Python Workshop

28. May 2022

Zacharias Dimitrakoudis

Charis Moutafidis

Agenda

- Introduction
- Iterations:
  - Discuss requirements
  - Write code
  - Write tests
  - Refactor code (if needed)
- Open discussion
- QnA

Introduction

What is this workshop about:

- Using OOP with just plain Python
- Testing and improving code
- Working on a scalable/maintainable environment

What is this workshop not about:

- Learning Python
  - The time is strict, so we cannot cover everything
  - We assume that the audience has at least the basic knowledge
- Learning specific frameworks/technologies integrated with Python
  - New technologies come and go very often
  - There is plenty of material and courses specialized for frameworks or technologies

Iteration #1

Requirements:
Welcome to the workshop! You are members of a special engineering team, working on a Banking system. You are asked to implement a plain Python program (use only built in functionalities) that manages **Bank accounts** and **Transactions**. Every bank account is restricted by a credit limit and belongs to a single customer.

BankAccount:
- owner: str
- credit_limit: int >=0

Tasks:
- Create a BankAccount class (file: BankAccount.py), with the necessary properties
- Write the following unit tests (in file: test_bank_account.py):
  - can create an object of this class, with values: owner=”John Doe”, credit_limit=5000
  - cannot create a bank account with negative credit_limit: owner=”John Doe”, credit_limit=-50
- Run tests (cmd: python3 -m unittest)

Notes:
- In cases of a forbidden value (e.g. negative credit_limit), the program should raise an exception. Suggested classes are RuntimeError or Exception.

Iteration #2

Requirements:
You just got the new requirements from the corresponding department, that you should introduce set a maximum amount to the credit limit, at 500.

Tasks:
- Restrict the credit_limit at 500 in BankAccoun class constructor
- Run the existing unit tests to see if something breaks
- What do you observe?

Refactor in Iteration #2

Notes:
One of the existing unit tests just failed, because the new change we introduced is not covering the previous requirements.
This is called a breaking change by the way.

Tasks:
- Fix the breaking test by setting any value <= 500, e.g. 250
- Include the following test case:
  - cannot create a bank account with credit_limit over 500: owner=”John Doe”, credit_limit=501

Iteration #3

Requirements:
It’s time to integrate transactions in your program. A transaction’s amount could be either positive (deposit), either negative (withdraw), but never 0.

Transaction:
- amount: float, positive or negative, != 0
- notes: str, optional

Tasks:
- Create a Transaction class (file: Transaction.py), with the necessary properties
- Write the following unit tests (in file: test_transaction.py):
  - can create an object of this class, with values: amount=50
  - cannot create a transaction with 0 amount: amount=0
- Run tests

Iteration #4

Requirements:
Transactions should be related with the accounts. Every transaction belongs to 1 bank account, so a bank account can contain multiple transactions. You should be able to add transactions to your BankAccount objects. The bank account's total balance should be calculated from a method.

BankAccount:
- owner: str
- credit_limit: int >=0
- transactions: list

Tasks:
- Include the property transactions in BankAccount class, with default value an empty list
- Implement get_balance method in BankAccount class, that will calculate balance based on transactions
- Implement add_transaction method in BankAccount class, that will add transactions in the bank account
- Write the following unit tests (in file: test_bank_account.py):
  - can can deposit on bank account: credit_limit=100, amount=200
  - can can withdraw from bank account: credit_limit=100, amount=-100
  - cannot withdraw from bank account exceeding credit limit: credit_limit=100, amount=-200
- Run tests
- What do you observe?

Refactor in Iteration #4

Notes:
If the last one test just failed, it’s because the code is not covering the corresponding requirement. If so, then don’t worry about it because that was exactly the plan: to let the failing tests guide us on how to refactor our code (it’s the basic principle of Test Driven Development). But to achieve this, you need to fully cover all possible cases with tests!

Tasks:
- Include a condition to restrict the withdraw amount in add_transaction method, based on the balance
- Run tests again

Open discussion

What is OOP

How to test the code

How to maintain code

QnA

Thank you for attending in this Workshop!

PRODYNA