The AI-Native Refactorer

Topic 3: Magic Numbers and Strings

Hands-On: Extracting Magic Values at the Right Scope

Overview

This hands-on proves the three core claims from the Topic 3 lecture using an AI coding assistant and a Python file riddled with all four shapes of magic values:

- Extracting magic values is the smallest refactor with the biggest readability win.
- The right scope (function-local vs module vs config) matters as much as the rename itself.
- AI catches magic values humans walk past every day — including conceptually-related literals that look unrelated.

Before You Start

- Have Python 3.10+ installed.
- Install pytest: pip install pytest.
- Open topic3_before.py from https://github.com/TheAI-Native/AI-NativeRefactorer
- Have an AI coding assistant ready in your editor.
- Have a terminal open in the same directory.

> **REMEMBER**
> The goal is NOT to extract every literal in the file. The goal is to extract every literal whose meaning isn't obvious from context. Over-extraction is real — a constant called ONE = 1 helps no one. Use judgment, and let the AI's suggested scope guide you.

During Hands-On — Three Steps at a Glance

- STEP 1 — Scan for magic values and classify them by shape and scope. ~4 min
- STEP 2 — Protect with tests, then extract by scope. ~7 min
- STEP 3 — Verify behavior is preserved and review the readability win. ~4 min

Step 1 — Scan and Classify

WHAT TO LEARN

There are four shapes of magic values, and each has a different right answer. Numeric literals become symbolic constants. Status strings become enums. Format strings become formatters. Configuration values get pulled out of business code. Knowing the shape determines the fix.

Run the file first to confirm everything works:

```
python3 topic3_before.py
```

Then ask the AI to find every magic value, classified by shape:

```
> Scan topic3_before.py for every literal value other than 0, 1, -1,
> and empty string. Group them into four categories: numeric literals,
> status/tier strings, format strings, and hardcoded configuration.
> For each one, suggest a name and the right scope.
```

The AI should identify roughly: 30 (days), 0.08 (tax rate), 86400 (seconds/day), the strings "active", "inactive", "pending", "premium", "standard", "free"; the date format and currency format; and the three PaymentGateway config values (timeout, max_retries, base_url).

WHAT YOU SHOULD GET

A categorized inventory. The numeric and string literals belong at module level. The format strings also at module level. The payment configuration belongs in a settings dataclass. TIER_LIMITS belongs as a module-level lookup table. INACTIVE_AFTER_DAYS, TAX_RATE, SECONDS_PER_DAY, FREE_TIER_CHARGE_LIMIT are the named constants you'll need.

Step 2 — Protect and Extract

WHAT TO LEARN

Renaming literals looks safe but isn't. Python is dynamically typed, and a literal you change might be compared against a literal you didn't. Tests catch this. Generate them first.

Generate the characterization tests:

```
> Write pytest tests that capture the current behavior of
> SubscriptionService and PaymentGateway. Cover active/inactive
> status logic, all three tier limit lookups, renewal-charge math
> with and without discount, and the free-tier payment limit.
```

Save as topic3_tests.py and run:

```
pytest topic3_tests.py -v
```

Now extract by scope. Start with module-level numeric constants:

```
> Replace 30 (days), 0.08 (tax rate), and 86400 (seconds per day) with
> module-level constants: INACTIVE_AFTER_DAYS, TAX_RATE, SECONDS_PER_DAY.
> Show the diff only.
```

Run the tests after each set of extractions:

```
pytest topic3_tests.py -v
```

Then introduce enums for the string literals:

```
> Replace the status strings ("active", "inactive", "pending") with a
> UserStatus enum, and the tier strings ("premium", "standard", "free")
> with a Tier enum. Update all callers. Keep User.status as a string
> (use enum.value) so existing serialization still works.
```

Then collapse the tier-limit branching into a lookup table:

```
> Replace the three if-tier branches in get_limits with a single
> module-level TIER_LIMITS dictionary keyed by the Tier enum. The
> method becomes a single dict lookup with a Free fallback.
```

Finally extract the configuration out of PaymentGateway:

```
> Replace the hardcoded timeout, max_retries, and base_url in
> PaymentGateway with a frozen PaymentSettings dataclass. Provide
> a DEFAULT_PAYMENT_SETTINGS module constant. The constructor should
> accept an optional settings parameter.
```

WHAT YOU SHOULD GET

All 12 tests still pass. The file now has a clean banner of named constants at the top, two Enums, one lookup table, and a settings dataclass. The business code (the methods) no longer contains any literal value with a non-obvious meaning. The shape of the code has changed; the behavior has not.

Step 3 — Verify and Compare

WHAT TO LEARN

The point of the refactor is the readability win, not the test pass. The test pass is just the safety net. Take a moment to read the before and after side by side, and notice which file you could explain to a new hire faster.

Diff against the reference target:

```
diff topic3_before.py topic3_after.py | less
```

Then ask the AI the questions that prove the refactor was worth it:

```
> Where would I add a new "ENTERPRISE" tier with a 25% discount and
> 200 max users? List the exact lines I would change.
```

In the original code, the answer is "edit three if-branches and remember to update get_limits." In the refactored code, the answer is one line in the Tier enum and one row in TIER_LIMITS. That gap — the difference between adding a feature in five places versus two lines — is the permanent payoff of this refactor.

WHAT YOU SHOULD GET

Confidence that adding a new tier, changing the tax rate, or pointing payment at a staging URL is now a one-line change made in one obvious place. That's the right measure of whether the refactor was successful — not lines of code added or removed.

Wrap-Up — The Pattern, The Assignment Handoff

REMEMBER

Magic-value refactoring is the cheapest, fastest, highest-leverage refactor in the catalog. It almost never breaks anything (with tests), it makes code dramatically more readable, and it sets up the foundation for any future config-driven architecture. If you have ten minutes and no plan, scan for magic values.

Key takeaways:

- Four shapes — numeric, status strings, format strings, configuration. Each has its own consolidation path.

- Scope is a judgment call. Function-local for one-time use, module-level for shared, settings module for environment-dependent.
- Enums beat strings for any fixed named domain — the type system catches typos.
- Lookup tables beat if-tier chains — adding a new tier becomes a one-line config change.

Source File Reference

The full source for this demo lives at https://github.com/AI-Native-Engineer-Course/AI-Native-Refactorer/tree/main/topic3/. It contains topic3_before.py (the magic-value-riddled service), topic3_after.py (with extracted constants, enums, and a settings dataclass), and topic3_tests.py (the characterization tests). Open all three before class and rehearse the diff between before and after.