Assignment Report

Assignment Number	1
Student 1	
Name	Emil Wallin
Email address	emil-wallin@live.se
Student 2	
Name	Erik Erlandsson
Email address	Erik.Erlandsson.mail@gmail.com

Contents

Changes to code

__init__(self)

Added **self** to the **cursor** and **conn** variables. **Screenshot**

check_valid_age(self, age)

added **an upper limit** to how old a person can be. We decided that it was unreasonable that a person older than 200 years should be added into the system. However if this was a system cataloging the age of planets, this limit would be unreasonable.

check_valid_text_field(self, input):

We changed the name of the parameter called **emptyTextNotAllowed** to **emptyTextAllowed**

because it was confusing with double negative in the name.

Added a check to see whether a **str** is passed. **Screenshot**

we found this bug when we tried to pass a list containing integers, which also contains the attribute **.len().** As an example**, this** would also pass as true before we added that constraint, since a list of course has a length.

customer_has_equipment_attached(self, customerID):

Added **self** to the **cursor** and **conn** variables.

Added an else statement which **return True**, for when the customer actually has equipment. otherwise the function return **this** when the customer has equipment

Deliverables

A motivated description of your unit tests

__init__(self)

We decided to mock the dataBase call using our own made call to replace the function call. then we checked to see whether **conn** and **cursor** have received any values.

The Reason for the mock was because we do not care what the database returns but we do care about if what is returned is assigned to **self**

check_valid_age(self, age)

The second test examines whether the age passed into the function is valid or not. We do so by passing multiple different arguments and testing if their return value is as expected.

arguments we tested:

- ❏ 10 - an integer in a reasonable span (0 > 10 < 201)
- ❏ ‘tio’ - a str
- ❏ -1 - a negative integer
- ❏ 0 - the number 0

- ❏ 7/5 - a float number
- ❏ 201 - an integer which we deemed to be unreasonably high, (200 > 201)

By doing these tests we think we have tested for all kinds of different input which could lead to a defect. (We did not test all different data types because there is a check in the function whether argument is of type integer or not, so it would be unnecessary)

check_valid_text_field()

The Function evaluates whether an input is a valid text or not. We do so by passing a str as an argument to the function and also passing an empty str.

we did this for when the input is allowed to be empty and when it is not.

Lastly we tested to send in something other than a str. in our case a list of integers.

customer_has_equipment_attached(self, customerID)

The last function we tested gathered some data from a database so we analyzed what it was really looking for and determined it first selected data about a certain customer, and thereafter checking if said customer had equipment or not, using a foreign key called IMEIPtr.

To investigate this function we tested with three different functions:

- ❏ customer with IMEIPtr and equipment
- ❏ customer *without IMEIPtr*
- ❏ customer with IMEIPtr but without equipment

By doing so we determined that we have tested all possible outcomes of that function

A list of identified errors in the code

__init__(self)

- ❏ There was an error when calling the conn and cursor objects which caused the code not to run.

check_valid_age(self, age)

- ❏ We decided that we should limit how old a person could be to have a valid age.
  Since the age refers to a person we decided to limit it to no more than 200

check_valid_text_field()

- ❏ We found that the parameter emptyTextNotAllowed was poorly named, and caused much confusion which would increase the chance of the function being misused.
- ❏ When we passed in a list of integers we found that the function deemed it a valid text, we decided that only pure text (**str**) should be considered a valid text. Or you could send in anything which had the property .len() and it would be a valid text.

customer_has_equipment_attached(self, customerID)

- ❏ We found that the function would never return true, only false, even though all data was in place. We deemed that as an error because the sole purpose of the function is to check if all data is in place or not, it is logical that it should return true if all data was present.
- ❏ Here as well there was an error when calling the conn and cursor objects which caused the code not to run.

An example of the application of mocking

In our case we decided to mock the database calls because we don't want to make sure the correct values are in the database. and we do not care if the database itself works or not.

In test_has_equipment we overwrite the functions which handles the calls to the database. This way we have full control over what id returns and our test will work independent of the database.

Here is an snapshot of **test_has_equipment.py**

An evaluation of your test coverage

Our test coverage can be viewed **here**

This coverage shows a 100% coverage. Which means all lines in datachecker are being executed during all of our tests. This is quite good as it would have indicated that there were untested parts of the code if it was not 100%. During Our testing process we discovered that we had forgotten a case in one of the test suites because our coverage landed on 96%.

However just because the coverage is 100% it does not mean that there are no bugs in the code. As the coverage only indicates which lines of code are being executed by the test files.

Rationale

In the function “check_valid_age” in the file “datachecker.py” we added an upper age limit and we set it to 200 years. We did this because we determined it is not valid to add dead people into the database since it is a record for people with an address and an email and no alive person is older than 200 years.

We decided that an age with a decimal should be allowed, for example 2.5 years. That is because it might be interesting to know in some cases. For example 0.1 years vs 0.9 years is a big difference. This depends on the database, if the column for age allows float or integers only. However it should not cause a defect or failure because it is easy to convert a float to integer before it is inserted if that would be the case.

We decided that the emptyTextNotAllowed parameter in “check_valid_text_filed” was poorly named as it contained a negative in the name. this could potentially cause confusion or even misuse when the function is being used.
We decided to change it to emptyTextAllowed as we found it to be less confusing.

We decided that it is not allowed to send in anything other than a str to “check_valid_text_field” as it would not be pure text. Otherwise anything with the attribute .len() would be a valid text.

We decided that the function which checks for equipment should return true if all the tests passed, (if the customer was in the database with a proper IMEIPtr and equipment) so we added a return statement for that case to be true.

Code

Below are hyperlinks of our python files containing the implementation of the tests & the base file datachecker. The links lead to our public github repository.

datachecker.py

test_has_equipment.py

test_init.py

test_valid_age.py

test_valid_input.py

Screenshot init method

```
self.conn = sqlite3.connect('pos
self.cursor = self.conn.cursor()
```

Screenshot no upper limit

```
if age > 200:
    print("Age can't be more than 200 years old.")
    return False
```

Screenshot check if str

```
if type(input) != str:
    print('Non-valid str input')
    return False
```

Screenshot len attribute

```
def test_passing_other_than_string(self):
    assert DataChecker.check_valid_text_field(self, [1,1,1]) == False
```

Screenshot no return value

```
assert None == True
```

Screenshot return True

```
if len(equipment) == 0:
    print('Customer has
    return False
else:
    return True
```

Screenshot test coverage

customer.py	20	2	0	90%
datachecker.py	46	0	0	100%
test_has_equipment.py	35	0	0	100%
test_init.py	13	0	0	100%
test_valid_age.py	20	0	0	100%
test_valid_input.py	12	0	0	100%

Screenshot test_has_equipment

```python
from datachecker import DataChecker

class MockCursor(object):
    def fetchall(self): ...

    def fetchall_no_equipment(self): ...

    def fetchone(self): ...

    def fetchone_no_equipment(self):
        return ""

    def execute(self,arg,args):
        return True

class MockConn(object):
    def commit(self):
        return True

class TestClass(object):
    def test_passing(self):
        self.cursor=MockCursor()
        self.conn=MockConn()
        assert DataChecker.customer_has_equipment_attached(self,340) == True

    def test_no_ptr(self):
        self.cursor=MockCursor()
        self.conn=MockConn()
        self.cursor.fetchall=self.cursor.fetchall_no_equipment
        assert DataChecker.customer_has_equipment_attached(self,340) == False

    def test_empty_equipment(self):
        self.cursor=MockCursor()
        self.conn=MockConn()
        self.cursor.fetchone=self.cursor.fetchone_no_equipment
        assert DataChecker.customer_has_equipment_attached(self,340) == False
```