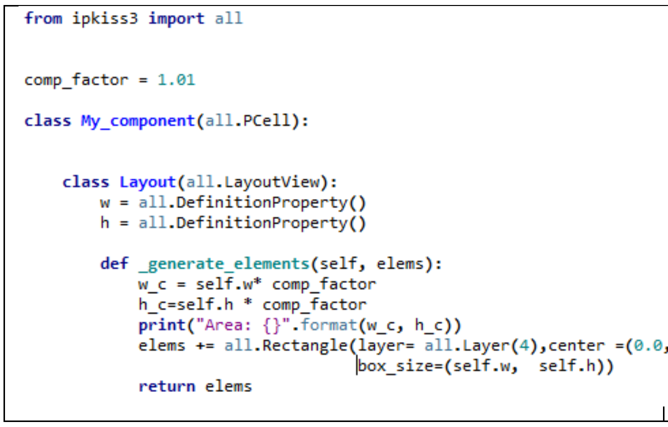
IPKISS Insight Series

Writing great code with IPKISS

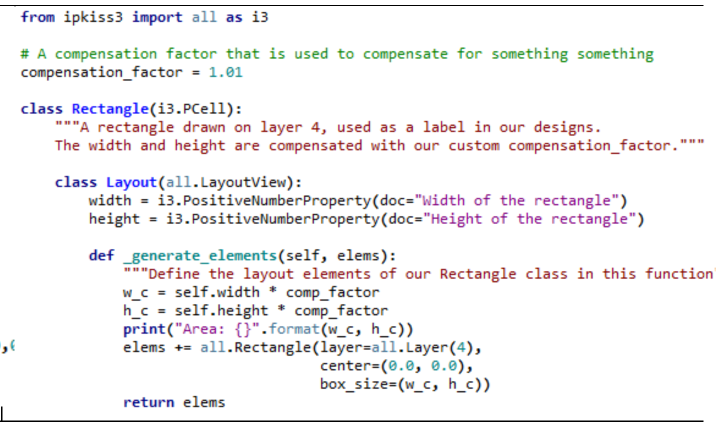
As a software company, we spend a considerable amount of time reviewing code of our colleagues. Code review - in short- means reading / testing the code that a colleague has written, giving comments, waiting for the code to be updated, reviewing again, etc. This iterative process is a great way of improving the overall quality of our codebase (quality in terms of bugs, readability, etc).

Over the years of writing a lot of Python code (and more specifically, IPKISS code) , we became more and more aligned on what we consider 'good code'. As a result, our review cycles became shorter, our code more readable, and so on.

Having a series of rules on how to write code really helps. This is not only true for me, in the role as developer, but also for our support staff, and certainly true for designers in other companies using our software. Consider the following example:



```python
from ipkiss3 import all


comp_factor = 1.01

class My_component(all.PCell):


    class Layout(all.LayoutView):
        w = all.DefinitionProperty()
        h = all.DefinitionProperty()

        def _generate_elements(self, elems):
            w_c = self.w* comp_factor
            h_c=self.h * comp_factor
            print("Area: {}".format(w_c, h_c))
            elems += all.Rectangle(layer= all.Layer(4),center =(0.0,
                                   box_size=(self.w,  self.h))

            return elems
```

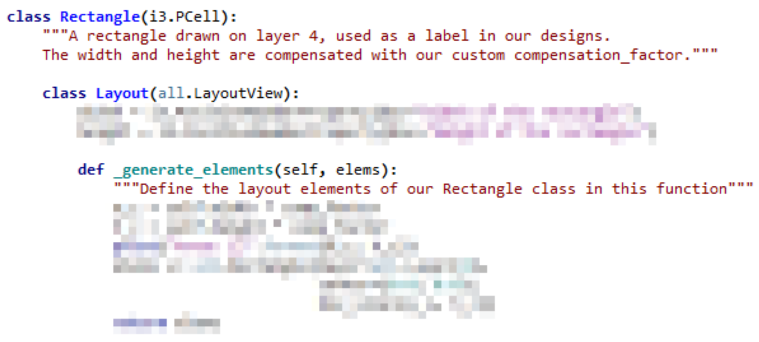


```python
from ipkiss3 import all as i3

# A compensation factor that is used to compensate for something something
compensation_factor = 1.01

class Rectangle(i3.PCell):
    """A rectangle drawn on layer 4, used as a label in our designs.
    The width and height are compensated with our custom compensation_factor."""

    class Layout(all.LayoutView):
        width = i3.PositiveNumberProperty(doc="Width of the rectangle")
        height = i3.PositiveNumberProperty(doc="Height of the rectangle")

        def _generate_elements(self, elems):
            """Define the layout elements of our Rectangle class in this function
            w_c = self.width * comp_factor
            h_c = self.height * comp_factor
            print("Area: {}".format(w_c, h_c))
            elems += all.Rectangle(layer=all.Layer(4),
                                   center=(0.0, 0.0),
                                   box_size=(w_c, h_c))
            return elems
```

Believe me, if you see a lot of code passing by which look like the code on the left, you get to appreciate a nice set of rules. Let's list what could improve on the left:

- A lot of crucial **documentation** is missing. When I read code for the first time, I scan the documentations and class structure. Without even looking at the implementation, I should be able to guess more or less what this script does. Unfortunately, the code on the left forces me to check the implementation - which sometimes even contains bugs!
- The **naming** is not consistent. There's underscores in the class names, 'all' is used instead of using 'i3' (short for 'ipkiss3'). For some variables it's not clear what they do (for simple components, w and l might still be ok, but for more complex components, this very quickly becomes cumbersome).
- A lot of **whitespaces** in different places, which decrease readability. The calculation of w_c and h_c are difficult to read because the spaces before/after the equal sign are missing.
- There is some **confusion** on the use of certain properties, such as w_c and h_c. Are we supposed to use w, h or w_c and h_c in the layout?

Now. At this point, I dare you to read ONLY the documentation lines, and the class names (doc starts with # or """) of the code on the right side (tip: you can hover your mouse over the code, it will hide all implementation details).



```
class Rectangle(i3.PCell):
    """A rectangle drawn on layer 4, used as a label in our designs.
    The width and height are compensated with our custom compensation_factor."""

    class Layout(all.LayoutView):
        [redacted]

        def _generate_elements(self, elems):
            """Define the layout elements of our Rectangle class in this function"""
            [redacted]
```

I hope that you agree with me, that writing documentation can help in understanding what the designer intended to build. It is also immediately clear that the user in example 1 (accidentally) wrote a bug: he wanted to use w_c and h_c for drawing the rectangle, but accidentally used w and h.

One could argue that if you write very good code, it doesn't need documentation (because it's self-explaining). Additionally, a drawback of writing doc is that it could get out of sync with what you implement, if you don't maintain it well. That being said, we're not software engineers, we're chip designers. We adapt ourselves to the need of chip designers, who clearly like annotations and documentation. Additionally, we used the docstrings (“””) and property documentation to generate the actual IPKISS documentation! So it clearly has certain additional advantages.

Below we'll summarize guidelines that you can use to improve your IPKISS components. Some of the guidelines are general Python guidelines (such as PEP-8), others are more IPKISS-related. A summary of this will appear in the documentation of an upcoming release.

Guidelines

General Python guidelines

1. Use PEP-8. This one requires a bit of discipline in the beginning, but once you're used to it, you don't want to come back. PEP-8 is a standard guideline in the Python community. There are even plugins that automatically convert your code to be PEP-8 compatible. Examples of PEP-8 usage are: a tab equals 4 spaces, use CamelCasing for class names, use spaces in variable definitions (a = b), but no spaces in function arguments (func(a=b, c=d)), and so on.

IPKISS guidelines

2. Write good documentation that describes the design intent. As explained in the example above, it can save a lot of trouble when other designers want to re-use your components. Documentation comes in docstrings (at the top of your design classes) and property documentation (the doc="abc" argument that can be provided to properties)

3. If you know what the property type should be, use the predefined properties, such as i3.IntProperty, i3.NumberProperty, i3.StringProperty, i3.ChildCellProperty. A lot of checking is performed

4. If you know limitations of your component, use validate properties to test these limitations.