# Introduction to Bioinformatics using Python - Assignment

The design of this program aimed to go above and beyond a solution that simply worked as per its specification, instead aiming to achieve this in a way that accommodates future developers to easily extend its functionality in a systematic, organised manner. This report aims to describe and justify this philosophy. This report describes the main scripts, classes, methods/functions, and most important variables.

## Argument Parsing.

The use of classes and inheritance was exploited to produce modular and well-encapsulated pieces of code. Each class was responsible for one task or a set of tasks, and to do this well, without interfering or accessing any external data structures (in other words, to be ‘loosely-coupled’). If a software update needs to be made in the future, then the developer will know exactly which method to alter or which class to extend - each of which having its own defined place in the file system (see Appendix for a diagram) for this change to happen. The package provides a class named “CustomArgumentParser” which *extends* argparse’s ArgumentParser (van Rossum, 2009).

This subclass is contained within its own file. It overrides the functionality of its super class’s constructor by first invoking it and then setting a few instance variables such as its description, which is read from a file, and a list of ‘actions’ (the list of output types requested by the user) to an empty list. Actions have slightly different semantics to other arguments such as “-v” (verbose mode), and as such are treated separately.

It invokes a private ‘\_construct()’ method on itself which registers all available arguments to this instance. Since this class inherits from ArgumentParser, this is handled by the super class. The list of available arguments is modelled as a static variable (“AVAILABLE\_STANDALONE\_ARGUMENTS”) belonging to the class CustomArgumentParser where the first member of each tuple is the argument name, and the second member being its helpful description. This definition is separate from the method which registers these arguments because it’s designed to be easily extended should the format of .vchk files ever be modified in future versions. The .\_construct() method can get more complicated and this way of organising the class provides a neat separation of concerns.

In addition to the arguments required by the specification, this package also provides:

* -v, produces coloured warning and error messages to stderr throughout the execution of the program.
* A “no” option for each of the output types. E.g. “-no-st”. This was implemented as a quality-of-life feature. Suppose a user wants all outputs except the “Substitution Type” section. They can then type “-all –no-st” instead of listing all the other nine output types manually. This feature makes use of ArgumentParser’s “mutually\_exclusive\_group” (Figure 1) functionality to prevent a user from passing “st” and “no-st” simultaneously.
* -h for outputting usage instructions.

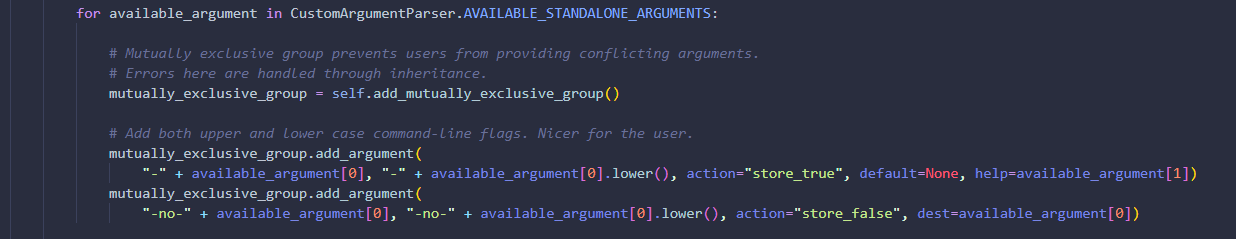


Figure 1

There exists some combination of arguments that will not cause the program to crash but are rather superfluous. For example, “st” is included within “-all” and so “-all -st” is meaningless. When the extended *.parse\_args()* method is invoked on an instance of this class, such problems are checked, and under verbose mode, a warning is issued to the user. This method was overridden to store all the user’s wanted output types into the \_actions instance variable so when this field is accessed in other areas of the program, it is clearer exactly what is being accessed (Figure 2).

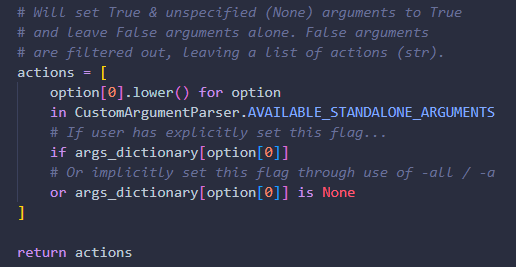


Figure 2

The superclass has its own error method, but this is overridden to make use of the package’s custom error method reporting (prints to stderr in the colour red).

When this class needs to be instantiated and arguments need to be retrieved, this can be done succinctly in the program’s main method, through using its API that makes it clear what it is doing (Figure 3).

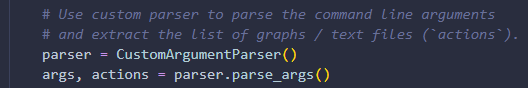


Figure 3

## Parsing the Input Files

In brief, the system for parsing an input file is comprised of three main classes:

* A top-level file-parser/handler class (FileHandler).
* A parser for some abstract section of the input file (Section).
* A StatsFile class which represents the parsed input file (StatsFileObject).

The main process for parsing a file is as follows:

1. The top-level class loads the input file into memory.
2. Line by line, it looks for a header of a section that is matched by a defined regular expression (using a for-loop: “for line in file”).
3. If there is a match, the iterator over the lines in the file is passed to a new instance of a Section (through its constructor). This parses a section by advancing the file-line iterator (by calling *next()*) and stores each line in the file so long as it doesn’t start with the pound symbol (#). It then transforms its collection of rows into a Pandas’ DataFrame (Team, 2020) so its data is easy to manipulate. It does this using Python’s ast module to transform strings into Python literals. The raw text (line by line) is also stored as a member variable so it can be written to a file later on.
4. Execution is returned to FileHandler and steps 2-3 are continued until the iterator has reached the end of the file.
5. Each of the parsed sections are collated into an instance of the StatsFileObject class. It contains a dictionary, “\_sections”, whose keys are section names (lower case, e.g. “st”) and values are the section objects.

The parser only fully parses the sections that the user wants. Specifically, the list of desired outputs (actions) is passed into the constructor of the FileHandler. When an instance of this finds a header line, it only parses the section fully if the name of this section can be found within the actions list. This saves some computation time and space in memory.

This system catches errors through “try/catch” statements. These include:

* Input files not existing.
* File not formatted correctly (missing data)
* Input data cannot be converted to appropriate data type.

## Plotting Framework and Writing Sections to Files

The plotting framework in this package makes use of a class system (Figure 4) to produce consistently themed plots. At the top level, the package defines an abstract DataPlotter class which contains: a constructor, that defines several styling parameters and stores the relevant data to produce the plot, a save() method, that saves the plot to a file, and an abstract plot() method.

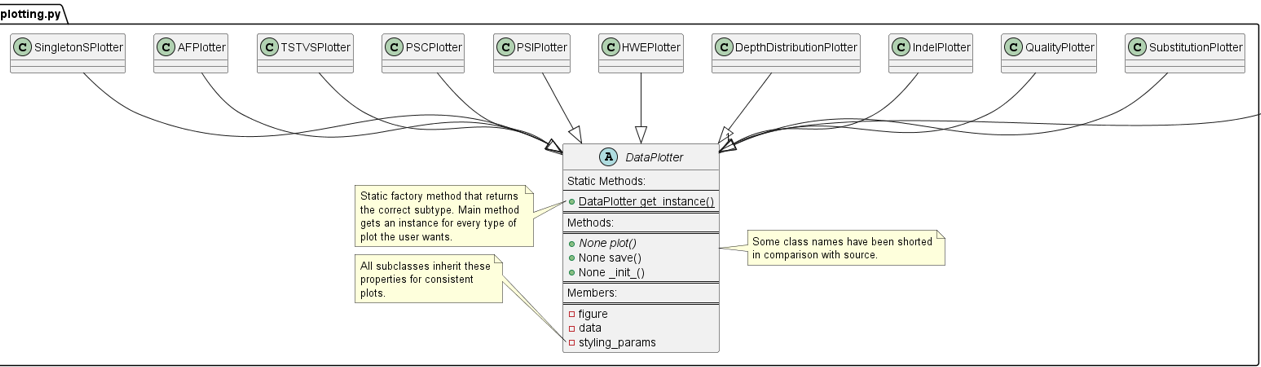


Figure 4

All subclasses inherit this common “save” method, meaning that this needs to only be defined in one place. They also inherit common styling parameters from the parent; these parameters are also only defined in one place and changing these values will modify the aesthetics of all plots at once, leaving consistently themed outputs. Subclasses can override these defaults to suit their needs, without affecting the styling of the other plots.

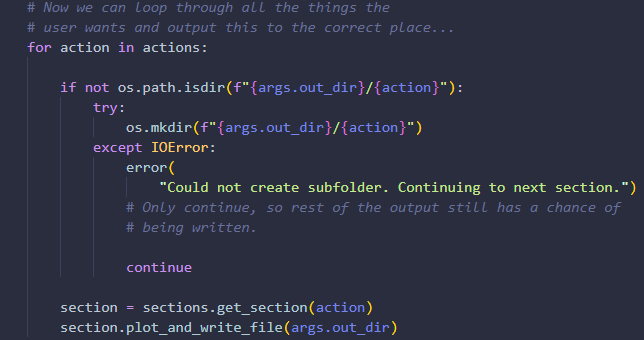


Figure 5

A section has one DataPlotter instance as a member variable that is instantiated when the section itself is instantiated. All the main program has to do is loop over the user’s desired actions to retrieve a section instance, which has a “plot\_and\_write\_file()” method, which invokes the DataPlotter’s plot() function, and writes the section’s raw text to a file.

In order to write the files to the output folder and respective subfolders, however, the program must determine whether the desired locations already exists on disk, and if not, create them. This may raise exceptions. Before the program loops through each of the actions the user wants, the program tries to create the base folder; it catches any exceptions here. If any, the program cannot continue and so terminates with non-zero exit status. If the base folder already exists, the user will be warned if they have opted into verbose mode.

The same technique is employed for creating the subfolders within the for-loop that iterates over the list of subfolders the user wants to be created. Instead of terminating the program when an exception is raised, the program simply continues to the next iteration of the for-loop. This has the benefit of producing as much output as possible (Figure 6).

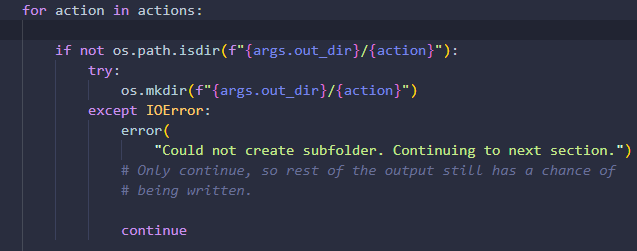


Figure 6

Once the program can guarantee these folders exist, the plots and raw text files are saved to the relevant subfolder. Note that the text files only contain the section header and its data, ***not the description.***

This class system makes use of MNE (Gramfort, 2013) to make chord plots to show strength of substitution types for the “ST” section. It also uses matplotlib (Hunter, 2007) in conjunction with seaborn (Waskom, 2021) to produce the plots. Numpy (Harris, 2020) is also used to help transform numerical data into forms suitable for plotting, where needed. (Termcolor, 2010) is used for pretty-printing messages.

The plots are designed to be well labelled and so do not need describing verbosely here. However, the substitution type (“st”) plot is depicted as a chord plot. This has the advantage of depicting two dimensions as opposed to one (the ‘to’ base and the ‘from’ base). This would allow the viewer to easily answer the question of which bases are likely to be substituted, and those which are likely to be substituted to.

## Professional Code Standards – Style Guide

All code and documentation within this package has been written in general accordance with Google’s Python Style Guide[[1]](#footnote-1). This includes docstrings. Adhering to these standards makes it easier for others to understand which in turns lends itself to being easily and readily updated. It also has the tendency to encourage the most efficient way of writing certain pieces of code.

### Type Annotations

Although Python employs a weak, dynamic type system, it is still possible to annotate method and function declarations with their return type through the ‘typing’ package. This has been used in this assignment and is what requires Python 3.10.

In conjunction with meaningful variable and function names, this combination makes for code that is very easy to read and understand.

## Utilities Module

The package contains a separate module for miscellaneous utilities that do not fit into a particular file. All warning and error messages are output to standard error, so that any normal output to stdout (the summary numbers) can be used in conjunction with the pipe operator as input to other programs. Warnings are encoded as yellow whilst critical errors are encoded as red.

## Testing and Outputs of Tests & Miscellaneous

The input data is located within ./test\_data and the outputs for each of the two input files can be found within ./test\_out/data\_1 and ./test\_out/data\_2 respectively. The commands (on linux) used to generate these plots were (**note that Python3.10 is mandatory**):

pip install . && VCHKPlotter ./test\_data/data2.vchk ./test\_out\_/test\_data\_2 -a

Refer to these outputs if the appropriate Python version cannot be installed.

Additional tests were also carried out. The Singleton Stats section was modified such that one of the rows was missing a value. The script was executed and the following output was observed (Figure 7).

The program functioned correctly; it notified the user that it could not parse that section and then it tried continuing to parse the other sections successfully.

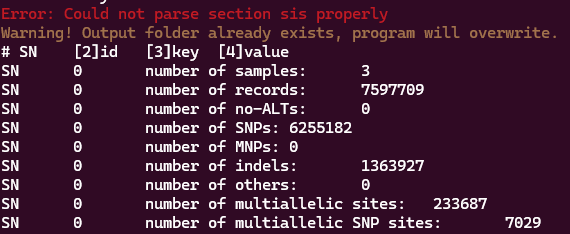


Figure 7 – Modified .vchk execution output

## Appendix

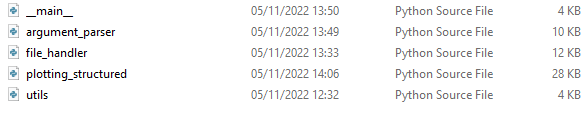


Figure 8 – Representation of the Package’s Main Files.

Refer to the manual for more precise installation instructions.

## References

Gramfort, A., Luessi, M., Larson, E., Engemann, D. A., Strohmeier, D., Brodbeck, C., … Hämäläinen, M. S. (2013). MEG and EEG Data Analysis with MNE-Python. *Frontiers in Neuroscience*, *7*(267), 1–13. doi:10.3389/fnins.2013.00267

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. doi:10.1038/s41586-020-2649-2

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi:10.1109/MCSE.2007.55

Team, T. P. D. (2020). pandas-dev/pandas: Pandas (Εκδοχή latest). doi:10.5281/zenodo.3509134

*Termcolor*. (2011, January 13). PyPI. <https://pypi.org/project/termcolor/>

Van Rossum, G., & Drake, F. L. (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

Waskom, M. L. (2021). seaborn: statistical data visualization. *Journal of Open Source Software*, *6*(60), 3021. doi:10.21105/joss.03021

1. <https://google.github.io/styleguide/pyguide.html> [↑](#footnote-ref-1)