FizzBuzz Kata with TDD

A Step-by-Step Journey Through Test-Driven Development

What is Test-Driven Development?

- Red-Green-Refactor Cycle
  - Write a failing test first (Red)
  - Write minimal code to pass the test (Green)
  - Refactor to improve code quality (Refactor)
  - Commit after each complete phase
- Benefits of TDD
  - Ensures code correctness from the start
  - Creates living documentation through tests
  - Makes refactoring safer
  - Encourages simple, focused design

FizzBuzz Problem

- Convert numbers to strings with these rules:
  - Multiples of 3 → 'Fizz'
  - Multiples of 5 → 'Buzz'
  - Multiples of both 3 and 5 → 'FizzBuzz'
  - Everything else → the number as a string
- Two APIs to build:
  - fizzbuzz_of(number) → string
  - fizzbuzz_1_to(n=100) → list[string]

Project Setup: Python Environment

- Used pyenv for Python version management
  - Python 3.10.13 selected
  - Allows easy switching between versions
- Modern project structure:
  - src/ - production code
  - src/tests/ - test code (tests inside src)
  - pyproject.toml - modern Python packaging

pyproject.toml Configuration

```
[project]
name = "fizz-buzz-kata"
version = "0.1.0"
requires-python = ">=3.10"

[project.optional-dependencies]
dev = [
    "pytest>=7.4.0",
    "pytest-cov>=4.1.0",
    "pytest-watch>=4.2.0",
    "flake8>=6.0.0",
    "pre-commit>=3.0.0",
]

[tool.pytest.ini_options]
testpaths = ["src/tests"]
addopts = ["-v", "--cov=src"]
```

Quality Gates: Pre-commit Hooks

- Automated checks before every commit:
  - flake8 - Code style and quality checks
  - pytest - All tests must pass
  - Trailing whitespace removal
  - YAML/TOML validation
- Benefits:
  - Prevents bad code from being committed
  - Enforces consistent code style
  - Catches errors early

Continuous Integration: GitHub Actions

- Automated testing on every push:
  - Tests on Python 3.10, 3.11, 3.12
  - Runs flake8 for code quality
  - Runs pytest with coverage
  - Matrix strategy for multiple versions
- Ensures code works across Python versions
- Provides confidence for collaboration

Git Setup: Local vs Global Config

- Work vs Personal Projects:
  - Global config for work (GitLab)
  - Local config for personal (GitHub)
- Commands used:
  - git config --local user.name 'TomSpencerLondon'
  - git config --local user.email 'tomspencerlondon@gmail.com'
- GitHub CLI (gh) for repository management
  - gh auth login
  - gh repo create

Phase A: Numbers (not multiples of 3 or 5)

- Goal: Return number as string for non-special numbers
- Test 1: 1 → '1'
  - Red: Write failing test
  - Green: return '1' (fake it)
- Test 2: 2 → '2'
  - Red: Add test for 2
  - Green: if/else for 1 and 2
- Test 3: 4 → '4'
  - Red: Add test for 4
  - Green: Extend logic
  - Refactor: Replace with return str(number)

Phase A: Final Code

```
def fizzbuzz_of(number):
    '''Convert a number to FizzBuzz string.'''
    return str(number)

# Tests
def test_1_is_string():
    assert fizzbuzz_of(1) == "1"

def test_2_is_string():
    assert fizzbuzz_of(2) == "2"

def test_4_is_string():
    assert fizzbuzz_of(4) == "4"
```

TDD Principle: One Behavior at a Time

- Why skip 3 in Phase A?
  - 3 introduces NEW behavior (Fizz)
  - Phase A focuses on 'number → string' behavior
  - Finish current behavior before adding new ones
- This keeps each phase focused and simple
- Triangulation with 1, 2, 4:
  - Multiple examples of same behavior
  - Forces general solution: str(number)

Phase B: Multiples of 3 → 'Fizz'

- Test 4: 3 → 'Fizz'
  - Red: Write failing test
  - Green: if number % 3 == 0: return 'Fizz'
- Test 5: 6 → 'Fizz' (triangulate)
  - Red: Add test for 6
  - Green: No code change needed!
  - The % 3 rule already handles it
- Triangulation confirms our solution is general

Phase B: Code Evolution

```
def fizzbuzz_of(number):
    '''Convert a number to FizzBuzz string.'''
    if number % 3 == 0:
        return "Fizz"
    return str(number)

# New tests
def test_3_is_fizz():
    assert fizzbuzz_of(3) == "Fizz"

def test_6_is_fizz():
    assert fizzbuzz_of(6) == "Fizz"
```

Phase C: Multiples of 5 → 'Buzz'

- Test 6: 5 → 'Buzz'
  - ■ Red: Write failing test
  - ■ Green: if number % 5 == 0: return 'Buzz'
- Test 7: 10 → 'Buzz' (triangulate)
  - ■ Red: Add test for 10
  - ■ Green: No code change needed!
- Pattern emerging: Test, implement, triangulate

Phase C: Adding Buzz Logic

```
def fizzbuzz_of(number):
    '''Convert a number to FizzBuzz string.'''
    if number % 3 == 0:
        return "Fizz"
    if number % 5 == 0:
        return "Buzz"
    return str(number)

# New tests
def test_5_is_buzz():
    assert fizzbuzz_of(5) == "Buzz"

def test_10_is_buzz():
    assert fizzbuzz_of(10) == "Buzz"
```

Phase D: The Combined Case Challenge

- Test 8: 15 → 'FizzBuzz'
- Problem: 15 is a multiple of BOTH 3 and 5
  - ■ Current code would return 'Fizz' and stop
  - ■ We need 'FizzBuzz' instead
- Solution: Order matters!
  - ■ Check combined case (% 15) first
  - ■ Then check individual cases
- Key TDD insight: Tests drive the design

Phase D: Final Implementation

```
def fizzbuzz_of(number):
    '''Convert a number to FizzBuzz string.'''
    if number % 15 == 0:
        return "FizzBuzz"
    if number % 3 == 0:
        return "Fizz"
    if number % 5 == 0:
        return "Buzz"
    return str(number)

# New test
def test_15_is_fizzbuzz():
    assert fizzbuzz_of(15) == "FizzBuzz"
```

Alternative: Build String Approach

```
def fizzbuzz_of(number):
    '''Alternative implementation.'''
    result = ""
    if number % 3 == 0:
        result += "Fizz"
    if number % 5 == 0:
        result += "Buzz"
    return result or str(number)
```

```
# This approach:
# - Checks both conditions independently
# - Builds the string progressively
# - No need to check % 15 explicitly
# - Same behavior, different structure
```

Phase E: Sequence API

- Goal: Generate sequence from 1 to n
- Test 9: fizzbuzz_1_to(5)
  - Red: Test expects ['1','2','Fizz','4','Buzz']
  - Green: Use list comprehension
  - Reuse fizzbuzz_of() for each number
- Test 10: Default to 100
  - Red: Test expects 100 items by default
  - Green: Add default parameter n=100

Phase E: Sequence Implementation

```
def fizzbuzz_1_to(n=100):
    '''Generate FizzBuzz sequence from 1 to n.

    Args:
        n: Upper limit (default 100)

    Returns:
        List of FizzBuzz strings from 1 to n
    '''
    return [fizzbuzz_of(i) for i in range(1, n + 1)]

# Tests
def test_sequence_to_5():
    assert fizzbuzz_1_to(5) == ["1", "2", "Fizz", "4", "Buzz"]

def test_sequence_default_is_100():
    result = fizzbuzz_1_to()
    assert len(result) == 100
    assert result[99] == "Buzz"
```

Power of Code Reuse

- fizzbuzz_1_to reuses fizzbuzz_of
- Benefits:
  - Single source of truth for FizzBuzz logic
  - Changes to rules only need one place
  - Composition over duplication
  - Easy to test and maintain
- This is the 'DRY' principle:
  - Don't Repeat Yourself

Final Test Results

- 10 tests, all passing ✓
- Coverage: 100%
  - Every line of production code is tested
  - High confidence in correctness
- Tests run in multiple places:
  - Pre-commit hooks (local)
  - GitHub Actions (CI/CD)
  - Python 3.10, 3.11, 3.12
- Total time: ~30 seconds per full CI run

Git History: The Story

- Initial commit: Add CLAUDE.md with FizzBuzz kata TDD guidelines
- Add Python project setup with pytest and TDD structure
- Add pre-commit hooks with flake8 and pytest
- Test pre-commit hooks with good code
- Clean up pre-commit hook test files
- Add GitHub Actions CI/CD pipeline
- Phase A & B: Implement basic number conversion and Fizz for multiples of 3
- docs: Add commit strategy to CLAUDE.md
- Phase C: Add Buzz for multiples of 5
- Phase D: Add FizzBuzz for multiples of both 3 and 5
- docs: Add FizzBuzz kata phases to README
- Phase E: Add sequence API (fizzbuzz_1_to)

Commit Strategy for Learning

- Commit after each phase completion
- Each commit message includes:
  - Phase identifier (A, B, C, D, E)
  - What tests were added
  - What production code changed
  - Test count and coverage
- Benefits for revision:
  - Easy to replay the progression
  - See how tests drive design
  - Understand each decision point

Key TDD Learnings

- 1. Write ONE failing test at a time
    - ■ Focus prevents overwhelm
- 2. Write minimal code to pass
    - ■ 'Fake it till you make it' is OK
- 3. Triangulation forces general solutions
    - ■ Multiple examples reveal patterns
- 4. Refactor only when tests are green
    - ■ Safety net prevents breaking changes
- 5. Test one behavior at a time
    - ■ Keeps phases focused and manageable

Quality Automation Stack

- Layer 1: Pre-commit Hooks (Local)
  - Runs before each commit
  - Fast feedback (< 5 seconds)
  - Prevents bad commits
- Layer 2: GitHub Actions (Remote)
  - Runs on push to GitHub
  - Tests multiple Python versions
  - Catches environment-specific issues
- Result: High confidence, low manual effort

Final Project Structure

```
fizz_buzz/
■■■ .github/
■   ■■■ workflows/
■       ■■■ ci.yml          # GitHub Actions
■■■ src/
■   ■■■ __init__.py
■   ■■■ fizzbuzz.py         # Production code
■   ■■■ tests/
■       ■■■ __init__.py
■       ■■■ test_fizzbuzz.py # Tests
■■■ .pre-commit-config.yaml  # Hooks config
■■■ .flake8                  # Linting rules
■■■ pyproject.toml           # Project config
■■■ CLAUDE.md                # AI pairing guide
■■■ README.md                # Documentation
```

Potential Refactorings (Stretch Goals)

- Make divisors configurable:
  - rules = [(3, 'Fizz'), (5, 'Buzz')]
  - More flexible for variations
- Add input validation:
  - Reject negative numbers
  - Reject non-integers
- Output formatting:
  - Join with newlines or spaces
  - Print directly to console
- All these would be test-driven!

Applying These Lessons

- This approach works for any kata:
- 1. Set up quality gates first
  - Pre-commit hooks
  - CI/CD pipeline
- 2. Break problem into phases
  - One behavior per phase
  - Simple to complex
- 3. Follow Red-Green-Refactor strictly
- 4. Commit regularly with good messages
- 5. Use git history as learning tool

Summary: FizzBuzz TDD Journey

- ✓ Modern Python project setup
- ✓ Automated quality gates (hooks + CI/CD)
- ✓ 5 phases, each building on the last
- ✓ 10 tests, 100% coverage
- ✓ Clean, maintainable code
- ✓ Git history tells the story

- TDD isn't just about testing—
- it's a design methodology that leads to:
  - ■ Better architecture
  - ■ Higher confidence
  - ■ Living documentation
  - ■ Easier refactoring