Best Practice

When writing your own scripts Python is great for new developers to pick up and become productive, but you can also pick up bad practices or at least write scripts that are not easy for others to understand. For your own work this is of course fine, but if you want to collaborate with others or have your work included with Blender there are practices we encourage.

Style Conventions

For Blender Python development we have chosen to follow Python suggested style guide to avoid mixing styles among our own scripts and make it easie to use Python scripts from other projects. Using our style guide for your own scripts makes it easier if you eventually want to contribute them to Blender.

This style guide is known as pep8 and here is a brief listing of pep8 criteria:

- Camel caps for class names: MyClass
- All lower case underscore separated module names: my_module
- Indentation of 4 spaces (no tabs)
- Spaces around operators: `1 + 1`, not `1+1`
- Only use explicit imports (no wildcard importing `*`)
- Don't use multiple statements on a single line: `if val: body`, separate onto two lines instead.

As well as pep8 we have additional conventions used for Blender Python scripts:

- Use single quotes for enums, and double quotes for strings.

  Both are of course strings, but in our internal API enums are unique items from a limited set, e.g:

  ```
  bpy.context.scene.render.image_settings.file_format = 'PNG'
  bpy.context.scene.render.filepath = "//render_out"
  ```

- pep8 also defines that lines should not exceed 79 characters, we have decided that this is too restrictive so it is optional per script.

User Interface Layout

Some notes to keep in mind when writing UI layouts:

UI code is quite simple. Layout declarations are there to easily create a decent layout. The general rule here is: If you need more code for the layout declaration, than for the actual properties, then you are doing it wrong.

Example layouts:

`layout()`

The basic layout is a simple top-to-bottom layout.

```
layout.prop()
layout.prop()
```

`layout.row()`

Use `row()`, when you want more than one property in a single line.

```
row = layout.row()
row.prop()
row.prop()
```

`layout.column()`

Use `column()`, when you want your properties in a column.

```
col = layout.column()
col.prop()
col.prop()
```

`layout.split()`

This can be used to create more complex layouts. For example, you can split the layout and create two `column()` layouts next to each other. Do not use split, when you simply want two properties in a row. Use `row()` instead.

```
split = layout.split()

col = split.column()
col.prop()
col.prop()

col = split.column()
col.prop()
col.prop()
```

Declaration names:

Try to only use these variable names for layout declarations:

row:
for a `row()` layout

col:
for a `column()` layout

split:
for a `split()` layout

flow:
for a `column_flow()` layout

sub:
for a sub layout (a column inside a column for example)

Script Efficiency

List Manipulation (General Python Tips)

Searching for List Items

In Python there are some handy list functions that save you having to search through the list. Even though you are not looping on the list data **Python is**,
you need to be aware of functions that will slow down your script by searching the whole list.

```
my_list.count(list_item)
my_list.index(list_item)
my_list.remove(list_item)
if list_item in my_list: ...
```

Modifying Lists

In Python you can add and remove from a list, this is slower when the list length is modified, especially at the start of the list, since all the data after the index of modification needs to be moved up or down one place.

The fastest way to add onto the end of the list is to use `my_list.append(list_item)` or `my_list.extend(some_list)` and to remove an item is `my_list.pop()` or `del my_list[-1]`.

To use an index you can use `my_list.insert(index, list_item)` or `list.pop(index)` for list removal, but these are slower.

Sometimes it's faster (but less memory efficient) to just rebuild the list. For example if you want to remove all triangular polygons in a list. Rather than:

```
polygons = mesh.polygons[:]  # make a list copy of the meshes polygons
p_idx = len(polygons)     # Loop backwards
while p_idx:            # while the value is not 0
    p_idx -= 1

    if len(polygons[p_idx].vertices) == 3:
        polygons.pop(p_idx)  # remove the triangle
```

It's faster to build a new list with list comprehension:

```
polygons = [p for p in mesh.polygons if len(p.vertices) != 3]
```

Adding List Items

If you have a list that you want to add onto another list, rather than:

```
for l in some_list:
    my_list.append(l)
```

Use:

```
my_list.extend([a, b, c...])
```

Note that insert can be used when needed, but it is slower than append especially when inserting at the start of a long list. This example shows a very suboptimal way of making a reversed list:

```
reverse_list = []
for list_item in some_list:
    reverse_list.insert(0, list_item)
```

Python provides more convenient ways to reverse a list using the slice method, but you may want to time this before relying on it too much:

```
some_reversed_list = some_list[::-1]
```

Removing List Items

Use `my_list.pop(index)` rather than `my_list.remove(list_item)`. This requires you to have the index of the list item but is faste since `remove()` will search the list. Here is an example of how to remove items in one loop, removing the last items first, which is faster (as explainec above):

```
list_index = len(my_list)

while list_index:
    list_index -= 1
    if my_list[list_index].some_test_attribute == 1:
        my_list.pop(list_index)
```

This example shows a fast way of removing items, for use in cases where you can alter the list order without breaking the script's functionality. This work by swapping two list items, so the item you remove is always last:

```
pop_index = 5

# swap so the pop_index is last.
```

```
my_list[-1], my_list[pop_index] = my_list[pop_index], my_list[-1]

# remove last item (pop_index)
my_list.pop()
```

When removing many items in a large list this can provide a good speed-up.

Avoid Copying Lists

When passing a list or dictionary to a function, it is faster to have the function modify the list rather than returning a new list so Python doesn't have to duplicate the list in memory.

Functions that modify a list in-place are more efficient than functions that create new lists. This is generally slower so only use for functions when it makes sense not to modify the list in place:

```
>>> my_list = some_list_func(my_list)
```

This is generally faster since there is no re-assignment and no list duplication:

```
>>> some_list_func(vec)
```

Also note that, passing a sliced list makes a copy of the list in Python memory:

```
>>> foobar(my_list[:])
```

If my_list was a large array containing 10,000's of items, a copy could use a lot of extra memory.

Writing Strings to a File (Python General)

Here are three ways of joining multiple strings into one string for writing. This also applies to any area of your code that involves a lot of string joining:

String concatenation

This is the slowest option, do **not** use this if you can avoid it, especially when writing data in a loop.

```
>>> file.write(str1 + " " + str2 + " " + str3 + "\n")
```

String formatting

Use this when you are writing string data from floats and ints.

```
>>> file.write("%s %s %s\n" % (str1, str2, str3))
```

String joining

Use this to join a list of strings (the list may be temporary). In the following example, the strings are joined with a space " " in between, other examples are "" or ", ".

```
>>> file.write(" ".join((str1, str2, str3, "\n")))
```

Join is fastest on many strings, string formatting is quite fast too (better for converting data types). String concatenation is the slowest.

Parsing Strings (Import/Exporting)

Since many file formats are ASCII, the way you parse/export strings can make a large difference in how fast your script runs.

There are a few ways to parse strings when importing them into Blender.

Parsing Numbers

Use `float(string)` rather than `eval(string)`, if you know the value will be an int then `int(string)`, `float()` will work for an too but it is faster to read ints with `int()`.

Checking String Start/End

If you are checking the start of a string for a keyword, rather than:

```
>>> if line[0:5] == "vert ": ...
```

Use:

```
>>> if line.startswith("vert "):
```

Using `startswith()` is slightly faster (around 5%) and also avoids a possible error with the slice length not matching the string length.

`my_string.endswith("foo_bar")` can be used for line endings too.

If you are unsure whether the text is upper or lower case, use the `lower()` or `upper()` string function:

```
>>> if line.lower().startswith("vert ")
```

Error Handling

The **try** statement is useful to save time writing error checking code. However, **try** is significantly slower than an **if** since an exception has to be set each time, so avoid using **try** in areas of your code that execute in a loop and runs many times.

There are cases where using **try** is faster than checking whether the condition will raise an error, so it is worth experimenting.

Value Comparison

Python has two ways to compare values `a == b` and `a is b`, the difference is that `==` may run the objects comparison function `__cmp__()` whereas `is` compares identity, this is, that both variables reference the same item in memory.

In cases where you know you are checking for the same value which is referenced from multiple places, `is` is faster.

Time Your Code

While developing a script it is good to time it to be aware of any changes in performance, this can be done simply:

```
import time
time_start = time.time()

# do something...

print("My Script Finished: %.4f sec" % (time.time() - time_start))
```