5 ways to write better Python

Demonstrated via a logistic regression classifier of breast tumour diagnosis.

Executable demo

Simple, containerised ML pipeline to classify tumours as malignant or benign based on features computed from digitized images of fine needle aspirates of breast mass.

1. **Loads and normalizes** dataset[1] – *helper functions*
2. **Splits** it into train/test sets – *multiple assignment unpacking*
3. **Validates** the files robustly – *exception raising*
4. **Selects** top features by variance – *dict/list comprehension*
5. **Trains and evaluates** a logistic regression model – *virtual environment*

[1] UCI ML Breast Cancer Wisconsin (Diagnostic) dataset

1: Write helper functions instead of complex expressions

```python
def minmax_normalize(df: pd.DataFrame) -> pd.DataFrame:
    """Normalize numeric columns to [0, 1] range; leave non-numeric intact."""
    num = df.select_dtypes(include="number")
    den = (num.max() - num.min()).replace(0, 1)  # avoid divide-by-zero
    scaled = (num - num.min()) / den
    out = df.copy()
    out[scaled.columns] = scaled
    return out
```

We **encapsulate** normalization, making preprocessing **reusable** and **readable**.

2: Prefer multiple assignment unpacking over indexing

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

We split data and **unpack the result directly** into four variables, making code **clearer** and therefore less error-prone.

3: Prefer raising exceptions to returning None

```python
def assert_exists(path: str):
    if not os.path.exists(path):
        raise FileNotFoundError(f"{path} is missing — run previous steps first.")

def assert_nonempty_dataframe(path: str):
    df = pd.read_csv(path)
    if df.empty:
        raise ValueError(f"{path} is empty.")
    return df
```

```python
# Simple sanity check: matching lengths
if len(X) != len(y):
    raise ValueError(f"Length mismatch: X={len(X)} vs y={len(y)}")
```

We **validate** that required artifacts **exist** and are have **sensible** format/values wherever possible, and otherwise raise explicit errors.

4: Use comprehensions instead of loops

```python
# Dict comprehension for variances
variances = {col: X_train[col].var() for col in X_train.columns}

# List comprehension to select top-K by variance
selected = [c for c, _ in sorted(variances.items(), key=lambda kv: kv[1], reverse=True)[:TOP_K]]
```

We compute variances and pick the top-K features using dict/list comprehensions, for **clarity**, **conciseness** and to maximise **performance**.

5: Use virtual environments for reproducibility

Examples: Docker, UV, Anaconda…
Result - a **clean, deterministic environment** enabling easy reproducibility and consistency.

UV ensures the environment is **fast to build and fully isolated and reproducible**, including transitive dependency behaviour through uv.lock .

Dockerfile > ...

```
1   FROM python:3.12-slim
2
3   WORKDIR /app
4   COPY . /app
5
6   # Install uv
7   RUN apt-get update && apt-get install -y curl && \
8       curl -LsSf https://astral.sh/uv/install.sh | sh && \
9       rm -rf /var/lib/apt/lists/*
10
11  ENV PATH="/root/.local/bin:$PATH"
12
13  # Install dependencies
14  RUN uv sync --frozen
15
16  # Prepare dataset
17  RUN uv run python prepare_data.py
18
19  ENTRYPOINT ["uv", "run", "python", "demo.py"]
```

pyproject.toml

```
1   [project]
2   name = "effective-python-tool"
3   version = "0.1.0"
4   readme = "README.md"
5   requires-python = ">=3.12"
6   dependencies = [
7       "pandas>=2.3.3",
8       "scikit-learn>=1.7.2",
9   ]
```