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/>
* And: [https://www.python.org/dev/peps/pep-0008/#programming-recommendations](https://www.python.org/dev/peps/pep-0008/" \l "programming-recommendations)
* 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)

# General advices

* How to import a git lib into python: https://stackoverflow.com/questions/19943022/import-a-python-library-from-github

# 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

## Configuration

The basic idea of this concept is, that one can import a ConfigLoader-class, that provides a uniform access to a configuration. Therefore in each module the condiguration can be easily loaded.

The implementation is based on two files, the module ConfigLoader, which provides a class, and config.yml, which contains the configuration. The class has basically two tasks: First, interaction with the config file (load and save configurations), and second, provides an interface, that other modules can easily access the needed configuration properties. Accordingly to the first task, the class has two methods (load and save) implemented which both uses the path-property. The second task includes properties for each section in the configuration and the corresponding getter methods. But this interface only provides shortcuts for sections in the config.yml-file and not for other .yml-files such as \*-fitting.yml files. This requires direct access to the dict via the config-attribute and may look like ConfigLoader.config.[“NAME”].

## Change styles of plots

The basic configuration like color and linestyle can be changed in the config.yml-file in the corresponding part “PLOT”. To add a new style instead of the default setting, one can change a few lines in the module Spectrum.py. In the method “get\_markup” one has to add the “linestyle”-key and the corresponding value from the configuration (see also “baseline” as template).

## 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.

# Signal-Slot-interface

* Make sense for inputs. Interaction with the ui triggers events and send signals. Easy to implement asynchrounus handling.
* Most times just one ui element as output, but also some more like in the result section of UIMain.py
* If the ui is changed, there will be no error raised, due to the signal is lost somewhere but not catched.

# Setter-Getter-methods

* Easy to implement.
* Outputs are set “manually”, method call whenever you want to update the value.
* Get paramters if the analysis is triggered by an event.
* Feedback like errors can be raised.

# Conclusion

The most times it makes sense to set up a signal-slot-interface for buttons and triggers, etc… In other cases it also make sense to check out the current values of the input elements, when an analysis is triggered. And regarding the outputs, often the signal-slot-interface may be used.

### Implementation for *input* elemtents

Using the element *tinCentralWavelength*:

* @property
  + The textChanged-signal can be used to update the current value of the property via the setter method.
  + Easy validation of input.
  + This option is indiviual, flexible but also a lot of work for multiple elements and rather hard to maintain if there is a general switch in programming style/concept
* @dataclass
  + Using one class for a group of multiple input elements and providing a object to retrieve the current values.
  + Easy validation of input.
  + Uncertainty how to connect signals to these properties.
* get\_central\_wavelength-method of the UI class
  + Easy to implement and verify and maintain.
  + Indiviual and a lot of writing for each element and each data access.
  + Don’t use try-except here, because it makes sense to raise an error if the element is removed.
  + Maybe use a log info to get the retrieved values in the log.

### Implementation for *output* elemtents

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>

## Analysis

This part enlightens the organization of the analysis. Basically the analysis gets some raw data, processes these data, and calculates some values out of it. In this application there a in general two different ways to analyse files.

1. Analysis of one file, which will be displayed as a plot and the characteristic values are displayed in the results section of the ui.
2. Batch analysis of multiple files. The characteristic values are exported to another file. Diplaying the plots are optional.

The flow are shown in the follwing illustration.

AnalysisWindow:

RawData AND Config(including wavelength, grating, fitting)

→ Calculate processded data, baseline, peak height/area/position

→ Update the spectra, the plots, and the displayed results

BatchAnalysis:

(For each file of the filelist)

RawData AND Config(including wavelength, grating, fitting)

→ Calculate processded data, baseline, peak height/area/position, characteristic value

→ Store the data to prepare the export.

Optional: Display the analyzing spectrum in a plot or display the peak area/characteristic value against the time in a separate plot.

The config basic configuration is retrieved by the method (UIMain).get\_basic\_setting()

# 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.
* https://stackoverflow.com/questions/11041405/why-dict-getkey-instead-of-dictkey

## Classes

Structure of methods:

* Define variables at the beginning of each method.
  + Changes can be made once, so maintenance becomes easier.
  + Like: spec = spectrum, where spectrum is the parameter of the method.

\_\_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.

## Decorator

A decorator function is executed whenever the method is called. So the original method is altered in such a way that it for example logs something or print something to the console.

For a explanation see: <https://www.youtube.com/watch?v=FsAPt_9Bf3U>

Especially regarding loggong: <https://www.youtube.com/watch?v=FsAPt_9Bf3U&t=1170s>

Especially regarding performance: <https://www.youtube.com/watch?v=FsAPt_9Bf3U&t=1320s>

Also have a look into @wraps from functools to maintain the original name of your funtion.

Also be aware of the arguments \*args and \*\*kwargs.

## yield

This keyword may improve the performance if one initialize a empty list, doing some work, appending an element after that work and in the end return the list. The yield keyword “returns” one element at a time, so one do not need to care about large memory polution.

# Glossary