SWCON201
Opensource & Software
Development Methods and Tools

Test and Enhancement

Department of
Software Convergence

경희대학교
KYUNG HEE UNIVERSITY

Contents

- Agenda
- Class
- Summary
- Reference
- Homework

Agenda

- **Right operation** is the end of development?

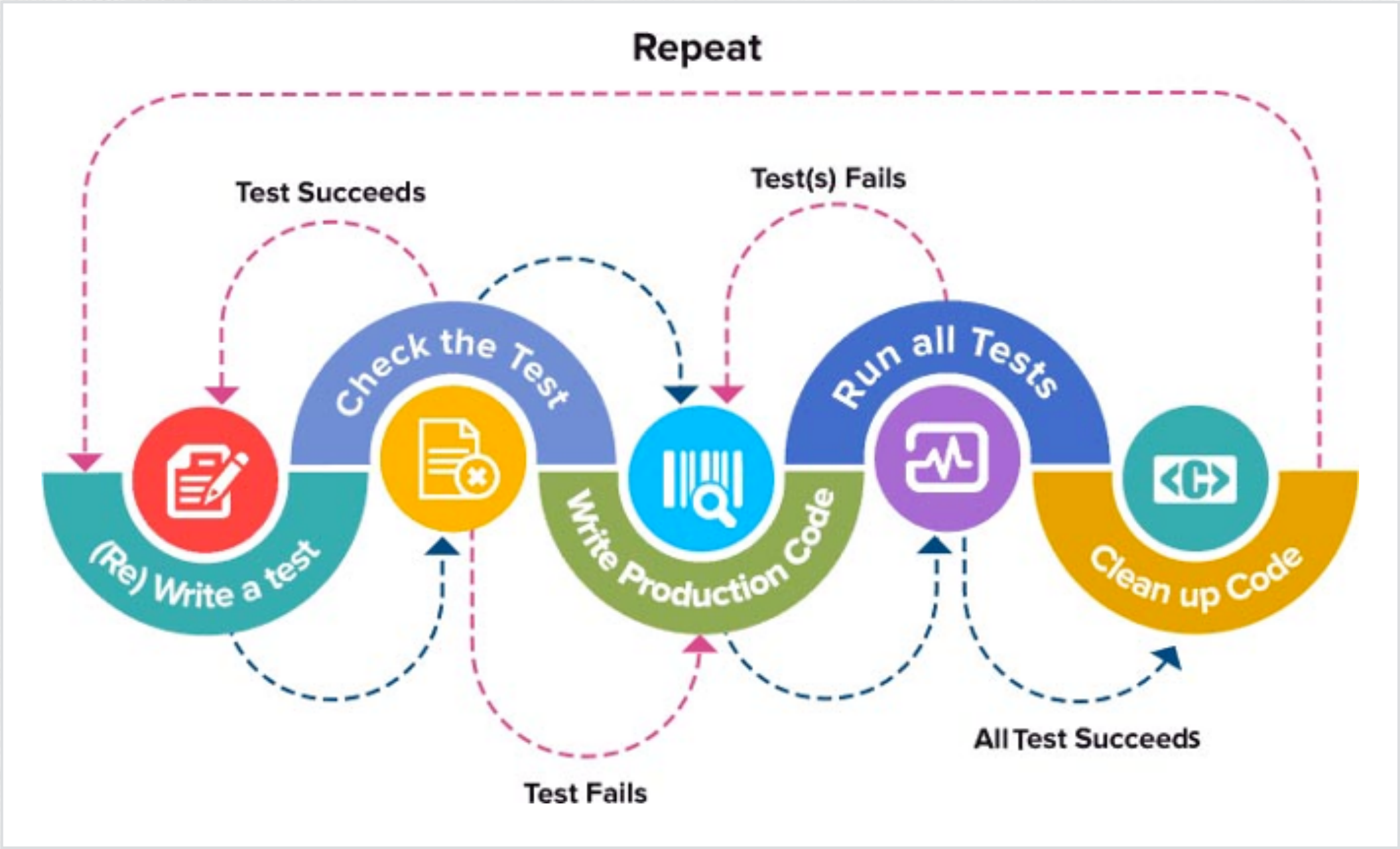
- We will talk about:
  - Reliability
  - Sustainability
  - Performance
  - Readability

Intro to Test-Driven-Development (TDD)?

- Agile in Practice: Test Driven Development

  https://www.youtube.com/watch?v=uGaNkTahrIw

What is TDD?



Reference: http://itandwebsolutionspro.com/test-driven-development-methodology.php

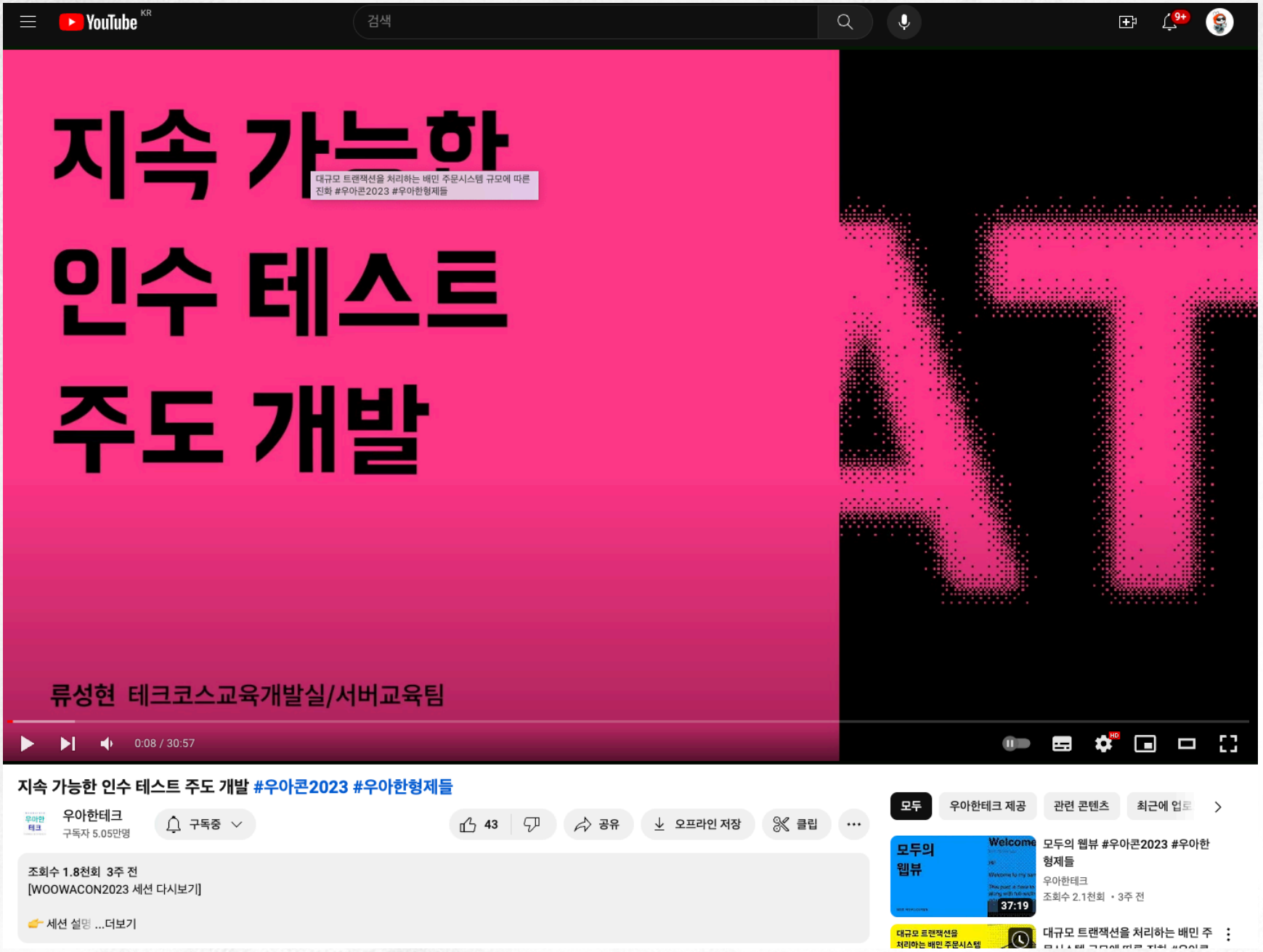
What is TDD?

- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.
- American software engineer Kent Beck, who is credited with having developed or "rediscovered" the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

What is TDD?

- Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.
- Programmers also apply the concept to improving and debugging legacy code developed with older techniques

[REF] TDD in 우아한형제들 (a.k.a 배달의 민족)



Reference: https://www.youtube.com/watch?v=birVFRb2d-E

TDD Cycle: (1) Add a test

- Each new feature begins with writing a test
- Write a test that defines a function or improvements of a function, which should be very succinct
- To write a test, the developer must clearly understand the feature's specification and requirements
- The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment
- It could be a modified version of an existing test
- This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference

TDD Cycle: (2) Run all tests and check fails

- This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass
- The new test should fail for the expected reason
- This step increases the developer's confidence in the new test

TDD Cycle: (3) Write the codes

- The next step is to write some code that causes the test to pass
- The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way
- That is acceptable because it will be improved and honed in Cycle.5 Refactoring
- At this point, the only purpose of the written code is to pass the test
- The programmer must not write code that is beyond the functionality that the test checks

TDD Cycle: (4) Run tests

- If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features
- If they do not, the new code must be adjusted until they do

TDD Cycle: (5) Refactor code

- The growing code base must be cleaned up regularly during test-driven development
- New code can be moved from where it was convenient for passing a test to where it more logically belongs
- Duplication must be removed
- Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added
- As features are added, method bodies can get longer and other objects larger
- They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle

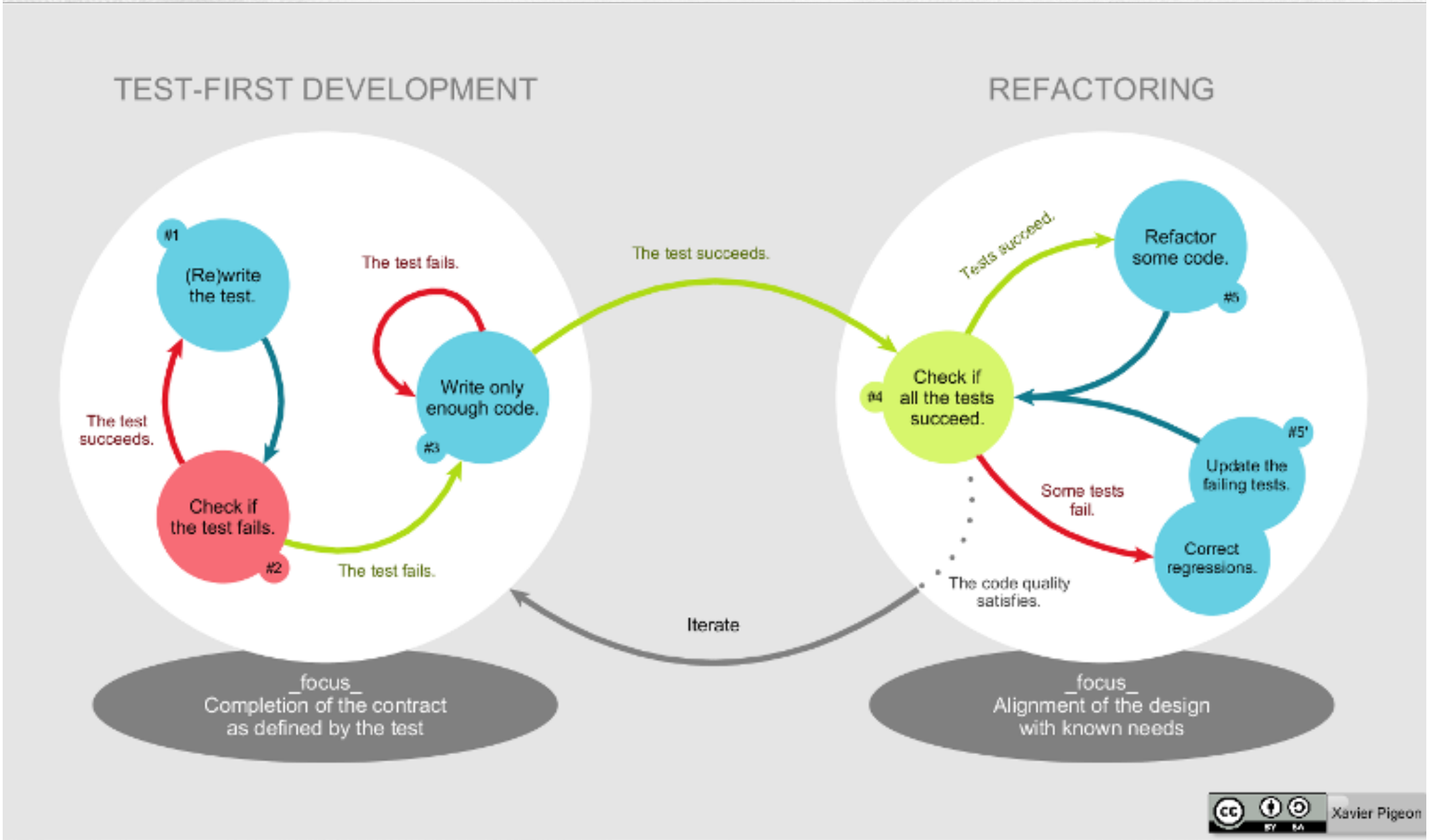
TDD Cycle: (5) Refactor code (continued)

- Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns
- There are specific and general guidelines for refactoring and for creating clean code
- By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality
- The concept of removing duplication is an important aspect of any software design
- In this case, however, it also applies to the removal of any duplication between the test code and the production code—for example magic numbers or strings repeated in both to make the test pass in Cycle.3

TDD Cycle: **REPEAT**!

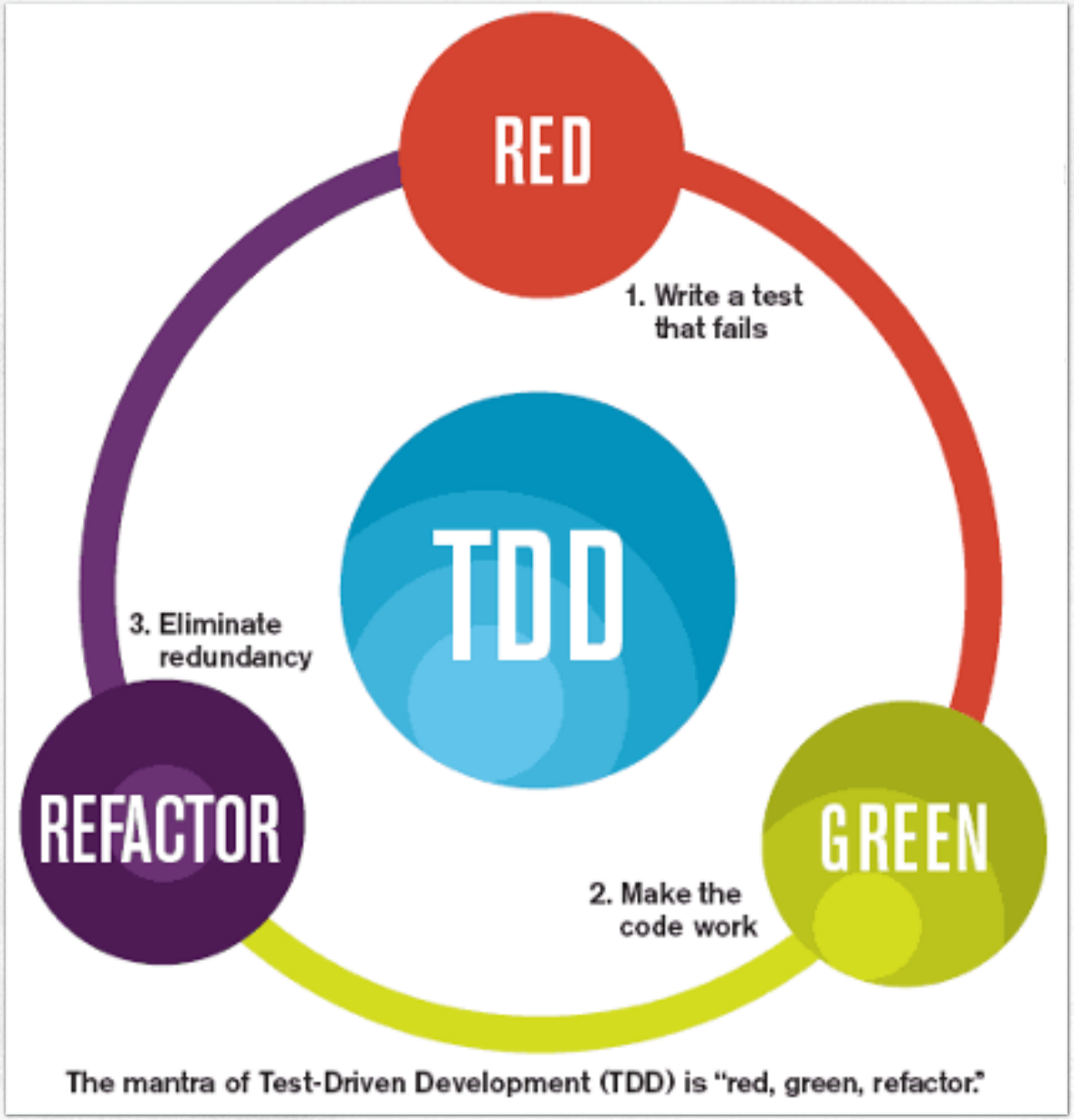
- Starting with another new test, the cycle is then repeated to push forward the functionality
- The size of the steps should always be small, with as few as one to ten edits between each test run
- If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging
- Continuous integration helps by providing revertible checkpoints
- When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

TDD Cycles in another view



Reference: https://en.wikipedia.org/wiki/Test-driven_development

TDD Cycles as Red-Green-Refactor



Reference: https://medium.com/@rinae/tdd-test-driven-development-%EB%A5%BC-%EC%97%B0%EC%8A%B5%ED%95%98%EB%A9%B4%EC%84%9C-%EC%B0%B8%EA%B3%A0%ED%95%98%EA%B8%B0-%EC%A2%8B%EC%9D%80-%ED%8C%81-10%EA%B0%80%EC%A7%80-d8cf46ae1806

TDD Principle

- "Keep the unit small"
  - For TDD, a unit is most commonly defined as a class, or a group of related functions often called a module
  - Keeping units relatively small is claimed to provide critical benefits, including:
    - Reduced debugging effort; When test failures are detected, having smaller units aids in tracking down errors.
    - Self-documenting tests; Small test cases are easier to read and to understand

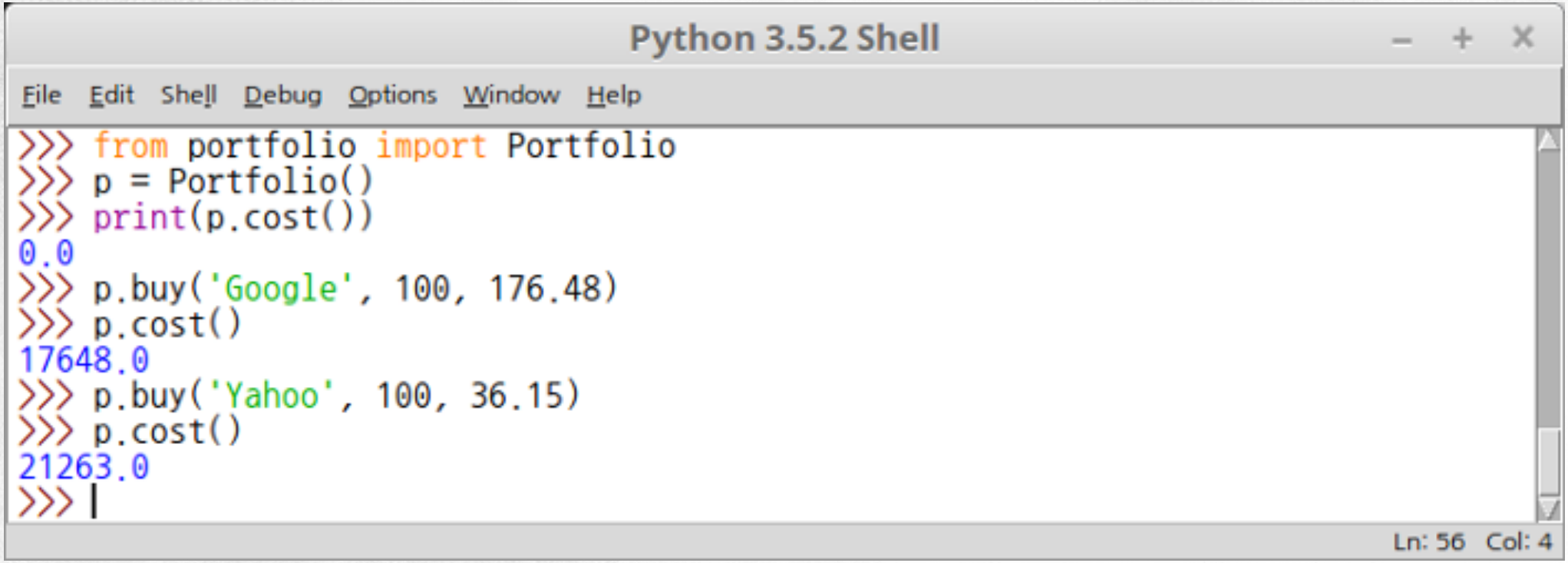
TDD in Python

- TDD Case Study using Python: Target module

```
# portfolio.py

class Portfolio(object):
    """A simple stock portfolio"""
    def __init__(self):
        # stocks is a list of lists:
        #   [[name, shares, price], ...]
        self.stocks = []

    def buy(self, name, shares, price):
        """Buy `name`: `shares` shares at `price`."""
        self.stocks.append([name, shares, price])

    def cost(self):
        """What was the total cost of this portfolio?"""
        amt = 0.0
        for name, shares, price in self.stocks:
            amt += shares * price
        return amt
```

TDD in Python

- TDD Case Study using Python: Test using Shell



Reference: https://www.slideshare.net/hosunglee948/python-52222334

TDD in Python

- TDD Case Study using Python: Test using Expected Value

```
from portfolio import Portfolio

p = Portfolio()
print("Empty portfolio cost: %s, should be 0.0" % p.cost())
p.buy('Google', 100, 176.48)
print("With 100 Google @ 176.48: %s, should be 17648.0" % p.cost())
p.buy('Yahoo', 100, 36.15)
print("With 100 Yahoo @ 36.15: %s, should be 21263.0" % p.cost())
```

```
Empty portfolio cost: 0.0, should be 0.0
With 100 Google @ 176.48: 17648.0, should be 17648.0
With 100 Yahoo @ 36.15: 21263.0, should be 21263.0
```

TDD in Python

- TDD Case Study using Python: Test using **assert** statement

```
from portfolio import Portfolio

p = Portfolio()
print("Empty portfolio cost: %s, should be 0.0" % p.cost())
p.buy('Google', 100, 176.48)
assert p.cost() == 17648.0 # Success
print("With 100 Google @ 176.48: %s, should be 17648.0" % p.cost())
p.buy('Yahoo', 100, 36.15)
assert p.cost() == 21263.0 # Success
print("With 100 Yahoo @ 36.15: %s, should be 21263.0" % p.cost())
```

```
Empty portfolio cost: 0.0, should be 0.0
With 100 Google @ 176.48: 17648.0, should be 17648.0
With 100 Yahoo @ 36.15: 21263.0, should be 21263.0
```


Reference: https://www.slideshare.net/hosunglee948/python-52222334

TDD in Python

- TDD Case Study using Python: Test using **assert** statement

```
from portfolio import Portfolio

p = Portfolio()
print("Empty portfolio cost: %s, should be 0.0" % p.cost())
p.buy('Google', 100, 176.48)
assert p.cost() == 17647.0 # Fail
print("With 100 Google @ 176.48: %s, should be 17648.0" % p.cost())
p.buy('Yahoo', 100, 36.15)
assert p.cost() == 21263.0 # Success
print("With 100 Yahoo @ 36.15: %s, should be 21263.0" % p.cost())
```

```
Empty portfolio cost: 0.0, should be 0.0
Traceback (most recent call last):
  File "/home/drsungwon/Development/msvc/HelloWorld_Python/HelloWorld.py", line 6, in <module>
    assert p.cost() == 17647.0 # Fail
AssertionError
```

TDD in Python

- TDD Case Study using Python: Test using **unittest**
- What is **unittest**?
  - Standard library
  - Automated test
  - Structured test
  - Test result report

Reference: https://www.slideshare.net/hosunglee948/python-52222334

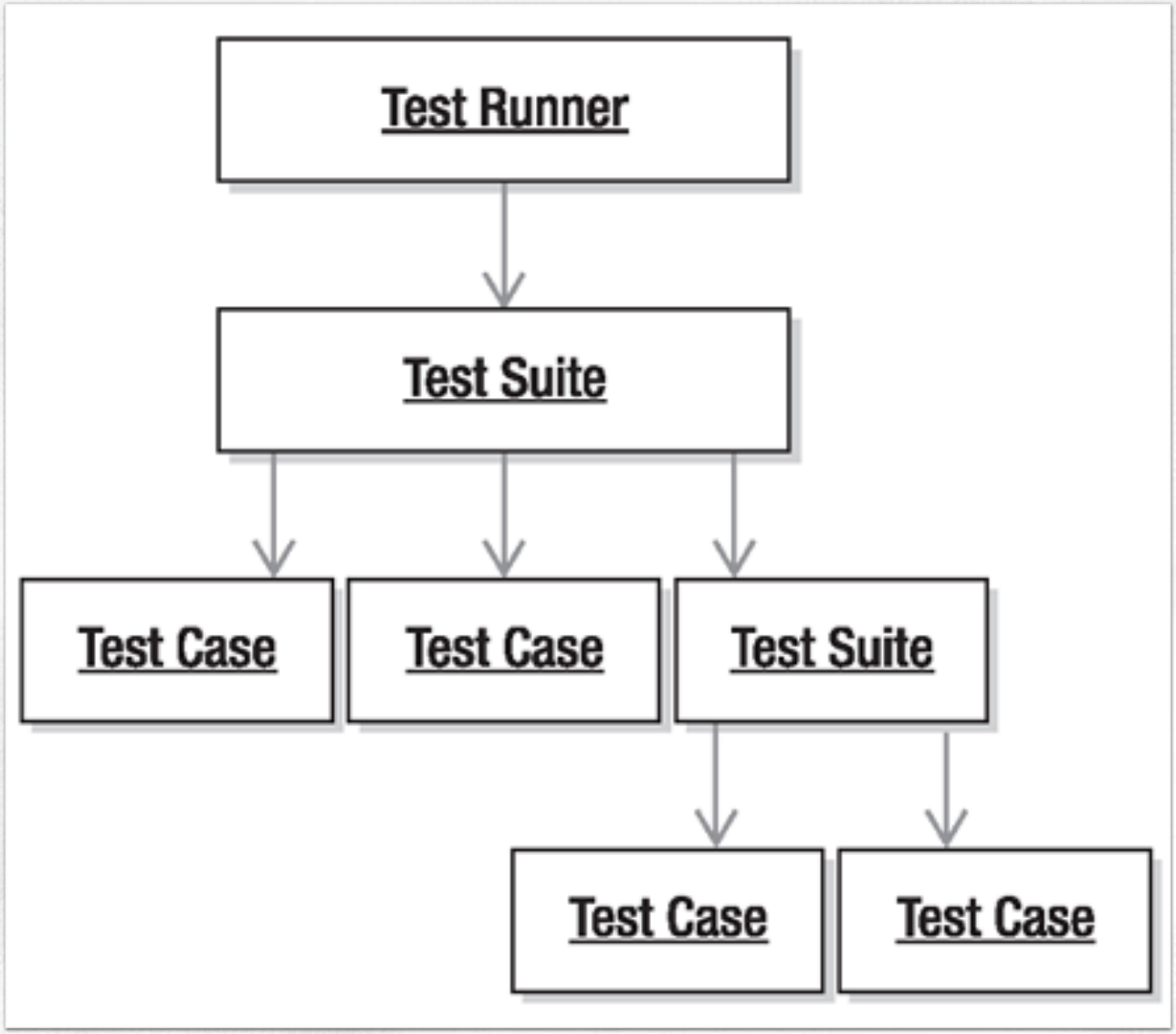
TDD in Python

- TDD Case Study using Python: Test using **unittest**
  - Test Case
    - ☑ A specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective
  - Test Suite
    - ☑ A collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviours
  - Test Runner
    - ☑ A component which orchestrates the execution of tests and provides the outcome to the user

Reference: https://www.slideshare.net/hosunglee948/python-52222334

TDD in Python

- TDD Case Study using Python: Test using **unittest**



Reference: https://www.slideshare.net/hosunglee948/python-52222334

TDD in Python

- TDD Case Study using Python: Test using **unittest**

```
# portfolio_test.py

import unittest
from portfolio import Portfolio

class PortfolioTest(unittest.TestCase):

    def test_google(self):
        p = Portfolio()
        p.buy('Google', 100, 176.48)
        self.assertEqual(17648.0, p.cost())

    def test_yahoo(self):
        p = Portfolio()
        p.buy('Yahoo', 100, 36.15)
        self.assertEqual(3615.0, p.cost())

if __name__ == '__main__':
    unittest.main()
```

TDD in Python

- TDD Case Study using Python: Test using **unittest**

```
# portfolio_test

import unittest
from portfolio i

class PortfolioTest(unittest.TestCase):

    def test_google(self):
        p = Portfolio()
        p.buy('Google', 100, 176.48)
        self.assertEqual(17648.0, p.cost())

    def test_yahoo(self):
        p = Portfolio()
        p.buy('Yahoo', 100, 36.15)
        self.assertEqual(3615.0, p.cost())

if __name__ == '__main__':
    unittest.main()
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

TDD in Python

- TDD Case Study using Python: Test using **unittest**

```
# portfolio_test

import unittest
from portfolio i

class PortfolioT

    def test_goo
        p = Port
        p.buy('Google', 100, 176.48)
        self.assertEqual(17647.0, p.cost()) # Fail

    def test_yahoo(self):
        p = Portfolio()
        p.buy('Yahoo', 100, 36.15)
        self.assertEqual(3615.0, p.cost())

if __name__ == '__main__':
    unittest.main()
```

```
F.
======================================================================
FAIL: test_google (__main__.PortfolioTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/drsungwon/Development/msvc/HelloWorld_Python/HelloWorld.py", line 9, in tes
t_google
    self.assertEqual(17647.0, p.cost())
AssertionError: 17647.0 != 17648.0

----------------------------------------------------------------------
Ran 2 tests in 0.004s

FAILED (failures=1)
```

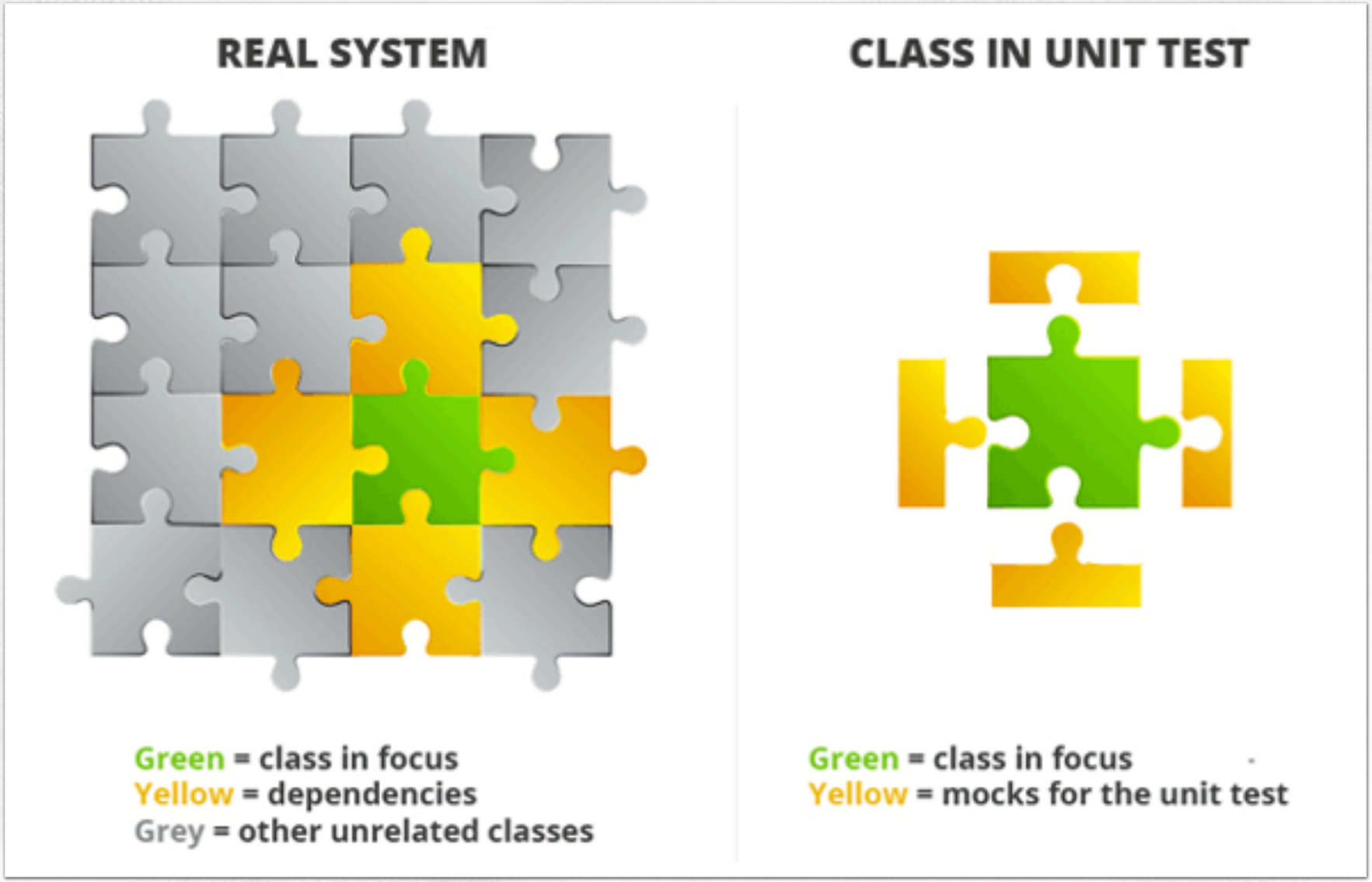
TDD in Python

- TDD Case Study using Python: Test using **unittest.mock**
  - **unittest.mock** is a library for testing in Python
  - It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used
  - mock objects are simulated objects that mimic the behavior of real objects in controlled ways
  - A programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts

Reference: https://www.slideshare.net/hosunglee948/python-52222334

TDD in Python

- TDD Case Study using Python: Test using **unittest.mock**



Reference: https://zeroturnaround.com/rebellabs/how-to-mock-up-your-unit-test-environment-to-create-alternate-realities/

TDD in C++

- TDD Case Study using C++
  - Google Test : https://github.com/google/googletest
  - CppUTest : http://cpputest.github.io/

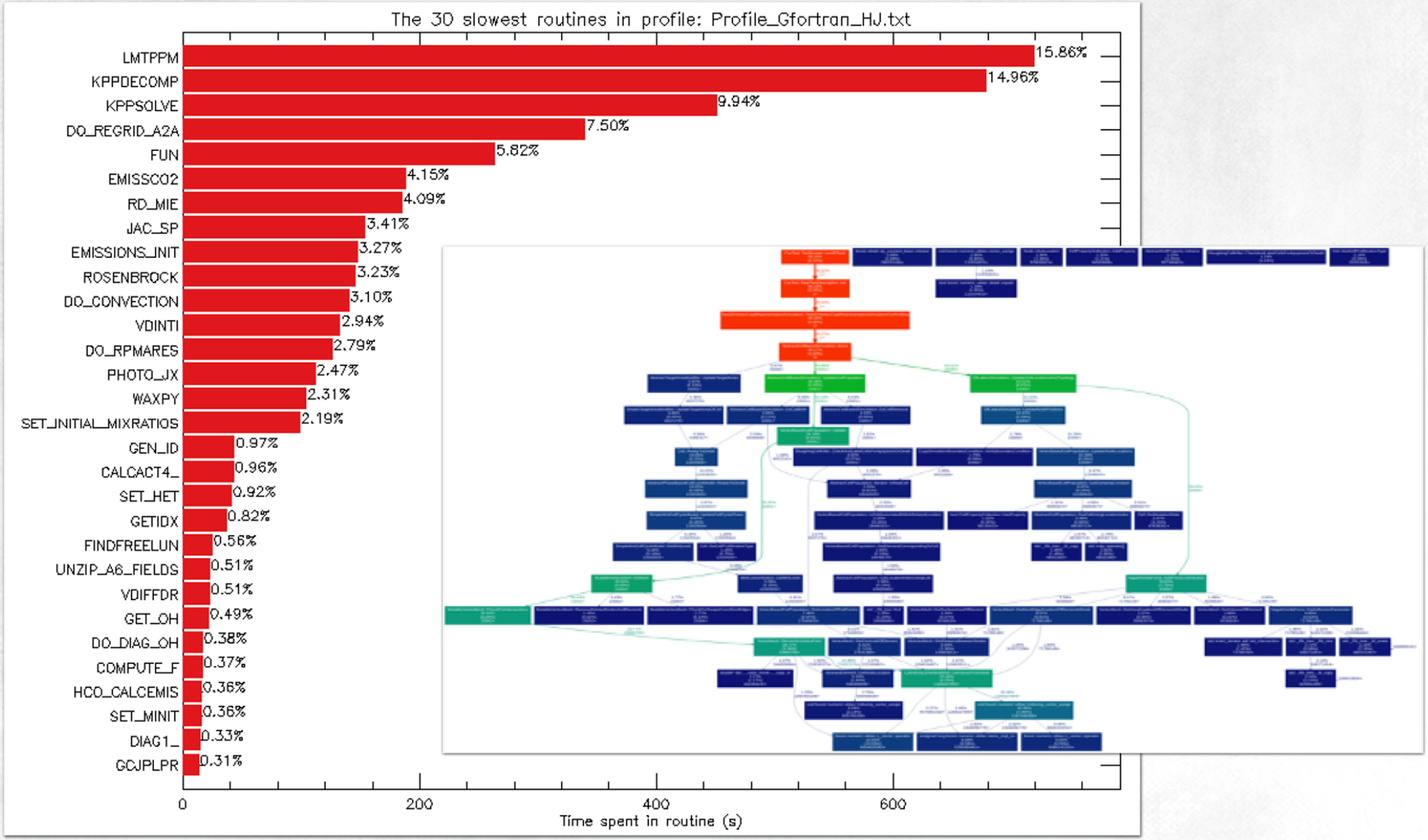
Profiling (computer programming)

- Profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls
- Most commonly, profiling information serves to aid program optimization
- Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler)

Profiler

- Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods
- Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters
- Profilers are used in the performance engineering process.

Profiler Screen Shots



Built-in Profiler in Python

`cProfile` and `profile` provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the `pstats` module.

The Python standard library provides two different implementations of the same profiling interface:

1. `cProfile` is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.
2. `profile`, a pure Python module whose interface is imitated by `cProfile`, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

참고: The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is `timeit` for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

Built-in Profiler in Python

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use `profile` instead of `cProfile` if the latter is not available on your system.)

The above action would run `re.compile()` and print profile results like the following:

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

Built-in Profiler in Python

To profile a function that takes a single ar

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")
```

(Use `profile` instead of `cProfile` if the la

The above action would run `re.compile()`

```
      197 function calls (192 primi

Ordered by: standard name

ncalls  tottime  percall  cumtime
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

The first line indicates that 197 calls were monitored. Of those calls, 192 were *primitive*, meaning that the call was not induced via recursion. The next line: `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include:

ncalls
for the number of calls.

tottime
for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall
is the quotient of `tottime` divided by `ncalls`

cumtime
is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall
is the quotient of `cumtime` divided by primitive calls

filename:lineno(function)
provides the respective data of each function

When there are two numbers in the first column (for example `3/1`), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

3rd-party Profiling Tools for Python

- Profiler Visualization Opensource for Python

RunSnakeRun

RunSnakeRun is a small GUI utility that allows you to view (Python) cProfile or Profile profiler dumps in a sortable GUI view. It allows you to explore the profiler information using a "square map" visualization or sortable tables of data. It also (experimentally) allows you to view the output of the Meliae "memory analysis" tool using the same basic visualisations.

Features

RunSnakeRun is a simple program, it doesn't provide all the bells-and-whistles of a program like KCacheGrind, it's intended to allow for profiling your Python programs, and just your Python programs. What it does provide, for profile viewing:

- sortable data-grid views for raw profile information
  - identity: function name, file-name, directory name
  - time-spent: cumulative, cumulative-per, local and local-per time
  - overall data-grid view
  - (all) callers-of-this-function, (all) callees-of-this-function views
- squaremap view of call tree
  - size proportional to amount of time spent by the given parent in the given function
- squaremap view of packages/modules/functions
  - size proportional to time spent in each package/module/function
- basic navigation (home, back, up)

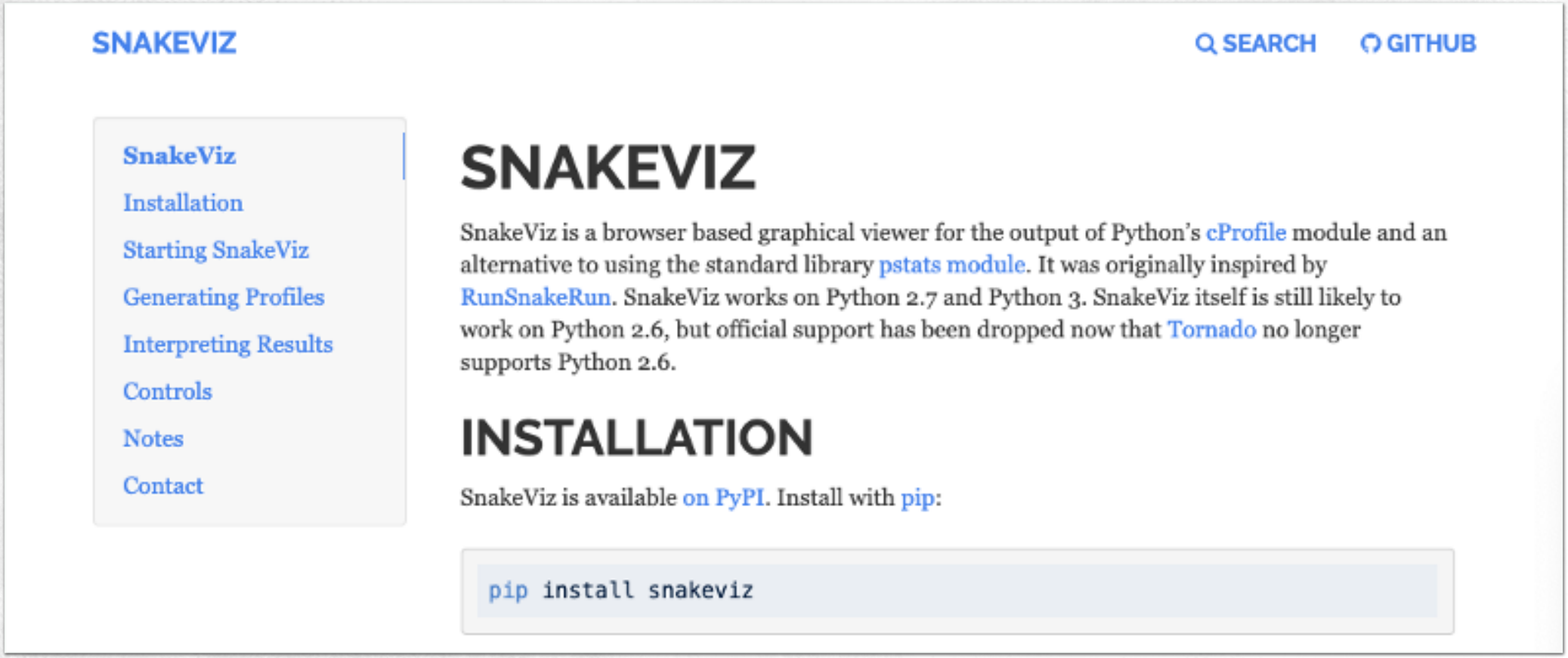
For Meliae memory-dump viewing, it provides:

- sortable data-grid views
- squaremap of memory-usage
- basic navigation

Reference: http://www.vrplumber.com/programming/runsnakerun/

3rd-party Profiling Tools for Python

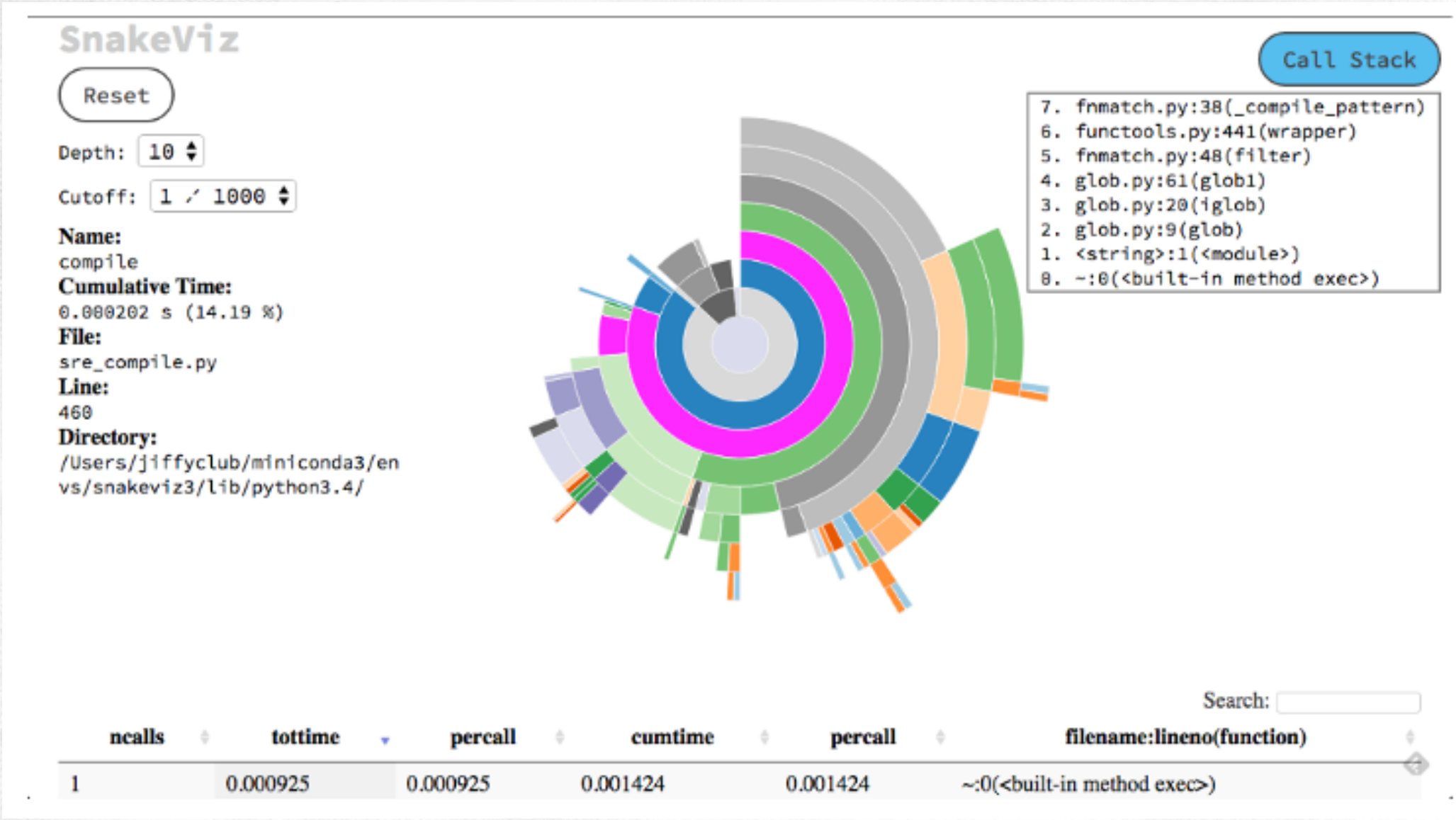
- Profiler Visualization Opensource for Python



Reference: https://jiffyclub.github.io/snakeviz/

3rd-party Profiling Tools for Python

- Profiler Visualization Opensource for Python



Reference: https://penandpants.com/tag/python-snakeviz-profiling/

FYI

- Profiler Softwares
  - https://en.wikipedia.org/wiki/List_of_performance_analysis_tools

Python 게시판

2018년 8월 19일

제목	[추천] 종강후 학습	2022-02-27 11:02
작성자	drsungwon	

수업 게시판에 이성원 교수가 작성한, "종강 후 파이선 공부 이어가기"를 위한 글 입니다.
한 학기 동안 배운 파이썬의 문제는 무엇이고, 어떻게 써야 하는지 등에 대한 RE:start의 계기가 되기 바랍니다.

"Poor Python" [바로가기]

Python 언어를 많이 쓰는 이유는 쉽고 편한 것이 가장 큰 비기술적 장점입니다. 다만, 성능면에서 워낙 느리고 컴퓨터 자원을 비효율적으로 쓰는 단점이 있습니다. 따라서, Python 언어를 배우는 웹파이선프로그래밍 수강자 혹은 수강을 마치고 C++을 배우는 소융학과 전공자는 Python에 대한 올바른 이해가 필요합니다. 이를 설명한 글 입니다.

"Google & NHN Python Guide" [바로가기]

구글/NHN(네이버) 등에서 Python으로 프로그래밍을 할 때, 지켜야 하는 것과 하지 말아야 하는 것을 구글이 자체 정리한 Coding Guideline 문서 입니다.

Coding Guideline (or Convention)

- A set of guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a program written in that language
- Usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc
- Guidelines for software structural quality
- Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier

Reference: https://en.wikipedia.org/wiki/Coding_conventions

Coding Guideline (or Convention) [continued]

- Coding conventions are only applicable to the human maintainers and peer reviewers of a software project
- Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual. Coding conventions are not enforced by compilers

Reference: https://en.wikipedia.org/wiki/Coding_conventions

Coding Guideline reference

- Python Style Guidel [Official]
  - https://peps.python.org/pep-0008/
- Google Coding Guideline [Python, C++, HTML/CSS, etc]
  - https://github.com/google/styleguide
- NHN Coding Guideline [HTML/CSS]
  - https://nuli.navercorp.com/data/convention/NHN_Coding_Conventions_for_Markup_Languages.pdf
- Microsoft Coding Guideline Checker
  - https://blogs.msdn.microsoft.com/vcblog/2017/08/11/c-core-guidelines-checker-in-visual-studio-2017/

Summary

- There is no single Golden-way.
- TDD is welcomed by many IT companies @ now.

- Start from unittest & mock.
- Performance enhancement via dynamic analysis.

Reference

[Python unittest]
https://docs.python.org/3/library/unittest.html
https://docs.python.org/ko/3/library/unittest.html

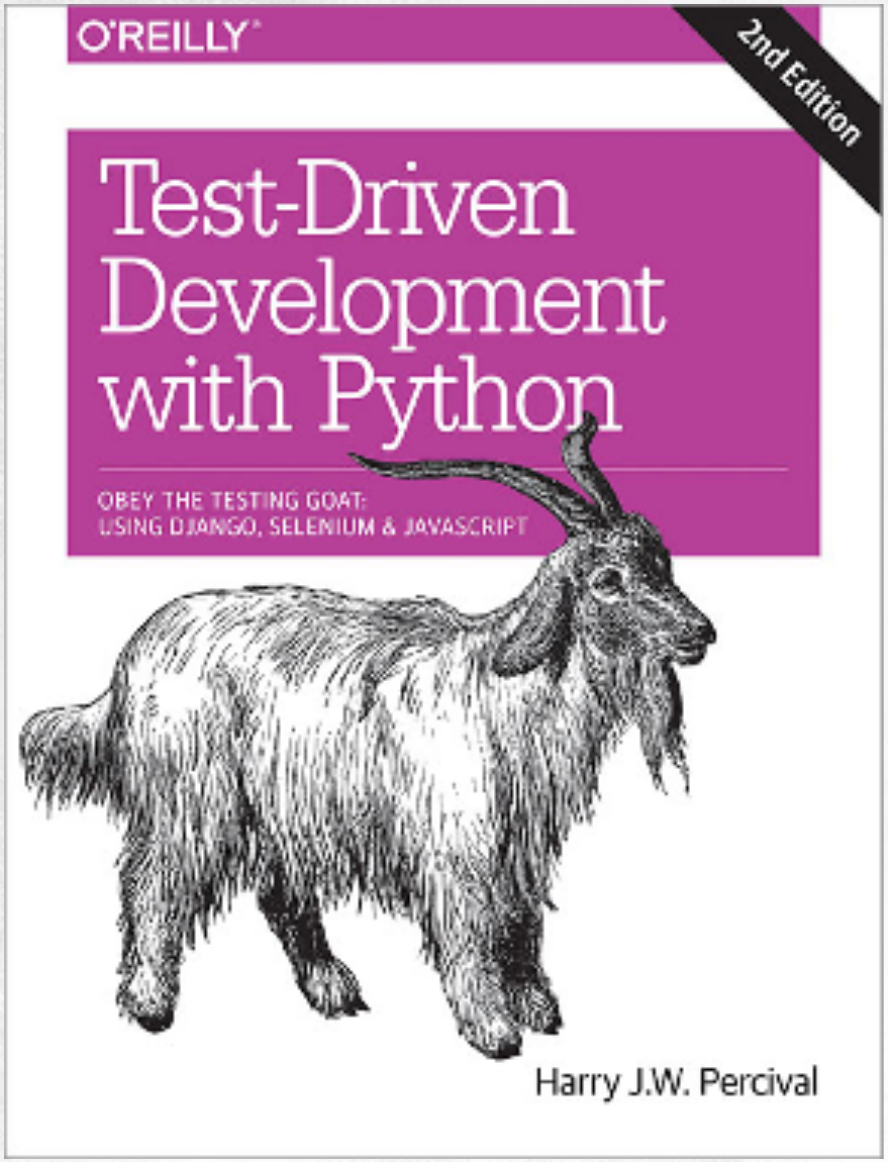
[Python unitest.mock]
https://docs.python.org/3/library/unittest.mock.html
https://docs.python.org/ko/3/library/unittest.mock.html

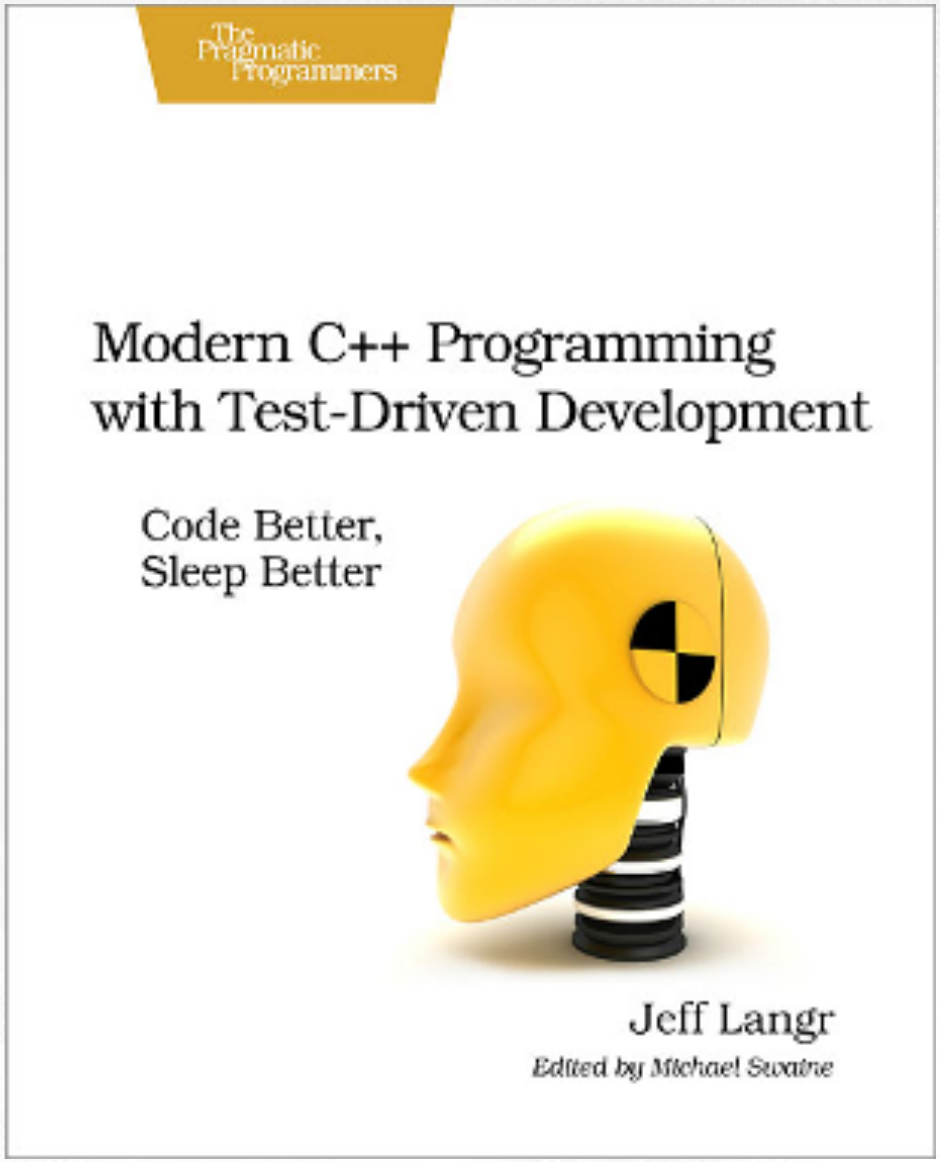
[testdriven.io]
https://testdriven.io/
https://testdriven.io/blog/modern-tdd/

[realpython]
https://realpython.com/
https://realpython.com/search?q=tdd

[PyTest]
https://docs.pytest.org/en/7.2.x/
https://docs.pytest.org/en/7.2.x/getting-started.html

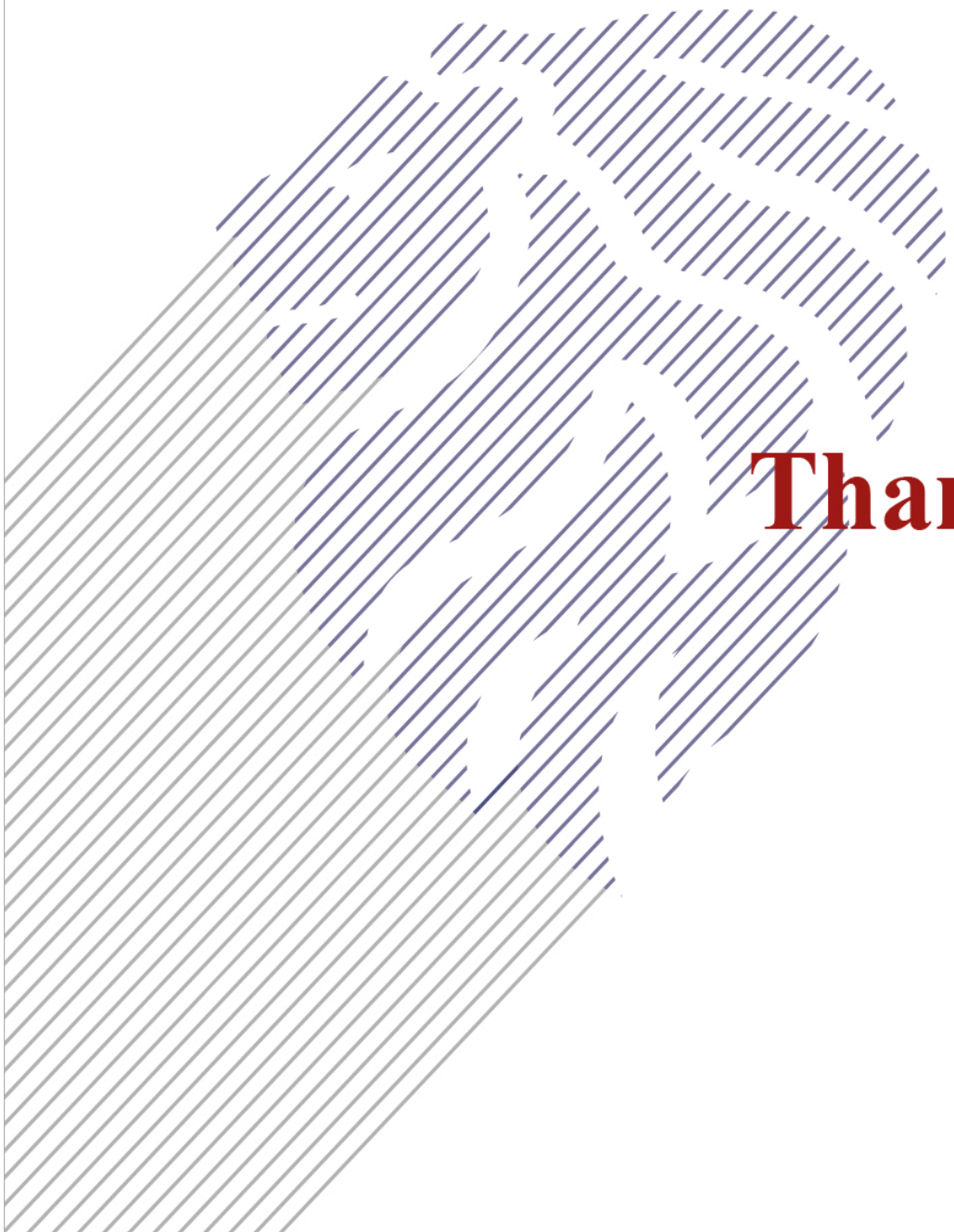
Reference


O'REILLY
2nd Edition
Test-Driven Development with Python
OBEY THE TESTING GOAT:
USING DJANGO, SELENIUM & JAVASCRIPT
Harry J.W. Percival


The Pragmatic Programmers
Modern C++ Programming with Test-Driven Development
Code Better, Sleep Better
Jeff Langr
Edited by Michael Swaine

Homework

- Try **unittest** & **unittest.mock** for your own SW.



Thank you

경희대학교
KYUNG HEE UNIVERSITY