Structuring a Data-Acquisition App

A quick acquisition script — open an instrument, loop, `print` the readings — is fine for a one-off. But once you're collecting data you'll actually analyze, you need it written to disk in a documented format, with metadata, while the program stays responsive. This guide shows how to structure that: a small file-writing class, an experiment class that owns the acquisition loop, and a way to run that loop in the background while you stay in control.

The goal isn't a framework — it's a handful of patterns that keep a growing acquisition program from turning into spaghetti.

> **Prerequisites**
>
> Instrument Wrapper Classes. If your setup needs concurrent bus access, also see Sharing an Instrument Across Processes.

1 A Reusable Data-File Class

Every acquisition writes data to a file, usually CSV with a few comment lines of metadata at the top. Factor that into a base class so you write it once:

```python
import os
import time


class CSVFile:
    """Creates (or appends to) a CSV file and writes comment lines and rows."""

    def __init__(self, path: str, name: str):
        os.makedirs(path, exist_ok=True)
        if not name.endswith(".csv"):
            name += ".csv"
        self.filename = os.path.join(path, name)
        self.is_new = not os.path.exists(self.filename)
        if self.is_new:
            self.write_comment(f"Created {time.ctime()}")

    def write_comment(self, text: str):
        """Write a metadata line, prefixed with # so analysis code can skip it."""
        with open(self.filename, "a") as f:
            f.write(f"# {text}\n")

    def write_row(self, values: list):
        """Append one comma-delimited row."""
        with open(self.filename, "a") as f:
            f.write(",".join(str(v) for v in values) + "\n")
```

Two habits worth building in from the start:

- **Metadata in comment lines.** Record date, instrument settings, sample ID — anything future-you needs to interpret the data. Prefixing with `#` means `np.loadtxt(..., comments='#')` and `pandas.read_csv(..., comment='#')` skip it automatically.

- **Append, don't overwrite.** Opening in append mode (`"a"`) and only writing the header for a new file means a crashed-and-restarted run adds to the data instead of destroying it.

2 An Experiment Class

Subclass the file writer into something that knows your specific measurement: which instruments it reads, what the columns are, and how to take one data point. Keeping a `running` flag lets an outside caller stop the loop cleanly.

```
from instruments import TemperatureController, VoltageSupply   # your wrappers


class ResistanceVsTemp(CSVFile):
    """Example: log resistance against temperature until told to stop."""

    columns = ["time_s", "temperature_K", "resistance_ohm"]

    def __init__(self, path, name, tc_resource, vs_resource):
        super().__init__(path, name)
        self.tc = TemperatureController(tc_resource)
        self.vs = VoltageSupply(vs_resource)
        self.running = False
        if self.is_new:
            self.write_row(self.columns)          # header row for a fresh file

    def take_point(self) -> list:
        """Read every instrument once and return a row."""
        temperature = self.tc.temperature("A")
        resistance = self.vs.measure_resistance()
        return [time.time(), temperature, resistance]

    def run(self, interval_s: float = 1.0):
        """Acquire continuously until self.running is set False."""
        self.running = True
        while self.running:
            self.write_row(self.take_point())
            time.sleep(interval_s)
```

The structure generalizes: `columns` , `take_point()` , and `run()` are all you change for a different experiment. A frequency sweep, for instance, would loop over frequencies inside `take_point()` and widen `columns` accordingly.

3 Developing Without the Hardware

Bench time is usually the scarcest thing you have. Almost none of the code above needs an instrument to be written or debugged — only `take_point()` actually touches one — so you can build the file writing, the loop, the shutdown, and the whole analysis at your desk and arrive at the bench with a program that already works.

The trick is a stand-in object with the same methods as the real wrapper:

```python
import numpy as np


class MockDiode:
    """Stand-in for a real instrument: same methods, simulated readings.

    Returns current through an ideal diode (the Shockley equation) so the
    curve has the shape the real measurement will have.
    """

    I0 = 1e-12          # reverse saturation current, A
    N = 1.8             # ideality factor
    VT = 0.02585        # thermal voltage at room temperature, V

    def __init__(self):
        self._voltage = 0.0
        self._rng = np.random.default_rng()

    def set_voltage(self, volts: float):
        self._voltage = volts

    def measure_current(self) -> float:
        current = self.I0 * (np.exp(self._voltage / (self.N * self.VT)) - 1)
        return float(current + self._rng.normal(0, max(abs(current), 1e-12) * 0.01))
```

Give the experiment class a flag that decides which one it builds, and everything downstream — the file writing, the loop, the plotting — runs unchanged:

```python
class DiodeSweep(CSVFile):
    columns = ["voltage_V", "current_A"]

    def __init__(self, path, name, resource, offline=False):
        super().__init__(path, name)
        self.smu = MockDiode() if offline else SourceMeasureUnit(resource)
        self.running = False
        if self.is_new:
            self.write_row(self.columns)

    def take_point(self, volts):
        self.smu.set_voltage(volts)
        return [volts, self.smu.measure_current()]


# At a desk, with no instrument attached:
sweep = DiodeSweep("data", "test", resource=None, offline=True)
```

One flag at the top, one decision in `__init__`, and nothing below it knows the difference. Architecting a Lab App takes this further and explains why the *shape* of the mock data matters more than the noise on it.

4 Running Acquisition in the Background

Call `run()` directly and it blocks — the program does nothing else until the loop ends. To keep control (to stop on command, or to update a display), run the loop in a **thread** and keep the main program free:

```
import threading

experiment = ResistanceVsTemp("data", "run1", "GPIB0::12::INSTR", "GPIB0::4::INSTR")

worker = threading.Thread(target=experiment.run, daemon=True)
worker.start()        # acquisition now runs in the background

# ... main program keeps going; later, to stop cleanly:
experiment.running = False
worker.join()         # wait for the loop to finish its current pass
```

Setting `experiment.running = False` and then `join()`-ing is the clean shutdown pattern: the flag tells the loop to stop after its current pass, and `join()` waits for it to actually finish before the program moves on (so the file is fully written).

> **Note**
>
> A background thread is the simplest way to keep one program responsive. When you need *multiple* programs sharing one instrument bus, use the instrument server instead. The two compose: a server can run its own acquisition thread alongside the request loop.

5 A Simple Command Loop

For a terminal-driven app, wrap the whole thing in a loop that reads user commands while acquisition runs in the background:

```
import sys

def main():
    # Name the file from a command-line argument, or default to a timestamp.
    name = sys.argv[1] if len(sys.argv) > 1 else f"run_{int(time.time())}"
    experiment = ResistanceVsTemp("data", name, "GPIB0::12::INSTR", "GPIB0::4::INSTR")

    worker = threading.Thread(target=experiment.run, daemon=True)
    worker.start()
    print("Acquiring. Commands: 'setpoint <K>', 'stop'.")

    while True:
        command = input("> ").strip().lower()
        if command == "stop":
            experiment.running = False
            worker.join()
            break
        elif command.startswith("setpoint"):
            experiment.tc.set_setpoint(float(command.split()[1]))
        else:
            print("Unknown command.")


if __name__ == "__main__":
    main()
```

This is a small program, but it has the bones of a real instrument: data going to a documented file, acquisition that doesn't block the interface, and a clean stop. When you want that interface to be graphical instead of a terminal prompt, the next guide takes the same pieces into a GUI.

6 Where This Leads

- **Building a GUI** replaces the command loop with a live, clickable interface and a real-time plot.
- **Architecting a Lab App (MVC)** organizes a larger app so the acquisition logic, hardware, and interface stay cleanly separated.

Back to Python for Lab Work