ruediPy documentation

Matthias Brennwald

Version November 7, 2017

Abstract

ruediPy is a collection of Python programs for instrument control and data acquisition using RUEDI instruments[(1)]. ruediPy also includes some GNU Octave (or Matlab) tools to load, process, and manipulate RUEDI data acquired with ruediPy Python classes.

ruediPy is distributed as free software under the GNU General Public License (see LICENSE.txt).

This document describes the ruediPy software only. The RUEDI instrument is described in a separate document[(1)].

Contents

1 Overview

ruediPy is a collection of Python programs for instrument control and data acquisition using RUEDI instruments. ruediPy also includes some GNU Octave (or Matlab) tools to load, process, and manipulate RUEDI data acquired with ruediPy Python classes. The RUEDI instrument itself is described in a separate document(1).

The Python classes for instrument control and data acquisition are designed to reflect the different hardware units of a RUEDI instrument, such as the mass spectrometer, selector valve, or probes for total gas pressure or temperature. These classes, combined with additional helper classes (e.g., for data file handling), allow writing simple Python scripts that perform user-defined procedures for a specific analysis task.

The GNU Octave tools (m-files) are designed to work hand-in-hand with the data files produced by the data acquisition parts of the Python classes. ⋆[1]

ruediPy is developed on Linux and Mac OS X systems, but should also work on any other system that runs Python and GNU Octave. ruediPy has been reported to (partially) work on Windows. Linux is the recommended choice and is assumed throughout this manual. Python 3.0 or newer is required.

2 Obtaining and installing ruediPy

ruediPy can be downloaded from http://brennmat.github.io/ruediPy either as a compressed archive file, or using Subversion or Git version control systems. ruediPy can be installed to just about any directory on the computer that is used for instrument control – but the user home directory (~/ruediPy) may seem like a sensible choice, and that's what is assumed throughout the examples shown in this manual.

As an example, here's a step-by-step list of terminal commands to install ruediPyon a Linux computer running Ubuntu 16.04. Other Linux distributions will be similar. The user account name in this example is "mRdemo", and this user account is enabled for sudo operations (i.e., it has 'admin' rights):

1. Update system software to latest versions and install basic software requirements for ruediPy:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install octave subversion python3-pip
```

[1] TO DO: expand this: load raw data, process / calibrate data, etc.

```
sudo apt-get install python3-serial python3-matplotlib python3-scipy
sudo pip3 install pydigitemp
```

2. Download ruediPy:

   ```
   svn co https://github.com/brennmat/ruediPy.git/trunk ~/ruediPy
   ```

3. Set permission to access the serial ports:

   ```
   sudo usermod -a -G dialout mRdemo
   ```

4. Prepare directories for `ruediPy` data files and measurement scripts:

   ```
   mkdir ~/data
   mkdir ~/scripts
   ```

5. The Shell and Python searchpaths for use with `ruediPy` are configured in a dedicated file (`ruediPy_paths.txt` in your home directory). Execute the following terminal commands to set up this file and the searchpaths (copy and paste to the Terminal prompt should work):

   ```
   echo PROJECT_SCRIPTS=~/scripts/my_project_scripts >> ~/ruediPy_paths.txt
   echo export PYTHONPATH=~/ruediPy/python >> ~/ruediPy_paths.txt
   echo export PYTHONPATH='$PYTHONPATH':'$PROJECT_SCRIPTS' >> ~/ruediPy_paths.txt
   echo export PATH='$PATH':'$PROJECT_SCRIPTS' >> ~/ruediPy_paths.txt
   echo source '$HOME'/ruediPy_paths.txt >> ~/.profile
   ```

   Adjust the `PROJECT_SCRIPTS` setting in the `ruediPy_paths.txt` file to reflect the directory where your measurement scripts are (or will be) stored.[2]

Log out and log back in to make the above changes active. You should also consider setting up the computer to avoid going to 'sleep' mode, because this might interrupt the measurement procedure.

[2] It is recommended to keep measurement scripts for different types of analysis or different projects in dedicated directories. Changing from one analysis type (or project) to another is achieved by adjusting `PROJECT_SCRIPTS` in the `ruediPy_paths.txt` file accordingly. Then log out and back in so the change will take effect.

3 Python classes

3.1 Overview

The Python classes are used to control the various hardware units of the RUEDI instruments, to acquire measurement data, and to write these data to well-formatted and structured data files.

Currently, the following classes are implemented:

- `rgams_SRS.py`: control and data acquisition from the SRS mass spectrometer
- `selectorvalve_VICI.py`: control of the VICI inlet valve
- `pressuresensor_WIKA.py`: control and data acquisition from the WIKA pressure sensor
- `datafile.py`: data file handling
- `misc.py`: helper functions

The Python class files are located at `~/ruediPy/python/classes/`. To make sure Python knows where to find the `ruediPy` Python classes, set your `PYTHONPATH` environment variable accordingly.[3]

These classes are continuously expanded and new classes are added to `ruediPy` as required by new needs or developments of the RUEDI instruments. The various methods / functions included are documented in the class files. Due to the ongoing development of the code, it seems futile to keep an up-to-date copy of the methods / functions documentation in this manual. Please refer to the detailed documentation in the class files directly.

3.2 Python classes reference

3.2.1 Class `rgams_SRS`

`ruediPy/python/classes/rgams_SRS.py`

ruediPy class for SRS RGA-MS control.

[3] A convenient method to achieve this on Linux or similar UNIXy systems is to put the following line to the `.profile` file: `export PYTHONPATH=~/ruediPy/python`

Method `calibrate_all`
`val = rgams_SRS.calibrate_all()`

Calibrate the internal coefficients for compensation of baseline offset offset and peak positions. This will zero the baseline for all noise-floor (NF) and detector combinations. See also the "CA" command int SRS RGA manual.

INPUT:
(none)

OUTPUT:
(none)

Method `filament_off`
`rgams_SRS.filament_off()`

Turn off filament current.

INPUT:
(none)

OUTPUT:
(none)

Method `filament_on`
`rgams_SRS.filament_on()`

Turn on filament current at default current value.

INPUT:
(none)

OUTPUT:

```
(none)
```

Method `get_DI`

```
x = rgams_SRS.get_DI(x)

Get current DI parameter value (peak-width tuning at low mz range)

INPUT:
(none)

OUTPUT:
x:  DI value (bit units)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method `get_DS`

```
x = rgams_SRS.get_DS(x)

Get current DS parameter value (peak-width tuning at high mz range)

INPUT:
(none)

OUTPUT:
x:  DS value (bit/amu units)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method get_RI

```
x = rgams_SRS.get_RI(x)

Get current RI parameter value (peak-position tuning at low mz range
/ RF voltage output at 0 amu, in mV).

INPUT:
(none)

OUTPUT:
x:  RI voltage (in mV)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method get_RS

```
x = rgams_SRS.get_RS(x)

Get current RS parameter value (peak-position tuning at high mz range
/ RF voltage output at 128 amu, in mV)

INPUT:
(none)

OUTPUT:
x:  RS voltage (in mV)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method get_default_RI

```
val = rgams_SRS.get_default_RI()

Return default RI value.
```

```
INPUT:
(none)

OUTPUT:
val:  default RI value
```

Method `get_default_RS`

```
val = rgams_SRS.get_default_RS()

Return default RS value.

INPUT:
(none)

OUTPUT:
val:  default RS value
```

Method `get_detector`

```
det = rgams_SRS.get_detector()

Return current detector (Faraday or electron multiplier)

INPUT:
(none)

OUTPUT:
det:  detecor (string):
det='F' for Faraday
det='M' for electron Multiplier
```

Method get_electron_emission
```
val = rgams_SRS.get_electron_emission()

Return electron emission current (in mA)

INPUT:
(none)

OUTPUT:
val:  electron emission current in mA (float)
```

Method get_electron_energy
```
val = rgams_SRS.get_electron_energy()

Return electron energy of the ionizer (in eV).

INPUT:
(none)

OUTPUT:
val:  electron energy in eV
```

Method get_multiplier_default_hv
```
val = rgams_SRS.get_multiplier_default_hv()

Return default value to be used for electron multiplier (CEM) high voltage
(bias voltage).
NOTE: the value returned is NOT the value stored in the memory of the
RGA head.  This function is just a wrapper that returns the default high
voltage value set in the RGA object (e.g., during initialisation of the
object).

INPUT:
(none)
```

```
OUTPUT:
val:  voltage
```

Method `get_multiplier_hv`

```
val = rgams_SRS.get_multiplier_hv()

Return electron multiplier (CEM) high voltage (bias voltage).

INPUT:
(none)

OUTPUT:
val:  voltage
```

Method `get_noise_floor`

```
val = rgams_SRS.get_noise_floor()

Get noise floor (NF) parameter for RGA measurements (noise floor controls
gate time, i.e., noise vs.  measurement speed).

INPUT:
(none)

OUTPUT:
val:  NF noise floor parameter value, 0...7 (integer)
```

Method `has_multiplier`

```
val = rgams_SRS.has_multiplier()

Check if MS has electron multiplier installed.
```

```
INPUT:
(none)

OUTPUT:
val:  result flag, val = 0 --> MS has no multiplier, val <> 0:  MS has
multiplier
```

Method `ionizer_degas`

```
val = rgams_SRS.ionizer_degas(duration)

Run the ionizer degas procedure (see SRS RGA manual).  Only run this
with sufficiently good vacuum!

INPUT:
duration:  degas time in minutes (0...20 / integer)

OUTPUT:
(none)
```

Method `label`

```
l = rgams_SRS.label()

Return label / name of the RGAMS object.

INPUT:
(none)

OUTPUT:
l:  label / name (string)
```

Method mz_max

```
val = rgams_SRS.mz_max()

Determine highest mz value supported by the MS.

INPUT:
(none)

OUTPUT:
val:  max.  supported mz value (int)
```

Method param_IO

```
ans = rgams_SRS.param_IO(cmd,ansreq)

Set / read parameter value of the SRS RGA.

INPUT:
cmd:  command string that is sent to RGA (see RGA manual for commands
and syntax)
ansreq:  flag indicating if answer from RGA is expected:
ansreq = 1:  answer expected, check for answer
ansreq = 0:  no answer expected, don't check for answer

OUTPUT:
ans:  answer / result returned from RGA
```

Method peak

```
val,unit = rgams_SRS.peak(mz,gate,f,add_to_peakbuffer=True)

Read out detector signal at single mass (m/z value).

INPUT:
mz:  m/z value (integer)
gate:  gate time (seconds) NOTE: gate time can be longer than the max.
```

gate time supported by the hardware (2.4 seconds). If so, the multiple peak readings will be averaged to achieve the requested gate time.
f: file object for writing data (see datafile.py). If f = 'nofile', data is not written to any data file.
add_to_peakbuffer (optional): flag to choose if peak value is added to peakbuffer (default: add_to_peakbuffer=True)

OUTPUT:
val: signal intensity (float)
unit: unit (string)

NOTE FROM THE SRS RGA MANUAL:
Single mass measurements are commonly performed in sets
where several different masses are monitored sequencially
and in a merry-go-round fashion.
For best accuracy of results, it is best to perform the consecutive mass measurements in a set with the same type of detector
and at the same noise floor (NF) setting.
Fixed detector settings eliminate settling time problems
in the electrometer and in the CDEM's HV power supply.

Method peak_zero_loop
peak_zero_loop (mz,detector,gate,ND,NC,datafile,clear_peakbuf_cond=True,clear_peakbuf_main=True,plot_cond=False)

Cycle PEAKS and ZERO readings given mz values.

INPUT:
mz: list of tuples with peak m/z value (for PEAK) and delta-mz (for ZERO). If delta-mz == 0, no ZERO value is read.
detector: detector string ('F' or 'M')
gate: integration time
ND: number of data cycles recorded to the current data file
NC: number of cycles used for conditioning of the detector and electronics before recording the data (not written to datafile)
datafile: file object for writing data (see datafile.py). If f = 'nofile', data is not written to any data file.

```
clear_peakbuf_cond:  flag to set clearing of peakbuffer before conditioning
cycles on/off (optional, default=True)
clear_peakbuf_main:  flag to set clearing of peakbuffer before main cycles
on/off (optional, default=True)
plot_cond:  flag to set plotting of readings used for detector conditioning
(inclusion of values in peakbuffer)

OUTPUT:
(none)
```

Method `peakbuffer_add`

```
rgams_SRS.peakbuffer_add(t,mz,intens,unit)

Add data to PEAKS data buffer

INPUT:
t:  epoch time
mz:  mz values
intens:  intensity value
det:  detector (char/string)
unit:  unit of intensity value (char/string)

OUTPUT:
(none)
```

Method `peakbuffer_clear`

```
rgams_SRS.peakbuffer_clear()

Clear data in PEAKS data buffer

INPUT:
(none)

OUTPUT:
```

(none)

Method peakbuffer_set_length
rgams_SRS.peakbuffer_set_length(N)

Set max. length of peakbuffer

INPUT:
N: number of PEAK values

OUTPUT:
(none)

Method plot_peakbuffer
rgams_SRS.plot_peakbuffer()

Plot trend (or update plot) of values in PEAKs data buffer (e.g. after adding data)
NOTE: plotting may be slow, and it may therefore be a good idea to keep the update interval low to avoid affecting the duty cycle.

INPUT:
(none)

OUTPUT:
(none)

Method plot_scan
rgams_SRS.plot_scan(mz,intens,unit,cumsum_mz=[],cumsum_val=[])

Plot scan data

```
INPUT:
mz:  mz values (x-axis)
intens:  intensity values (y-axis)
unit:  intensity unit (string)
cumsum_mz,cumsum_val (optional):  cumulative sum of peak data (mz and
sum values), as used for peak centering

OUTPUT:
(none)
```

Method `print_status`

```
rgams_SRS.print_status()

Print status of the RGA head.

INPUT:
(none)

OUTPUT:
(none)
```

Method `scan`

```
M,Y,unit = rgams_SRS.scan(low,high,step,gate,f)

Analog scan

INPUT:
low:  low m/z value (integer or decimal)
high:  high m/z value (integer or decimal)
step:  scan resolution (number of mass increment steps per amu)
step = integer number (10...25) --> use given number (high number equals
small mass increments between steps)
step = '*' use default value (step = 10)
```

```
gate:  gate time (seconds)
f:  file object or 'nofile':
if f is a DATAFILE object, the scan data is written to the current data
file
if f = 'nofile' (string), the scan data is not written to a datafile

OUTPUT:
M: mass values (mz, in amu)
Y: signal intensity values (float)
unit:  unit of Y (string)
```

Method `set_DI`

```
rgams_SRS.set_DI(x)

Set DI parameter value (Peak width parameter at m/z = 0)

INPUT:
x:  parameter value (bit units)

OUTPUT:
(none)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method `set_DS`

```
rgams_SRS.set_DS(x)

Set DS parameter value (Peak width parameter for m/z > 0)

INPUT:
x:  parameter value (bit/amu units)

OUTPUT:
```

(none)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"

Method set_RI
rgams_SRS.set_RI(x)

Set RI parameter value (peak-position tuning at low mz range / RF voltage output at 0 amu, in mV)

INPUT:
x: RI voltage (mV)

OUTPUT:
(none)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"

Method set_RS
rgams_SRS.set_RS(x)

Set RS parameter value (peak-position tuning at high mz range / RF voltage output at 128 amu, in mV)

INPUT:
x: RS voltage (mV)

OUTPUT:
(none)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"

Method set_detector
rgams_SRS.set_detector(det)

Set current detetector used by the MS (direct the ion beam to the Faraday or electron multiplier detector).
NOTE: To activate the electron multiplier (CEM), the default high voltage (bias voltage) as returned by self.get_multi_default_hv() is used (this is NOT necessarily the same as the default value stored in the RGA head).

INPUT:
det: detecor (string):
det='F' for Faraday
det='M' for electron multiplier

OUTPUT:
(none)

Method set_electron_emission
rgams_SRS.set_electron_emission(val)

Set electron emission current.

INPUT:
val: electron emission current in mA (0 ... 3.5 mA)

OUTPUT:
(none)

Method set_electron_energy
rgams_SRS.set_electron_energy(val)

```
Set electron energy of the ionizer.

INPUT:
val:  electron energy in eV

OUTPUT:
(none)
```

Method `set_gate_time`

```
val = rgams_SRS.set_gate_time()

Set noise floor (NF) parameter for RGA measurements according to desired
gate time (by choosing the best-match NF value).

INPUT:
gate:  gate time in (fractional) seconds

OUTPUT:
(none)

NOTE (1):
FROM THE SRS RGA MANUAL:
Single mass measurements are commonly performed in sets
where several different masses are monitored sequencially
and in a merry-go-round fashion.
For best accuracy of results, it is best to perform the consecutive
mass measurements in a set with the same type of detector
and at the same noise floor (NF) setting.
Fixed detector settings eliminate settling time problems
in the electrometer and in the CDEM HV power supply.

NOTE (2):
Experiment gave the following gate times vs NF parameter values:

NF gate (seconds)
0 2.4
```

```
1 1.21
2 0.48
3 0.25
4 0.163
5 0.060
6 0.043
7 0.025
```

Method `set_multiplier_hv`

```
rgams_SRS.set_multiplier_hv(val)

Set electron multiplier (CEM) high voltage (bias voltage).

INPUT:
val:  voltage

OUTPUT:
(none)
```

Method `set_noise_floor`

```
val = rgams_SRS.set_noise_floor()

Set noise floor (NF) parameter for RGA measurements (noise floor controls
gate time, i.e., noise vs.  measurement speed).

INPUT:
NF: noise floor parameter value, 0...7 (integer)

OUTPUT:
(none)
```

Method set_peakbuffer_plot_max_y
rgams_SRS.set_peakbuffer_plot_max_y(val)

Set upper limit of y range in peakbuffer plot.

INPUT:
val: upper limit of y-axis range

OUTPUT:
(none)

Method set_peakbuffer_plot_min_y
rgams_SRS.set_peakbuffer_plot_min_y(val)

Set lower limit of y range in peakbuffer plot.

INPUT:
val: lower limit of y-axis range

OUTPUT:
(none)

Method tune_peak_position
rgams_SRS.tune_peak_position(mz,gate,det,max_iter=10,max_delta_mz=0.05,use_-
defaults=False)

Automatically adjust peak positions in mass spectrum to make sure peaks
show up at the correct mz values. This is done by scanning peaks at
different mz values, and determining their offset in the mz spectrum.
The mass spectromter parameters are then adjusted to minimize the mz
offsets (parameters RI and RF, which define the peak positions at mz=0
and mz=128). The procedure start with the currently set RI and RS values
(if use_defaults = False) or the default values (if they are set and
use_defaults = True). This needs at least two distinct peak mz values,

```
one at a low and one at a high mz value.  The procedure is repeated until
either the peak position offsets at mz=0 and mz=128 are less than max_-
delta_mz or the number of iterations has reached max_iter.

INPUT:
peaks:  list of (mz,width,gate,detector) tuples, where peaks should be
scanned and tuned
mz = mz value of peak (center of the scan)
width = width of the peak (relative to center mz value)
gate:  gate time to be used for the scan
detector:  detector to be used for the scan ('F' or 'M')
max_iter (optional):  max.  number of repetitions of the tune procedure
maxdelta_mz (optional):  tolerance of mz offset at mz=0 and mz=128.  If
the absolute offsets at mz=0 and mz=128 after tuning are less than maxdelta_-
z after tuning, the tuning procedure is stopped.
use_defaults:  flag to set if default RI and RS values are used to start
the tuning procedure.  Default value:  use_defaults = False

OUTPUT:
(none)

EXAMPLE:
>>> MS = rgams_SRS ( serialport = '/dev/serial/by-id/usb-WuT_USB_Cable_-
2_WT2016234-if00-port0' , label = 'MS_MINIRUEDI_TEST', max_buffer_points
= 1000 )
>>> MS.filament_on()
>>> MS.tune_peak_position([14,18,28,32,40,44,84],[0.2,0.2,0.025,0.1,0.4,0.1,2.4],[

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method `warning`

```
rgams_SRS.warning(msg)

Issue warning about issues related to operation of MS.

INPUT:
```

```
msg:  warning message (string)

OUTPUT:
(none)
```

Method zero

```
val,unit = rgams_SRS.zero(mz,mz_offset,gate,f)

Read out detector signal at single mass with relative offset to given
m/z value (this is useful to determine the baseline near a peak at a
given m/z value), see rgams_SRS.peak())
The detector signal is read at mz+mz_offset

INPUT:
mz:  m/z value (integer)
mz_offset:  offset relative m/z value (integer).
gate:  gate time (seconds) NOTE: gate time can be longer than the max.
gate time supported by the hardware (2.4 seconds).  If so, the multiple
zero readings will be averaged to achieve the requested gate time.
f:  file object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.

OUTPUT:
val:  signal intensity (float)
unit:  unit (string)

NOTE FROM THE SRS RGA MANUAL:
Single mass measurements are commonly performed in sets
where several different masses are monitored sequencially
and in a merry-go-round fashion.
For best accuracy of results, it is best to perform the consecutive
mass measurements in a set with the same type of detector
and at the same noise floor (NF) setting.
Fixed detector settings eliminate settling time problems
in the electrometer and in the CDEM's HV power supply.
```

3.2.2 Class `selectorvalve_VICI`

`ruediPy/python/classes/selectorvalve_VICI.py`

ruediPy class for VICI valve control. This assumes the serial protocol used with VICI's older "microlectric" actuators. For use with the newer "universal" actuators, they must be set to "legacy mode" using the "LG1" command (see page 8 of VICI document "Universal Electric Actuator Instruction Manual"). The self.set_legacy command may be useful for this.

Method `getpos`

```
pos = selectorvalve_VICI.getpos()

Get valve position

INPUT:
(none)

OUTPUT:
pos:  valve postion (integer)
```

Method `label`

```
label = selectorvalve_VICI.label()

Return label / name of the SELECTORVALVE object

INPUT:
(none)

OUTPUT:
label:  label / name (string)
```

Method `set_legacy`

```
selectorvalve_VICI.set_legacy()
```

```
Set communication protocol to LEGACY mode (useful to make the newer valve
controlers compatible with the LEGACY protocol).

INPUT:
(none)

OUTPUT:
(none)
```

Method `setpos`

```
selectorvalve_VICI.setpos(val,f)

Set valve position

INPUT:
val:  new valve position (integer)
f:  datafile object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.

OUTPUT:
(none)
```

Method `warning` No method description available.

3.2.3 Class `pressuresensor_WIKA`

`ruediPy/python/classes/pressuresensor_WIKA.py`

ruediPy class for WIKA pressure sensor control.

Method `label`

```
label = pressuresensor_WIKA.label()
```

Return label / name of the PRESSURESENSOR object

INPUT:
(none)

OUTPUT:
label: label / name (string)

Method plot_pressbuffer
pressuresensor_WIKA.plot_pressbuffer()

Plot trend (or update plot) of values in pressure data buffer (e.g. after adding data)
NOTE: plotting may be slow, and it may therefore be a good idea to keep the update interval low to avoid affecting the duty cycle.

INPUT:
(none)

OUTPUT:
(none)

Method pressbuffer_add
pressuresensor_WIKA.pressbuffer_add(t,p,unit)

Add data to pressure data buffer

INPUT:
t: epoch time
p: pressure value
unit: unit of pressure value (char/string)

OUTPUT:

```
(none)
```

Method `pressure`

```
press,unit = pressuresensor_WIKA.pressure(f,add_to_pressbuffer=True)

Read out current pressure value.

INPUT:
f:  file object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.
add_to_pressbuffer (optional):  flag to indicate if data get appended
to pressure buffer (default=True)

OUTPUT:
press:  pressure value in hPa (float)
unit:  unit of pressure value (string)
```

Method `serial_checksum`

```
cs = pressuresensor_WIKA.serial_checksum( cmd )

Return checksum used for serial port communication with WIKA pressure
sensor.

INPUT:
cmd:  serial-port command string without checksum

OUTPUT:
cs:  checksum byte
```

Method `warning`

```
pressuresensor_WIKA.warning(msg)
```

```
Issue warning about issues related to operation of pressure sensor.

INPUT:
msg:  warning message (string)

OUTPUT:
(none)
```

3.2.4 Class `temperaturesensor_MAXIM`

`ruediPy/python/classes/temperaturesensor_MAXIM.py`

ruediPy class for MAXIM DS1820 type temperature sensors (wrapper class for pydigitemp package).

Method `label`

```
label = temperaturesensor_MAXIM.label()

Return label / name of the TEMPERATURESENSOR object

INPUT:
(none)

OUTPUT:
label:  label / name (string)
```

Method `plot_tempbuffer`

```
temperaturesensor_MAXIM.plot_tempbuffer()

Plot trend (or update plot) of values in temperature data buffer (e.g.
after adding data)
NOTE: plotting may be slow, and it may therefore be a good idea to keep
the update interval low to avoid affecting the duty cycle.
```

```
INPUT:
(none)

OUTPUT:
(none)
```

Method `tempbuffer_add`

```
temperaturesensor_MAXIM.tempbuffer_add(t,T,unit)

Add data to temperature data buffer

INPUT:
t:  epoch time
T: temperature value
unit:  unit of pressure value (char/string)

OUTPUT:
(none)
```

Method `temperature`

```
temp,unit = temperaturesensor_MAXIM.temperature(f)

Read out current temperaure value.

INPUT:
f:  file object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.
add_to_tempbuffer (optional):  flag to indicate if data get appended
to temperature buffer (default=True)

OUTPUT:
temp:  temperature value (float)
unit:  unit of temperature value (string)
```

Method `warning`
`temperaturesensor_MAXIM.warning(msg)`

```
Issue warning about issues related to operation of pressure sensor.

INPUT:
msg:  warning message (string)

OUTPUT:
(none)
```

3.2.5 Class `datafile`

`ruediPy/python/classes/datafile.py`

ruediPy class for handling of data files.

Method `basepath`
`pat = datafile.basepath()`

```
Return the base path where datafiles are stored

INPUT:
(none)

OUTPUT:
pat:  datafile base path (string)
```

Method `close`
`datafile.close()`

```
Close the currently open data file (if any)

INPUT:
(none)

OUTPUT:
(none)
```

Method `fid`

```
f = datafile.fid()

Return the file ID / object of the current file

INPUT:
(none)

OUTPUT:
f:  datafile object
```

Method `label`

```
lab = datafile.label()

Return label / name of the DATAFILE object

INPUT:
(none)

OUTPUT:
lab:  label / name (string)
```

Method name

```
n = datafile.name()

Return the name the current file (or empty string if not datafile has
been created)

INPUT:
(none)

OUTPUT:
n:  ile name (string)
```

Method next

```
datafile.next( typ='MISC' , samplename='' , standardconc=[] )

Close then current data file (if it's still open) and start a new file.

INPUT:
typ (optional):  analysis type (string, default:  typ = 'MISC'). The
analysis type is written to the data file, and is appended to the file
name.  typ can be one of the following analysis types:
typ = 'SAMPLE' (for sample analyses)
typ = 'STANDARD' (for standard / calibration analyses)
typ = 'BLANK' (for blank analyses)
typ = 'MISC' (for miscellaneous analysis types, useful for testing, maintenance,
or similar purposes)
samplename (optional, only used if typ='SAMPLE'): description, name,
or ID of sample (string)
standardconc (optional, only used if typ='STANDARD'): standard gas information,
list of 3-tuples, one tuple for each mz-value).  Each tuple has the following
3 fields:
field-1:  name of species (string)
field-2:  volumetric species concentration in standard gas
field-3:  mz value used for analysis of this species

example for N2 and Ar-40 in air, analyzed on mz=28 and mz=40:  standardconc
= [ ('N2',0.781,28) , ('Ar-40',0.9303,40) ]
```

```
OUTPUT:
(none)
```

Method `warning`

```
datafile.warning(msg)

Warn about issues related to DATAFILE object

INPUT:
msg:  warning message (string)

OUTPUT:
(none)
```

Method `write_comment`

```
datafile.write_comment(caller,cmt)

Write COMMENT line to the data file.

INPUT:
caller:  label / name of the calling object (string)
cmt:  comment string

OUTPUT:
(none)
```

Method `write_peak`

```
datafile.write_peak(caller,mz,intensity,unit,det,gate,timestmp)

Write PEAK data line to the data file.
```

```
INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
label:  name/label of the calling object (string)
mz:  mz value (integer)
intensity:  peak intensity value (float)
unit:  unit of peak intensity value (string)
det:  detector (string), e.g., det='F' for Faraday or det='M' for multiplier
gate:  gate time (float)
timestmp:  timestamp of the peak measurement (see misc.now_UNIX)

OUTPUT:
(none)
```

Method `write_pressure`

```
datafile.write_pressure(caller,label,value,unit,timestmp)

Write PRESSURE data line to the data file.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
label:  name/label of the calling object (string)
value:  pressure value (float)
unit:  unit of peak intensity value (string)
timestmp:  timestamp of the pressure measurement (see misc.now_UNIX)

OUTPUT:
(none)
```

Method `write_sample_desc`

```
datafile.write_sample_desc(self,desc)

Write line with sample description (e.g., name or ID of sample)
```

```
INPUT:
desc:  sample description, name, or ID (string)

OUTPUT:
(none)
```

Method `write_scan`

```
datafile.write_scan(caller,mz,intensity,unit,det,gate,timestmp)

Write PEAK data line to the data file.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
label:  name/label of the calling object (string)
mz:  mz values (floats)
intensity:  intensity values (floats)
unit:  unit of intensity values (string)
det:  detector (string), e.g., det='F' for Faraday or det='M' for multiplier
gate:  gate time (float)
timestmp:  timestamp of the peak measurement (see misc.now_UNIX)

OUTPUT:
(none)
```

Method `write_standard_conc`

```
datafile.write_standard_conc(species,conc,mz)

Write line with standard/calibration gas information to data file:  name,
concentration/mixing ratio, and mz value of gas species.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
species:  name of gas species (string)
conc:  volumetric concentration / mixing ratio (float)
```

```
mz:  mz value (integer)

OUTPUT:
(none)
```

Method `write_temperature`

```
datafile.write_temperature(caller,label,value,unit,timestmp)

Write TEMPERATURE data line to the data file.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
label:  name/label of the calling object (string)
value:  temperature value (float)
unit:  unit of peak intensity value (string)
timestmp:  timestamp of the temperature measurement (see misc.now_UNIX)

OUTPUT:
(none)
```

Method `write_valve_pos`

```
datafile.write_valve_pos(caller,position,timestmp)

Write multi-port valve position data line to the data file.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
label:  name/label of the calling object (string)
position:  valve position (integer)
timestmp:  timestamp of the peak measurement (see misc.now_UNIX)

OUTPUT:
(none)
```

Method write_zero
datafile.write_zero(caller,mz,mz_offset,intensity,unit,det,gate,timestmp)

Write ZERO data line to the data file.

INPUT:
caller: type of calling object, i.e. the "data origin" (string)
label: name/label of the calling object (string)
mz: mz value (integer)
mz_offset: mz offset value (integer, positive offset corresponds to higher mz value)
intensity: zero intensity value (float)
unit: unit of peak intensity value (string)
det: detector (string), e.g., det='F' for Faraday or det='M' for multiplier
gate: gate time (float)
timestmp: timestamp of the peak measurement (see misc.now_UNIX)

OUTPUT:
(none)

Method writeln
datafile.writeln(caller,identifier,data,timestmp)

Write a text line to the data file (format: TIMESTAMP CALLER[LABEL] IDENTIFIER: DATA). CALLER, LABEL, and IDENTIFIER should not contain spaces or similar white space (will be removed before writing to file). If LABEL == '' or LABEL == CALLER, the [LABEL] part is omitted.

INPUT:
caller: type of calling object, i.e. the "data origin" (string)
label: name/label of the calling object (string)
identifier: data type identifier (string)
data: data / info string
timestmp: timestamp of the data in unix time (see misc.now_UNIX)

```
OUTPUT:
(none)
```

3.2.6 Class `misc`

`ruediPy/python/classes/misc.py`

ruediPy class with helper functions.

Method `ask_for_value`

```
x = misc.ask_for_value(msg='Enter value = ')

Print a message asking the user to enter something, wait until the user
presses the ENTER key, and return the value.

INPUT:
msg (optional):  message

OUTPUT:
x:  user value (string)
```

Method `now_UNIX`

```
dt = misc.now_UNIX()

Return date/time as UNIX time / epoch (seconds after Jan 01 1970 UTC)

INPUT:
(none)

OUTPUT:
dt:  date-time (UNIX / epoch time)
```

Method now_string

```
dt = misc.now_string()

Return string with current date and time

INPUT:
(none)

OUTPUT:
dt:  date-time (string) in YYYY-MM-DD hh:mm:ss format
```

Method sleep

```
misc.sleep( wait , msg='' )

Wait for a specified time and print a countdown message.  The user can
skip the countdown by pressing CTRL-C.

INPUT:
wait:  waiting time (seconds)
msg (optional):  message

OUTPUT:
(none)
```

Method user_menu

```
x = misc.user_menu(menu,title='Choose an option')

Show a "menu" for selection of different user options, return user choice
based on key pressed by user.

INPUT:
menu:  menu entries (tuple of strings)
title (optional):  title of the menu (default='Choose an option')
```

```
OUTPUT:
x:  number of menu choice

EXAMPLE:
k = misc.user_menu( title='Choose dinner' , menu=('Chicken','Burger','Veggies')
)
```

Method `wait_for_enter`

```
misc.wait_for_enter(msg='Press ENTER to continue.')

Print a message and wait until the user presses the ENTER key.

INPUT:
msg (optional):  message

OUTPUT:
(none)
```

Method `warnmessage`

```
misc.warnmessage(caller,msg)

Print a warning message

INPUT:
caller:  caller label / name of the calling object (string)
msg:  warning message

OUTPUT:
(none)
```

4 GNU Octave tools

$\star$[4]

5 Examples

$\star$[5]

References

[1] M. S. Brennwald, M. Schmidt, J. Oser, and R. Kipfer. A portable and autonomous mass spectrometric system for on-site environmental gas analysis. *Environmental Science and Technology*, 50(24):13455–13463, 2016. doi: 10.1021/acs.est.6b03669.

[4] TO DO: add content

[5] TO DO: add content