Python Ways

5 ways to write better Python

Mohammed Sultan

Write helper functions

instead of complex expressions

Avoid Complex Single-Line Expressions

Python's syntax allows for complex one-liners, but overly concise code can be difficult to understand and debug.

Create Helper Functions for Reusable Logic

When expressions grow complicated, or logic is reused, it’s best to move this into a helper function to simplify the main code, enhance readability, and avoid repetition.

Prioritise Readability Over Brevity

Readability should always take precedence. Cleaner, well-structured code helps maintain functionality and makes it easier for others to understand, especially for complex or repeated logic.

Item 5 in the book.

```
from urllib.parse import parse_qs

# Initial dictionary with query parameters
query_string = 'red=5&blue=0&green='
my_values = parse_qs(query_string, keep_blank_values=True)

red = int(my_values.get('red', [''])[0] or 0)
```

```
def get_first_int(values, key, default=0):
    found = values.get(key, [''])
    if found[0]:
        return int(found[0])
    return default
```

Raise Exceptions Instead of Returning *None*

Raise exceptions instead of returning *None* for errors to make issues explicit and ensure the caller handles them properly.

Avoid Returning None for Errors

Returning None to indicate errors is risky, as it can be misinterpreted in conditionals. Instead, raise exceptions for clear, explicit error handling.

Use Type Annotations for Clarity

Type annotations signal that a function will not return None, making the function's behaviour and error-handling requirements clearer to the caller.

Use Comprehensions instead of *map* and *filter*

Useful for creating new lists, dictionaries, or sets by transforming or filtering an existing sequence.

Item 27 in the book.

For Clarity in Code

List comprehensions are cleaner and more readable than map and filter, as they avoid lambda expressions.

Efficient Filtering

List comprehensions enable single-line filtering, unlike map, which needs filter, adding complexity.

Other Data Structures

Comprehensions create dictionaries and sets directly, offering a compact way to build data structures from sequences.

An example demonstrating comprehensions.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = []

even_squares = [x**2 for x in a if x % 2 == 0] # List comprehension + filtering

alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a)) # The alternative
```

Item 30 in the book.

Generators instead of lists

Generators yield items one at a time instead of returning the whole list at once, saving memory and allowing efficient handling of large data.

Generators Simplify Code

Generators replace lists by yielding results one at a time, making functions cleaner and easier to read.

Efficient Memory Use

Generators handle large datasets smoothly, as they don’t store all results in memory, reducing risk of memory issues.

Flexible Output

Generators can easily process large or streamed data, like reading files line-by-line, and convert to a list if needed.

Item 83 in the book.

Virtual Environments

Virtual environments isolate dependencies, prevent version conflicts, and ensure each project has the packages it needs—ideal for collaboration, reproducibility, and managing complex projects.

Isolated Dependencies

Virtual environments keep project dependencies separate, avoiding conflicts between packages needed by different projects.

Easy Setup and Activation

Create a virtual environment with **python -m venv**, activate it with **source bin/activate**, and deactivate it with **deactivate**.

Reproducibility

Use **pip freeze** to save all dependencies to a **requirements.txt file**, allowing easy setup of the same environment on another machine by running **pip install -r requirements.txt**.