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 tl 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-o 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.