API reference for labbench

Release labbench

**Dan Kuester, Shane Allman,
Paul Blanchard, Yao Ma**

Aug 31, 2023

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Single devices

A `Device` object is the central encapsulates python control over a single piece of laboratory equipment or a software instance. Organizing automation with the `Device` classes in this way immediately provides shortcuts for

- automatic logging
- coercion between python types and low-level/over-the-wire data types
- constraints on instrument parameters
- multi-threaded connection management
- hooks for real-time heads-up displays

1.1.1 A basic VISA device wrapper

Let's start by a simple demonstration with VISA (https://en.wikipedia.org/wiki/Virtual_instrument_software_architecture) instrument automation. To do this, we write wrapper that specializes the general-purpose `labbench.VISADevice` backend, which uses the `pyvisa` (https://pyvisa.readthedocs.io/) library. This helps to expedite SCPI (https://en.wikipedia.org/wiki/Standard_Commands_for_Programmable_Instruments) messaging patterns that arise often.

The example below gives a simplified working example of a more complete commercial power sensor (https://github.com/usnistgov/ssmdevices/blob/main/ssmdevices/instruments/power_sensors.py):

```
import labbench as lb

@lb.property.visa_keying(
    # these are the default SCPI query and write formats
    query_fmt="{key}?",
    write_fmt="{key} {value}",

    # map python True and False values to these SCPI strings
    remap={True: "ON", False: "OFF"}
)
class PowerSensor(lb.VISADevice):
    RATES = "NORM", "DOUB", "FAST"

    # SCPI string keys and bounds on the parameter values,
    # taken from the instrument programming manual
    initiate_continuous = lb.property.bool(
```

(continues on next page)

(continued from previous page)

```
        key="INIT:CONT", label="trigger continuously if True"
    )
    trigger_count = lb.property.int(
        key="TRIG:COUN", min=1, max=200,
        help="acquisition count", label="samples"
    )
    measurement_rate = lb.property.str(
        key="SENS:MRAT", only=RATES, case=False, label="Hz"
    )
    sweep_aperture = lb.property.float(
        key="SWE:APER", min=20e-6, max=200e-3,
        help="measurement duration", label="s"
    )
    frequency = lb.property.float(
        key="SENS:FREQ",
        min=10e6,
        max=18e9,
        step=1e-3,
        help="input signal center frequency",
        label="Hz",
    )

    def preset(self):
        """revert to instrument preset state"""
        self.write("SYST:PRES")

    def fetch(self):
        """acquire measurements as configured"""
        response = self.query("FETC?")

        if self.trigger_count == 1:
            return float(response)
        else:
            return [float(s) for s in response.split(",")]
```

This object is already enough for us to automate an RF power measurements!

1.1.2 Usage of device wrappers

An automation script uses

```
# specify the VISA address to use the power sensor
sensor = PowerSensor("USB0::0x2A8D::0x1E01::SG56360004::INSTR")

# connection to the power sensor hardware at the specified address
# is held open until exiting the "with" block
with sensor:
    # apply the instrument preset state
    sensor.preset()

    # set acquisition parameters on the power sensor
```

(continues on next page)

(continued from previous page)

```
    sensor.frequency = 1e9
    sensor.measurement_rate = "FAST"
    sensor.trigger_count = 200
    sensor.sweep_aperture = 20e-6
    sensor.initiate_continuous = True

    # retreive the 200 measurement samples
    power = sensor.fetch()
```

The usage here is simple because the methods and traits for automation can be discovered easily through tab completion in most IDEs. The device connection remains open for all lines inside the `with` block..

1.2 Multiple devices

To organize and operate multiple Device instances, `labbench` provides `Rack` objects. These act as a container for aspects of automation needed to perform into a resuable automation task, including `Device` objects, other `Rack` objects, and automation functions. On exception, they ensure that all `Device` connections are closed.

1.2.1 Basic implementation

The following example creates simple automation tasks for a swept-frequency microwave measurement built around one `Device` each:

```
import labbench as lb

# some custom library of Device drivers
from myinstruments import MySpectrumAnalyzer, MySignalGenerator


class Synthesizer(lb.Rack):
    # inputs needed to run the rack: in this case, a Device
    inst: MySignalGenerator

    def setup(self, *, center_frequency):
        self.inst.preset()
        self.inst.set_mode("carrier")
        self.inst.center_frequency = center_frequency
        self.inst.bandwidth = 2e6

    def arm(self):
        self.inst.rf_output_enable = True

    def stop(self):
        self.inst.rf_output_enable = False


class Analyzer(lb.Rack):
    # inputs needed to run the rack: in this case, a Device
    inst: MySpectrumAnalyzer
```

(continues on next page)

(continued from previous page)

```
    def setup(self, *, center_frequency):
        self.inst.load_state("savename")
        self.inst.center_frequency = center_frequency

    def acquire(self, *, duration):
        self.inst.trigger()
        lb.sleep(duration)
        self.inst.stop()

    def fetch(self):
        # testbed data will have a column called 'spectrogram', which
        # point to subdirectory containing a file called 'spectrogram.csv'
        return dict(spectrogram=self.inst.fetch_spectrogram())


class SweptMeasurement(lb.Rack):
    # inputs needed to run the rack: in this case, child Rack objects
    generator: Synthesizer
    detector: Analyzer

    def single(self, center_frequency, duration):
        self.generator.setup(center_frequency)
        self.detector.setup(center_frequency)

        self.generator.arm()
        self.detector.acquire(duration)
        self.generator.stop()

        return self.detector.fetch()

    def run(self, frequencies, duration):
        ret = []

        for freq in frequencies:
            ret.append(self.single(freq, duration))

        return duration
```

1.2.2 Usage in test scripts

When executed to run test scripts, create Rack instances with input objects according to their definition:

```
sa = MySpectrumAnalyzer(resource="a")
sg = MySignalGenerator(resource="b")

with SweptMeasurement(generator=Synthesizer(sg), detector=Analyzer(sa)) as sweep:
    measurement = sweep.run(frequencies=[2.4e9, 2.44e9, 2.48e9], duration=1.0)
```

They open and close connections with all `Device` children by use of `with` methods. The connection state of all `SweptMeasurement` children are managed together, and all are closed in the event of an exception.

CHAPTER

TWO

MULTI-THREADED CONCURRENCY

`labbench` includes simplified concurrency support for this kind of I/O-constrained operations like waiting for instruments to perform long operations. It is not suited for parallelizing CPU-intensive tasks because the operations share a single process on one CPU core, instead of multiprocessing, which may be able to spread operations across multiple CPU cores.

Here are very fake functions that just use `time.sleep` to block. They simulate longer instrument calls (such as triggering or acquisition) that take some time to complete.

Notice that `do_something_3` takes 3 arguments (and returns them), and that `do_something_4` raises an exception.

```
import time

def do_something_1 ():
    print('start 1')
    time.sleep(1)
    print('end 1')
    return 1

def do_something_2 ():
    print('start 2')
    time.sleep(2)
    print('end 2')
    return 2

def do_something_3 (a,b,c):
    print('start 3')
    time.sleep(2.5)
    print('end 3')
    return a,b,c

def do_something_4 ():
    print('start 4')
    time.sleep(3)
    raise ValueError('I had an error')
    print('end 4')
    return 4

def do_something_5 ():
    print('start 5')
    time.sleep(4)
    raise IndexError('I had a different error')
```

(continues on next page)

(continued from previous page)

```
    print('end 5')
    return 4
```

Here is the simplest example, where we call functions `do_something_1` and `do_something_2` that take no arguments and raise no exceptions:

```
import labbench as lb

results = lb.concurrently(do_something_1, do_something_2)
print(f'results: {results}')
```

```
start 1
start 2
end 1
end 2
```

```
{'do_something_1': 1, 'do_something_2': 2}
```

We can also pass functions by wrapping the functions in `Call()`, which is a class designed for this purpose:

```
results = lb.concurrently(do_something_1, lb.Call(do_something_3, 1,2,c=3))
results
```

```
start 1
start 3
end 1
end 3
```

```
{'do_something_1': 1, 'do_something_3': (1, 2, 3)}
```

More than one of the functions running concurrently may raise exceptions. Tracebacks print to the screen, and by default `ConcurrentException` is also raised:

```
from labbench import concurrently, Call

results = concurrently(do_something_4, do_something_5)
results
```

```
start 4
start 5
Traceback (most recent call last):
  File "<ipython-input-1-73606d5b193d>", line 24, in do_something_4
    raise ValueError('I had an error')
ValueError: I had an error
Traceback (most recent call last):
  File "<ipython-input-1-73606d5b193d>", line 31, in do_something_5
    raise IndexError('I had a different error')
IndexError: I had a different error
Traceback (most recent call last):
  File "<ipython-input-1-73606d5b193d>", line 24, in do_something_4
    raise ValueError('I had an error')
```

(continues on next page)

(continued from previous page)

```
ValueError: I had an error
Traceback (most recent call last):
  File "<ipython-input-1-73606d5b193d>", line 31, in do_something_5
    raise IndexError('I had a different error')
IndexError: I had a different error
```

```
---------------------------------------------------------------------------
ConcurrentException                       Traceback (most recent call last)
<ipython-input-5-6e564c4e58e6> in <module>
      1 from labbench import concurrently, Call
      2
----> 3 results = concurrently(do_something_4, do_something_5)
      4 results

ConcurrentException: 2 call(s) raised exceptions
```

the `catch` flag changes concurrent exception handling behavior to return values of functions that did not raise exceptions (instead of raising `ConcurrentException`). The return dictionary only includes keys for functions that did not raise exceptions.

```
from labbench import concurrently, Call

results = concurrently(do_something_4, do_something_1, catch=True)
results
```

```
start 4
start 1
end 1
Traceback (most recent call last):
  File "<ipython-input-1-73606d5b193d>", line 24, in do_something_4
    raise ValueError('I had an error')
ValueError: I had an error
```

```
{'do_something_1': 1}
```

CHAPTER

THREE

LOGGING DEVICE STATES AND TEST RESULTS TO A DATABASE

A number of tools are included in `labbench` to streamline acquisition of test data into a database. A couple of methods are

- Automatically monitoring attributes in `state` and logging changes
- Saving postprocessed data in the as a new column

The data management supports automatic relational databasing. Common non-scalar data types (`pandas.DataFrame`, `numpy.array`, long strings, files generated outside of the data tree, etc.) are automatically stored relationally — placed in folders and referred to in the database. Other data can be forced to be relational by dynamically generating relational databases on the fly.

3.1 File conventions

All labbench data save functionality is implemented in tables with *pandas* DataFrame backends. Here are database storage formats that are supported:

Format	File extension(s)	Data management class	flag to use record file format (http://ssm.ipages.nist.gov/labbench/labbencl	Comments
sqlite	.db	labbench.SQLiteLogger (http://ssm.ipages.nist.gov/labbenc	'sqlite'	Scales to larger databases than csv
csv	.csv,.csv.g	labbench.CSVLogger (http://ssm.ipages.nist.gov/labbenc	'csv'	Easy to inspect

Several formats are supported only as relational data (data stored in a file in the subdirectory instead of directly in the). Certain types of data as values into the database manager automatically become relational data when you call the `append` method of the data manager:

Format	File extension(s	python type conversion	set_record file format (http://ssm.ipages.nist.gov/labbench/labben flag	Comments anagedata.Relat
feather	.f	iterables of numbers and strings; pd.DataFrame	'feather'	Python 3.x only
json (http://www.json.c	.json	iterables of numbers and strings; pd.DataFrame	'json'	
csv	.csv	iterables of numbers and strings; pd.DataFrame	'csv'	
python pickle (https://docs.pythc	.pickle	any	'pickle'	fallback if the chosen relational format fails
text files	.txt	string or bytes longer than `text_relatio`	N/A	set `text_relational_min` when you instantiate the database manager
arbitrary files generated outside the file tree	*	strings containing filesystem path	N/A	

In the following example, we will use an sqlite master database, and csv record files.

3.2 Example

Here is a emulated "dummy" instrument. It has a few state settings similar to a simple power sensor. The state descriptors (`initiate_continuous`, `output_trigger`, etc.) are defined as local types, which means they don't trigger communication with any actual devices. The `fetch_trace` method generates a "trace" drawn from a uniform distribution.

```
import sys
sys.path.insert(0,'..')
import labbench as lb
import numpy as np
import pandas as pd

class EmulatedInstrument(lb.EmulatedVISADevice):
    """ This "instrument" makes mock data and instrument states to
        demonstrate we can show the process of setting
        up a measurement.
    """
    class state (lb.EmulatedVISADevice.state):
```

(continues on next page)

(continued from previous page)

```
        initiate_continuous:bool = lb.property(key='INIT:CONT')
        output_trigger:bool = lb.property(key='OUTP:TRIG')
        sweep_aperture:float = lb.property(min=20e-6, max=200e-3,help='s')
        frequency:float = lb.property(min=10e6, max=18e9,step=1e-3,help='Hz')

    def trigger(self):
        """ This would tell the instrument to start a measurement
        """
        pass

    def fetch_trace(self, N=1001):
        """ Generate N points of junk data as a pandas series.
        """
        values = np.random.normal(size=N)
        index = np.linspace(0,self.state.sweep_aperture,N)
        series = pd.Series(values,index=index,name='voltage')
        series.index.name = 'time'
        return series
```

Now make a loop to execute 100 test runs with two emulated instruments, and log the results with a relational SQLite database. I do a little setup to start:

1. Define a couple of functions `inst1_trace` and `inst2_trace` that collect my data
2. Instantiate 2 instruments, `inst1` and `inst2`
3. Instantiate the logger with `lb.SQLiteLogger('test.db', 'state')`. The arguments specify the name of the sqlite database file and the name of the table where the following will be stored: 1) the instrument state info will be stored, 2) locations of data files, and 3) any extra comments we add with `db.write()`.

Remember that use of the `with` statement automatically connects to the instruments, and then ensures that the instruments are properly closed when we leave the `with` block (even if there is an exception).

```
def inst1_trace ():
    """ Return a 1001-point trace
    """
    inst1.trigger()
    return inst1.fetch_trace(51)

def inst2_trace ():
    """ This one returns only one point
    """
    inst2.trigger()
    return inst2.fetch_trace(1).values[0]

# Root directory of the database
db_path = r'data'

# Seed the data dictionary with some global data
data = {'dut': 'DUT 15'}

Nfreqs = 101

with EmulatedInstrument()        as inst1,\
```

(continues on next page)

(continued from previous page)

```
    EmulatedInstrument()        as inst2,\
    lb.SQLiteLogger(db_path)  as db:
        # Catch any changes in inst1.state and inst2.state
        db.observe_states([inst1,inst2])

        # Update inst1.state.sweep_aperture on each db.append
        db.observe_states(inst1, always='sweep_aperture')

        # Store trace data in csv format
        db.set_relational_file_format('csv')

        # Perform a frequency sweep. The frequency will be logged to the
        # database, because we configured it to observe all state changes.
        inst2.state.frequency = 5.8e9
        for inst1.state.frequency in np.linspace(5.8e9, 5.9e9, Nfreqs):
            # Collect "test data" by concurrently calling
            # inst1_trace and inst2_trace
            data.update(lb.concurrently(inst1_trace, inst2_trace))

            # Append the new data as a row to the database.
            # Each key is a column in the database (which will be added
            # dynamically to the database if needed). More keys and values
            # are also added corresponding to attributes inst1.state and inst2.state
            db.append(comments='trying for 1.21 GW to time travel',
                      **data)
```

```
..\labbench\data.py:841: UserWarning: set_nonscalar_file_type is deprecated; set when␣
↪creating
                        the database object instead with the nonscalar_output flag
  the database object instead with the nonscalar_output flag""")
```

3.2.1 Reading and exploring the data

The master database is now populated with the test results and subdirectories are populated with trace data. `labbench` provides the function `read` as a shortcut to load the sqlite database into a pandas dataframe. Each state is a column in the database. The logger creates columns named as a combination of the device name ('inst1') and name of the corresponding device state.

```
%pylab inline
master = lb.read(f'{db_path}/master.db')
master.head()
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
                           comments     dut  \
id
0   trying for 1.21 GW to time travel  DUT 15
1   trying for 1.21 GW to time travel  DUT 15
2   trying for 1.21 GW to time travel  DUT 15
3   trying for 1.21 GW to time travel  DUT 15
```

(continues on next page)

(continued from previous page)

```
4   trying for 1.21 GW to time travel  DUT 15

                                    host_log                   host_time  \
id
0   0 2019-06-26 112657.415229\host_log.txt  2019-06-26 11:26:57.415229
1   1 2019-06-26 112657.416226\host_log.txt  2019-06-26 11:26:57.416226
2   2 2019-06-26 112657.418228\host_log.txt  2019-06-26 11:26:57.418228
3   3 2019-06-26 112657.419227\host_log.txt  2019-06-26 11:26:57.419227
4   4 2019-06-26 112657.420241\host_log.txt  2019-06-26 11:26:57.420241

    inst1_frequency  inst1_sweep_aperture  \
id
0      5.800000e+09              0.178945
1      5.801000e+09              0.045648
2      5.802000e+09              0.010702
3      5.803000e+09              0.155814
4      5.804000e+09              0.180664

                                    inst1_trace  inst2_frequency  \
id
0   0 2019-06-26 112657.415229\inst1_trace.csv     5.800000e+09
1   1 2019-06-26 112657.416226\inst1_trace.csv     5.800000e+09
2   2 2019-06-26 112657.418228\inst1_trace.csv     5.800000e+09
3   3 2019-06-26 112657.419227\inst1_trace.csv     5.800000e+09
4   4 2019-06-26 112657.420241\inst1_trace.csv     5.800000e+09

    inst2_sweep_aperture  inst2_trace
id
0               0.184968     1.833018
1               0.086791    -0.099672
2               0.039928    -1.026572
3               0.148706     0.692184
4               0.142890    -1.635934
```

This is a pandas DataFrame object. There is extensive information about how to use dataframes on the pandas website (http://pandas.pydata.org/pandas-docs/stable/). Suppose we want to bring in the data from the traces, which are in a collection of waveform files specified under the `inst1_trace` column. The function `labbench.expand` serves to flatten the database with respect to data files that were generated on each row.

```
waveforms = lb.read_relational(f'{db_path}/master.db', 'inst1_trace', ['dut', 'inst1_
↪frequency'])
waveforms
```

```
         dut  inst1_frequency                                    inst1_trace  \
0     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
1     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
2     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
3     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
4     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
5     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
6     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
7     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
```

(continues on next page)

(continued from previous page)

```
8     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
9     DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
10    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
11    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
12    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
13    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
14    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
15    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
16    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
17    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
18    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
19    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
20    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
21    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
22    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
23    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
24    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
25    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
26    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
27    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
28    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
29    DUT 15     5.800000e+09    0 2019-06-26 112657.415229\inst1_trace.csv
...      ...              ...                                          ...
5121  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5122  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5123  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5124  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5125  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5126  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5127  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5128  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5129  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5130  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5131  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5132  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5133  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5134  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5135  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5136  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5137  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5138  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5139  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5140  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5141  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5142  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5143  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5144  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5145  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5146  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5147  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5148  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
5149  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv
```

(continues on next page)

(continued from previous page)

```
5150  DUT 15     5.900000e+09  100 2019-06-26 112657.549225\inst1_trace.csv

      inst1_trace_id  inst1_trace_time  inst1_trace_voltage
0                  0          0.000000            -0.353873
1                  1          0.000849             0.778037
2                  2          0.001699             1.586201
3                  3          0.002548            -0.088197
4                  4          0.003397             0.144149
5                  5          0.004247             0.142975
6                  6          0.005096            -0.324535
7                  7          0.005945             1.900828
8                  8          0.006795            -0.685564
9                  9          0.007644            -0.889614
10                10          0.008493             0.527775
11                11          0.009343            -0.831135
12                12          0.010192             0.196392
13                13          0.011041             0.661497
14                14          0.011891             1.032419
15                15          0.012740             0.258957
16                16          0.013589            -0.868437
17                17          0.014439             0.460822
18                18          0.015288            -1.503195
19                19          0.016137            -1.182400
20                20          0.016987             0.132486
21                21          0.017836            -0.168882
22                22          0.018685            -1.107198
23                23          0.019535            -0.326377
24                24          0.020384            -0.461302
25                25          0.021233            -0.016560
26                26          0.022083            -0.246392
27                27          0.022932            -0.164883
28                28          0.023781            -0.537337
29                29          0.024631            -0.007081
...              ...               ...                  ...
5121              21          0.080807             0.742581
5122              22          0.084654            -0.674925
5123              23          0.088502             0.327832
5124              24          0.092350             1.471126
5125              25          0.096198            -0.133763
5126              26          0.100046             0.625860
5127              27          0.103894             0.154129
5128              28          0.107742            -0.905493
5129              29          0.111590             0.040421
5130              30          0.115438            -0.261152
5131              31          0.119286            -0.333401
5132              32          0.123134            -0.097314
5133              33          0.126982             0.263614
5134              34          0.130830             0.321361
5135              35          0.134678            -0.307449
5136              36          0.138525             0.437606
5137              37          0.142373            -0.512926
5138              38          0.146221             1.005609
```

(continues on next page)

(continued from previous page)

```
5139              39          0.150069             0.273236
5140              40          0.153917            -0.274306
5141              41          0.157765             0.974446
5142              42          0.161613             1.196802
5143              43          0.165461            -1.151176
5144              44          0.169309             0.255143
5145              45          0.173157            -0.271001
5146              46          0.177005            -0.221826
5147              47          0.180853            -0.037811
5148              48          0.184701            -0.286925
5149              49          0.188549            -0.254916
5150              50          0.192396             0.366316

[5151 rows x 6 columns]
```

now we can manipulate the results to look for meaningful information in the data.

```
import seaborn as sns; sns.set(context='notebook', style='ticks', font_scale=1.5) #␣
↪Theme stuff

waveforms.plot(x='inst1_frequency',y='inst1_trace_voltage',kind='hexbin')
xlabel('Frequency (Hz)')
ylabel('Voltage (arb units)')
```

```
Text(0, 0.5, 'Voltage (arb units)')
```

CHAPTER

FOUR

DEVICE OBJECTS

A series of short working examples here illustrate the use of labbench `Device` classes for experiment automation. The python programming interface is in the module of the same name, but it is convenient to import it as `lb` for shorthand.

```
import labbench as lb
lb.show_messages('debug')
```

Laboratory automation wrappers are implemented as classes derived from `lb.Device`. All of them share common basic types features designed to make their usage discoverable and convenient. The goal here is to show how to navigate these objects to get started quickly automating lab tasks.

Wrappers for specific instruments are not included with `labbench`, only low-level python plumbing and utility functions to streamline lab automation. Specific implementation is left for other libraries.

4.1 Overview

The `Device` class and subclasses represent in a sense only a definition with instructions for automating a specified type of lab tool. To bring these to life and control objects in the lab, the most general steps are to

1. construct an object from the class,
2. open a connection, and then
3. use the object's attributes to perform automation tasks as needed.

Let's start with a simple automation demo for a simple 2 instrument experiment.

```
import labbench as lb
import numpy as np
from sim_visa import PowerSupply, SpectrumAnalyzer

# VISA Devices take a standard address string to create a resource
spectrum_analyzer = SpectrumAnalyzer('GPIB::15::INSTR')
supply = PowerSupply('USB::0x1111::0x2222::0x2468::INSTR')

# show SCPI traffic
lb.show_messages('debug')

# `with` blocks open the devices, then closes them afterward
with supply, spectrum_analyzer:
    print(supply.backend, supply._rm, repr(supply.read_termination))
    supply.voltage = 5
```

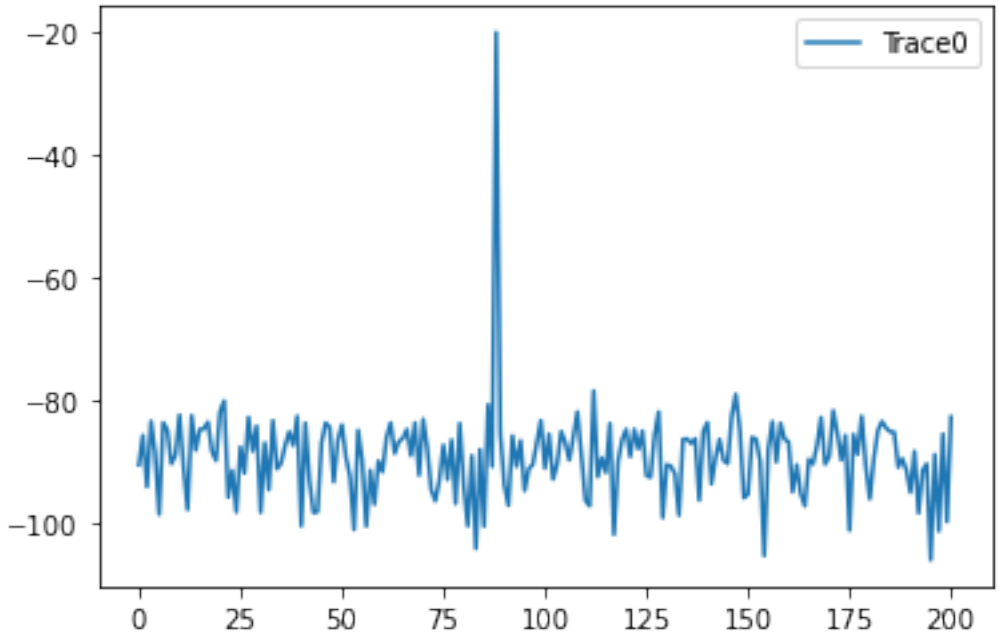
(continues on next page)

(continued from previous page)

```
    supply.output_enabled = True

    trace_dB = 10*np.log10(spectrum_analyzer.fetch_trace())
trace_dB.plot();
```

```
 DEBUG  2021-05-27 12:30:37.611 • PowerSupply('USB::0x1111::0x2222::0x2468::INSTR'):␣
↪opened
 DEBUG  2021-05-27 12:30:37.612 • SpectrumAnalyzer('GPIB::15::INSTR'): opened
 DEBUG  2021-05-27 12:30:37.612 • PowerSupply('USB::0x1111::0x2222::0x2468::INSTR'):␣
↪write "':VOLT:IMM:AMPL 5.0'"
 DEBUG  2021-05-27 12:30:37.613 • PowerSupply('USB::0x1111::0x2222::0x2468::INSTR'):␣
↪write "'OUTP 1'"
 DEBUG  2021-05-27 12:30:37.614 • SpectrumAnalyzer('GPIB::15::INSTR'): query_ascii_
↪values 'TRACE?'
 DEBUG  2021-05-27 12:30:37.621 • SpectrumAnalyzer('GPIB::15::INSTR'):       ->␣
↪(DataFrame with shape (201, 1))
 DEBUG  2021-05-27 12:30:37.637 • SpectrumAnalyzer('GPIB::15::INSTR'): closed
 DEBUG  2021-05-27 12:30:37.638 • PowerSupply('USB::0x1111::0x2222::0x2468::INSTR'):␣
↪closed
USBInstrument at USB0::0x1111::0x2222::0x2468::0::INSTR @sim '\n'
```



These instruments are emulated - under the hood they are *pyvisa-sim* instruments, configured in [sim_visa.yaml], which act as simple value stores for a few fake SCPI commands and sources of "canned" arrays of data. The demo labbench Device classes that control them are implemented in [sim_visa.py] (subclassed from `lb.Device` -> `lb.VISADevice` -> `lb.SimulatedVISADevice`).

4.2 Workflow

4.2.1 Constructing objects

These Device classes (like other VISA instruments) need a VISA address in order to point to a specific instrument. To discover information about this and other available initialization parameters, use python help() or the '?' magic in ipython or jupyter:

```
SpectrumAnalyzer?
```

```
Init signature:
SpectrumAnalyzer(
    resource: str = '',
    *,
    read_termination: str = '\n',
    write_termination: str = '\n',
    sweeps: int = 1,
)
Docstring:
a fake Spectrum Analyzer that returns fixed trace data

Value Attributes:
    resource (str): VISA address string (allow_none=True)
    read_termination (str): end of line string to expect in query replies (cache=True)
    write_termination (str): end of line string to send after writes (cache=True)
    sweeps (int): number of traces to acquire (min=1)

Property Attributes:
    isopen (bool): is the backend ready?
    status_byte (dict): instrument status decoded from '*STB?' (sets=False)
    identity (str): identity string reported by the instrument (key='*IDN',sets=False,
↪cache=True)
    options (str): options reported by the instrument (key='*OPT',sets=False,cache=True)
    frequency (float): center frequency (key=':FREQ',min=10000000.0,max=180000000000.0)
Init docstring:
Arguments:
    resource (str): VISA address string (allow_none=True)
    read_termination (str): end of line string to expect in query replies (cache=True)
    write_termination (str): end of line string to send after writes (cache=True)
    sweeps (int): number of traces to acquire (min=1)
File:           c:\users\dkuester\documents\src\labbench\examples\sim_visa.py
Type:           HasTraitsMeta
Subclasses:
```

Other options are also available here, such as the transport settings `read_termination` and `read_termination`, or the number of traces to acquire in calls to `fetch_trace`.

These can also be set or changed after object construction by setting the value attributes, for example `spectrum_analyzer.resource = 'GPIB::15::INSTR'` or `supply.resource = 'USB::0x1111::0x2222::0x2468::INSTR'`. The complete list of these parameters is shown under "Value Attributes", which also lists read-only values that can't be changed and are not constructor arguments.

4.2.2 Opening device connections

In automation scripts, it is good practice to use a context block (that `with` statement) to open connections. This ensures all of the devices open and close together, even when exceptions are raised.

For interactive use on the python/ipython/jupyter prompt, this is less convenient. For this purpose, device objects also expose explicit `open` and `close` methods. As an example, a simple check for instrument response to automation could look like this,

```
>>> supply.open()
>>> print(supply.output_enabled)
False
>>> # (...look at the instrument to verify output is disabled)
>>> supply.output_enabled = True
>>> # (...verify instrument output is enabled)
```

This type of exploration is a good way to learn the capabilities of a device interactively.

4.2.3 Automating with open devices

Python's introspection tools give more opportunities to discover the API exposed by a device object. This is important because the methods and other attributes vary from one type of Device class to another. The below uses `dir` to show the list of all *public* attributes (those that don't start with `'_'`).

```
attrs = [
    name
    for name in dir(SpectrumAnalyzer)
    if not name.startswith('_') # filter by name
]

print(f'public attributes of SpectrumAnalyzer: {attrs}\n')

# discover the 'query' method common to VISA all devices
SpectrumAnalyzer.query?
```

```
public attributes of SpectrumAnalyzer: ['backend', 'close', 'concurrency', 'fetch_trace',
↪ 'frequency', 'get_key', 'identity', 'isopen', 'list_resources', 'open', 'options',
↪'overlap_and_block', 'preset', 'query', 'query_ascii_values', 'read_termination',
↪'resource', 'set_key', 'status_byte', 'suppress_timeout', 'sweeps', 'wait', 'write',
↪'write_termination', 'yaml_source']
```

```
Signature: SpectrumAnalyzer.query(self, msg: str, timeout=None) -> str
Docstring:
queries the device with an SCPI message and returns its reply.

Handles debug logging and adjustments when in overlap_and_block
contexts as appropriate.

Arguments:
    msg: the SCPI message to send
File:      c:\users\dkuester\anaconda3\lib\site-packages\labbench\_backends.py
Type:      function
```

Trait attributes that cast to python types with validation are definitions in classes, but become interactive values in device objects:

```
print(f'class: SpectrumAnalyzer.sweeps == {SpectrumAnalyzer.sweeps}')
print(f'object: spectrum_analyzer.sweeps == {signal_analyzer.sweeps}')
```

```
class: SpectrumAnalyzer.sweeps == value.int(default=1,min=1)
object: spectrum_analyzer.sweeps == 1
```

```
signal_analyzer.open
SpectrumAnalyzer.open
```

```
<function labbench._backends.VISADevice.open(self)>
```

4.3 Generalizing from the example

Different subclasses expose different method functions and attribute variables to wrap the underling low-level API. Still, several characteristics are standardized:

- connection management through `with` block or `open`/`close` methods
- an `isopen` property to indicate connection status
- `resource` is accepted by the constructor, and may be changed afterward as a class attribute
- hooks are available for data loggers and UIs to observe automation calls

Device subclasses for different types of instruments and software differ in

- the types of resource and configuration information
- the specific resource of the class provided to control the device

This gets more complicated when handling multiple devices, because connection failures leave a combination of open and closed:

```
try:
    base.open()
    visa.open() # fails because its resource doesn't exist on the host

    # we don't get this far after visa.open() raises an exception
    print("doing useful automation here")
    visa.close()
    base.close()
except:
    # we're left with a mixture of connection states
    assert base.isopen==True and visa.isopen==False

    # ...so we have to clean up the stray connection manually :(
    base.close()
```

Context management is easier and more clear. Everything inside the `with` block executes only if all devices open successfully, and ensures cleanup so that all devices are closed afterward.

```
try:
    with base, visa: # does both base.open() and visa.open()
        print('we never get in here, because visa.open() fails!')
except:
    # context management ensured a base.close() after visa.open() failed,
    assert base.isopen==False and visa.isopen==False
```

```
data logging, type checking,and numerical bounds validation.

These features are common to all `Device` classes (and derived classes). To get started,␣
↪provide  by minimum working examples. Examples will use  we'll look into the more␣
↪specialized capabilities provided by other `Device` subclasses included `labbench` for␣
↪often-used backend APIs like serial and VISA.
```

4.3.1 Example

Here are very fake functions that just use `time.sleep` to block. They simulate longer instrument calls (such as triggering or acquisition) that take some time to complete.

Notice that `do_something_3` takes 3 arguments (and returns them), and that `do_something_4` raises an exception.

```
import labbench as lb
```

Here is the simplest example, where we call functions `do_something_1` and `do_something_2` that take no arguments and raise no exceptions:

```
from labbench import concurrently

results = concurrently(do_something_1, do_something_2)
results
```

```
start 1
start 2
end 1
end 2
```

```
{'do_something_1': 1, 'do_something_2': 2}
```

```
results
```

```
{'do_something_1': 1, 'do_something_2': 2}
```

```
do_something_1.__name__
```

```
'do_something_1'
```

We can also pass functions by wrapping the functions in `Call()`, which is a class designed for this purpose:

```
from labbench import concurrently, Call

results = concurrently(do_something_1, Call(do_something_3, 1,2,c=3))
results
```

```
start 1
start 3
end 1
end 3
```

```
{'do_something_1': 1, 'do_something_3': (1, 2, 3)}
```

More than one of the functions running concurrently may raise exceptions. Tracebacks print to the screen, and by default `ConcurrentException` is also raised:

```
from labbench import concurrently, Call

results = concurrently(do_something_4, do_something_5)
results
```

```
start 4
start 5
```

```
Traceback (most recent call last):
  File "<ipython-input-1-736d6d5b193d>", line 24, in do_something_4
    raise ValueError('I had an error')
ValueError: I had an error
Traceback (most recent call last):
  File "<ipython-input-1-736d6d5b193d>", line 31, in do_something_5
    raise IndexError('I had a different error')
IndexError: I had a different error
Traceback (most recent call last):
  File "C:\Users\dkuester\AppData\Local\Continuum\anaconda3\lib\site-packages\IPython\
↪core\interactiveshell.py", line 3267, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-6-6e564c4e58e6>", line 3, in <module>
    results = concurrently(do_something_4, do_something_5)
  File "C:\Users\dkuester\AppData\Local\Continuum\anaconda3\lib\site-packages\labbench\
↪util.py", line 899, in concurrently
    return concurrently_call(*objs, **kws)
  File "C:\Users\dkuester\AppData\Local\Continuum\anaconda3\lib\site-packages\labbench\
↪util.py", line 600, in concurrently_call
    f'{len(tracebacks)} call(s) raised exceptions')
labbench.util.ConcurrentException: 2 call(s) raised exceptions
```

the `catch` flag changes concurrent exception handling behavior to return values of functions that did not raise exceptions (instead of raising `ConcurrentException`). The return dictionary only includes keys for functions that did not raise exceptions.

```
from labbench import concurrently, Call

results = concurrently(do_something_4, do_something_1, catch=True)
```

(continues on next page)

(continued from previous page)

```
results
```

```
start 4
start 1
end 1
```

```
{'do_something_1': 1}
```

CHAPTER

FIVE

API REFERENCE

5.1 labbench

class labbench.**CSVLogger**(*path=None*, ***, *append=False*, *text_relational_min=1024*, *force_relational=['host_log']*, *dirname_fmt='{id} {host_time}'*, *nonscalar_file_type='csv'*, *metadata_dirname='metadata'*, *tar=False*, *git_commit_in=None*)

Bases: `RelationalTableLogger`

Store data, value traits, and property traits to disk into a root database formatted as a comma-separated value (CSV) file.

This extends `Aggregator` to support

1. queuing aggregate property trait of devices by lists of dictionaries;
2. custom metadata in each queued aggregate property trait entry; and
3. custom response to non-scalar data (such as relational databasing).

Parameters

- **path** (`str`) – Base path to use for the root database
- **overwrite** (`bool`) – Whether to overwrite the root database if it exists (otherwise, append)
- **text_relational_min** – Text with at least this many characters is stored as a relational text file instead of directly in the database
- **force_relational** – A list of columns that should always be stored as relational data instead of directly in the database
- **nonscalar_file_type** – The data type to use in non-scalar (tabular, vector, etc.) relational data
- **metadata_dirname** – The name of the subdirectory that should be used to store metadata (device connection parameters, etc.)
- **tar** – Whether to store the relational data within directories in a tar file, instead of subdirectories

open()

Instead of calling *open* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with CSVLogger('my.csv') as db:
    ### do the data acquisition here
    pass
```

would instantiate a *CSVLogger* instance, and also guarantee a final attempt to write unwritten data is written, and that the file is closed when exiting the *with* block, even if there is an exception.

class labbench.**Call**(*func*, **args*, ***kws*)

Bases: `object`

Wrap a function to apply arguments for threaded calls to *concurrently*. This can be passed in directly by a user in order to provide arguments; otherwise, it will automatically be wrapped inside *concurrently* to keep track of some call metadata during execution.

set_queue(*queue*)

Set the queue object used to communicate between threads

classmethod wrap_list_to_dict(*name_func_pairs*)

adjusts naming and wraps callables with Call

class labbench.**Device**(*resource: str = ''*)

Bases: `HasTraits`, `Ownable`

base class for labbench device wrappers.

Drivers that subclass *Device* share

- standardized connection management via context blocks (the *with* statement)
- hooks for automatic data logging and heads-up displays
- API style consistency
- bounds checking and casting for typed attributes

Note: This *Device* base class has convenience functions for device control, but no implementation.

Some wrappers for particular APIs labbench Device subclasses:

- VISADevice: pyvisa,
- ShellBackend: binaries and scripts
- Serial: pyserial
- DotNetDevice: pythonnet

(and others). If you are implementing a driver that uses one of these backends, inherit from the corresponding class above, not *Device*.

---

close()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

open()

Backend implementations overload this to open a backend connection to the resource.

class labbench.**DotNetDevice**(*resource: str = ''*)

Bases: *Device*

Base class for .NET library wrappers based on pythonnet.

To implement a DotNetDevice subclass:

```
import labbench as lb

class MyLibraryWrapper(lb.DotNetDevice, libary=<imported python module colocated␣
↪with dll>, dll_name='mylibrary.dll')
    ...
```

When a DotNetDevice is instantiated, it looks to load the dll from the location of the python module and dll_name.

- `backend` is None after open and is available for replacement by the subclass

open()

dynamically import a .net CLR as a python module at self.dll

class labbench.**Email**(*resource: str = 'smtp.nist.gov'*, ***, *port: int = 25*, *sender: str = 'myemail@nist.gov'*, *recipients: list = ['myemail@nist.gov']*, *success_message: str = 'Test finished normally'*, *failure_message: str = 'Exception ended test early'*)

Bases: *Device*

Sends a notification message on disconnection. If an exception was thrown, this is a failure subject line with traceback information in the main body. Otherwise, the message is a success message in the subject line. Stderr is also sent.

close()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

open()

Backend implementations overload this to open a backend connection to the resource.

send_summary()

Send the email containing the final property trait of the test.

class labbench.**HDFLogger**(*path*, ***, *append=False*, *key_fmt='{id} {host_time}'*, *git_commit_in=None*)

Bases: `RelationalTableLogger`

Store data and activity from value and property sets and gets to disk into a root database formatted as an HDF file.

This extends `Aggregator` to support

1. queuing aggregate property trait of devices by lists of dictionaries;
2. custom metadata in each queued aggregate property trait entry; and
3. custom response to non-scalar data (such as relational databasing).

Parameters

- **path** (`str`) – Base path to use for the root database
- **append** (`bool`) – Whether to append to the root database if it already exists (otherwise, raise IOError)
- **key_fmt** (`str`) – format to use for keys in the h5

open()

Instead of calling *open* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with HDFLogger('my.csv') as db:
    ### do the data acquisition here
    pass
```

would instantiate a *CSVLogger* instance, and also guarantee a final attempt to write unwritten data is written, and that the file is closed when exiting the *with* block, even if there is an exception.

class labbench.**LabviewSocketInterface**(*resource: str = '127.0.0.1'*, ***, *tx_port: int = 61551*, *rx_port: int = 61552*, *delay: float = 1*, *timeout: float = 2*, *rx_buffer_size: int = 1024*)

Bases: *Device*

Base class demonstrating simple sockets-based control of a LabView VI.

Keyed get/set with lb.property are implemented by simple ' command value'. Subclasses can therefore implement support for commands in specific labview VI similar to VISA commands by assigning the commands implemented in the corresponding labview VI.

- backend

connection object mapping {'rx': rxsock, 'tx': txsock}

Type
dict

clear()

Clear any data present in the read socket buffer.

close()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

open()

Backend implementations overload this to open a backend connection to the resource.

read(*convert_func=None*)

Receive from the rx socket until *self.rx_buffer_size* samples are received or timeout happens after *self.timeout* seconds.

Optionally, apply the conversion function to the value after it is received.

write(*msg*)

Send a string over the tx socket.

class labbench.**Rack**

Bases: Owner, Ownable

A Rack provides context management and methods for groups of Device instances.

The Rack object provides connection management for all devices and data managers for *with* block:

```
with Rack() as testbed:
    # use the testbed here
    pass
```

For functional validation, it is also possible to open only a subset of devices like this:

```
testbed = Rack()
with testbed.dev1, testbed.dev2:
    # use the testbed.dev1 and testbed.dev2 here
    pass
```

The following syntax creates a new Rack class for an experiment:

import labbench as lb

class MyRack(lb.Rack):
db = lb.SQLiteManager() sa = MySpectrumAnalyzer()

spectrogram = Spectrogram(db=db, sa=sa)

class labbench.**SQLiteLogger**(*path=None*, ***, *append=False*, *text_relational_min=1024*, *force_relational=['host_log']*, *dirname_fmt='{id} {host_time}'*, *nonscalar_file_type='csv'*, *metadata_dirname='metadata'*, *tar=False*, *git_commit_in=None*)

Bases: `RelationalTableLogger`

Store data and property traits to disk into an an sqlite database.

This extends `Aggregator` to support

1. queuing aggregate property trait of devices by lists of dictionaries;
2. custom metadata in each queued aggregate property trait entry; and
3. custom response to non-scalar data (such as relational databasing).

Parameters

- **path** (`str`) – Base path to use for the root database
- **overwrite** (`bool`) – Whether to overwrite the root database if it exists (otherwise, append)
- **text_relational_min** – Text with at least this many characters is stored as a relational text file instead of directly in the database
- **force_relational** – A list of columns that should always be stored as relational data instead of directly in the database
- **nonscalar_file_type** – The data type to use in non-scalar (tabular, vector, etc.) relational data
- **metadata_dirname** – The name of the subdirectory that should be used to store metadata (device connection parameters, etc.)
- **tar** – Whether to store the relational data within directories in a tar file, instead of subdirectories

key(*name*, *attr*)

The key determines the SQL column name. df.to_sql does not seem to support column names that include spaces

open()

Instead of calling *open* directly, consider using *with* statements to guarantee proper disconnection if there is an error. For example, the following sets up a connected instance:

```
with SQLiteLogger('my.db') as db:
    ### do the data acquisition here
    pass
```

would instantiate a *SQLiteLogger* instance, and also guarantee a final attempt to write unwritten data is written, and that the file is closed when exiting the *with* block, even if there is an exception.

class labbench.**Sequence**(**specification*, *shared_names=[]*, *input_table=None*)

Bases: `Ownable`

An experimental procedure defined with methods in Rack instances. The input is a specification for sequencing these steps, including support for threading.

Sequence are meant to be defined as attributes of Rack subclasses in instances of the Rack subclasses.

exception_allowlist

alias of `NeverRaisedException`

return_on_exceptions(*exception_or_exceptions*, *cleanup_func=None*)

Configures calls to the bound Sequence to swallow the specified exceptions raised by constitent steps. If an exception is swallowed, subsequent steps Sequence are not executed. The dictionary of return values from each Step is returned with an additional 'exception' key indicating the type of the exception that occurred.

class labbench.**SerialDevice**(*resource: str = ''*, ***, *timeout: float = 2*, *write_termination: bytes = b'\n'*, *baud_rate: int = 9600*, *parity: bytes = b'N'*, *stopbits: float = 1*, *xonxoff: bool = False*, *rtscts: bool = False*, *dsrdtr: bool = False*)

Bases: *Device*

Base class for wrappers that communicate via pyserial.

This implementation is very sparse because there is in general no messaging string format for serial devices.

- backend

control object, after open

Type
serial.Serial

close()

Disconnect the serial instrument

classmethod from_hwid(*hwid=None*, **args*, ***connection_params*)

Instantiate a new SerialDevice from a windows `hwid' string instead of a comport resource. A hwid string in windows might look something like:

r'PCIVEN_8086&DEV_9D3D&SUBSYS_06DC1028&REV_213&11583659&1&B3'

static list_ports(*hwid=None*)

List USB serial devices on the computer

Returns
list of port resource information

open()

Connect to the serial device with the VISA resource string defined in self.resource

class labbench.**SerialLoggingDevice**(*resource: str = ''*, ***, *timeout: float = 2*, *write_termination: bytes = b'\n'*, *baud_rate: int = 9600*, *parity: bytes = b'N'*, *stopbits: float = 1*, *xonxoff: bool = False*, *rtscts: bool = False*, *dsrdtr: bool = False*, *poll_rate: float = 0.1*, *data_format: bytes = b''*, *stop_timeout: float = 0.5*, *max_queue_size: int = 100000*)

Bases: *SerialDevice*

Manage connection, acquisition, and data retreival on a single GPS device. The goal is to make GPS devices controllable somewhat like instruments: maintaining their own threads, and blocking during setup or stop command execution.

Listener objects must implement an attach method with one argument consisting of the queue that the device manager uses to push data from the serial port.

clear()

Throw away any log data in the buffer.

close()

Disconnect the serial instrument

configure()

This is called at the beginning of the logging thread that runs on a call to *start*.

This is a stub that does nothing — it should be implemented by a subclass for a specific serial logger device.

fetch()

Retrieve and return any log data in the buffer.

Returns
any bytes in the buffer

running()

Check whether the logger is running.

Returns
True if the logger is running

start()

Start a background thread that acquires log data into a queue.

Returns
None

stop()

Stops the logger acquisition if it is running. Returns silently otherwise.

Returns
None

class labbench.**ShellBackend**(*resource: str = ''*, *, *binary_path: Path = None*, *timeout: float = 1*)

Bases: *Device*

Virtual device controlled by a shell command in another process.

Data can be captured from standard output, and standard error pipes, and optionally run as a background thread.

After opening, *backend* is *None*. On a call to run(background=True), *backend* becomes is a subprocess instance. When EOF is reached on the executable's stdout, the backend resets to None.

When *run* is called, the program runs in a subprocess. The output piped to the command line standard output is queued in a background thread. Call read_stdout() to retreive (and clear) this queued stdout.

clear_stdout()

Clear queued standard output. Subsequent calls to *self.read_stdout()* will return ''.

close()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

kill()

If a process is running in the background, kill it. Sends a console warning if no process is running.

open()

The *open()* method implements opening in the *Device* object protocol. Call the `execute()` method when open to execute the binary.

read_stdout(*wait_for=0*)

Pop any standard output that has been queued by a background run (see *run*). Afterward, the queue is cleared. Starting another background run also clears the queue.

Returns
stdout

running()

Check whether a background process is running.

Returns
True if running, otherwise False

write_stdin(*text*)

Write characters to stdin if a background process is running. Raises Exception if no background process is running.

class labbench.**TelnetDevice**(*resource: str = '127.0.0.1:23'*, *, *timeout: float = 2*)

Bases: *Device*

A general base class for communication devices via telnet. Unlike (for example) VISA instruments, there is no standardized command format like SCPI. The implementation is therefore limited to open and close, which open or close a pyserial connection object: the *backend* attribute. Subclasses can read or write with the backend attribute like they would any other telnetlib instance.

A TelnetDevice *resource* string is an IP address. The port is specified by *port*. These can be set when you instantiate the TelnetDevice or by setting them afterward in *value traits*.

Subclassed devices that need property trait descriptors will need to implement get_key and set_key methods to implement the property trait set and get operations (as appropriate).

close()

Disconnect the telnet connection

open()

Open a telnet connection to the host defined by the string in self.resource

labbench.**Undefined**

alias of `_empty`

class labbench.**VISADevice**(*resource: str = ''*, *, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *open_timeout: float = None*, *identity_pattern: str = None*, *timeout: float = None*)

Bases: *Device*

base class for VISA device wrappers with pyvisa.

Examples

Autodetect a list of valid *resource* strings on the host:

```
print(VISADevice.list_resources())
```

Fetch the instrument identity string:

```
with VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR') as instr:
    print(inst.identity)
```

Write ':FETCH?' to the instrument, read an expected ASCII CSV response, and return it as a pandas DataFrame:

```
with VISADevice('USB0::0x2A8D::0x1E01::SG56360004::INSTR') as instr:
    print(inst.query_ascii_values(':FETCH?'))
```

See also: .. _installing a proprietary OS service for VISA:

https://pyvisa.readthedocs.io/en/latest/faq/getting_nivisa.html#faq-getting-nivisa

backend

instance of a pyvisa instrument object (when open)

Type
pyvisa.Resource

close()

closes the instrument.

When managing device connection through a *with* context, this is called automatically and does not need to be invoked.

open()

opens the instrument.

When managing device connection through a *with* context, this is called automatically and does not need to be invoked.

overlap_and_block(*timeout=None*, *quiet=False*)

context manager that sends '*OPC' on entry, and performs a blocking '*OPC?' query on exit.

By convention, these SCPI commands give a hint to the instrument that commands sent inside this block may be executed concurrently. The context exit then blocks until all of the commands have completed.

Example:

```
with inst.overlap_and_block():
    inst.write('long running command 1')
    inst.write('long running command 2')
```

Parameters

- **timeout** – maximum time to wait for '*OPC?' reply, or None to use *self.backend.timeout*
- **quiet** – Suppress timeout exceptions if this evaluates as True

Raises
TimeoutError – on '*OPC?' query timeout

preset()

sends '*RST' to reset the instrument to preset

query(*msg: str*, *timeout=None*) → str

queries the device with an SCPI message and returns its reply.

Handles debug logging and adjustments when in overlap_and_block contexts as appropriate.

Parameters
msg – the SCPI message to send

class suppress_timeout(**exceptions*)

Bases: suppress

context manager that suppresses timeout exceptions on *write* or *query*.

Example:

```
with inst.suppress_timeout():
    inst.write('long command 1')
    inst.write('long command 2')

If the command 1 raises an exception, command 2 will not execute
the context block is complete, and the exception from command 1
is swallowed.
```

wait()

sends '*WAI' to wait for all commands to complete before continuing

write(*msg: str*)

sends an SCPI message to the device.

Wraps *self.backend.write*, and handles debug logging and adjustments when in overlap_and_block contexts as appropriate.

Parameters
msg – the SCPI command to send

Returns
None

class labbench.**Win32ComDevice**(*resource: str = ''*)

Bases: *Device*

Basic support for calling win32 COM APIs.

a dedicated background thread. Set concurrency=True to decide whether this thread support wrapper is applied to the dispatched Win32Com object.

open()

Connect to the win32 com object

labbench.**adjusted**(*trait: Trait | str*, *default: Any*, */*, ***trait_params*) → HasTraits

decorates a Device subclass to adjust parameters of this trait name.

This can be applied to inherited classes that need traits that vary the parameters of a trait defined in a parent. Multiple decorators can be applied to the same class definition.

Parameters

- **trait** (*Union[Trait, str]*) – trait or name of trait to adjust in the wrapped class

- **default** (*Any, optional*) – new default value (for value traits only)

Raises

- **ValueError** – invalid type of Trait argument, or when d
- **TypeError** – _description_
- **ValueError** – _description_

Returns
HasTraits or Device with adjusted trait value

labbench.**concurrently**(**objs*, ***kws*)

If *objs are callable (like functions), call each of
**objs* in concurrent threads. If **objs* are context managers (such as Device instances to be connected), enter each context in concurrent threads.

Multiple references to the same function in *objs* only result in one call. The *catch* and *nones* arguments may be callables, in which case they are executed (and each flag value is treated as defaults).

Parameters

- **objs** – each argument may be a callable (function or class that defines a __call__ method), or context manager (such as a Device instance)
- **catch** – if *False* (the default), a *ConcurrentException* is raised if any of *funcs* raise an exception; otherwise, any remaining successful calls are returned as normal
- **nones** – if not callable and evalues as True, includes entries for calls that return None (default is False)
- **flatten** – if *True*, results of callables that returns a dictionary are merged into the return dictionary with update (instead of passed through as dictionaries)
- **traceback_delay** – if *False*, immediately show traceback information on a thread exception; if *True* (the default), wait until all threads finish

Returns
the values returned by each call

Return type
dictionary keyed by function name

Here are some examples:

Example
Call each function *myfunc1* and *myfunc2*, each with no arguments:

```
>>> def do_something_1 ():
>>>     time.sleep(0.5)
>>>     return 1
>>> def do_something_2 ():
>>>     time.sleep(1)
>>>     return 2
>>> rets = concurrent(myfunc1, myfunc2)
>>> rets[do_something_1]
```

Example
To pass arguments, use the Call wrapper

```
>>> def do_something_3 (a,b,c):
>>>     time.sleep(2)
>>>     return a,b,c
>>> rets = concurrent(myfunc1, Call(myfunc3,a,b,c=c))
>>> rets[do_something_3]
a, b, c
```

Caveats

- Because the calls are in different threads, not different processes, this should be used for IO-bound functions (not CPU-intensive functions).
- Be careful about thread safety.

When the callable object is a Device method, :func concurrency: checks the Device object state.concurrency for compatibility before execution. If this check returns *False*, this method raises a ConcurrentException.

labbench.**find_owned_rack_by_type**(*parent_rack:* Rack, *target_type:* Rack, *include_parent: bool = True*)

return a rack instance of *target_type* owned by *parent_rack*. if there is not exactly 1 for *target_type*, TypeError is raised.

labbench.**import_as_rack**(*import_string: str, *, cls_name: str | None = None, append_path: list = [], base_cls: type = <class 'labbench.Rack'>, replace_attrs: list = ['__doc__', '__module__']*)

Creates a Rack subclass with the specified module's contents. Ownable objects are annotated by type, allowing the resulting class to be instantiated.

Parameters

- **import_string** – for the module that contains the Rack to import
- **cls_name** – the name of the Rack subclass to import from the module (or None to build a new subclass with the module contents)
- **base_cls** – the base class to use for the new subclass
- **append_path** – list of paths to append to sys.path before import
- **replace_attrs** – attributes of *base_cls* to replace from the module

Exceptions:
NameError: if there is an attribute name conflict between the module and base_cls

Returns
A dynamically created subclass of base_cls

labbench.**list_devices**(*depth=1*)

Look for Device instances, and their names, in the calling code context (depth == 1) or its callers (if depth in (2,3,...)). Checks locals() in that context first. If no Device instances are found there, search the first argument of the first function argument, in case this is a method in a class.

labbench.**load_rack**(*output_path: str*, *defaults: dict = {}*, *apply: bool = True*) → *Rack*

instantiates a Rack object from a config directory created by dump_rack.

After instantiation, the current working directory is changed to output_path.

labbench.**observe**(*obj*, *handler*, *name=<class 'labbench._traits.Any'>*, *type_=('get', 'set')*)

Register a handler function to be called whenever a trait changes.

The handler function takes a single message argument. This dictionary message has the keys

- *new*: the updated value
- *old*: the previous value
- *owner*: the object that owns the trait
- *name*: the name of the trait
- 'event': 'set' or 'get'

Parameters

- **handler** – the handler function to call when the value changes
- **names** – notify only changes to these trait names (None to disable filtering)

class labbench.**rack_input_table**(*table_path: str*)

Bases: MethodTaggerDataclass

tag a method defined in a Rack to support execution from a flat table.

In practice, this often means a very long argument list.

Parameters

table_path – location of the input table

class labbench.**rack_kwargs_skip**(**arg_names*)

Bases: MethodTaggerDataclass

tag a method defined in a Rack to replace a **kwargs argument using the signature of the specified callable.

In practice, this often means a very long argument list.

Parameters

- **callable_template** – replace variable keyword arguments (**kwargs) with the keyword arguments defined in this callable
- **skip** – list of column names to omit

class labbench.**rack_kwargs_template**(*template: callable | None = None*)

Bases: MethodTaggerDataclass

tag a method defined in a Rack to replace a **kwargs argument using the signature of the specified callable.

In practice, this often means a very long argument list.

Parameters

- **callable_template** – replace variable keyword arguments (**kwargs) with the keyword arguments defined in this callable
- **skip** – list of column names to omit

labbench.**read**(*path_or_buf*, *columns=None*, *nrows=None*, *format='auto'*, ***kws*)

Read tabular data from a file in one of various formats using pandas.

Parameters

- **path** (str) – path to the data file.
- **columns** – a column or iterable of multiple columns to return from the data file, or None (the default) to return all columns
- **nrows** – number of rows to read at the beginning of the table, or None (the default) to read all rows

- **format** (str) – data file format, one of ['pickle','feather','csv','json','csv'], or 'auto' (the default) to guess from the file extension
- **kws** – additional keyword arguments to pass to the pandas read_<ext> function matching the file extension

Returns

pandas.DataFrame instance containing data read from file

labbench.**read_relational**(*path*, *expand_col*, *root_cols=None*, *target_cols=None*, *root_nrows=None*, *root_format='auto'*, *prepend_column_name=True*)

Flatten a relational database table by loading the table located each row of *root[expand_col]*. The value of each column in this row is copied to the loaded table. The columns in the resulting table generated on each row are downselected according to *root_cols* and *target_cols*. Each of the resulting tables is concatenated and returned.

The expanded dataframe may be very large, making downselecting a practical necessity in some scenarios.

TODO: Support for a list of expand_col?

Parameters

root (*pandas.DataFrame*) – the root database, consisting of columns containing data and columns containing paths to data files expand_col (str): the column in the root database containing paths to data files that should be expanded root_cols: a column (or array-like iterable of multiple columns) listing the root columns to include in the expanded dataframe, or None (the default) pass all columns from *root* target_cols: a column (or array-like iterable of multiple columns) listing the root columns to include in the expanded dataframe, or None (the default) to pass all columns loaded from each root[expand_col] root_path: a string containing the full path to the root database (to help find the relational files) prepend_column_name (bool): whether to prepend the name of the expanded column from the root database

Returns

the expanded dataframe

labbench.**retry**(*exception_or_exceptions*, *tries=4*, *delay=0*, *backoff=0*, *exception_func=<function <lambda>>*)

This decorator causes the function call to repeat, suppressing specified exception(s), until a maximum number of retries has been attempted. - If the function raises the exception the specified number of times, the underlying exception is raised. - Otherwise, return the result of the function call.

Example

The following retries the telnet connection 5 times on ConnectionRefusedError:

```
import telnetlib

# Retry a telnet connection 5 times if the telnet library raises␣
↪ConnectionRefusedError
@retry(ConnectionRefusedError, tries=5)
def open(host, port):
    t = telnetlib.Telnet()
    t.open(host,port,5)
    return t
```

Inspired by https://github.com/saltycrane/retry-decorator which is released under the BSD license.

Parameters

- **exception_or_exceptions** – Exception (sub)class (or tuple of exception classes) to watch for

- **tries** (*int delay: initial delay between retries in seconds*) – number of times to try before giving up

labbench.**sequentially**(**objs*, ***kws*)

If *objs are callable (like functions), call each of

**objs* in the given order. If **objs* are context managers (such as Device instances to be connected), enter each context in the given order, and return a context manager suited for a *with* statement. This is the sequential implementation of the *concurrently* function, with a compatible convention of returning dictionaries.

Multiple references to the same function in *objs* only result in one call. The *nones* argument may be callables in case they are executed (and each flag value is treated as defaults).

Parameters

- **objs** – callables or context managers or Device instances for connections
- **kws** – dictionary of additional callables or Device instances for connections
- **nones** – *True* to include dictionary entries for calls that return None (default: False)
- **flatten** – *True* to flatten any *dict* return values into the return dictionary

Returns

a dictionary keyed on the object name containing the return value of each function

Return type

dictionary of keyed by function

Here are some examples:

Example

Call each function *myfunc1* and *myfunc2*, each with no arguments:

```
>>> def do_something_1 ():
>>>     time.sleep(0.5)
>>>     return 1
>>> def do_something_2 ():
>>>     time.sleep(1)
>>>     return 2
>>> rets = concurrent(myfunc1, myfunc2)
>>> rets[do_something_1]
1
```

Example

To pass arguments, use the Call wrapper

```
>>> def do_something_3 (a,b,c):
>>>     time.sleep(2)
>>>     return a,b,c
>>> rets = concurrent(myfunc1, Call(myfunc3,a,b,c=c))
>>> rets[do_something_3]
a, b, c
```

Caveats

- Unlike *concurrently*, an exception in a context manager's __enter__ means that any remaining context managers will not be entered.

When the callable object is a Device method, :func concurrency: checks the Device object state.concurrency for compatibility before execution. If this check returns *False*, this method raises a ConcurrentException.

labbench.**show_messages**(*minimum_level*, *colors=True*)

filters logging messages displayed to the console by importance

Parameters
minimum_level – 'debug', 'warning', 'error', or None (to disable all output)

Returns
None

labbench.**sleep**(*seconds*, *tick=1.0*)

Drop-in replacement for time.sleep that raises ConcurrentException if another thread requests that all threads stop.

labbench.**stopwatch**(*desc: str = ''*, *threshold: float = 0*)

Time a block of code using a with statement like this:

```
>>> with stopwatch('sleep statement'):
>>>     time.sleep(2)
sleep statement time elapsed 1.999s.
```

Parameters

- **desc** – text for display that describes the event being timed
- **threshold** – only show timing if at least this much time (in s) elapsed

: :returns: context manager

labbench.**timeout_iter**(*duration*)

sets a timer for *duration* seconds, yields time elapsed as long as timeout has not been reached

labbench.**trait_info**(*device:* Device, *name: str*) → dict

returns the keywords used to define the trait attribute named *name* in *device*

labbench.**unobserve**(*obj*, *handler*)

Unregister a handler function from notifications in obj.

labbench.**until_timeout**(*exception_or_exceptions*, *timeout*, *delay=0*, *backoff=0*, *exception_func=<function <lambda>>*)

This decorator causes the function call to repeat, suppressing specified exception(s), until the specified timeout period has expired. - If the timeout expires, the underlying exception is raised. - Otherwise, return the result of the function call.

Inspired by https://github.com/saltycrane/retry-decorator which is released under the BSD license.

Example

The following retries the telnet connection for 5 seconds on ConnectionRefusedError:

```
import telnetlib

@until_timeout(ConnectionRefusedError, 5)
def open(host, port):
    t = telnetlib.Telnet()
    t.open(host,port,5)
    return t
```

Parameters

- **exception_or_exceptions** – Exception (sub)class (or tuple of exception classes) to watch for
- **timeout** (*float delay: initial delay between retries in seconds*) – time in seconds to continue calling the decorated function while suppressing exception_or_exceptions

labbench.**visa_list_resources**(*resourcemanager: str | None = None*)

autodetects and returns a list of valid resource strings

5.2 labbench.datareturn

class labbench.datareturn.**NetworkAddress**(**args*, ***kws*)

Bases: *NetworkAddress*

class labbench.datareturn.**Path**(**args*, ***kws*)

Bases: *Path*

must_exist: *bool*

does the path need to exist when set?

class labbench.datareturn.**bool**(**args*, ***kws*)

Bases: *bool*

class labbench.datareturn.**bytes**(**args*, ***kws*)

Bases: *bytes*

class labbench.datareturn.**complex**(**args*, ***kws*)

Bases: *complex*

class labbench.datareturn.**dict**(**args*, ***kws*)

Bases: *dict*

class labbench.datareturn.**float**(**args*, ***kws*)

Bases: *float*

class labbench.datareturn.**int**(**args*, ***kws*)

Bases: *int*

class labbench.datareturn.**list**(**args*, ***kws*)

Bases: *list*

class labbench.datareturn.**str**(**args*, ***kws*)

Bases: *str*

class labbench.datareturn.**tuple**(**args*, ***kws*)

Bases: *tuple*

5.3 labbench.property

class labbench.property.**NetworkAddress**(**args*, ***kws*)

Bases: *NetworkAddress*

class labbench.property.**Path**(**args*, ***kws*)

Bases: *Path*

must_exist: *bool*

does the path need to exist when set?

class labbench.property.**bool**(**args*, ***kws*)

Bases: *bool*

class labbench.property.**bytes**(**args*, ***kws*)

Bases: *bytes*

class labbench.property.**complex**(**args*, ***kws*)

Bases: *complex*

class labbench.property.**dict**(**args*, ***kws*)

Bases: *dict*

class labbench.property.**float**(**args*, ***kws*)

Bases: *float*

class labbench.property.**int**(**args*, ***kws*)

Bases: *int*

class labbench.property.**list**(**args*, ***kws*)

Bases: *list*

class labbench.property.**message_keying**(**args*, ***kws*)

Bases: PropertyKeyingBase

Device class decorator that implements automatic API that triggers API messages for labbench properties.

Example usage:

```python
import labbench as lb

@lb.message_keying(query_fmt='{key}?', write_fmt='{key} {value}', query_func='get', write_func='set') class MyDevice(lb.Device):

    def set(self, set_msg: str):
        # do set pass

    def get(self, get_msg: str):
        # do get pass
```

Decorated classes connect traits that are defined with the *key* keyword to trigger backend API calls based on the key. The implementation of the *set* and *get* methods in subclasses of MessagePropertyAdapter determines how the key is used to generate API calls.

get(*device: HasTraits*, *scpi_key:* str, *trait=None*)

queries a parameter named *scpi_key* by sending an SCPI message string.

The command message string is formatted as f'{scpi_key}?'. This is automatically called in wrapper objects on accesses to property traits that defined with 'key=' (which then also cast to a pythonic type).

Parameters

- **key** (str) – the name of the parameter to set
- **name** (str, *None*) – name of the trait setting the key (or None to indicate no trait) (ignored)

Returns

response (str)

set(*device: HasTraits*, *scpi_key:* str, *value*, *trait=None*)

writes an SCPI message to set a parameter with a name key to *value*.

The command message string is formatted as f'{scpi_key} {value}'. This This is automatically called on assignment to property traits that are defined with 'key='.

Parameters

- **scpi_key** (str) – the name of the parameter to set
- **value** (str) – value to assign
- **name** (str, *None*) – name of the trait setting the key (or None to indicate no trait) (ignored)

class labbench.property.**str**(**args*, ***kws*)

Bases: *str*

class labbench.property.**tuple**(**args*, ***kws*)

Bases: *tuple*

class labbench.property.**visa_keying**(**args*, ***kws*)

Bases: *message_keying*

Device class decorator that automates SCPI command string interactions for labbench properties.

Example usage:

```python
import labbench as lb

@lb.property.visa_keying(query_fmt='{key}?', write_fmt='{key} {value}')
class MyDevice(lb.VISADevice):
    pass
```

This causes access to property traits defined with 'key=' to interact with the VISA instrument. By default, messages in VISADevice objects trigger queries with the *'{key}?'* format, and writes formatted as f'{key} {value}'.

5.4 labbench.value

class labbench.value.**NetworkAddress**(**args*, ***kws*)

Bases: *NetworkAddress*

class labbench.value.**Path**(**args*, ***kws*)

Bases: *Path*

must_exist: *bool*

does the path need to exist when set?

class labbench.value.**any**(**args*, ***kws*)

Bases: *any*

class labbench.value.**bool**(**args*, ***kws*)

Bases: *bool*

class labbench.value.**bytes**(**args*, ***kws*)

Bases: *bytes*

class labbench.value.**complex**(**args*, ***kws*)

Bases: *complex*

class labbench.value.**dict**(**args*, ***kws*)

Bases: *dict*

class labbench.value.**float**(**args*, ***kws*)

Bases: *float*

class labbench.value.**int**(**args*, ***kws*)

Bases: *int*

class labbench.value.**list**(**args*, ***kws*)

Bases: *list*

class labbench.value.**str**(**args*, ***kws*)

Bases: *str*

class labbench.value.**tuple**(**args*, ***kws*)

Bases: *tuple*