, the free and open source 3D creation suite.

This site can be used offline: Download the full documentation (zipped HTML files)

Documentation

- Quickstart: New to Blender or scripting and want to get your feet wet?
- API Overview: A more complete explanation of Python integration.
- API Reference Usage: Examples of how to use the API reference docs.
- Best Practice: Conventions to follow for writing good scripts.
- Tips and Tricks: Hints to help you while writing scripts for Blender.
- Gotchas: Some of the problems you may encounter when writing scripts.
- Advanced: Topics which may not be required for typical usage.
- Change Log: List of changes since last Blender release

APPLICATION MODULES

Context Access (bpy.context)

Data Access (bpy.data)

Message Bus (bpy.msgbus)

Operators (bpy.ops)

Types (bpy.types)

Utilities (bpy.utils)

Path Utilities (bpy.path)

Application Data (bpy.app)

Property Definitions (bpy.props)

STANDALONE MODULES

Audio System (aud)

OpenGL Wrapper (bgl)

Additional Math Functions (bl_math)

Font Drawing (blf)

BMesh Module (bmesh)

Extra Utilities (bpy_extras)

Freestyle Module (freestyle)

GPU Module (gpu)

GPU Utilities (gpu_extras)

ID Property Access (idprop.types)

Image Buffer (imbuf)

Math Types & Utilities (mathutils)

Indices

- Index
- Module Index

N
Quickst

Report issue on this page

Advanced

This chapter covers advanced use (topics which may not be required for typical usage).

Skip to content

Blender as a Python Module

Blender supports being built as a Python module, allowing `import bpy` to be added to any Python script, providing access to Blender's features.

Note

Blender as a Python Module isn't provided on Blender's official download page.

- A pre-compiled `bpy` module is available via PIP.
- Or you may compile this yourself using the build instructions.

Use Cases

Python developers may wish to integrate Blender scripts which don't center around Blender.

Possible uses include:

- Visualizing data by rendering images and animations.
- Image processing using Blender's compositor.
- Video editing (using Blender's sequencer).
- 3D file conversion.
- Development, accessing `bpy` from Python IDE's and debugging tools for example.
- Automation.

Usage

For the most part using Blender as a Python module is equivalent to running a script in background-mode (passing the command-line arguments `--background` or `-b`), however there are some differences to be aware of.

Blender's Executable Access

The attribute `bpy.app.binary_path` defaults to an empty string.

If you wish to point this to the location of a known executable you may set the value.

This example searches for the binary, setting it when found:

```
import bpy
import shutil

blender_bin = shutil.which("blender")
if blender_bin:
   print("Found:", blender_bin)
   bpy.app.binary_path = blender_bin
else:
   print("Unable to find blender!")
```

Blender's Internal Modules

There are many modules included with Blender such as `gpu` and `mathuils`. It's important that these are imported after `bpy` or they will n be found.

Command Line Arguments Unsupported

Functionality controlled by command line arguments (shown by calling `blender --help` aren't accessible).

Typically this isn't such a limitation although there are some command line arguments that don't have equivalents in Blender's Python API (`--threads` and `--log` for example).

Note

Access to these settings may be added in the future as needed.

Resource Sharing (GPU)

It's possible other Python modules make use of the GPU in a way that prevents Blender/Cycles from accessing the GPU.

Signal Handlers

Blender's typical signal handlers are not initialized, so there is no special handling for `Control-C` to cancel a render and a crash log is not written in the event of a crash.

Startup and Preferences

When the `bpy` module loads it contains the default startup scene (instead of an "empty" blend-file as you might expect), so there is a default cub camera and light.

If you wish to start from an empty file use: `bpy.ops.wm.read_factory_settings(use_empty=True)`.

The users startup and preferences are ignored to prevent your local configuration from impacting scripts behavior. The Python module behaves as `--factory-startup` was passed as a command line argument.

The users preferences and startup can be loaded using operators:

```
import bpy

bpy.ops.wm.read_userpref()
bpy.ops.wm.read_homefile()
```

Limitations

Most constraints of Blender as an application still apply:

Reloading Unsupported

Reloading the `bpy` module via `importlib.reload` will raise an exception instead of reloading and resetting the module.

Instead, the operator `bpy.ops.wm.read_factory_settings()` can be used to reset the internal state.

Single Blend File Restriction

Only a single `.blend` file can be edited at a time.

Hint

As with the application it's possible to start multiple instances, each with their own `bpy` and therefor Blender state. Python provides the `multiprocessing` module to make communicating with sub-processes more convenient.

In some cases the library API may be an alternative to starting separate processes, although this API operates on reading and writing ID data-blocks and isn't a complete substitute for loading `.blend` files, see:

- `bpy.types.BlendDataLibraries.load()`
- `bpy.types.BlendDataLibraries.write()`
- `bpy.types.BlendData.temp_data()` supports a temporary data-context to avoid manipulating the current `.blend` file.

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 ID 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 reference 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.

Best Practice

When writing your own scripts Python is great for new developers to pick up and become productive, but you can also pick up bad practices or at least write scripts that are not easy for others to understand. For your own work this is of course fine, but if you want to collaborate with others or have your work included with Blender there are practices we encourage.

Style Conventions

For Blender Python development we have chosen to follow Python suggested style guide to avoid mixing styles among our own scripts and make it easie to use Python scripts from other projects. Using our style guide for your own scripts makes it easier if you eventually want to contribute them to Blender.

This style guide is known as pep8 and here is a brief listing of pep8 criteria:

- Camel caps for class names: MyClass
- All lower case underscore separated module names: my_module
- Indentation of 4 spaces (no tabs)
- Spaces around operators: `1 + 1`, not `1+1`
- Only use explicit imports (no wildcard importing `*`)
- Don't use multiple statements on a single line: `if val: body`, separate onto two lines instead.

As well as pep8 we have additional conventions used for Blender Python scripts:

- Use single quotes for enums, and double quotes for strings.

  Both are of course strings, but in our internal API enums are unique items from a limited set, e.g:

  ```
  bpy.context.scene.render.image_settings.file_format = 'PNG'
  bpy.context.scene.render.filepath = "//render_out"
  ```

- pep8 also defines that lines should not exceed 79 characters, we have decided that this is too restrictive so it is optional per script.

User Interface Layout

Some notes to keep in mind when writing UI layouts:

UI code is quite simple. Layout declarations are there to easily create a decent layout. The general rule here is: If you need more code for the layout declaration, than for the actual properties, then you are doing it wrong.

Example layouts:

`layout()`

The basic layout is a simple top-to-bottom layout.

```
layout.prop()
layout.prop()
```

`layout.row()`

Use `row()`, when you want more than one property in a single line.

```
row = layout.row()
row.prop()
row.prop()
```

`layout.column()`

Use `column()`, when you want your properties in a column.

```
col = layout.column()
col.prop()
col.prop()
```

`layout.split()`

This can be used to create more complex layouts. For example, you can split the layout and create two `column()` layouts next to each other. Do not use split, when you simply want two properties in a row. Use `row()` instead.

```
split = layout.split()

col = split.column()
col.prop()
col.prop()

col = split.column()
col.prop()
col.prop()
```

Declaration names:

Try to only use these variable names for layout declarations:

row:
for a `row()` layout

col:
for a `column()` layout

split:
for a `split()` layout

flow:
for a `column_flow()` layout

sub:
for a sub layout (a column inside a column for example)

Script Efficiency

List Manipulation (General Python Tips)

Searching for List Items

In Python there are some handy list functions that save you having to search through the list. Even though you are not looping on the list data **Python is**, s
you need to be aware of functions that will slow down your script by searching the whole list.

```
my_list.count(list_item)
my_list.index(list_item)
my_list.remove(list_item)
if list_item in my_list: ...
```

Modifying Lists

In Python you can add and remove from a list, this is slower when the list length is modified, especially at the start of the list, since all the data after the index of modification needs to be moved up or down one place.

The fastest way to add onto the end of the list is to use `my_list.append(list_item)` or `my_list.extend(some_list)` and to remove an item is `my_list.pop()` or `del my_list[-1]`.

To use an index you can use `my_list.insert(index, list_item)` or `list.pop(index)` for list removal, but these are slower.

Sometimes it's faster (but less memory efficient) to just rebuild the list. For example if you want to remove all triangular polygons in a list. Rather than:

```
polygons = mesh.polygons[:]  # make a list copy of the meshes polygons
p_idx = len(polygons)      # Loop backwards
while p_idx:            # while the value is not 0
    p_idx -= 1

    if len(polygons[p_idx].vertices) == 3:
        polygons.pop(p_idx)  # remove the triangle
```

It's faster to build a new list with list comprehension:

```
polygons = [p for p in mesh.polygons if len(p.vertices) != 3]
```

Adding List Items

If you have a list that you want to add onto another list, rather than:

```
for l in some_list:
    my_list.append(l)
```

Use:

```
my_list.extend([a, b, c...])
```

Note that insert can be used when needed, but it is slower than append especially when inserting at the start of a long list. This example shows a very suboptimal way of making a reversed list:

```
reverse_list = []
for list_item in some_list:
    reverse_list.insert(0, list_item)
```

Python provides more convenient ways to reverse a list using the slice method, but you may want to time this before relying on it too much:

```
some_reversed_list = some_list[::-1]
```

Removing List Items

Use `my_list.pop(index)` rather than `my_list.remove(list_item)`. This requires you to have the index of the list item but is faste since `remove()` will search the list. Here is an example of how to remove items in one loop, removing the last items first, which is faster (as explainec above):

```
list_index = len(my_list)

while list_index:
    list_index -= 1
    if my_list[list_index].some_test_attribute == 1:
        my_list.pop(list_index)
```

This example shows a fast way of removing items, for use in cases where you can alter the list order without breaking the script's functionality. This work by swapping two list items, so the item you remove is always last:

```
pop_index = 5

# swap so the pop_index is last.
```

```
my_list[-1], my_list[pop_index] = my_list[pop_index], my_list[-1]

# remove last item (pop_index)
my_list.pop()
```

When removing many items in a large list this can provide a good speed-up.

Avoid Copying Lists

When passing a list or dictionary to a function, it is faster to have the function modify the list rather than returning a new list so Python doesn't have to duplicate the list in memory.

Functions that modify a list in-place are more efficient than functions that create new lists. This is generally slower so only use for functions when it makes sense not to modify the list in place:

```
>>> my_list = some_list_func(my_list)
```

This is generally faster since there is no re-assignment and no list duplication:

```
>>> some_list_func(vec)
```

Also note that, passing a sliced list makes a copy of the list in Python memory:

```
>>> foobar(my_list[:])
```

If my_list was a large array containing 10,000's of items, a copy could use a lot of extra memory.

Writing Strings to a File (Python General)

Here are three ways of joining multiple strings into one string for writing. This also applies to any area of your code that involves a lot of string joining:

String concatenation
: This is the slowest option, do **not** use this if you can avoid it, especially when writing data in a loop.

```
>>> file.write(str1 + " " + str2 + " " + str3 + "\n")
```

String formatting
: Use this when you are writing string data from floats and ints.

```
>>> file.write("%s %s %s\n" % (str1, str2, str3))
```

String joining
: Use this to join a list of strings (the list may be temporary). In the following example, the strings are joined with a space " " in between, other examples are "" or ", ".

```
>>> file.write(" ".join((str1, str2, str3, "\n")))
```

Join is fastest on many strings, string formatting is quite fast too (better for converting data types). String concatenation is the slowest.

Parsing Strings (Import/Exporting)

Since many file formats are ASCII, the way you parse/export strings can make a large difference in how fast your script runs.

There are a few ways to parse strings when importing them into Blender.

Parsing Numbers

Use `float(string)` rather than `eval(string)`, if you know the value will be an int then `int(string)`. `float()` will work for an

too but it is faster to read ints with `int()`.

Checking String Start/End

If you are checking the start of a string for a keyword, rather than:

```
>>> if line[0:5] == "vert ": ...
```

Use:

```
>>> if line.startswith("vert "):
```

Using `startswith()` is slightly faster (around 5%) and also avoids a possible error with the slice length not matching the string length.

`my_string.endswith("foo_bar")` can be used for line endings too.

If you are unsure whether the text is upper or lower case, use the `lower()` or `upper()` string function:

```
>>> if line.lower().startswith("vert ")
```

Error Handling

The **try** statement is useful to save time writing error checking code. However, **try** is significantly slower than an **if** since an exception has to be set each time, so avoid using **try** in areas of your code that execute in a loop and runs many times.

There are cases where using **try** is faster than checking whether the condition will raise an error, so it is worth experimenting.

Value Comparison

Python has two ways to compare values `a == b` and `a is b`, the difference is that `==` may run the objects comparison function `__cmp__()` whereas `is` compares identity, this is, that both variables reference the same item in memory.

In cases where you know you are checking for the same value which is referenced from multiple places, `is` is faster.

Time Your Code

While developing a script it is good to time it to be aware of any changes in performance, this can be done simply:

```
import time
time_start = time.time()

# do something...

print("My Script Finished: %.4f sec" % (time.time() - time_start))
```

Gotchas

This document attempts to help you work with the Blender API in areas that can be troublesome and avoid practices that are known to cause instability.

Skip to content

Bones & Armatures

Edit Bones, Pose Bones, Bone… Bones

Armature Bones in Blender have three distinct data structures that contain them. If you are accessing the bones through one of them, you may not have access to the properties you really need.

> Note
>
> In the following examples `bpy.context.object` is assumed to be an armature object.

Edit Bones

`bpy.context.object.data.edit_bones` contains an edit bones; to access them you must set the armature mode to Edit-Mode first (ed bones do not exist in Object or Pose-Mode). Use these to create new bones, set their head/tail or roll, change their parenting relationships to other bone etc.

Example using `bpy.types.EditBone` in armature Edit-Mode which is only possible in Edit-Mode:

```
>>> bpy.context.object.data.edit_bones["Bone"].head = Vector((1.0, 2.0, 3.0))
```

This will be empty outside of Edit-Mode:

```
>>> mybones = bpy.context.selected_editable_bones
```

Returns an edit bone only in Edit-Mode:

```
>>> bpy.context.active_bone
```

Bones (Object-Mode)

`bpy.context.object.data.bones` contains bones. These *live* in Object-Mode, and have various properties you can change, note that the head and tail properties are read-only.

Example using `bpy.types.Bone` in Object or Pose-Mode returning a bone (not an edit bone) outside of Edit-Mode:

```
>>> bpy.context.active_bone
```

This works, as with Blender the setting can be edited in any mode:

```
>>> bpy.context.object.data.bones["Bone"].use_deform = True
```

Accessible but read-only:

```
>>> tail = myobj.data.bones["Bone"].tail
```

Pose Bones

`bpy.context.object.pose.bones` contains pose bones. This is where animation data resides, i.e. animatable transformations are applied pose bones, as are constraints and IK-settings.

Examples using `bpy.types.PoseBone` in Object or Pose-Mode:

```
# Gets the name of the first constraint (if it exists)
bpy.context.object.pose.bones["Bone"].constraints[0].name
```

```
bpy.context.object.pose.bones["Bone"].constraints[0].name

# Gets the last selected pose bone (Pose-Mode only)
bpy.context.active_pose_bone
```

Note

Notice the pose is accessed from the object rather than the object data, this is why Blender can have two or more objects sharing the same armature in different poses.

Note

Strictly speaking pose bones are not bones, they are just the state of the armature, stored in the `bpy.types.Object` rather than the `bpy.types.Armature`, yet the real bones are accessible from the pose bones via `bpy.types.PoseBone.bone`.

Armature Mode Switching

While writing scripts that deal with armatures you may find you have to switch between modes, when doing so take care when switching out of Edit-Mo not to keep references to the edit bones or their head/tail vectors. Further access to these will crash Blender so it's important that the script clearly separates sections of the code which operate in different modes.

This is mainly an issue with Edit-Mode since pose data can be manipulated without having to be in Pose-Mode, yet for operator access you may still nee to enter Pose-Mode.

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.

Skip to content

File Paths & String Encoding

Relative File Paths

Blender's relative file paths are not compatible with standard Python modules such as `sys` and `os`. Built-in Python functions don't understand Blender's `//` prefix which denotes the blend-file path.

A common case where you would run into this problem is when exporting a material with associated image paths:

```
>>> bpy.path.abspath(image.filepath)
```

When using Blender data from linked libraries there is an unfortunate complication since the path will be relative to the library rather than the open blend-file. When the data block may be from an external blend-file pass the library argument from the `bpy.types.ID`.

```
>>> bpy.path.abspath(image.filepath, library=image.library)
```

These returns the absolute path which can be used with native Python modules.

Unicode Problems

Python supports many different encodings so there is nothing stopping you from writing a script in `latin1` or `iso-8859-15`. See PEP 263.

However, this complicates matters for Blender's Python API because `.blend` files don't have an explicit encoding. To avoid the problem for Python integration and script authors we have decided that all strings in blend-files **must** be `UTF-8`, `ASCII` compatible. This means assigning strings with different encodings to an object name, for instance, will raise an error.

Paths are an exception to this rule since the existence of non-UTF-8 paths on the user's file system cannot be ignored. This means seemingly harmless expressions can raise errors, e.g:

```
>>> print(bpy.data.filepath)
UnicodeEncodeError: 'ascii' codec can't encode characters in position 10-21: ordinal not i
```

```
>>> bpy.context.object.name = bpy.data.filepath
Traceback (most recent call last):
  File "<blender_console>", line 1, in <module>
TypeError: bpy_struct: item.attr= val: Object.name expected a string type, not str
```

Here are two ways around file-system encoding issues:

```
>>> print(repr(bpy.data.filepath))
```

```
>>> import os
>>> filepath_bytes = os.fsencode(bpy.data.filepath)
>>> filepath_utf8 = filepath_bytes.decode('utf-8', "replace")
>>> bpy.context.object.name = filepath_utf8
```

Unicode encoding/decoding is a big topic with comprehensive Python documentation, to keep it short about encoding problems – here are some suggestions:

- Always use UTF-8 encoding or convert to UTF-8 where the input is unknown.
- Avoid manipulating file paths as strings directly, use `os.path` functions instead.
- Use `os.fsencode()` or `os.fsdecode()` instead of built-in string decoding functions when operating on paths.
- To print paths or to include them in the user interface use `repr(path)` first or `"%r" % path` with string formatting.

Note

Sometimes it’s preferable to avoid string encoding issues by using bytes instead of Python strings, when reading some input it’s less trouble to read it as binary data though you will still need to decide how to treat any strings you want to use with Blender, some importers do this.

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 nc 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 reasc 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)
```

Skip to content

Modes and Mesh Access

When working with mesh data you may run into the problem where a script fails to run as expected in Edit-Mode. This is caused by Edit-Mode having it own data which is only written back to the mesh when exiting Edit-Mode.

A common example is that exporters may access a mesh through `obj.data` (a `bpy.types.Mesh`) when the user is in Edit-Mode, where the mesh data is available but out of sync with the edit mesh.

In this situation you can…

- Exit Edit-Mode before running the tool.
- Explicitly update the mesh by calling `bmesh.types.BMesh.to_mesh`.
- Modify the script to support working on the edit-mode data directly, see: `bmesh.from_edit_mesh`.
- Report the context as incorrect and only allow the script to run outside Edit-Mode.

N-Gons and Tessellation

Since 2.63 n-gons are supported, this adds some complexity since in some cases you need to access triangles still (some exporters for example).

There are now three ways to access faces:

- `bpy.types.MeshPolygon` – this is the data structure which now stores faces in Object-Mode (access as `mesh.polygons` rather than `mesh.faces`).
- `bpy.types.MeshLoopTriangle` – the result of tessellating polygons into triangles (access as `mesh.loop_triangles`).
- `bmesh.types.BMFace` – the polygons as used in Edit-Mode.

For the purpose of the following documentation, these will be referred to as polygons, loop triangles and BMesh-faces respectively.

Faces with five or more sides will be referred to as `ngons`.

Support Overview

Usage	`bpy.types.MeshPolygon`	`bpy.types.MeshLoopTriangle`	`bmesh.types.BMFace`
Import/Create	Poor *(inflexible)*	Unusable *(read-only)*.	Best
Manipulate	Poor *(inflexible)*	Unusable *(read-only)*.	Best
Export/Output	Good *(n-gon support)*	Good *(When n-gons cannot be used)*	Good *(n-gons, extra memory overhead)*

Note

Using the `bmesh` API is completely separate API from `bpy`, typically you would use one or the other based on the level of editing needed, not simply for a different way to access faces.

Creating

All three data types can be used for face creation:

- Polygons are the most efficient way to create faces but the data structure is *very* rigid and inflexible, you must have all your vertices and faces ready a create them all at once. This is further complicated by the fact that each polygon does not store its own vertices, rather they reference an index and siz in `bpy.types.Mesh.loops` which are a fixed array too.
- BMesh-faces are most likely the easiest way to create faces in new scripts, since faces can be added one by one and the API has features intended fo mesh manipulation. While `bmesh.types.BMesh` uses more memory it can be managed by only operating on one mesh at a time.

Editing

Editing is where the three data types vary most.

- Polygons are very limited for editing, changing materials and options like smooth works, but for anything else they are too inflexible and are only intended for storage.
- Loop-triangles should not be used for editing geometry because doing so will cause existing n-gons to be tessellated.
- BMesh-faces are by far the best way to manipulate geometry.

Exporting

All three data types can be used for exporting, the choice mostly depends on whether the target format supports n-gons or not.

- Polygons are the most direct and efficient way to export providing they convert into the output format easily enough.
- Loop-triangles work well for exporting to formats which don't support n-gons, in fact this is the only place where their use is encouraged.
- BMesh-Faces can work for exporting too but may not be necessary if polygons can be used since using BMesh gives some overhead because it's no the native storage format in Object-Mode.

Using Operators

Blender's operators are tools for users to access, that can be accessed with Python too which is very useful. Still operators have limitations that can mak them cumbersome to script.

The main limits are:

- Can't pass data such as objects, meshes or materials to operate on (operators use the context instead).
- The return value from calling an operator is the success (if it finished or was canceled), in some cases it would be more logical from an API perspectiv to return the result of the operation.
- Operators' poll function can fail where an API function would raise an exception giving details on exactly why.

Why does an operator's poll fail?

When calling an operator it gives an error like this:

```
>>> bpy.ops.action.clean(threshold=0.001)
RuntimeError: Operator bpy.ops.action.clean.poll() failed, context is incorrect
```

Which raises the question as to what the correct context might be?

Typically operators check for the active area type, a selection or active object they can operate on, but some operators are more strict when they run. In most cases you can figure out what context an operator needs by examining how it's used in Blender and thinking about what it does.

If you're still stuck, unfortunately, the only way to eventually know what is causing the error is to read the source code for the poll function and see what is checking. For Python operators it's not so hard to find the source since it's included with Blender and the source file and line is included in the operato reference docs. Downloading and searching the C code isn't so simple, especially if you're not familiar with the C language but by searching the operator name or description you should be able to find the poll function with no knowledge of C.

Note

Blender does have the functionality for poll functions to describe why they fail, but it's currently not used much, if you're interested to help improve the API feel free to add calls to `bpy.types.Operator.poll_message_set` (`CTX_wm_operator_poll_msg_set` in C) where it's not obvious why poll fails, e.g:

```
>>> bpy.ops.gpencil.draw()
RuntimeError: Operator bpy.ops.gpencil.draw.poll() Failed to find Grease Pencil data to d
```

The operator still doesn't work!

Certain operators in Blender are only intended for use in a specific context, some operators for example are only called from the properties editor where they check the current material, modifier or constraint.

Examples of this are:

- `bpy.ops.texture.slot_move`
- `bpy.ops.constraint.limitdistance_reset`
- `bpy.ops.object.modifier_copy`
- `bpy.ops.buttons.file_browse`

Another possibility is that you are the first person to attempt to use this operator in a script and some modifications need to be made to the operator to ru in a different context. If the operator should logically be able to run but fails when accessed from a script it should be reported to the bug tracker.

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'}
```

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
```

Skip to content

Tips and Tricks

Here are various suggestions that you might find useful when writing scripts. Some of these are just Python features that you may not have thought to use with Blender, others are Blender-specific.

Use the Terminal

When writing Python scripts, it's useful to have a terminal open, this is not the built-in Python console but a terminal application which is used to start Blender.

The three main use cases for the terminal are:

- You can see the output of `print()` as your script runs, which is useful to view debug info.
- The error traceback is printed in full to the terminal which won't always generate an report message in Blender's user interface (depending on how th script is executed).
- If the script runs for too long or you accidentally enter an infinite loop, `Ctrl - C` in the terminal (`Ctrl - Break` on Windows) will quit the script early.

See also

Launching from the Command Line.

Interface Tricks

Access Operator Commands

You may have noticed that the tooltip for menu items and buttons includes the `bpy.ops.[...]` command to run that button, a handy (hidden) feature is that you can press `Ctrl - C` over any menu item or button to copy this command into the clipboard.

Access Data Path

To find the path from an `ID` data-block to its setting isn't always so simple since it may be nested away. To get this quickly open the context menu of th setting and select *Copy Data Path*, if this can't be generated, only the property name is copied.

Note

This uses the same method for creating the animation path used by `bpy.types.FCurve.data_path` and `bpy.types.DriverTarget.data_path` drivers.

Show All Operators

While Blender logs operators in the Info editor, this only reports operators with the `REGISTER` option enabled so as not to flood the *Info* view with calls to `bpy.ops.view3d.smoothview` and `bpy.ops.view3d.zoom`. Yet for testing it can be useful to see **every** operator called in a terminal, do this by enabling the debug option either by passing the `--debug-wm` argument when starting Blender or by setting `bpy.app.debug_wm` to `True` while Blender is running.

Use an External Editor

Blender's text editor is fine for small changes and writing tests but its not full featured, for larger projects you'll probably want to use a standalone editor Python IDE. Editing a text file externally and having the same text open in Blender does work but isn't that optimal so here are two ways you can use an external file from Blender. Using the following examples you'll still need text data-block in Blender to execute, but reference an external file rather than including it directly.

Executing External Scripts

This is the equivalent to running the script directly, referencing a script's path from a two line code block.

```
filename = "/full/path/to/myscript.py"
exec(compile(open(filename).read(), filename, 'exec'))
```

You might want to reference a script relative to the blend-file.

```
import bpy
import os

filename = os.path.join(os.path.dirname(bpy.data.filepath), "myscript.py")
exec(compile(open(filename).read(), filename, 'exec'))
```

Executing Modules

This example shows loading a script in as a module and executing a module function.

```
import myscript
import importlib

importlib.reload(myscript)
myscript.main()
```

Notice that the script is reloaded every time, this forces use of the modified version, otherwise the cached one in `sys.modules` would be used until Blender was restarted.

The important difference between this and executing the script directly is it has to call a function in the module, in this case `main()` but it can be any function, an advantage with this is you can pass arguments to the function from this small script which is often useful for testing different settings quickly.

The other issue with this is the script has to be in Python's module search path. While this is not best practice – for testing purposes you can extend the search path, this following example adds the current blend-file's directory to the search path and then loads the script as a module.

```
import sys
import os
import bpy

blend_dir = os.path.dirname(bpy.data.filepath)
if blend_dir not in sys.path:
   sys.path.append(blend_dir)

import myscript
import importlib
importlib.reload(myscript)
myscript.main()
```

Use Blender without it's User Interface

While developing your own scripts Blender's interface can get in the way, manually reloading, running the scripts, opening file import, etc. adds overhead For scripts that are not interactive it can end up being more efficient not to use Blender's interface at all and instead execute the script on the command line.

```
blender --background --python myscript.py
```

You might want to run this with a blend-file so the script has some data to operate on.

```
blender myscene.blend --background --python myscript.py
```

Note

Depending on your setup you might have to enter the full path to the Blender executable.

Once the script is running properly in background mode, you'll want to check the output of the script, this depends completely on the task at hand, however, here are some suggestions:

- Render the output to an image, use an image viewer and keep writing over the same image each time.
- Save a new blend-file, or export the file using one of Blender's exporters.
- If the results can be displayed as text then print them or write them to a file.

While this can take a little time to setup, it can be well worth the effort to reduce the time it takes to test changes. You can even have Blender running the script every few seconds with a viewer updating the results, so no need to leave your text editor to see changes.

Use External Tools

When there are no readily available Python modules to perform specific tasks it's worth keeping in mind you may be able to have Python execute an external command on your data and read the result back in.

Using external programs adds an extra dependency and may limit who can use the script but to quickly setup your own custom pipeline or writing one-of scripts this can be handy.

Examples include:

- Run Gimp in batch mode to execute custom scripts for advanced image processing.
- Write out 3D models to use external mesh manipulation tools and read back in the results.
- Convert files into recognizable formats before reading.

Bundled Python & Extensions

The Blender releases distributed from blender.org include a complete Python installation on all platforms, this has the disadvantage that any extensions yo have installed on your system's Python environment will not be found by Blender.

There are two ways to work around this:

- Remove Blender Python subdirectory, Blender will then fallback on the system's Python and use that instead.

  Depending on your platform, you may need to explicitly reference the location of your Python installation using the `PYTHONPATH` environment variable, e.g:

  ```
  PYTHONPATH=/usr/lib/python3.7 ./blender --python-use-system-env
  ```

  Warning

  The Python (major, minor) version must match the one that Blender comes with. Therefor you can't use Python 3.6 with Blender built to use Python 3.7.

- Copy or link the extensions into Blender's Python subdirectory so Blender can access them, you can also copy the entire Python installation into Blender's subdirectory, replacing the one Blender comes with. This works as long as the Python versions match and the paths are created in the same relative locations. Doing this has the advantage that you can redistribute this bundle to others with Blender including any extensions you rely on.

Insert a Python Interpreter into your Script

In the middle of a script you may want to inspect variables, run functions and inspect the flow.

```
import code
code.interact(local=locals())
```

If you want to access both global and local variables run this:

```
import code
namespace = globals().copy()
namespace.update(locals())
code.interact(local=namespace)
```

The next example is an equivalent single line version of the script above which is easier to paste into your code:

```
__import__('code').interact(local=dict(globals(), **locals()))
```

`code.interact` can be added at any line in the script and will pause the script to launch an interactive interpreter in the terminal, when you're don you can quit the interpreter and the script will continue execution.

If you have **IPython** installed you can use its `embed()` function which uses the current namespace. The IPython prompt has auto-complete and some useful features that the standard Python eval-loop doesn't have.

```
import IPython
IPython.embed()
```

Admittedly this highlights the lack of any Python debugging support built into Blender, but its still a handy thing to know.

Advanced

Blender as a Module

From a Python perspective it's nicer to have everything as an extension which lets the Python script combine many components.

Advantages include:

- You can use external editors or IDEs with Blender's Python API and execute scripts within the IDE (step over code, inspect variables as the script runs).
- Editors or IDEs can auto-complete Blender modules and variables.
- Existing scripts can import Blender APIs without having to be run inside of Blender.

This is marked advanced because to run Blender as a Python module requires a special build option. For instructions on building see Building Blender as Python module.

Python Safety (Build Option)

Since it's possible to access data which has been removed (see Gotchas), it can be hard to track down the cause of crashes. To raise Python exceptions on accessing freed data (rather than crashing), enable the CMake build option `WITH_PYTHON_SAFETY`. This enables data tracking which makes data access about two times slower which is why the option isn't enabled in release builds.

Skip to content

BVHTree Utilities (mathutils.bvhtree)

BVH tree structures for proximity searches and ray casts on geometry.

class mathutils.bvhtree.**BVHTree**

classmethod FromBMesh(bmesh, epsilon=0.0)

BVH tree based on `BMesh` data.

PARAMETERS:

- **bmesh** (`BMesh`) – BMesh data.
- **epsilon** (*float*) – Increase the threshold for detecting overlap and raycast hits.

classmethod FromObject(object, depsgraph, deform=True, render=False, cage=False, epsilon=0.0)

BVH tree based on `Object` data.

PARAMETERS:

- **object** (`Object`) – Object data.
- **depsgraph** (`Depsgraph`) – Depsgraph to use for evaluating the mesh.
- **deform** (*bool*) – Use mesh with deformations.
- **cage** (*bool*) – Use modifiers cage.
- **epsilon** (*float*) – Increase the threshold for detecting overlap and raycast hits.

classmethod FromPolygons(vertices, polygons, all_triangles=False, epsilon=0.0)

BVH tree constructed geometry passed in as arguments.

PARAMETERS:

- **vertices** (*Sequence[Sequence[float]]*) – float triplets each representing `(x, y, z)`
- **polygons** (*Sequence[Sequence[int]]*) – Sequence of polygons, each containing indices to the vertices argument.
- **all_triangles** (*bool*) – Use when all **polygons** are triangles for more efficient conversion.
- **epsilon** (*float*) – Increase the threshold for detecting overlap and raycast hits.

find_nearest(origin, distance=1.84467e+19)

Find the nearest element (typically face index) to a point.

PARAMETERS:

- **co** (`Vector`) – Find nearest element to this point.
- **distance** (*float*) – Maximum distance threshold.

RETURNS:

Returns a tuple: (position, normal, index, distance), Values will all be None if no hit is found.

RETURN TYPE:

tuple[`Vector` | None, `Vector` | None, int | None, float | None]

find_nearest_range(origin, distance=1.84467e+19)

Find the nearest elements (typically face index) to a point in the distance range.

PARAMETERS:

- **co** (`Vector`) – Find nearest elements to this point.
- **distance** (*float*) – Maximum distance threshold.

RETURNS:

Returns a list of tuples (position, normal, index, distance)

RETURN TYPE:

list[tuple[`Vector`, `Vector`, int, float]]

overlap(other_tree)

Find overlapping indices between 2 trees.

PARAMETERS:

other_tree (`BVHTree`) – Other tree to perform overlap test on.

RETURNS:

Returns a list of unique index pairs, the first index referencing this tree, the second referencing the **other_tree**.

RETURN TYPE:

list[tuple[int, int]]

ray_cast(origin, direction, distance=sys.float_info.max)

Cast a ray onto the mesh.

PARAMETERS:

- **origin** (`Vector`) – Start location of the ray in object space.
- **direction** (`Vector`) – Direction of the ray in object space.
- **distance** (*float*) – Maximum distance threshold.

RETURNS:

Returns a tuple: (position, normal, index, distance), Values will all be None if no hit is found.

RETURN TYPE:

tuple[`Vector` | None, `Vector` | None, int | None, float | None]

Geometry Utilities (mathutils.geometry)

The Blender geometry module

mathutils.geometry.**area_tri(v1, v2, v3)**

Returns the area size of the 2D or 3D triangle defined.

PARAMETERS:

- **v1** (`mathutils.Vector`) – Point1
- **v2** (`mathutils.Vector`) – Point2
- **v3** (`mathutils.Vector`) – Point3

RETURN TYPE:

float

mathutils.geometry.**barycentric_transform(point, tri_a1, tri_a2, tri_a3, tri_b1, tri_b2, tri_b3)**

Return a transformed point, the transformation is defined by 2 triangles.

PARAMETERS:

- **point** (`mathutils.Vector`) – The point to transform.
- **tri_a1** (`mathutils.Vector`) – source triangle vertex.
- **tri_a2** (`mathutils.Vector`) – source triangle vertex.
- **tri_a3** (`mathutils.Vector`) – source triangle vertex.
- **tri_b1** (`mathutils.Vector`) – target triangle vertex.
- **tri_b2** (`mathutils.Vector`) – target triangle vertex.
- **tri_b3** (`mathutils.Vector`) – target triangle vertex.

RETURNS:

The transformed point

RETURN TYPE:

`mathutils.Vector`

mathutils.geometry.**box_fit_2d(points)**

Returns an angle that best fits the points to an axis aligned rectangle

PARAMETERS:

points (*Sequence[Sequence[float]]*) – Sequence of 2D points.

RETURNS:

angle

RETURN TYPE:

float

mathutils.geometry.**box_pack_2d(boxes)**

Returns a tuple with the width and height of the packed bounding box.

PARAMETERS:

boxes (*list[list[float]]*) – list of boxes, each box is a list where the first 4 items are [X, Y, width, height, …] other items are ignored. The X and Y values in this list are modified to set the packed positions.

RETURNS:

The width and height of the packed bounding box.

RETURN TYPE:

tuple[float, float]

mathutils.geometry.**closest_point_on_tri(pt, tri_p1, tri_p2, tri_p3)**

Takes 4 vectors: one is the point and the next 3 define the triangle.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **tri_p1** (`mathutils.Vector`) – First point of the triangle
- **tri_p2** (`mathutils.Vector`) – Second point of the triangle
- **tri_p3** (`mathutils.Vector`) – Third point of the triangle

RETURNS:

The closest point of the triangle.

RETURN TYPE:

`mathutils.Vector`

mathutils.geometry.**convex_hull_2d(points)**

Returns a list of indices into the list given

PARAMETERS:

points (*Sequence[Sequence[float]]*) – Sequence of 2D points.

RETURNS:

a list of indices

RETURN TYPE:

list[int]

mathutils.geometry.**delaunay_2d_cdt(vert_coords, edges, faces, output_type, epsilon, need_ids=True)**

Computes the Constrained Delaunay Triangulation of a set of vertices, with edges and faces that must appear in the triangulation. Some triangles ma be eaten away, or combined with other triangles, according to output type. The returned verts may be in a different order from input verts, may be moved slightly, and may be merged with other nearby verts. The three returned orig lists give, for each of verts, edges, and faces, the list of input element indices corresponding to the positionally same output element. For edges, the orig indices start with the input edges and then continue with the edges implied by each of the faces (n of them for an n-gon). If the need_ids argument is supplied, and False, then the code skips the preparatio of the orig arrays, which may save some time.

PARAMETERS:

- **vert_coords** (Sequence[`mathutils.Vector`]) – Vertex coordinates (2d)
- **edges** (*Sequence[Sequence[int, int]]*) – Edges, as pairs of indices in `vert_coords`
- **faces** (*Sequence[Sequence[int]]*) – Faces, each sublist is a face, as indices in *vert_coords* (CCW oriented)
- **output_type** (*int*) – What output looks like. 0 => triangles with convex hull. 1 => triangles inside constraints. 2 => the input constraints, intersected. 3 => like 2 but detect holes and omit them from output. 4 => like 2 but with extra edges to make valid BMesh faces. 5 => like 4 but detect holes and omit them from output.
- **epsilon** (*float*) – For nearness tests; should not be zero
- **need_ids** – are the orig output arrays needed?

RETURNS:

Output tuple, (vert_coords, edges, faces, orig_verts, orig_edges, orig_faces)

RETURN TYPE:

tuple[list[`mathutils.Vector`], list[tuple[int, int]], list[list[int]], list[list[int]], list[list[int]], list[list[int]]]

mathutils.geometry.**distance_point_to_plane(pt, plane_co, plane_no)**

Returns the signed distance between a point and a plane (negative when below the normal).

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **plane_co** (`mathutils.Vector`) – A point on the plane

- **plane_no** (`mathutils.Vector`) – The direction the plane is facing

RETURN TYPE:

float

mathutils.geometry.**interpolate_bezier(knot1, handle1, handle2, knot2, resolution)**

Interpolate a bezier spline segment.

PARAMETERS:

- **knot1** (`mathutils.Vector`) – First bezier spline point.
- **handle1** (`mathutils.Vector`) – First bezier spline handle.
- **handle2** (`mathutils.Vector`) – Second bezier spline handle.
- **knot2** (`mathutils.Vector`) – Second bezier spline point.
- **resolution** (*int*) – Number of points to return.

RETURNS:

The interpolated points.

RETURN TYPE:

list[`mathutils.Vector`]

mathutils.geometry.**intersect_line_line(v1, v2, v3, v4)**

Returns a tuple with the points on each line respectively closest to the other.

PARAMETERS:

- **v1** (`mathutils.Vector`) – First point of the first line
- **v2** (`mathutils.Vector`) – Second point of the first line
- **v3** (`mathutils.Vector`) – First point of the second line
- **v4** (`mathutils.Vector`) – Second point of the second line

RETURNS:

The intersection on each line or None when the lines are co-linear.

RETURN TYPE:

tuple[`mathutils.Vector`, `mathutils.Vector`] | None

mathutils.geometry.**intersect_line_line_2d(lineA_p1, lineA_p2, lineB_p1, lineB_p2)**

Takes 2 segments (defined by 4 vectors) and returns a vector for their point of intersection or None.

> Warning
>
> Despite its name, this function works on segments, and not on lines.

PARAMETERS:

- **lineA_p1** (`mathutils.Vector`) – First point of the first line
- **lineA_p2** (`mathutils.Vector`) – Second point of the first line
- **lineB_p1** (`mathutils.Vector`) – First point of the second line
- **lineB_p2** (`mathutils.Vector`) – Second point of the second line

RETURNS:

The point of intersection or None when not found

RETURN TYPE:

`mathutils.Vector` | None

mathutils.geometry.**intersect_line_plane(line_a, line_b, plane_co, plane_no, no_flip=False)**

Calculate the intersection between a line (as 2 vectors) and a plane. Returns a vector for the intersection or None.

PARAMETERS:

PARAMETERS:

- **line_a** (mathutils.Vector) – First point of the first line
- **line_b** (mathutils.Vector) – Second point of the first line
- **plane_co** (mathutils.Vector) – A point on the plane
- **plane_no** (mathutils.Vector) – The direction the plane is facing

RETURNS:

The point of intersection or None when not found

RETURN TYPE:

mathutils.Vector | None

mathutils.geometry.**intersect_line_sphere(line_a, line_b, sphere_co, sphere_radius, clip=True)**

Takes a line (as 2 points) and a sphere (as a point and a radius) and returns the intersection

PARAMETERS:

- **line_a** (mathutils.Vector) – First point of the line
- **line_b** (mathutils.Vector) – Second point of the line
- **sphere_co** (mathutils.Vector) – The center of the sphere
- **sphere_radius** (*float*) – Radius of the sphere

RETURNS:

The intersection points as a pair of vectors or None when there is no intersection

RETURN TYPE:

tuple[mathutils.Vector | None, mathutils.Vector | None]

mathutils.geometry.**intersect_line_sphere_2d(line_a, line_b, sphere_co, sphere_radius, clip=True)**

Takes a line (as 2 points) and a sphere (as a point and a radius) and returns the intersection

PARAMETERS:

- **line_a** (mathutils.Vector) – First point of the line
- **line_b** (mathutils.Vector) – Second point of the line
- **sphere_co** (mathutils.Vector) – The center of the sphere
- **sphere_radius** (*float*) – Radius of the sphere

RETURNS:

The intersection points as a pair of vectors or None when there is no intersection

RETURN TYPE:

tuple[mathutils.Vector | None, mathutils.Vector | None]

mathutils.geometry.**intersect_plane_plane(plane_a_co, plane_a_no, plane_b_co, plane_b_no)**

Return the intersection between two planes

PARAMETERS:

- **plane_a_co** (mathutils.Vector) – Point on the first plane
- **plane_a_no** (mathutils.Vector) – Normal of the first plane
- **plane_b_co** (mathutils.Vector) – Point on the second plane
- **plane_b_no** (mathutils.Vector) – Normal of the second plane

RETURNS:

The line of the intersection represented as a point and a vector or None if the intersection can't be calculated

RETURN TYPE:

tuple[mathutils.Vector, mathutils.Vector] | tuple[None, None]

mathutils.geometry.**intersect_point_line(pt, line_p1, line_p2)**

Takes a point and a line and returns a tuple with the closest point on the line and its distance from the first point of the line as a percentage of the length of the line.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **line_p1** (`mathutils.Vector`) – First point of the line
- **line_p1** – Second point of the line

RETURN TYPE:

tuple[`mathutils.Vector`, float]

mathutils.geometry.**intersect_point_quad_2d(pt, quad_p1, quad_p2, quad_p3, quad_p4)**

Takes 5 vectors (using only the x and y coordinates): one is the point and the next 4 define the quad, only the x and y are used from the vectors. Returns 1 if the point is within the quad, otherwise 0. Works only with convex quads without singular edges.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **quad_p1** (`mathutils.Vector`) – First point of the quad
- **quad_p2** (`mathutils.Vector`) – Second point of the quad
- **quad_p3** (`mathutils.Vector`) – Third point of the quad
- **quad_p4** (`mathutils.Vector`) – Fourth point of the quad

RETURN TYPE:

int

mathutils.geometry.**intersect_point_tri(pt, tri_p1, tri_p2, tri_p3)**

Takes 4 vectors: one is the point and the next 3 define the triangle. Projects the point onto the triangle plane and checks if it is within the triangle.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **tri_p1** (`mathutils.Vector`) – First point of the triangle
- **tri_p2** (`mathutils.Vector`) – Second point of the triangle
- **tri_p3** (`mathutils.Vector`) – Third point of the triangle

RETURNS:

Point on the triangles plane or None if its outside the triangle

RETURN TYPE:

`mathutils.Vector` | None

mathutils.geometry.**intersect_point_tri_2d(pt, tri_p1, tri_p2, tri_p3)**

Takes 4 vectors (using only the x and y coordinates): one is the point and the next 3 define the triangle. Returns 1 if the point is within the triangle, otherwise 0.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **tri_p1** (`mathutils.Vector`) – First point of the triangle
- **tri_p2** (`mathutils.Vector`) – Second point of the triangle
- **tri_p3** (`mathutils.Vector`) – Third point of the triangle

RETURN TYPE:

int

mathutils.geometry.**intersect_ray_tri(v1, v2, v3, ray, orig, clip=True)**

Returns the intersection between a ray and a triangle, if possible, returns None otherwise.

PARAMETERS:

- **v1** (`mathutils.Vector`) – Point1
- **v2** (`mathutils.Vector`) – Point2
- **v3** (`mathutils.Vector`) – Point3
- **ray** (`mathutils.Vector`) – Direction of the projection
- **orig** (`mathutils.Vector`) – Origin
- **clip** (*bool*) – When False, don't restrict the intersection to the area of the triangle, use the infinite plane defined by the triangle.

RETURNS:

The point of intersection or None if no intersection is found

RETURN TYPE:

`mathutils.Vector` | None

mathutils.geometry.**intersect_sphere_sphere_2d(p_a, radius_a, p_b, radius_b)**

Returns 2 points on between intersecting circles.

PARAMETERS:

- **p_a** (`mathutils.Vector`) – Center of the first circle
- **radius_a** (*float*) – Radius of the first circle
- **p_b** (`mathutils.Vector`) – Center of the second circle
- **radius_b** (*float*) – Radius of the second circle

RETURNS:

2 points on between intersecting circles or None when there is no intersection.

RETURN TYPE:

tuple[`mathutils.Vector`, `mathutils.Vector`] | tuple[None, None]

mathutils.geometry.**intersect_tri_tri_2d(tri_a1, tri_a2, tri_a3, tri_b1, tri_b2, tri_b3)**

Check if two 2D triangles intersect.

RETURN TYPE:

bool

mathutils.geometry.**normal(vectors)**

Returns the normal of a 3D polygon.

PARAMETERS:

vectors (*Sequence[Sequence[float]]*) – 3 or more vectors to calculate normals.

RETURN TYPE:

`mathutils.Vector`

mathutils.geometry.**points_in_planes(planes, epsilon_coplanar=1e-4, epsilon_isect=1e-6)**

Returns a list of points inside all planes given and a list of index values for the planes used.

PARAMETERS:

- **planes** (list[`mathutils.Vector`]) – List of planes (4D vectors).
- **epsilon_coplanar** (*float*) – Epsilon value for interpreting plane pairs as co-plannar.
- **epsilon_isect** (*float*) – Epsilon value for intersection.

RETURNS:

Two lists, once containing the 3D coordinates inside the planes, another containing the plane indices used.

RETURN TYPE:

tuple[list[`mathutils.Vector`], list[int]]

mathutils.geometry.**tessellate_polygon(polylines)**

Takes a list of polylines (each point a pair or triplet of numbers) and returns the point indices for a polyline filled with triangles. Does not handle degenerate geometry (such as zero-length lines due to consecutive identical points).

PARAMETERS:

polylines (*Sequence[Sequence[Sequence[float]]] :return: A list of triangles.*) – Polygons where each polygon is a sequence of 2D or 3D points.

RETURN TYPE:

list[tuple[int, int, int]]

mathutils.geometry.**volume_tetrahedron(v1, v2, v3, v4)**

Return the volume formed by a tetrahedron (points can be in any order).

PARAMETERS:

- **v1** (`mathutils.Vector`) – Point1
- **v2** (`mathutils.Vector`) – Point2
- **v3** (`mathutils.Vector`) – Point3
- **v4** (`mathutils.Vector`) – Point4

RETURN TYPE:

float

Skip to content

Math Types & Utilities (mathutils)

This module provides access to math operations.

> Note
>
> Classes, methods and attributes that accept vectors also accept other numeric sequences, such as tuples, lists.

The `mathutils` module provides the following classes:

- `Color`,
- `Euler`,
- `Matrix`,
- `Quaternion`,
- `Vector`,

SUBMODULES

Geometry Utilities (mathutils.geometry)

BVHTree Utilities (mathutils.bvhtree)

KDTree Utilities (mathutils.kdtree)

Interpolation Utilities (mathutils.interpolate)

Noise Utilities (mathutils.noise)

```
import mathutils
from math import radians

vec = mathutils.Vector((1.0, 2.0, 3.0))

mat_rot = mathutils.Matrix.Rotation(radians(90.0), 4, 'X')
mat_trans = mathutils.Matrix.Translation(vec)

mat = mat_trans @ mat_rot
mat.invert()

mat3 = mat.to_3x3()
quat1 = mat.to_quaternion()
quat2 = mat3.to_quaternion()

quat_diff = quat1.rotation_difference(quat2)

print(quat_diff.angle)
```

class mathutils.**Color(rgb)**

This object gives access to Colors in Blender.

Most colors returned by Blender APIs are in scene linear color space, as defined by the OpenColorIO configuration. The notable exception is user interface theming colors, which are in sRGB color space.

PARAMETERS:

rgb (*Sequence[float]*) – (red, green, blue) color values where (0, 0, 0) is black & (1, 1, 1) is white.

```
import mathutils

# color values are represented as RGB values from 0 - 1, this is blue
```

```
col = mathutils.Color((0.0, 0.0, 1.0))

# as well as r/g/b attribute access you can adjust them by h/s/v
col.s *= 0.5

# you can access its components by attribute or index
print("Color R:", col.r)
print("Color G:", col[1])
print("Color B:", col[-1])
print("Color HSV: {:.2f}, {:.2f}, {:.2f}".format(*col))


# components of an existing color can be set
col[:] = 0.0, 0.5, 1.0

# components of an existing color can use slice notation to get a tuple
print("Values: {:f}, {:f}, {:f}".format(*col))

# colors can be added and subtracted
col += mathutils.Color((0.25, 0.0, 0.0))

# Color can be multiplied, in this example color is scaled to 0-255
# can printed as integers
print("Color: {:d}, {:d}, {:d}".format(*(int(c) for c in (col * 255.0))))

# This example prints the color as hexadecimal
print("Hexadecimal: {:02x}{:02x}{:02x}".format(int(col.r * 255), int(col.g * 255), int(
```

copy()

Returns a copy of this color.

RETURNS:

A copy of the color.

RETURN TYPE:

Color

Note

use this to get a copy of a wrapped color with no reference to the original data.

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

from_aces_to_scene_linear()

Convert from ACES2065-1 linear to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

Color

from_rec709_linear_to_scene_linear()

Convert from Rec.709 linear color space to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

Color

from_scene_linear_to_aces()

Convert from scene linear to ACES2065-1 linear color space.

RETURNS:

A color in ACES2065-1 linear color space.

RETURN TYPE:

Color

from_scene_linear_to_rec709_linear()

Convert from scene linear to Rec.709 linear color space.

RETURNS:

A color in Rec.709 linear color space.

RETURN TYPE:

Color

from_scene_linear_to_srgb()

Convert from scene linear to sRGB color space.

RETURNS:

A color in sRGB color space.

RETURN TYPE:

Color

from_scene_linear_to_xyz_d65()

Convert from scene linear to CIE XYZ (Illuminant D65) color space.

RETURNS:

A color in XYZ color space.

RETURN TYPE:

Color

from_srgb_to_scene_linear()

Convert from sRGB to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

Color

from_xyz_d65_to_scene_linear()

Convert from CIE XYZ (Illuminant D65) to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

Color

b

Blue color channel.

TYPE:

float

g

Green color channel.

TYPE:

float

h

HSV Hue component in [0, 1].

TYPE:

float

hsv

HSV Values in [0, 1].

TYPE:

float triplet

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

owner

The item this is wrapping or None (read-only).

r

Red color channel.

TYPE:

float

s

HSV Saturation component in [0, 1].

TYPE:

float

v

HSV Value component in [0, 1].

TYPE:
float

class mathutils.**Euler(angles, order='XYZ')**

This object gives access to Eulers in Blender.

See also

Euler angles on Wikipedia.

PARAMETERS:

- **angles** (*Sequence[float]*) – (X, Y, Z) angles in radians.
- **order** (*str*) – Optional order of the angles, a permutation of `XYZ`.

```
import mathutils
import math

# create a new euler with default axis rotation order
eul = mathutils.Euler((0.0, math.radians(45.0), 0.0), 'XYZ')

# rotate the euler
eul.rotate_axis('Z', math.radians(10.0))

# you can access its components by attribute or index
print("Euler X", eul.x)
print("Euler Y", eul[1])
print("Euler Z", eul[-1])

# components of an existing euler can be set
eul[:] = 1.0, 2.0, 3.0

# components of an existing euler can use slice notation to get a tuple
print("Values: {:f}, {:f}, {:f}".format(*eul))

# the order can be set at any time too
eul.order = 'ZYX'

# eulers can be used to rotate vectors
vec = mathutils.Vector((0.0, 0.0, 1.0))
vec.rotate(eul)

# often its useful to convert the euler into a matrix so it can be used as
# transformations with more flexibility
mat_rot = eul.to_matrix()
mat_loc = mathutils.Matrix.Translation((2.0, 3.0, 4.0))
mat = mat_loc @ mat_rot.to_4x4()
```

copy()

Returns a copy of this euler.

RETURNS:

RETURNS:

A copy of the euler.

RETURN TYPE:

`Euler`

Note

use this to get a copy of a wrapped euler with no reference to the original data.

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

make_compatible(other)

Make this euler compatible with another, so interpolating between them works as intended.

Note

the rotation order is not taken into account for this function.

rotate(other)

Rotates the euler by another mathutils value.

PARAMETERS:

other (`Euler` | `Quaternion` | `Matrix`) – rotation component of mathutils value

rotate_axis(axis, angle)

Rotates the euler a certain amount and returning a unique euler rotation (no 720 degree pitches).

PARAMETERS:

- **axis** (*str*) – single character in ['X, 'Y', 'Z'].
- **angle** (*float*) – angle in radians.

to_matrix()

Return a matrix representation of the euler.

RETURNS:

A 3x3 rotation matrix representation of the euler.

RETURN TYPE:

`Matrix`

to_quaternion()

Return a quaternion representation of the euler.

RETURNS:

Quaternion representation of the euler.

RETURN TYPE:

`Quaternion`

zero()

Set all values to zero.

is_frozen

True when this object has been frozen (read-only).

TYPE:
bool

is_valid

True when the owner of this data is valid.

TYPE:
bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:
bool

order

Euler rotation order.

TYPE:
str in ['XYZ', 'XZY', 'YXZ', 'YZX', 'ZXY', 'ZYX']

owner

The item this is wrapping or None (read-only).

x

Euler axis angle in radians.

TYPE:
float

y

Euler axis angle in radians.

TYPE:
float

z

Euler axis angle in radians.

TYPE:
float

class mathutils.**Matrix([rows])**

This object gives access to Matrices in Blender, supporting square and rectangular matrices from 2x2 up to 4x4.

PARAMETERS:
rows (*Sequence[Sequence[float]]*) – Sequence of rows. When omitted, a 4x4 identity matrix is constructed.

```
import mathutils
import math

# Create a location matrix.
mat_loc = mathutils.Matrix.Translation((2.0, 3.0, 4.0))

# Create an identity matrix.
```

```
mat_sca = mathutils.Matrix.Scale(0.5, 4, (0.0, 0.0, 1.0))

# Create a rotation matrix.
mat_rot = mathutils.Matrix.Rotation(math.radians(45.0), 4, 'X')

# Combine transformations.
mat_out = mat_loc @ mat_rot @ mat_sca
print(mat_out)

# Extract components back out of the matrix as two vectors and a quaternion.
loc, rot, sca = mat_out.decompose()
print(loc, rot, sca)

# Recombine extracted components.
mat_out2 = mathutils.Matrix.LocRotScale(loc, rot, sca)
print(mat_out2)

# It can also be useful to access components of a matrix directly.
mat = mathutils.Matrix()
mat[0][0], mat[1][0], mat[2][0] = 0.0, 1.0, 2.0

mat[0][0:3] = 0.0, 1.0, 2.0

# Each item in a matrix is a vector so vector utility functions can be used.
mat[0].xyz = 0.0, 1.0, 2.0
```

classmethod Diagonal(vector)

Create a diagonal (scaling) matrix using the values from the vector.

PARAMETERS:

vector (`Vector`) – The vector of values for the diagonal.

RETURNS:

A diagonal matrix.

RETURN TYPE:

`Matrix`

classmethod Identity(size)

Create an identity matrix.

PARAMETERS:

size (*int*) – The size of the identity matrix to construct [2, 4].

RETURNS:

A new identity matrix.

RETURN TYPE:

`Matrix`

classmethod LocRotScale(location, rotation, scale)

Create a matrix combining translation, rotation and scale, acting as the inverse of the decompose() method.

Any of the inputs may be replaced with None if not needed.

PARAMETERS:

- **location** (`Vector` | None) – The translation component.
- **rotation** (`Matrix` | `Quaternion` | `Euler` | None) – The rotation component as a 3x3 matrix, quaternion, euler or None for nc

rotation.

- **scale** (`Vector` | None) – The scale component.

RETURNS:

Combined transformation as a 4x4 matrix.

RETURN TYPE:

`Matrix`

```
# Compute local object transformation matrix:
if obj.rotation_mode == 'QUATERNION':
    matrix = mathutils.Matrix.LocRotScale(obj.location, obj.rotation_quaternion, obj
else:
    matrix = mathutils.Matrix.LocRotScale(obj.location, obj.rotation_euler, obj.scal
```

classmethod OrthoProjection(axis, size)

Create a matrix to represent an orthographic projection.

PARAMETERS:

- **axis** (str | `Vector`) – Can be any of the following: ['X', 'Y', 'XY', 'XZ', 'YZ'], where a single axis is for a 2D matrix. Or a vector fo an arbitrary axis
- **size** (*int*) – The size of the projection matrix to construct [2, 4].

RETURNS:

A new projection matrix.

RETURN TYPE:

`Matrix`

classmethod Rotation(angle, size, axis)

Create a matrix representing a rotation.

PARAMETERS:

- **angle** (*float*) – The angle of rotation desired, in radians.
- **size** (*int*) – The size of the rotation matrix to construct [2, 4].
- **axis** (str | `Vector`) – a string in ['X', 'Y', 'Z'] or a 3D Vector Object (optional when size is 2).

RETURNS:

A new rotation matrix.

RETURN TYPE:

`Matrix`

classmethod Scale(factor, size, axis)

Create a matrix representing a scaling.

PARAMETERS:

- **factor** (*float*) – The factor of scaling to apply.
- **size** (*int*) – The size of the scale matrix to construct [2, 4].
- **axis** (`Vector`) – Direction to influence scale. (optional).

RETURNS:

A new scale matrix.

RETURN TYPE:

`Matrix`

classmethod Shear(plane, size, factor)

Create a matrix to represent an shear transformation.

PARAMETERS:

- **plane** (*str*) – Can be any of the following: ['X', 'Y', 'XY', 'XZ', 'YZ'], where a single axis is for a 2D matrix only.
- **size** (*int*) – The size of the shear matrix to construct [2, 4].
- **factor** (*float* | *Sequence[float]*) – The factor of shear to apply. For a 2 *size* matrix use a single float. For a 3 or 4 *size* matrix pass a pa of floats corresponding with the *plane* axis.

RETURNS:

A new shear matrix.

RETURN TYPE:

`Matrix`

classmethod Translation(vector)

Create a matrix representing a translation.

PARAMETERS:

vector (`Vector`) – The translation vector.

RETURNS:

An identity matrix with a translation.

RETURN TYPE:

`Matrix`

adjugate()

Set the matrix to its adjugate.

RAISES:

ValueError – if the matrix cannot be adjugate.

See also

Adjugate matrix on Wikipedia.

adjugated()

Return an adjugated copy of the matrix.

RETURNS:

the adjugated matrix.

RETURN TYPE:

`Matrix`

RAISES:

ValueError – if the matrix cannot be adjugated

copy()

Returns a copy of this matrix.

RETURNS:

an instance of itself

RETURN TYPE:

`Matrix`

decompose()

Return the translation, rotation, and scale components of this matrix.

RETURNS:

RETURNS:

Tuple of translation, rotation, and scale.

RETURN TYPE:

tuple[`Vector`, `Quaternion`, `Vector`]

determinant()

Return the determinant of a matrix.

RETURNS:

Return the determinant of a matrix.

RETURN TYPE:

float

See also

Determinant on Wikipedia.

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

identity()

Set the matrix to the identity matrix.

Note

An object with a location and rotation of zero, and a scale of one will have an identity matrix.

See also

Identity matrix on Wikipedia.

invert(fallback=None)

Set the matrix to its inverse.

PARAMETERS:

fallback (`Matrix`) – Set the matrix to this value when the inverse cannot be calculated (instead of raising a `ValueError` exception).

See also

Inverse matrix on Wikipedia.

invert_safe()

Set the matrix to its inverse, will never error. If degenerated (e.g. zero scale on an axis), add some epsilon to its diagonal, to get an invertible one. If tweaked matrix is still degenerated, set to the identity matrix instead.

See also

Inverse Matrix on Wikipedia.

inverted(fallback=None)

Return an inverted copy of the matrix.

PARAMETERS:

PARAMETERS:

fallback (*Any*) – return this when the inverse can't be calculated (instead of raising a `ValueError`).

RETURNS:

The inverted matrix or fallback when given.

RETURN TYPE:

`Matrix` | Any

inverted_safe()

Return an inverted copy of the matrix, will never error. If degenerated (e.g. zero scale on an axis), add some epsilon to its diagonal, to get an invertible one. If tweaked matrix is still degenerated, return the identity matrix instead.

RETURNS:

the inverted matrix.

RETURN TYPE:

`Matrix`

lerp(other, factor)

Returns the interpolation of two matrices. Uses polar decomposition, see "Matrix Animation and Polar Decomposition", Shoemake and Duff, 1992.

PARAMETERS:

- **other** (`Matrix`) – value to interpolate with.
- **factor** (*float*) – The interpolation value in [0.0, 1.0].

RETURNS:

The interpolated matrix.

RETURN TYPE:

`Matrix`

normalize()

Normalize each of the matrix columns.

> Note
>
> for 4x4 matrices, the 4th column (translation) is left untouched.

normalized()

Return a column normalized matrix

RETURNS:

a column normalized matrix

RETURN TYPE:

`Matrix`

> Note
>
> for 4x4 matrices, the 4th column (translation) is left untouched.

resize_4x4()

Resize the matrix to 4x4.

rotate(other)

Rotates the matrix by another mathutils value.

PARAMETERS:

other (`Euler` | `Quaternion` | `Matrix`) – rotation component of mathutils value

Note

If any of the columns are not unit length this may not have desired results.

to_2x2()

Return a 2x2 copy of this matrix.

RETURNS:

a new matrix.

RETURN TYPE:

`Matrix`

to_3x3()

Return a 3x3 copy of this matrix.

RETURNS:

a new matrix.

RETURN TYPE:

`Matrix`

to_4x4()

Return a 4x4 copy of this matrix.

RETURNS:

a new matrix.

RETURN TYPE:

`Matrix`

to_euler(order, euler_compat)

Return an Euler representation of the rotation matrix (3x3 or 4x4 matrix only).

PARAMETERS:

- **order** (*str*) – Optional rotation order argument in [‘XYZ’, ‘XZY’, ‘YXZ’, ‘YZX’, ‘ZXY’, ‘ZYX’].
- **euler_compat** (`Euler`) – Optional euler argument the new euler will be made compatible with (no axis flipping between them). Useful for converting a series of matrices to animation curves.

RETURNS:

Euler representation of the matrix.

RETURN TYPE:

`Euler`

to_quaternion()

Return a quaternion representation of the rotation matrix.

RETURNS:

Quaternion representation of the rotation matrix.

RETURN TYPE:

`Quaternion`

to_scale()

Return the scale part of a 3x3 or 4x4 matrix.

RETURNS:

Return the scale of a matrix.

RETURN TYPE:

`Vector`

Note

This method does not return a negative scale on any axis because it is not possible to obtain this data from the matrix alone.

to_translation()

Return the translation part of a 4 row matrix.

RETURNS:

Return the translation of a matrix.

RETURN TYPE:

`Vector`

transpose()

Set the matrix to its transpose.

See also

Transpose on Wikipedia.

transposed()

Return a new, transposed matrix.

RETURNS:

a transposed matrix

RETURN TYPE:

`Matrix`

zero()

Set all the matrix values to zero.

col

Access the matrix by columns, 3x3 and 4x4 only, (read-only).

TYPE:

Matrix Access

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_identity

True if this is an identity matrix (read-only).

TYPE:

bool

is_negative

True if this matrix results in a negative scale, 3x3 and 4x4 only, (read-only).

TYPE:

bool

bool

is_orthogonal

True if this matrix is orthogonal, 3x3 and 4x4 only, (read-only).

TYPE:

bool

is_orthogonal_axis_vectors

True if this matrix has got orthogonal axis vectors, 3x3 and 4x4 only, (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

median_scale

The average scale applied to each axis (read-only).

TYPE:

float

owner

The item this is wrapping or None (read-only).

row

Access the matrix by rows (default), (read-only).

TYPE:

Matrix Access

translation

The translation component of the matrix.

TYPE:

`Vector`

class mathutils.**Quaternion([seq[, angle]])**

This object gives access to Quaternions in Blender.

PARAMETERS:

- **seq** (`Vector`) – size 3 or 4
- **angle** (*float*) – rotation angle, in radians

The constructor takes arguments in various forms:

(), *no args*

Create an identity quaternion

(*wxyz*)

Create a quaternion from a `(w, x, y, z)` vector.

(*exponential_map*)

Create a quaternion from a 3d exponential map vector.

See also

`to_exponential_map()`

(*axis, angle*)

Create a quaternion representing a rotation of *angle* radians over *axis*.

See also

`to_axis_angle()`

```
import mathutils
import math

# a new rotation 90 degrees about the Y axis
quat_a = mathutils.Quaternion((0.7071068, 0.0, 0.7071068, 0.0))

# passing values to Quaternion's directly can be confusing so axis, angle
# is supported for initializing too
quat_b = mathutils.Quaternion((0.0, 1.0, 0.0), math.radians(90.0))

print("Check quaternions match", quat_a == quat_b)

# like matrices, quaternions can be multiplied to accumulate rotational values
quat_a = mathutils.Quaternion((0.0, 1.0, 0.0), math.radians(90.0))
quat_b = mathutils.Quaternion((0.0, 0.0, 1.0), math.radians(45.0))
quat_out = quat_a @ quat_b

# print the quat, euler degrees for mere mortals and (axis, angle)
print("Final Rotation:")
print(quat_out)
print("{:.2f}, {:.2f}, {:.2f}".format(*(math.degrees(a) for a in quat_out.to_euler())))
print("({:.2f}, {:.2f}, {:.2f}), {:.2f}".format(*quat_out.axis, math.degrees(quat_out.a

# multiple rotations can be interpolated using the exponential map
quat_c = mathutils.Quaternion((1.0, 0.0, 0.0), math.radians(15.0))
exp_avg = (quat_a.to_exponential_map() +
           quat_b.to_exponential_map() +
           quat_c.to_exponential_map()) / 3.0
quat_avg = mathutils.Quaternion(exp_avg)
print("Average rotation:")
print(quat_avg)
```

conjugate()

Set the quaternion to its conjugate (negate x, y, z).

conjugated()

Return a new conjugated quaternion.

RETURNS:

a new quaternion.

RETURN TYPE:

`Quaternion`

copy()

Returns a copy of this quaternion.

RETURNS:

A copy of the quaternion.

RETURN TYPE:

`Quaternion`

Note

use this to get a copy of a wrapped quaternion with no reference to the original data.

cross(other)

Return the cross product of this quaternion and another.

PARAMETERS:

other (`Quaternion`) – The other quaternion to perform the cross product with.

RETURNS:

The cross product.

RETURN TYPE:

`Quaternion`

dot(other)

Return the dot product of this quaternion and another.

PARAMETERS:

other (`Quaternion`) – The other quaternion to perform the dot product with.

RETURNS:

The dot product.

RETURN TYPE:

float

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

identity()

Set the quaternion to an identity quaternion.

invert()

Set the quaternion to its inverse.

inverted()

Return a new, inverted quaternion.

RETURNS:

the inverted value.

RETURN TYPE:

Quaternion

make_compatible(other)

Make this quaternion compatible with another, so interpolating between them works as intended.

negate()

Set the quaternion to its negative.

normalize()

Normalize the quaternion.

normalized()

Return a new normalized quaternion.

RETURNS:

a normalized copy.

RETURN TYPE:

Quaternion

rotate(other)

Rotates the quaternion by another mathutils value.

PARAMETERS:

other (Euler | Quaternion | Matrix) – rotation component of mathutils value

rotation_difference(other)

Returns a quaternion representing the rotational difference.

PARAMETERS:

other (Quaternion) – second quaternion.

RETURNS:

the rotational difference between the two quat rotations.

RETURN TYPE:

Quaternion

slerp(other, factor)

Returns the interpolation of two quaternions.

PARAMETERS:

- **other** (Quaternion) – value to interpolate with.
- **factor** (*float*) – The interpolation value in [0.0, 1.0].

RETURNS:

The interpolated rotation.

RETURN TYPE:

Quaternion

to_axis_angle()

Return the axis, angle representation of the quaternion.

RETURNS:

Axis, angle.

RETURN TYPE:

RETURN TYPE:

tuple[`Vector`, float]

to_euler(order, euler_compat)

Return Euler representation of the quaternion.

PARAMETERS:

- **order** (*str*) – Optional rotation order argument in [‘XYZ’, ‘XZY’, ‘YXZ’, ‘YZX’, ‘ZXY’, ‘ZYX’].
- **euler_compat** (`Euler`) – Optional euler argument the new euler will be made compatible with (no axis flipping between them). Usefu for converting a series of matrices to animation curves.

RETURNS:

Euler representation of the quaternion.

RETURN TYPE:

`Euler`

to_exponential_map()

Return the exponential map representation of the quaternion.

This representation consist of the rotation axis multiplied by the rotation angle. Such a representation is useful for interpolation between multipl orientations.

RETURNS:

exponential map.

RETURN TYPE:

`Vector` of size 3

To convert back to a quaternion, pass it to the `Quaternion` constructor.

to_matrix()

Return a matrix representation of the quaternion.

RETURNS:

A 3x3 rotation matrix representation of the quaternion.

RETURN TYPE:

`Matrix`

to_swing_twist(axis)

Split the rotation into a swing quaternion with the specified axis fixed at zero, and the remaining twist rotation angle.

PARAMETERS:

axis (*str*) – Twist axis as a string in [‘X’, ‘Y’, ‘Z’].

RETURNS:

Swing, twist angle.

RETURN TYPE:

tuple[`Quaternion`, float]

angle

Angle of the quaternion.

TYPE:

float

axis

Quaternion axis as a vector.

TYPE:

TYPE:

Vector

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

magnitude

Size of the quaternion (read-only).

TYPE:

float

owner

The item this is wrapping or None (read-only).

w

Quaternion axis value.

TYPE:

float

x

Quaternion axis value.

TYPE:

float

y

Quaternion axis value.

TYPE:

float

z

Quaternion axis value.

TYPE:

float

class mathutils.**Vector(seq)**

This object gives access to Vectors in Blender.

PARAMETERS:

seq (*Sequence[float]*) – Components of the vector, must be a sequence of at least two.

```
import mathutils

# zero length vector
vec = mathutils.Vector((0.0, 0.0, 1.0))

# unit length vector
vec_a = vec.normalized()

vec_b = mathutils.Vector((0.0, 1.0, 2.0))

vec2d = mathutils.Vector((1.0, 2.0))
vec3d = mathutils.Vector((1.0, 0.0, 0.0))
vec4d = vec_a.to_4d()

# Other `mathutils` types.
quat = mathutils.Quaternion()
matrix = mathutils.Matrix()

# Comparison operators can be done on Vector classes:

# (In)equality operators == and != test component values, e.g. 1,2,3 != 3,2,1
vec_a == vec_b
vec_a != vec_b

# Ordering operators >, >=, > and <= test vector length.
vec_a > vec_b
vec_a >= vec_b
vec_a < vec_b
vec_a <= vec_b


# Math can be performed on Vector classes
vec_a + vec_b
vec_a - vec_b
vec_a @ vec_b
vec_a * 10.0
matrix @ vec_a
quat @ vec_a
-vec_a


# You can access a vector object like a sequence
x = vec_a[0]
len(vec)
vec_a[:] = vec_b
vec_a[:] = 1.0, 2.0, 3.0
vec2d[:] = vec3d[:2]


# Vectors support 'swizzle' operations
# See https://en.wikipedia.org/wiki/Swizzling_(computer_graphics)
vec.xyz = vec.zyx
vec.xy = vec4d.zw
```

```
vec.xyz = vec4d.wzz
vec4d.wxyz = vec.yxyx
```

classmethod Fill(size, fill=0.0)

Create a vector of length size with all values set to fill.

PARAMETERS:

- **size** (*int*) – The length of the vector to be created.
- **fill** (*float*) – The value used to fill the vector.

classmethod Linspace(start, stop, size)

Create a vector of the specified size which is filled with linearly spaced values between start and stop values.

PARAMETERS:

- **start** (*int*) – The start of the range used to fill the vector.
- **stop** (*int*) – The end of the range used to fill the vector.
- **size** (*int*) – The size of the vector to be created.

classmethod Range(start, stop, step=1)

Create a filled with a range of values.

PARAMETERS:

- **start** (*int*) – The start of the range used to fill the vector.
- **stop** (*int*) – The end of the range used to fill the vector.
- **step** (*int*) – The step between successive values in the vector.

classmethod Repeat(vector, size)

Create a vector by repeating the values in vector until the required size is reached.

PARAMETERS:

- **vector** (`mathutils.Vector`) – The vector to draw values from.
- **size** (*int*) – The size of the vector to be created.

angle(other, fallback=None)

Return the angle between two vectors.

PARAMETERS:

- **other** (`Vector`) – another vector to compare the angle with
- **fallback** (*Any*) – return this when the angle can't be calculated (zero length vector), (instead of raising a `ValueError`).

RETURNS:

angle in radians or fallback when given

RETURN TYPE:

float | Any

angle_signed(other, fallback=None)

Return the signed angle between two 2D vectors (clockwise is positive).

PARAMETERS:

- **other** (`Vector`) – another vector to compare the angle with
- **fallback** (*Any*) – return this when the angle can't be calculated (zero length vector), (instead of raising a `ValueError`).

RETURNS:

angle in radians or fallback when given

RETURN TYPE:

float | Any

copy()

Returns a copy of this vector.

RETURNS:

A copy of the vector.

RETURN TYPE:

`Vector`

Note

use this to get a copy of a wrapped vector with no reference to the original data.

cross(other)

Return the cross product of this vector and another.

PARAMETERS:

other (`Vector`) – The other vector to perform the cross product with.

RETURNS:

The cross product as a vector or a float when 2D vectors are used.

RETURN TYPE:

`Vector` | float

Note

both vectors must be 2D or 3D

dot(other)

Return the dot product of this vector and another.

PARAMETERS:

other (`Vector`) – The other vector to perform the dot product with.

RETURNS:

The dot product.

RETURN TYPE:

float

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

lerp(other, factor)

Returns the interpolation of two vectors.

PARAMETERS:

- **other** (`Vector`) – value to interpolate with.
- **factor** (*float*) – The interpolation value in [0.0, 1.0].

RETURNS:

The interpolated vector.

RETURN TYPE:

`Vector`

negate()

Set all values to their negative.

normalize()

Normalize the vector, making the length of the vector always 1.0.

> Warning
>
> Normalizing a vector where all values are zero has no effect.

> Note
>
> Normalize works for vectors of all sizes, however 4D Vectors w axis is left untouched.

normalized()

Return a new, normalized vector.

RETURNS:

a normalized copy of the vector

RETURN TYPE:

`Vector`

orthogonal()

Return a perpendicular vector.

RETURNS:

a new vector 90 degrees from this vector.

RETURN TYPE:

`Vector`

> Note
>
> the axis is undefined, only use when any orthogonal vector is acceptable.

project(other)

Return the projection of this vector onto the *other*.

PARAMETERS:

other (`Vector`) – second vector.

RETURNS:

the parallel projection vector

RETURN TYPE:

`Vector`

reflect(mirror)

Return the reflection vector from the *mirror* argument.

PARAMETERS:

mirror (`Vector`) – This vector could be a normal from the reflecting surface.

RETURNS:

The reflected vector matching the size of this vector.

RETURN TYPE:

RETURN TYPE:

`Vector`

resize(size=3)

Resize the vector to have size number of elements.

resize_2d()

Resize the vector to 2D (x, y).

resize_3d()

Resize the vector to 3D (x, y, z).

resize_4d()

Resize the vector to 4D (x, y, z, w).

resized(size=3)

Return a resized copy of the vector with size number of elements.

RETURNS:

a new vector

RETURN TYPE:

`Vector`

rotate(other)

Rotate the vector by a rotation value.

Note

2D vectors are a special case that can only be rotated by a 2x2 matrix.

PARAMETERS:

other (`Euler` | `Quaternion` | `Matrix`) – rotation component of mathutils value

rotation_difference(other)

Returns a quaternion representing the rotational difference between this vector and another.

PARAMETERS:

other (`Vector`) – second vector.

RETURNS:

the rotational difference between the two vectors.

RETURN TYPE:

`Quaternion`

Note

2D vectors raise an `AttributeError`.

slerp(other, factor, fallback=None)

Returns the interpolation of two non-zero vectors (spherical coordinates).

PARAMETERS:

- **other** (`Vector`) – value to interpolate with.
- **factor** (*float*) – The interpolation value typically in [0.0, 1.0].
- **fallback** (*Any*) – return this when the vector can't be calculated (zero length vector or direct opposites), (instead of raising a `ValueError`).

RETURNS:

The interpolated vector.

RETURN TYPE:

Vector

to_2d()

Return a 2d copy of the vector.

RETURNS:

a new vector

RETURN TYPE:

Vector

to_3d()

Return a 3d copy of the vector.

RETURNS:

a new vector

RETURN TYPE:

Vector

to_4d()

Return a 4d copy of the vector.

RETURNS:

a new vector

RETURN TYPE:

Vector

to_track_quat(track, up)

Return a quaternion rotation from the vector and the track and up axis.

PARAMETERS:

- **track** (*str*) – Track axis in ['X', 'Y', 'Z', '-X', '-Y', '-Z'].
- **up** (*str*) – Up axis in ['X', 'Y', 'Z'].

RETURNS:

rotation from the vector and the track and up axis.

RETURN TYPE:

Quaternion

to_tuple(precision=-1)

Return this vector as a tuple with.

PARAMETERS:

precision (*int*) – The number to round the value to in [-1, 21].

RETURNS:

the values of the vector rounded by *precision*

RETURN TYPE:

tuple[float, …]

zero()

Set all values to zero.

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

length

Vector Length.

TYPE:

float

length_squared

Vector length squared (v.dot(v)).

TYPE:

float

magnitude

Vector Length.

TYPE:

float

owner

The item this is wrapping or None (read-only).

w

Vector W axis (4D Vectors only).

TYPE:

float

ww

TYPE:

Vector

www

TYPE:

Vector

wwww

TYPE:

Vector

wwwx

TYPE:

Vector

wwwy

TYPE:

Vector

wwwz

TYPE:

Vector

wwx

TYPE:

Vector

wwxw

TYPE:

Vector

wwxx

TYPE:

Vector

wwxy

TYPE:

Vector

wwxz

TYPE:

Vector

wwy

TYPE:

Vector

wwyw

TYPE:

Vector

wwyx

TYPE:

Vector

wwyy

TYPE:

Vector

wwyz

TYPE:

Vector

---

wwz

TYPE:

Vector

wwzw

TYPE:

Vector

wwzx

TYPE:

Vector

wwzy

TYPE:

Vector

wwzz

TYPE:

Vector

wx

TYPE:

Vector

wxw

TYPE:

Vector

wxww

TYPE:

Vector

wxwx

TYPE:

Vector

wxwy

TYPE:

Vector

wxwz

TYPE:

Vector

wxx

TYPE:

Vector

wxxw

TYPE:

Vector

wxxx

TYPE:

Vector

wxxy

TYPE:

Vector

wxxz

TYPE:

Vector

wxy

TYPE:

Vector

wxyw

TYPE:

Vector

wxyx

TYPE:

Vector

wxyy

TYPE:

Vector

wxyz

TYPE:

Vector

wxz

TYPE:

Vector

wxzw

TYPE:

Vector

wxzx

TYPE:

Vector

wxzy

TYPE:

Vector

wxzz

TYPE:

Vector

wy

TYPE:

vector

wyw

TYPE:

Vector

wyww

TYPE:

Vector

wywx

TYPE:

Vector

wywy

TYPE:

Vector

wywz

TYPE:

Vector

wyx

TYPE:

Vector

wyxw

TYPE:

Vector

wyxx

TYPE:

Vector

wyxy

TYPE:

Vector

wyxz

TYPE:

Vector

wyy

TYPE:

Vector

wyyw

TYPE:

Vector

wyyx

TYPE:

Vector

wyyy

TYPE:

Vector

wyyz

TYPE:

Vector

wyz

TYPE:

Vector

wyzw

TYPE:

Vector

wyzx

TYPE:

Vector

wyzy

TYPE:

Vector

wyzz

TYPE:

Vector

wz

TYPE:

Vector

wzw

TYPE:

Vector

wzww

TYPE:

Vector

wzwx

TYPE:

Vector

wzwy

TYPE:

Vector

wzwz

TYPE:

Vector

wzx

TYPE:

Vector

wzxw

TYPE:

Vector

wzxx

TYPE:

Vector

wzxy

TYPE:

Vector

wzxz

TYPE:

Vector

wzy

TYPE:

Vector

wzyw

TYPE:

Vector

wzyx

TYPE:

Vector

wzyy

TYPE:

Vector

wzyz

TYPE:

Vector

wzz

TYPE:

Vector

wzzw

TYPE:

Vector

wzzx

TYPE:

Vector

wzzy

TYPE:

Vector

wzzz

TYPE:

Vector

x

Vector X axis.

TYPE:

float

xw

TYPE:

Vector

xww

TYPE:

Vector

xwww

TYPE:

Vector

xwwx

TYPE:

Vector

xwwy

TYPE:

Vector

xwwz

TYPE:

Vector

xwx

TYPE:

Vector

xwxw

TYPE:

Vector

xwxx

TYPE:

Vector

xwxy

TYPE:

Vector

xwxz

TYPE:

TYPE:

Vector

xwy

TYPE:

Vector

xwyw

TYPE:

Vector

xwyx

TYPE:

Vector

xwyy

TYPE:

Vector

xwyz

TYPE:

Vector

xwz

TYPE:

Vector

xwzw

TYPE:

Vector

xwzx

TYPE:

Vector

xwzy

TYPE:

Vector

xwzz

TYPE:

Vector

xx

TYPE:

Vector

xxw

TYPE:

Vector

xxww

TYPE:

Vector

Vector

xxwx

TYPE:

Vector

xxwy

TYPE:

Vector

xxwz

TYPE:

Vector

xxx

TYPE:

Vector

xxxw

TYPE:

Vector

xxxx

TYPE:

Vector

xxxy

TYPE:

Vector

xxxz

TYPE:

Vector

xxy

TYPE:

Vector

xxyw

TYPE:

Vector

xxyx

TYPE:

Vector

xxyy

TYPE:

Vector

xxyz

TYPE:

Vector

xxz

TYPE:

Vector

xxzw

TYPE:

Vector

xxzx

TYPE:

Vector

xxzy

TYPE:

Vector

xxzz

TYPE:

Vector

xy

TYPE:

Vector

xyw

TYPE:

Vector

xyww

TYPE:

Vector

xywx

TYPE:

Vector

xywy

TYPE:

Vector

xywz

TYPE:

Vector

xyx

TYPE:

Vector

xyxw

TYPE:

Vector

xyxx

TYPE:

Vector

xyxy

TYPE:

Vector

xyxz

TYPE:

Vector

xyy

TYPE:

Vector

xyyw

TYPE:

Vector

xyyx

TYPE:

Vector

xyyy

TYPE:

Vector

xyyz

TYPE:

Vector

xyz

TYPE:

Vector

xyzw

TYPE:

Vector

xyzx

TYPE:

Vector

xyzy

TYPE:

Vector

xyzz

TYPE:

Vector

xz

TYPE:

Vector

xzw

TYPE:

Vector

xzww

TYPE:

Vector

xzwx

TYPE:

Vector

xzwy

TYPE:

Vector

xzwz

TYPE:

Vector

xzx

TYPE:

Vector

xzxw

TYPE:

Vector

xzxx

TYPE:

Vector

xzxy

TYPE:

Vector

xzxz

TYPE:

Vector

xzy

TYPE:

Vector

xzyw

TYPE:

Vector

xzyx

TYPE:

Vector

xzyy

TYPE:

Vector

xzyz

TYPE:

Vector

xzz

TYPE:

Vector

xzzw

TYPE:

Vector

xzzx

TYPE:

Vector

xzzy

TYPE:

Vector

xzzz

TYPE:

Vector

y

Vector Y axis.

TYPE:

float

yw

TYPE:

Vector

yww

TYPE:

Vector

ywww

TYPE:

Vector

ywwx

TYPE:

Vector

ywwy

TYPE:

Vector

ywwz

TYPE:

Vector

ywx

TYPE:

Vector

ywxw

TYPE:

Vector

ywxx

TYPE:

Vector

ywxy

TYPE:

Vector

ywxz

TYPE:

Vector

ywy

TYPE:

Vector

ywyw

TYPE:

Vector

ywyx

TYPE:

Vector

ywyy

TYPE:

Vector

ywyz

TYPE:

Vector

ywz

TYPE:

Vector

ywzw

TYPE:

Vector

ywzx

TYPE:

Vector

ywzy

TYPE:

Vector

ywzz

TYPE:

Vector

yx

TYPE:

Vector

yxw

TYPE:

Vector

yxww

TYPE:

Vector

yxwx

TYPE:

Vector

yxwy

TYPE:

Vector

yxwz

TYPE:

Vector

yxx

TYPE:

Vector

yxxw

TYPE:

Vector

yxxx

TYPE:

Vector

yxxy

TYPE:

Vector

yxxz

TYPE:

Vector

yxy

TYPE:

Vector

yxyw

TYPE:

Vector

yxyx

TYPE:

Vector

yxyy

TYPE:

Vector

yxyz

TYPE:

Vector

yxz

TYPE:

Vector

yxzw

TYPE:

Vector

yxzx

TYPE:

Vector

yxzy

TYPE:

Vector

yxzz

TYPE:

Vector

yy

TYPE:

Vector

yyw

TYPE:

Vector

yyww

TYPE:

Vector

yywx

TYPE:

Vector

yywy

TYPE:

Vector

yywz

TYPE:

Vector

yyx

TYPE:

Vector

yyxw

TYPE:

Vector

yyxx

TYPE:

Vector

yyxy

TYPE:

Vector

yyxz

TYPE:

Vector

yyy

TYPE:

Vector

yyyw

TYPE:

Vector

yyyx

TYPE:

Vector

yyyy

TYPE:

Vector

yyyz

TYPE:

Vector

yyz

TYPE:

Vector

yyzw

TYPE:

Vector

yyzx

TYPE:

Vector

yyzy

TYPE:

Vector

yyzz

TYPE:

Vector

yz

TYPE:

Vector

yzw

TYPE:

Vector

yzww

TYPE:

Vector

yzwx

TYPE:

Vector

yzwy

TYPE:

Vector

yzwz

TYPE:

Vector

yzx

TYPE:

Vector

yzxw

TYPE:

Vector

yzxx

TYPE:

Vector

yzxy

TYPE:

Vector

yzxz

TYPE:

Vector

yzy

TYPE:

Vector

yzyw

TYPE:

Vector

yzyx

TYPE:

Vector

yzyy

TYPE:

Vector

yzyz

TYPE:

Vector

yzz

TYPE:

Vector

yzzw

TYPE:

Vector

yzzx

TYPE:

Vector

yzzy

TYPE:

Vector

yzzz

TYPE:

Vector

z

Vector Z axis (3D Vectors only).

TYPE:

float

zw

TYPE:

Vector

zww

TYPE:

Vector

zwww

TYPE:

Vector

zwwx

TYPE:

Vector

zwwy

TYPE:

Vector

zwwz

TYPE:

Vector

zwx

TYPE:

Vector

zwxw

TYPE:

Vector

zwxx

TYPE:

Vector

zwxy

TYPE:

Vector

zwxz

TYPE:

Vector

zwy

TYPE:

Vector

zwyw

TYPE:

Vector

zwyx

TYPE:

Vector

zwyy

TYPE:

Vector

zwyz

TYPE:

Vector

zwz

TYPE:

Vector

zwzw

TYPE:

Vector

zwzx

TYPE:

Vector

zwzy

TYPE:

Vector

zwzz

TYPE:

Vector

zx

TYPE:

Vector

zxw

TYPE:

Vector

zxww

TYPE:

Vector

zxwx

TYPE:

Vector

zxwy

TYPE:

vector

zxwz

TYPE:

Vector

zxx

TYPE:

Vector

zxxw

TYPE:

Vector

zxxx

TYPE:

Vector

zxxy

TYPE:

Vector

zxxz

TYPE:

Vector

zxy

TYPE:

Vector

zxyw

TYPE:

Vector

zxyx

TYPE:

Vector

zxyy

TYPE:

Vector

zxyz

TYPE:

Vector

zxz

TYPE:

Vector

zxzw

TYPE:

Vector

zxzx

TYPE:

Vector

zxzy

TYPE:

Vector

zxzz

TYPE:

Vector

zy

TYPE:

Vector

zyw

TYPE:

Vector

zyww

TYPE:

Vector

zywx

TYPE:

Vector

zywy

TYPE:

Vector

zywz

TYPE:

Vector

zyx

TYPE:

Vector

zyxw

TYPE:

Vector

zyxx

TYPE:

Vector

zyxy

TYPE:

Vector

zyxz

TYPE:

Vector

zyy

TYPE:

Vector

zyyw

TYPE:

Vector

zyyx

TYPE:

Vector

zyyy

TYPE:

Vector

zyyz

TYPE:

Vector

zyz

TYPE:

Vector

zyzw

TYPE:

Vector

zyzx

TYPE:

Vector

zyzy

TYPE:

Vector

zyzz

TYPE:

Vector

zz

TYPE:

Vector

zzw

TYPE:

Vector

zzww

TYPE:

Vector

zzwx

TYPE:

Vector

zzwy

TYPE:

Vector

zzwz

TYPE:

Vector

zzx

TYPE:

Vector

zzxw

TYPE:

Vector

zzxx

TYPE:

Vector

zzxy

TYPE:

Vector

zzxz

TYPE:

Vector

zzy

TYPE:

Vector

zzyw

TYPE:

Vector

zzyx

TYPE:

Vector

zzyy

TYPE:

Vector

zzyz

TYPE:

Vector

zzz

TYPE:

Vector

zzzw

TYPE:

Vector

zzzx

TYPE:

Vector

zzzy

TYPE:

Vector

zzzz

TYPE:

Vector

Interpolation Utilities (mathutils.interpolate)

The Blender interpolate module

mathutils.interpolate.**poly_3d_calc(veclist, pt)**

Calculate barycentric weights for a point on a polygon.

PARAMETERS:

- **veclist** (*Sequence[Sequence[float]]*) – Sequence of 3D positions.
- **pt** – 2D or 3D position. :type pt: Sequence[float] :return: list of per-vector weights.

RETURN TYPE:

list[float]

Skip to content

KDTree Utilities (mathutils.kdtree)

Generic 3-dimensional kd-tree to perform spatial searches.

```
import mathutils

# create a kd-tree from a mesh
from bpy import context
obj = context.object

mesh = obj.data
size = len(mesh.vertices)
kd = mathutils.kdtree.KDTree(size)

for i, v in enumerate(mesh.vertices):
    kd.insert(v.co, i)

kd.balance()


# Find the closest point to the center
co_find = (0.0, 0.0, 0.0)
co, index, dist = kd.find(co_find)
print("Close to center:", co, index, dist)

# 3d cursor relative to the object data
co_find = obj.matrix_world.inverted() @ context.scene.cursor.location

# Find the closest 10 points to the 3d cursor
print("Close 10 points")
for (co, index, dist) in kd.find_n(co_find, 10):
    print("    ", co, index, dist)


# Find points within a radius of the 3d cursor
print("Close points within 0.5 distance")
for (co, index, dist) in kd.find_range(co_find, 0.5):
    print("    ", co, index, dist)
```

class mathutils.kdtree.**KDTree**

KdTree(size) -> new kd-tree initialized to hold `size` items.

> Note
>
> `KDTree.balance` must have been called before using any of the `find` methods.

balance()

Balance the tree.

> Note
>
> This builds the entire tree, avoid calling after each insertion.

find(co, filter=None)

Find nearest point to `co`.

PARAMETERS:

- **co** (*Sequence[float]*) – 3D coordinates.
- **filter** (*Callable[[int], bool]*) – function which takes an index and returns True for indices to include in the search.

RETURNS:

Returns (position, index, distance).

RETURN TYPE:

tuple[`Vector`, int, float]

find_n(co, n)

Find nearest `n` points to `co`.

PARAMETERS:

- **co** (*Sequence[float]*) – 3D coordinates.
- **n** (*int*) – Number of points to find.

RETURNS:

Returns a list of tuples (position, index, distance).

RETURN TYPE:

list[tuple[`Vector`, int, float]]

find_range(co, radius)

Find all points within `radius` of `co`.

PARAMETERS:

- **co** (*Sequence[float]*) – 3D coordinates.
- **radius** (*float*) – Distance to search for points.

RETURNS:

Returns a list of tuples (position, index, distance).

RETURN TYPE:

list[tuple[`Vector`, int, float]]

insert(co, index)

Insert a point into the KDTree.

PARAMETERS:

- **co** (*Sequence[float]*) – Point 3d position.
- **index** (*int*) – The index of the point.

Skip to content

Noise Utilities (mathutils.noise)

The Blender noise module

mathutils.noise.**cell(position)**

Returns cell noise value at the specified position.

PARAMETERS:

position (`mathutils.Vector`) – The position to evaluate the selected noise function.

RETURNS:

The cell noise value.

RETURN TYPE:

float

mathutils.noise.**cell_vector(position)**

Returns cell noise vector at the specified position.

PARAMETERS:

position (`mathutils.Vector`) – The position to evaluate the selected noise function.

RETURNS:

The cell noise vector.

RETURN TYPE:

`mathutils.Vector`

mathutils.noise.**fractal(position, H, lacunarity, octaves, noise_basis='PERLIN_ORIGINAL')**

Returns the fractal Brownian motion (fBm) noise value from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **H** (*float*) – The fractal increment factor.
- **lacunarity** (*float*) – The gap between successive frequencies.
- **octaves** (*int*) – The number of different noise frequencies used.
- **noise_basis** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’, ‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].

RETURNS:

The fractal Brownian motion noise value.

RETURN TYPE:

float

mathutils.noise.**hetero_terrain(position, H, lacunarity, octaves, offset, noise_basis='PERLIN_ORIGINAL')**

Returns the heterogeneous terrain value from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **H** (*float*) – The fractal dimension of the roughest areas.
- **lacunarity** (*float*) – The gap between successive frequencies.
- **octaves** (*int*) – The number of different noise frequencies used.
- **offset** (*float*) – The height of the terrain above ‘sea level’.
- **noise_basis** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’, ‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].

RETURNS:

The heterogeneous terrain value.

RETURN TYPE:

float

mathutils.noise.**hybrid_multi_fractal(position, H, lacunarity, octaves, offset, gain, noise_basis='PERLIN_ORIGINAL')**

Returns hybrid multifractal value from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **H** (*float*) – The fractal dimension of the roughest areas.
- **lacunarity** (*float*) – The gap between successive frequencies.
- **octaves** (*int*) – The number of different noise frequencies used.
- **offset** (*float*) – The height of the terrain above 'sea level'.
- **gain** (*float*) – Scaling applied to the values.
- **noise_basis** (*str*) – Enumerator in ['BLENDER', 'PERLIN_ORIGINAL', 'PERLIN_NEW', 'VORONOI_F1', 'VORONOI_F2', 'VORONOI_F3', 'VORONOI_F4', 'VORONOI_F2F1', 'VORONOI_CRACKLE', 'CELLNOISE'].

RETURNS:

The hybrid multifractal value.

RETURN TYPE:

float

mathutils.noise.**multi_fractal(position, H, lacunarity, octaves, noise_basis='PERLIN_ORIGINAL')**

Returns multifractal noise value from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **H** (*float*) – The fractal increment factor.
- **lacunarity** (*float*) – The gap between successive frequencies.
- **octaves** (*int*) – The number of different noise frequencies used.
- **noise_basis** (*str*) – Enumerator in ['BLENDER', 'PERLIN_ORIGINAL', 'PERLIN_NEW', 'VORONOI_F1', 'VORONOI_F2', 'VORONOI_F3', 'VORONOI_F4', 'VORONOI_F2F1', 'VORONOI_CRACKLE', 'CELLNOISE'].

RETURNS:

The multifractal noise value.

RETURN TYPE:

float

mathutils.noise.**noise(position, noise_basis='PERLIN_ORIGINAL')**

Returns noise value from the noise basis at the position specified.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **noise_basis** (*str*) – Enumerator in ['BLENDER', 'PERLIN_ORIGINAL', 'PERLIN_NEW', 'VORONOI_F1', 'VORONOI_F2', 'VORONOI_F3', 'VORONOI_F4', 'VORONOI_F2F1', 'VORONOI_CRACKLE', 'CELLNOISE'].

RETURNS:

The noise value.

RETURN TYPE:

float

mathutils.noise.**noise_vector(position, noise_basis='PERLIN_ORIGINAL')**

Returns the noise vector from the noise basis at the specified position.

Returns the noise vector from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **noise_basis** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’, ‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].

RETURNS:

The noise vector.

RETURN TYPE:

`mathutils.Vector`

mathutils.noise.**random()**

Returns a random number in the range [0, 1).

RETURNS:

The random number.

RETURN TYPE:

float

mathutils.noise.**random_unit_vector(size=3)**

Returns a unit vector with random entries.

PARAMETERS:

size (*int*) – The size of the vector to be produced, in the range [2, 4].

RETURNS:

The random unit vector.

RETURN TYPE:

`mathutils.Vector`

mathutils.noise.**random_vector(size=3)**

Returns a vector with random entries in the range (-1, 1).

PARAMETERS:

size (*int*) – The size of the vector to be produced.

RETURNS:

The random vector.

RETURN TYPE:

`mathutils.Vector`

mathutils.noise.**ridged_multi_fractal(position, H, lacunarity, octaves, offset, gain, noise_basis='PERLIN_ORIGINAL')**

Returns ridged multifractal value from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **H** (*float*) – The fractal dimension of the roughest areas.
- **lacunarity** (*float*) – The gap between successive frequencies.
- **octaves** (*int*) – The number of different noise frequencies used.
- **offset** (*float*) – The height of the terrain above ‘sea level’.
- **gain** (*float*) – Scaling applied to the values.
- **noise_basis** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’, ‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].

RETURNS:

The ridged multifractal value.

RETURN TYPE:

float

mathutils.noise.**seed_set(seed)**

Sets the random seed used for random_unit_vector, and random.

PARAMETERS:

seed (*int*) – Seed used for the random generator. When seed is zero, the current time will be used instead.

mathutils.noise.**turbulence(position, octaves, hard, noise_basis='PERLIN_ORIGINAL', amplitude_scale=0.5, frequency_scale=2.0)**

Returns the turbulence value from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **octaves** (*int*) – The number of different noise frequencies used.
- **hard** (*bool*) – Specifies whether returned turbulence is hard (sharp transitions) or soft (smooth transitions).
- **noise_basis** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’, ‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].
- **amplitude_scale** (*float*) – The amplitude scaling factor.
- **frequency_scale** (*float*) – The frequency scaling factor

RETURNS:

The turbulence value.

RETURN TYPE:

float

mathutils.noise.**turbulence_vector(position, octaves, hard, noise_basis='PERLIN_ORIGINAL', amplitude_scale=0.5, frequency_scale=2.0)**

Returns the turbulence vector from the noise basis at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **octaves** (*int*) – The number of different noise frequencies used.
- **hard** (*bool*) – Specifies whether returned turbulence is hard (sharp transitions) or soft (smooth transitions).
- **noise_basis** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’, ‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].
- **amplitude_scale** (*float*) – The amplitude scaling factor.
- **frequency_scale** (*float*) – The frequency scaling factor

RETURNS:

The turbulence vector.

RETURN TYPE:

`mathutils.Vector`

mathutils.noise.**variable_lacunarity(position, distortion, noise_type1='PERLIN_ORIGINAL', noise_type2='PERLIN_ORIGINAL')**

Returns variable lacunarity noise value, a distorted variety of noise, from noise type 1 distorted by noise type 2 at the specified position.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **distortion** (*float*) – The amount of distortion.
- **noise_type1** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’, ‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].
- **noise_type2** (*str*) – Enumerator in [‘BLENDER’, ‘PERLIN_ORIGINAL’, ‘PERLIN_NEW’, ‘VORONOI_F1’, ‘VORONOI_F2’,

‘VORONOI_F3’, ‘VORONOI_F4’, ‘VORONOI_F2F1’, ‘VORONOI_CRACKLE’, ‘CELLNOISE’].

RETURNS:

The variable lacunarity noise value.

RETURN TYPE:

float

mathutils.noise.**voronoi(position, distance_metric='DISTANCE', exponent=2.5)**

Returns a list of distances to the four closest features and their locations.

PARAMETERS:

- **position** (`mathutils.Vector`) – The position to evaluate the selected noise function.
- **distance_metric** (*str*) – Enumerator in [‘DISTANCE’, ‘DISTANCE_SQUARED’, ‘MANHATTAN’, ‘CHEBYCHEV’, ‘MINKOVSKY ‘MINKOVSKY_HALF’, ‘MINKOVSKY_FOUR’].
- **exponent** (*float*) – The exponent for Minkowski distance metric.

RETURNS:

A list of distances to the four closest features and their locations.

RETURN TYPE:

list[list[float] | list[`mathutils.Vector`]]

Previous
Interpolation Utilities (mathutils.interpolate)
Report issue on this page

Skip to content

Python Module Index

a | **b** | **f** | **g** | **i** | **m**

a

aud

b

bgl
bl_math
blf
bmesh
  bmesh.geometry
  bmesh.ops
  bmesh.types
  bmesh.utils
bpy
  bpy.app
  bpy.app.handlers
  bpy.app.icons
  bpy.app.timers
  bpy.app.translations
  bpy.context
  bpy.data
  bpy.msgbus
  bpy.ops
  bpy.ops.action
  bpy.ops.anim
  bpy.ops.armature
  bpy.ops.asset
  bpy.ops.boid
  bpy.ops.brush
  bpy.ops.buttons
  bpy.ops.cachefile
  bpy.ops.camera
  bpy.ops.clip
  bpy.ops.cloth
  bpy.ops.collection
  bpy.ops.console
  bpy.ops.constraint
  bpy.ops.curve
  bpy.ops.curves
  bpy.ops.cycles
  bpy.ops.dpaint
  bpy.ops.ed
  bpy.ops.export_anim
  bpy.ops.export_scene

bpy.ops.extensions
bpy.ops.file
bpy.ops.fluid
bpy.ops.font
bpy.ops.geometry
bpy.ops.gizmogroup
bpy.ops.gpencil
bpy.ops.graph
bpy.ops.grease_pencil
bpy.ops.image
bpy.ops.import_anim
bpy.ops.import_curve
bpy.ops.import_scene
bpy.ops.info
bpy.ops.lattice
bpy.ops.marker
bpy.ops.mask
bpy.ops.material
bpy.ops.mball
bpy.ops.mesh
bpy.ops.nla
bpy.ops.node
bpy.ops.object
bpy.ops.outliner
bpy.ops.paint
bpy.ops.paintcurve
bpy.ops.palette
bpy.ops.particle
bpy.ops.pose
bpy.ops.poselib
bpy.ops.preferences
bpy.ops.ptcache
bpy.ops.render
bpy.ops.rigidbody
bpy.ops.scene
bpy.ops.screen
bpy.ops.script
bpy.ops.sculpt
bpy.ops.sculpt_curves
bpy.ops.sequencer
bpy.ops.sound
bpy.ops.spreadsheet
bpy.ops.surface
bpy.ops.text
bpy.ops.text_editor
bpy.ops.texture
bpy.ops.transform
bpy.ops.ui
bpy.ops.uilist

f

g

i

idprop

- idprop.types

imbuf

- imbuf.types

m

mathutils

- mathutils.bvhtree
- mathutils.geometry
- mathutils.interpolate
- mathutils.kdtree
- mathutils.noise

Skip to content