3.7.2 - Editors - Other - Python Console

Python Console

The Python console is a quick way to execute commands, with access to the entire Python API, command history and auto-complete.

Its a good way to explore possibilities, which can then be pasted into larger scripts.

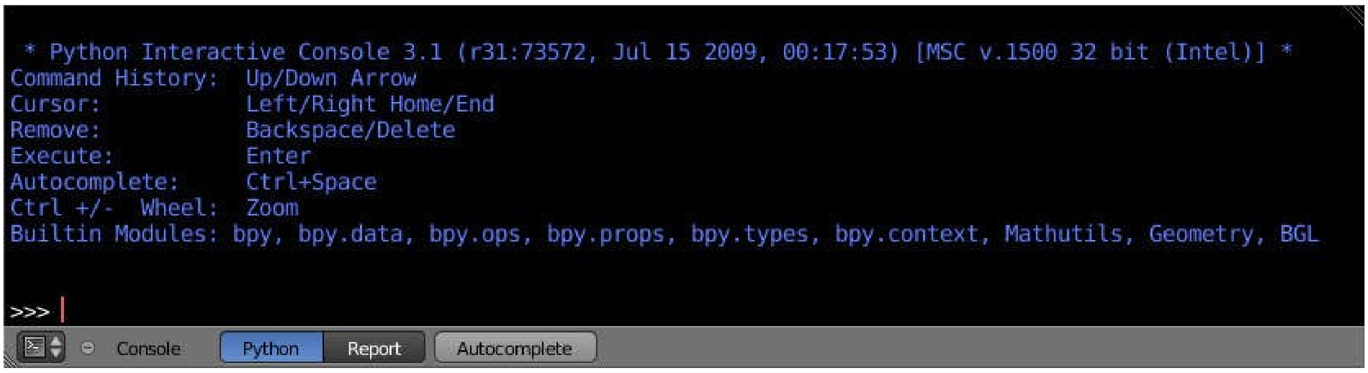
Usage

Accessing Built-in Python Console

Launching the Console using mouse.

https://www.youtube.com/watch?v=Ge2Kwy5EGE0

By pressing `Shift-F4` in any Blender Editor Type (3D View, Timeline etc.,) you can change it to a Console Editor.



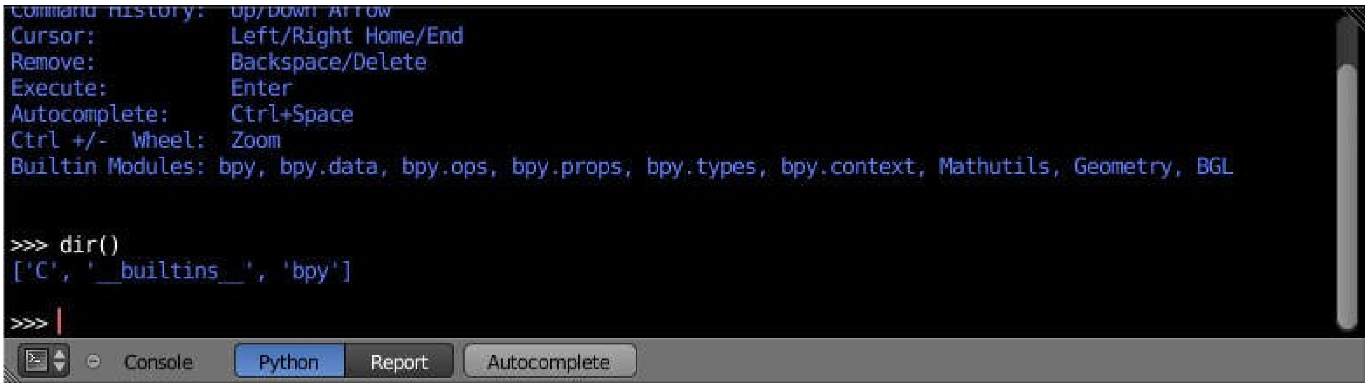
```
 * Python Interactive Console 3.1 (r31:73572, Jul 15 2009, 00:17:53) [MSC v.1500 32 bit (Intel)] *
Command History:  Up/Down Arrow
Cursor:           Left/Right Home/End
Remove:           Backspace/Delete
Execute:          Enter
Autocomplete:     Ctrl+Space
Ctrl +/-  Wheel:  Zoom
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, Mathutils, Geometry, BGL


>>>
```

From the screen shot above, you will notice that apart from the usual hot keys that are used to navigate, by pressing `Ctrl-Spacebar` you can enable Auto-complete feature.

The command prompt is typical for Python 3.x, the interpreter is loaded and is ready to accept commands at the prompt >>>

First look at the Console Environment

To check what is loaded into the interpreter environment, type dir() at the prompt and execute it.



```
Command History:  Up/Down Arrow
Cursor:           Left/Right Home/End
Remove:           Backspace/Delete
Execute:          Enter
Autocomplete:     Ctrl+Space
Ctrl +/-  Wheel:  Zoom
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, Mathutils, Geometry, BGL


>>> dir()
['C', '__builtins__', 'bpy']

>>> |
```

Following is a quick overview of the output

C

Quick access to `bpy.context`

D

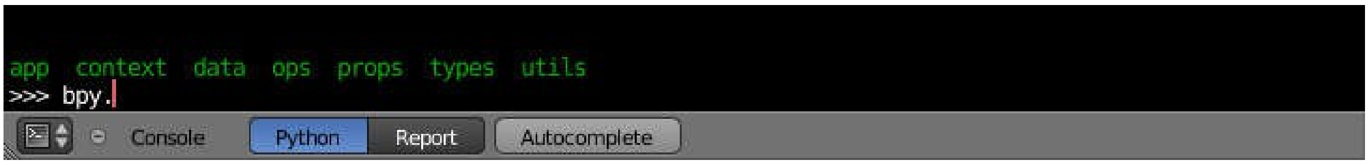
Quick access to `bpy.data`

bpy

Top level Blender Python API module.

Auto Completion at work

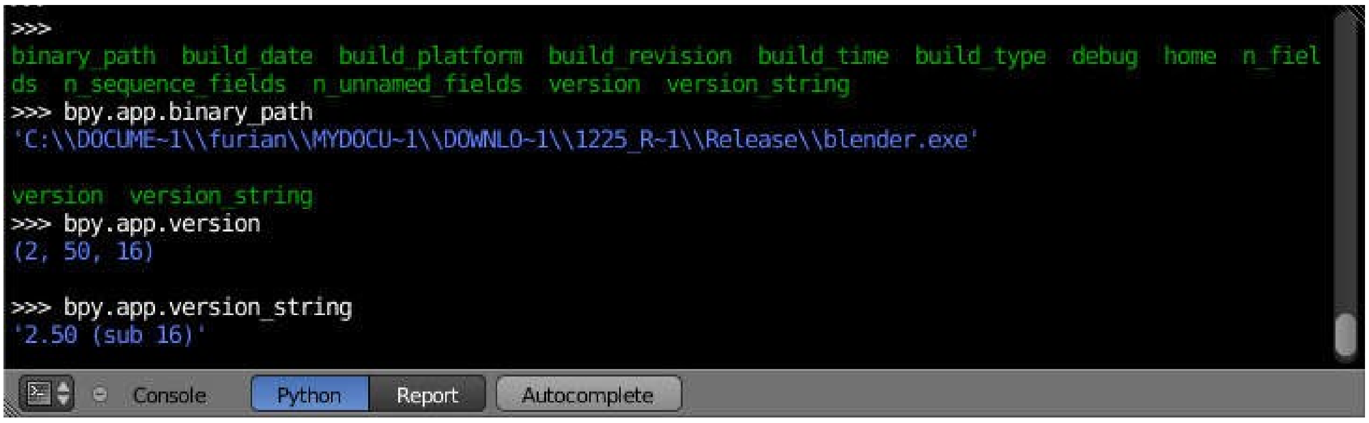
Now, type bpy. and then press `Ctrl-Spacebar` and you will see the Console auto-complete feature in action.



```
app  context  data  ops  props  types  utils
>>> bpy.|
```

You will notice that a list of sub-modules inside of bpy appear. These modules encapsulate all that we can do with Blender Python API and are very powerful tools.

Lets list all the contents of bpy.app module.



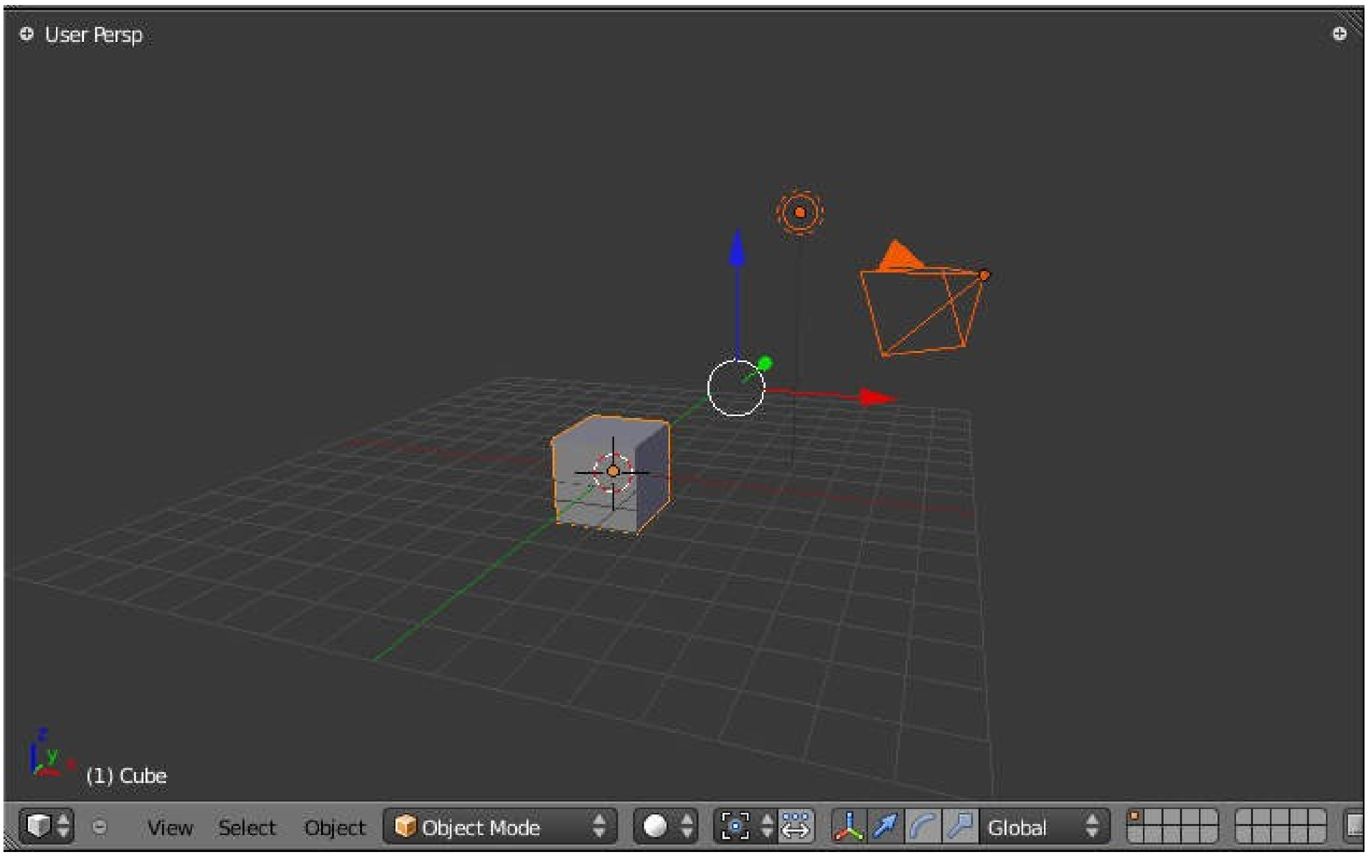
```
>>>
binary_path  build_date  build_platform  build_revision  build_time  build_type  debug  home  n_fiel
ds  n_sequence_fields  n_unnamed_fields  version  version_string
>>> bpy.app.binary_path
'C:\\DOCUME~1\\furian\\MYDOCU~1\\DOWNLO~1\\1225_R~1\\Release\\blender.exe'

version  version_string
>>> bpy.app.version
(2, 50, 16)

>>> bpy.app.version_string
'2.50 (sub 16)'
```

Notice the green output above the prompt where you enabled auto-completion. What you see is the result of auto completion listing. In the above listing all are module attribute names, but if you see any name end with '(', then that is a function.

We will make use of this a lot to help our learning the API faster. Now that you got a hang of this, lets proceed to investigate some of modules in bpy.

Before tinkering with the modules..

If you look at the 3D Viewport in the default Blender scene, you will notice 3 objects: Cube, Lamp and Camera.



- All objects exist in a context and there can be various modes under which they are operated upon.
- At any instance, only one object is active and there can be more than one selected objects.
- All objects are data in the Blender file.
- There are operators/functions that create and modify these objects.

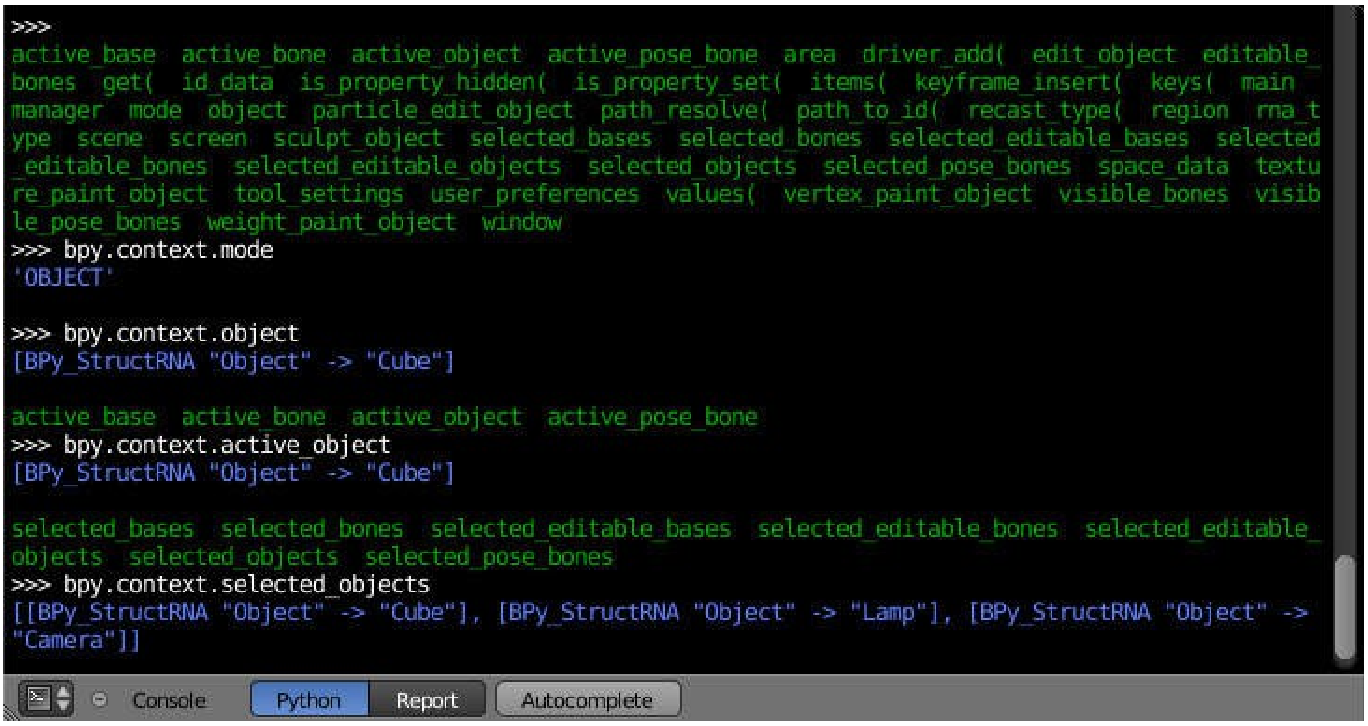
For all the scenarios listed above (not all were listed, mind you..) the bpy module provides functionality to access and modify data.

Examples

bpy.context

Note

For the commands below to show the proper output, make sure you have selected object(s) in the 3D view.



```
>>>
active_base  active_bone  active_object  active_pose_bone  area  driver_add(  edit_object  editable_
bones  get(  id_data  is_property_hidden(  is_property_set(  items(  keyframe_insert(  keys(  main
manager  mode  object  particle_edit_object  path_resolve(  path_to_id(  recast_type(  region  rna_t
ype  scene  screen  sculpt_object  selected_bases  selected_bones  selected_editable_bases  selected
_editable_bones  selected_editable_objects  selected_objects  selected_pose_bones  space_data  textu
re_paint_object  tool_settings  user_preferences  values(  vertex_paint_object  visible_bones  visib
le_pose_bones  weight_paint_object  window
>>> bpy.context.mode
'OBJECT'

>>> bpy.context.object
[BPy_StructRNA "Object" -> "Cube"]

active_base  active_bone  active_object  active_pose_bone
>>> bpy.context.active_object
[BPy_StructRNA "Object" -> "Cube"]

selected_bases  selected_bones  selected_editable_bases  selected_editable_bones  selected_editable_
objects  selected_objects  selected_pose_bones
>>> bpy.context.selected_objects
[[BPy_StructRNA "Object" -> "Cube"], [BPy_StructRNA "Object" -> "Lamp"], [BPy_StructRNA "Object" ->
"Camera"]]
```

Try it out!

bpy.context.mode

Will print the current 3D View mode (Object, Edit, Sculpt etc.,)

bpy.context.object or bpy.context.active_object

Will give access to the active object in the 3D View

```
>>> bpy.context.object.location.x = 1
```

Change x location to a value of 1

```
>>> bpy.context.object.location.x += 0.5
```

Move object from previous x location by 0.5 unit

```
>>> bpy.context.object.location = (1, 2, 3)
```

Changes x, y, z location

```
>>> bpy.context.object.location.xyz = (1, 2, 3)
```

Same as above

```
>>> type(bpy.context.object.location)
```

Data type of objects location

```
>>> dir(bpy.context.object.location)
```

Now that is a lot of data that you have access to

bpy.context.selected_objects

Will give access to a list of all selected objects.

```
>>> bpy.context.selected_objects
```

... then press `Ctrl-Spacebar`

```
>>> bpy.context.selected_objects[0]
```

Prints out name of first object in the list

```
>>> [object for object in bpy.context.selected_objects if object != bpy.context.object]
```

Complex one... But this prints a list of objects not including the active object

bpy.data

`bpy.data` has functions and attributes that give you access to all the data in the Blender file.

You can access following data in the current Blender file: objects, meshes, materials, textures, scenes, screens, sounds, scripts, ... etc.

That's a lot of data.

Try it out!

Exercise

```
>>> for object in bpy.data.scenes['Scene'].objects: print(object.name)
```

`Return` twice Prints the names of all objects belonging to the Blender scene with name "Scene"

```
>>> bpy.data.scenes['Scene'].objects.unlink(bpy.context.active_object)
```

Unlink the active object from the Blender scene named 'Scene'

```
>>> bpy.data.materials['Material'].shadows
```

```
>>> bpy.data.materials['Material'].shadows = False
```

bpy.ops

The tool/action system in Blender 2.5 is built around the concept of operators. These operators can be called directly from console or can be executed by click of a button or packaged in a python script. Very powerful they are..

For a list of various operator categories, click here

Lets create a set of five Cubes in the 3D Viewport. First, delete the existing Cube object by selecting it and pressing `X`

Try it out!

The following commands are used to specify that the objects are created in layer 1. So first we define an array variable for later reference:

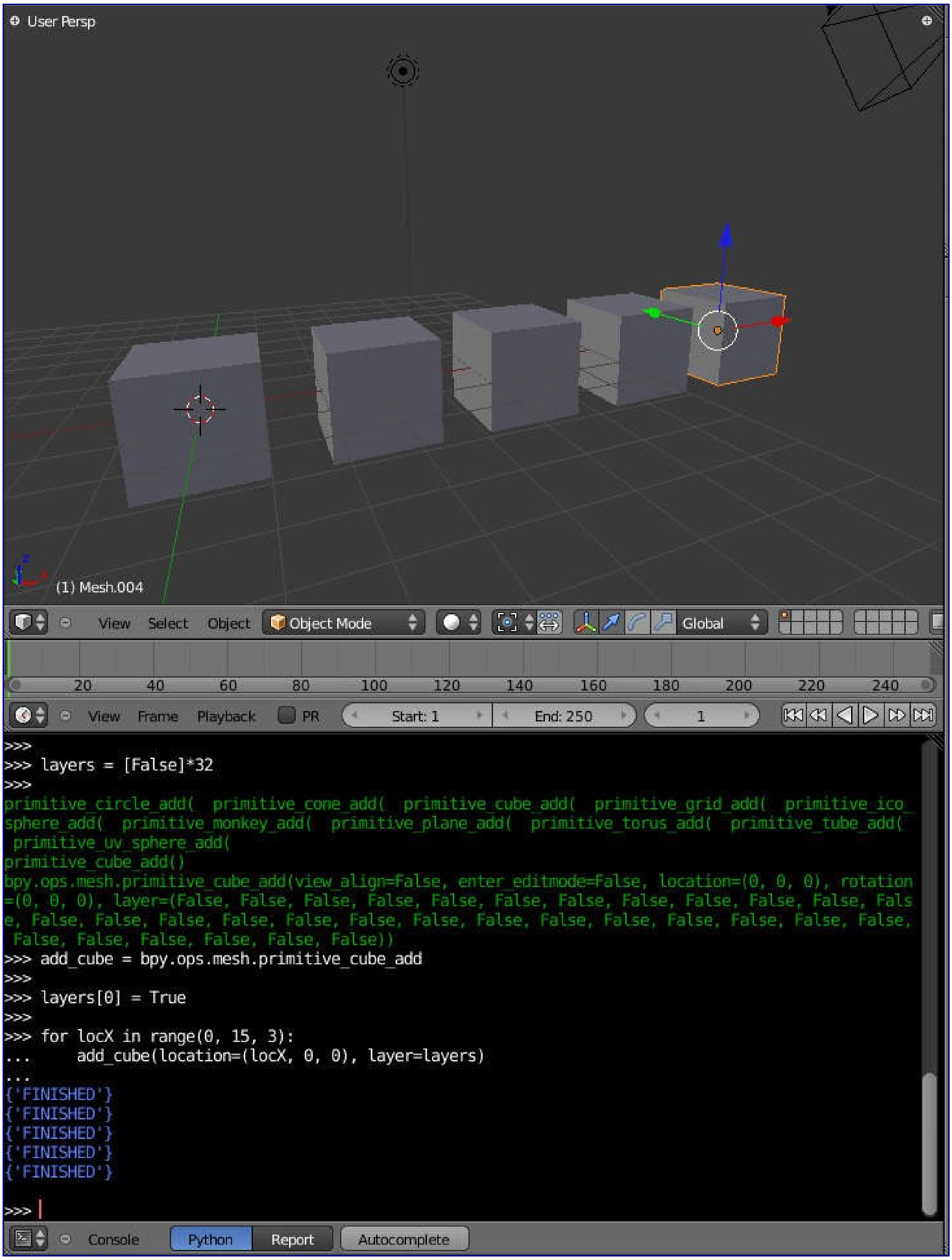
```
>>> mylayers = [False] * 20
>>> mylayers[0] = True
```

We create a reference to the operator that is used for creating a cube mesh primitive

```
>>> add_cube = bpy.ops.mesh.primitive_cube_add
```

Now in a for loop, we create the five objects like this (In the screenshot above, I used another method) Press ENTER-KEY twice after entering the command at the shell prompt.

```
>>> for index in range(0, 5):
...     add_cube(location=(index * 3, 0, 0), layers=mylayers)
```



```
>>>
>>> layers = [False]*32
>>>
primitive_circle_add(  primitive_cone_add(  primitive_cube_add(  primitive_grid_add(  primitive_ico_
sphere_add(  primitive_monkey_add(  primitive_plane_add(  primitive_torus_add(  primitive_tube_add(
 primitive_uv_sphere_add(
primitive_cube_add()
bpy.ops.mesh.primitive_cube_add(view_align=False, enter_editmode=False, location=(0, 0, 0), rotation
=(0, 0, 0), layer=(False, False, False, False, False, False, False, False, False, False, False, Fals
e, False, False, False, False, False, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False))
>>> add_cube = bpy.ops.mesh.primitive_cube_add
>>>
>>> layers[0] = True
>>>
>>> for locX in range(0, 15, 3):
...     add_cube(location=(locX, 0, 0), layer=layers)
...
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}

>>>
```