DANIELA TRIPON
HOMEWORK WEEK 4

TASK 1 (Git and GitHub)

Question 1

Complete definitions for key Git & GitHub terminology

GIT WORKFLOW FUNDAMENTALS

- Working Directory
    - Is also called the working tree and is the state in which a project is during one version, after a single checkout and includes any modifications to the files. It holds the files in a database stored in Git directory and pulls them on disk for developers to work on them. The files can be in two states untracked (new files that haven't been committed), or tracked (existing files, known to Git due to previous commits, and can be in any of the states: staged, modified, or committed). "git status" is the command used to check the state of a file.
- Staging Area
    - It is also known as index in Git terminology and is found in Git directory. This is a file that holds information about the changes occurred in the project and which are going to be committed.
- Local Repo (head)
    - Local repository is the computer on which we are working. We can work on the local repository without internet as well, but other people won't be able to see the changes unless we push them to the main branch (we need internet to do this).
    - HEAD references to the branch on which the last commit was submitted. This reference can point to a feature branch, checked out from the main branch to test different functionalities, or to the main branch itself, depending on where the last commit happened. Head can be detached and moved to any point in commit's history.
- Remote repo (master)
    - This is a remote repository, also referred to as the main or master branch, and is the result of commits to a remote computer which stores data in a Git repository. From this branch we can create other branches on which we can work locally. This is the main repository from where all team members can pull data to update their codes locally and to push their changes for others to reflect on them.

WORKING DIRECTORY STATES:

- Staged
    - A file is staged if there were modifications on it since the last commit, or the file is new, and it was staged (like added in a queue and waiting to be committed). To stage a file, we need to use the ‘git add’ command.
- Modified
    - A file that has been modified but hasn’t been staged since the last commit. To modify a file, some or all content can be deleted, we can add new lines, or modify existing lines.
- Committed
    - After we stage the file(s), we can commit them which means storing them the Git directory. To commit the file, we use the command “git commit”.

GIT COMMANDS:

- Git add
    - This command (“git add <filename>”) is used when the file we are working on in the working directory is ready to be staged. In case we want to add all files, we can use the command “git add --all” or “git add -A”, where the “--all” and “-A” mean all the files ready to be staged. We can also use it on directory to stage all changes that took place there, using “git add <directory>” command. To check the status of the files before or after staging, we can use the command “git status”.
- Git commit
    - Commit command, together with git add, are the two commands mostly used by developers. Commit is Git’s “save” way to capture a project state as a snapshot and to add it to a repository. Multiple commits on a repository, create a history of commits. We can go back to any version of the project based on this history. The command is written “git commit -m” where “m” is abbreviation for message. The message should be short and descriptive. We can also use the commit without staging files beforehand if the changes are minimal and the files have been tracked already. The command used for this “git commit -a -m”, where “a” means that the changes will get staged automatically. If we need to check the history of commits, we can use the command “log”, and this will display a list of commits made on that repository.
- Git push
    - Once our files have been committed, we can send them from our local repository to the remote repository using the “git push <repository_name> <branch_name>” command. These commits can be pushed to the main branch or to another branch, and they will overwrite files in the remote repository, according to the changes that happened to the files. Once the files are on the remote repository, the whole team has access to these files.
- Git fetch
    - “git fetch <remote_repo>” or “git fetch <remote_repo> <branch_name>” is used to download all commits and files that were made in the remote repository by other team members that worked on different parts of the project. It does not overwrite the files in the local repository, is a safe way to compare code, and to keep you up to date.
- Git merge

- This command is used to combine two or more branches into one. Let's suppose that we are working on two branches, the main, and another branch named branch_name. After we worked on our local repository, on branch_name for a while and we built a history of commits, we are happy with the result, and we would like to join this branch with the main one. To do this, we need to first change to our main branch using "git checkout main" command, then we can merge our branch_name using the command "git merge branch_name" and delete the branch using "git branch -d branch_name" as is no longer needed. Merge is looking for a common point between the two branches, in history, and merges their histories together. If the evolution of the branch is linear, then the main's tip will end up in the same position as the branch_name tip. If the branches evolved differently, then merge is finding the common point in history, merges their history, and creates a new tip where both main and branch_name combine. This merge is called a 3-way merge.

TASK 2 (Exception handling)

Question 1

Using exception handling code blocks such as try/ except / else / finally, write a program that simulates an ATM machine to withdraw money.

(NB: the more code blocks the better, but try to use at least two key words e.g. try/except)

Tasks:

1. Prompt user for a pin code

2. If the pin code is correct then proceed to the next step, otherwise ask a user to type in a password again. You can give a user a maximum of 3 attempts and then exit a program.

3. Set account balance to 100.

4. Now we need to simulate cash withdrawal

5. Accept the withdrawal amount

6. Subtract the amount from the account balance and display the remaining balance (NOTE! The balance cannot be negative!)

7. However, when a user asks to 'withdraw' more money than they have on their account, then you need to raise an error an exit the program.

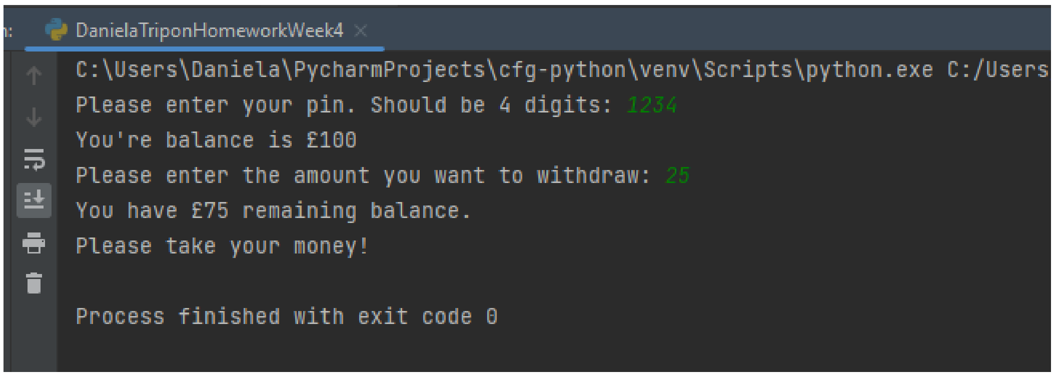
Answer:

```
# QUESTION 2

# Using exception handling code blocks such as try/ except / else /
finally,
# write a program that simulates an ATM machine to withdraw money.
#  (NB: the more code blocks the better, but try to use at least two key
words e.g. try/except)
```

```python
# Tasks:
# 1. Prompt user for a pin code
# 2. If the pin code is correct then proceed to the next step, otherwise
ask a user to type in a password again.
# You can give a user a maximum of 3 attempts and then exit a program.
# 3. Set account balance to 100.
# 4. Now we need to simulate cash withdrawal
# 5. Accept the withdrawal amount
# 6. Subtract the amount from the account balance and display the remaining
balance (NOTE! The balance cannot be
# negative!)
# 7. However, when a user asks to ‘withdraw’ more money than they have on
their account, then you need to raise an
# error an exit the program.

# I wrote 2 solutions

# First solution

is_success = False
correct_pin = 1234


def validate_pin(pin):
    correct = pin == correct_pin
    cast_pin = str(pin)
    pin_length = len(cast_pin)
    if not (pin_length == 4):
        raise ValueError("Your pin must be 4 digits long")
    if not correct:
        raise ValueError("Incorrect pin.")


for attemptNo in range(3):
    pin_input = int(input('Please enter your pin. Should be 4 digits: '))
    try:
        validate_pin(pin_input)
        is_success = True
        break
    except ValueError as exc:
        print("Invalid input: %s" % exc)


def validate_amount(amount):
    available_balance = 100
    remaining_balance = available_balance - amount
    if remaining_balance >= 0:
        print(f"You have £{remaining_balance} remaining balance.")

    if amount > available_balance:
        raise ValueError('Please withdraw £100 or less')
    if remaining_balance < 0:
        raise ValueError('Insufficient funds.')


if is_success:
    print("You're balance is £100")
    cash_amount = int(input('Please enter the amount you want to withdraw:
'))
    try:
        validate_amount(cash_amount)
        print('Please take your money!')
    except ValueError as exc:
        print('Insufficient funds: %s' % exc)

###########################################
# Second solution


# is_success = False
# correct_pin = 1234
#
#
# def validate_pin(pin):
#     count = 0
#     correct = pin == correct_pin
#     while count < 2:
#         if correct:
#             break
#         count += 1
#         pin = int(input('Try again. Pin should be 4 digits long: '))
#     cast_pin = str(pin)
#     pin_length = len(cast_pin)
#     if not (pin_length == 4):
#         raise ValueError("Your pin must be 4 digits long")
#     if not correct:
#         raise ValueError("Incorrect pin.")
#
#
# def validate_amount(amount):
#     available_balance = 100
#     remaining_balance = available_balance - amount
#     if remaining_balance >= 0:
#         print(f"You have £{remaining_balance} remaining balance.")
#
#     if amount > available_balance:
#         raise ValueError('Not enough funds. Please withdraw £100 or
less')
#     if remaining_balance < 0:
#         raise ValueError('Insufficient funds. Try a different amount')
#
#
# try:
#     pin_input = int(input('Please enter your pin. Should be 4 digits: '))
#
#     validate_pin(pin_input)
#
#     cash_amount = int(input('Please enter the amount you want to
withdraw: '))
#     validate_amount(cash_amount)
#
#
# except ValueError as exc:
#     print("Invalid input: %s" % exc)
# else:
#     is_success = True
#     print("Thank you for using our services!")
# finally:
#     if is_success:
#         print('Please take your money!')
#      else:
#          print('Could not complete cash withdrawal')
```

Console:

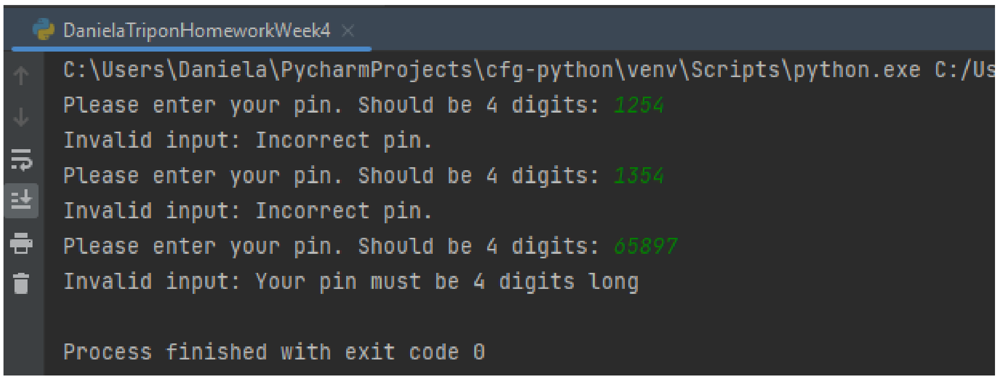
Correct pin:



DanielaTriponHomeworkWeek4

```
C:\Users\Daniela\PycharmProjects\cfg-python\venv\Scripts\python.exe C:/Users
Please enter your pin. Should be 4 digits: 1234
You're balance is £100
Please enter the amount you want to withdraw: 25
You have £75 remaining balance.
Please take your money!

Process finished with exit code 0
```

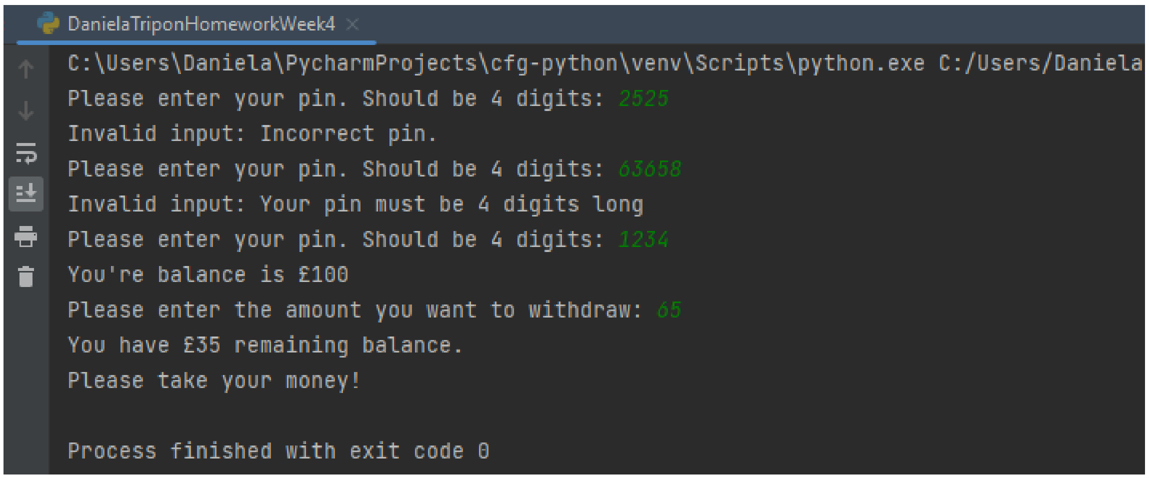
Wrong pin 3 times:



DanielaTriponHomeworkWeek4

```
C:\Users\Daniela\PycharmProjects\cfg-python\venv\Scripts\python.exe C:/Us
Please enter your pin. Should be 4 digits: 1254
Invalid input: Incorrect pin.
Please enter your pin. Should be 4 digits: 1354
Invalid input: Incorrect pin.
Please enter your pin. Should be 4 digits: 65897
Invalid input: Your pin must be 4 digits long

Process finished with exit code 0
```

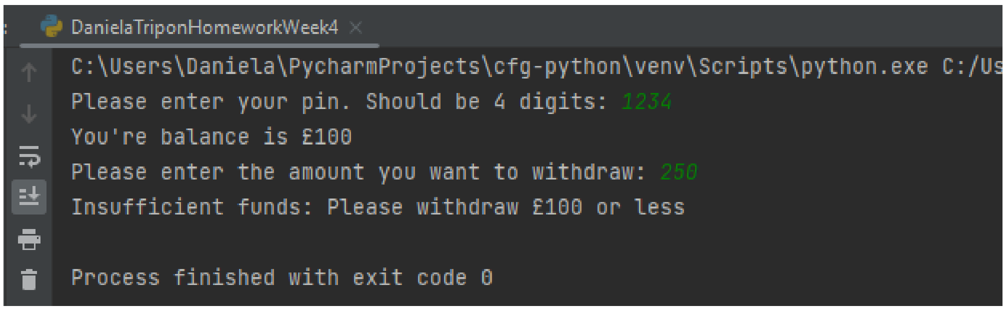
Wrong pin 2 times, and last time correct:



DanielaTriponHomeworkWeek4

```
C:\Users\Daniela\PycharmProjects\cfg-python\venv\Scripts\python.exe C:/Users/Daniela
Please enter your pin. Should be 4 digits: 2525
Invalid input: Incorrect pin.
Please enter your pin. Should be 4 digits: 63658
Invalid input: Your pin must be 4 digits long
Please enter your pin. Should be 4 digits: 1234
You're balance is £100
Please enter the amount you want to withdraw: 65
You have £35 remaining balance.
Please take your money!

Process finished with exit code 0
```

Withdrawal amount greater than balance



```
C:\Users\Daniela\PycharmProjects\cfg-python\venv\Scripts\python.exe C:/Us
Please enter your pin. Should be 4 digits: 1234
You're balance is £100
Please enter the amount you want to withdraw: 250
Insufficient funds: Please withdraw £100 or less

Process finished with exit code 0
```

TASK 3 (Testing)

Question 1

Use the Simple ATM program to write unit tests for your functions.

You are allowed to re-factor your function to 'untangle' some logic into smaller blocks of code to make it easier to write tests.

Try to write at least 5 unit tests in total covering various cases.

Answer

Re-factored functions:

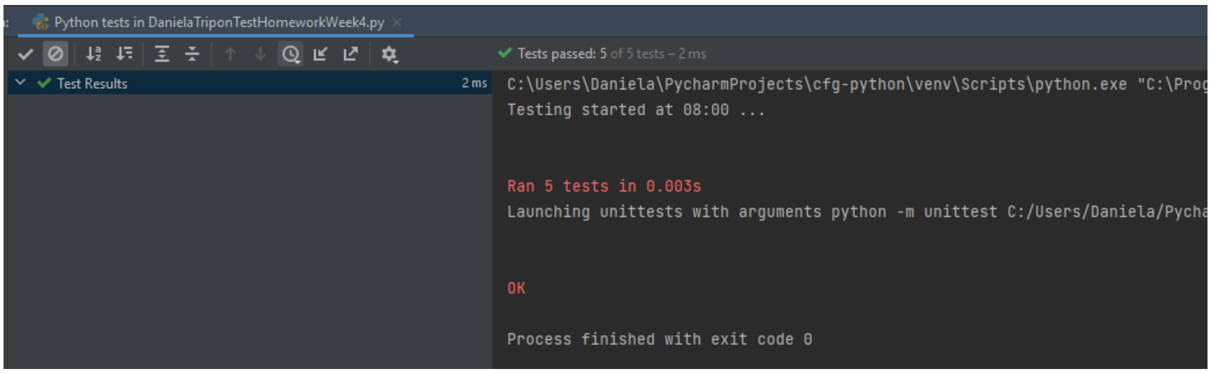
```
correct_pin = 1234


def validate_pin1(pin):
    correct = pin == correct_pin
    if correct:
        return pin

    if not correct:
        raise ValueError("Incorrect pin.")


def validate_pin2(pin):
    correct = pin == correct_pin
    if correct:
        return pin
    cast_pin = str(pin)
    pin_length = len(cast_pin)
    if not (pin_length == 4):
        raise ValueError("Your pin must be 4 digits long")


def validate_amount1(amount):
    available_balance = 100
    remaining_balance = available_balance - amount
    if remaining_balance >= 0:
```

```python
        print(f"You have £{remaining_balance} remaining balance.")

    if amount > available_balance:
        raise ValueError('Not enough funds. Please withdraw £100 or less')


def validate_amount2(amount):
    available_balance = 100
    remaining_balance = available_balance - amount
    if remaining_balance >= 0:
        print(f"You have £{remaining_balance} remaining balance.")

    if remaining_balance < 0:
        raise ValueError('Insufficient funds. Try a different amount')


# validate_pin1(1234)
# validate_pin1(1234)
# validate_pin2(1234)
# validate_amount1(200)
# validate_amount2(110)
```

Test code:

```python
import unittest
from unittest import TestCase
# from DanielaTriponHomeworkWeek4 import validate_pin
from DanielaTriponHwWeek4Simple import validate_pin1, validate_pin2,
validate_amount1, validate_amount2


class TestValidatePinAndAmount(TestCase):
    def test_validate_pin(self):
        actual = 1234
        expected = validate_pin1(pin=1234)
        self.assertEqual(actual, expected)

    def test_validate_pin_exception(self):
        wrong_pin = 1222
        with self.assertRaises(ValueError) as exc:
            validate_pin1(pin=wrong_pin)
        self.assertEqual(
            str(exc.exception),
            "Incorrect pin."
        )

    def test_validate_pin_another_exception(self):
        wrong_pin = 12345
        with self.assertRaises(ValueError) as exc:
            validate_pin2(pin=wrong_pin)
        self.assertEqual(
            str(exc.exception),
            "Your pin must be 4 digits long"
        )

    def test_validate_amount_exception(self):
        big_amount = 200
        with self.assertRaises(ValueError) as exc:
            validate_amount1(amount=big_amount)
        self.assertEqual(
            str(exc.exception),
            'Not enough funds. Please withdraw £100 or less'
        )

    def test_remaining_balance_exception(self):
        rem_balance = 110
        with self.assertRaises(ValueError) as exc:
            validate_amount2(amount=rem_balance)
        self.assertEqual(
            str(exc.exception),
            'Insufficient funds. Try a different amount'
        )


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

Console:


Python tests in DanielaTriponTestHomeworkWeek4.py
Tests passed: 5 of 5 tests – 2 ms
Test Results
2 ms

```
C:\Users\Daniela\PycharmProjects\cfg-python\venv\Scripts\python.exe "C:\Prog
Testing started at 08:00 ...


Ran 5 tests in 0.003s
Launching unittests with arguments python -m unittest C:/Users/Daniela/Pycha


OK

Process finished with exit code 0
```