Refactoring to Functions

Python Course · Module 4.5

ElnurBDa

Module 4.5 — From Loops to Reusable Functions

Goals:

- Recognize when repeated patterns should become **functions**
- Practice extracting logic from loops into named helpers
- Prepare for multi-file organization in Module 5

Think of this as “cleaning up” your scripts before modularizing them.

Why Refactor to Functions?

Example of a script **without** functions:

```
scores = [80, 95, 60, 70]

total = 0
for s in scores:
    total += s
average = total / len(scores)
print("Average:", average)

passed = 0
for s in scores:
    if s >= 70:
        passed += 1
print("Passed:", passed)
```

Problems:

- Harder to reuse logic with other lists
- No clear names for key operations

We'll refactor step by step.

Step 1 — Wrap Logic in a Function

```python
def average(scores):
    total = 0
    for s in scores:
        total += s
    return total / len(scores)

scores = [80, 95, 60, 70]
print("Average:", average(scores))
```

———————————— [finished] ————————————

```
Average: 76.25
```

Benefits:

- Name `average` describes intent
- Easy to test with different lists

Try writing `count_passed(scores, threshold)` in the same style.

Step 2 — Use Existing Constructs

After writing the basic function with a loop, you can improve it:

```
def average(scores):
    return sum(scores) / len(scores)

def count_passed(scores, threshold=70):
    passed = 0
    for s in scores:
        if s >= threshold:
            passed += 1
    return passed

scores = [80, 95, 60, 70]
print("Average:", average(scores))
print("Passed:", count_passed(scores))
```

———————— [finished] ————————

```
Average: 76.25
Passed: 3
```

Pattern:

1. Make it work
2. Make it **clear**
3. Only then make it shorter if it’s still readable

Mini Practice — Refactor Temperature Script

Take a plain script like:

```
temps = [20, 25, 18, 30]

for t in temps:
    if t > 25:
        print(t, "Hot")
    else:
        print(t, "OK")
```

Refactor into:

- `label_temp(t)` → returns `"Hot"` or `"OK"`
- `print_labeled_temps(temps)` → loops and prints

Then call these functions from `if __name__ == "__main__":`.

Separating Concerns — Input vs Logic vs Output

Good structure:

```python
def read_scores():
    raw = input("Enter scores separated by commas: ")
    parts = raw.split(",")
    return [int(p.strip()) for p in parts]

def compute_average(scores):
    return sum(scores) / len(scores)

def main():
    scores = read_scores()
    avg = compute_average(scores)
    print("Average:", avg)

if __name__ == "__main__":
    main()
```

———————————— [finished with error] ————————————

```
Enter scores separated by commas: Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermF5X1aP/snippet.py", line 15, in
<module>
    main()
    ~~~~^^
  File "/tmp/nix-shell-61594-0/.presentermF5X1aP/snippet.py", line 10, in main
    scores = read_scores()
  File "/tmp/nix-shell-61594-0/.presentermF5X1aP/snippet.py", line 2, in
read_scores
    raw = input("Enter scores separated by commas: ")
EOFError: EOF when reading a line
```

Roles:

- `read_scores` → input parsing
- `compute_average` → pure calculation
- `main` → wiring everything together

This pattern is exactly what Module 5 will build on.

Mini Project — Number Analyzer

Create `number_analyzer.py` that:

1. Has a `read_numbers()` function that reads a comma-separated line and returns a list of `int`.
2. Has `summary_stats(numbers)` that returns a dict with:
   - `count`, `min`, `max`, `average`
3. Has `print_report(stats)` that prints a formatted summary.
4. Uses a `main()` function plus the `if __name__ == "__main__":` pattern.

You'll reuse this style when you move to multiple files and modules next.
