Programming Course and Project

Summer Term 2024/25

Tutorial 2 - Test-Driven Development & Project Basics

Felix Lundt - April 28, 2025

Tentative outline for the first phase

	Content Software Carpentry	Algorithm/ Game Play	General
Week 1 **April 14**	Project Setup		Intro, Python & Numpy primer
Week 2 **April 28**	TDD	Code skeleton	
Week 3 **May 5**	Git Modules & Packages	Code to play Random Agent Algorithms	
Week 4 **May 12**	Debugging & Documentation		Exam Registration (May 12th) Submission?
Week 5 **May 19**	Profiling		Submission Prototype?

Recap Week 1 & Organizational Stuff

- Everybody good with:
  - Python and numpy
  - Virtual environments
  - IDE
- Exam registration
- Office hours: Tuesdays, 1pm-2pm (with prior notice)

Plan for today

- Test-driven development
  - Brief introduction
  - Demo
- Introduction to the project

Test-Driven Development

What qualities are we looking for in our code?

- Assumption: You will write custom code as means to an end (e.g., scientific research)
- How should this tool be used, or how should users be able to use it (including yourself)? What will you want to **do** with your code?
  - Reliable results/Correct code
  - Efficient development/Implementation
  - Share/Publish code
  - Maintain/Update/Improve code
  - Easy and intuitive use
- **Critical for me: Flexible/adaptable/expandable code**
  → code as a tool to test hypotheses
- Correctness is essential, and so tests are as well

- How do we tell if code is 'good'?
  - It works!
  - Easy to Read
  - (Easily) Testable
  - Efficient
  - Easy to change
  - Simple

Test-driven development: A technique for code design

- How do we tell if code is 'good'?
  - It works!
  - Easy to Read
  - (Easily) Testable
  - Efficient
  - Easy to change
  - Simple

So how can we develop good code in the first place?

Attributes (of good and testable code):

- Modular
- Loosely coupled
- Cohesive
- Separation of Concerns
- Information Hiding

Test-driven development: A technique for code design

So how can we develop good code in the first place?

Attributes (of good and testable code):

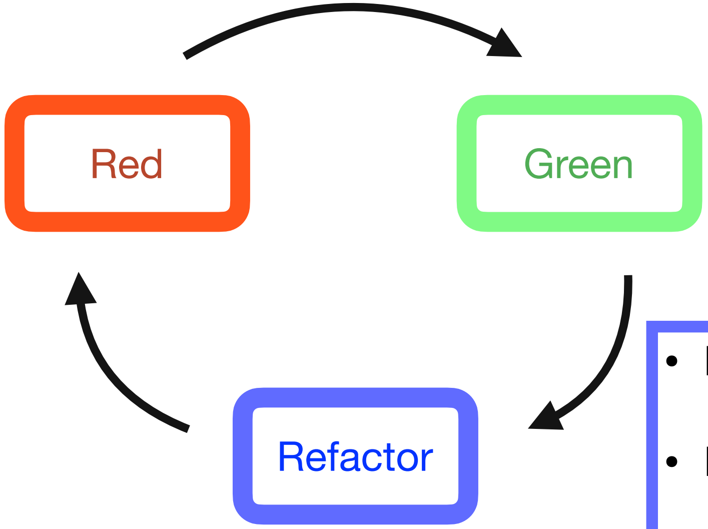
- Modular
- Loosely coupled
- Cohesive
- Separation of Concerns
- Information Hiding

- Tests are useful
  (sanity checks, easy to know if something went wrong, ..)
- But tests can be even more powerful:
  Write better code in less time
- Idea: Center the development process around testing
- TDD is not about testing as such, it's a design principle
- Goes beyond input-output checks
- Write code with testability in mind
  → enforced by writing tests first

Red-Green-Refactor

Goal: code iteratively, use tests to guide us, to help us focus and to keep it simple

- Add one simple test for one piece of functionality
- If you can't make the test simple → rethink!
- Easy tests are easy to pass
- Objective: Design interface



- Simplest modification to pass test
- Don't overthink it!
- Focus only the present test
- Objective: Implementation, not design

- Revisit code + try to improve it
- Make sure you still pass tests
- Also restructure tests!
- Objective: Improve design overall

Code smells:

- Long functions
- Repeated code
- If/else statements
- Logic simplifications
- Other design issues

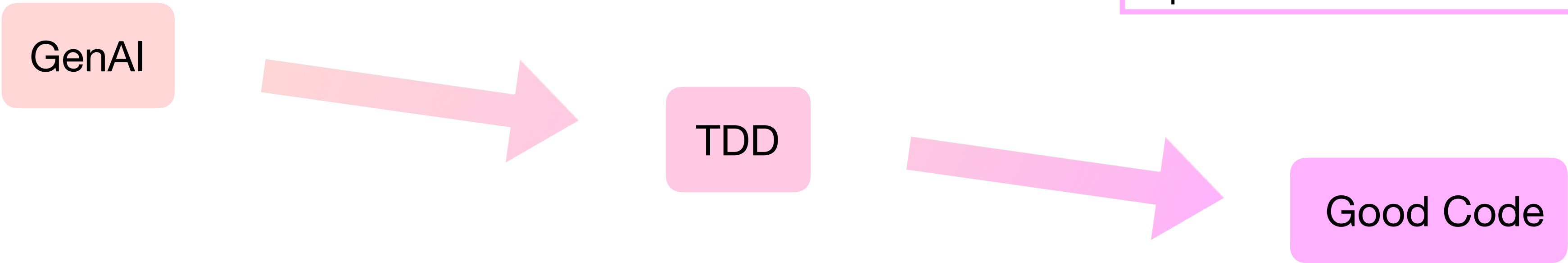
Leverage TDD & GenAI

- Works best if result can be verified quickly
- Typically: domain experience, or something like plots, styling, web interfaces
- It's very easy to generate unnecessary complexity and get stuck, if you don't know what you are supposed to do

- Key idea: Start by writing a test
- iterative, from trivial cases to full problem
- Interface is determined by **first** test!
  → Strong incentive for simplicity,
- Tests as minimal run environments for functions
  → Ideal starting point for debugging
- Focus: one change at a time, always protected by tests

- Reliable, correct
- Readable
- Testable and tested
- Efficient
- Easy to change
- Simple

Flexible/adaptable code:
→ iterative approach, instead of managing full complexity at all times
→ separation of line-by-line implementation from overall project



GenAI

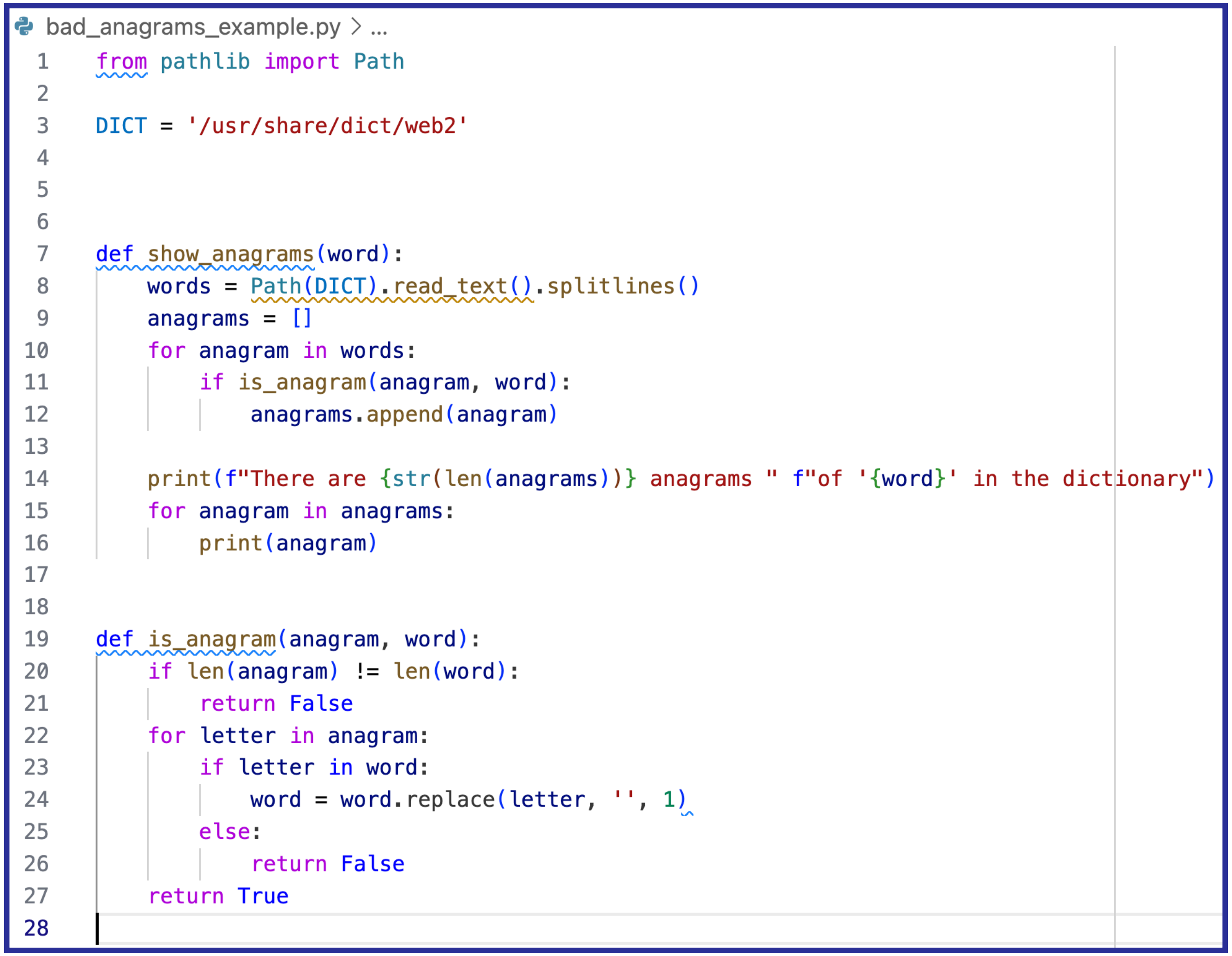
TDD

Good Code

Simple demo: Anagram solver

Task: Write a program that allows the user to find anagrams of any word in a specific file stored on disk.

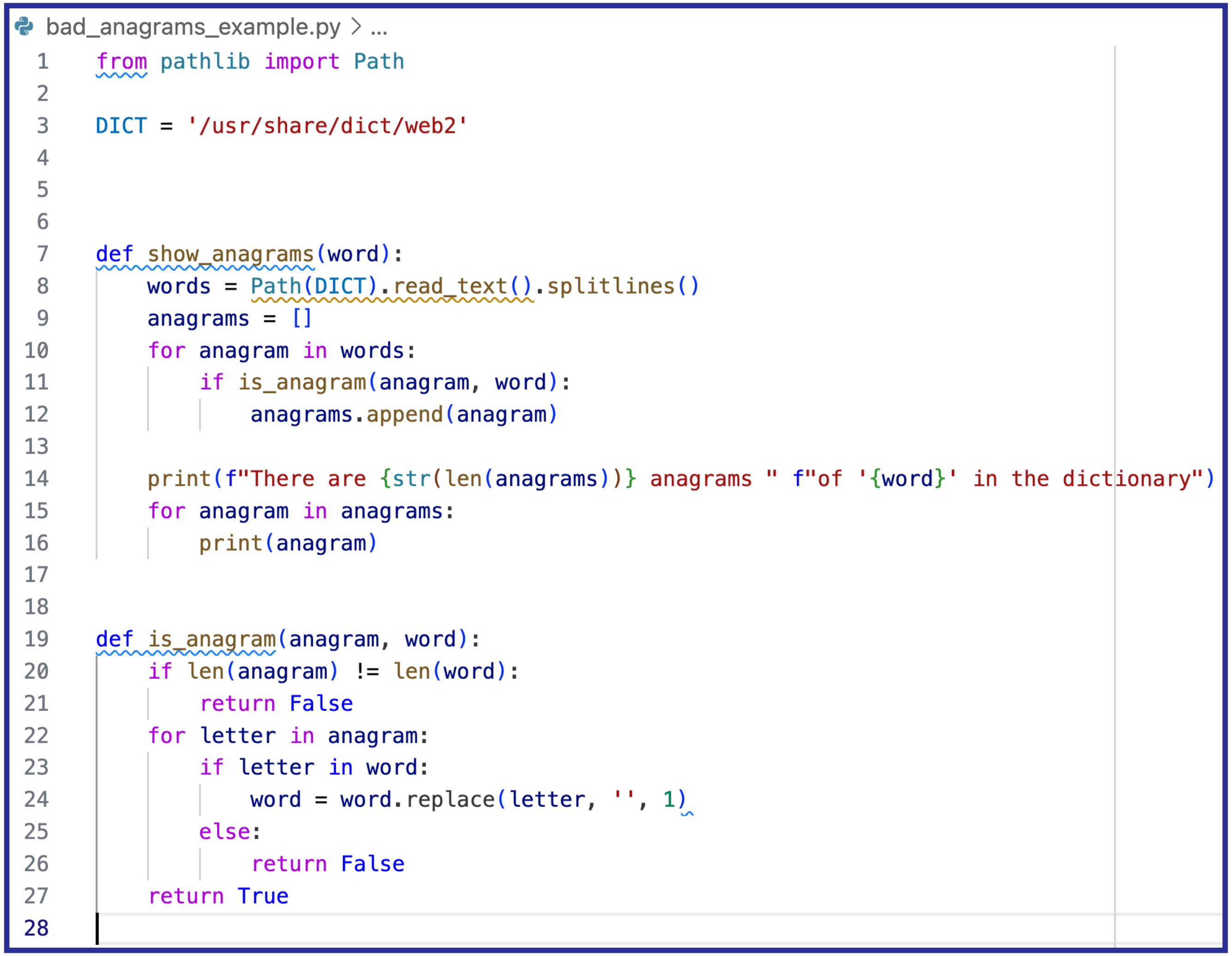
- Ignore checking if words are actually proper words
- Everything is lower case
- No spaces



```python
from pathlib import Path

DICT = '/usr/share/dict/web2'



def show_anagrams(word):
    words = Path(DICT).read_text().splitlines()
    anagrams = []
    for anagram in words:
        if is_anagram(anagram, word):
            anagrams.append(anagram)

    print(f"There are {str(len(anagrams))} anagrams " f"of '{word}' in the dictionary")
    for anagram in anagrams:
        print(anagram)


def is_anagram(anagram, word):
    if len(anagram) != len(word):
        return False
    for letter in anagram:
        if letter in word:
            word = word.replace(letter, '', 1)
        else:
            return False
    return True

```

Simple demo: Anagram solver

- Not modular
- Not cohesive
- No separation of concerns
- Little abstraction/ information hiding
- Tightly coupled

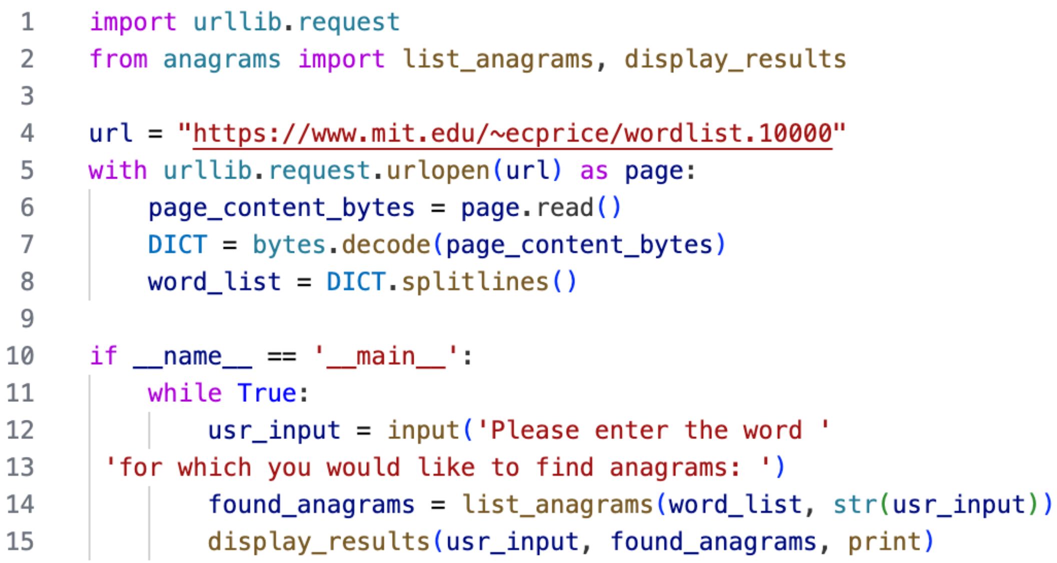


```python
from pathlib import Path

DICT = '/usr/share/dict/web2'



def show_anagrams(word):
    words = Path(DICT).read_text().splitlines()
    anagrams = []
    for anagram in words:
        if is_anagram(anagram, word):
            anagrams.append(anagram)

    print(f"There are {str(len(anagrams))} anagrams " f"of '{word}' in the dictionary")
    for anagram in anagrams:
        print(anagram)


def is_anagram(anagram, word):
    if len(anagram) != len(word):
        return False
    for letter in anagram:
        if letter in word:
            word = word.replace(letter, '', 1)
        else:
            return False
    return True

```

TDD approach to the Anagram solver

Result is clearly better:

- More modular
- Better abstraction/information hiding
- More loosely coupled
- Easily adaptable! Further improvements absolutely possible



```python
import urllib.request
from anagrams import list_anagrams, display_results

url = "https://www.mit.edu/~ecprice/wordlist.10000"
with urllib.request.urlopen(url) as page:
    page_content_bytes = page.read()
    DICT = bytes.decode(page_content_bytes)
    word_list = DICT.splitlines()

if __name__ == '__main__':
    while True:
        usr_input = input('Please enter the word '
 'for which you would like to find anagrams: ')
        found_anagrams = list_anagrams(word_list, str(usr_input))
        display_results(usr_input, found_anagrams, print)
```

Recap:

- Simple tests, easy to write/pass
- Tests encourage good design
- TDD gives immediate feedback
- GenAI speeds up the process

Take-home message:

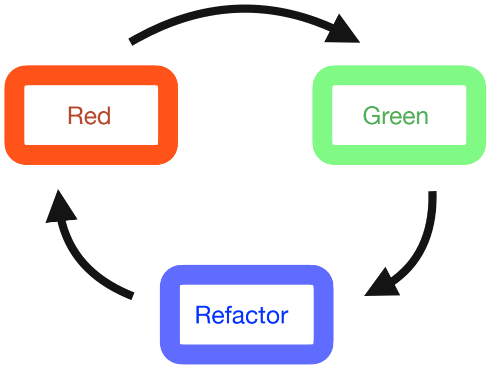
1. You should have automated tests for your code.
2. You should have testable code.

Why not write tests first? :)

Getting started on the project

- Material Online:
  - Project structure + code skeleton
  - Detailed TDD demo & 'Good Code' explanation
- Complete the functions in the `game_utils` module using TDD
  - Think about the problem structure first and what that means for design
  - Define additional helpers if necessary/appropriate!
  - First Red: Write as much code to get an Assertion Error

```
<projdir>
    |-> agents
    |   |-> agent_xy
    |   |   |-> __init__.py
    |   |   \-> xy_implementation.py
    |   |-> common.py
    |   |-> __init__.py
    |-> tests
    |   |-> __init__.py
    |   \-> test_game_utils.py
    |-> game_utils.py
    \-> main.py
```


Red
Green
Refactor