Testing example

The specification was different last year.

2021-22 Coursework specification

Select and set-up an appropriate unit testing library to use and demonstrate that you can use it by creating and running at least two unit tests.

1. Select and install an appropriate unit testing library (e.g. unittest or pytest)
2. Create an appropriately named test directory in your project
3. Choose Python code to write the tests for (the choice has no implication for marking):
    a. Create your own Python method/function or a class that could be used in your application code. If you created a class diagram you should have a class you could create.
    b. If you create a Python function in coursework 1 as part of the data preparation or exploration activity you may be able to test that.
    c. Use the sample User class provided at the end of this section.
4. Write at least two tests that test the selected Python code. The tests should test for different aspects e.g., a test that shows it work with correct data and one that shows it fails appropriately with incorrect data.
5. Run the tests and capture the results.

Consider the quality of your Python test code. Code quality is covered in the course materials on Moodle.

Students aiming for a higher mark are encouraged to explore setting up a continuous integration pipeline using GitHub Actions or similar. There are template pipelines in GitHub actions that will configure and run pytest unit tests as well as tools such as linters. Consider the benefit of using such tools in the development phase of a project.

User class code

```
import hashlib


class User(object):

    def __init__(self, first_name, last_name, email_address, username=""):
        self.first_name = first_name
        self.last_name = last_name
        self.email_address = email_address
        self.username = username
        self._password_hash = None

    def __repr__(self):
        """Returns the attributes (excluding password) of the user as a string"""
        return '{}, {}, {}, {}'.format(self.first_name, self.last_name,
self.email_address, self.username)

    @property
    def password_hash(self):
        """Get the current hashed password. If no password is set then the default
is None"""
        return self._password_hash
```

```
    @password_hash.setter
    def password_hash(self, password):
        """
        Takes a password and converts it to a hashed value and assigns it to the
_password property
            :param password: password entered
            :type password: string
        """
        self._password_hash = hashlib.sha256(str.encode(password)).hexdigest()

    def check_password(self, password):
        """
        Takes a string password and checks that is matches the assigned hash.
            :param password: password entered
            :type password: string
            :return: True if the password matches, otherwise False
            :rtype: bool
         """
        password_hash = hashlib.sha256(str.encode(password)).hexdigest()
        return password_hash == self._password_hash
```

Merit example

Tutor comment

Student feedback “The tests are straightforward though well done with appropriate structure, use of a fixture, meaningful names and appropriate asserts. GitHub actions used for testing and linting.”

Code not provided since it would be too easy to copy it. The following should give a reasonable understanding of what the student did.

Student submission

After installing the pytest library, from the sample user class code provided, we decided to focus on testing the following two methods for this project: create_full_name and calculate_age. We are assuming that performing unit testing on this code will help us test part of the Account model identified in the design section. Both of the tests have been determined and described below using the GIVEN-WHEN-THEN Approach. With this approach, we realized that the set-up condition is the same for both tests. Indeed, both methods (create_full_name and calculate_age) required us to create a new user for unit testing. Therefore, we used fixtures to provide a common function (general_user) for both tests, allowing us to reduce common code. As the fixture will need to be executed for each of the test functions (Gao, 2020), we will be using the "function" scope. Finally, each of the functions has been tested twice: both with the correct data and the incorrect data.

Test function 1: test_create_full_name

The first test function test_create_full_name aims to test whether the correct full name is returned for a new user. It can be, therefore, be described with the following GIVEN-WHEN-THEN Approach:

```
"""
GIVEN a new user (created as fixture) named James White
WHEN his first_name and last_name are passed to the "create_full_name"
function
THEN the full_name should be James White

"""
```

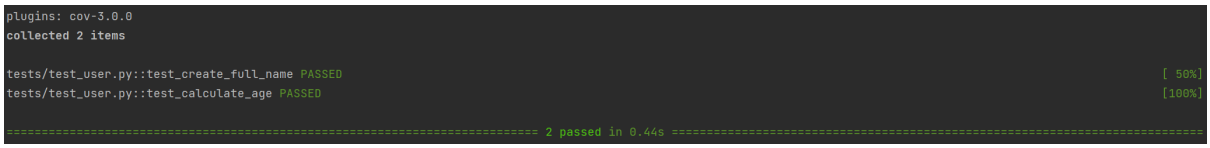
Test Function 2: test_calculate_age

The second function test_calculate_age aims to test whether the correct age is returned given the date of birth of a new user. It was necessary to convert the date of birth from string to date, in order to correctly test this function. It can be described with the following GIVEN-WHEN-THEN Approach:

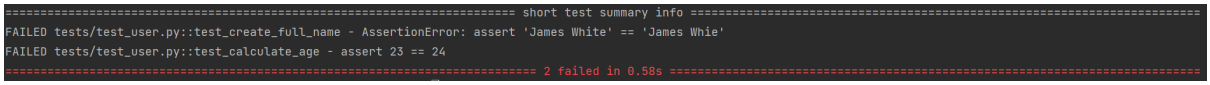
```
"""
GIVEN a new user (created as fixture) born in 1998
WHEN his date of birth (dob) is passed to the "calculate_age" function
THEN the age should be equal to 23

"""
```

Test results

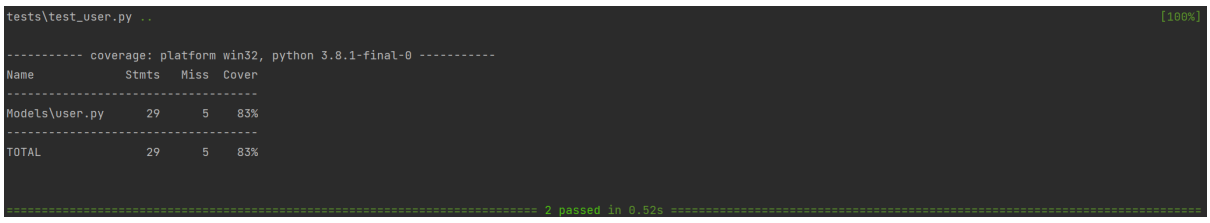
Testing Results with Correct Data

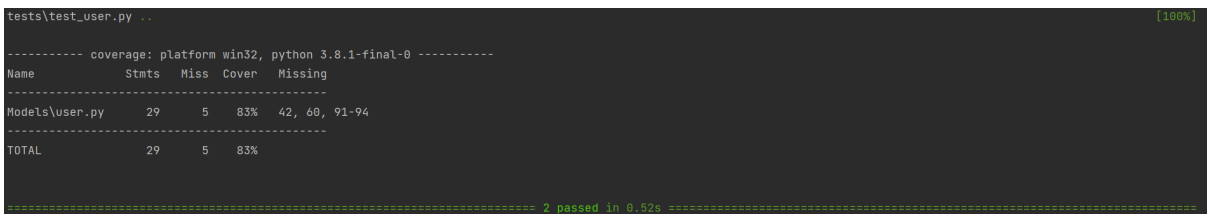


Testing Results with Incorrect Data (for both functions)



Coverage Reports

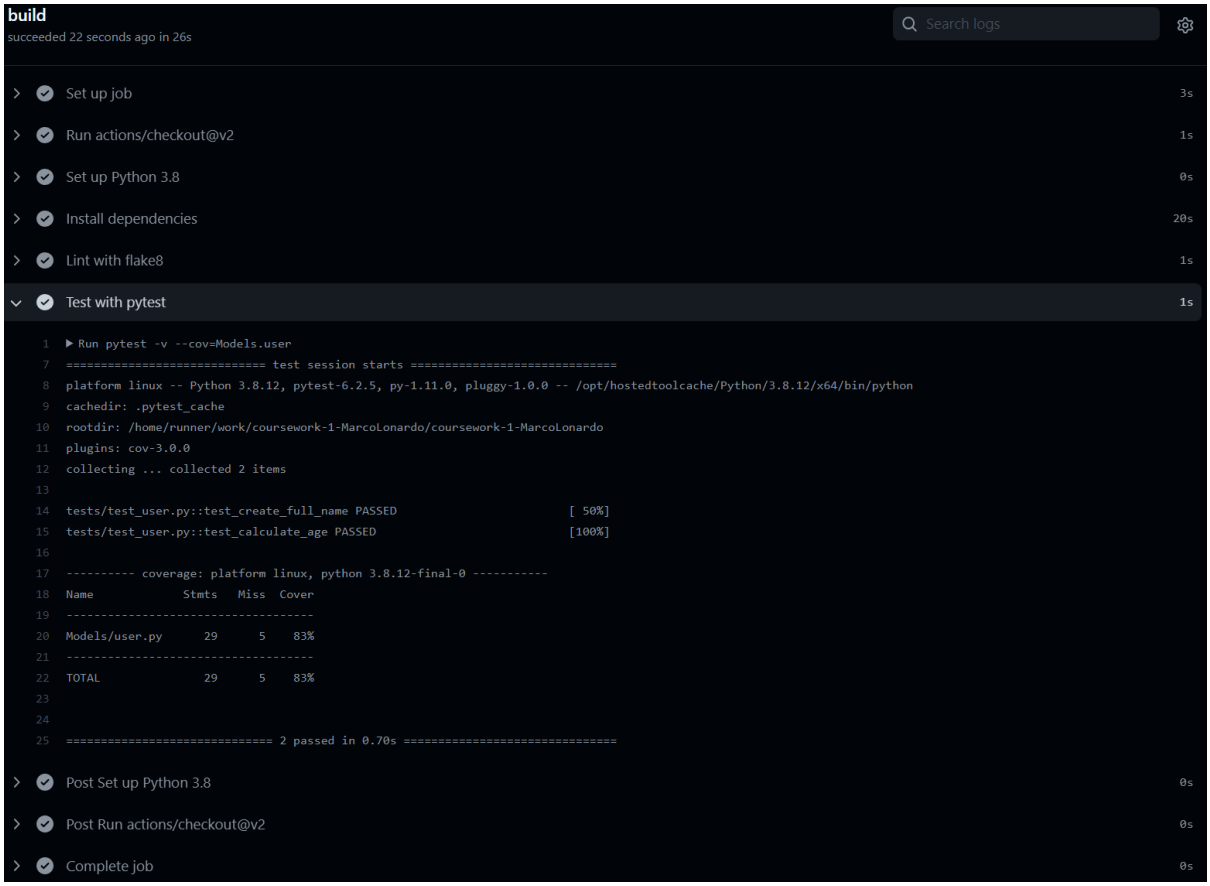




Continuous integration

{{student provided their yml here}}

Results with correct data for both methods



Results with incorrect data only for the calculate_full_name method

Pass example

Tutor comment

Feedback given to the student: “Your tests have a reasonable structure, however there are a lot of code quality issues. You have 28 PEP warnings on a file that is only 39 lines long. Can I suggest you look up how to use the auto linting features in your IDE as these are easy to correct. You could also improve the naming convention, it isn't easy to see what the difference between for example test_logon_1 and test_logon_5, i.e. don't use numbers, add a meaningful word that describes the test. Test files should be named starting `test_` or ending `_test` and by convention places in a folder called 'tests' or 'test'.”

Student submission

{The student only provided the following code. No use of conftest.py fixtures nor GitHub Actions}

```
import pytest

def login(username,password):

    if username=='ILOVEPYTHONSOMUCH' and password=='20211215':
        print('Login Successfully')
        return username,password
    else:
        print('Login Failed')
        return username,password


def test_login_1():
    """To check whether the username and the password matches
or not"""
    user_login=login('ILOVEPYTHONSOMUCH','20211215')
    #By Using Assert to compare the values.
    assert(user_login==('ILOVEPYTHONSOMUCH','20211215'))
@pytest.mark.failtest
def test_login_2():
    """Username and password are null"""
    user_login=login('','')
    #By using assert to check whether the login is successful
    assert(user_login==('ILOVEPYTHONSOMUCH','20211215'))
def test_login_3():
    """Correct username but wrong password"""
    user_login=login('ILOVEPYTHONSOMUCH','000000000')
    #By using assert to check whether the login is successful
    assert (user_login == ('ILOVEPYTHONSOMUCH','20211215'))
def test_login_4():
```

```
    """Wrong Username but correct password"""
    user_login=login('Ihatepython','20211215')
    # By using assert to check whether the login is successful
    assert (user_login == ('ILOVEJAVASOMUCH', '20211215'))
def test_login_5():
    """Both Username and password are wrong"""
    user_login=login('IDISLIKEPYTHONSOMUCH','000000000')
    # By using assert to check whether the login is successful
    assert (user_login == ('ILOVEPYTHONSOMUCH', '20211215'))
```