Nanosurf Python Library Overview

Introduction to the usage of the Nanosurf Python Library and app development

Content

How to create nice python applications

Overview of the Nanosurf library content

Getting started with the app_template

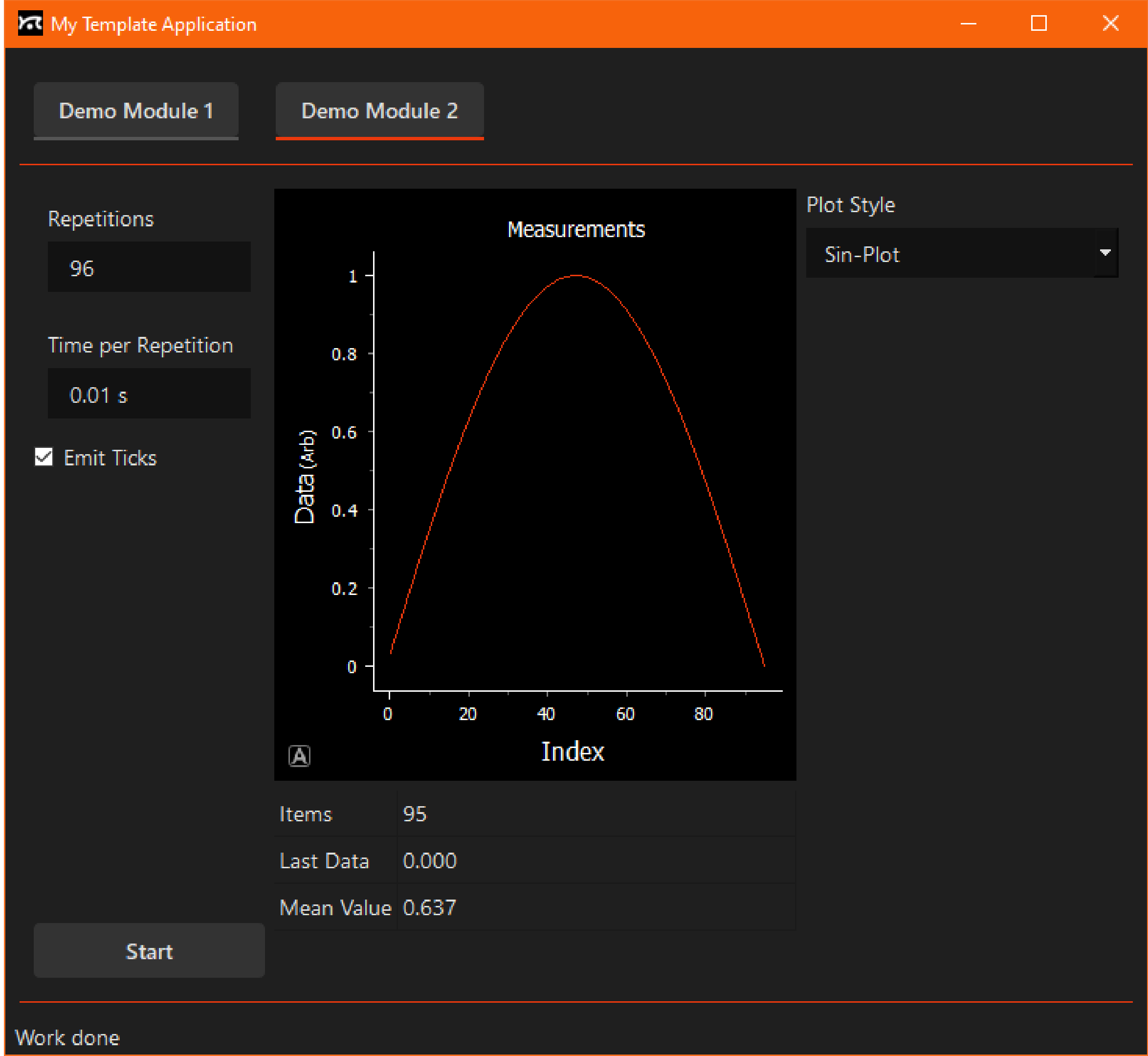
How to create nice applications

Creating nice Python apps

Use libraries.

Separate gui from logic.

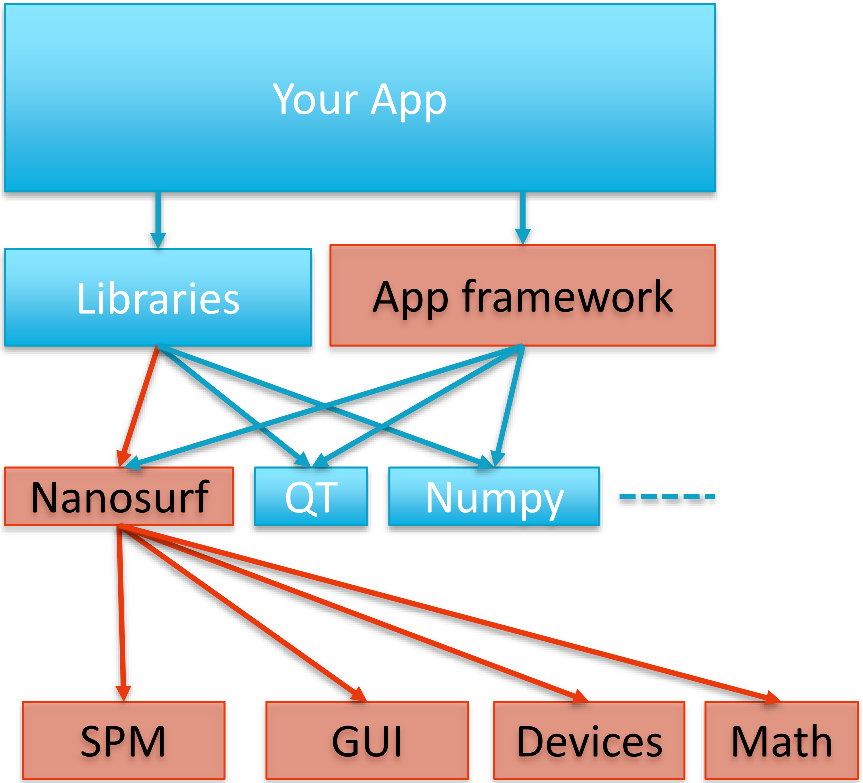
Follow code style guides.



Structure of an app

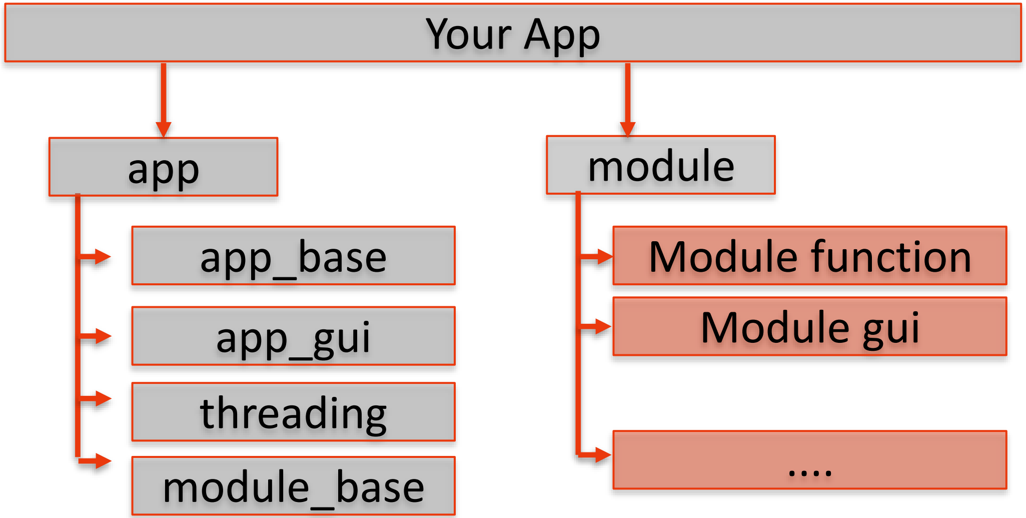
Developer of apps are using frameworks and libraries to reuse common and repetitive tasks.

Nanosurf provides an app framework and a library. They are marked in orange in the chart.



App Framework

The Nanosurf app framework provided in the app_template, give you a jump start to create applications without bothering of boring and nasty details.



The app framework cares about proper startup/shutdown, initialize logging, handle loading/storing configuration settings, handle debugger support for task, and many more...

It prepares the window for Studio look and feel. Add a status and menu bar (if needed).

And other details ...

You focus to your problem to solve in the ***module.py*** and design a nice looking gui in the ***gui.py***.

If needed you create background task for long running activities.

The result is an app with a good gui <-> function separation, readability and easy to enhance

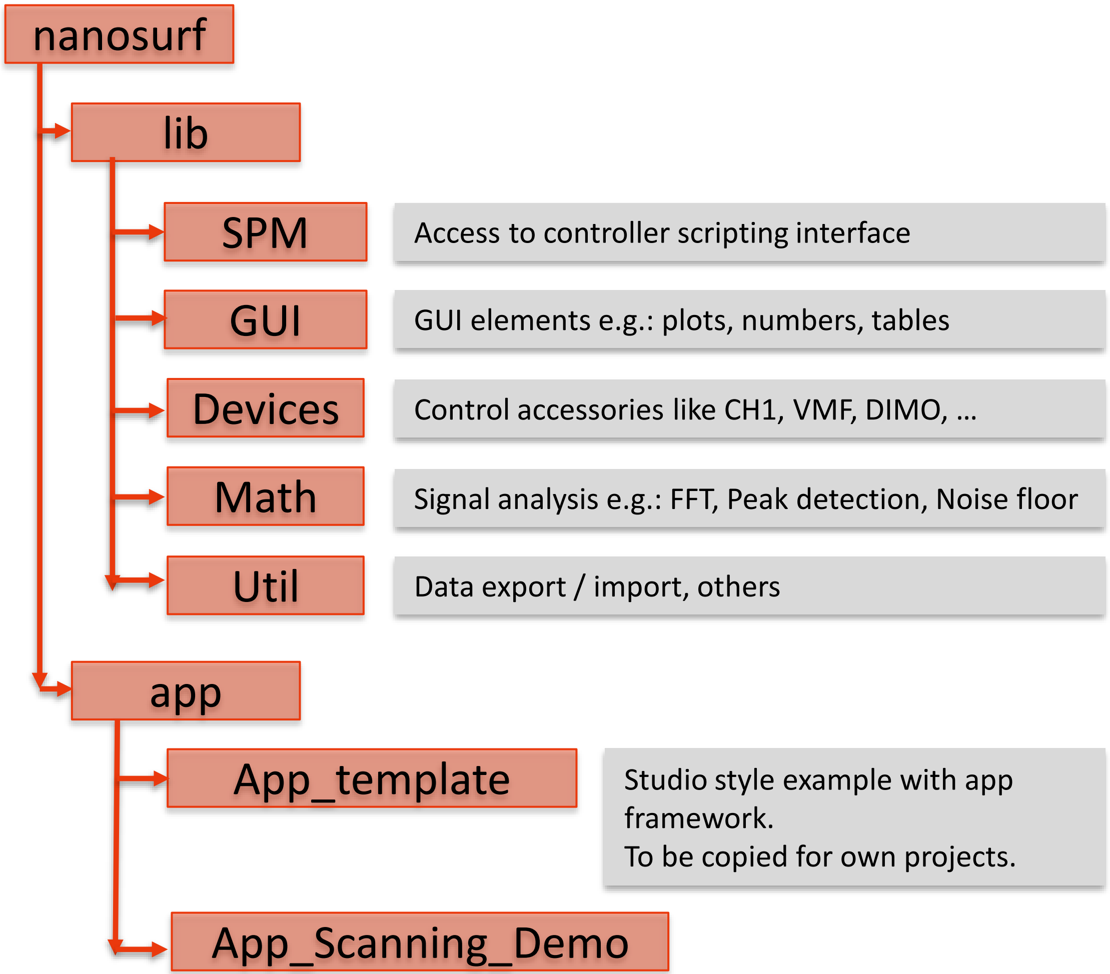
Overview of the Nanosurf library content

Nanosurf library

The library provides classes sorted into different topics.

This library is free and available on PyPI (https://pypi.org/).

Get it with “pip install nanosurf” entered in a command shell.



nanosurf.lib.spm

Base class to connect to all Nanosurf controller software

Give access to full scripting as described in "Script Programmers Manual.pdf"

Give access to powerful lowlevel controller interface.

Provide classes for individual scan heads, direct motor control and more.

```
import nanosurf

# connect with running controller software
spm = nanosurf.SPM()
spm.application.Visible = True

# call actions
spec = spm.application.Spec
spec.Repetition = 4
spec.Start()
while spec.IsMeasuring:
    pass
```

```
import nanosurf

spm = nanosurf.SPM()
if spm.is_lowlevel_scripting_enabled():
    ll = spm.lowlevel
    lu_dac = ll.AnalogHiResOut(ll.AnalogHiResOut.Instance.USER1)

    lu_dac.input.value = lu_dac.InputChannels.InTipCurrent
    current_val = lu_dac.current_output_value.value
    range_max = lu_dac.current_output_value.value_max
    value_unit = lu_dac.current_output_value.unit
    print(f"User DAC: {current_val:0.3g}{value_unit}, max={range_max}")
else:
    print("sorry lowlevel scripting is not available")
del spm
```

```
import time
import nanosurf
from nanosurf.lib.spm.scanhead import drive_afm

spm = nanosurf.SPM()
afm_head = drive_afm.DriveAFMScanhead(spm)
afm_head.start_photo_detector_auto_adjustment()
while afm_head.is_photo_detector_auto_adjustment_running():
    time.sleep(0.5)
print("Detector adjusted")
```

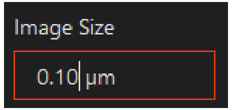
nanosurf.lib.gui

Based on *Python Qt library PySide2* enhanced gui elements are provided. Number edit with units in *NSFSciEdit.*

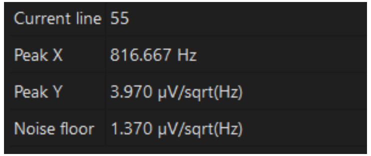
Based on *pyqtgraph*, simplified plot and colormaps are provided. They support mouse cursor, markers and layers.

In nanosurf.app you find a demo app: “app_demo_scanning_and_lib_usage” which shows some use cases

nsf_sci_edit.NSFSciEdit

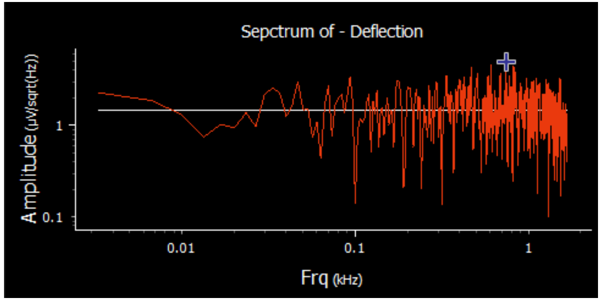


nsf_tables.NSFNameValueTable



Current line	55
Peak X	816.667 Hz
Peak Y	3.970 µV/sqrt(Hz)
Noise floor	1.370 µV/sqrt(Hz)

nsf_plots.NSFChart



nanosurf.lib.datatypes

Here new data types are introduced which are commonly useful in scientific application. Many other library components can handle them for ease of use.

SciVal: A floating point data type supporting units and nice formatting of numbers (is used by *gui.nsf_edit.NSFSciEdit* to show and edit numbers

PropVal: A class to stores values of different types. It emits a signal when the content is changed. It is designed to hold parameters and let gui-element connect to them. See *gui.bind_gui.connect_to_property()*

PropStore: A class to hold multiple ProVal attributes. It can be saved and restored to/from configuration files. See *datatypes.prop_val.save_to_ini_file()*

sci_channel: Stores an array of data points with units. Works with *list* and *numpy ndarray*. Sci_channels values can be directly plotted by *NSFChart* or analyzed by *math.sci_math* functions.

sci_stream: Stores multiple sci_channels as lists together with a 'timeline' array. sci_stream values can be directly plotted by *NSFChart* and used by *math.sci_math* functions

```
from nanosurf.lib.datatypes import sci_val
val = sci_val.from_str("20nm")
print(val)
>> 20.000 nm
```

```
from nanosurf.lib.datatypes import prop_val
My_time_prop = prop_val.PropVal(sci_val.SciVal(2.0, "s"))

class MySettings(prop_val.PropStore):
    def __init__(self):
        self.repetitions = prop_val.PropVal(int(30))
        self.show_ticks = prop_val.PropVal(bool(False))
```

```
from nanosurf.lib.datatypes import sci_stream as ss
mystream = ss.SciStream((data_time, data_sensor))
self.chart_plot.plot_stream(mystream)

fit_coeff = sci_math.calc_poly_fit(mystream, degree=1)
```

nanosurf.lib.math.sci_math

```
samplefrq = myscanline.get_stream_length()/ time_per_line

spec = sci_math.calc_fft(mystream.get_channel(0), samplerate=samplefrq)

noise_floor = sci_math.get_noise_floor(spec)

found, peak_x ,peak_y = sci_math.find_highest_peak(spec)
if found:
    self.spec_plot.set_marker(peak_x, peak_y)
    self.tableResults.set_value(0, peak_x, spec.get_stream_unit())
    self.tableResults.set_value(1, peak_y, spec.get_channel(0).unit)
```

Math functions not found in other libraries but useful for signal analysis are here.

Many of them can be feed with sci_channel or sci_stream data.

Sci-math.finde_peaks: Find all peaks in a sci_channel. Mostly useful for spectrum data.

Sci-math.finde_highest_peak: return the highest peak found in the sci_channel data

Sci-math.calc_poly_fit: Calculates the polynomial fit of a sci_stream.

Sci-math.calc_fft: Calculates the spectrum of an amplitude in a sci_channel with different windows. optional power spectrum too. Returns a spectrum as sci_stream.

Sci-math.create_compressed_log_spectrum: Reduces number of data points in large spectrum

Sci-math.get_total_harmonic_distortion Calculates the THD value of a spectrum

Sci-math.get_noise_floor Calculates the noise floor of a spectrum

And some more ……………………

nanosurf.lib.util

Various utility functions. Mostly for data export / import:

fileutil.create_filename_with_timestamp: return a string with a unique filename, optional with timestamp

fileutil.create_unique_folder: Find all peaks in a sci_channel. Mostly useful for spectrum data.

fileutil.create_folder: Make sure the folder exists

dataexport.savedata_txt/loaddata_txt: load/saves data in 'list' or sci_channel into a file

dataexport.saveplot_png: saves a chart plot into an image file

dataexport.save_results: saves a NSFNameValueTable content into a text file

And some more ....

nanosurf.lib.devices

Device drivers for different Nanosurf accessories can be found here

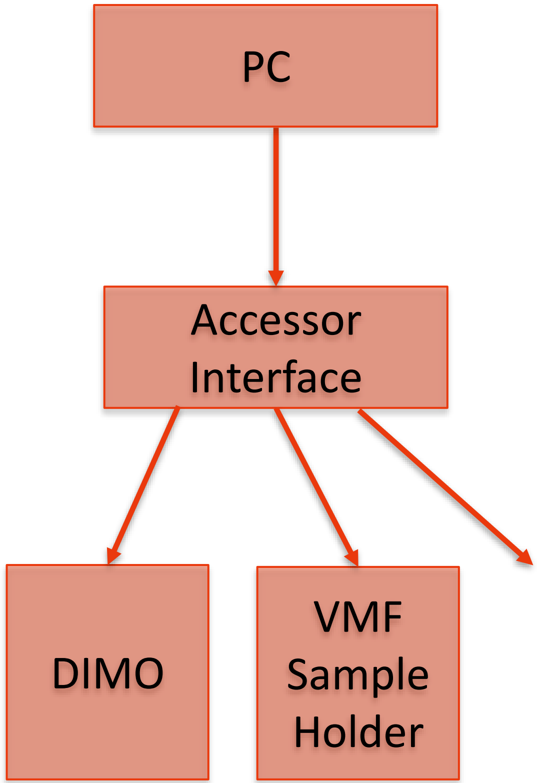
devises.accessor_interface: Class for communicating to devices connected to the accessory interface

Upcomming devices:

devises.dimo: Class to communicating with the digital inverted microscope accessories (not yet implemented)

devises.vmf Class to communicating with the variable magnetic field accessory (not yet implemented)

devisec.tc1: Class to communicate with the temperature controller (not yet implemented)



Getting started

Using the app_themplate. Coding principle, style guide

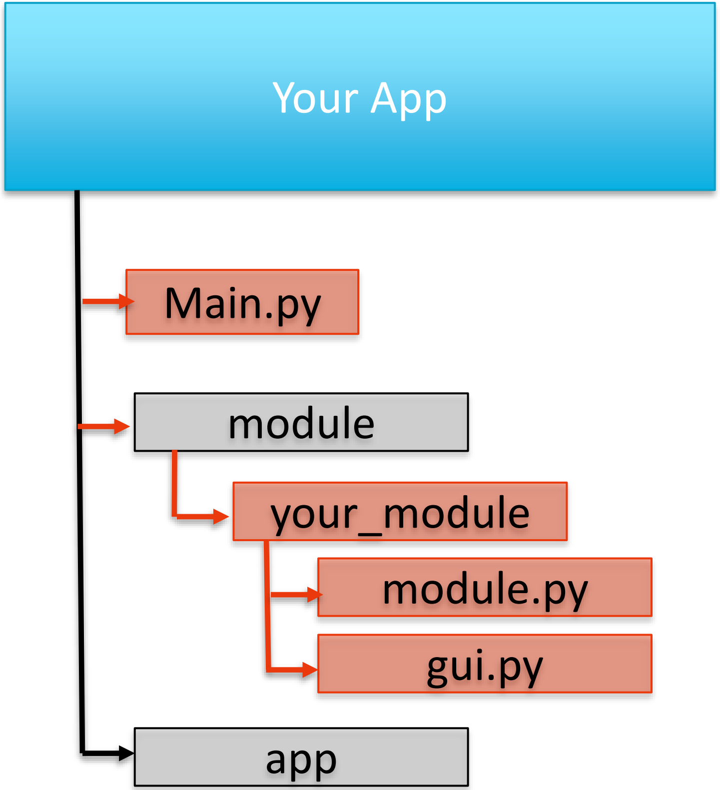
Getting Started

Creating your own app based on the app_template:

- Copy the app_template folder and rename it to your own project name.
- In your new project, copy the demo_module folder and rename it to your own function.
- Open Visual-Studio-Code and select File->Open Folder. Select your project folder just created.
- In *main.py* change *MyAppName* (line 15) to your project name..
- Add *import statements* of your new module (line 10) and add it with *self.add_module()* (line 30)
- Use demo_module as a reference how to create your own module. Later you can delete it and remove it from main-py

That's it. Now start coding your functionality in *module.py* and create the gui in *gui.py*

Start your app in debugging mode from any code window with F5-Key



Separating function and gui

A common mistake it to mangle functionality and gui elements in one. This make code maintenance and increasing functionality hard. A common method to solve this issue is the model/view pattern:

- The functionality of a software is written in one peace of code: You derive your code by subclassing the *ModuleBase* class and you program all the functionality as members in *module.py*
- Visual components of your functionality (entry elements and result visualization) are programmed in a class derived from *ModuleScreen* class and placed in *gui.py*
- Interaction from module -> gui is done by emitting and the gui is listening to such signals by connection to it (Use the Signal/Slot mechanism of Qt).
- Parameters of the module are defined by *ProVal* type attributes.
- Gui-Input-Elements are connected to them by the *bind_gui.connect_to_property()* function in *gui.py/bind_gui_elements()*
- By this the gui element get updated when the property is updated by the module and the module get informed when the user change a value in the gui.

module.py

```
class MyModul(ModuleBase):
    def __init__(self):
            self.image_size = PropVal(SciVal(2,"m"))
            self.image_size.connect(self.size_changed)

    def size_changed(self):
            # do something with a (e.g send to controller)
            self.spm_scan.ImageWidth = self.image_size.value
```

gui.py

```
class MyScreen(ModuleScreen):
    def do_setup_screen(self):
            self.my_edit = nsf_sci_edit.SciSciEdit("Size")
            bind_gui.connect_to_property(self.my_edit, module.image_size)
```

Signal/Slot Communication

A function module typically process data or measures data and has new information about its state or data content. Such state transition could be *file_loaded*, *start_measuring*, *new_data_available*, …

We use Qt.Signal/Slot mechanism for such communication. A function module send at events signals and any receivers interested in this new state or information get called.

A receiver interested in such information is the gui. So, it connects to signals and get a function called when the signal is emitted. The gui then can react accordingly (e.g., plot new data, disable parameters during measurement, …)

module.py

```
class MyModul(ModuleBase):
          sig_work_started = Signal()
          sig_work_done = Signal()

def do_something(self):
          sig_work_started.emit()
          # do some work
          sig_work_done.emit()
```

gui.py

```
class MyScreen(ModuleScreen):
  def do_setup_screen(self):
          self.module.sig_work_started.connect(self.measurement_started)
          self.module.sig_work_done.connect(self.show_result)

def measurement_started(self):
  # disable some gui elements

def show_result(self):
  # enable gui elemets
  # read result from module and plot result
```

Background Tasks

If a function module must do long lasting processing (e.g., measuring a data stream over 10s) then this task must be executed in a background thread, if not, the gui would be blocked and changing parameter or pressing a "Stop" button would not be possible.

The framework proved a class *SingleRunWorker* to simplify such background tasks. Derive a new class (e.g mytask) from it and implement the *do_work()* function. The background task can then be started by *mytask.start().*

It emits *sig_started*, *sig_finished* automatically. The module or the gui can connect to this signals.

module.py

```
class MyTask(SingleRunWorker):
  def do_work(self):
        # do some work (e.g measuring data)
        sig_new_data.emit()

class MyModul(ModuleBase):
        self.mylongwork = MyTask()

def start_measure_data(self):
        self.mylongwork.start()
```

gui.py

```
class MyScreen(ModuleScreen):
  self.module.mylongwork.sig_started.connect(self. measurement_started)
  self.module.mylongwork.sig_new_data.connect(self.show_result)
```

Coding and Doc Style

To create maintainable and readable code by others, programmers must follow coding style guides and documentation.

Luckily, Python has its code style guide well defined in PEP8.
(PEP is a naming convention like we have in Jira with NANO1245)

We follow this guide and in addition we use the typing hint style defined in PEP484. This helps the Visual Studio Code Editor to help programmers with tips and color the code correctly.

As documentation style we follow the numpy library doc style. Defined here: Numpy doc. Also, VS Code can read them and help programmers during coding.

Function names are lower case

Type hint

Return type

```
def create_filename(base_name: str, ext: str = '.dat', sep: str = "_") -> str:
    """ Construct a file name based on pattern and current date/time.
        The result will be something like 'my_data_20210613-100543.dat’
        this with base_name 'my_data’

    Parameters
    ----------
    base_name: str
        Mask of the name (e.g., 'my_data')

    ........

    Result
    ------
    str:
        constructed file name
    """
    current_datetime = datetime.now().strftime("%Y%m%d-%H%M%S")
    filename = base_name + sep + current_datetime + ext
    return filename
```

Numpy doc style: Description, Parameters and Result

```
""" This is the screen of the module
Copyright Nanosurf AG 2021
License - MIT
"""
import numpy as np
from PySide2 import QtWidgets
import nanosurf.lib.datatypes.sci_val as sci_val
from nanosurf.lib.gui import nsf_tables
from app import app_gui
from modules.scan_module import module, settings

class ResultTableID(nsf_tables.TableEntryIDs):
    """ identifier id are used in a nsf_table widget"""
    Items = 0
    Marker_X = 1
```

Import sequence: First general public lib, second nanosurf lib, third application

Class name and class attributs are uppercase