
# Code Craft Kick-Start

## Exercises & Links

## The Course Book

http://codemanship.co.uk/tdd_jasongorman_codemanship.pdf

## Codemanship Video Playlists

Python

https://www.youtube.com/playlist?list=PL1tIFPlmF4ykpjOJKVDwYkpBIeXsoak6S

C#

https://www.youtube.com/playlist?list=PL1tIFPlmF4ylQpWloxdDP5nK_EDmESbqN

JavaScript

https://www.youtube.com/playlist?list=PL1tIFPlmF4ykUGElb7NIUbaI5r4A8DuCQ

Java

https://www.youtube.com/playlist?list=PL1tIFPlmF4ymILyzmcQmimv7RHBlY3N15

# TDD

## Exercise #1

**The Mars Rover Kata (simplified)**

Test-drive code to drive a rover over the idealised surface of Mars, represented as a 2-D grid of x, y coordinates (e.g., [4, 7]).

The rover can be "dropped" on to the grid at a specific set of coordinates, facing in one of four directions: North, East, South or West.

Sequences of instructions are sent to the rover telling it where to go. This sequence is a string of characters, each representing a single instruction:

- R = turn right
- L = turn left
- F = move on square forward
- B = move one square back

The rover ignores invalid instructions.

*How to tackle this exercise:*

1. *Make a test list for your rover based on these requirements*
2. *Work through your test list on test case at a time, remembering to Red-Green-Refactor with each test*
3. *When refactoring, don't forget that test code has to be maintained, too!*

Links

The complete Mars Rover TDD kata is described at https://kata-log.rocks/mars-rover-kata

Videos

The 3 Steps of TDD

https://www.youtube.com/watch?v=vBJM5pzBfhY

Good Habits For TDD

https://www.youtube.com/watch?v=oF5jTHSV28s

What Should We Test?

https://www.youtube.com/watch?v=Ji2KKShNKic

# Refactoring
# Exercise #2

**Mars Rover refactored**

Carefully refactor your Mars Rover code to remove code smells and make it as easy to understand and easy to change as you can.

If your code isn't where you'd like to start (e.g., if it's already perfect!), clone our Python solution files using the link below.

*How to tackle this exercise:*

1. *Use version control, and commit whenever your tests pass.*
2. *If you mess up, revert the code to the previous working version.*
3. *Make sure you're seeing your tests pass frequently.*
4. *Try to use automated refactorings and other IDE shortcuts where possible.*
5. *If you're using a stripped-down editor with limited refactoring support, be extra disciplined about taking baby steps, performing one refactoring (Rename, Extract Method, Introduce Parameter, etc) at a time and making sure you run the tests whenever you believe they should pass.*

Links

Source code - https://github.com/jasongorman/python_refactoring

Videos

Refactoring Python

https://www.youtube.com/watch?v=HW2lLotb6_o

PyCharm's Automated Refactorings

https://www.youtube.com/watch?v=lmvF6Y5vctc

# Design Principles

## Exercise #3

**CD Warehouse**

Test-drive some code that satisfies the following requirements:

```
Customers can buy CDs, searching on the title and the
artist. Their credit card payment is processed by an external provider.
Record labels send batches of CDs to the warehouse. Keep a
stock count of how many copies of each title are in the warehouse.

Customers can leave reviews for CDs they've bought through the
warehouse, which gives each title an integer rating from 1- 10 and
the text of their review if they want to say more.
```

*How to tackle this exercise:*

1. *Identify the user actions we need to test, and make a Test List for each of them to drive your TDD*
2. *Run the requirements through a tag cloud generator to build a customer word list to use as inspiration for names in your code*
3. *At every refactoring step, carefully review your code to ensure it follows the principles of Simple Design, Modular Design and that it has a strong conceptual correlation with the customer's language*
4. *Don't forget to apply the disciplines of TDD and refactoring that we've learned today!*

Links

Tag Cloud Generator - https://www.wordclouds.com/

Videos

Simple Design in Python

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

Modular Design in Python

https://www.youtube.com/watch?v=cYWfk6XBxtw

# Test Doubles

## Exercise #4

**CD Warehouse++**

As always happens, our customer has added some requirements for a CD Warehouse:

```
When someone buys a CD, we need to notify the charts of the sale, telling
them the artist and title of the CD bought, and how many copies were sold.

We offer a price guarantee for albums that are in the Top 100. We guarantee
to beat the lowest competitor's price by £1.
```

*How to tackle this exercise:*

1. *Use test doubles to enable you to retrieve test data (such as the CD's current chart position and the lowest competitor price) and to test that messages were sent to external systems*
2. *Don't forget to work backwards from assertions (or the mock object equivalent of an assertion), and don't declare interfaces until the test doubles require them.*

Links

unittest.mock guide - https://docs.python.org/3/library/unittest.mock.html

Videos

Scaling TDD with Stubs & Mocks

https://www.youtube.com/watch?v=L2Gtln3IokQ

# Specification By Example

## Exercise #5

**Solve A Problem With Software**

In your pair, specify a set of features for a simple software product you believe could solve a real-world problem.

*How to tackle this exercise:*

1. *Elect someone in your pair as the **customer***
2. *Working with your customer, define a real-world **problem** (NOT A SOLUTION!) in a single problem statement (e.g., it's very hard to find good drummers in my local area)*
3. *Envision a single **headline feature** that you imagine will solve the problem (e.g., drummers within an hour of me are listed in order of proximity)*
4. *List up to 5 **supporting features** that would be required to make the headline feature work (e.g., drummers can register their location, travel time is calculated, etc)*
5. *Working one feature at a time, work with the customer to precisely capture how they envision the feature working in key **scenarios** using **examples***
6. *Do not write any code. This can all be captured in e.g., a spreadsheet*

Links

Videos

Specification By Example for Python

https://www.youtube.com/watch?v=M-Rl4CIACLw

# Outside-In TDD

## Exercise #6

**Solve A Problem With Software II**

In a test-driven way, implement software that passes your customer's tests.

*How to tackle this exercise:*

1. *Work one feature, one scenario and one example at a time*
2. *Create a test fixture for each scenario. Write a unit test for each unique outcome of each example. Copy and paste the data from your example directly into the automated test.*
3. *Name your automated tests to easily identify which feature, which scenario and which outcome they are testing.*
4. *If a scenario has multiple examples, consider using a parameterized test.*
5. *Work outside-in, stubbing or mocking any dependencies you wish to make separate concerns (e.g., a database, or a web service, or another component of your system you wish to have its own tests later)*
6. *When a layer of your design is passing its tests, moving inwards to TDD the next layer (the parts you stubbed and mocked), and keep moving inwards until you have a working end-to-end core implementation that excludes any external dependencies.*
7. *Refactor when your code passes the tests but doesn't conform to the rules of Simple Design, Modular design or Conceptual Correlation. Keep extracted elements (e.g., helper methods or classes) hidden from the test code where possible.*
8. *Use version control to ensure you can easily get back to a working version if you mess up.*
9. *DON'T FORGET WHO YOUR CUSTOMER IS! Any questions about requirements should be directed to them.*

Links

Videos

Inside-Out TDD

https://www.youtube.com/watch?v=FerGg9E3p1w

Outside-In TDD

https://www.youtube.com/watch?v=p0UooferVbE

# Continuous Delivery

## Exercise #7

**Evil FizzBuzz**

Working as a team, produce a single simple executable program that satisfies the following requirements:

- Generate a list of integers from 1 to 100
- Substitute any integers divisible by three with “Fizz”
- Substitute any integers divisible by five with “Buzz”
- Substitute any integers divisible by three and five with “FizzBuzz”
- Concatenate “Whizz” to the end of the output strings for integers that are prime numbers (e.g., “2Whizz” for 2, “FizzWhizz” for 3, “BuzzWhizz” for 5)
- Display the resulting FizzBuzzed list as a single comma-delimited string (i.e., “1,2Whizz,FizzWhizz…” etc

The Rules:

- Each pair takes one of these requirements, and can **only write the code (including tests) for that requirement**
- The team must work on the same shared GitHub repository, integrating their code into it
- The team must set up a **Continuous Integration server** for their code that builds and tests it on every check-in
- Once the build goes green for the first time, it must not go red again, or the exercise will be over
- The finished program must be *demonstrable on the instructor’s computer*
- The final check-in must take place before **60 minutes** have elapsed from the start of the exercise

Links

Videos

CI/CD in Python

https://www.youtube.com/watch?v=g4ZaSsSyBrg

# Property-Based Tests

## Exercise #8

**Property-Based Fibonacci Numbers**

Test-drive a simple algorithm that calculates Fibonacci numbers at specific positions in the Fibonacci sequence.

Refactor your tests into property-based tests to check that your algorithm works for the first 50 Fibonacci numbers

Links

https://en.wikipedia.org/wiki/Fibonacci_number

Videos

Refactoring to Property-Based Tests

https://www.youtube.com/watch?v=jndUAg2q6G0

# Mutation Testing

## Exercise #9

Using an appropriate mutation testing tool for your programming language, apply mutation testing to one of the code bases you have created on this course and check your test suite's mutation coverage. If any code is determined to not be covered by your tests, examine why and change or add tests to improve the mutation coverage.

Links

MutPy Home Page for Python

https://pypi.org/project/MutPy/

Videos

Mutation Testing in Python

https://www.youtube.com/watch?v=LjiwZqdYkA8

# Continuous Inspection

## Exercise #10

In your pair, discuss and agree on a coding standard that you would like to apply to your code.

Write real examples to illustrate code that doen't meet your standard, and code that just meets your standard (e.g., if you agree methods or functions should have no more than 5 branches, write an example with 6 and an example with 5).

Define how your coding standard would be applied in a Continuous Inspection quality gate:

- Would check-ins that break the standard be rejected (i.e., break the build)? Should a peer review be required? Or just a warning?

Links

Videos

Python Continuous Inspection

https://youtu.be/wQ-r8JimJ-4