Documentation for the software tool for analysing optical emission spectra

# Organization of Python code in the files

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""Summary line.

Extended description of function.

"""

* **Python interpreter:** The first line is for \*nix users. It will choose the Python interpreter in the user path, so will automatically choose the user preferred interpreter.
* **File encoding:** The second one is the file encoding. Nowadays every file must have a encoding associated. UTF-8 will work everywhere. Just legacy projects would use other encoding.
* **Documentation:** And a very simple documentation. It can fill multiple lines.
* See also: <https://www.python.org/dev/peps/pep-0263/>
* If the file just implements a class, the documentation goes into the class documentation.

(Section above adapted from: https://stackoverflow.com/questions/1523427/what-is-the-common-header-format-of-python-files)

# Style Guide

The default style guide that is followed is the PEP 8. The full description can be found here: <https://pep8.org/>. Because it is a general style guide, there are complements for specific cases in the software:

Classes:

* Each class is defined in a separate file.
* The name of the class is the same as the name of the file (except the ending obviously)
* Methods of the class are named in…
  + … camelCase, if the method has to be recognized by the Qt-backend, e.g. events like dragEnterEvent
  + … snake\_case, if the method is independent of the Qt-backend, like file\_open

User Interface (UI)

* Elements are named in camelCase and follow the convention “type”+”name”, e.g. btnClear where “btn” is the type and “Clear” is the name of the button.
* Groups of elements are named in PascalCase, e.g. BtnParameters are several parameters grouped.
  + Might be important for loop-implementation
* Overview of abbreviations for naming elements
  + btn  button
  + cb  checkbox
  + list  list
  + tin  text input
  + tout  text output
  + act  action
  + menu  menu
  + fout  file output
  + dd  dropdown
  + bar  progress bar
  + lbl  label
  + mpl  plot from matplotlibary
  + layout  horizontal/vertical layout

# Concepts

## Separating UI and logic

The idea is to seperate the UI and the logic as much as possible to change the behaviour of the UI in a single file (maybe in another config/enum file) and use the other files for the logic, that one has to care as less as possible about the UI when fixing classes and calculation, etc. Therefore, the UI-class has to provide a proper interface to connect to slots and connect functions to signals. So, the properties of ui elements should be accesible via the interface.

Involved modules:

* modules/AnalysisWindow.py
* ui/UIMain.py
  + allows signal-slot connections according to a mapping using the enum *UI\_RESULTS*
  + provides connections to signals using interface methods
* modules/DataHandler.py
  + Maps signals and the enum *UI\_RESULTS* to call a provided function to connect signals to slots. Function have to be provided by UIMain.
* custom\_types/UI\_RESULTS.py
  + Configurational datatype, that all modules uses the same keywords.

# AnalysisWindow.py

This module is the “main” module and works as an organizer. It initialize the instances of the objects, the UI and the DataHandler. And, provides arguments for the corresponding methods of these instances.

# UIMain.py

This module contains all UI elements and set them up to build the UI. Furthermore, it provides methods for connecting methods/functions to signals. For example *connect\_openFile(self, fun)* is a interface that AnalysisWindow uses to connect the signal (qt naming of an event) with the given function *fun.* Therefore, the module UIMain needs no information within the module about any function of other modules, where the signals are connected to.

It also provides opportunities to connect a QtSignal to a given slot of an UI element to update the e.g. text, whenever needed. The function *ConnectSlotResults(self, signals:dict)* uses dictionary and the enum *UI\_RESULTS* to provide a interface. The signals must have a key which is also apparent in *UI\_RESULTS.*

# DataHandler.py

This module has the signals which are allowed to alter the displayed text of UI elements. These signals are mapped as values to the corresponding keys of the enum *UI\_RESULTS,* such that the enum is the configurational part of the interface between UI elements and signals.

# See also

https://www.riverbankcomputing.com/static/Docs/PyQt5/signals\_slots.html

# Testing

* Using the module: unittest to run unittests in python.
* See also: <https://www.youtube.com/watch?v=1Lfv5tUGsn8>

Naming convention:

* Files: test\_name.py
  + test\_ is necessary to recognize the module as a test module.
  + Can be run in terminal or in other test suites.
* Methods: test\_method(self, args)
  + test\_ is necessary to recognize the module as a test method.

How to run tests:

* Open Anaconda Prompt
* Navigate to the root directory
* Type in: python -m unittest [name]
  + m  instruct the python module to run as a script
  + [name] (optional)  run a specific test, if omitted it will run all tests in that directory
  + additional option -v: verbose documentation of the current tests.
* Run all tests in a directory: python -m unittest discover modules -v
* Run a specific Testcase e.g.: python -m unittest modules.test\_BatchAnalysis.TestBatchAnalysis.test\_set\_filename using the module.class.method to specifiy the test

## Test environment

The file runner.py in the root directory is designed to run all included test, when executed. That means, the script is flexible to add or remove tests from each sub-directory. But additional test modules and tests have to be inserted manually. Further documentation about the procedure is given in the module.

### QtTest

Implements different methods to ease the tests for ui elements.

## Additional information

### Run a test in Spyder console

To run a test in the Spyder console, one need to run the unittest.main function.

import unittest

If \_\_name\_\_ = ‘\_\_main\_\_‘:

unittest.main()

The code above will run the unittest.main() if the module is executed as main (e.g. in the Spyder console). Therefore, it has the same effect like running the unittest script and giving the testmodule as argument via prompt. Hence, it is executable in an console.

### PICT

Pairwise Independent Combinatorial Tool. Can be used to reduce the amount of tests if a lot of combinatorics must be tested. Uses the minimum amount of combinations to test the function with still a high test coverage.

# Python Syntax

Ternary operator:

* <https://stackoverflow.com/questions/394809/does-python-have-a-ternary-conditional-operator>
* x = a if True else b

Chaining of comparator:

* 5 < x < 15

Type Hinting:

* [https://realpython.com/documenting-python-code/#why-documenting-your-code-is-so-important](https://realpython.com/documenting-python-code/" \l "why-documenting-your-code-is-so-important)
* def hello\_name(name: str) -> str:

return(f"Hello {name}")

* + Input variable name is of type string (name: str)
  + Return variable is of type string (-> str)

@property:

* [https://www.youtube.com/watch?v=jCzT9XFZ5bw](https://www.youtube.com/watch?v=jCzT9XFZ5bw&t=328s)
* Using Getter, Setter, Deleter methods. Access methods as attributes to enable further evaluation/checking/handling.
* Useful for linked attributes, ui, …

@classmethod:

* <https://www.youtube.com/watch?v=rq8cL2XMM5M>
* Takes the class (cls) as an argument other than a static method.
* Useful for setting attributes across all instances of objects.
* Useful for alternative constructors.

@dataclass:

* <https://www.youtube.com/watch?v=Udz4jjd46ho>&t=1287s
* Mutable, flexible (or not), easy to handle (like a class)
* Provides a boilerplate and extra methods by default.
* Useful if you just need a simple object with some data in it.
* <https://python-forum.io/Thread-A-look-at-dataclass>
* <https://docs.python.org/3/library/dataclasses.html>
* https://realpython.com/python-data-classes/#alternatives-to-data-classes

# General advises

## Dict

Dict.get(‘key’, None):

* Gets the value of the key like Dict[‘key’].
* If the key is not in the dict, it will return the provided default value (here: None).
* So it never raises a KeyError.
* See also: Defensive programming.

## Classes

\_\_validate():

* General a method to validate the data and values inside the instance of the class.
* Useful to call it at the end of the constructor or whenever one do a critical change.

\_\_post\_init\_\_():

* Method to call at the end of the init for everything that has to be done after the normal constructor.
* See also the usage in @dataclass.