Manual of the python application for the Gas Respiration Tests

*Read this manual with the application in front of you and open the corresponding files at each step.  
Your free to use the code for your own projects or make a copy of the application and change it for your own project. Please refer to me in your acknowledgements as: R.G.M. Grobben.  
One critical note: be aware that you must be sure that your calculations are correct. Check often and write tests. It is your responsibility. Date of writing: 2 July 2023.*

The python application is based on just four pillars:

1. The package openpyxl ([link](https://openpyxl.readthedocs.io/en/stable/index.html)) for reading and (over)writing the Excel file.
2. The panda package ([link](https://pandas.pydata.org/)) for using data frames for the calculations.
3. A structured application by using classes and methods rather than just one script. Build with the principle of Objected Oriented Programming [(OOP)](https://realpython.com/python3-object-oriented-programming/)
4. Writing with the pep-8 convention, adding docstrings and type hints. ([link](https://peps.python.org/pep-0008/))

Excel is used as data carrier and Jupiter Notebook is used as interface. You can find that notebook in the directory “Run”. In the notebook are descriptions written to give an understanding of the code-flow.

As can be seen in the application everything is written full out, with less as possible abbreviations. This is improving the understanding of the code and the code-flow. Please do that too.

**The Excel explained**If working with python, standard format should be used. This means that every sheet has the same structure and naming conventions. Only the data can differ. As an example, the column names and the names of the constants must be the same throughout Excel. Otherwise, the code will break.   
  
Ideally, one should work with Excels that contain only a table (with or without figures does not matter). This facilitates reading and avoids errors. The Excels below can be found in the directory “excel\_sheets”.

As an example:

Afbeelding met tekst, schermopname, nummer, software

Automatisch gegenereerde beschrijving*A Excel with only a table (multiple sheets with each a table is not a problem but all in the same format):*

Afbeelding met tekst, schermopname, nummer, software

Automatisch gegenereerde beschrijving*A Excel with the constants:*

The request was to work with an Excel from which only the data is extracted and copied to the same file again. With this request, the constants were placed above the table.

There are conventions:

* Place all constants used for calculations below each other.
* Do not place single phrases or annotations in the cells at the bottom of the table. Keep them empty, otherwise errors will occur. Pictures can be placed at the bottom of the table.

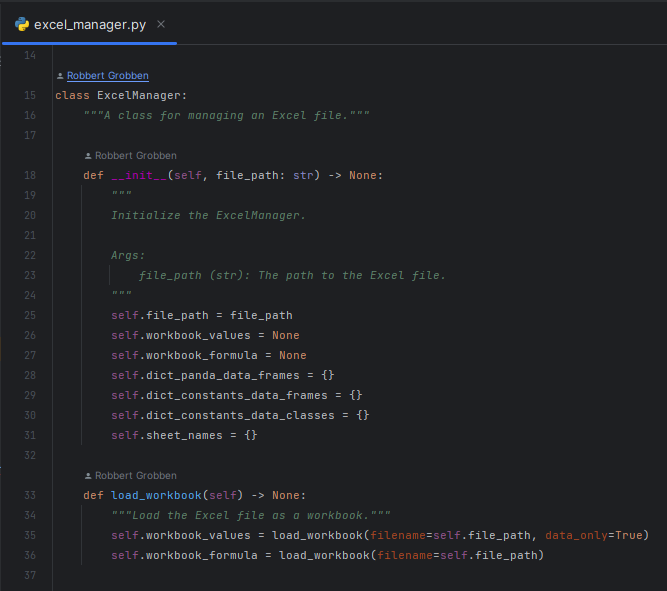
The Excel:

Afbeelding met tekst, schermopname, nummer, Parallel

Automatisch gegenereerde beschrijving

**The structure of the application**  
As said before the interface of the application is a Jupiter Notebook. The code that is stored in the different classes and files are activated in the Notebook. As a result, the notebook remains compact and well-organised. By using classes and Object Oriented Programming (OOP) the code is easily customisable, testable and scalable. It is also easier to reuse the code in other projects. If you do not know what classes are than you must read the link of OOP on the first page. Below, the different files with classes are briefly explained.

**General structure Classes**Every class and method needs to be written with docstrings and type hints. Docstrings are the green-texts which explains what the class or method does. Type hints are written after a variable or method that gives python information about what to expect as input or as output. You don't have to do this but this makes the code more robust and readable. More about docstrings and type hints: ([link](https://towardsdatascience.com/python-type-hints-docstrings-7ec7f6d3416b))



Docstrings

Type hint: the method is expecting to have a specific output. In this case nothing (None).

Type hint: variable expects a string

Input arguments class

General stored objects inside class

First method of class

Using the method decorator *“@staticmethod*” inside the class the method is stored in a class but it can be callable without making a instance of the class. It is basically a way of storing methods in a structured way. More about decorators use this [link](https://realpython.com/instance-class-and-static-methods-demystified/). Example:

Afbeelding met tekst, schermopname, software

Automatisch gegenereerde beschrijving

**File: excel\_manager.py:**In this file the class “ExcelManager” is located. This class was created to manage the Excel. Via this class, you easily load the Excel and the Excel is also being "stored" in the class.

**File: data\_classes.py:**In this file all the data classes are stored. As the name explained, data classes are specific objects in python where you can easily store data. Please read this [link](https://realpython.com/python-data-classes/) for more information about data classes.

**File: data\_frame\_processor.py**  
In this file the class “DataFrameProcessor” is stored. This class is for all kinds of methods generally related to panda data frames. Such as adding a column of days, repositioning a column or filling nan values. You can make a trade-off between 'storing' the code in a method or using it loosely in your script. If you use it often, it is advisable to store it in the class.

**File: data\_frame\_calculations\_standard\_fro\_gas\_respiration.py** **& data\_frame\_calcualtions.py**The calculation classes for the data frames are stored in two separated files. The file *data\_frame\_calculations\_standard\_fro\_gas\_respiration.py,* contains the standard calculations of the gas respiration tests for the data frames. Such as correcting the values of the gas composition or converting the gas composition to moles. In the file, *data\_frame\_calcualtions.py*, the other data frame calculation methods are stored. Feel free to use other files for your own classes. A good overview via a structured application is the main goal.

**File: run\_data\_frame\_calculations.py**In this file is the class used to combine all the separate data frame calculations together. In the other files the general methods are written. But in this class one general method can be used multiple times but with there own context. Another conveniency is that we can met the “single point of truth” principle which makes the code more robust. For example, if we need for multiple data frame calculations the name of column A, we only need to write just one time the name of this column. When you must do that multiple times than there is a higher change for errors. You need to see this file by yourself for a full picture and understanding.

**Test files and Unit Tests**By making your application even more robust, it is important to use unit testing. These tests can be found in the directory "test" and are designed to test your code for errors. This is important for two things: (1) after modifying existing code and (2) while writing new code. It also forces you to programme more with the principle of OOP. This is especially important for the methods you write for calculations. More information on testing: [testing](https://realpython.com/python-testing/).

**How do I add a calculation and write a unit test?**An example describes the process from the creation of a method to its implementation. First, you want to know where to store the function. First, you need to get an idea of where you want to write the method. Can it join an already existing class (does it make sense that this method belongs to this class?) or should I create a new one? Also see if it can be written into an existing file or if you'll write a new one. In this example, we create a new class in a new file. We keep the data frame calculation very simple. The calculation is as follows: column C = column B \* (column A / constant).

1. Afbeelding met tekst, schermopname, software, Multimediasoftware

   Automatisch gegenereerde beschrijvingCreate a file, create the class:

* First you create a new file.
* Create the new class
* Write the “def \_\_init\_\_(self)”

Here write you the input argument of the class. In this case it is the data\_frame.

1. Write the function:

- Note that you make the function generic. So the function has as input argument the names of the columns and the new column. All with type hint string. The constant is changeble but has as default the number 12 and has the type hint float.

Afbeelding met tekst, schermopname

Automatisch gegenereerde beschrijving

1. Afbeelding met tekst, schermopname, software, Lettertype

   Automatisch gegenereerde beschrijvingIn this case it is more convenient to make the class static. This means that the method can be used without the whole implementation of the class. The end result of the static method is below. Do not forget the docstring! In this case it is clearly what the function does. But The constant is specific. You can choose to only specify the constant and the context of usage of the function or to specify everything.

1. Go to the place in the Notebook where you want to use the function and call the function.

* Call the function to first mention the class name and then the name of the method. Fill in the input arguments and activate the cell.

Afbeelding met tekst, schermopname, software, Multimediasoftware

Automatisch gegenereerde beschrijving

1. Make the unit test for the function. Sometimes (or often) it is better to write first the test and then the function. Via this way of programming, you are forced to think more carefully about what kind of function you want. In addition, you can program much faster and more focused because you can check the outcome faster. With the unit test you write a dummy with a known outcome and with this dummy the function can be tested. Write the test in a different file in the test directory. With the green triangles you can activate the tests. Note that you can use an extra class to combine multiple calculations if needed (see run\_data\_frame\_calculations.py as example). Example:

Afbeelding met tekst, schermopname, software, Multimediasoftware

Automatisch gegenereerde beschrijving

**Code for validating input data**Besides that code is written to process the data, code was also written to validate the input data. This code is stored in the "validation\_input\_data" directory and the code is activated again in a Jupiter Notebook. This notebook can be found in the "Run" directory. This code is checking if the cells are correctly filled, For example, that a cell has a float or a string or is filled with a condition. In this project, it was a condition that the sample had to be weighed at each flush.

**Code for statistics, like outliers**In the directory “statistics” code is written for statics applications. This is not much but it is still an valuable application.

**Improvements for the application**

1. Write tests for all functions and methods
   1. All the calculations has already tests.
   2. Write tests for all the other functions and classes.
2. The code for the validation of the input data is working. Still there is a need of improvements in the writing style and the interface.
3. All the functions in the class below are good and the code is working. But the data\_frame as input argument (init) for the class has to be removed. And for every method in this class there must be an extra input argument: “data\_frame: pd.DataFrame”.

Afbeelding met tekst, schermopname, software

Automatisch gegenereerde beschrijving

Afbeelding met tekst, schermopname

Automatisch gegenereerde beschrijving

**Further building:**

1. Write code to automate the generation of Excel figures.
2. Write code for statistical purpose. Search on the internet for specific build statistical packages.
3. Do what you can't resist and make it public. 😊