**Heart Rate Viewer in Python (HRVY)**

**Figure 1.** *HRVY program mascot. Modelling for the logo is Harvey Franz, one of the developer’s cats. Graphic Illustration by Analia Treviño-Flitton.*

Table of Contents

**Uses** **3**

**Capabilities** **3**

**Libraries and Operating Systems3**

**HRVY Functions Deep Dive** **4**

**Functions by HeartPy 4**

**Functions by MARS Group 6**

**Directions11**

**Troubleshooting 16**

**Known Issues 16**

**Process Flow17**

\*\* *For additional information please see the Technical Report at* [*https://github.com/mars-group-2021/test\_code*](https://github.com/mars-group-2021/test_code)

**Uses**

This program was designed specifically for Nationwide Children's Hospital (NCH) by Alex Mancera, Stephen Panossian, and Analia Treviño-Flitton as part of the UMGC Master of Science in Biotechnology, Bioinformatics Specialization Capstone in Spring 2021. It was built to view ECG data from output files in .csv and .hdr format. Find the full code at <https://github.com/mars-group-2021/mars-hrvy>

**Capabilities**

1. Fills in missing data
   1. User has the option to use the modified data or original data to proceed
2. Projects ECG graphs to the screen.
   1. User can zoom in and scroll through the data

**Libraries and Operating Systems**

**Table 1**

*Python libraries incorporated into HRVY.*

|  |  |  |  |
| --- | --- | --- | --- |
| Library | Version | Content | Website |
| HeartPy | 1.2.7 | Heart rate analysis functions in Python | <https://pypi.org/project/heartpy/> |
| Plotly | 4.14.3 | Graphing libraries in Python | https://plotly.com/python/ |
| numpy | 1.20.0 | Mathematical functions for Python | <https://numpy.org> |
| scipy | 1.6.1 | Mathematical, science and engineering functions for Python | <https://scipy.org> |
| progressbar\* | 2.0 | Progress bar functions in Python (optional) | <https://progressbar-2.readthedocs.io/en/latest/> |

*\* Suggested incorporation*

**Table 2**

*Versions of Python and Operating Systems Used.*

|  |  |
| --- | --- |
| Python Version | Operating System / Platform |
| 3.9.1 | macOS v11.2.3 Big Sur |
| 3.9.2 | macOS v11.2.3 Big Sur |
| 3.9.2 | Windows 10 Home 10.0.18363 Build 18363 |

**HRVY Functions Deep Dive**

**Functions by HeartPy**

**remove\_baseline\_wander:** Corrects baseline wander. Data provided in list format (or 1-dimensional numpy array) is returned in a 1-dimensional numpy array containing filtered data. We used the ‘notch’ filter for more accuracy and some degree of noise reduction.

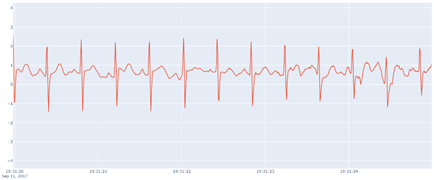
**Figure 2**

*Raw data before baseline correction()-zoomed in.*



**Figure 3.**

*Raw data after baseline correction()-zoomed in.*



**smooth\_signal:** Provides noise correction for low-level background noise. Data provided in list format (or 1-dimensional numpy array) is returned in a 1-dimensional numpy array containing filtered data.

**Figure 4**

*Raw data before smooth\_signal()-zoomed in.*



**Figure 5**

*Raw data after smooth\_signal()-zoomed in.*



**get\_samplerate\_datetime:** Calculates the sample\_rate for use in HeartPy’s functions.

**Functions by MARS Group**

**file\_opener:** Takes the user’s input file name and searches the local working directory for a .csv and a .hdr file with the same name. If they are found, they will be opened, read into lists and returned to main, exiting the file\_opener(). If a matching .csv and .hdr file are not found, but a .csv file with the same name is present, it will be sent to the cat\_file\_parse() to determine if there is an .hdr file nested in the .csv file. If there is a .cat file denoting a concatenated file found, it will also be sent to the cat\_file\_parser(). If the cat\_file\_parser() finds both the .csv and .hdr file the contents will be returned to the file\_opener() which will then package them in a tuple before returning them to main. If the cat\_file\_parser() does not find file contents, and a matching .csv or .hdr file has not been found it will return a keyword to return to main to exit the main loop causing the program to end.

**cat\_file\_parser:** Called when there is not a linked .csv and .hdr file found in the directory based on file name, but there is a .csv file present. This function will check to see if there is nested .hdr file information at the beginning of the .csv file present. If there is .hdr file information in the .csv file, it will be parsed from the .csv file. Following, both the .hdr and .csv file contents will be returned to main. If there is not .hdr file information at the beginning of the .csv file, the cat\_file\_parser() will return a keyword to exit the main loop where the end user can check their files and directory to retry the program again.

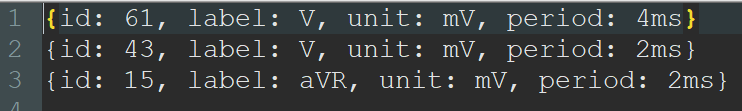
**hdr\_data:** Data collection from hdr file:

* ‘Label’ data necessary for creating header line in outfile
* ‘Period’ interval recording data (2 ms, 4 ms, etc.) necessary for filling in gaps later
* Count of columns that should exist in csv file for use in filling in blank columns

Currently, all of the data from the hdr file is stored in a nested dictionary, although only some of the data is currently used. The idea was that if other data was needed later on, it would already be readily available. As a memory saving measure, all data from the hdr\_dat dictionary is cleared out after necessary data is retrieved.

**Figure 6**

*Electrode data from .hdr file.*



**align\_plus\_dat:** Compares the number of nodes in the .hdr file to the number of columns at the beginning of the .csv file. The user is alerted that the .csv file is not aligned, the number of missing columns that need to be filled in is recorded, and the ‘label’ values from the .hdr file become the keys for the dictionary that will contain the data from the .csv file.

**time\_out:** Collects all data line by line (into ‘file\_dat’ dictionary) while:

* Generating missing timestamp data and generating time gap notification
* Filling in blank columns with baseline values
* Filling in interval recording gaps to convert all data to 2 ms interval, capable of filling in any gap of **any** interval

This is the main function of the entire program. The first minor hurdle of the program was to fill in the missing timestamp data since in the original .csv file, the datetime values were only provided every 2,560 lines (5,120 ms). This was accomplished by adding 2 ms to recorded datetime from the previous line. When the program encounters a provided datetime that doesn’t fit (i.e. is not 2 ms after the previous line’s datetime), it is assumed that there is a time gap that may have occurred if the ECG machine was disconnected for a short period of time. The program notifies the user of the time gap and its duration, and adds in new lines that contain the missing datetime values from the time gap with blank, baseline values (assumed to be 0.5) for each node.

For any nodes with a regular timer interval of 2 ms, a baseline value was inserted if there were any periods with null data. For nodes that have a longer interval (4 ms, 8 ms, 16 ms, etc.), the program automatically detects and fills in the gap taking an average of the data before and after the interval gap and filling in the values. As expected, nodes with longer time recording intervals (over 12 ms) end up having slightly inaccurate graphs.

**new\_folder:** prints the user’s chosen dataset (unaltered or corrected data pulled from the original .csv file and stored within a dictionary) to a .csv outfile with header line data pulled from the .hdr file.

**corrected\_dict:** This is where all the data correction functions above are applied to each dataset. First, get\_samplerate\_datetime() is run to obtain the sample\_rate specific for each dataset. Then, correction functions are applied in the following order:

1. flat\_peak\_reduct()
2. remove\_baseline\_wander()
3. tall\_peak\_reduct()
4. smooth\_signal()

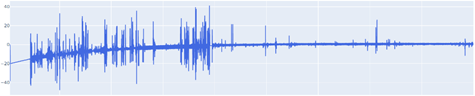
Data enters as a list and is returned in a 1-dimensional numpy array nestled within a tuple. We stored the data in a tuple as a space saving measure.

**flat\_peak\_reduct:** Performs two main functions:

1. Reduces long, flat peaks that are over 20 ms long. This detects segments of identical, repeated values sustained for more than 10 data points (10 x 2 ms = 20 ms) and reduces them down to the baseline (assumed to be 0.5). This is run before the baseline correction filter in the corrected\_dict() function to help reduce issues we found with some data sets that had impossibly tall flat peaks. Only running the baseline correction without accounting for these data anomalies would cause the entire baseline of our graphs to curve sharply. Initially, the program assumed that these were very tall heartbeats and curved the entire graph to fit them into an expected baseline (see example below). Flattening these flat “peaks” down to the baseline corrected this error.

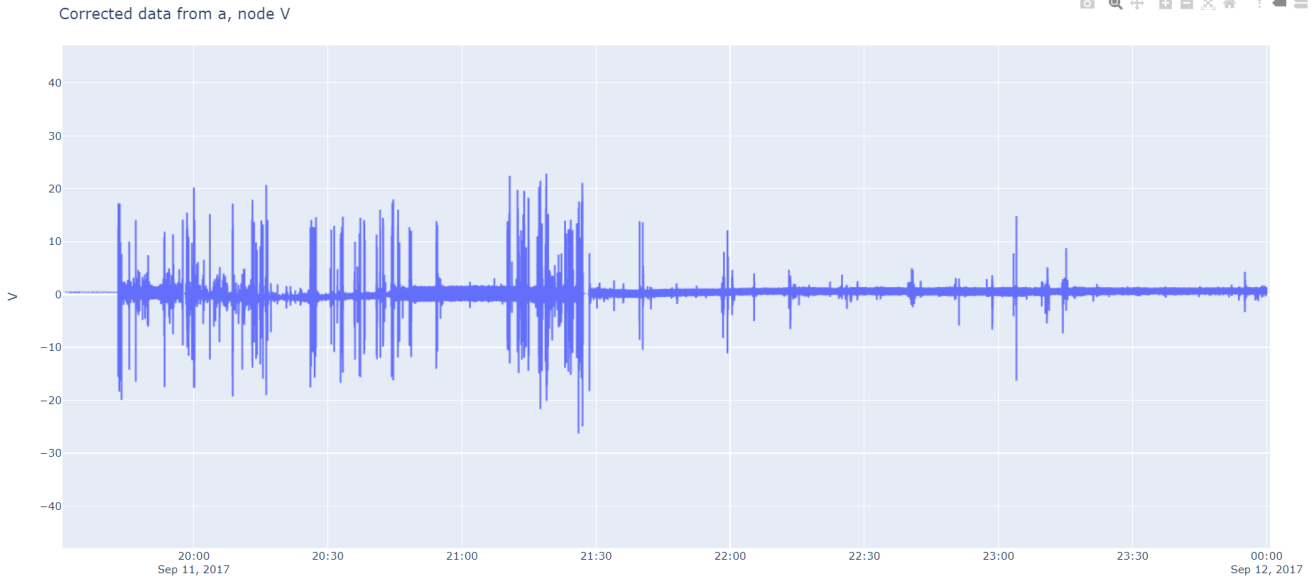
**Figure 7**

*Baseline correction before flat\_peak\_reduct.()*



**Figure 8**

*Baseline correction after the older version of flat\_peak\_reduct()*

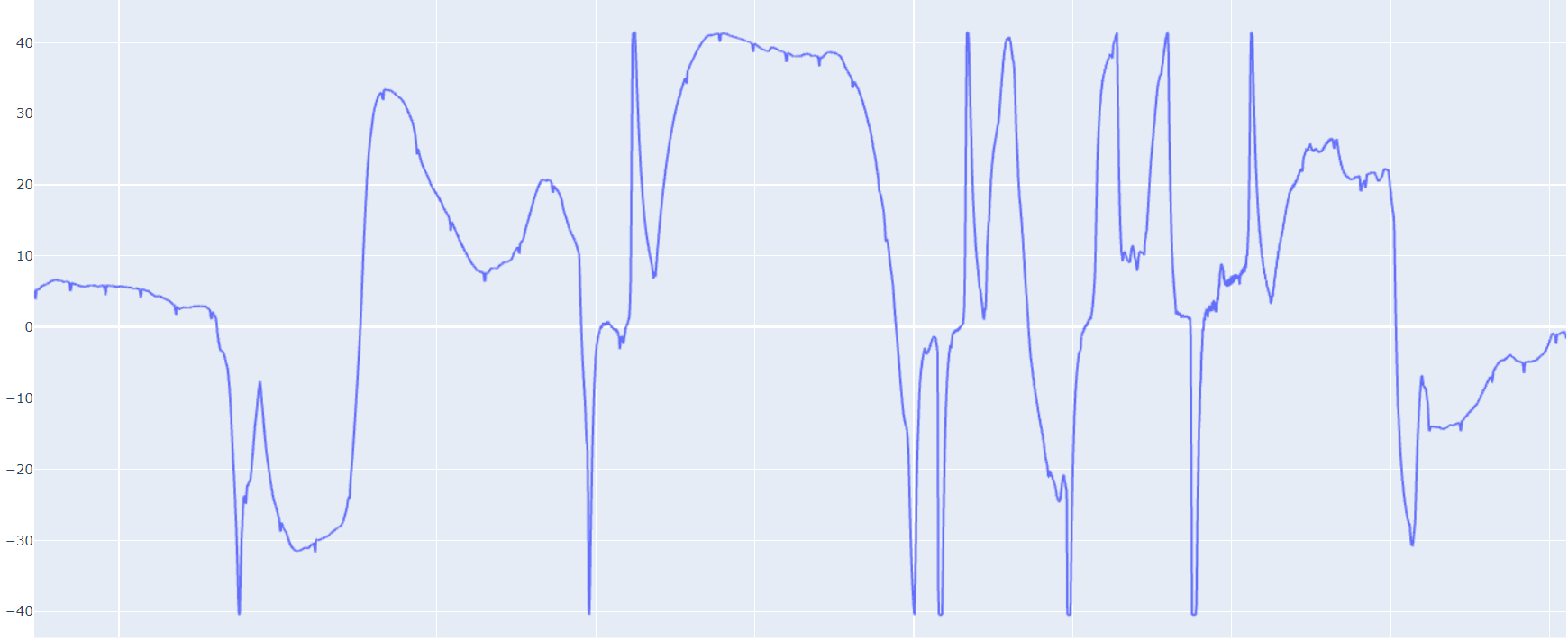


*Note*: this does not include the newer sub-function below.

1. Reduces tall, steep peaks that are outside of the normal capabilities of the human heart. This detects peaks that have a slope that is too steep and a peak that is too tall. This is also run before baseline correction to avoid issues with baseline correction causing a curved graph because of impossibly tall peaks. It is also run concurrently with the first part of flat\_peak\_reduc() to avoid iterating through the data multiple times when it is not necessary, to speed up processing time.

**Figure 9**

*Unaltered data before flat\_peak\_reduct()-zoomed in.*

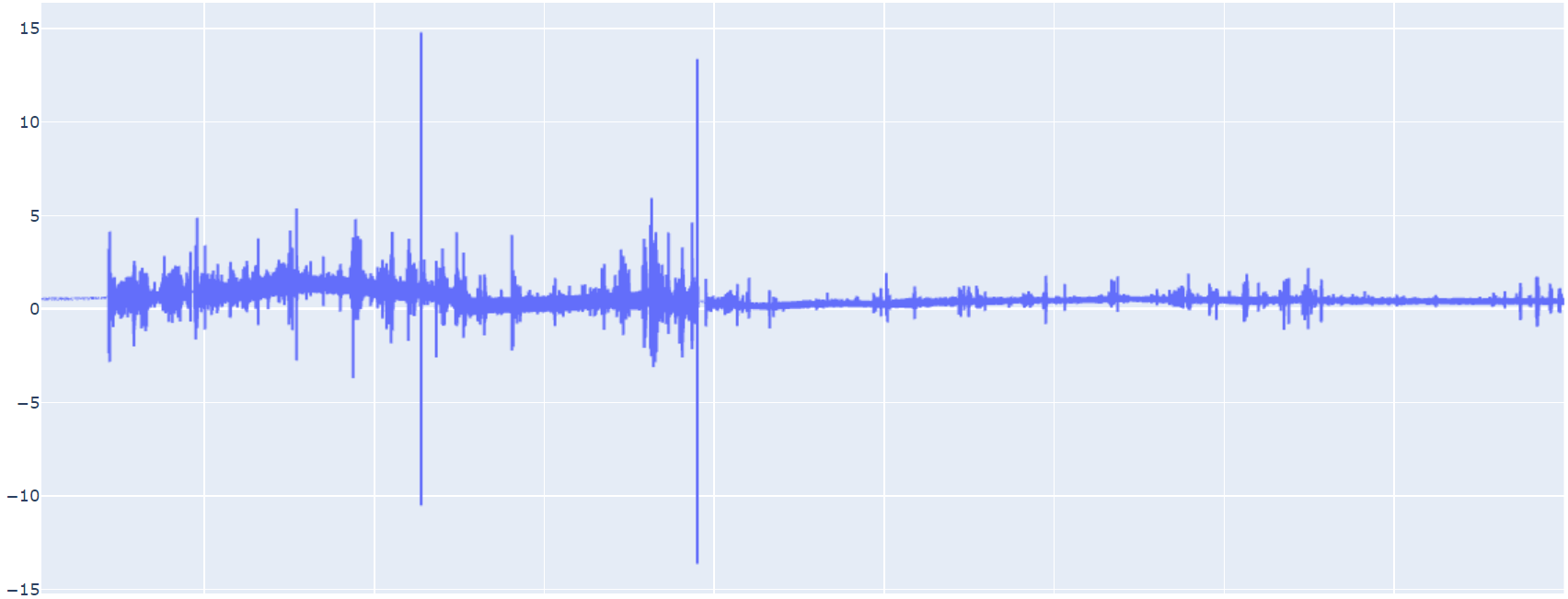


**Figure 10** *Data after only running flat\_peak\_reduct()-zoomed in.* 

**tall\_peak\_reduct:** A function nearly identical to the second part of flat\_peak\_reduct(). This is run after baseline correction to clean up any other impossibly tall peaks that may have been created in the baseline correction process.

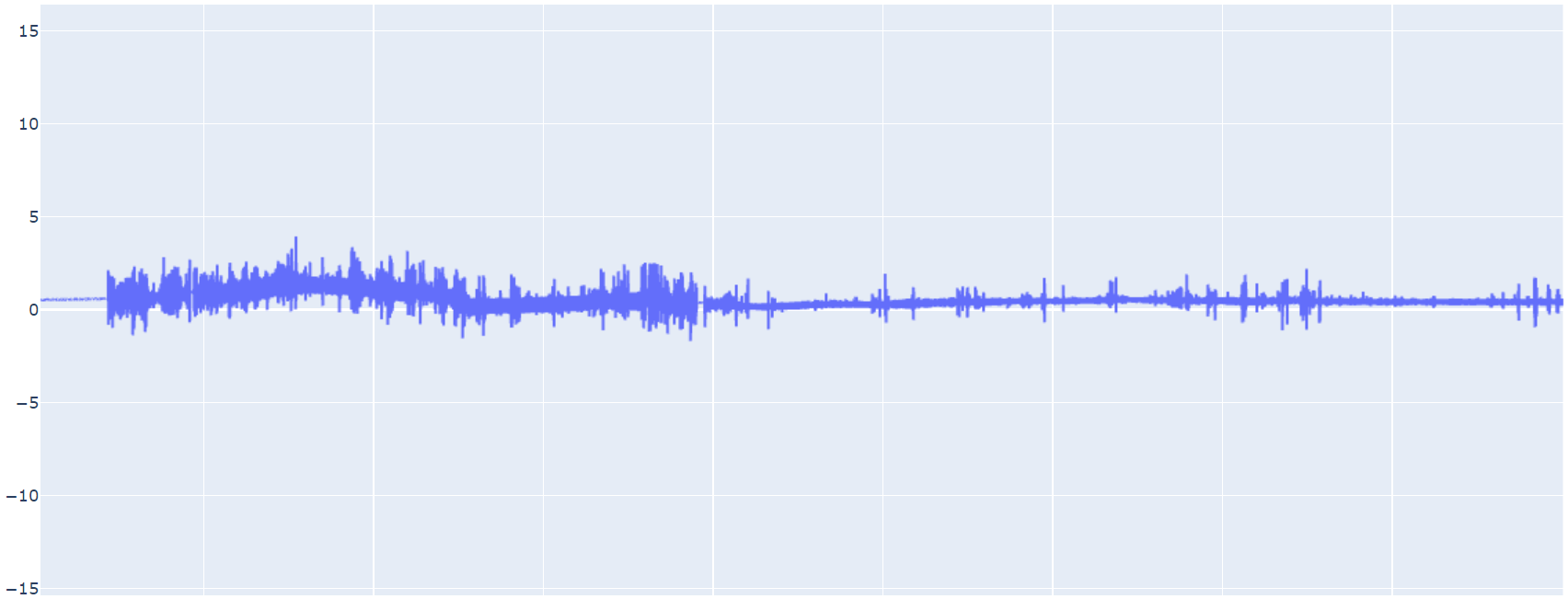
**Figure 11**

*Data after flat\_peak\_reduct() and baseline correction only.*



**Figure 12**

*Data after flat\_peak\_reduct(), baseline correction, and tall\_peak\_reduct().*



**print\_dict:** Prints the chosen dictionary (unaltered or corrected data) to a .csv outfile with header line (‘label’ data from .hdr file) in similar format to original .csv file.

**Directions**

1. Load the libraries: Ensure the libraries for scipy, numpy, heartpy, plotly, and plotly.express are installed on the local machine.
   1. Go to the computer’s command line prompt and type in:
      1. >pip install numpy
      2. >pip install heartpy
      3. >pip install scipy
      4. >pip install plotly
      5. >pip install plotly.express
2. Run the program: The team recommends utilizing the file manager of your local system as the standard method for running HRVY. However, there are alternative methods to running the program’s script, though they can become complicated and overwhelming to some. The next best direct method, that does not require any additional program or integrated development environment (IDE) downloads, would be through utilizing the command line. Additionally, we have provided advanced methods for running HRVY, though we suggest they should be utilized as the last option.
   1. Through your system’s local file manager:
      1. Linux/Unix: Open your system’s file manager and locate hrvy.py to make sure the .csv and .hdr files are located in the same folder, and then double click hrvy.py.

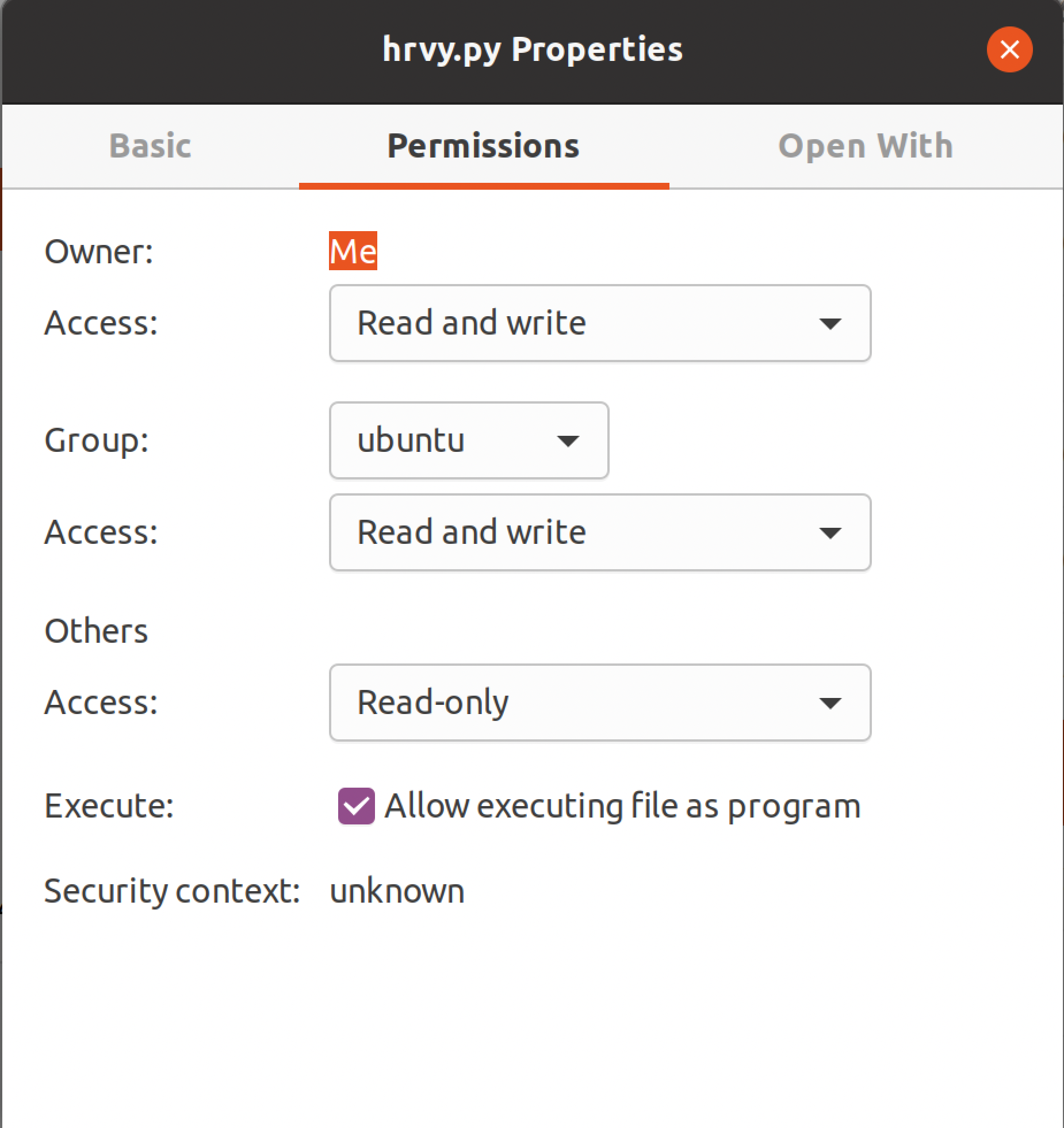
*\*\* This will only work if the hrvy.py file has execution permissions. This can be accomplished by entering:*

> chmod --x hrvy.py on the command line.

Or on a Linux system with a graphical user interface such as Ubuntu, select hrvy.py and right-click to see a pop-up menu, then scroll down to Properties. Next, select Permissions and then click the Execute box to allow hrvy.py to run as a program, as shown in Figure 12.

**Figure 12**

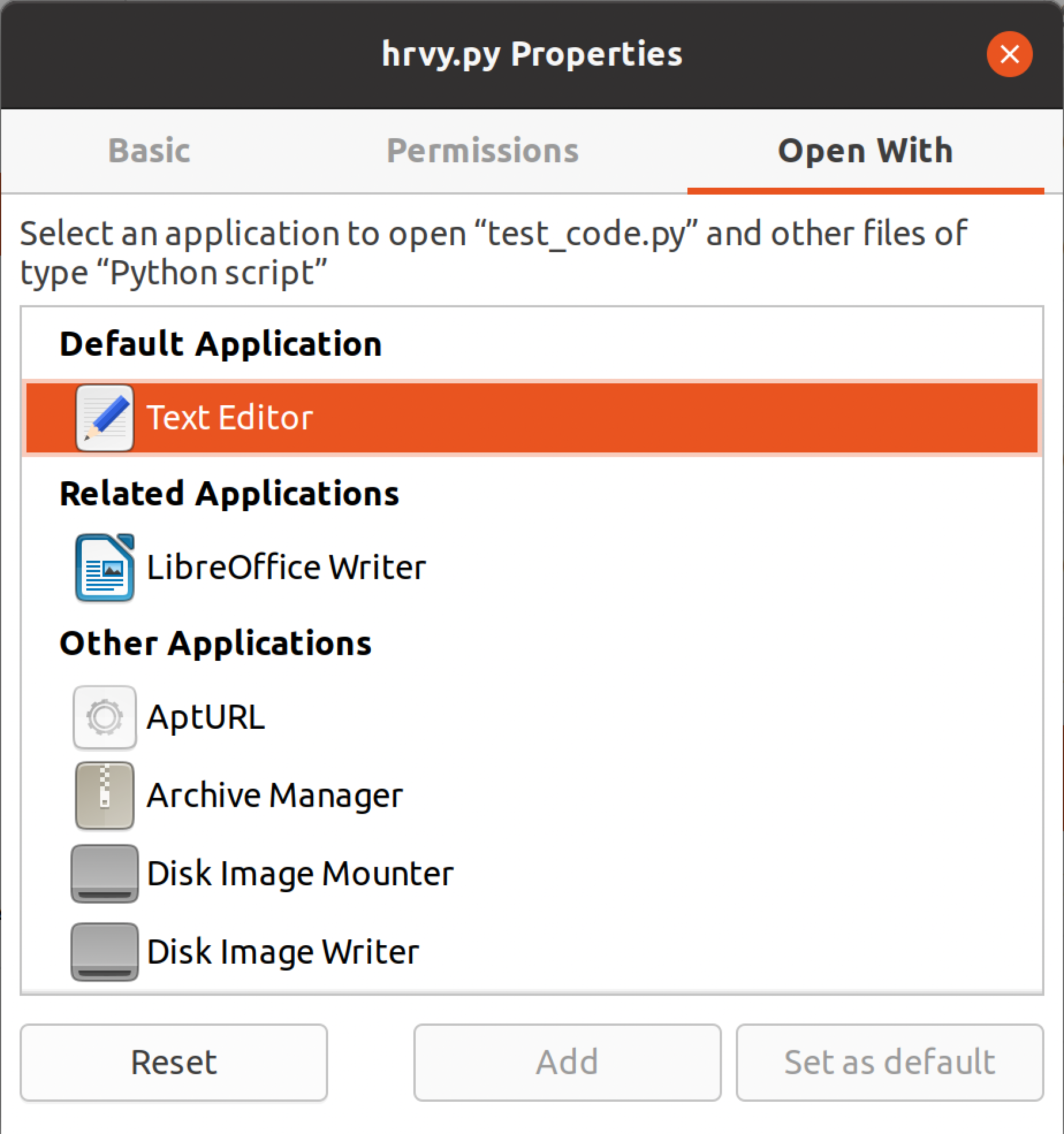
*Changing hrvy.py permissions in Ubuntu.*

**

Alternatively, you can select Open With and then choose the Python development environment you want to run hrvy.py under, as shown in Figure 13 .

**Figure 13**

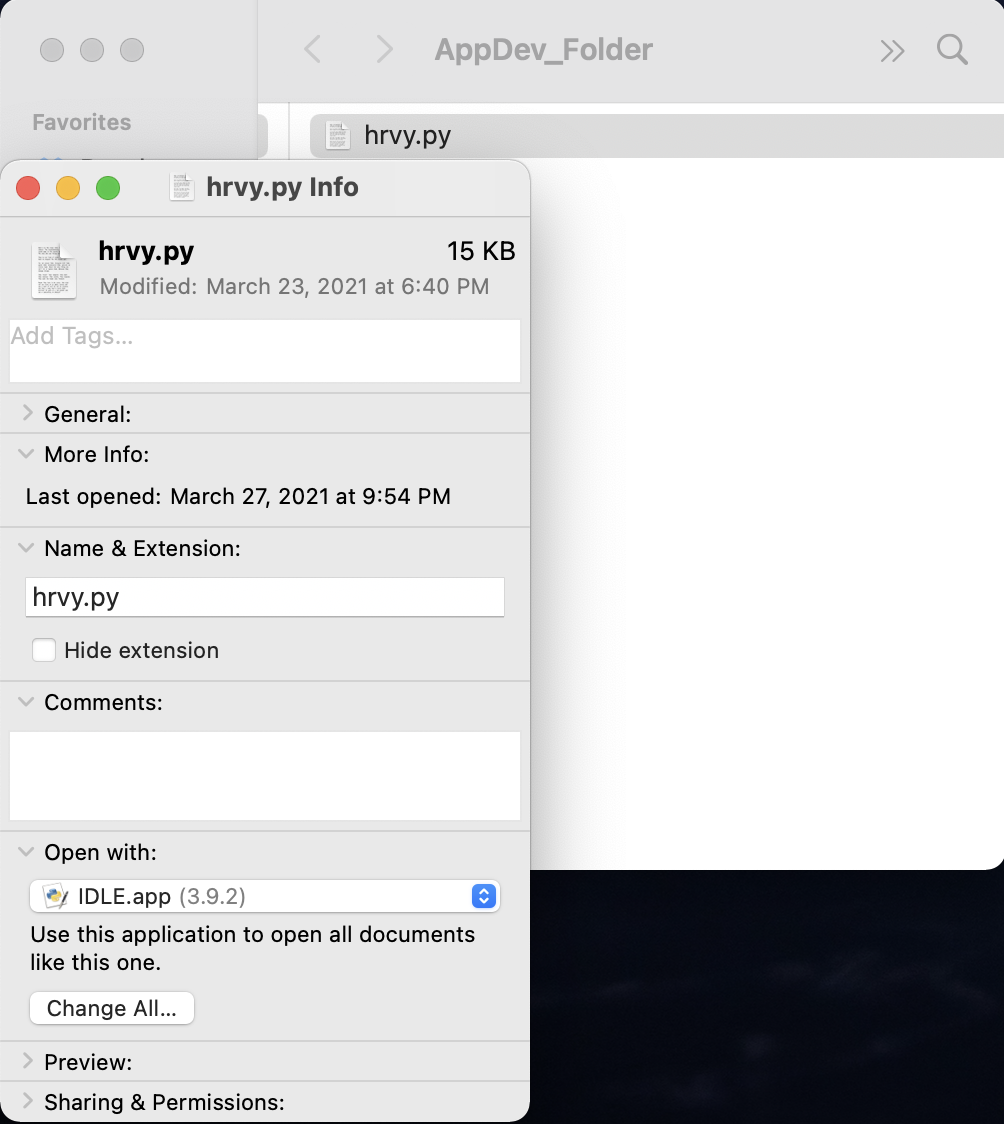
*Selecting hrvy.py’s development environment in Ubuntu.*

**

* + 1. WindowsOS: Open the file manager to locate the program’s script, hrvy.py. Double check to make sure the .csv and .hdr files are located in the same folder. If they are, proceed to double click hrvy.py.
    2. MacOS: Open a Finder window to navigate to the directory of the program’s script, hrvy.py. Double check to make sure the .csv and .hdr files are located in the same folder. Next, select hrvy.py and either right-click to select Get Info from the pop-up menu, or go to the main menu bar at the top of the screen and select Finder → File → Get Info. Then, under the “Open with” option, select the application you use typically to run .py files (as shown in Figure 14). Click the Get Info window closed, and then double-click the hrvy.py file to enter your Python development environment and run the file. (Note: you may also follow the Linux instructions in (i) above by opening a Terminal window.)

**Figure 14**

*hrvy.py’s Get Info window in macOS.*



* 1. From the command-line enter:
     1. Open the command prompt, make sure you are in the correct directory where hrvy.py and the .csv and .hdr files are located.
        1. If you need to change the directory in a Linux system enter:
           1. > cd [insert the full path]
        2. If you need to change the directory in a Windows system enter:
           1. > cd [insert the full path]
        3. If you need to change the directory in a Windows system enter:
           1. > cd [insert the full path]
     2. Once the files are in the set working directory, enter:
        1. > hrvy.py
  2. With Python’s IDE IDLE:
     1. Locate and open IDLE in your system.
     2. If you are unsure of the current working directory IDLE is using enter:
        1. >>> import os
        2. >>> os.getcwd()
     3. If this is not the directory where the .hdr and .csv files that you are planning to run are located, and you would like to change IDLE’s working directory to that folder, enter:
        1. >>> os.chdir(“[insert full path]”)

*\*\* If you do not know the full path to the folder, open the folder and right click on any file, then select properties. Find location and copy the path. Paste the path into the code but before pushing enter, make sure to change every instance of the backslash ( \ ) to a forward slash ( / ) or you will receive an error. If there are no files in the directory folder, open the folder with the File Manager and right click on the address bar to copy the path. The backslash will still need to be adjusted.*

* + 1. Make sure hrvy.py script is also in the directory folder along with the .hdr and .csv files.
    2. From the IDLE window, click on File then go to Open and click on the hrvy.py.
    3. The entire script will then open in a new window, press F5 to run the script or go to the Run tab and select Run Module

*\*\* If you receive a module error for missing packages, return to step 1 and reload the packages through the command line.*

1. Enter requested input:
   1. The program will prompt you to enter the file name or the name of the data set, it is expected to be the same for both the .hdr and .csv files. Only the data set name without any file extension is required.

*\*\*If both .hdr and .csv file contents are not found for the dataset, including in a concatenated file, the program will prompt you to double check the file name and directory, and then end.*

* 1. You will then be prompted to select the filtering method to be used in this run of the program. You are only required to enter the number
  2. Following this, you will be asked if you would like the data printed to a file in a local folder, you will have been notified if a new folder for the dataset’s data has been generated. If it has, this is where the graphs and files can be found.

**Troubleshooting**

1. Ensure the *file\_name.csv* and *file\_name.hd*r files are located in the same directory/folder as hrvy.py.
2. Make sure these open-source modules have been updated to their latest versions: HeartPy, Plotly, Numpy, and SciPy.
3. Depending on the size of the dataset, your browser may have difficulties displaying the signal plot. Be sure to clear your browser’s cache prior to loading a plot.

**Known Issues**

1. In macOS Big Sur, the filename of the output plot is not written specifically, and contains the prefix “file:///Users/. . .”. This does not affect the plot display.

**Process Flow**

***Global Variables List***

* str ( dataset ): file name
* tuple ( contents ) : A tuple of lists used to shuttle returned the .hdr & .csv file contents from the **file\_opener**
* list ( hdr ): the .hdr file contents read into a list
* list ( csv ) : the .csv file contents read into a list
* tuple ( hdr\_info ): A tuple of a dictionary and a list used to shuttle the hdr dictionary and the nodes\_not\_2ms list back to main
* dictionary ( hdr\_dat ): data dictionary with hdr contents
* list ( nodes\_not\_2ms ) : recording the node information from non-2 ms intervals
* tuple ( dat\_align ): A tuple of a tuples and a dictionary used to return the tuple align\_content and the dictionary file\_dat back to main
* dictionary ( file\_dat ): the working data dictionary
* dictionary ( final\_dat ): The final adjusted file data to be graphed with plotly and where the user will have printing options presented

***Current Functions & Variable List ( in Order of Process Flow )***

* file\_opener( dataset )
  + Utility: Used to find and open .hdr, .csv, and .cat files from within the user’s current working directory
  + Input: dataset or the name of the file
  + Return: contents (hdr and csv) the .hdr and .csv files read into separate lists or the keyword ‘end’ used to exit the main loop and end the program
  + Process:
    - Begins: The user’s current working directory is accessed and each file is searched for approved extensions with the given dataset name.
    - The first loop of the if statement will check for a .csv extension with the matching dataset name. If one is found, the nested iteration of second\_file checks the current directory for a matching dataset with an .hdr file extension.
      * If one is found, both the files will be opened using ‘open with’. By using this method, an execution block is created. This results in an ‘opening’ object getting assigned to the file, ensuring the object will be destroyed and the file closed when the block is complete. Following this, the .hdr and .csv file contents are read into a list with the respective titles and hdr\_found is set to True.
        + If there is no matching .hdr file in the directory, hdr\_found will still be set to False. The **cat\_file\_parser** will then be called to check for hidden .hdr file data in the .csv file. If the **cat\_file\_parser** returns with no hdr data, the loop will continue checking the directory for the next extension.
    - If a .cat file is found in the directory with the correct dataset name, the **cat\_file\_parser** will be called. It will return with either a tuple titled contents with the .hdr and .csv file content read in as separate lists, or it will return with the keyword ‘end’ to exit the main function loop, if no concatenated file is found.
    - Regardless of what is found in **file\_opener**, contents will get returned at the end of the function, if no .csv and .hdr file data has been found meeting the requirements, contents will be returned with ‘end’ to exit to loop in the program’s main. If the files have been found, they will be returned in the tuple.
    - Variables:
      * boolean ( hdr\_found ): to check for a matching .hdr and .csv file
      * str ( file ): each file in the user’s current working directory
      * str ( second\_file ): searching the user’s current working directory for a nested iteration
      * str ( cur\_dir ): the path to the user’s current working directory
      * str ( dataset ): the data set name or file name
      * file object ( h ): value assigned to open the .hdr file
      * file object ( v ): value assigned to open the .csv file
      * tuple or str ( contents): the value to be returned from the function. It will either return as a tuple with two lists from the .hdr and .csv file contents or it will be returned as a string with the keyword ‘end’ that will exit the main loop, print ‘not found’, and end the program.
* cat\_file\_parser( cat\_file )
  + Utility: Used to separate .hdr file contents from a .csv file when nested or concatenated
  + Input: .cat file- a concatenated .hdr and .csv file or a single .csv file
  + Return: contents\_tup a tuple with the .hdr and .csv file contents, the .csv file is named intro until it is returned
  + Process:
    - Begins: The boolean value check and a list called hdr are initialized. Additionally, an integer variable, del\_num that will identify how many lines need to be deleted from the .csv file contents to completely remove the .hdr file contents before returning the two separate file lists. The cat\_file is opened and read line by line into a list titled intro.
    - Every line in the cat\_file is checked to see if the line starts with the beginning characters of the .hdr file ('{id:').
      * If they do, then each line that starts with characters will be added to the hdr list. At the same time, one is added to the value of del\_num, which starts at zero. This is to keep track of how many lines need to be removed to completely separate the .hdr and .csv file contents.
      * If the line does not start with the beginning characters of the .hdr file ('{id:'), and there have no been any line found beginning with those characters, resulting in del\_num remaining at zero, then no file is found, the user will be notified and the function exited.
      * If there is a value for del\_num and a hidden .hdr file was found, for the value j in a range starting at 0 and ending at the line number, the cat\_file currently named intro will remove a line from the list specified at the index value zero to account for removing the .hdr file contents. Zero is used as the line numbers are shifting back down each iteration with deleting the line, so the first item in the list will always have been the next item. Now, intro has become a complete .csv file and both the .hdr and .csv file lists can be returned
    - hdr and intro are packed into a tuple called contents\_tup and returned to **file\_opener** where they will then be returned to main.
  + Variables:
    - int ( del\_num ): an integer counter to keep track of the number of lines need to be deleted from the cat\_file at the end of the function
    - list ( hdr ): holds each line of the hdr file per index position
    - object ( f ): variable where the open file object is assigned
    - list ( intro ): where all of the .csv file contents from cat\_file are saved
    - int ( i ): value assigned to count through the iteration of intro
    - int ( j ): value assigned to count through the iteration of del\_num
    - tuple ( contents\_tup ): where hdr and intro will be returned to **file\_opener**
* hdr\_data( hdr )
  + Utility: Writes the hdr file contents into a nested data dictionary
  + Input: hdr list with the .hdr file contents
  + Return: hdr\_info a tuple of dictionaries holding the hdr data in hdr\_dat and node\_not\_2ms where intervals that are not 2 ms have been saved
  + Process:
    - Begins: Two different dictionaries are intialized, they are titled hdr\_dat and node\_not\_2ms. An integer variable titled node is assigned the value of one. It depicts the starting point for the node count to keep track of the total nodes present in the data set.
    - Each index position of the hdr list represents a line of data from the original .hdr file. Everyone of these lines will be processed for reformatting to be conducive to string manipulation in Python. Every line of the hdr list will have all newline symbols and the opening and closing brackets removed. Additionally, each individual line will be further separated at every instance of the comma. This sends all the individual values into another list called l\_split.
      * While still on the same iteration of the line of the hdr index, a new nested dictionary is initialized within hdr\_dat where the node information is utilized as a key for each line of values from the original .hdr file.
        + The values for each node key in l\_split will be processed. Starting with the first value at index 0, all the colons will be removed and the string will be split at the space. This will then become a key in the nested hdr\_dat dictionary. The item at the next index position will become the value for that key.

If this item at position 0 has a string for the time ‘period’ instead of a colon and the item at index 1, which will be the value for the key, is not 2 ms then the key and value pair will also be added to the nodes\_not\_2ms dictionary where it will be reformatted similarly. The time interval will now be the key in the nodes\_not\_2ms library and the current node count is the value for that key.

* + - * + Following this, the node count will increase with each line in the hdr list, mirroring the .hdr file, that is processed.
  + Variables:
    - dictionary ( hdr\_dat [ node ] ): holds the node specifications listed in the original .hdr file
    - int ( node ): number of nodes in the hdr file
    - list ( node\_not\_2ms ): alternate node recording time interval
    - int ( line ): value assigned to access each index position of the hdr list
    - list ( l\_split ): ID, label, unit, time period info from nodes in the hdr file
    - int ( item ): value assigned to access each index position of the l\_split list
    - tuple ( hdr\_info ): returns the hdr\_dat dictionary and the nodes\_not\_2ms list back to main
* align\_plus\_dat( hdr\_dat, csv )
  + Utility: Checks the alignment of the .csv file information and generates file\_dat dictionary
  + Input: hdr\_dat dictionary and csv list
  + Returns: dat\_plus\_align a nested tuple the file\_dat dictionary and the align\_contents tuple holding one string and many integer variables utilized in **time\_out**
  + Process:
    - Begins: An empty string variable titled aligned is initialized, and the length of the hdr\_dat dictionary is assigned to num\_nodes.
    - The first item in the csv list is separated at the comma into a nested list and the length is taken. This determines how many recording nodes have contributed to the data in the file, this is assigned to num\_cols\_start.
    - To find the total number of missing nodes, or columns of missing data represented by num\_cols\_mi, num\_nodes has one add to it to offset the 0-index of Python. Then num\_cols\_start is subtracted from it.
    - If there are not any missing columns in the data, aligned will be set to yes. If there are missing columns that will later need to be filled in, aligned is set to no. The user receives a print out updating them of the alignment status.
    - The file\_dat dictionary is then initialized where the is set to the time.
      * Every key in the hdr\_dat dictionary is checked to see if it has a label value where it is added to the file\_dat dictionary as a key.
    - The hdr\_dat dictionary is then cleared and num\_nodes, num\_cols\_start, num\_cols\_mis, and aligned are packed into a tuple titled align\_contents.
    - The align\_contents tuple is then packed into another tuple titled dat\_plus\_align with the file\_dat dictionary to be returned to main.
  + Variables:
    - str ( aligned ): identifies if missing data will need to be generated in **time\_out**
    - int ( num\_nodes ): the number of nodes present in the .hdr file
    - int ( num\_cols\_start ): the starting number of columns in the .csv file
    - int ( num\_cols\_mis ): the number of missing columns from the .csv file
    - dictionary ( file\_dat ): initialized with the keys for csv data entry occurring in **time\_out**
    - tuple ( align\_contents ): holds num\_nodes, num\_cols\_start, num\_cols\_mis, and aligned
    - tuple ( dat\_plus\_align ): returns align\_contents and file\_dat back to main
* time\_out( csv, file\_dat, nodes\_not\_2ms, align\_cont )
  + Utility: Checks for time gap
  + Input: the csv list of data points, the file\_dat dictionary with only keys, the nodes\_not\_2ms list with the alternate time inveterals, and the align\_cont tuple with the column and node information
  + Return: file\_dat dictionary
  + Process:
    - Begins: The datetime package is imported and assigned an alias at the beginning of the function.
    - The align\_contents tuple is unpacked, restoring the integer values num\_nodes, num\_cols\_start, num\_cols\_mis, and the string aligned from **align\_plus\_dat**.
    - An empty string titled last\_line\_time is initialized and two\_ms is set to 2 ms from dt.
    - Each line from the .csv file in the csv list is checked to see if it doesn’t have a comma at the beginning, if it does, then the timestamp is formatted and assigned to curr\_line\_time, it is then checked for the standard 2 ms time gap.
      * If the time gap is not 2 ms and it is less than a minute away from the last time stamp, the user will be notified of the time gap in seconds. If the gap is more than a minute the user will be notified of the gap in minutes.
        + The file\_dat dictionary then has the data from the line read in as values paired to a key for the time intervals. If there is a missing time, 0.5 is added to the value as a baseline substitution.
        + The user is notified when the missing values are filled and last\_line\_time is adjusted to reflect what should be accurate for the next iteration.
      * Following this, if the dataset was missing any data points they are filled in with the baseline. If there are missing nodes, or empty columns, they are filled in.
      * If there are nodes that are not the standard 2 ms interval, the time gap or interval is filled in to account for every 2 ms based on the average of the last known values
  + Variables:
    - alias ( dt ): an alias reference for the datetime module
    - tuple ( align\_contents ): holds values identified in **align\_plus\_dat**
    - int ( num\_nodes ): the number of nodes present in the .hdr file
    - int ( num\_cols\_start ): the starting number of columns in the .csv file
    - int ( num\_cols\_mis ): the number of missing columns from the .csv file
    - str ( aligned ): identifies if missing data will need to be generated
    - dictionary ( file\_dat ): begins with keys and has values generated from the .csv file data
    - str ( last\_line\_time ): the time recorded from the last iteration and line in the .csv file
    - time ( two\_ms ): instance of timedelta from the datetime module
    - list ( csv ) : the .csv file contents read into a list
    - int ( line ): value assigned to access each index position of the csv list
    - time ( curr\_line\_time ): the time recorded for the current iteration and line in the .csv file
    - time ( time\_dif ): the calculated time difference between the current .csv file line and the previous one
    - int ( n ): value assigned to count through the iteration of num\_cols\_start
    - int ( interval ): value assigned to access the non-2 ms time gap
    - list ( node\_list ): all of the node details temporarily stored
    - int ( min\_start ): the last known value
    - int ( empty\_vals ): the values to be filled
    - int ( mis\_val\_int ): the last known value
    - int ( mult ): used for computing the average
* plot\_out( file\_dat, dataset )
  + Utility: Presents the user with data processing and printing options, then plots the data
  + Input: file\_dat the completed data dictionary and the dataset name
  + Return: - no variables returned to main, an offline Plotly chart is saved locally
  + Process:
    - Begins: The Plotly packages needed for graphing are imported at the beginning of the function and the user is presented with filtering options.
      * If the user selects the unaltered data, the final\_dat dictionary will be set to data\_pref, and the title will be set accordingly
      * If the user chooses the baseline correction and noise reduction, the data dictionary, final\_dat, will be processed through **corrected\_dict**. The new filtered data dictionary will return from **corrected\_dict** as corrected. With the corrected data set to data\_pref, the final\_dat dictionary is no longer needed so it is emptied to save memory and the title is set.
    - Following this, the user will be presented with the option to save the completed data locally for printing.
    - The final data stored in data\_pref is now accessed by the key and utilized to set the Plotly figure specifications. The Plotly graphs are then saved locally for offline viewing and data\_pref is emptied.
  + Variables:
    - int ( opt1 ): user input for processing selection
    - dictionary ( data\_pref ): the dictionary used for plotting
    - dictionary ( corrected ): the dictionary returned from **corrected\_dict**
    - str ( title ): title for the plot according to data selection
    - str ( opt2 ): user input for printing selection
    - file object ( out ): value assigned to open the .csv output file being written at the user’s request
    - str ( key ): references the dictionary keys
    - figure ( fig ): the specifications for a Plotly graph
* new\_folder( )
  + Utility: Used to create a local folder where the processed data from the data set and the completed graphs will be stored
    - Called from **plot\_out**
  + Input: - no input required
  + Return: - no true return variables within the program, a new folder is created
  + Process:
    - Begins: With the os package standard to base Python, the path of the current working directory on the local machine is accessed and the dataset or the file name is added to the end of the path.
    - In the try block, the folder is created unless there is an error throw by the operating system.
      * If the folder is successfully created, the program will let the end user know through an output message. The current working directory of the program will then be changed to the new folder before exiting the function.
      * If an error has been thrown, the end user will receive a message being notified it was not created.
  + Variables:
    - str( path ): the local path to the current working directory with the dataset name added onto the end
    - str( dataset ): the data set or file name used to title the new folder
* corrected\_dict( dictionary )
  + Utility: Generates a dictionary of the data that has been filtered with some of HeartPy’s functions, **flat\_peak\_reduc**, and **tall\_peak\_reduct**
    - Called from **plot\_out**
  + Input: final\_dat dictionary with final version of the file\_dat
  + Return: corrected a dictionary of the final corrected data that will be used for plotting
  + Process:
    - Begins: A dictionary titled corrected is initialized, the dictionary keys are searched for time labels, where they are added to the corrected dictionary and packed into a tuple. Everything else is a time data value and packed into a tuple that is called by a list of nested functions for filtering and processing the raw data. The corrected data dictionary is returned to **plot\_out** for graphing
  + Variables:
    - dictionary ( corrected ): the data that has been processed and filtered
    - str ( key ): references the dictionary keys
    - int ( sample\_rate ): sample rate used in remove\_**baseline\_wander**
* flat\_peak\_reduct( lst )
  + Utility: Reduces long flat segments to baseline
    - Called from **corrected\_dict**
  + Input: lst represents the uncorrected data
  + Return: lst represents the corrected data
  + Process:
    - Begins: Several variables are initialized, if the segment has started and the value is not at baseline, the first and last instance of repeated values are identified and stored as rv\_start and rv\_end. If the repeated values are longer than 20 ms, they are replaced with baseline and added back to the data list, lst.
    - Following this, the slope of segments are determined based on value and index, the first of each are recorded as i\_start and v\_start. The slope is checked to determine if the slope for the segment ss1 is in a reasonable range. If not they are not in what is considered a normal range, they are adjusted. Values are also checked for additional extreme outliers.
  + Variables:
    - list ( lst ): where the data originates and returns to, used in intermediate steps as storage
    - object ( rep\_val ): intermediate of repeated value
    - float ( steep\_slope ): value to be considered steep
    - float ( not\_flat ): baseline value
    - object ( i\_start ): starting index of the segment under analysis
    - object (i\_end ): last index of the segment under analysis
    - object ( v\_start ): starting value of the segment under analysis
    - object ( v\_end ): last value of the segment under analysis
    - str ( ss\_check1 ): segment slope check- Y/N
    - str ( ss\_check2 ): segment slope check- Y/N
    - int ( scale\_max ): maximum value for the scale
    - int ( ss1 ): slope standard for segment
    - int ( rv\_start ): the first repeated value
    - int ( rv\_end ): the last repeated value
    - int ( slope ): the slope of the segment
* tall\_peak\_reduct( lst )
  + Utility: Reduces unrealistically tall speaks and filters outliers. Checks if inversion of data is needed, if it is, **negdata\_flip** is called
    - Called from **corrected\_dict**
  + Input: lst represents the uncorrected data
  + Return: lst represents the corrected data
  + Process:
    - Begins: Follows the same process at **flat\_peak\_reduc** checking the slope of values looking for outliers. If abnormal outliers are found, they are adjusted back to baseline.
    - Next, samples of 1000 data points are tested to determine if the data is inverted. Only uneventful samples (ones that had neither tall or steep peaks) qualify for evaluation.
    - The minimum, maximum, and median values are recorded for the sample data set. The data points are scanned to determine what part of the range they are in.
      * If at least 75% of the data points are in the top half of the sample range, the segment is considered inverted. Likewise, if 75% of the data points in the dataset occur in the bottom half of the range, that segment is considered normal. A sample with a distribution of values outside of these parameters is considered to have an unknown orientation. These classification guesses are added to a list inv\_guesses.
      * After iterating through the entire dataset, all of the guesses are analyzed and a probability is calculated to determine the likelihood of the data being inverted. If at least 60% of the guesses are that the data samples are inverted, the entire dataset is assumed to be inverted. The **negdata\_flip** function is then called and the inverted data is returned.
  + Variables:
    - float ( steep\_slope ): values considered abnormal
    - float ( not\_flat ): baseline value
    - object ( i\_start ): starting index of the segment under analysis
    - object (i\_end ): last index of the segment under analysis
    - object ( v\_start ): starting value of the segment under analysis
    - object ( v\_end ): last value of the segment under analysis
    - str ( ss\_check1 ): segment slope check- Y/N
    - str ( ss\_check2 ): segment slope check- Y/N
    - int ( scale\_max ): maximum value for the scale
    - int ( ss1 ): slope standard for segment
    - int ( samp\_start ): starting index of the sample
    - int ( samp\_end ): last index of the sample
    - list( inv\_guesses ): guess at classification- inverted, normal, or unknown
    - int ( max\_sv ): maximum sample value
    - int ( min\_sv ): minimum sample value
    - int ( mid\_sv ): median of sample value
    - int ( top ): first half percentage from the sample
    - Int ( bottom ): second half percentage from the sample
* negdata\_flip( data\_selection )
  + Utility: Actively writes to the out file with the completed data dictionary
    - Called from **tall\_peak\_reduct**
  + Input: data\_selection, the data to be evaluated
  + Return: out\_array the inverted signal data
  + Process:
    - Begins: Required specifications for the HeartPy function **flip\_signal** are initialized, the function is then called and the results are returned
  + Variables:
    - boolean ( enhance\_peaks ): required for the **flip\_signal**
    - boolean ( keep\_range ): required for the **flip\_signal**
    - list ( out\_array ): the inverted signal data
* print\_dict( dictionary, outfile )
  + Utility: Actively writes to the out file with the completed data dictionary
    - Called from **plot\_out**
  + Input: dictionary, the final data dictionary and outfile the file object
  + Return: nothing to the program,the file written is closed upon completion
  + Process:
    - Begins: A list titled header is initialized where every key in the dictionary is added. The header lines of the output file is then written.
    - A new list titled print\_line is initialized where both the key and value stored in the dictionary are added for printing format assembly. The line is written to the line and print\_line is then emptied for the next line.
  + Variables:
    - list ( header ): accesses the header values from the dictionary they were stored in for printing
    - str ( key ): references the dictionary keys
    - list ( print\_line ): temporarily holds the formatted output file line to be written