

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. Grobбен.

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](#)) for reading and (over)writing the Excel file.
2. The `panda` package ([link](#)) 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](#))
4. Writing with the pep-8 convention, adding docstrings and type hints. ([link](#))

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 Excls that contain only a table (with or without figures does not matter). This facilitates reading and avoids errors. The Excls below can be found in the directory “excel sheets”.

As an example:

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

[illegible]

A Excel with the constants:

	A	B	C	D	E	F
1	Name	Value	Unit			
2	Rgas	8314.5	Lpa/Kmol			
3	expTemp	293.15	K			
4	volume_headspace	0.961	l			
5	MM_C	12	g/mol			
6	water_volume	0.098	l			
7	dry_mass_sample	153.6077853	g			
8	henryeff_20	0.00523				
9						

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:

Experiment description

Klembord

Lettertype

Uitlijning

Getal

ALS

:

✕

✓

fx

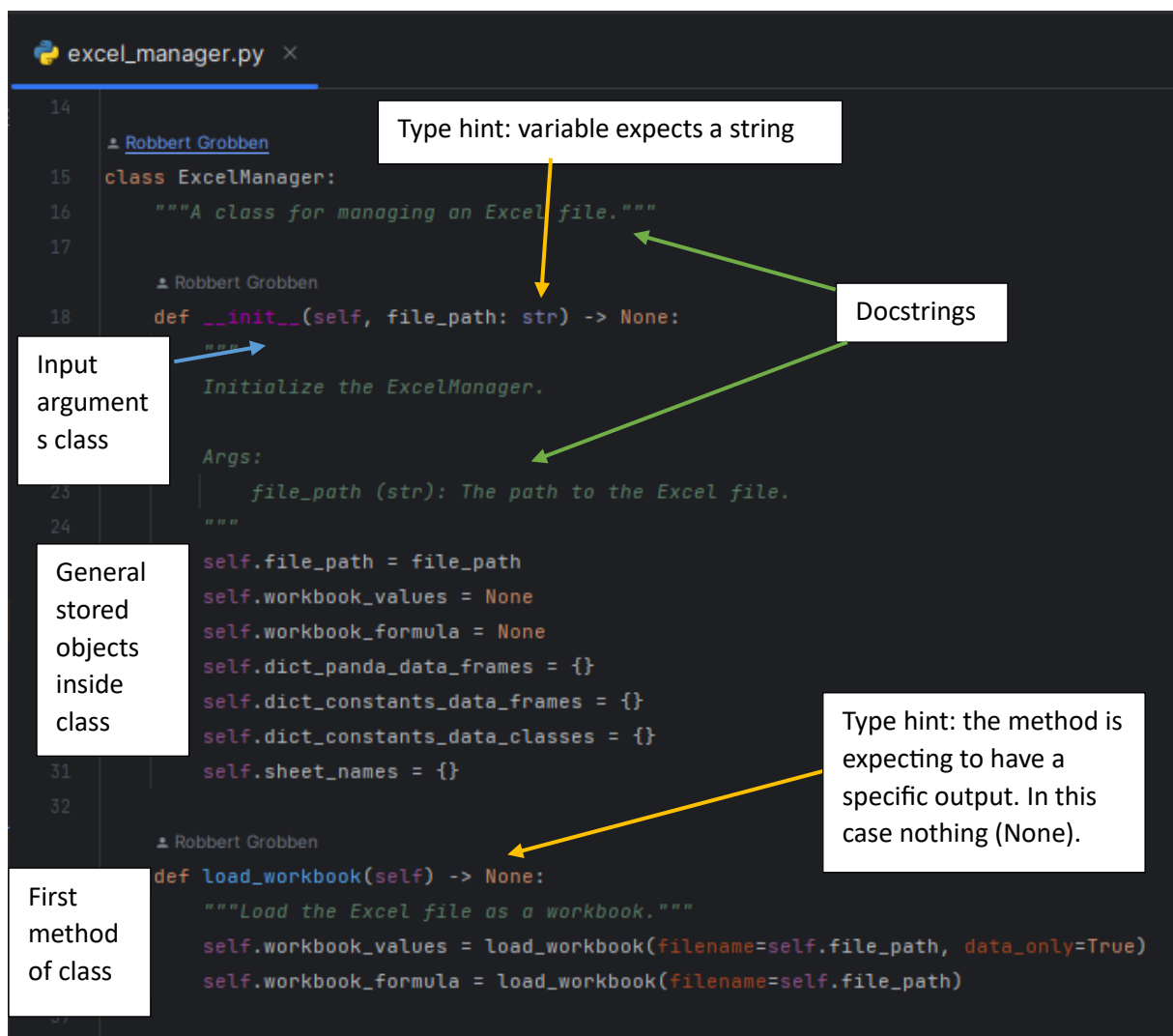
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	constant	Rgas		8314.5	Lpa/Kmol												
2	constant	expTemp		293.15	K												
3		Vol headspace		0.961	l												
4	constant	MM _C		12	g/mol												
5		Water volume		0.098	l												
6		Dry mass sample		153.6	g												
7	constant	Henryell 20 °C		#####													

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](#))



Using the method decorator “@staticmethod” inside the class the method is stored in a class but it can be callable without making an instance of the class. It is basically a way of storing methods in a structured way. More about decorators use this [link](#). Example:

```

5 usages  ⤴ Robert Grobber
class DataFrameProcessor:
    """
    A class for processing pandas DataFrames standard calculations.
    """

    ⤴ Robert Grobber
    @staticmethod
    def fill_nan_values(data_frame: pd.DataFrame, column_name: str, value: Any) -> None:
        """
        Fill the NaN values in a column with a specified value.

        :param data_frame:
        :param column_name: The name of the column to fill NaN values.
        :param value: The value to fill NaN values.
        """
        data_frame[column_name] = data_frame[column_name].fillna(value)

```

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](#) 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_calculations.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_calculations.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.

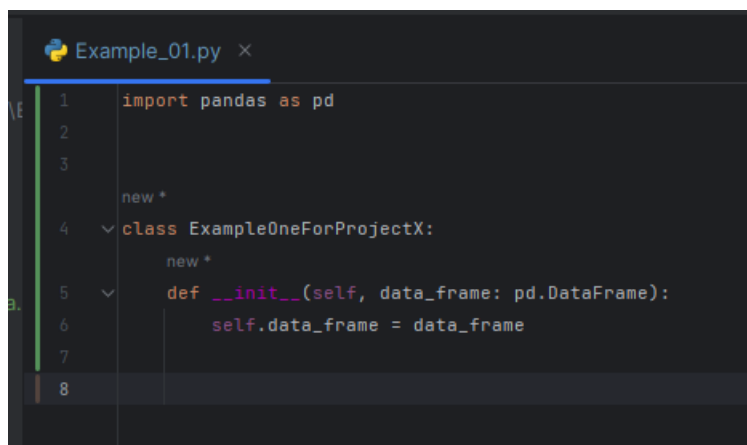
Test files and Unit Tests

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](#).

How do I add a calculation and write a unit test?

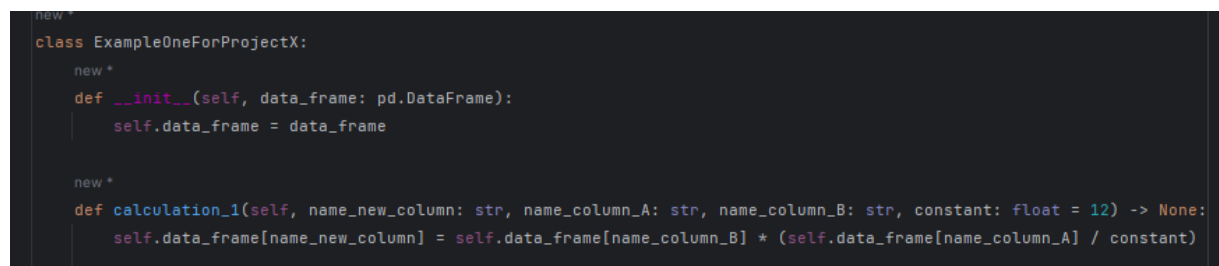
An example describes the process from the creation of a method to its implementation. First, you want to know where to store the function. First, you need to get an idea of where you want to write the method. Can it join an already existing class (does it make sense that this method belongs to this class?) 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. Create 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.



```
Example_01.py x
1 import pandas as pd
2
3
4 new *
5 class ExampleOneForProjectX:
6     new *
7     def __init__(self, data_frame: pd.DataFrame):
8         self.data_frame = 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 changeable but has as default the number 12 and has the type hint float.



```
new *
class ExampleOneForProjectX:
    new *
    def __init__(self, data_frame: pd.DataFrame):
        self.data_frame = data_frame

    new *
    def calculation_1(self, name_new_column: str, name_column_A: str, name_column_B: str, constant: float = 12) -> None:
        self.data_frame[name_new_column] = self.data_frame[name_column_B] * (self.data_frame[name_column_A] / constant)
```

2. In 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.

```
new *
class ExampleOneForProjectX:
    new *
    @staticmethod
    def calculation_1(data_frame: pd.DataFrame, name_new_column: str,
                     name_column_A: str, name_column_B: str, constant: float = 12) -> None:
        """
        Explain for what the function is used.

        :param data_frame: The data frame for the calculation.
        :param name_new_column: The name of the new column
        :param name_column_A: The column for ...
        :param name_column_B: The column for..
        :param constant: The constant is .... and is used for... etc.
        """
        data_frame[name_new_column] = data_frame[name_column_B] * (data_frame[name_column_A] / constant)
```

3. 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.

```
In 2 1 from Example_01 import ExampleOneForProjectX
2 import pandas as pd
3
4 # Create a sample DataFrame
5 data = {'A': [10, 20, 30, 40, 50],
6         'B': [1, 2, 3, 4, 5]}
7 df = pd.DataFrame(data)
8
9 # Call the calculation_1 method
10 ExampleOneForProjectX.calculation_1(data_frame=df, name_new_column='new_column', name_column_A='A', name_column_B='B', constant=5)
11
12 # Print the modified DataFrame
13 print(df)
14
15
16
```

Executed at 2023/07/02 20:30:24 in 18ms

	A	B	new_column
0	10	1	2.0
1	20	2	8.0
2	30	3	18.0
3	40	4	32.0
4	50	5	50.0

4. 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:

```
1 import unittest
2 import pandas as pd
3
4 from Example_01 import ExampleOneForProjectX
5
6
7 new *
8 class TestExampleOneForProjectX(unittest.TestCase):
9     new *
10     def test_calculation_1(self):
11         # Create a sample DataFrame
12         data = {'A': [10, 20, 30, 40, 50],
13                 'B': [1, 2, 3, 4, 5]}
14         df = pd.DataFrame(data)
15
16         # Call the calculation_1 method
17         ExampleOneForProjectX.calculation_1(df, 'new_column', 'A', 'B', constant=5)
18
19         # Verify the correctness of the calculation
20         expected_result = [2.0, 8.0, 18.0, 32.0, 50.0]
21         self.assertEqual(list(df['new_column']), expected_result)
```

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
 - a. All the calculations has already tests.
 - b. 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".

```
8
9
10 class RunDataFrameCalculationsForOneDataFrame:
11     """Combine all the calculations to do it for one data frame."""
12     data_frame: object
13
14     def __init__(self, data_frame: pd.DataFrame):
15         self.data_frame = data_frame
16
17         self.get_column_name_date: str = "Date"
18         self.get_name_column_time: str = "Time"
19         self.get_name_column_ch4: str = "CH4 [%]"
20         self.get_name_column_co2: str = "CO2 [%]"
21         self.get_name_column_o2: str = "O2 [%]"
22         self.get_name_column_n2: str = "N2 [%]"
23         self.get_name_column_flush: str = "Flush (1=yes; 0=no)"
24         self.get_column_name_pressure_before: str = "P sample before gc [hPa]"
25         self.get_column_name_pressure_after: str = "P sample after gc [hPa]"
26
27         self.create_name_column_summation_correction: str = "Sum-corr [%]"
```



```

66         self.create_name_column_ratio_O2_CO2: str = "Ratio O2/CO2"
67
68         ⚡ Robbert Grobber
69     def run_data_frame_processor_calculations(self, dayfirst: bool = False):
70         DataFrameProcessor.add_day_column(data_frame=self.data_frame, date_column_name=self.get_column_name_date,
71                                         time_column_name=self.get_name_column_time, dayfirst=dayfirst)
72
73         ⚡ Robbert Grobber
74     def run_gas_composition_calculations(self,
75                                         set_values_gas_composition_first_row: bool = False,
76                                         ch4: float = 0, co2: float = 0, o2: float = 0, n2: float = 0,
77                                         index: int = 0,
78                                         ):
79         # set the values of the first row for the gas composition.
80         if set_values_gas_composition_first_row:
81             GasComposition.set_gas_composition(data_frame=self.data_frame,
82                                               ch4=ch4, co2=co2, o2=o2, n2=n2, index=index,
83                                               name_column_ch4=self.get_name_column_ch4,
84                                               name_column_co2=self.get_name_column_co2,
85                                               name_column_o2=self.get_name_column_o2,
86                                               name_column_n2=self.get_name_column_n2,
87                                               )

```

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. 😊