Das Blender-Buch

3D-Grafik und Animation mit Blender

Bearbeitet von
Carsten Wartmann

5., aktualisierte Auflage 2014. Buch. 426 S. Kartoniert
ISBN 978 3 86490 051 8
Format (B x L): 18,5 x 24,5 cm

10 Keine Angst vor Schlangen: Python

In diesem Kapitel beschreibe ich die Grundlagen und einige Anwendungsmöglichkeiten der Programmiersprache Python, die seit Version 1.68 in Blender eingebettet ist. Dieses Kapitel ist allerdings kein kompletter Python-Lehrgang, denn diese Sprache ist zwar leicht zu erlernen, aber auch sehr komplex.

Sprachkonzept

Python ist eine portable, interpretative, objektorientierte Skriptsprache. Die Sprache und ihre Standardbibliotheken sind auf allen Plattformen als ausführbare Programme und im Quelltext frei verfügbar. Entwickelt wurde Python von Guido van Rossum, der die Entwicklung heute immer noch maßgeblich steuert.

Seit Version 2.5x von Blender kommt Python eine noch viel bedeutendere Rolle als bisher in Blender zu. Mittlerweile wird Python 3.3 in Blender benutzt, und – noch viel wichtiger – die komplette grafische Benutzerschnittstelle von Blender wird nur noch durch Python-Skripte definiert und gezeichnet. Auch viele Erweiterungen und Im- oder Exportskripte sind jetzt Python-Skripte, ohne dass man es bei der Arbeit merken würde. Aufgrund der hohen Verzahnung von Python und Blender bringt Blender mittlerweile auch eine komplette Python-Installation mit, so dass es nicht mehr zwingend nötig ist, Python extra zu installieren, nur um in Blender damit zu arbeiten.

Für weitere Informationen zu Python ist die Homepage [PYTHON] die offizielle englischsprachige Anlaufstelle. Hier finden Sie viele Tutorials und die komplette Dokumentation zu Python.

Obwohl solch ein Kapitel über Programmierung sehr trocken sein kann, will ich versuchen, Ihnen die Beschäftigung mit Python schmerzfrei und kurzweilig nahezubringen – auch wenn Sie bisher keine Ambitionen hatten, eine Programmiersprache zu lernen. Wundern Sie sich also nicht, dass Wortwahl und Formulierungen in diesem Kapitel etwas locker und lässig daherkommen. Denn: Wichtiger als eine wissenschaftlich korrekte Ausfor-

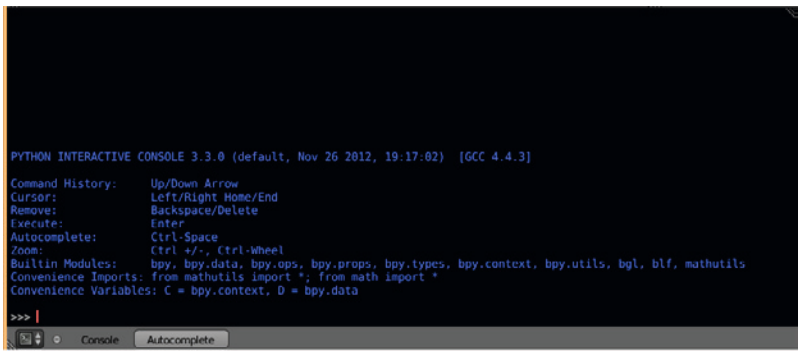
mulierung von Python ist mir hier die kreative Beschäftigung (Hacken im ursprünglichen Wortsinn) und der Spaß bei der Sache.

10.1 Erste Schritte mit Python

Python Console

Für den ersten Kontakt mit Python aus Blender heraus benutzt man am besten die interaktive Python Console. Die Python Console wird per ⇧-F4 oder über das Editor-Type-Menü aufgerufen. In der Standard-Szene von Blender existiert auch ein Screen »Scripting«, auf dem schon alles für die Programmierung in Python vorbereitet ist.

Abb. 10.1
Interaktive Python Console



Was bedeutet nun interaktive Konsole? Tippen Sie doch mal etwas ein, d. h., bewegen Sie die Maus über die Console und tippen Sie drauflos:

```
>>> hallo?
 File "<blender_console>", line 1
   hallo?
        ^
SyntaxError: invalid syntax
>>>
```

Blender versteht mich nicht!

Ich habe hier mal `hallo?` getippt und dann ↵ gedrückt. Die folgende Fehlermeldung weist mich darauf hin, dass dieses `hallo?` kein sinnvoller Python-Befehl war. Dies zeigt, dass Python nur darauf wartet, benutzt zu werden, und sofort auf unsere Befehle reagiert, also interaktiv ist.

Hallo Blender!

Also tippen wir einmal den für alle Programmiersprachen traditionell ersten Befehl ein:

```
>>> print("Hello Blender!")
Hello Blender!
>>>
```

Das sieht schon besser aus, denn `print()` ist ein eingebauter Befehl, der auf dem aktuellen Ausgabekanal etwas ausgibt. Das `"Hello Blender!"` ist auf Computernesisch ein sogenannter String-Literal, also ein Text, den Python an den Anführungszeichen erkennt. Python versteht übrigens auch einfache Anführungszeichen (`'...'`).

Python 2.x gegen 3.x

Blender verwendet Python 3.3

Wenn Sie schon Python 2.x kennen, werden Sie sich erst einmal daran gewöhnen müssen, dass print jetzt eine Funktion ist und nicht mehr als `print "Hallo"` geschrieben werden darf.

Diesen String nimmt also `print()` entgegen und gibt ihn aus. Dies ist natürlich nicht besonders aufregend, probieren Sie nun einmal `print(12*9)` oder Folgendes:

```
>>> for i in range(1,10):
...     print(i*2)
...
2
4
6
8
10
12
14
16
18
>>>
```

Anweisungsblöcke

Wichtig ist das ⇥ (Tabulator) in der zweiten Zeile: Python verwendet Einrückungen der Programmzeilen, um Anweisungsblöcke zusammenzufassen. Also wird alles, was hinter der Zeile mit dem Doppelpunkt steht und um ein ⇥ eingerückt ist, neun Mal ausgeführt. Die `for`-Schleife zählt demnach im Bereich (`range()`) 1 bis 10 hoch und schreibt die jeweilige Zahl in die Variable `i`.

Also versuchen wir doch mal, mit Blender zu »reden«. Die Console hat uns in der Startmeldung schon ein paar Hinweise gegeben, denn es ist dort von »Builtin Modules«, also eingebauten Modulen, die Rede. Module in Blender sind Funktionssammlungen, über die Python sich mit neuen Fähigkeiten erweitern lässt. Um nun die Funktionen der Module kennen zu lernen, kann man Bücher lesen, Online-Dokumentationen benutzen oder aber Python selbst befragen.

Dokumentation eingebaut

Python ist eine selbstdokumentierende Sprache. Ein `dir()`, angewendet auf einen Modulnamen, verrät uns z. B. einiges:

```
>>> print(dir(bpy))
['__all__', '__builtins__', '__cached__', '__doc__', '__
file__', '__initializing__', '__loader__', '__name__', '__
package__',
'__path__', 'app', 'context', 'data', 'ops', 'path', 'props',
'types', 'utils']
```

Das Modul »bpy« (was für Blender Python steht) hat also diese oben genannten Informationen zu bieten. Interessant scheint doch `data`, also weiter bohren: Python-typisch wird mit einem Punkt (».«) die Referenz zwischen den Modulen hergestellt:

```
>>> print(dir(bpy.data))
['__doc__', '__module__', '__slots__', 'actions', 'armatures',
'bl_rna', 'brushes', 'cameras', 'curves', 'filepath', 'fonts',
'grease_pencil', 'groups', 'images', 'is_dirty', 'is_saved',
'lamps', 'lattices', 'libraries', 'masks', 'materials',
'meshes', 'metaballs', 'movieclips', 'node_groups', 'objects',
'particles', 'rna_type', 'scenes', 'screens', 'scripts',
'shape_keys', 'sounds', 'speakers', 'texts', 'textures',
'window_managers', 'worlds']
>>>
```

Szenendaten

Aha, das sieht aus wie das »Who is Who« von Blender-Szenen. »Objects« klingt spannend – das schauen wir uns näher an:

```
>>> print(bpy.data.objects)
<bpy_collection[3], BlendDataObjects>
>>>
```

Die Ausgabe beschreibt uns eine Sammlung (Collection) von drei BlendDataObjects. Wenn Sie schon eine Weile mit Blender arbeiten, ahnen Sie bereits, welche das sein könnten:

```
>>> for obj in bpy.data.objects:
...     print(obj)
...
<bpy_struct, Object("Camera")>
<bpy_struct, Object("Cube")>
<bpy_struct, Object("Lamp")>
>>>
```

Auf diese Weise kann man sich nun zu allen Informationen und Daten in Blender durchhangeln, egal ob es Objekte, Materialien, F-Curves, Bilder, Welten oder Metaballs sind. Aber schauen wir uns noch etwas bei den Objekten um:

```
>>> print(dir(bpy.data.objects["Cube"]))
['MhAlpha8', 'MhxMesh', 'MhxRig', 'MhxRigify',
'MhxShapekeyDrivers', 'MhxSnapExact', 'MhxStrength', '__
doc__', '__module__', '__qualname__', '__slots__', 'active_
material', 'active_material_index', 'active_shape_key',
'active_shape_key_index', 'animation_data', 'animation_data_
clear', 'animation_data_create', 'animation_visualization',
'bl_rna', 'bound_box', 'children', 'closest_point_on_mesh',
'collision', 'color', 'constraints', 'convert_space', 'copy',
'cycles_visibility', 'data', 'delta_location', 'delta_
rotation_euler', 'delta_rotation_quaternion', 'delta_scale',
'dimensions', 'draw_bounds_type', 'draw_type', 'dupli_faces_
scale', 'dupli_frames_end', 'dupli_frames_off', 'dupli_frames_
on', 'dupli_frames_start', 'dupli_group', 'dupli_list',
'dupli_list_clear', 'dupli_list_create', 'dupli_type', 'empty_
draw_size', 'empty_draw_type', 'empty_image_offset', 'extra_
recalc_data', 'extra_recalc_object', 'field', 'find_armature',
'game', 'grease_pencil', 'hide', 'hide_render', 'hide_select',
'is_deform_modified', 'is_duplicator', 'is_library_indirect',
'is_modified', 'is_updated', 'is_updated_data', 'is_visible',
'layers', 'layers_local_view', 'library', 'location', 'lock_
location', 'lock_rotation', 'lock_rotation_w', 'lock_
rotations_4d', 'lock_scale', 'material_slots', 'matrix_basis',
'matrix_local', 'matrix_parent_inverse', 'matrix_world',
'mode', 'modifiers', 'motion_path', 'name', 'parent', 'parent_
bone', 'parent_type', 'parent_vertices', 'particle_systems',
'pass_index', 'pose', 'pose_library', 'proxy', 'proxy_group',
'ray_cast', 'rigid_body', 'rigid_body_constraint', 'rna_type',
'rotation_axis_angle', 'rotation_euler', 'rotation_mode',
'rotation_quaternion', 'scale', 'select', 'shape_key_add',
'show_all_edges', 'show_axis', 'show_bounds', 'show_name',
'show_only_shape_key', 'show_texture_space', 'show_
transparent', 'show_wire', 'show_x_ray', 'slow_parent_offset',
'soft_body', 'tag', 'to_mesh', 'track_axis', 'type', 'up_
axis', 'update_from_editmode', 'update_tag', 'use_dupli_faces_
scale', 'use_dupli_frames_speed', 'use_dupli_vertices_
rotation', 'use_dynamic_topology_sculpting', 'use_fake_user',
'use_shape_key_edit_mode', 'use_slow_parent', 'user_clear',
'users', 'users_group', 'users_scene', 'vertex_groups']
>>>
```

Das bringt uns zu folgenden Erkenntnissen:

- Wir können Objekte nicht nur mit ihrer Nummer ansprechen, sondern auch mit ihrem Namen (`objects["Cube"]`).
- 3D-Objekte in Blender-Szenen haben Massen von Informationen und Eigenschaften, mit denen sie sich manipulieren lassen.
- Ohne eine gute Dokumentation ist man verloren.

API-Referenz

Den dritten Punkt können wir durch das Help-Menü und den Menüpunkt Python API Reference schon mal abhaken, denn hier gibt es immer die aktuelle und gut durchsuchbare Dokumentation zur API (API steht für Application Programming Interface und bezeichnet in unserem Fall die Programmierschnittstelle von Python und Blender).

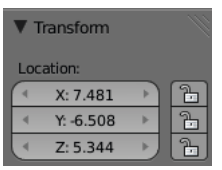
Der zweite Punkt der Liste zeigt nur, wie flexibel und komplett die Python-Integration in Blender ist. In den nächsten Jahren werden sicherlich noch einige sehr hilfreiche Skripte und Add-ons erscheinen, denn letztlich ist es doch einfacher, etwas in Python zu entwickeln, als direkt in den C- oder C++-Quellcode von Blender einzusteigen.

Mit der Erkenntnis aus dem ersten Punkt der Liste (Objekte mit dem Namen ansprechen) machen wir gleich weiter:

```
>>> print(bpy.data.objects["Cube"].name)
Cube
>>>
```

Ach nee … echt? Na, wenigstens ist man sich in Blender-Internalien einig über die Namen seiner Einwohner. Aber geht es nicht doch etwas sinnvoller? Sicherlich:

```
>>> print(bpy.data.objects["Cube"].location)
Vector((0.0, 0.0, 0.0))
>>> print(bpy.data.objects["Camera"].location)
<Vector (7.4811, -6.5076, 5.3437)>
>>>
```



Wenn wir uns die Szene einmal anschauen, dann scheint das zu stimmen: Der Würfel ist eindeutig im Weltmittelpunkt und das Transform-Panel für die Camera Location: gibt uns auch die passenden Werte zurück. Wenn wir die Position der Objekte auslesen können, dann müssten wir doch auch …

```
>>> bpy.data.objects["Cube"].location = (0,0,4)
>>>
```

Jawohl, wir können auch seine Position ändern: Der Würfel rückt im 3D View um vier Blender-Einheiten hoch – q.e.d.

Context

Nun ist es natürlich schwierig, sich immer alle Objektnamen zu merken, um dann eine Operation damit auszuführen – vor allem wenn man geschlampt und seine Objekte nicht konsequent benannt hat. Wir erinnern uns an die interaktive Arbeit im 3D View: Die Namen haben wir kaum verwendet und trotzdem wusste Blender immer, welches Objekt wir meinen. Klar, denn wir haben es ja auch vorher selektiert. Diese Information bekommen wir natürlich auch per Python, und zwar durch den sogenannten Context, also Zusammenhang:

```
>>> print(bpy.context.object.name)
Cube
>>>
```

Der schon wieder. Selektieren Sie doch mal den Würfel mit 🖱 und dann die Kamera dazu mit ⇧-🖱. Anschließend führen Sie den Befehl nochmals aus (↑,↵)

```
>>> print(bpy.context.object.name)
Camera
>>>
```

Aktives Objekt und selektierte Objekte

Hmm. Aber wir haben doch zwei Objekte selektiert? `bpy.context.object` gibt uns immer das aktive Objekt aus. Das ergibt ja auch Sinn nach der Blender-Philosophie und so kennen wir das ja schon. Alle selektierten Objekte bekommen wir wie folgt:

```
>>> print(bpy.context.selected_objects)
[bpy.data.objects["Cube"], bpy.data.objects["Camera"]]
>>>
```

Machen Sie dann folgendermaßen weiter:

```
>>> for obj in bpy.context.selected_objects:
...     obj.location[2]=0
...
>>>
```

Und alle selektierten Objekte klatschen auf den Z = 0-Boden. Im Prinzip haben wir jetzt alles zusammen, um eigene Befehle in Blender programmieren zu können, denn natürlich gibt es neben dem Objektzugriff noch Schnittstellen zu den Meshes, Materialien und so weiter. Aber es gibt in Blender noch eine Ebene über diesem »Low Level«-Zugriff, die Operatoren.