



Everything here ships with Python. There are no installs or imports beyond the standard library. These are the pieces you will still be using once the work turns into loading datasets, counting labels, seeding a run so it repeats, and reading somebody else's function signature. Learn them on small data now and the libraries you meet later will feel like shortcuts rather than magic. **Bolded** points you toward later uses in data and AI engineering.

Types · f-strings

Four types carry most code: `int`, `float`, `str`, `bool`. An f-string reads a variable inline, and `:.3f` rounds it, the usual way to print a **machine learning score**.

```
epochs = 10          # int
loss = 0.4271         # float
name = "model-a"      # str

print(f"{name}: loss {loss:.3f}")
print(f"{0.87:.1%}")
```

```
model-a: loss 0.427
87.0%
```

Lists · slicing

An ordered, mutable sequence, and your first **vector**. Remember: slices copy a range; the end index is excluded; a negative index counts from the back.

```
scores = [0.1, 0.5, 0.9, 0.3]

scores.append(0.7)
scores[0]      # 0.1
scores[-1]     # 0.7
scores[:3]     # first three
len(scores)    # 5
```

dict

Key to value, the shape of every config and **JSON payload** you will meet. `get` returns a default instead of raising when the key is absent.

```
config = {"model": "gpt-4o", "temp": 0.2}

config["temp"] = 0.7
config.get("top_p", 1.0) # default

for key, value in config.items():
    print(key, value)
```

Comprehensions

Build a list from another in one line. The `if` at the end filters; this is the same map-and-filter idea that **array libraries** later do for whole columns.

```
raw = [1, 2, 3, 4, 5]

squares = [x * x for x in raw]
evens = [x for x in raw if x % 2 == 0]

lookup = {w: len(w) for w in ["cat", "horse"]}
```

```
[1, 4, 9, 16, 25]
[2, 4]
{'cat': 3, 'horse': 5}
```

enumerate · zip

`enumerate` gives you the index with the item. `zip` walks two sequences in step. **Pairing predictions against labels** is exactly this.

```
preds = ["cat", "dog", "cat"]
truth = ["cat", "cat", "cat"]

for i, p in enumerate(preds):
    print(i, p)

correct = sum(p == t for p, t in zip(preds, truth))
accuracy = correct / len(truth)
```

Functions

Arguments with defaults are optional at the call site. ***args** and **kwargs collect the rest, which is why** library signatures** so often end that way.

```
def normalize(values, low=0.0, high=1.0):
    lo, hi = min(values), max(values)
    span = hi - lo or 1
    return [low + (v - lo) / span * (high - low)
            for v in values]

normalize([2, 4, 6])
```

```
[0.0, 0.5, 1.0]
```

Type hints

Annotations that Python does not enforce but humans and editors read. Knowing the notation is how you **decode documentation**.

```
def mean(values: list[float]) -> float:
    return sum(values) / len(values)

def load(path: str, limit: int | None = None
         ) -> dict[str, list[int]]:
    ...
```

pathlib

Paths as objects, not strings. `/` joins them correctly on any OS, and the methods answer the questions you actually have. Essential for point to and **loading datasets**.

```
from pathlib import Path

data = Path("data")
train = data / "train.csv"

train.exists()
train.suffix      # '.csv'
list(data.glob("*.csv"))
data.mkdir(exist_ok=True)
```

open · with

`with` closes the file for you, even if the code inside fails. Read a whole file at once, or loop it line by line when it is large. Great for **controlling file loading**: text, pipelines, or outputs.

```
with open("notes.txt") as f:
    text = f.read()

with open("notes.txt") as f:
    for line in f:
        line = line.strip()

with open("out.txt", "w") as f:
    f.write("done\n")
```

json

The format of configs, **API requests and API replies**. Load/dump work on files; loads/dumps on strings; the `s` is for string.

```
import json

with open("config.json") as f:
    config = json.load(f)

text = json.dumps(config, indent=2)
back = json.loads(text)
```

CSV

Read datasets without a third-party library. `DictReader` gives one dictionary per row, keyed by the header.

```
import csv

with open("sales.csv", newline="") as f:
    rows = list(csv.DictReader(f))

rows[0]["price"]          # always a str
prices = [float(r["price"]) for r in rows]
```

Counter · defaultdict

`Counter` tallies anything iterable — class balance, word frequency. A `defaultdict` creates the missing value instead of raising `KeyError`.

```
from collections import Counter, defaultdict

labels = ["spam", "ham", "spam", "spam"]
counts = Counter(labels)
counts.most_common(1)

groups = defaultdict(list)
for lab, txt in zip(labels, texts):
    groups[lab].append(txt)
```

```
Counter({'spam': 3, 'ham': 1})
[('spam', 3)]
```

itertools

Lazy building blocks for loops. `batched` cuts a sequence into fixed-size chunks. `product` enumerates every combination of settings. Good for **memory management** with large ranges.

```
from itertools import batched, product

for chunk in batched(range(10), 4):
    print(chunk)          # (0,1,2,3) ...

grid = list(product([0.1, 0.01], [16, 32]))
# (0.1, 16), (0.1, 32), (0.01, 16) ...
```

random · seed

Set the seed and the same "random" numbers come back every run. Shuffles and splits start here, before any framework is involved. Essential for **reproducibility**.

```
import random

random.seed(42)

random.random()          # 0.0-1.0
random.choice(["a", "b"])
random.sample(rows, k=100) # no repeats
random.shuffle(rows)      # in place
```