VISA Instrument Control with Python

This page covers controlling bench instruments — oscilloscopes, function generators, power supplies, multimeters — from Python over the **VISA** protocol using the `pyvisa` package.

VISA (Virtual Instrument Software Architecture) is a standard for talking to test equipment over USB, Ethernet, or GPIB. Learn the pattern once and it transfers to almost any modern bench instrument.

> **Prerequisites**
>
> The NI-VISA drivers (from National Instruments) and the Python package (`pip install pyvisa`). For GPIB instruments, also install the **NI-488.2** driver — see GPIB Instruments below.

> **Note**
>
> The SCPI command strings below are written for specific instrument models (noted in each section). The *pattern* is universal; the exact commands vary by model. Consult your instrument's programming guide to adapt them.

1 Finding Your Instruments

List connected resources and ask each to identify itself with `*IDN?` (a near-universal command):

```
import pyvisa

rm = pyvisa.ResourceManager()
for resource in rm.list_resources():
    try:
        inst = rm.open_resource(resource)
        inst.timeout = 2000
        print(f"{resource} -> {inst.query('*IDN?').strip()}")
        inst.close()
    except Exception as e:
        print(f"{resource} -> error: {e}")
```

A USB resource string looks like `USB0::0xVENDOR::0xPRODUCT::SERIAL::0::INSTR` . Vendor IDs you'll see often: Tektronix `0x0699` , Keysight `0x2A8D` , Keithley `0x05E6` .

2 The Basic Pattern

Every VISA interaction is some combination of three moves: **open**, **write/query**, **close**.

```
import pyvisa

rm = pyvisa.ResourceManager()
inst = rm.open_resource("USB0::...::INSTR")
inst.timeout = 5000

try:
```

```
    print(inst.query("*IDN?"))      # query: send a command and read the reply
    inst.write("*RST")              # write: send a command, no reply expected
    value = inst.query("MEASURE:VOLTAGE?")
finally:
    inst.close()
```

- `write()` — send a command, expect no response
- `query()` — send a command and read the response
- `read()` — read a response after a separate write

Resetting with `*RST` at the start puts the instrument in a known state.

3 GPIB Instruments

GPIB (IEEE-488) is just another VISA transport, so everything above applies — the only differences are the driver and the resource-string format.

Driver: GPIB instruments are reached through an NI GPIB controller, which needs the **NI-488.2** driver in addition to NI-VISA. No extra Python package — `pyvisa` talks to GPIB the same way it talks to USB.

Addressing: a GPIB resource string is `GPIB<board>::<primary_address>::INSTR`, where `<board>` is the controller index (usually `0` if you have one GPIB interface) and `<primary_address>` is the instrument's GPIB address (0–30), set on its front panel. So an instrument at address 13 on the first controller is `GPIB0::13::INSTR`:

```
import pyvisa

rm = pyvisa.ResourceManager()
inst = rm.open_resource("GPIB0::13::INSTR")
inst.timeout = 5000

try:
    print(inst.query("*IDN?"))
finally:
    inst.close()
```

`rm.list_resources()` discovers connected GPIB instruments alongside USB ones, so the Finding Your Instruments loop works unchanged. Once open, an instrument behaves identically regardless of transport: the function generator, power supply, and oscilloscope examples below apply whether the instrument is on USB or GPIB.

> See also
>
> When two instruments share a GPIB bus and you want several scripts or threads to reach them at once, you need to arbitrate access so commands don't collide. That's covered in Sharing an Instrument Across Processes.

4 Function Generator

Commands below are for a **Keysight EDU33212A**; adapt for your model.

```python
import pyvisa

rm = pyvisa.ResourceManager()
fgen = rm.open_resource("USB0::0x2A8D::...::INSTR")

try:
    fgen.write("*RST")
    fgen.write("OUTPUT1:LOAD INF")       # high-Z load
    fgen.write("APPLY:SIN 1000,2,0")     # sine: 1 kHz, 2 Vpp, 0 V offset
    fgen.write("OUTPUT1 ON")
finally:
    fgen.close()
```

Other waveforms follow the same form: `APPLY:SQU` (square), `APPLY:TRI` (triangle), `APPLY:RAMP` (sawtooth), `APPLY:DC DEF,DEF,2.5` (2.5 V DC).

4.1 Sweeping Frequency

Once the output is configured, changing only the frequency is a single command — which makes a sweep a plain loop. This is the driving half of any frequency-response measurement:

```python
import numpy as np
import time

fgen.write("APPLY:SIN 100,2,0")     # configure once: 2 Vpp sine
fgen.write("OUTPUT1 ON")

for f in np.logspace(2, 5, 31):     # 100 Hz to 100 kHz, 31 points, log-spaced
    fgen.write(f"FREQUENCY {f}")
    time.sleep(0.2)                 # let the output and the DUT settle
    # measure here — scope, DAQ, or DMM
```

Use `np.logspace` rather than `np.linspace` when the range spans decades, so the points land evenly on the log axis you'll eventually plot them on. The settle delay matters more than it looks: read too soon and you measure the previous frequency.

> **`OUTPUT1:LOAD INF` changes the numbers, not just the setting**
>
> A function generator assumes a 50 Ω load unless told otherwise, and its displayed amplitude is what it expects to deliver *into that load*. Drive a high-impedance input with the default setting and you get twice the amplitude you asked for. Set the load to match reality — `INF` for a scope input, `50` for a 50 Ω termination — or your amplitudes are wrong by a factor of two throughout.

5 Power Supply

Commands below are for a **Keysight EDU36311A**; adapt for your model.

```python
import pyvisa

rm = pyvisa.ResourceManager()
```

```
psu = rm.open_resource("USB0::0x2A8D::...::INSTR")

try:
    psu.write("*RST")
    psu.write("INSTRUMENT:SELECT CH1")
    psu.write("VOLTAGE 5.0")     # 5 V
    psu.write("CURRENT 0.1")     # 100 mA limit
    psu.write("OUTPUT ON")
finally:
    psu.close()
```

A controlled ramp is just a loop that sets voltage, waits to settle, and repeats — useful for I–V sweeps and controlled-input experiments:

```
import numpy as np
import time

for v in np.arange(0, 5.001, 0.5):
    psu.write(f"VOLTAGE {v}")
    time.sleep(0.5)             # settle
    # read a sensor here (DAQ, scope, DMM) to build a curve
```

6 Source-Measure Units and Compliance Limits

A source-measure unit (SMU) sources one quantity and measures the other on the same pair of terminals — source a voltage and it reports the current that flowed, or source a current and it reports the developed voltage. That saves you wiring a supply and a meter together, and it is why an SMU sweep is a handful of lines.

Commands below are for a **Keithley 2450**; adapt for your model.

```
import numpy as np
import pyvisa

rm = pyvisa.ResourceManager()
smu = rm.open_resource("USB0::0x05E6::0x2450::...::INSTR")   # placeholder address

try:
    smu.write("*RST")
    smu.write("SOUR:FUNC VOLT")           # source voltage
    smu.write("SOUR:VOLT:RANG 2")         # source range, volts
    smu.write("SOUR:VOLT:ILIM 0.05")      # compliance: never exceed 50 mA
    smu.write('SENS:FUNC "CURR"')         # measure current — note the inner quotes
    smu.write("SENS:CURR:RANG 0.1")       # fixed range beats autorange for speed
    smu.write("OUTP ON")

    for v in np.arange(0, 1.001, 0.05):
        smu.write(f"SOUR:VOLT {v}")
        current = float(smu.query("MEAS:CURR?"))
        print(f"{v:.2f} V  {current*1e3:+.4f} mA")
finally:
    smu.write("OUTP OFF")                 # output off before anything else
    smu.close()
```

Set the compliance limit before the output goes on

Compliance is the ceiling on the quantity you are *not* sourcing — the most current the SMU will push while holding a commanded voltage. It is a safety limit for your sample, not a formality.

An SMU sourcing voltage will deliver whatever current the load draws. Into a device whose current rises exponentially with voltage — any diode, LED, or junction — a fraction of a volt past the knee is the difference between a microamp and destruction. The instrument does exactly what you asked; the sample is the thing that is gone.

Set the source's current limit — `SOUR:VOLT:ILIM` on this model, and note that older Keithley instruments spell it `SENS:CURR:PROT` — to a value your sample survives *before* enabling the output, and start any sweep from zero rather than from wherever the last run left it.

Two more habits worth building in, both visible in the example above:

- **Turn the output off in a `finally` block.** An exception mid-sweep otherwise leaves the source energised at whatever it last reached, with nobody watching. Same reasoning as closing the resource — and more urgent, because a leaked handle costs you a restart while a live output costs you a sample.
- **Autorange costs time.** Letting the SMU re-range on every point is convenient and slow, since each change means a settling delay. If the current spans decades you usually want it; if you know the range, fix it and the sweep speeds up considerably.
- **Use four-wire sensing for small resistances.** With two wires you measure your leads and contacts along with the sample, which is invisible at kilohms and dominant at milliohms. `SENS:CURR:RSEN ON` switches the 2450 to remote sensing, which needs the extra pair of leads connected but removes the lead resistance from the answer.

7 Lock-In Amplifiers

A lock-in recovers a small signal at a known modulation frequency out of noise that dwarfs it — you chop or modulate the thing you are measuring, hand the lock-in a reference at that frequency, and it reports the component of the input that matches. If you have a physical lock-in on the bench, use it: the automation job is then just reading it in a loop.

Commands below are for an **SRS SR510**; adapt for your model. Unlike the other instruments on this page it speaks **RS-232 rather than USB or GPIB**, so the resource is a serial port and the line settings have to match the instrument: 19200 baud, odd parity, two stop bits, carriage-return terminated.

```
import time

import numpy as np
import pyvisa

rm = pyvisa.ResourceManager()
lia = rm.open_resource("ASRL3::INSTR")       # placeholder — your COM port
lia.baud_rate = 19200
lia.parity = pyvisa.constants.Parity.odd
lia.stop_bits = pyvisa.constants.StopBits.two
lia.read_termination = "\r"
lia.write_termination = "\r"
```

```
lia.write("G 18")                # sensitivity, code 1–24 (18 = 5 mV)
lia.write("T 1,5")               # pre time constant  (5 = 100 ms)
lia.write("T 2,1")               # post time constant (1 = 0.1 s)
time.sleep(5)                    # let the filters settle after changing them

results = []
for setpoint in np.linspace(0, 10, 21):
    move_or_set(setpoint)        # your stage, wavelength, field, ...
    time.sleep(1.0)              # several time constants — see below
    results.append((setpoint, float(lia.query("Q"))))
```

`Q` returns the output; `P` sets the phase in degrees; `G` with no argument reads the sensitivity code back. The sensitivity codes run 1–24 in a 1-2-5 sequence from 10 nV to 500 mV, so code 18 is 5 mV.

Three things worth knowing before you automate one.

The SR510 is single-phase, so the phase adjustment is not optional. It has one output — the component of the input in phase with the reference, proportional to cos(θ). Set the phase wrong and you read a fraction of your signal, or zero, with no indication anything is amiss. This is the opposite of a **dual-phase** instrument (the SR830 and SR860, among others), which reports X and Y and computes $R = \sqrt{X^2 + Y^2}$; *that* is phase-independent, and with one of those you can skip the adjustment entirely. With an SR510 you cannot.

Find the phase by **nulling rather than maximising**. The output varies as cos(θ), which is flat near its peak and steepest at the null — so the null locates the phase far more precisely. Sweep for the minimum, then add 90°:

```
import numpy as np

phases = np.arange(0, 180, 2.0)
readings = []
for p in phases:
    lia.write(f"P {p}")
    time.sleep(1.0)                      # settle after every phase change
    readings.append(abs(float(lia.query("Q"))))

null = phases[int(np.argmin(readings))]
lia.write(f"P {(null + 90) % 360}")      # 90° from the null is the signal
```

The time constant is just how long it averages, and the response is what bites in an automated sweep. Change a setpoint and read too soon and you record the tail of the previous point. A sweep whose settle delay is shorter than its time constant produces data that looks smooth and plausible with every feature smeared toward the one before it — wait at least 5 time constants after *any* change, including to the sensitivity or the time constant itself.

It measures the fundamental, not your whole waveform. The SR510 detects against an internal sine reference, so a square-wave signal — which is what an optical chopper gives you — contributes only its fundamental Fourier component. The reading is therefore not the peak amplitude of the chopped signal, and converting between them takes a correction factor that depends on the waveform. Determine yours by measuring a signal of known amplitude rather than by assuming one.

No lock-in free? You can do it in software

If every box is spoken for, or you are already digitizing the signal and the reference on two channels, the same detection can be done numerically: multiply the signal by the reference, average, and only the component at the reference frequency survives. `scipy.signal.hilbert` generates the 90°-shifted copy needed for the quadrature channel.

The one thing to get right is the **scale factor**, and it depends on the shape of your reference. Normalizing the reference to unit RMS and then taking `2 * mean(signal * reference)` recovers the true amplitude for a *square* (TTL) reference — but over-reports by √2 for a sine reference, and by 4/π for a sine signal against a square reference. Verify yours against a synthetic signal of known amplitude before trusting it, exactly as in Validating a Measurement Chain.

8 Oscilloscope

Commands below are for a **Tektronix TBS2000-series** scope; adapt for your model.

8.1 Set Up the Channel

A scope reading is only as good as its vertical scale and trigger. Configure both before measuring, rather than relying on whatever the last person left:

```
scope.write("*RST")                      # known state
scope.write("SELECT:CH1 ON")
scope.write("CH1:COUPLING DC")           # AC to reject a DC offset
scope.write("CH1:PROBE 1")               # match your probe: 1 or 10
scope.write("CH1:SCALE 0.5")             # volts per division
scope.write("HORIZONTAL:SCALE 1E-3")     # seconds per division

scope.write("TRIGGER:A:EDGE:SOURCE CH1")
scope.write("TRIGGER:A:EDGE:SLOPE RISE")
scope.write("TRIGGER:A:LEVEL 0.0")
```

Two settings cause most bad captures. **`CH1:PROBE`** must match the probe you actually attached — a 10× probe read as 1× reports a tenth of the real voltage, with nothing on screen to suggest anything is wrong. And **vertical scale** sets resolution: the digitiser covers about eight divisions, so a signal occupying one division is being measured with a fraction of the available bits. Fill the screen.

If you'd rather let the scope choose, `scope.write("AUTOSET EXECUTE")` followed by a couple of seconds' wait is a reasonable start — but it re-scales between measurements, so don't use it inside a loop where you're comparing amplitudes.

8.2 Read a Measurement

```
import pyvisa
import time

rm = pyvisa.ResourceManager()
scope = rm.open_resource("USB0::0x0699::...::INSTR")
```

```
scope.timeout = 10000

try:
    scope.write("MEASUREMENT:MEAS1:SOURCE CH1")
    scope.write("MEASUREMENT:MEAS1:TYPE PK2PK")
    scope.write("MEASUREMENT:MEAS1:STATE ON")   # must enable the measurement
    time.sleep(1)
    vpp = float(scope.query("MEASUREMENT:MEAS1:VALUE?"))
    print("no signal" if vpp > 1e30 else f"Vpp = {vpp:.3f} V")
finally:
    scope.close()
```

A returned value near `9.9e37` is a “no valid measurement” sentinel — check the signal and trigger.

8.3 Capture the Raw Waveform

The scope returns the waveform as raw bytes plus scaling factors you apply to recover real time and voltage:

```
import numpy as np

def capture_waveform(scope, channel="CH1"):
    scope.write("header 0")
    scope.write("data:encdg RIBINARY")
    scope.write(f"data:source {channel}")
    scope.write("data:start 1")
    n = int(scope.query("wfmpre:nr_pt?"))
    scope.write(f"data:stop {n}")
    scope.write("wfmpre:byt_nr 1")

    raw = scope.query_binary_values("curve?", datatype='b', container=np.array)
    t_scale = float(scope.query("wfmpre:xincr?"))
    v_scale = float(scope.query("wfmpre:ymult?"))
    v_off   = float(scope.query("wfmpre:yoff?"))

    t = np.arange(n) * t_scale
    v = (raw - v_off) * v_scale
    return t, v
```

The raw bytes are 8-bit signed digitiser counts, not volts — `ymult` and `yoff` are what convert them. Skip that step and you get a plausible-looking waveform with meaningless amplitudes.

Other measurement types follow the same pattern as `PK2PK`: `MEAN`, `RMS`, `FREQUENCY`, `PERIOD`, `AMPLITUDE`, `MAXIMUM`, `MINIMUM`. A scope typically supports several simultaneous measurement slots (`MEAS1` through `MEAS4`), so you can enable a few once and query them together each iteration.

8.4 Save a Screenshot

Sometimes the useful record is the screen itself — a trigger setup, an anomaly worth showing someone:

```
def save_screenshot(scope, filename="scope.png"):
    scope.write("SAVE:IMAGE:FILEFORMAT PNG")
```

```
    scope.write("HARDCOPY START")
    image = scope.read_raw()
    with open(filename, "wb") as f:
        f.write(image)
```

`read_raw()` rather than `read()` because the reply is binary. Give the scope a generous `timeout` — a screen image is far larger than a normal query reply and the default will cut it off mid-transfer.

9 Multi-Instrument Automation: Frequency Response

Coordinating two instruments in a loop is the heart of many automated measurements. This example measures a **frequency response** (Bode magnitude plot): drive a device under test with the function generator while reading the output amplitude on the scope, stepping through frequencies.

```
+--------------+     +--------------+     +--------------+
|   Function   |---->| Device Under |---->| Oscilloscope |
|   Generator  |     |  Test (DUT)  |     |    (CH1)     |
+--------------+     +--------------+     +--------------+
```

```
import pyvisa
import time
import numpy as np

def frequency_response(fgen_addr, scope_addr, frequencies, amplitude_vpp=1.0):
    rm = pyvisa.ResourceManager()
    fgen = rm.open_resource(fgen_addr)
    scope = rm.open_resource(scope_addr)
    scope.timeout = 10000
    amplitudes = []

    try:
        # Function generator: start with a known signal
        fgen.write("*RST")
        fgen.write("OUTPUT1:LOAD INF")
        fgen.write(f"APPLY:SIN 1000,{amplitude_vpp},0")
        fgen.write("OUTPUT1 ON")

        # Scope: auto-scale, then enable a peak-to-peak measurement
        scope.write("AUTOSET EXECUTE")
        time.sleep(3)
        scope.write("MEASUREMENT:MEAS1:SOURCE CH1")
        scope.write("MEASUREMENT:MEAS1:TYPE PK2PK")
        scope.write("MEASUREMENT:MEAS1:STATE ON")
        time.sleep(1)

        for freq in frequencies:
            fgen.write(f"FREQUENCY {freq}")
            scope.write(f"HORIZONTAL:SCALE {0.25 / freq}")  # ~2-3 cycles on screen
            time.sleep(max(0.5, 10.0 / freq))               # settle (≥10 cycles)
            vpp = float(scope.query("MEASUREMENT:MEAS1:VALUE?"))
            amplitudes.append(np.nan if vpp > 1e30 else vpp)
    finally:
        fgen.write("OUTPUT1 OFF")
```

```
        fgen.close(); scope.close(); rm.close()

    return np.array(frequencies), np.array(amplitudes)


def plot_bode(frequencies, amplitudes, reference_amplitude=None):
    import matplotlib.pyplot as plt
    ref = reference_amplitude or amplitudes[0]
    gain_db = 20 * np.log10(amplitudes / ref)

    plt.figure(figsize=(10, 6))
    plt.semilogx(frequencies, gain_db, 'b-o', markersize=4)
    plt.axhline(-3, color='r', linestyle='--', label='-3 dB (cutoff)')
    plt.xlabel("Frequency (Hz)"); plt.ylabel("Gain (dB)")
    plt.grid(True, which="both", alpha=0.7); plt.legend()
    plt.tight_layout(); plt.show()

# freqs = np.logspace(2, 5, 20)   # 100 Hz – 100 kHz, 20 points
# freqs, amps = frequency_response(FGEN_ADDR, SCOPE_ADDR, freqs)
# plot_bode(freqs, amps, reference_amplitude=1.0)
```

On the resulting Bode plot, the flat region is the passband, the roll-off marks the bandwidth limit, and the **−3 dB** point is the cutoff frequency. The same set-wait-measure-record loop adapts to other measurements — phase response, distortion vs. amplitude, rise time vs. frequency.

10 Troubleshooting

No instruments found — check USB connections; verify NI-VISA is installed; try reconnecting the instrument.

Timeout (`VI_ERROR_TMO`) — increase `inst.timeout`; confirm the instrument responds from its front panel; some commands need a delay after `write`.

Garbled responses — set `inst.read_termination = '\n'`; try `inst.encoding = 'latin_1'`; clear errors with `inst.write('*CLS')`.

Instrument not responding — another program may hold the resource; the instrument may be in local mode (press its “Local” button); try `inst.write('*RST')`.

Back to Python resources