Working Effectively in Python

For anyone past the basics who is writing code they'll use more than once. Assumes you can write a function — see Python Basics.

Most lab code gets written twice: once with the numbers you predicted, and again with the numbers you actually measured. The habits on this page are about making the second pass cheap, and about not being the person who spends lab time retyping values into a calculation.

1 Write calculations as functions

You'll often work out a quantity ahead of time using nominal values. Say you need the parallel resistance of a 50 Ω and a 100 Ω resistor:

$$\frac{1}{R_{\text{total}}} = \frac{1}{R_1} + \frac{1}{R_2}$$

The obvious thing to write is:

```
r = 1 / (1/50 + 1/100)
```

Then you get to the bench, measure the resistors you actually pulled out of the drawer, and they're 50.5 Ω and 98.4 Ω. Now you're editing numbers inside an expression — and doing it again for the next pair, and again after you swap a component.

Write it as a function instead:

```
def parallel_r(r1, r2):
    """Resistance of two resistors in parallel."""
    return 1 / (1/r1 + 1/r2)

parallel_r(50, 100)        # 33.33  — the prediction
parallel_r(50.5, 98.4)     # 33.42  — the measurement
```

Same calculation, now reusable. This is a small discipline with a large payoff: it makes updating a prediction with measured values a one-line change, and it means the calculation is written down once where you can check it, rather than scattered through a notebook in slightly different forms.

The test

If you find yourself copying a line of arithmetic and editing the numbers in the copy, that line wanted to be a function.

2 Collect your functions in one file

Functions you write for one measurement often turn out to be useful for the next. Rather than copying them between notebooks — where they immediately start drifting apart — put them in a `.py` file next to your work.

Create `lab.py`:

```
"""Reusable calculations for lab work."""

def parallel_r(r1, r2):
    """Resistance of two resistors in parallel."""
    return 1 / (1/r1 + 1/r2)


def voltage_divider(v_in, r1, r2):
    """Output voltage of a two-resistor divider."""
    return v_in * r2 / (r1 + r2)
```

Then import it wherever you need it:

```
import lab

lab.parallel_r(50.5, 98.4)
lab.voltage_divider(5.0, 2200, 4700)
```

The file has to sit in the same folder as the notebook or script importing it (or somewhere else on Python's path). Keep it commented, and it becomes a genuinely useful reference — a record of every calculation you've derived, in runnable form.

If you edit `lab.py` and nothing changes

A notebook imports a module once and caches it, so edits to `lab.py` won't show up in a kernel that already imported it. Restart the kernel, or during active development use:

```
%load_ext autoreload
%autoreload 2
```

which re-imports changed modules automatically.

3 Name things so you can read them later

Code you wrote three weeks ago is code someone else wrote. Two habits pay for themselves:

- **Names that say what the thing is.** `r_load` and `v_supply` over `r1` and `x`. Include the unit when there's any doubt: `wavelength_nm`, `t_ms`, `pressure_torr`. A surprising share of lab bugs are unit errors that a name would have caught.
- **A docstring on anything non-obvious.** One line saying what goes in and what comes out. If the formula came from somewhere, say where.

```
def beam_waist(w1, w2):
    """Waist of a Gaussian beam from two width measurements.

    Both widths in mm; returns mm. Valid only in the far field.
    """
```

That last line — the assumption — is the part you will not remember and the part that will silently invalidate a result.

4 Keep raw data separate from analysis

Never edit a raw data file. Read it, work on it in memory, and write derived results somewhere else:

```
my-experiment/
├── data/          # raw files, exactly as the instrument wrote them
├── analysis.ipynb # reads from data/, writes to results/
├── lab.py         # reusable calculations
└── results/       # derived tables and figures
```

If an analysis turns out to be wrong, you re-run it. If you edited the raw file, you re-take the data.

For the same reason, don't retype numbers out of one file into another. Load the file. A number you retyped is a number that can disagree with its source, and finding out which one is right is not how you want to spend a Sunday.

5 Comment the why, not the what

```
# BAD — restates the code
r_total = parallel_r(r1, r2)   # compute parallel resistance

# GOOD — explains a decision the code can't
# Using the measured values, not nominal: these resistors are 5% parts
# and the divider ratio is sensitive to the difference.
r_total = parallel_r(50.5, 98.4)
```

The code already says what it does. Comments are for what you'd otherwise have to reconstruct: why this approach, why this value, what you tried that didn't work.

Back to Python for Lab Work