Title: How to keep your Python code tidy with 4 easy tips.

Subtitle: Defining functions, classes, modules and config files to standardize your workflow in python.

Featured image:

Intro:

Those of us using python for data science without a solid programming background may agree that working code is good-enough code. But good enough may depend on how many times you are going to use it, and how much effort you will have to put into deciphering it in the future. Your perfectly working code can get very long very quickly and loosing track of what part of your code was doing what is only natural. I will share 4 easy tips that will help keep your code clean. Let’s get started.

Body:

When parts of your code are repeated and used several times, it is useful to wrap these sections up as **functions**. These functions have to be defined once at the beginning and can be later called repeatedly in your code in a single, very informative line.

If these functions get somewhat bigger, and especially when they need several arguments, it may be useful to **define a class** instead, with specific attributes and methods.

A further problem with an unclear script structure appears when you need to input hardcoded parameters directly into the source code (e.g., file paths). A very handy trick to make your life easier in the future, and also help out colleagues that may use your code, is to bundle and outsource hardcoded parameters to a separate **config file** (e.g., config.yaml). This file will contain all parameters that need to be adapted and will be human readable (for those not used to read code).

Last but not least, you can also outsource classes and functions to other python files and load them as **modules** in your tidy and manageable script.

Usecase:

Let’s see an example: I previously extracted 3D coordinates from a face video using DeepLabCut and Anipose (see post [here]( <https://guillermohidalgogadea.com/openlabnotebook/upgrade-your-next-zoom-meeting)>).

VIDEO HERE

The following python script (1) reads a csv dataset (2) filters some variables, (3) builds a skeleton for each time frame connecting some body parts of interest, (4) creates 3D plots at every timeframe, and (5) creates a video of the skeleton over time. Although 67 lines may still be manageable, it is easy to lose track of each block when upscaling to the entire analysis pipeline, or even when repeating the code for a second dataset.

PoseAnalysis.py :

```

code

```

# Tip 1: Define functions

The first tip is probably the easiest: find functional blocks in your code, wrap them up as a function and move them to the top of your script. In your pipeline you will only need to call the function with the arguments you specified before.

PoseAnalysisTip1.py :

```

code

```

# Tip 2: Define classes

When you have functions that build upon each other like in this example, you can run them sequentially like above, or nested, with the output of the first being the input of the second (e.g., \*create\_video\_from\_skeleton\*(data = \*create\_skeleton\*(data = coordinates), elevation = 10, azimuth = -90), which gets complex, quickly.

An alternative would be to treat both as methods of the same class. For this we define a class Pose\_3D with some attributes like the filepath it comes from, and even the filtered coordinates, and assign the functions defined before as methods for this class. In the pipeline below, you will only need to create an object of this class, and then apply the methods you need to the object.

PoseAnalysisTip2.py :

```

code

```

# Tip 3: Outsource parameters to config files

Although the code is already quite compact and nested, you will have noticed that parameters like fielpath and the elevation and azimuth for the 3D plot are still hardcoded and may need to be changed for future analysis. This is only a brief example and you probably remember what you need to change in which line, but colleagues using your code may not be aware of where to find there variables quickly (try to imagine a script with hundreds of lines and several parameters to change at different parts of the code).

A trick that comes in handy is to outsource these parameters to a separate config file, and let the classes initialize their attributes by reading the config file themselves. This way, you only need to set your parameters in the config file and then run the code, without even looking at it and trying to find parameters to change.

config.yaml :

```

code

```

PoseAnalysisTip3.py :

```

code

```

# Tip 4: Outsource functions and classes to modules

After outsourcing hardcoded parameters, the last tip is to keep going and outsource functions and classes, too. You can move these bulky blocks of code to a separate script and then load them as modules from your new sleek script, or even from the command line. Make sure to name these files properly and have them in your working directory to avoid import errors (see [documentation](<https://docs.python.org/3/reference/import.html)>).

PoseAnalysisTip4.py :

```

code

```

Conclusion:

I hope these tips help you keep your code clean and readable. Not only to optimize your workflow but also to save you some serious debugging effort in the future. Check the [Github repository]( <https://github.com/Guillermo-Hidalgo-Gadea/UQOAB/tree/main/Pose%20Analysis>) for all files described above, sample dataset and output video.