Python Basics

A refresher on the language itself. Assumes you have Python installed and can run code — see Setup and Workflows — but no prior Python experience.

Enough Python to do lab work: values, functions, loops, and imports. If you've programmed before in any language, skim this and move on. If you haven't, work through it once and come back when something doesn't behave.

This is a floor, not a course. For depth, the official Python tutorial is genuinely good.

1 Values and variables

A variable is a name for a value. Python works out the type; you don't declare it.

```
wavelength = 632.8        # float — a decimal number
n_samples = 100           # int — a whole number
filename = "run3.csv"     # str — text
is_saturated = False      # bool — True or False
```

The distinction that bites most often is `int` vs `float`. Division always gives a float, but some operations care:

```
7 / 2       # 3.5   — true division
7 // 2      # 3     — floor division, discards the remainder
7 % 2       # 1     — remainder
2 ** 10     # 1024  — exponent (not ^, which is something else entirely)
```

> `^` is not exponentiation
>
> In Python, `^` is bitwise XOR. Writing `2^10` gives you `8`, silently and with no error. Use `**`.

2 Printing results

Use an f-string — a string prefixed with `f`, with expressions in braces:

```
resistance = 98.437
print(f"R = {resistance} ohms")        # R = 98.437 ohms
print(f"R = {resistance:.1f} ohms")    # R = 98.4 ohms
print(f"R = {resistance:.2e} ohms")    # R = 9.84e+01 ohms
```

The `:.1f` part is a format specifier: `.1f` means one decimal place, `.2e` means scientific notation with two. Report numbers to a sensible precision rather than dumping every digit Python has.

3 Functions

A function packages a calculation so you can run it again with different inputs. This matters more in lab work than it might seem — see Working Effectively for why.

```python
def photon_energy(wavelength_nm):
    """Return photon energy in joules for a wavelength in nanometres."""
    h = 6.626e-34          # Planck constant, J·s
    c = 2.998e8            # speed of light, m/s
    return h * c / (wavelength_nm * 1e-9)

energy = photon_energy(632.8)
print(f"{energy:.3e} J")
```

Three things to notice:

- **`def` starts the definition; the indented block is the body.** Indentation is syntax in Python, not decoration.
- **`return` hands a value back.** A function with no `return` gives you `None`, which is a common source of confusion when a calculation silently produces nothing.
- **The string on the first line is a docstring.** Optional, but it's how you'll remember what the function expects when you reuse it three weeks later.

Arguments can have defaults, which lets you call a function without repeating what usually doesn't change:

```python
def voltage_divider(v_in, r1, r2=1000.0):
    """Output voltage of a two-resistor divider. r2 defaults to 1 kΩ."""
    return v_in * r2 / (r1 + r2)

voltage_divider(5.0, 2200)            # uses r2 = 1000
voltage_divider(5.0, 2200, r2=4700)   # overrides it
```

4 Lists

An ordered, changeable collection:

```python
voltages = [0.1, 0.5, 1.2, 2.4]

voltages[0]        # 0.1   — first element (counting starts at zero)
voltages[-1]       # 2.4   — last element
voltages[1:3]      # [0.5, 1.2] — a slice: from index 1 up to but not including 3
len(voltages)      # 4

voltages.append(3.0)   # add to the end
```

Lists are fine for a handful of values. For numerical work — arrays of measurements you want to do arithmetic on — use NumPy arrays instead; see Arrays and Plotting.

5 Loops

A `for` loop runs a block once per item:

```python
for v in [0.1, 0.5, 1.2]:
    print(f"{v} V  ->  {v / 50:.4f} A")
```

`range(n)` gives you the whole numbers from 0 to n−1, which is how you loop a fixed number of times:

```
for i in range(5):
    print(i)          # 0 1 2 3 4
```

To loop over items *and* know their position, use `enumerate`:

```
for i, v in enumerate(voltages):
    print(f"point {i}: {v} V")
```

6 Conditionals

```
if voltage > 10:
    print("Over range — reduce the input.")
elif voltage > 9:
    print("Approaching full scale.")
else:
    print("Fine.")
```

Comparisons: `==` (equal), `!=` (not equal), `<`, `>`, `<=`, `>=`. Combine with `and`, `or`, `not`.

> `=` assigns, `==` compares
>
> `x = 5` sets `x` to 5. `x == 5` asks whether it already is. Using `=` where you meant `==` is a syntax error inside an `if`, which is the good outcome — the bad one is using `==` where you meant `=` and wondering why nothing changed.

Comparing floats for exact equality is unreliable, because floating-point arithmetic carries small rounding errors:

```
0.1 + 0.2 == 0.3        # False
abs((0.1 + 0.2) - 0.3) < 1e-9   # True — compare within a tolerance
```

7 Imports

Most of what you'll use lives in packages you import at the top of the file:

```
import numpy as np                      # whole package, under a short alias
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit    # one specific function
```

The `as np` / `as plt` aliases are near-universal conventions. Follow them — every example you'll find online assumes them, including the ones on this site.

8 When something goes wrong

Python errors name the problem on the last line. The four you'll meet first:

Error	What it usually means
`NameError: name 'x' is not defined`	Typo, or you're using a variable before setting it — or you edited a cell and didn't re-run it
`IndentationError` / `TabError`	Inconsistent indentation. Use spaces, not tabs; your editor can enforce this
`TypeError: unsupported operand type(s)`	Mixing incompatible types, often a string where a number belongs
`ModuleNotFoundError: No module named 'numpy'`	The package isn't installed, or the editor is pointed at a different Python — see Setup

Read the last line first. The traceback above it shows how execution got there, which matters once your code spans several functions.

Back to Python for Lab Work