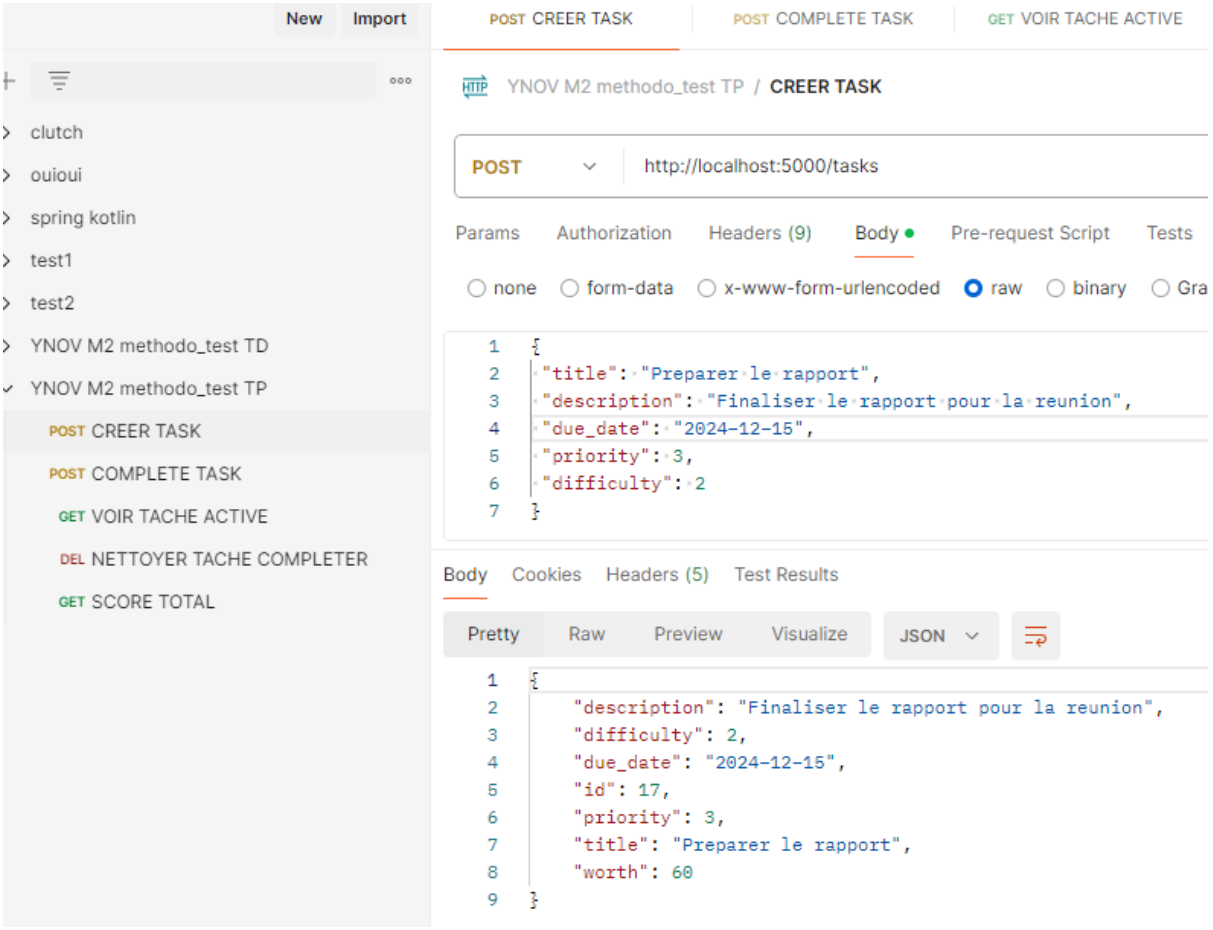
Travail Pratique : Tests sur une Application déjà existante

Réalisé par Gaetan Corin, M2 Data-Enginner Ynov

Étape 1 : Tests Fonctionnels et Boîte Noire

En premier, j'ai créer un fichier "start2.ps1" qui lance l'applications tasks.py.
J'ai ensuite lancer le serveur d'application de "tasks"

Le première étape de ce TP a était pour moi de réaliser l'ensemble des Appels API sur Postman afin de mieux comprendre les point d'API et de me faciliter la compréhension de l'application tasks.py



J'ai ensuite créer une duplication des tests existant de "test_librarie.py" dans un nouveau fichier de test appelé "test-tasks" (car les 2 applications sont très similaires en terme de cas d'usage des APIs).
J'ai commenté l'ensemble du fichierde test test_library, j'ai adapté les imports du fichier test_tasks, puis j'ai lancer la commande "pytest" dans le terminal.

Les deux tests ont échoué car ils ne sont pas adaptés à l'application tasks, mais je n'ai pas eu de message d'erreur, ma configuration est bonne, je suis donc prêt à créer mon premier test.

1er test:

Le premier test utilise les apis

- /tasks
- /tasks/active

Le test crée une tache dans le futur, puis va récupérer toute les taches active, filtrer sur l'id de la tache qui vient d'être créer et en vérifier son contenu
objectif: vérifier que la tache a bien été créer dans la base de donnée

```
def test_activate_one_task(client):
    response = client.post('/tasks', json={
        "title": "Repasser les chausettes",
        "description": "Consiste a repasser les 2 chausettes dedans et
dehors",
        "due_date": "2024-12-20",
        "priority": 1,
        "difficulty": 2
    })
    id = response.get_json()['id']
    tasks_active = client.get(f'/tasks/active').get_json()
    new_create_active_task = next((item for item in tasks_active if
item["id"] == id), None)
    potential_result = {
        'completed': 0,
        'description': 'Consiste a repasser les 2 chausettes dedans et
dehors',
        'difficulty': 2, 'due_date': '2024-12-20',
        'id': id,
        'priority': 1,
        'title': 'Repasser les chausettes',
        'worth': 20
    }
    assert potential_result == new_create_active_task
```

2ème test:

Ledeuxième test utilise les apis

- /tasks/cleanup
- /tasks/
- /tasks/{id}/complete
- /tasks/cleanup

Le test retire les taches terminé, puis va créer une nouvelle tache, la considéré comme terminé, puis va retiré à nouveau les taches terminé pour voir si cette nouvelle tache terminé est bien retiré.

objectif: vérifié que la suppression des taches terminé fonctionne correctement

```
def test_complete_one_task(client):
    client.delete('/tasks/cleanup')
    response = client.post('/tasks', json={
        "title": "Finir le packet de chips",
        "description": "Finaliser le packet de chips et manger les
miettes",
        "due_date": "2024-12-18",
        "priority": 3,
        "difficulty": 2
    })
    id = response.get_json()['id']
    client.post(f'/tasks/{id}/complete')
    assert client.delete('/tasks/cleanup').get_json()['message'] == "1
tâches obsolètes ou complétées supprimées"
def fake_execute(request, args):
    pass
```

3ème test:

Le troisième test utilise les apis

- /scores/total
- /tasks/
- /tasks/{id}/complete
- /scores/total

Le test récupère le score, crée une nouvelle tache et la considère comme terminé, puis récupère le score a nouveau pour le comparé avec le précédent.

objectif: Vérifie que le score a bien été incrémenter avec la tache terminé.

```
def test_incremente_score(client):
    score_before =
client.get('/scores/total').get_json()['total_score']
    print(score_before)
    response = client.post('/tasks', json={
        "title": "Une fausse tache pour fainéantiser 1 heure",
        "description": "Consiste a finaliser un faux ticket pour
montrer que... ca bosse",
        "due_date": "2024-12-25",
        "priority": 3,
        "difficulty": 2
    })
    id = response.get_json()['id']
    client.post(f'/tasks/{id}/complete')
    score_after = client.get('/scores/total').get_json()['total_score']
    assert score_after == score_before + 60
```

4ème test:

Le quatrième test utilise les apis

- /tasks/active
- /tasks/{id}/complete
- /tasks/active

Le test récupère les taches active, pour chacune des taches, va les terminé, puis va vérifier qu'il n'y ai plus de taches active.

objectif: Vérifie que la liste de taches active fonctionne correctement, et remet a 0 les taches active en base de donnée.

```
def test_complete_all_active_tasks(client):
    tasks_active = client.get(f'/tasks/active').get_json()
    for task in tasks_active:
        client.post(f'/tasks/{task["id"]}/complete')
    assert client.get(f'/tasks/active').get_json()['message'] ==
'Aucune tâche active trouvée.'
```

Étape 2 : Tests Boîte Blanche et Couverture de Code

J'utilise ensuite "pytest --cov" pour vérifier que toutes mes APIS sont bien utilisés.

```
---------- coverage: platform win32, python 3.10.0-final-0 -----------
Name                    Stmts   Miss  Cover
-------------------------------------------
src\__init__.py             0      0   100%
src\tasks.py              111     14    87%
tests\__init__.py           0      0   100%
tests\test_librairie.py     0      0   100%
tests\test_tasks.py        37      1    97%
-------------------------------------------
TOTAL                     148     15    90%
```

Mes tests me signale qu'ils couvrent 87% de l'application, car je ne teste pas les tests dans le passé.

Je crée donc un nouveau test qui crée un test dans le passé,

5ème test:

Le quatrième test utilise les apis

- /tasks
- /tasks/active

Le test crée une tache dans le passé, puis vérifie si la tache est considéré comme active.

objectif: Vérifie que les taches dans le passé sont bien considéré comme non active.

```
def test_activate_past_task(client):
    client.post('/tasks', json={
        "title": "Marty, Pense a nourrir le chat",
        "description": "Penser a nourrir le chat avant de partir vers
le futur",
        "due_date": "1985-10-30",
        "priority": 1,
        "difficulty": 2
    })
    assert client.get(f'/tasks/active').get_json()['message'] ==
'Aucune tâche active trouvée.'
```

Après avoir réalisé mon nouveau test, je refais “pytest --cov”, mais cela n’améliore pas le pourcentage de couverture….

Je réalise donc la commande “pytest --cov=src --cov-report term-missing tests/” qui me donne les lignes non couvertes:

```
--------- coverage: platform win32, python 3.10.0-final-0 -----------
Name             Stmts   Miss  Cover   Missing
------------------------------------------------
src\__init__.py      0      0   100%
src\librairie.py   108    108     0%   2-206
src\tasks.py       111     14    87%   28, 31, 39, 64-84, 91, 142, 146, 221-222
------------------------------------------------
TOTAL              219    122    44%
```

Après vérification des lignes, mon problème vient du fait que:

- je n’ai pas testé le fait de mettre la date de rendu en None,(ligne28)
- je n’ai pas essayer de compléter une tache avec une date dans le passé (ligne39)
- je n’ai pas essayer de créer une tache sans titre (ligne91)
- je n’ai pas essayer de complete une tache avec un id introuvable (ligne142)
- je n’ai pas essaye de complete une tache deja complete (ligne 146)

Je suis découragé car il reste encore beaucoup de travail, alors je vais me faire un café, et je reprend juste après…

Voici les nouvelles implémentation des lignes manquantes:

- je n’ai pas testé le fait de mettre la date de rendu en None, (ligne28)

```
def test_create_task_without_date(client):
    response = client.post('/tasks', json={
        "title": "Un jour mon prince viendra",
        "description": "Trouve toi un boulot, plutôt...",
        "due_date": None,
        "priority": 1,
        "difficulty": 10
    })
    print(response)
    assert response.status_code == 201
```

-

-

- je n’ai pas essayer de compléter une tache avec une date dans le passé(ligne39)

```
def test_activate_past_task(client):
    response = client.post('/tasks', json={
        "title": "Marty, Pense a nourrir le chat",
        "description": "Penser a nourrir le chat avant de partir vers le futur",
        "due_date": "1985-10-30",
        "priority": 1,
        "difficulty": 10
    })
    assert client.get(f'/tasks/active').get_json()['message'] == 'Aucune tâche active trouvée.'
    id = response.get_json()['id']
    response = client.post(f'/tasks/{id}/complete')
    assert response.status_code == 200
```

- je n’ai pas essayer de créer une tache sans titre(ligne91)
-

```
def test_create_task_without_title(client):
    response = client.post('/tasks', json={
        "description": "Untitled",
        "due_date": "2024-12-20",
        "priority": 1,
        "difficulty": 10
    })
    assert response.status_code == 400
```

- je n’ai pas essayer de complete une tache avec un id introuvable (ligne142)

```
def test_complete_task_with_inexisting_id(client):
    response = client.post(f'/tasks/15487845854585/complete')
    assert response.status_code == 404
```

- je n’ai pas essaye de complete une tache deja complete(ligne 146)

```
def test_complete_one_task(client):
    client.delete('/tasks/cleanup')
    response = client.post('/tasks', json={
        "title": "Finir le packet de chips",
        "description": "Finaliser le packet de chips et manger les
miettes",
        "due_date": "2024-12-18",
        "priority": 3,
        "difficulty": 2
    })
    id = response.get_json()['id']
    client.post(f'/tasks/{id}/complete')
    assert client.delete('/tasks/cleanup').get_json()['message'] == "1
tâches obsolètes ou complétées supprimées"
    response = client.post(f'/tasks/{id}/complete')
    assert response.status_code == 404
```

Je relance la commande “pytest --cov=src --cov-report term-missing tests/” qui me donne les lignes non couvertes:

---------- coverage: platform win32, python 3.10.0-final-0 ----------

Name	Stmts	Miss	Cover	Missing
src__init__.py	0	0	100%	
src\librairie.py	108	108	0%	2-206
src\tasks.py	111	11	90%	31, 39, 64-84, 146, 221-222
TOTAL	219	119	46%	

ligne 31: je ne sais pas pourquoi elle n’est pas exécuté, , je donne toutes mes date en format str
ligne 39: je ne sais pas pourquoi elle n’est pas éxécuté, j’ai un test qui complete une tache de 1985.
ligne 146: lorsque une tache est déjà complete, elle est considéré comme non trouvé(ligne 141 142 error 404), le programme ne peux donc jamais passer sur ligne 146.

ligne 64-84: création de la base de donnée lorsque inexistante, a ne pas tester

ligne 221-222: lancement du programme, a ne pas tester.

Étape 3 : Utilisation de Mocks

Je crée une fonction qui utilise un mock pour créer un faux enregistrement en base de donnée sans avoir a y donner de valeur.

```
def test_incremente_score_with_mock(mocker, client):
    mock_db = mocker.patch('src.tasks.get_existing_or_create_db')
    mock_cursor = mocker.MagicMock()
    mock_cursor.fetchone.return_value = (1,) # return value doit etre
"subscriptable", par exemple un tuple
    mock_cursor.execute = fake_execute
    mock_db.return_value.cursor.return_value = mock_cursor
    response = client.post(f'/tasks/1/complete')
    assert response.status_code == 200
```

C’est génial quand cela marche car cela permet de créer des tests sans créer constamment de la fausse donnée a la main.
Malheureusement, j’ai des erreurs a cause du tuple (1,) lorsque je lance le programme.

```
    @app.route("/tasks/<int:task_id>/complete", methods=["POST"])
    def complete_task(task_id):
        db = get_existing_or_create_db()
        cursor = db.cursor()
        cursor.execute(
            "SELECT priority, difficulty, completed, due_date FROM tasks WHERE id =
?",
            (task_id,),
        )
        task = cursor.fetchone()

        if task is None:
            abort(404, "Tâche non trouvée.")

        print(task)
>       if task["completed"] == 1:
E       TypeError: tuple indices must be integers or slices, not str

src\tasks.py:145: TypeError
```

je me retrouve donc a modifier mon return value pour faire tourner mon programme

```
def test_incremente_score_with_mock(mocker, client):
    mock_db = mocker.patch('src.tasks.get_existing_or_create_db')
    mock_cursor = mocker.MagicMock()
    mock_cursor.fetchone.return_value = {'completed': 0, 'priority':1,
'difficulty': 1, 'due_date':"2024-12-25"} # return value doit etre
"subscriptable", par exemple un tuple
    mock_cursor.execute = fake_execute
    mock_db.return_value.cursor.return_value = mock_cursor
    response = client.post(f'/tasks/1/complete')
    assert response.status_code == 200
```

et cela perd tout son intérêt, puisque je dois donner des fausses valeurs à nouveau. Il y a donc un grand intérêt à utiliser un mock UNIQUEMENT lorsque l'on a pas à donner de fausses valeurs comme j'ai dû le faire…

Étape 4 : Tests Bout-en-Bout (E2E)

La fonction a déjà été réalisé auparavant, la voici:

```
def test_complete_one_task(client):
    client.delete('/tasks/cleanup')
    response = client.post('/tasks', json={
        "title": "Finir le packet de chips",
        "description": "Finaliser le packet de chips et manger les
miettes",
        "due_date": "2024-12-18",
        "priority": 3,
        "difficulty": 2
    })
    id = response.get_json()['id']
    client.post(f'/tasks/{id}/complete')
    assert client.delete('/tasks/cleanup').get_json()['message'] == "1
tâches obsolètes ou complétées supprimées"
    response = client.post(f'/tasks/{id}/complete')
    assert response.status_code == 404
```

Étape 5 : Tests d'Intégration et Validation de l'État de la Base de Données

La fonction a déjà été réalisé auparavant, la voici:

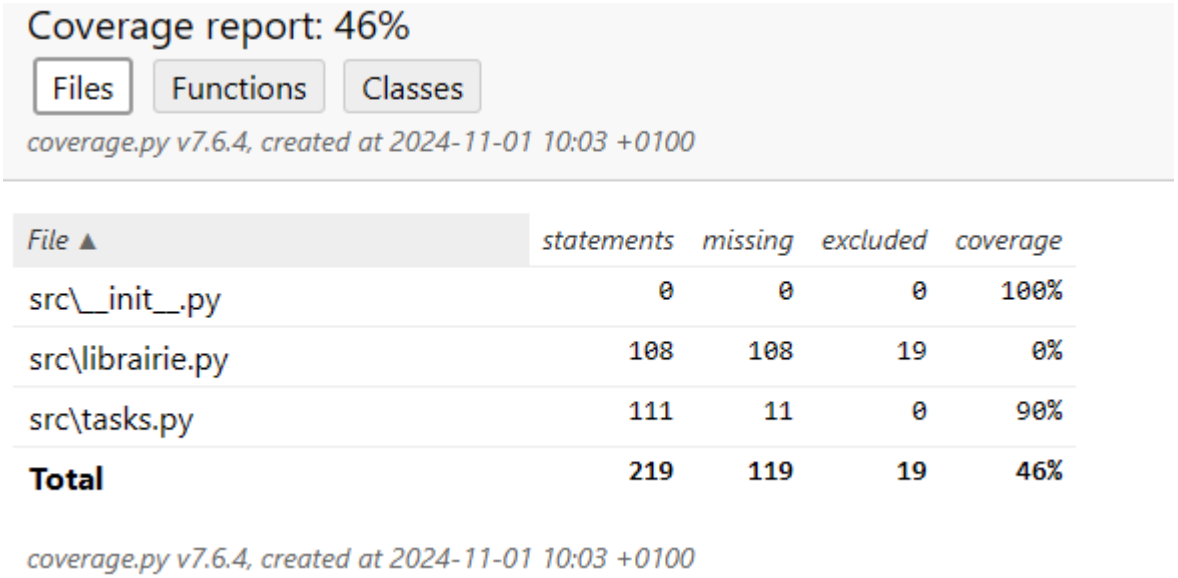
```
def test_incremente_score(client):
    score_before =
client.get('/scores/total').get_json()['total_score']
    print(score_before)
    response = client.post('/tasks', json={
        "title": "Une fausse tache pour fainéantiser 1 heure",
        "description": "Consiste a finaliser un faux ticket pour
montrer que... ca bosse",
        "due_date": "2024-12-25",
        "priority": 3,
        "difficulty": 2
    })
    id = response.get_json()['id']
    client.post(f'/tasks/{id}/complete')
    score_after = client.get('/scores/total').get_json()['total_score']
    assert score_after == score_before + 60
```

Étape 6 : Génération de Rapport de Tests

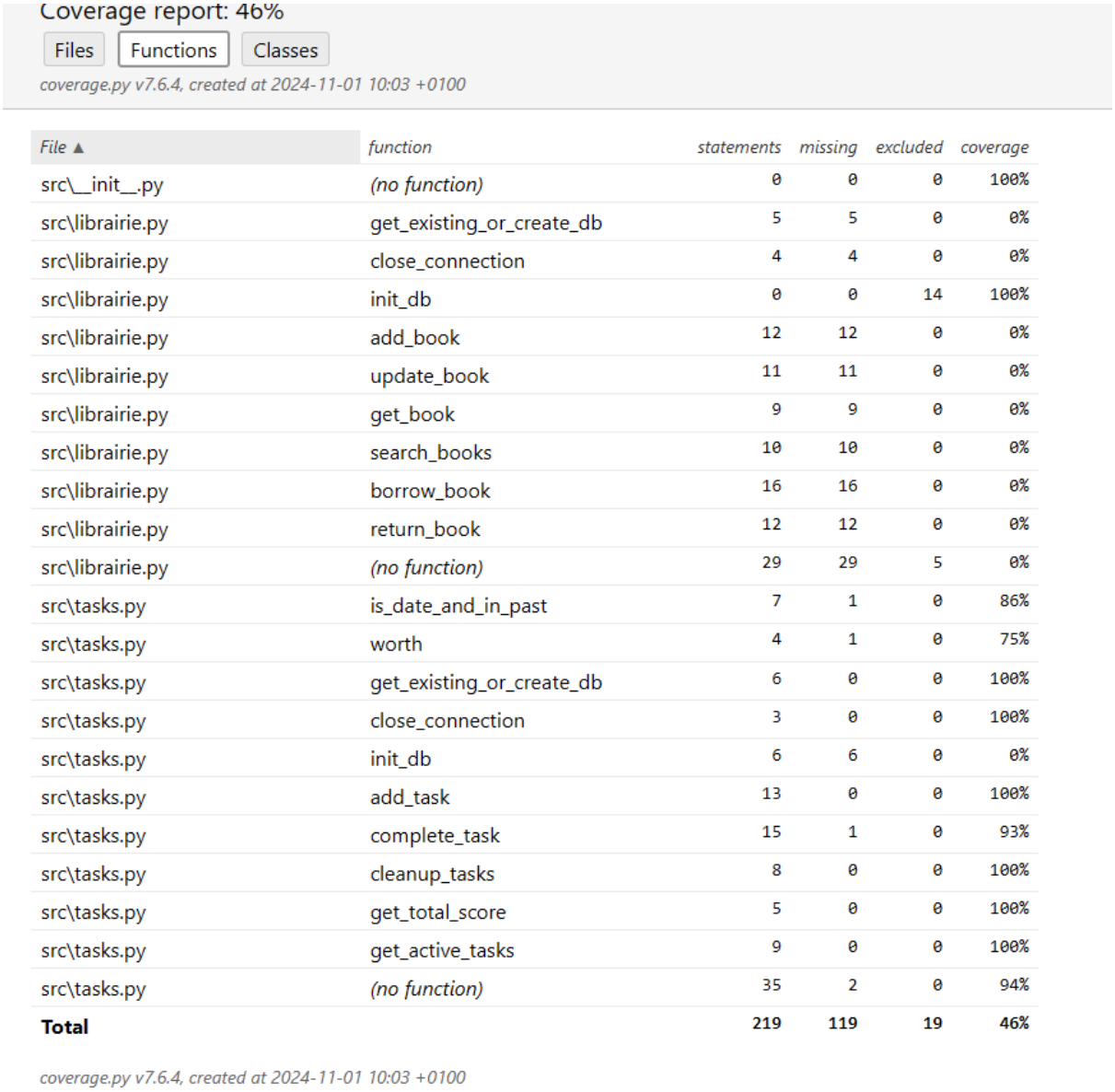
J'utilise la ligne de commande "pytest --cov=src --cov-report=html" dans le terminal.

Cela me crée un rapport de test html situé dans le dossier "htmlcover"

La première page me permet de voir le pourcentage de lignes de codes couvertes par les tests. Pour le programme "tasks" couvert à 90%.



Il me permet aussi d'avoir une vision fonction par fonction du taux de couvertures des tests.



Coverage report: 46%

Files | Functions | Classes

coverage.py v7.6.4, created at 2024-11-01 10:03 +0100

File ▲	function	statements	missing	excluded	coverage
src__init__.py	*(no function)*	0	0	0	100%
src\librairie.py	get_existing_or_create_db	5	5	0	0%
src\librairie.py	close_connection	4	4	0	0%
src\librairie.py	init_db	0	0	14	100%
src\librairie.py	add_book	12	12	0	0%
src\librairie.py	update_book	11	11	0	0%
src\librairie.py	get_book	9	9	0	0%
src\librairie.py	search_books	10	10	0	0%
src\librairie.py	borrow_book	16	16	0	0%
src\librairie.py	return_book	12	12	0	0%
src\librairie.py	*(no function)*	29	29	5	0%
src\tasks.py	is_date_and_in_past	7	1	0	86%
src\tasks.py	worth	4	1	0	75%
src\tasks.py	get_existing_or_create_db	6	0	0	100%
src\tasks.py	close_connection	3	0	0	100%
src\tasks.py	init_db	6	6	0	0%
src\tasks.py	add_task	13	0	0	100%
src\tasks.py	complete_task	15	1	0	93%
src\tasks.py	cleanup_tasks	8	0	0	100%
src\tasks.py	get_total_score	5	0	0	100%
src\tasks.py	get_active_tasks	9	0	0	100%
src\tasks.py	*(no function)*	35	2	0	94%
Total		**219**	**119**	**19**	**46%**

coverage.py v7.6.4, created at 2024-11-01 10:03 +0100

Ce rapport est très utile car il permet rapidement de voir les endroits ou les tests ne sont pas couverts.

C'est un moyen élégant de présenter les fonctions non couvertes.

Je préfère personnellement utiliser la ligne de commande
"pytest --cov=src --cov-report term-missing tests", qui donne les lignes de codes non couverts directement dans le terminal.