Leap Year Kata

A Test-Driven Development Journey

Introduction

This document walks through the complete Test-Driven Development (TDD) implementation of the Leap Year kata. We'll build the solution incrementally, one test at a time, following the Red-Green-Refactor cycle.

Problem Statement

Implement a function **is_leap_year(year)** that determines if a year is a leap year according to the Gregorian Calendar rules:

1. Years divisible by 400 ARE leap years (e.g., 2000)

2. Years divisible by 100 but not by 400 are NOT leap years (e.g., 1900)

3. Years divisible by 4 but not by 100 ARE leap years (e.g., 2016)

4. Years not divisible by 4 are NOT leap years (e.g., 2017)

TDD Approach

We follow the Red-Green-Refactor cycle:

■ **RED**: Write a failing test

■ **GREEN**: Write the simplest code to make it pass

■ **REFACTOR**: Improve the code without changing behavior

■ **COMMIT**: Document the progress with a clear commit message

■ **REPEAT**: Continue with the next test

Phase A: Non-Leap Years

Start with the simplest case: years that are NOT leap years.

Test 1: 2017 → False

■ RED: Write the test

```
def test_2017_is_not_leap_year():
    assert is_leap_year(2017) is False
```

■ GREEN: Make it pass with the simplest code

```
def is_leap_year(year):
    return False
```

■ **Learning**: "Fake it till you make it" - start with hardcoded values.

Tests 2 & 3: 2018, 2019 → False

Add two more tests for triangulation. They both pass with the existing hardcoded `return False`. This confirms our simple solution works for all non-leap years.

Result: 3 tests passing ■

Phase B: Basic Leap Years

Now add years that ARE leap years (divisible by 4).

Test 4: 2016 → True

■ RED: Write the test (it will fail!)

```
def test_2016_is_leap_year():
    assert is_leap_year(2016) is True
```

Test fails: returns False, expected True

■ GREEN: Add real logic

```
def is_leap_year(year):
    if year % 4 == 0:
        return True
    return False
```

■ **Learning**: Tests force us to implement general logic, not hardcoded values.

Tests 5 & 6: 2012, 2008 → True

Add triangulation tests. They pass with existing logic, confirming the modulo 4 rule is general.

Result: 6 tests passing ■

Phase C: Century Exceptions

Century years (divisible by 100) are NOT leap years unless divisible by 400.

Test 7: 1900 → False

■ RED: Write the test (it will break our logic!)

```
def test_1900_is_not_leap_year():
    assert is_leap_year(1900) is False
```

Test fails: returns True (because 1900 % 4 == 0), expected False

■ GREEN: Add century check BEFORE the modulo 4 check

```
def is_leap_year(year):
    if year % 100 == 0 and year % 400 != 0:
        return False
    if year % 4 == 0:
        return True
    return False
```

■ **Learning**: Order matters! Check exceptions BEFORE general rules.

Tests 8-10: 1800, 1700, 2100 → False

Add more century year tests. They all pass with existing logic.

Result: 10 tests passing ■

Phase D: 400-Year Rule

Century years divisible by 400 ARE leap years.

Test 11: 2000 → True

■ RED: Write the test

```
def test_2000_is_leap_year():
    assert is_leap_year(2000) is True
```

Surprisingly, test passes! Our logic accidentally worked because 2000 % 400 == 0, so the century check evaluates to False and we fall through to year % 4 == 0.

■ REFACTOR: Make the 400-year rule explicit

```
def is_leap_year(year):
    if year % 400 == 0:
        return True
    if year % 100 == 0:
        return False
    if year % 4 == 0:
        return True
    return False
```

■ **Learning**: Make implicit logic explicit. Check most specific rules first (400 → 100 → 4).

Tests 12 & 13: 2400, 1600 → True

Add triangulation tests. They pass with refactored logic.

Result: 13 tests passing ■ | 100% coverage ■

Final Implementation

Explicit version (more readable):

```
def is_leap_year(year):
    """Determine if a year is a leap year."""
    # Years divisible by 400 ARE leap years
    if year % 400 == 0:
        return True
    # Century years are NOT leap years
    if year % 100 == 0:
        return False
    # Years divisible by 4 are leap years
    if year % 4 == 0:
        return True
    return False
```

Condensed version (more elegant):

```
def is_leap_year(year):
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
```

Test Coverage Breakdown

Phase A: Non-leap years (3 tests)

2017, 2018, 2019 → False

Phase B: Basic leap years (3 tests)

2016, 2012, 2008 → True

Phase C: Century exceptions (4 tests)

1900, 1800, 1700, 2100 → False

Phase D: 400-year rule (3 tests)

2000, 2400, 1600 → True

Total: 13 tests, 100% code coverage ■

Key TDD Principles

1. **Red-Green-Refactor**: Every change followed this cycle
2. **Fake it till you make it**: Started with hardcoded values
3. **Triangulation**: Multiple tests confirm general solutions
4. **Small steps**: One test at a time, minimal code changes
5. **Order matters**: Most specific rules checked first (400→100→4)
6. **Frequent commits**: Clear history showing progression

Git Commit History

```
fe5f47c Phase A: Non-leap years (not divisible by 4)

23759ba Phase B: Basic leap years (divisible by 4)

2969d65 Phase C: Century years NOT divisible by 400

bb08149 Phase D: Century years divisible by 400
```

Each commit includes Red-Green-Refactor details, test counts, coverage, and learning points.

What We Learned

- TDD drives design incrementally
- Start simple, let tests demand complexity
- Rule order matters (check 400 before 100 before 4)
- Explicit code is better than implicit
- Triangulation validates general solutions
- Small commits create a reviewable story

Conclusion

This kata demonstrates the power of TDD. We started with no implementation, let tests drive the design, and arrived at a correct, well-tested solution with 100% coverage. The journey is as valuable as the destination!