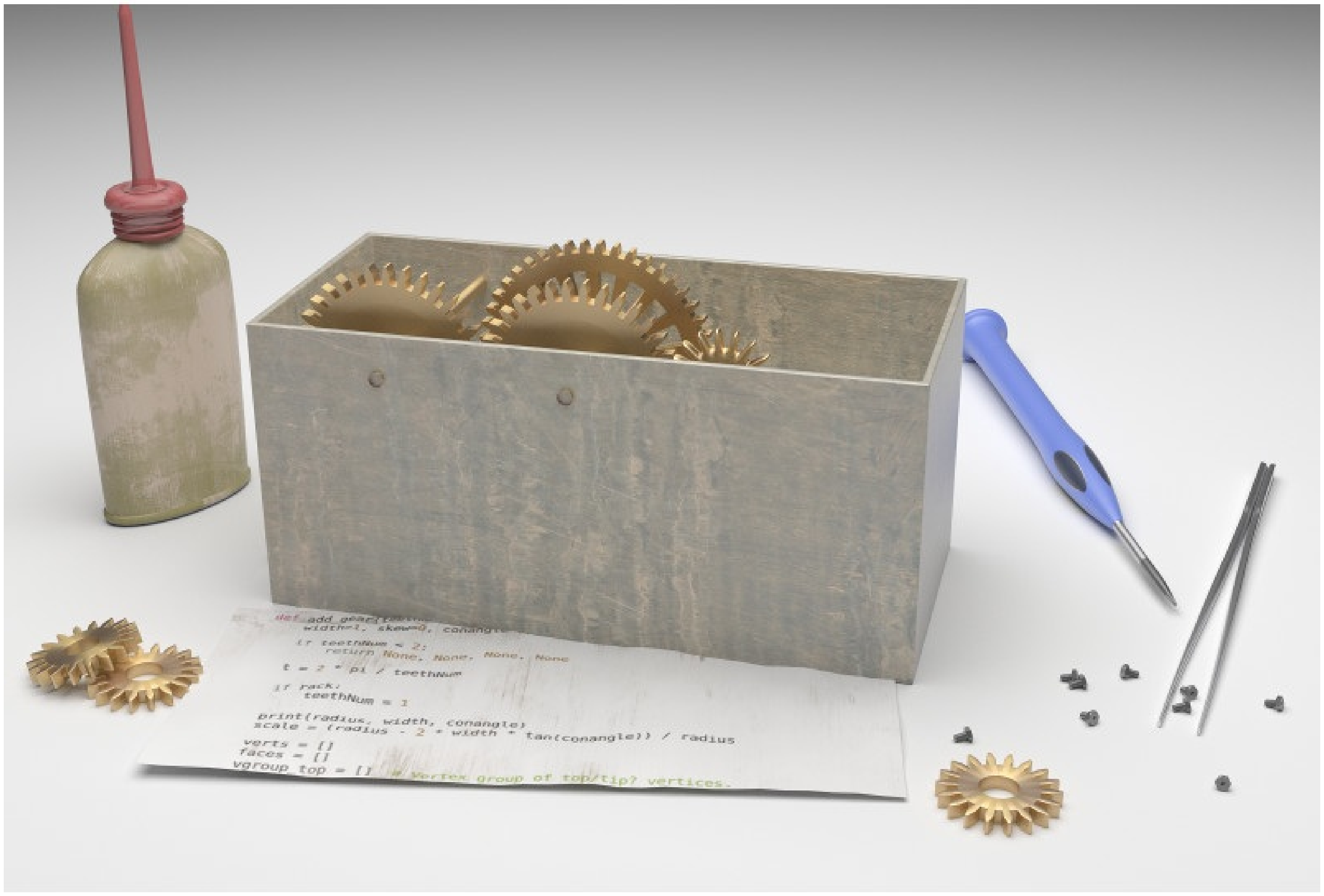
Michel Anders



```
def add_gear(teethNu
    width=1, skew=0, conangle
  if teethNum < 2:
      return None, None, None, None
  t = 2 * pi / teethNum
  if rack:
     teethNum = 1
  print(radius, width, conangle)
  scale = (radius - 2 * width * tan(conangle)) / radius
  verts = []
  faces = []
  vgroup_top = []  # Vertex group of top/tip? vertices.
```

Creating add-ons for Blender A practical primer

Creating add-ons for Blender

By Michel Anders

Smashwords Edition

Table of Contents

Introduction

This primer on writing Blender add-ons aims to be short an sweet and pretty fast paced. This means that we cover quite a lot of concepts in a short time but I have provided links to additional information where relevant.

You also need to have some experience in writing Python programs but other than that you will find that writing Blender add-ons is surprisingly simple. Almost anything is possible due to the large amount of built-in functionality available from within Python and this huge collection of classes and functions might be a little bewildering to navigate but the basics of a Blender add-on are easy to grasp in just a few hours.

In this primer you will start with an almost trivial add-on and work your way to a full fledged add-on that manipulates meshes and is fully integrated with Blender own graphical user interface, complete with user configurable properties and custom icons.

Overview

In chapter 1 we will introduce the basic concepts of an Operator and have a look at Blenders data model. Here we will learn that almost anything in a scene is accessible to an add-on and how to add information to a Python file so that your add-on can be distributed and installed by other people. Even though this first add-on is trivial it will be fully integrated in Blenders menus.

Chapter two expands on the basic concepts by ensuring that our add-on can only be used in meaningful context. It also shows how to add custom icons to a menu and explains how you can create and distribute add-ons that consist of more than one file.

Most add-ons offer the end user options that can be changed to change the behavior of the add-on. These options, or Properties as they are called in Blender are introduced in chapter 3 where we will also see how the values of these options can be saved in so called presets.

In chapter 4 we have a look at mesh manipulation. We discover how to select vertices or add vertex color.

Chapter 5 focuses on the creation of mesh objects from scratch. It introduces Blenders powerful mesh manipulation library BMesh and shows how to add data layers to your meshes like vertex colors and uv-maps.

In the final chapter we stay with meshes but show how we can create parametric objects, objects that store the values that where used to create them along with all the other data. This makes it possible to recreate or tweak complex objects even after they were stored or copied and creates the opportunity to animate these values.

Notes on code

All the example code is available for download and each chapter contains a link to the relevant code.

Formatting code is sometimes difficult, especially in Python where indentation is relevant. I did my best not to mangle the code in the examples listed in the book but when in doubt make sure you check the code that is available for download first.

A special note for Python purists: I use the terms function, member function and method interchangeably. Even though some people might argue that there are differences in their exact meaning I feel this would detract from the subject, which is creating add-ons for Blender. The same goes for pep-8 compliance: due to formatting requirements in books, it is not feasible to adhere to pep-8 all the time.

Have fun!

A Blender add-on: the basics

Almost anything is possible when writing add-ons for Blender but to get a good understanding we start simple. Nevertheless the first add-on we will create can be installed and removed just like any other add-on, provides some extra functionality in the form of an operator and creates a menu entry so the user can easily access this new functionality.

Topics covered

- Providing information about the add-on
- Defining an operator
- Registering an operator
- Creating a menu entry

Example: move an object

The new functionality we will provide with our first add-on is minimal: the add-on will create a menu entry in the Object menu of the 3D view that will move the active object one unit along the x-axis. This will not make this add-on the next killer app but will illustrate nicely how easy it is to provide extra functionality that is fully integrated in Blender's existing user interface.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub. It is called* ***`move_01.py`***

The anatomy of an add-on

In its simplest form an add-on is a single Python file that

- provides some general information about the add-on, like its name and version,
- defines some code to perform an action, often in the form of an operator, and
- makes sure this operator is registered so that it can be used.

We'll have a look at each of these components in turn.

bl_info

The general information about the add-on is defined in a dictionary with the name **`bl_info`** which is normally located at the beginning of the file.

```
bl_info = {
        "name": "MoveObject",
        "author": "Michel Anders (varkenvarken)",
        "version": (0, 0, 20160104121212),
        "blender": (2, 76, 0),
        "location": "View3D > Object > Move",
        "description": "Moves and object",
        "warning": "",
        "wiki_url": "",
        "tracker_url": "",
        "category": "Object"}
```

Each key provides Blender with specific information about our add-on although not all are equally important. Most of the information is used in the user preferences dialog and helps the user to find and select an installed add-on.

name
: A short and memorable name

author
: Always nice if people can credit you for your work

version
: The version of your add-on. You can use any numbering scheme you like, as long as it is a tuple of three integers. I simply use a time stamp in the last position but you might choose for a more structured scheme.

blender
: The minimal Blender version needed by this add-on. Again a tuple of three integers. Even if you expect your add-on to work with older versions it might be a good idea to list the earliest version that you actually tested your add-on with!

category
: The category in the user preferences your add-on is grouped under. Our add-on will operate on an object so it makes sense to add it to the Object category.

location
: Where to find the add-on once it is enabled. This might a reference to a specific panel or in out case, a description of its location in a menu.

description
: A concise description of what the add-on does.

warning

If this is not an empty string, the add-on will show up with a warning sign in the user preferences. You might use this to mark an add-on as experimental for example.

wiki_url

If you provide on-line documentation, you can provide a url here. It will be a click-able item in the user preferences.

tracker_url

If your add-on ends up as a bundled part of Blender it will have its own bug tracker entry associated with it and this key will provide a pointer to it.

An Operator class

Most add-ons define new *operators*, classes that implement specific functionality. Our MoveObject add-on is no exception and will implement a single operator to do the actual moving.

The actual definition of the operator takes the form a class that is derived from **`bpy.types.Operator`**

```
import bpy

class MoveObject(bpy.types.Operator):
    """Moves an object"""
    bl_idname = "object.move_object"
    bl_label = "Move an object"
    bl_options = {'REGISTER', 'UNDO'}
```

The docstring at the start of the class definition will be used as a tooltip anywhere this operator will be available, for example in a menu, while the **`bl_label`** defines the actual label used in the menu entry itself. Here we kept both the same. Operators will be part of Blender's data, and operators are stored in the module **`bpy.ops`**. This **`bl_idname`** will make sure this operator's entry will be called **`bpy.ops.object.move_object`**. Operators are normally registered in order to make them usable and that is indeed the default of **`bl_options`**. However, if we also want the add-on to show up in the history so it can be undone or repeated, we should add **`UNDO`** to the set of flags that is assigned to **`bl_options`**, as is done here.

The *execute()* function

An operator class can have any number of member functions but to be useful it normally overrides the **execute()** function:

```
    def execute(self, context):
        context.active_object.location.x += 1
        return {'FINISHED'}
```

The **execute()** function is passed a reference to a context object. This context object contains a among other things an **active_object** attribute which points to, you guessed it, Blenders active object. Each object in Blender has a **location** attribute which is a vector with x, y and z components. Changing the location of an object is as simple as changing one of these components, which is exactly what we do in line 2. The **execute()** function signals successful completion by returning a set of flags, in this case a set consisting solely of a string **FINISHED**.

Registering and adding a menu entry

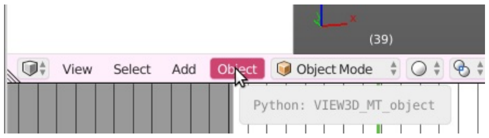
Defining an operator is not in itself enough to make this operator usable. In order for the user to find and use an operator, for example by pressing SPACE in the 3D view window and typing the label of the operator, we must *register* the operator. Adding a registered operator to a menu requires a separate action.

```
def register():
    bpy.utils.register_module(__name__)
    bpy.types.VIEW3D_MT_object.append(menu_func)

def unregister():
    bpy.utils.unregister_module(__name__)
    bpy.types.VIEW3D_MT_object.remove(menu_func)

def menu_func(self, context):
        self.layout.operator(MoveObject.bl_idname,
                             icon='MESH_CUBE')
```

When we check the **Enable an add-on** check-box in the user preferences, Blender will look for a **register()** function and execute it. Likewise, when disabling an add-on the **unregister()** function is called. Here we use this to both register our operator with Blender and to insert a menu entry that refers to our operator.

The **bpy.utils.register_module()** function will register any class in a module that has **REGISTER** defined in its **bl_options** set. In order to create a menu entry we have to do two things: create a function that will produce a menu entry and append this function to a suitable menu.

Now almost everthing in Blender is available as a Python object and menus are no exception. We want to add our entry to the Object menu in the 3D view so we call **bpy.types.VIEW3D_MT_object.append()** and pass it a reference to the function we define in the highlighted line. How do we know how this menu object is called? If you have checked **File ⇒ User preferences ⇒ Interface ⇒ Python Tooltips** the name of the menu will be shown in a tooltip when you hover over a menu.



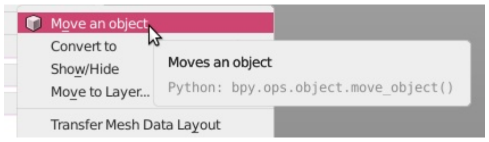
Python: VIEW3D_MT_object

From the image above we can see that we can use **bpy.types.VIEW3D_MT_object.append()** to add something to the Object menu because **VIEW3D_MT_object** is shown in the balloon text.

Note that the **menu_func()**function does not implement an action itself but will, when called, append a user interface element to the object that is passed to it in the self parameter. This user interface element in turn will interact with the user.

Here we will simply add an operator entry (that is, an item that will execute our operator when clicked). The self argument that is passed to **menu_func()** refers to the menu. This menu has a **layout** attribute with an **operator()** function that we pass the *name* of our operator. This will ensure that every time a user hovers over the Object menu, our operator will be shown in the list of options. The name of our new MoveObject operator can be found in its bl_idname attribute so that is why we pass **MoveObject.bl_idname**.

The name of the entry and its tooltip is determined by looking at the **bl_label** and docstring defined in our **MoveObject** class and the icon used in the menu is determined by passing an optional **icon** parameter to the **operator()** function. Once added our menu entry will look like this:



Moves an object
Python: bpy.ops.object.move_object()

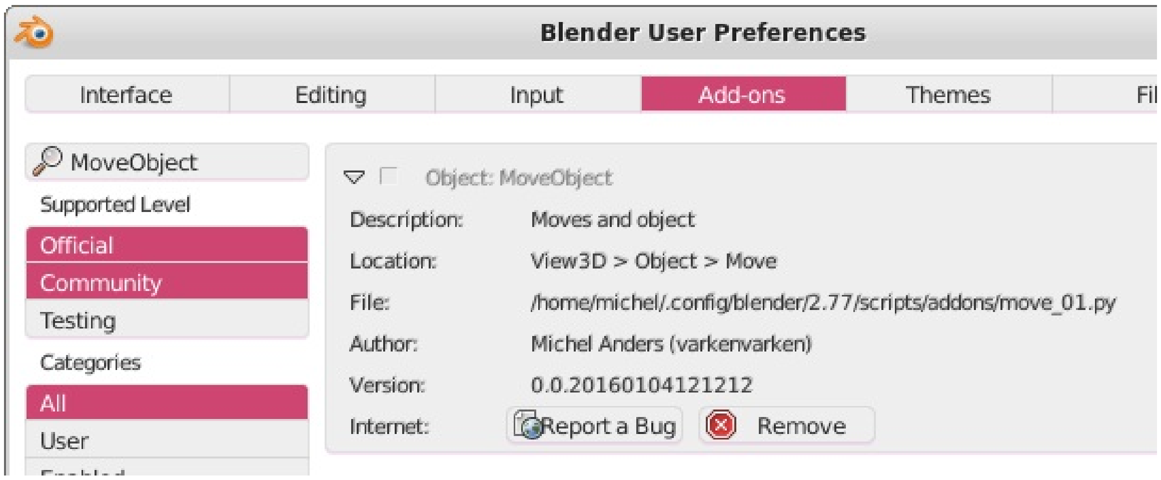
This may sound overly complicated but it makes it possible for example to show different things than just click-able entries in a menu for example to group several operators in a box.

Distributing and installing an add-on

If you typed in all the code snippets in this chapter and saved it to a file **`move_01.py`** or downloaded the complete file from GitHub the add-on can now be installed just like any other add-on:

- Click on **`File ⇒ user preferences ⇒ Add-ons`**
- Click on **`Install from file`** (near the bottom of the dialog)
- Select **`move_01.py`** in the file browser
- Click on **`Install from file`** (near the top right corner)
- Enable the add-on by checking the box after of its name

If you want to find the add-on later on, it is grouped under the `Object` category:



Object: MoveObject
Description: Moves and object
Location: View3D > Object > Move
File: /home/michel/.config/blender/2.77/scripts/addons/move_01.py
Author: Michel Anders (varkenvarken)
Version: 0.0.20160104121212
Internet:

Summary

In this chapter we created our first add-on. Its functionality was implemented by defining an operator and the add-on was integrated seamlessly into Blender's user interface by registering the operator and creating a menu entry. The add-on can also be installed by any user from the user preferences because we added proper information to the python file that describes the add-on in a manner that Blender can interpret.

This first add-on is still very simple indeed. In the next chapter we will expand its functionality with options and we will make the operator more robust.

A Blender add-on: more complexity

An add-on might be only useful in a certain context and may consist of more than one file. In this chapter we see how we can automatically disable a menu entry based on certain conditions and how we can create and distribute an add-on that consists of more than a single file. We also get a glimpse of Blender's powerful **`Vector`** class and as a bonus we'll have a look at how we can adorn our menu entry with a custom made icon. And while doing this we will be implementing an add-on that actually might be considered useful, one that allows the user to arrange a selection of objects into a circle.

Topics covered

- Enabling an operator with the **`poll()`** function
- Working with Blender's **`Vector`** class
- Distributing an add-on with multiple files
- Adding custom icons

Example: Arranging selected objects in a circle

In the previous chapter we created an add-on that could move the active object by a single Blender unit, which is about the least interesting thing imaginable. In this chapter we'll get a little bit more ambitious and create and add-on that arranges any number of selected object is a circle. It will also be a bit more sophisticated as the menu entry will only be enabled when we are in object mode and have at least three objects selected.

The step we take to implement this add-on are similar to the one we created in the previous chapter: we provide some information in the **`bl_info`** dictionary, define an operator with an overridden **`execute()`** function and make sure this operator is registered and added to a meneu.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub (**`circle_02.zip`**)*

The add-on information

In order to allow the installation of our add-on we need a proper **`bl_info`** definition at the start of our add-on:

```
bl_info = {
        "name": "CircleObjects",
        "author": "Michel Anders (varkenvarken)",
        "version": (0, 0, 201601061418),
        "blender": (2, 76, 0),
        "location": "View3D > Object > Circle",
        "description": "Arranges selected objects in a circle",
        "warning": "",
        "wiki_url": "",
        "tracker_url": "",
        "category": "Object"}
```

It closely mimics the information for our Move objects add-on as we want it to appear in the Objects category again.

Defining an operator class

We will again implement an operator, so our definition will closely resemble the the operator we defined in the first chapter. It will derive from **`bpy.types.Operator`** and we will give it a proper docstring, **`bl_idname`** and **`bl_label`**

```
class CircleObjects(bpy.types.Operator):
        """Arrange selected objects in a circle in the xy plane"""
        bl_idname = "object.circle_objects"
        bl_label = "Circle objects"
        bl_options = {'REGISTER', 'UNDO'}

        scale = 100
```

We also define a class variable **`scale`** which is just a fixed value for now. In the next chapter we will see have we can provide the end user with an option to change this.

The poll() function

Trying to arrange a selection of objects into a circle would be useless if we had no objects