TP2 - Mocking et Fixtures

Objectifs

- Comprendre pourquoi mocker les dépendances externes
- Apprendre à créer et configurer des mocks
- Maîtriser les fixtures pour organiser les tests
- Tester des fonctions avec dépendances réelles

Prérequis

- Python 3.6+ installé
- Module `requests` : `pip install requests`

Structure du projet

```
tp_mocking/
├── weather_service.py    # À créer
└── test_weather.py      # À créer
```

Partie 1 : Le problème des dépendances

Étape 1 : Créer un service météo

Créez `weather_service.py` avec cette fonction :

```python
import requests

def get_temperature(city):
    """Récupère la température d'une ville via une API"""
    url = f"http://api.openweathermap.org/data/2.5/weather"
    params = {
        'q': city,
        'appid': 'fake_api_key',  # Clé bidon pour ce TP
        'units': 'metric'
    }

    response = requests.get(url, params=params)

    if response.status_code == 200:
        data = response.json()
        return data['main']['temp']
    else:
        return None
```

Étape 2 : Tester sans mock (problème)

Créez `test_weather.py` et essayez ce test :

```
import unittest
from weather_service import get_temperature

class TestWeather(unittest.TestCase):

    def test_get_temperature_paris(self):
        """Test basique qui va poser problème"""
        temp = get_temperature("Paris")
        # Comment tester ça ? L'API peut être en panne, lente, différente...
        self.assertIsNotNone(temp)  # Test très faible

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

Lancez ce test : `python test_weather.py`

Questions :

1. Que se passe-t-il si vous n'avez pas internet ?
2. Comment tester le cas où l'API retourne une erreur ?
3. Comment être sûr que votre fonction gère bien les différents cas ?

Partie 2 : Introduction au mocking

Étape 3 : Votre premier mock

Modifiez `test_weather.py` :

```
import unittest
from unittest.mock import patch, Mock
from weather_service import get_temperature

class TestWeather(unittest.TestCase):

    @patch('weather_service.requests.get')
    def test_get_temperature_success(self, mock_get):
        """Premier test avec mock - À COMPLÉTER"""

        # 1. CRÉEZ un objet Mock pour simuler la réponse HTTP
        fake_response = Mock()
        fake_response.status_code = 200

        # 2. CRÉEZ les données JSON que l'API retournerait
        fake_response.json.return_value = {
            'main': {
```

```
                'temp': 25.5
            }
        }

        # 3. CONFIGUREZ le mock pour retourner votre fake_response
        mock_get.return_value = fake_response

        # 4. TESTEZ votre fonction
        result = get_temperature("Paris")

        # 5. VÉRIFIEZ le résultat
        # TODO: Vérifiez que result == 25.5

        # 6. VÉRIFIEZ que requests.get a été appelé correctement
        # TODO: Utilisez mock_get.assert_called_once_with() pour vérifier l'URL et
les paramètres

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

À faire :

- Complétez les TODO
- Lancez le test et vérifiez qu'il passe
- Observez : plus besoin d'internet !

Partie 3 : Tester les cas d'erreur

Étape 4 : Test du cas d'erreur 404

Ajoutez ce test (à compléter) :

```
@patch('weather_service.requests.get')
def test_get_temperature_city_not_found(self, mock_get):
    """Test quand la ville n'existe pas"""

    # TODO: Créez un Mock qui retourne status_code = 404

    # TODO: Configurez mock_get.return_value

    # TODO: Testez get_temperature("VilleInexistante")

    # TODO: Vérifiez que le résultat est None

    pass  # Remplacez par votre code
```

Questions :

- Que doit retourner votre fonction si status_code != 200 ?

- Comment vérifier que l'appel API a bien été fait ?

Étape 5 : Test avec exception réseau

Ajoutez ce test :

```
@patch('weather_service.requests.get')
def test_get_temperature_network_error(self, mock_get):
    """Test quand il y a une erreur réseau"""

    # TODO: Configurez le mock pour lever une exception
    # Indice: mock_get.side_effect = requests.exceptions.RequestException()

    # TODO: Testez que votre fonction gère l'exception
    # Vous devrez peut-être modifier weather_service.py pour gérer ce cas

    pass
```

Partie 4 : Fixtures et données de test

Étape 6 : Organiser vos données avec setUp

Refactorisez vos tests pour éviter la duplication :

```
class TestWeather(unittest.TestCase):

    def setUp(self):
        """Fixture : prépare les données avant chaque test"""
        # TODO: Créez self.sample_weather_data avec des données météo types
        # TODO: Créez self.test_city avec une ville de test
        pass

    @patch('weather_service.requests.get')
    def test_get_temperature_success(self, mock_get):
        """Test avec données de la fixture"""
        fake_response = Mock()
        fake_response.status_code = 200
        # TODO: Utilisez self.sample_weather_data ici
        fake_response.json.return_value = self.sample_weather_data

        mock_get.return_value = fake_response

        # TODO: Utilisez self.test_city
        result = get_temperature(self.test_city)

        # TODO: Complétez les assertions
```

Partie 5 : Service plus complexe

Étape 7 : Ajouter une fonction avec multiple dépendances

Ajoutez dans `weather_service.py` :

```
import json
from datetime import datetime

def save_weather_report(city, filename="weather_log.json"):
    """Récupère la météo et la sauvegarde dans un fichier"""

    # 1. Récupérer la température
    temp = get_temperature(city)
    if temp is None:
        return False

    # 2. Créer le rapport
    report = {
        'city': city,
        'temperature': temp,
        'timestamp': datetime.now().isoformat()
    }

    # 3. Sauvegarder dans le fichier
    try:
        # Lire le fichier existant
        with open(filename, 'r') as f:
            reports = json.load(f)
    except FileNotFoundError:
        reports = []

    reports.append(report)

    with open(filename, 'w') as f:
        json.dump(reports, f)

    return True
```

Étape 8 : Tester la fonction complexe

Ajoutez cette classe de test :

```
from unittest.mock import mock_open, patch
from weather_service import save_weather_report

class TestWeatherReport(unittest.TestCase):

    def setUp(self):
        # TODO: Préparez vos données de test
```

```
        pass

    @patch('weather_service.datetime')
    @patch('builtins.open', new_callable=mock_open, read_data='[]')
    @patch('weather_service.get_temperature')
    def test_save_weather_report_success(self, mock_get_temp, mock_file,
mock_datetime):
        """Test sauvegarde rapport météo - EXERCICE PRINCIPAL"""

        # TODO: Configurez mock_get_temp pour retourner 20.5

        # TODO: Configurez mock_datetime.now().isoformat() pour retourner une date
fixe

        # TODO: Appelez save_weather_report("Paris")

        # TODO: Vérifiez que le résultat est True

        # TODO: Vérifiez que get_temperature a été appelé avec "Paris"

        # TODO: Vérifiez que le fichier a été ouvert en lecture puis en écriture

        pass
```

Indices :

- `mock_get_temp.return_value = 20.5`
- `mock_datetime.now.return_value.isoformat.return_value = "2024-01-01T12:00:00"`
- `mock_file.assert_called_with()` pour vérifier les appels de fichier

Partie 6 : Tests paramétrés (bonus)

Étape 9 : Tester plusieurs villes d'un coup

Ajoutez ce test paramétré :

```
@patch('weather_service.requests.get')
def test_multiple_cities(self, mock_get):
    """Test plusieurs villes avec une seule méthode"""

    cities_and_temps = [
        ("Paris", 25.0),
        ("Londres", 18.5),
        ("Tokyo", 30.2)
    ]

    for city, expected_temp in cities_and_temps:
        with self.subTest(city=city):
            # TODO: Configurez le mock pour cette ville
            # TODO: Testez get_temperature(city)
```

```
        # TODO: Vérifiez le résultat
        pass
```

Partie 7 : Réflexion et bonnes pratiques

Questions d'analyse

1. **Avant/Après** : Comparez vos premiers tests (sans mock) avec les derniers (avec mock). Quels avantages voyez-vous ?
2. **Isolation** : Pourquoi est-il important que chaque test soit indépendant ?
3. **Réalisme** : Les mocks sont-ils "réalistes" ? Comment s'assurer qu'ils correspondent à la vraie API ?

Bonnes pratiques observées

- **Nommage** : `test_fonction_cas_specifique`
- **Structure** : Arrange (mock) → Act (appel) → Assert (vérification)
- **Isolation** : Chaque test vérifie un seul comportement
- **Fixtures** : Données partagées dans `setUp()`

Pour aller plus loin

Exercice final (optionnel)

Créez une classe `WeatherService` avec ces méthodes :

- `get_forecast(city, days=5)` : prévisions sur N jours
- `is_good_weather(city)` : retourne True si > 20°C et pas de pluie
- `compare_cities(city1, city2)` : compare les températures

Défi : Écrivez les tests avec mocks pour ces 3 fonctions !

Récapitulatif des concepts

- **Mock** : Objet factice qui simule un comportement
- **Patch** : Remplace temporairement un objet par un mock
- **Fixture** : Prépare l'environnement de test
- **Isolation** : Chaque test est indépendant des autres
- **Déterminisme** : Résultats prévisibles grâce aux mocks

Bravo ! Vous savez maintenant tester du code avec des dépendances ! 🎯