Python Athletics (py_athletics)

A tool for gathering, organizing and analyzing fitness activities

Richard Robbins

Usage

`python py_athletics.py`

`py_athletics.py` is contained in the `py_athletics/py_athletics/src/` directory of the repository. All examples in this document assume that the user's current directory is that directory. If the user is in another directory, examples with filename references included in this document or in the system's help system will not work without making changes to relative pathname prefixes. The examples are illustrative only, there are no special limits on imposed by **py_athletics** on the pathname component of filenames supplied as optional arguments to its commands.

Quick Start

The information about **py_athletics** included below is also available as part of the program's help system, accessed with the `help` command.

The `py_athletics/py_athletics/test` directory contains useful sample scripts and data sets that users can experiment with. In particular, `test/garmin_data` includes several smaller exercise data sets.

To read a full set of data, use `read ../test/Activities.csv` which provides nearly ten full months of exercise data.

To load a representative set of goals instead of crafting your own, use `run_script ../test/goals.cmd`

To restore a representative full session, use `load ../test/py_athletics.pickle`

Detailed documentation derived from the source code by `pdoc3` can be found here. That same collection is also included as part of the **py_athletics** repository in both `html` and `md` format. See the `py_athletics/py_athletics/documents/modules/` directory. Information about `pdoc3` can be found here.

The `README` file contains a comprehensive description of the user interface and examples of various commands.

Testing

One way to test the system would be to use the `read` command on a small dataset, such as `test/garmin_data/2021-01.csv` and experiment with variations of `show_activities` and `summarize_activities`, including limiting the output to particular exercises. From there you can load a larger cumulative data set, for example, the full dataset (`test/Activities.csv`) and then loading in a set of goals, either of the `cmd` files in the `test` folder will suffice. You can always use `! ls ../test/` to see what is available `! cat ../test/goals.cmd` will show the contents of that file.

To save and restore a working session, use `save` and then exit the program. Start it up again and use `load` to restore your session.

Aspirations For the Future

I built the kind of system that I want to use. It reflects, to a degree, things that I like from the Garmin fitness system as well as some concepts I see in other fitness tracking systems. If you review my original design concept you will note that my design proposal went far beyond what I was able to implement. There are two primary areas where I needed to scale back. First, I did not attempt to implement my `gear` management related concepts. Second, I was not able to make much use of attributes I had proposed that go beyond what Garmin supplies. Those limitations do not surprise me. For me, this project was as much about trying to successfully execute a minimal viable project deployment approach as anything else and about making wise decisions on where to draw the line along the way.

I would like to build out what I originally proposed and refit the project with a much more refined user interface.

I would also like to incorporate data visualization tools to bring the raw numbers alive.

That said, I am very happy with how my core loop turned out. The core loop I had in place was more than satisfactory but when I discovered the `cmd` module I was able to port my prior work easily and take advantage of the inherited help system and bash-like command history. The Python examples showed me how to build the scripting function in a very compact way.

Challenges

I struggled to get my modules and submodules organized into discrete folders and properly assembled. I would have been lost without guidance from Mumin to get over that bump.

It took me a little bit of time to get the structures in place for properly handling collections of `Activities` and `Goals` inside an `Athlete`. I started with simple lists, but then it became to difficult for me to sort and access the information I needed at various points in time. The solution I landed on in both cases was to move to nested dictionaries. By grouping `Activities` and also `Goals` by `Activity` subclass (exercise types) in separate dictionaries I didn't have to tease them apart later on. Working from data partitioned in this manner made things much easier. Here, the abstraction afforded by object oriented programming worked to my advantage. I was able to swap out the underlying methodology for handling these object collections without needing to modify code outside of the relevant parts of the `Athlete` class itself.

I need to have a more consistent and thoughtful approach to how I handle internal class references. When I began the project and needed to differentiate among `Activity` subclasses I got in the habit of passing the class object as an argument. That got to be inconvenient at times. So, as I moved forward, and especially when I focused on the user interface, I started to pass `__name__` string objects for classes instead and then would map them to the corresponding class object when necessary. My code is inconsistent in this regard. I should pick one style and stick to it.

I am not entirely satisfied with how I split handling certain concepts between the various classes. For example, when thinking about methods to handle collections of `Activities` I could take the view that managing these things in the aggregate is a task inherent to `Athlete` and therefore the methods belong there. On the other hand, I could take the position that this is something inherent in the `Activity` or `Goal` classes.

I used a few third-party utilities to help me. I like to take a step back and work from source code print outs periodically. I am reasonably satisfied with what I am getting from `encscript` but there's room to improve there. See the collection of source code pdf files in the `documents/pdf-source-listings` folder.

Similarly, I am just getting accustomed to `pdoc3` and want to explore how I can do more with it and similar tools to generate useful documentation from type hints and doc strings. I'd like to get better at type hinting, the code includes a decent amount of it, but I struggle with some of the more elaborate structures.

Regarding Code Volume

I was very sensitive to the code volume ceiling imposed on the system. I used the SLOCCount tool to count total physical source lines of code. That number comes to 847.