Instrument Wrapper Classes

The VISA Instrument Control page shows the adapt-this-snippet way to talk to an instrument: open a resource, write and query SCPI strings, close it. That's the right tool for a one-off measurement. Once you're reusing the same instrument across scripts — or building an acquisition app — those loose calls become a liability: the connection logic, error handling, and device-specific commands get copy-pasted and drift out of sync.

A **wrapper class** fixes that. You write the connection and communication logic once, give the instrument clean Python methods, and every script that imports it gets the same reliable interface. This is the first step from "scripting an instrument" to "building lab software," and the foundation the rest of these building-lab-software guides build on.

> Prerequisites
>
> VISA Instrument Control and comfort with Python classes (`__init__` , methods, inheritance).

1 A Base `Device` Class

Start with a class that wraps the open/write/query/close cycle and handles the one error that bites every instrument program — a communication timeout — without crashing the whole script.

```python
import pyvisa


class Device:
    """A thin wrapper around a single VISA instrument.

    Works for any VISA resource (USB, Ethernet, or GPIB). Pass a full resource
    string, e.g. "GPIB0::13::INSTR" or "USB0::0x2A8D::...::INSTR".
    """

    def __init__(self, resource: str, timeout: int = 5000):
        self.resource = resource
        self.rm = pyvisa.ResourceManager()
        self.dev = self.rm.open_resource(resource)
        self.dev.timeout = timeout

    def write(self, msg: str) -> str:
        """Send a command; expect no reply. Returns "sent" or "timed out"."""
        try:
            self.dev.write(msg)
            return "sent"
        except pyvisa.errors.VisaIOError:
            return "timed out"

    def read(self) -> str:
        """Read a pending response."""
        try:
            return self.dev.read()
        except pyvisa.errors.VisaIOError:
            return "timed out"
```

```
    def query(self, msg: str) -> str:
        """Send a command and read the reply (write + read in one call)."""
        try:
            return self.dev.query(msg)
        except pyvisa.errors.VisaIOError:
            return "timed out"

    def query_values(self, msg: str):
        """Query a comma-separated numeric block (e.g. a scope trace)."""
        try:
            return self.dev.query_ascii_values(msg)
        except pyvisa.errors.VisaIOError:
            return "timed out"

    def get_id(self) -> str:
        """Return the instrument's *IDN? identity string."""
        return self.query("*IDN?")

    def close(self):
        self.dev.close()
        self.rm.close()
```

Three things this buys you over raw `pyvisa` calls:

- **One place for connection logic.** Resource string, timeout, and resource-manager lifecycle live in `__init__` / `close`, not scattered across every script.
- **Timeouts don't crash.** Each method catches `VisaIOError` and returns `"timed out"` so a slow instrument doesn't kill a long acquisition — the caller decides what to do about it.
- **A clean vocabulary.** `dev.query("*IDN?")` reads better than the resource-manager dance, and it's identical whether the instrument is on USB or GPIB.

1.1 Use it as a context manager (optional but tidy)

Adding two methods lets you use `with`, which guarantees the instrument closes even if an error is raised:

```
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        self.close()
```

```
with Device("GPIB0::13::INSTR") as dev:
    print(dev.get_id())
# automatically closed here
```

2 Subclassing for a Specific Instrument

The base class speaks raw SCPI. The payoff comes from **subclassing** it to give a particular instrument methods that speak physics, not protocol. Below, a temperature controller gets `temperature()` and `set_setpoint()` methods — callers never see the SCPI strings.

```
class TemperatureController(Device):
    """Example subclass for a bench temperature controller.

    Command strings here are illustrative — replace them with the SCPI from
    your instrument's programming guide.
    """

    def __init__(self, resource: str, timeout: int = 5000):
        super().__init__(resource, timeout)
        # Cache values that rarely change so we don't re-query every call.
        self.setpoint = float(self.query("SETP?"))

    def temperature(self, channel: str = "A") -> float:
        """Read the temperature on a sensor channel, in kelvin."""
        reply = self.query(f"KRDG? {channel}")
        return float(reply)

    def set_setpoint(self, kelvin: float):
        """Command a new setpoint and update the cached value."""
        self.write(f"SETP {kelvin}")
        self.setpoint = kelvin
```

```
with TemperatureController("GPIB0::12::INSTR") as tc:
    print(f"Current: {tc.temperature('A'):.3f} K")
    tc.set_setpoint(4.2)
```

Why this is worth the extra file:

- **Callers think in physics.** `tc.temperature("A")` instead of parsing `KRDG? A`. The SCPI lives in one place; if a command changes, you fix it once.
- **Swappable hardware.** If you replace the controller with a different model, you rewrite the subclass to keep the same method names — and every script that uses `tc.temperature()` keeps working unchanged. (This “same interface, different innards” idea is the controller layer in Architecting a Lab App.)
- **A natural home for caching and validation.** Values that change only when *you* change them (a setpoint) can be cached; values the instrument changes on its own (a live temperature) should always be re-queried. The subclass is where that judgment lives.

3 Where This Leads

You now have reusable, well-behaved instrument objects. The rest of the guides build on them:

- **Sharing an Instrument Across Processes** — when several scripts or threads need the same instrument at once, put a server in front of it so requests don’t collide.

- **Structuring a Data-Acquisition App** — combine instrument objects into a program that records data to disk.
- **Building a GUI** — drive these objects from a live interface.

Back to Python for Lab Work