ECE 364 Prelab Assignment 09

Modules and Exceptions

Due: October 28, 2018

This prelab contributes to course objectives CO2, CO3, CO7

Instructions

- Work in your Prelab09 directory.
- Copy all files from ~ee364/labfiles/Prelab09 into your Prelab08 directory. You may use the following command:
  `cp -r ~ee364/labfiles/Prelab09/* ./`
- To submit, commit your files to SVN. We will only grade the version of the file that is in SVN.
- Make sure you file compiles. **You will not receive any credit if your file does not compile.**
- Name and spell the file, and the functions, exactly as instructed. Your scripts will be graded by an automated process. **You will lose some points, per our discretion, for any function that does not match the expected name.**
- Make sure your output from all functions match the given examples. **You will not receive points for a question whose output mismatches the expected result.**
- Do not use any external library unless specifically allowed. You may use any module in the **Python Standard Library**.
- Use Python 3.4 (not 2.x, 3.3, 3.5, etc.). In PyCharm, go to:

  File Menu → Settings → Project Interpreter

  Make sure Python 3.4 (`/usr/local/bin/python3.4`) is selected.

Failure to meet all base requirements will result in zero credit. This includes submitting code with syntax errors. Test your code with Python 3.4 before your final submission. Be careful not to leave a dangling `if __name__ == "__main__":` (i.e., without an indented code block below it). If you did any part of your work outside of ecegrid, be sure to test with Python 3.4. Code that works with other versions of Python (e.g., 2.x, 3.3, 3.5, etc.) may cause syntax errors when run in Python 3.4.

Modules and Exceptions

Create a Python file named `module_tasks.py`. This file should only consists of function blocks, and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`). Do not include any loose Python statements. You may write any number of additional utility/helper functions, as long as they are called (directly or indirectly) from at least one of the required functions, or from your test code under `if __name__ == "__main__":`.

Part I

- **[10pts]** Assume that you are given a module called `ex_module`, and that module has a function called `run_network_code(**kwargs)` that performs network testing and may throw one or more exceptions based on the network condition. Write a function called `check_network(**kwargs)` that invokes the aforementioned function and passes the same input parameters to it. Your function should behave as follows:

  - If the invoked function does not throw an exception, return `True`.
  - If it throws one of the OS Errors, return the string:
    `"An issue encountered during runtime. The name of the error is: <Name of Exception>"`
    where `<Name of Exception>` is the name of the exception thrown, e.g. `BlockingIOError`, `InterruptedError`, etc.
  - If it throws a `ConnectionError`, re-throw that error.
  - If it throws any other exception, return `False`.

  Notes:

  - In this exercise, you are supposed to guard against the behavior of a function whose code you did not write but you need to consume.
  - To simulate the desired function, create a file called `ex_module.py`, create the function in the file as:

    ```
    def run_network_code(**kwargs):
        pass
    ```

    then import it in your code as:

    ```
    from ex_module import run_network_code
    ```

  - Do NOT check in the file `ex_module.py`, as we will have our own implementation. You can modify the content of the function as you wish to test your code.

Part II

A sensor-array system records several sensor signals into text files, where each signal is stored in its own file. The file name is formatted as `<signal_name>.txt` where the signal name must be 3 uppercase letters, followed by a hyphen '-', then three digits, e.g. `DAS-900` and `SWE-314` are both valid signal names. These files may have been corrupted, and your task is to process them to obtain proper data. (The folder `signals` contains some files for you to test, but we will use a different folder name, with a different set of files, for grading.)

Continue working with the same file, `module_tasks.py`, and add the following functions to it.

Requirements

- **[10pts]** Write a function called `is_ok(signal_name)` that takes a signal name string as its argument and returns `True` if the name is valid, otherwise returns `False`.
- **[20pts]** Write a function called `load_data_from(signal_name, folder_name)` that takes a signal name as a string, and a folder name to search in. Note the following:
  - You must check the signal name passed. If the signal name is invalid, raise a `ValueError` with some message containing the bad signal name, e.g. `"<signal_name> is invalid."` where `<signal_name>` contains the passed argument.
  - If the signal name is valid, but the file is *not* present in the `signal_folder`, raise an `OSError` with some message containing the name of the missing file.
  - The file may contain non-float data, which cannot be read and should be ignored. If so, you must keep track of their count.

  This function should return the tuple `(list, int)` where the first element is a list of **float** values, and the second element is the number of non-float values. Note that the list may be empty, if no valid floats were present.
- **[20pts]** In this system, most signals fluctuate within known bounds, but occasionally they may overshoot outside these bounds. If the number of samples outside the bounds is less than some threshold, the signal is considered acceptable. Otherwise, we consider the signal completely corrupt due to an overloaded sensor.

  Write a function called `is_bounded(signal_values, bounds, threshold)` that takes a list containing the signal values, and the tuple, `bounds`, containing two float values representing the expected range, i.e. minimum value and maximum value, of that signal, and an integer, `threshold`, containing the acceptable maximum number of values *outside* the valid range.

  This function should return `True` if the number of float values outside the valid range is less than or equal to the acceptable maximum value. Otherwise, return `False`.

  Note: If the signal contains no values, raise a `ValueError` with the message: `"Signal contains no data."`

Part III

In this part, create another Python file named `signals.py`, and **import the module** you implemented earlier to consume its functionality. This file should only consists of function blocks, and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`). Do not include any loose Python statements, but you can, however, write any number of additional utility/helper functions that you might need.

Requirements

- **[20pts]** In the previous part, you wrote a function to load a signal, and it kept track of the number of non-float values. These bad values may indicate a problem with the sensor, so we may decide to accept or reject a signal based on the number of bad values.

  Write a function called `load_multiple(signal_names, folder_name, max_count)` that takes in a list of signal names, a folder to search in, and the acceptable maximum number of non-float values present in a signal file. This function should return a dictionary, where the key is the signal name, and the value is one of the following:

  - If the signal name is not valid, or if the signal file is not present for that signal name, the dictionary value should be `None`.
  - If the signal name is valid, and the number of non-float values is less than or equal to the passed parameter, indicating a good sensor, we are going to accept that signal. For that signal, the dictionary value is a list of the float values of that signal.
  - If the signal name is valid, and the number of non-float values is larger than the passed parameter, indicating a bad sensor, we are going to reject that signal. For that signal, the dictionary value is an empty list.

- **[20pts]** In the previous function, we handled all non-float values, and now we are interested in verifying that the signal bounds are within an acceptable range. We are also interested in saving the good data to a different folder.

  Write a function called `save_data(signals_dictionary, target_folder, bounds, threshold)` that takes in the dictionary produced by the previous function; the name of an empty target folder (assume that folder already exists); the tuple, `bounds`, containing two float values representing the expected range, i.e. minimum value and maximum value, of every signal; and the integer, `threshold`, containing the acceptable maximum number of values *outside* the valid range of every signal.

  This function should not return anything, but for every acceptable signal, i.e. if the number of float values outside the valid range is less than or equal to the acceptable maximum value, write that signal to the target folder, using the file name `<signal_name>.txt`, and following the same format as the input file, i.e. one value per line, formatted to have 3 decimal digits (You may use the format specifier `".3f"`). If the signal is not acceptable, you should not write it to a file.