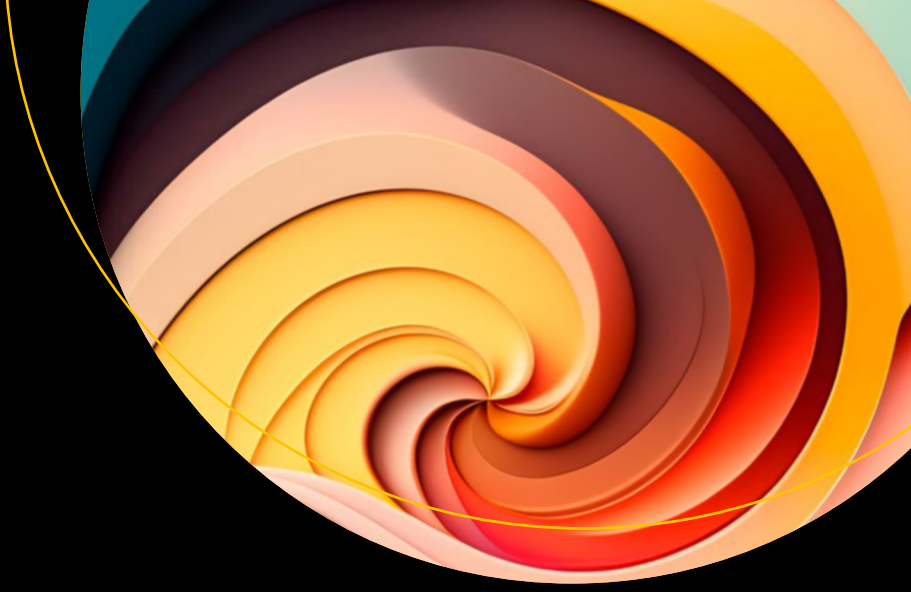

Blender Scripting with Python

Automate Tasks, Write Helper Tools, and Procedurally Generate Models in Blender 4

—

Isabel Lupiani

Apress®

Blender Scripting with Python

Automate Tasks, Write Helper Tools, and Procedurally Generate Models in Blender 4

Isabel Lupiani

Apress®

Blender Scripting with Python: Automate Tasks, Write Helper Tools, and Procedurally Generate Models in Blender 4

Isabel Lupiani
Orlando, FL, USA

ISBN-13 (pbk): 979-8-8688-1126-5 ISBN-13 (electronic): 979-8-8688-1127-2
https://doi.org/10.1007/979-8-8688-1127-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Editorial Project Manager: Kripa Joseph
Desk Editor: James Markham

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

For James, Zoe, and Caleb

Table of Contents

About the Author

Isabel Lupiani is a software engineer by day and a maker by night, who enjoys handcrafting 3D models as much as procedurally generating them. She received her MS in Computer Science from Georgia Tech and has worked at several game studios in the past as an AI engineer for PC/Xbox games. Most recently, Isabel was a lead AI engineer in the simulation industry.

About the Technical Reviewer

Ajit Deolikar is a mechanical engineer from Pune, India, and has experience in new product design and development. His work area involves designing aesthetic accessories for four wheelers and two wheelers (motorcycles as well as scooters), structural systems, power parts, bodywork, etc. He is also involved in designing and engineering of farm equipment and other testing equipment.

He is passionate about art and started using the open source software Blender as a hobby many years ago. He likes to create short animations as well as explanatory videos of various training modules for educational purposes.

He extended Blender experience in his professional work for preparing product styling surfaces and CAD construction using hard surface modeling. In addition, he has used it to solve complex mechanical motions using a physics engine and for various other tasks. With Blender Python scripting, he has created specific add-ons for simplifying and automating many repetitive tasks. He has even taken efforts to customize the project workflow for improving product quality to shorten delivery schedules. His product marketing videos and brochure, made using Blender, make him stand out from the competition.

In his spare time, he likes to play chess and analyze game strategies played by the world's great Grandmasters. He would someday love to write at least one book on those approaches. He can be reached at ajitb502@gmail.com.

Acknowledgments

I've always found the thought of anyone willing to read my work incredibly humbling and flattering. Thank you to all the readers out there, whether you've supported the previous incarnations of the book or are picking it up for the first time—the book exists (and lives on) because of you. I'd also like to express my gratitude to Spandana Chatterjee, Shobana Srinivasan, Kripa Joseph, James Markham, Joseph Quatela, Jessica Vakili, Sowmya Thodur, and the rest of the Apress team, for your support and guidance throughout the publishing process. Thank you to the technical reviewer for taking the time to read the chapters, dotting my Blender i's, and crossing my Python t's. Last but not least, thank you to my husband James, my daughter Zoe, and my son Caleb, for your eternal love and support, and not to forget Tiddles, the red-footed tortoise, for graciously modeling for the Chapter 8 photos.

Introduction

Art, games, and I go way back. My kindergarten teacher folded a paper crane in class one day, and I was hooked. By the time I was in first grade, I was designing and making my own pop-up cards. To this day, I love telling the story of the day the stars aligned and I won a Sega Genesis in an art competition—henceforth my indoctrination to the fascinating world of video games. You could call it fate, since my parents probably never would've bought me a game console!

Fast forward to 2007, I was a year into the game industry as an AI programmer and had picked up Blender in my spare time to communicate with artists at work better and to fuel my own creative outlet. It would take another 10 years, however, when I took on a project to port Blender to room-scale VR that I finally attempted to work with Blender Python—and boy, was that difficult! Even though I'd been using Blender for years and programming professionally, getting one thing to work in Blender Python would sometimes mean days of trial and error plus digging through documentation and poring over online posts. It may be cliché when authors say they wanted to write a book they wished had existed—in my case, I honestly did. This book is the accumulation of everything I learned working with Blender Python over the years, with solutions to problems that'd only come up when you try to write tools for a real production environment.

The book will show you how to write extension add-ons for Blender 4 from start to finish. Chapter 1 opens with a hands-on tour of the Scripting workspace with basics like loading and running scripts and turning hand modeling steps into Python by capturing them in the Info Editor. Chapter 2 explains the structure of operators and add-ons and shows how to use various input widgets to create user interfaces. You'll learn various

strategies for finding the Python equivalent of Blender menu options and hotkeys in a systematic way. In addition, you'll dissect add-ons shipped with Blender to see how they work and take advantage of built-in templates to quickly create new scripts.

With a firm grasp of scripting basics, in Chapters 3, 4, and 5, you'll find out how to create add-ons for editing and generating 3D models. In Chapter 3, you'll learn the basics of using modifiers and the `bmesh` module to edit meshes in Python and then write your own script to generate barrels from scratch with interesting variations. In Chapter 4, you'll add more advanced mesh editing functionalities to your add-ons, like extrude/bridge edge loops, loop cut-and-slides, plus all the essentials for manipulating vertices, edges, and faces like merge, rip, join, rotate, scale, bevel, and inset. You'll also write functions for cleaning up meshes and correcting normals. In addition, you'll learn how to handle imports (i.e., scripts that reference functions defined in one another) for both scripts meant to run in the Text Editor as well as packaged add-ons.

The second half of the book takes you through developing a series of practical and production-quality add-ons, inspired by various stages of a 3D modeling pipeline. In Chapter 5, you'll develop an add-on to procedurally generate stylized fire hydrant meshes with parametric controls. In Chapter 6, you'll create a suite of helper tools that collect user inputs from Grease Pencil strokes marked directly on sculpted meshes, for carving and in/outsetting selected regions as well as retopologizing them. In Chapter 7, you'll learn to write Python functions that automate key steps of the UV mapping process. Building on skills from Chapter 7, in Chapter 8, you'll create tools for projection painting textures by sampling from the same reference photos used to model the mesh. After mastering add-on development, in Chapter 9, you'll find out how to package, distribute, and market your extension add-ons through different channels. With the help of modal operators, you'll also create time-lapsed and interactive demos to showcase your procedural generation algorithm building a mesh gradually in the viewport with 3D text as captions.

Who This Book Is For

The intended audience of this book are 3D artists and programmers who want to create custom Blender add-ons to automate part of their workflow. Readers are assumed to have a high-level understanding of the 3D art pipeline, and either already use Blender or have experience with other CAD software such as 3Ds Max, Maya, or Rhino. Knowing basic Python is immensely helpful, although it is possible for motivated readers to learn it along the way by supplementing with resources outside the book.

Suggested Reading Road Map

The book is designed for a linear read through from Chapters 1 to 9. However, if you already have experience with Blender Python or are only interested in certain stages of the 3D modeling pipeline, it is possible to skip some chapters. In this section, I will suggest some alternative road maps through the book.

Chapters 1 and 2 provide a comprehensive overview of operators and add-ons, along with a plethora of strategies for systematically converting Blender edits by hand into Python. I suggest you read Chapters 1 and 2 regardless of any prior experience with Blender Python, as they'll serve as good refreshers and likely offer tips you've not seen elsewhere.

If you are primarily interested in mesh editing or procedural generation but not UV mapping or texturing, you can read Chapters 1–5 and safely skip Chapters 7 and 8. If the reverse is true, you can read Chapters 1, 2, 3, 7, and 8 and refer to Chapter 4 for an explanation on how to handle imports for both scripts run from the Text Editor and packaged add-ons. If you are not concerned with sculpting or retopology, you can safely skip Chapter 6; however, Chapter 6 does cover how to configure the Grease Pencil and process Grease Pencil input using Python.

If you are not looking to sell your add-ons, you can skip the parts of Chapter 9 that discuss marketing, promotion, and pricing strategies. Even if you are not concerned with making time-lapse demos like those mentioned in Chapter 9, they are built with modal operators that only run when certain type(s) of events are detected (like timer ticks or keystrokes), which may still be of interest and worth a read through.

CHAPTER 1

Getting Started on Blender Scripting

Just about any action you perform by hand in Blender can be automated with a script. In this chapter, I'll introduce you to Blender's built-in scripting interface, which includes the *Python Console* and the *Text Editor*. The Python Console is a convenient way to experiment with individual commands and see their effects in real time, whereas the Text Editor is great for editing and running scripts from files. We'll start by playing with API functions from the Python Console and observe immediate feedback happening in the viewport, followed by running one of Blender's built-in Template scripts in the Text Editor and observing its effects. Along the way, I will show you a variety of developer features that will help you discover which Python operator is behind a Blender menu item or hotkey and how to easily look up its implementation.

Introduction to Blender's Scripting Interface

Blender comes equipped with a *Scripting workspace*, which can be accessed by clicking the "Scripting" tab at the top of the screen as shown in Figure 1-1. The Scripting workspace contains a Text Editor for editing and running scripts from files, as well as an interactive Python Console that has been customized around the Blender Python API.

I. Lupiani, *Blender Scripting with Python*, https://doi.org/10.1007/979-8-8688-1127-2_1

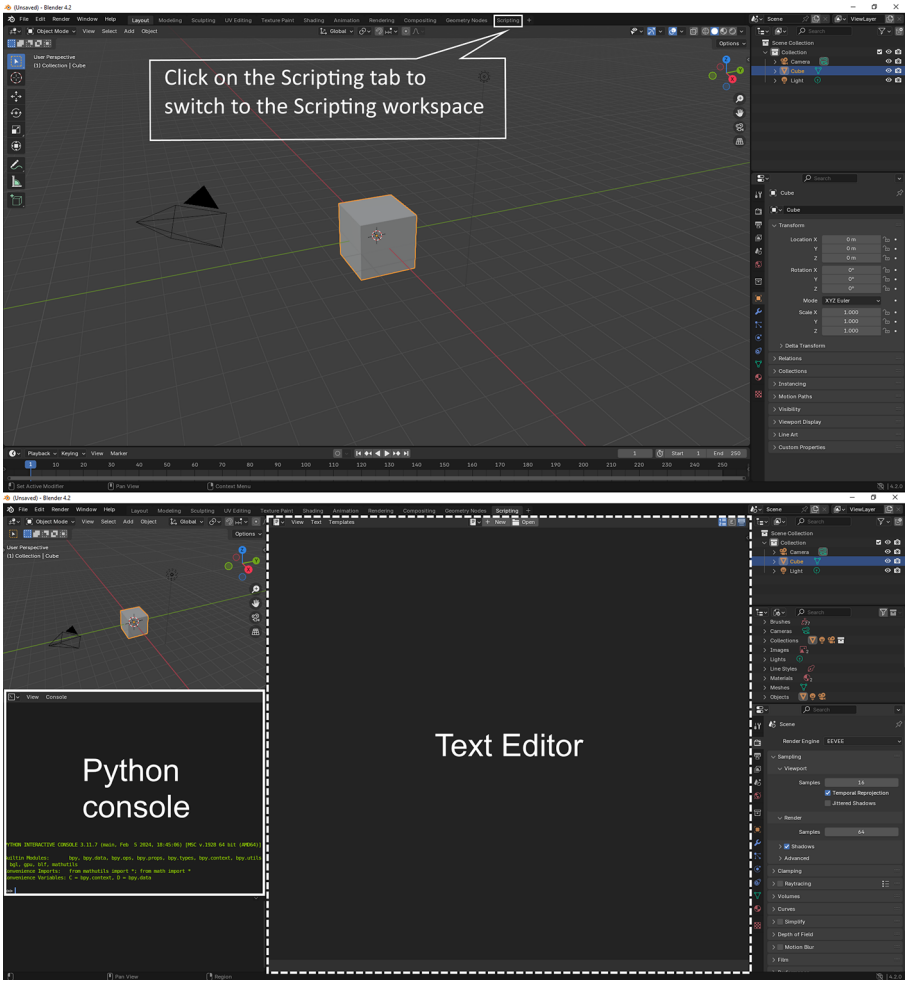

Figure 1-1. *Accessing the Scripting workspace (top). The Text Editor and Python Console within the Scripting workspace (bottom)*

Convenience Variables

Blender's built-in Python Console is customized specifically around Blender's scripting API and provides many features not found in a native Python installation. The first of these are *convenience variables*, which

are shorthand aliases for certain Blender modules. For example, the two convenience variables `C = bpy.context` and `D = bpy.data` are already defined for you inside the console.

A convenience variable is declared with the syntax `<new alias> = <name of variable>`. So, any time you find the need to type `bpy.context` at the prompt, you can enter `C` instead, which is a lot shorter and quicker to type.

Automatic Imports and Autocomplete

Another feature of the built-in console is that some of the frequently used Blender modules like `mathutils` as well as Python libraries like `math` are automatically imported for quicker access.

Yet another feature (which is also my favorite) is *Autocomplete*. If you're not sure what the name of a command is or are curious about what functions are under a module, you can enter a partial command at the prompt and use *Console* ➤ *Autocomplete* to search for a list of options for completing that command, as shown in Figure 1-2.

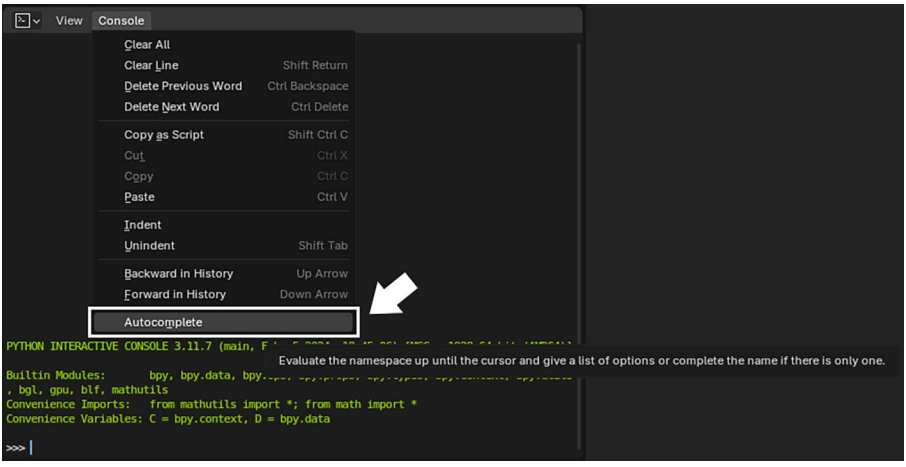

Figure 1-2. *The Autocomplete feature of the Python console*

Let's try this in an example. Inside the Python Console, type `import bmesh`, and hit the *enter* key to import the `bmesh` module (which is a built-in Blender Python module for mesh editing we'll go over in Chapters 3, 4, and 5). At the following prompt, type `bmesh.utils.`(be sure to include the dot at the end), then, without hitting *enter*, click *Console* ➤ *Autocomplete*. You should see the list of available functions under `bmesh.utils.` automatically brought up, as shown in Listing 1-1.

Listing 1-1. Retrieving the list of functions under `bmesh.utils` using Console ➤ Autocomplete

```
>>> import bmesh
>>> bmesh.utils.
                edge_rotate(
                edge_split(
                face_flip(
                face_join(
                face_split(
                face_split_edgenet(
                face_vert_separate(
                loop_separate(
                vert_collapse_edge(
                vert_collapse_faces(
                vert_dissolve(
                vert_separate(
                vert_splice(
>>> bmesh.utils.
```

Tip If the list of results returned by *Autocomplete* is too long to read without scrolling, you can use *View* ➤ *Area* ➤ *Toggle Maximize Area* (Ctrl-Spacebar) to maximize it, then either hit Ctrl-Spacebar a second time or click the "Back to Previous" button along the top menu bar to bring it back down to size when you're done. This functionality is available in many other screen areas such as the 3D Viewport, Text Editor, and so on.

If you have an idea of what a function or property's name might be but are unsure of the exact wording, you can type in a partial name as the search term and utilize *Autocomplete* as a search tool. For instance, if you remember that some of the debug settings are under the `bpy.app` module, you can find their precise names by typing `bpy.app.debug` at the prompt, and without pressing the enter key, click *Console* ➤ *Autocomplete*, as shown in Listing 1-2.

Listing 1-2. Searching for debug options under `bpy.app` using Console ➤ Autocomplete

```
>>> bpy.app.debug

                _depsgraph
                _depsgraph_build
                _depsgraph_eval
                _depsgraph_pretty
                _depsgraph_tag
                _depsgraph_time
                _events
                _ffmpeg
                _freestyle
                _handlers
```

```
                    _io
                    _python
                    _simdata
                    _value
                    _wm
>>> bpy.app.debug_events
False
```

The results from *Autocomplete* in this example suggest that `bpy.app.debug_events` is a valid variable in the Blender API (note that you can tell it's a variable as opposed to a function, since its name does not have a trailing pair of parentheses). To find out more about `bpy.app.debug_events`'s usage, you can type it at the next prompt in the console and hit enter. The console returns `False`, which indicates that `bpy.app.debug_events` is a boolean property with its current value set to `False`. If you had typed `bpy.app.debug_Events` instead, which is the same term but with the wrong capitalization (notice the E instead of e for `event`), *Autocomplete* will fail to return any results and display an AttributeError instead:

```
>>> bpy.app.debug_Events
Traceback (most recent call last):
  File "<blender_console>", line 1, in <module>
AttributeError: 'bpy.app' object has no attribute
'debug_Events'
```

If you spell out the full name of a function and hit *Autocomplete*, the console will bring up that function's entry in the documentation where available. This is helpful for learning the correct usage of a function, like the types and ordering of its input parameters and what values it returns.

If you need more information on how to use a method, you can always look it up in the Blender Python API online documentation at https://docs.blender.org/api/current/index.html. Notice, however, since

Blender is continuously in development, you might not see the same level of support for all parts of the API, particularly those that are newer or have recently undergone changes.

Caution *Autocomplete*'s search capabilities are rather limited since the partial term's capitalization and the ordering of the words both need to be correct.

Example: Move Mesh Vertices in the Viewport with the Python Console

Playing with commands at the built-in console with the help of *Autocomplete* is a great way to learn your way around the Blender API. Let's try this with an example. You'll run a series of commands at the console to move a corner vertex of the cube in the default startup blend file.

Switch to the *Scripting workspace*, and go over to the Python Console. At the prompt, enter the following command:

```
>>> cube = bpy.context.scene.objects["Cube"]
```

This retrieves a reference to the cube by its name (*"Cube"*) from the current list of scene objects and stores that reference in a variable called `cube`. Later on, you'll make edits to the cube with script commands through the `cube` variable.

We're going to use the built-in Blender module `bmesh` to manipulate `cube`'s mesh data. As shown in the previous example, since `bmesh` is not imported to the console by default, you'll need to explicitly import it by entering the command

```
>>> import bmesh
```

Next, switch cube to *Edit mode* in the *3D Viewport*, which will let you create a bmesh instance bm based on the cube's mesh data, cube.data, with the following command:

```
>>> bm = bmesh.from_edit_mesh(cube.data)
```

Through bm, you'll have access to the edges, faces, and vertices of cube's mesh data. For example, bm.verts is a sequence containing all of cube's vertices. Since a cube has eight corners, you can verify that the length of bm.verts is 8 with len(), which is a built-in Python function that tells you how many items are in a container, like this:

```
>>> len(bm.verts)
8
```

Next, you'll randomly pick one of cube's vertices and experiment by moving it through script commands. Enter the following command to select one of cube's vertices—let's say the vertex with index 3—and store a reference to it in the variable v. You can verify that v's index is 3 by printing it to the console with v.index.

```
>>> v = bm.verts[3]
>>> v.index
3
```

Note Sequences in Python (such as a list) use zero-based indexing, which means the first item of a sequence has index 0, the second item has index 1, and so on. The last item has an index that is one less than the total number of items in the sequence. For example, if a sequence has five items, the last item has index 4.

The numerical indices of vertices, edges, and faces in Blender reflect the order in which they are created (elements with smaller indices are created first). Note that unlike editing meshes in the 3D Viewport by hand, where you have to select a portion of the mesh with the mouse or keyboard first, many script operations allow you to directly manipulate part of a mesh without explicitly making a selection.

Next, we'll move `v` through commands at the console and verify its movement in the viewport. `v.co` is a trio of X, Y, and Z coordinates (a `Vector`) that represents `v`'s location in 3D. `v.co[0]` is its X coordinate, `v.co[1]` its Y coordinate, and `v.co[2]` its Z coordinate. Enter `v.co` at the prompt to display its current location:

```
>>> v.co
Vector((1.0, -1.0, -1.0))
```

To move `v`, you can edit one or more components of `v.co` or assign a new `Vector` to `v` altogether. Enter the following command to add 1 to `v.co[1]`, which moves `v` in the positive Y direction by one unit:

```
>>> v.co[1] += 1
>>> v.co
Vector((1.0, 0.0, -1.0))
```

In general, when you edit a mesh through a `bmesh` instance, the changes are queued up on the `bmesh` instance and not reflected on the mesh until you call the method `bmesh.update_edit_mesh(<name of mesh>)`. Ensure that the `cube` is in *Edit mode*, then, type the following line into the console to flush the change to `v.co` from `bm` to `cube`'s mesh data:

```
>>> bmesh.update_edit_mesh(cube.data)
```

You can verify `v`'s movement in the 3D Viewport, which should look like the right-hand side of Figure 1-3.

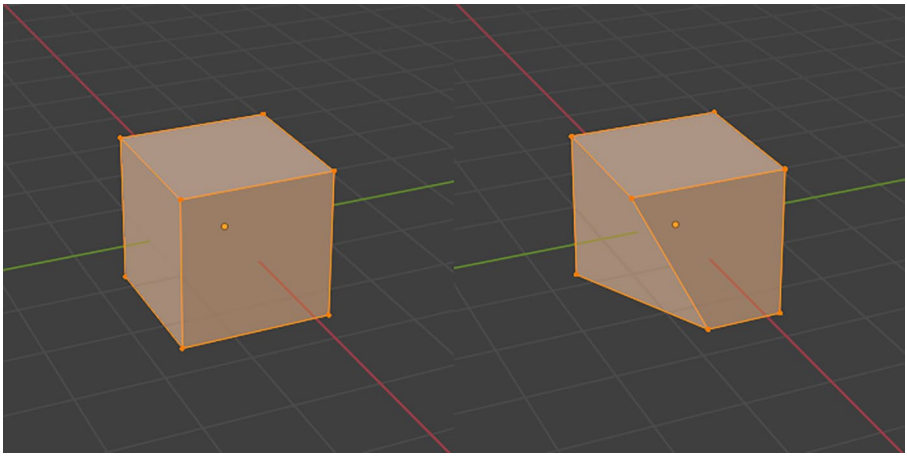

Figure 1-3. *Left: The cube in the viewport in its original state. Right: After its vertex at index 3 has been moved 1 unit in the positive Y direction*

Tip The Python Console in Blender acts very much like a typical command line interface. If you are already familiar with one, you'll pleasantly discover that many common shortcuts like the Up or Down Arrow keys for cycling through command history also work in the Python Console.

Transferring Console Contents into a Script

After experimenting with some commands in the console, you might find yourself arriving at a pretty solid basis for a script or add-on. To copy the console session in its entirety, with lines automatically formatted according to Python syntax, go to the top of the console window and select *Console* ➤ *Copy as Script* (***Shift-Ctrl-C***), then, paste (***Ctrl-V***) into any text editor. This will remove the >>> prompts and convert the console output lines to Python comments by prepending them with "#~".

Notice that within the Console menu, there are several additional options for editing commands at the prompt, such as *Indent/Unindent* for formatting a multiline function definition, *Clear Line* for erasing the current line, and *Clear All* for erasing the entire console history so far.

Editing and Running Script Files

You can quickly load, edit, and run existing scripts as well as create new ones in Blender's built-in *Text Editor*. It provides some basic Python programming support like line numbers and syntax highlighting. The main advantage of using the Text Editor is you can quickly run a script, observe its effects in the 3D Viewport or another area of Blender, make revisions to the script as necessary, and repeat, without having to switch back and forth between an external editor and Blender. Figure 1-4 shows the *Text Editor*'s user interface when it first starts up without a file loaded.

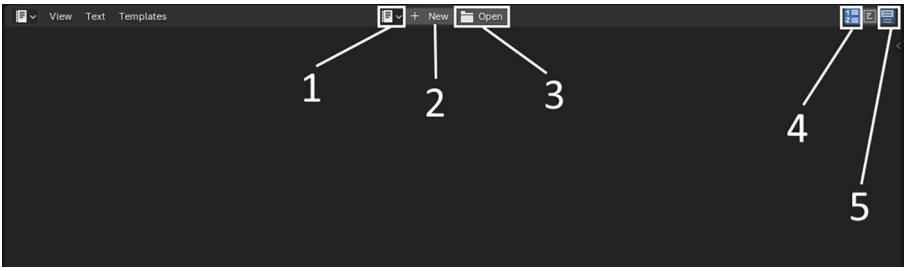

Figure 1-4. *Text Editor at startup without a file loaded. (1) Click the button to show the list of newly created files or files loaded from disk. (2) Create a new script file. (3) Open an existing script file on disk. (4) Toggle the display of line numbers in scripts. (5) Toggle syntax highlighting*

Let's open one of Blender's built-in template script files in the Text Editor. On the top menu bar, click *Templates* ➤ *Python* ➤ *Ui Panel Simple*. You'll see the contents of the file `ui_panel_simple.py` linked in as a new text data block and displayed inside the Text editor, as shown in Figure 1-5.