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.