PROGRAMMING 2

Final Assignment 27-5-2022

Hendrik Reitsma
h.reitsma@st.hanze.nl

Contents

1. Code description

The code is from the project of semester 1. The goal was to create a dashboard that supports the research project with the data collected during the research.

The data consists of fitbit data and questionnaire data on sleep, diet data from cronometer and self-reported data consisting of amount of pushups, benchpress max, jump height and scale data including bodyfat- and muscle mass percentage.

The code is written in a notebook. Running the whole notebook will do everything from loading the data, cleaning and preparing the data and creating the dashboard and making separate plots used in a presentation.

Notebook

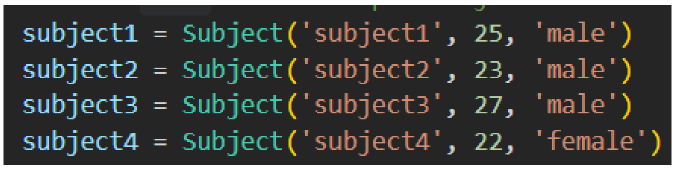
Because all of the code is in one notebook, it is difficult to maintain the code. The code consists of classes, loose functions and code outside of functions. It is not obvious where the classes are instantiated. Because of the code outside of function it is difficult to see where the variable changes. It would be better to have multiple scripts that contain a limited amount of code with a naming method that makes it easy to understand what the script does. This way, if the code has to be maintained, the right code can be found easily. It is also better to do it this way for reuseability. It is a lot easier to import any of these functions or classes in a different project if they were in separate scripts. The scripts would then be run by a main.py script that deals with all the instantiating classes and running of the code. An example of what the structure of the files can be is seen below:

```
my_project/
├── main.py
├── requirements.txt
└── src/
    ├── preprocess_data.py
    ├── graphs.py
    ├── dashboard.py
    └── stats.py
```

2. Classes

The notebook has two classes. The class dataParsers is not used in the code at all. It is clearly a piece of code that was used earlier but is now redundant. When a piece of code is not being used it should not be there.

The subject class is a good use of a class. For every test subject in the study the data has to be loaded, processed and merged. By putting these function in a class and making a different instance the code can be reused for new subjects. Reading the files can be done more efficient. All the files in the data folder can be listed and then this list can be used to read all of the files. The name of the file can be used to identify the different subjects. Thereafter, all the desired data preprocessing steps can be conducted. This way there doesn't have to be an instance of this class for every new subject (see picture below)



```
subject1 = Subject('subject1', 25, 'male')
subject2 = Subject('subject2', 23, 'male')
subject3 = Subject('subject3', 27, 'male')
subject4 = Subject('subject4', 22, 'female')
```

This codebase does not make use of classes very well. There should be separate classes handling the data cleaning, preparation, visualization, and statistics. These classes can be in different files with a logically named folder structure so that it is easier to maintain.

3. Meaningful Names, Comments and Formatting

Meaningful names

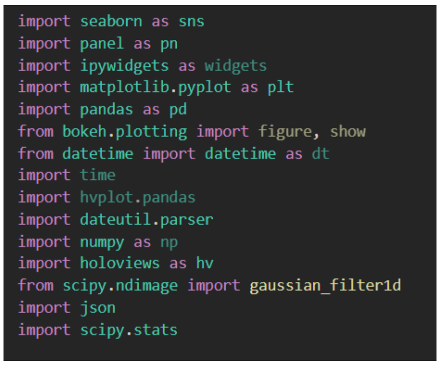
The Subject class name does not really say anything about the functionality of the class. Most of the other functions in the class have descriptive names. The names of a class, function or variable should explain why it exists, what it does and how it is used. If a comment is required then the name is not meaningful.

Comments

In the codebase there are some comments explaining code. This is necessary because some of the functions are too long or the naming is not descriptive enough. If these problems are fixed, then the comments can go as well.

Formatting

Some of the imported libraries and functions are not used (see figure to the right). This is undesirable in code. Also the order of importing should be according to the PEP-8 import ordering. First the standard library imports, then the related third-party imports and lastly the local imports.

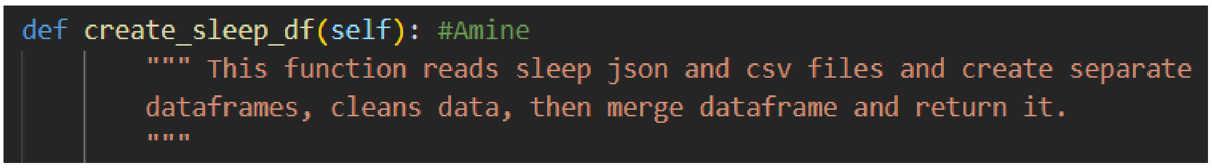


```
import seaborn as sns
import panel as pn
import ipywidgets as widgets
import matplotlib.pyplot as plt
import pandas as pd
from bokeh.plotting import figure, show
from datetime import datetime as dt
import time
import hvplot.pandas
import dateutil.parser
import numpy as np
import holoviews as hv
from scipy.ndimage import gaussian_filter1d
import json
import scipy.stats
```

The functions have descriptions that tell the user what the function does. A function should be short and simple enough to see what is going on without a description. The docstrings are mostly written according to PEP-257.

4. Functions

Function length



```
def create_sleep_df(self): #Amine
        """ This function reads sleep json and csv files and create separate
        dataframes, cleans data, then merge dataframe and return it.
        """
```

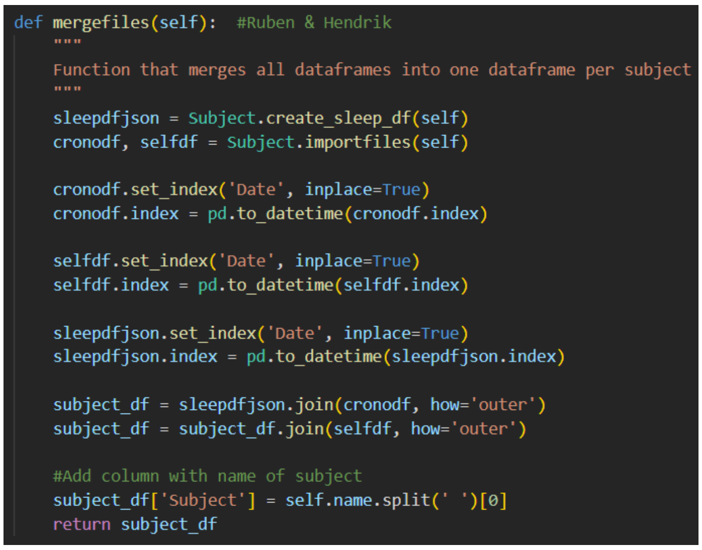
The method create_sleep_df (see above) is too long. When a function needs explanations, it is an indication that the method has too many lines, does more than one thing or does not have a descriptive name. The features method ticks all three boxes. It reads multiple files, extracts the relevant data and merges them. This goes against the single responsibility principle.

One of the reasons the function is too long, is because it has repeat code in it. For example, another method also has the function of merging different dataframes. Converting the json file to a pandas dataframe can probably be done more efficiently with the use of libraries.

The longer a method is, the harder it becomes to understand and maintain it. Also, a long method can have unwanted duplicate code.

Reusable functions

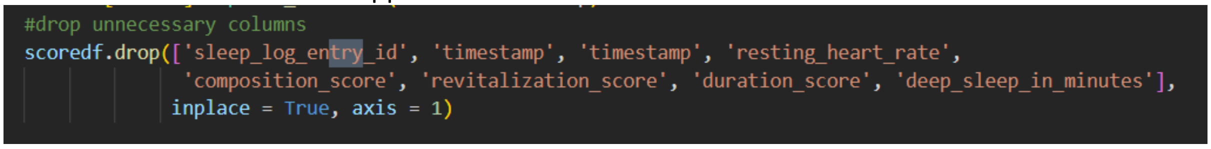
The mergefiles function is a good example of a function that is not reusable. It merges dataframes into one dataframe and adds the subject name. There are no arguments passed to the function. The variables are created inside the function so that it can not have any other functionality other than the one that was needed at the time of writing. A better way would be to create a function that takes any number of args (dataframes) and merges them. Adding the name of the subject to the dataframe is something for another method.



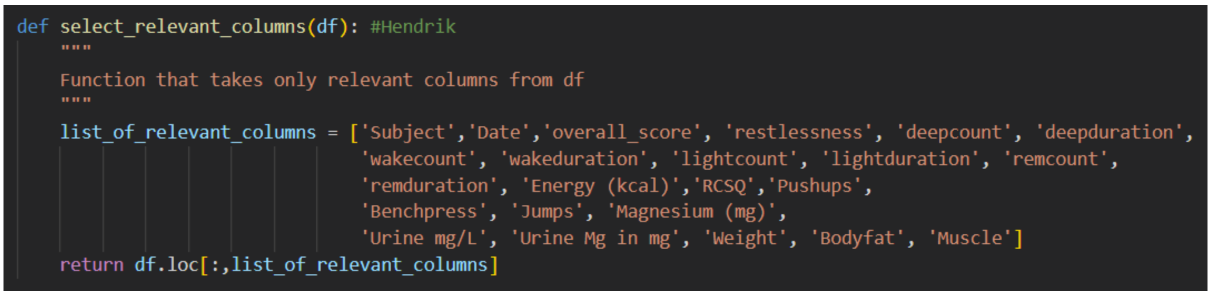
```python
def mergefiles(self):  #Ruben & Hendrik
    """
    Function that merges all dataframes into one dataframe per subject
    """
    sleepdfjson = Subject.create_sleep_df(self)
    cronodf, selfdf = Subject.importfiles(self)

    cronodf.set_index('Date', inplace=True)
    cronodf.index = pd.to_datetime(cronodf.index)

    selfdf.set_index('Date', inplace=True)
    selfdf.index = pd.to_datetime(selfdf.index)

    sleepdfjson.set_index('Date', inplace=True)
    sleepdfjson.index = pd.to_datetime(sleepdfjson.index)

    subject_df = sleepdfjson.join(cronodf, how='outer')
    subject_df = subject_df.join(selfdf, how='outer')

    #Add column with name of subject
    subject_df['Subject'] = self.name.split(' ')[0]
    return subject_df
```

Repeat code

Some of the code is repeated in this notebook. Some example are seen below. At different points the unwanted columns are dropped.

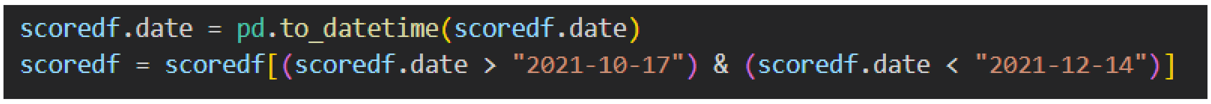


```python
#drop unnecessary columns
scoredf.drop(['sleep_log_entry_id', 'timestamp', 'timestamp', 'resting_heart_rate',
              'composition_score', 'revitalization_score', 'duration_score', 'deep_sleep_in_minutes'],
             inplace = True, axis = 1)
```

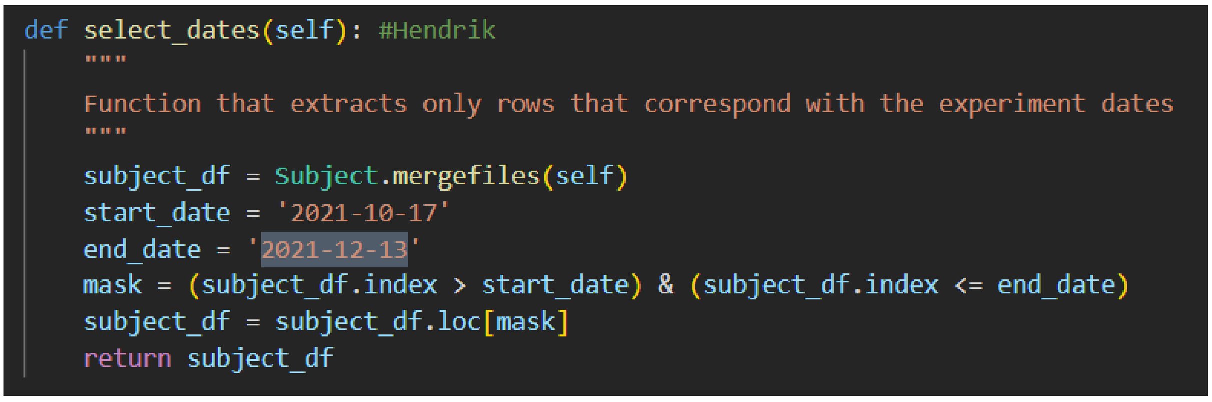


```python
def select_relevant_columns(df): #Hendrik
    """
    Function that takes only relevant columns from df
    """
    list_of_relevant_columns = ['Subject','Date','overall_score', 'restlessness', 'deepcount', 'deepduration',
                                'wakecount', 'wakeduration', 'lightcount', 'lightduration', 'remcount',
                                'remduration', 'Energy (kcal)','RCSQ','Pushups',
                                'Benchpress', 'Jumps', 'Magnesium (mg)',
                                'Urine mg/L', 'Urine Mg in mg', 'Weight', 'Bodyfat', 'Muscle']
    return df.loc[:,list_of_relevant_columns]
```

At different times the start and end date of the experiment are defined. Too much of this repeat code and it is impossible to reuse the code. Also, the dates are hard-coded. A better way would be to have a configuration file where all the different parameters are defined so that one wouldn't have to go through the code to change the dates for example.



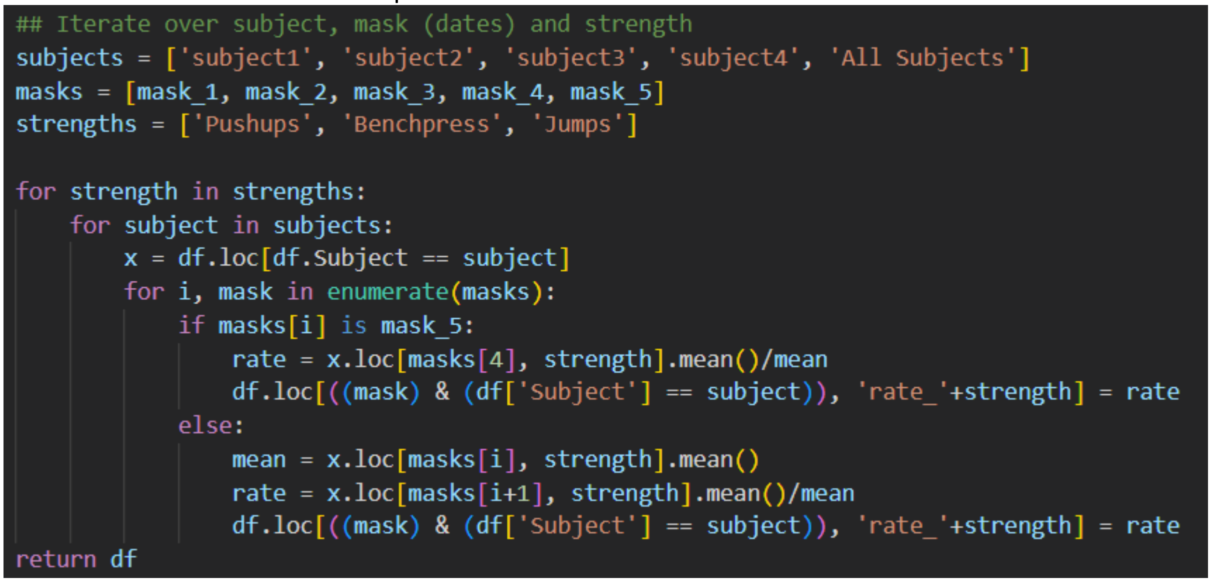
```python
scoredf.date = pd.to_datetime(scoredf.date)
scoredf = scoredf[(scoredf.date > "2021-10-17") & (scoredf.date < "2021-12-14")]
```



```python
def select_dates(self): #Hendrik
    """
    Function that extracts only rows that correspond with the experiment dates
    """
    subject_df = Subject.mergefiles(self)
    start_date = '2021-10-17'
    end_date = '2021-12-13'
    mask = (subject_df.index > start_date) & (subject_df.index <= end_date)
    subject_df = subject_df.loc[mask]
    return subject_df
```

Simplify

The method add_rate_columns (see below) is an example of unnecessary complicated code. The aim is to add the rate of increase/decrease of several columns to the dataframe for later plotting. Part of the code of the function is seen below. It is difficult to understand what is going on and there is a triple for-loop used. Combining a lambda x function and the .apply method from pandas, this piece of code can be written much simpler.



```python
## Iterate over subject, mask (dates) and strength
subjects = ['subject1', 'subject2', 'subject3', 'subject4', 'All Subjects']
masks = [mask_1, mask_2, mask_3, mask_4, mask_5]
strengths = ['Pushups', 'Benchpress', 'Jumps']

for strength in strengths:
    for subject in subjects:
        x = df.loc[df.Subject == subject]
        for i, mask in enumerate(masks):
            if masks[i] is mask_5:
                rate = x.loc[masks[4], strength].mean()/mean
                df.loc[((mask) & (df['Subject'] == subject)), 'rate_'+strength] = rate
            else:
                mean = x.loc[masks[i], strength].mean()
                rate = x.loc[masks[i+1], strength].mean()/mean
                df.loc[((mask) & (df['Subject'] == subject)), 'rate_'+strength] = rate
return df
```

5. Error handling

In the code there are no try and except clauses. The program will ‘crash’ if for example the data files are not in the correct place. A try and except could handle errors like that so that the user knows what exactly the problem is. Also as an example, when the wrong amount of system arguments are specified. Normally, an IndexError would be thrown and the program would terminate. A way to handle this is with a loop where the user is asked to please specify the right amount of system arguments. With a possible button (input) to terminate the program anyway.