

A vintage illustration of a lion tamer in a cage, surrounded by several large, roaring lions and tigers. The tamer is dressed in a red and gold costume with a plumed hat. He is holding a whip and appears to be interacting with the animals. The scene is set in a cage with a stone wall and spectators visible in the background.

DOMPTER LES MOCKS !

INTRODUCTION

JULIEN LENORMAND

Slides : <https://github.com/Lenormju/talk-mocking-python/>



POURQUOI PARLER DES MOCKS ?

- sujet un peu has been ?
- outil pratique !
- mais difficile à utiliser ...

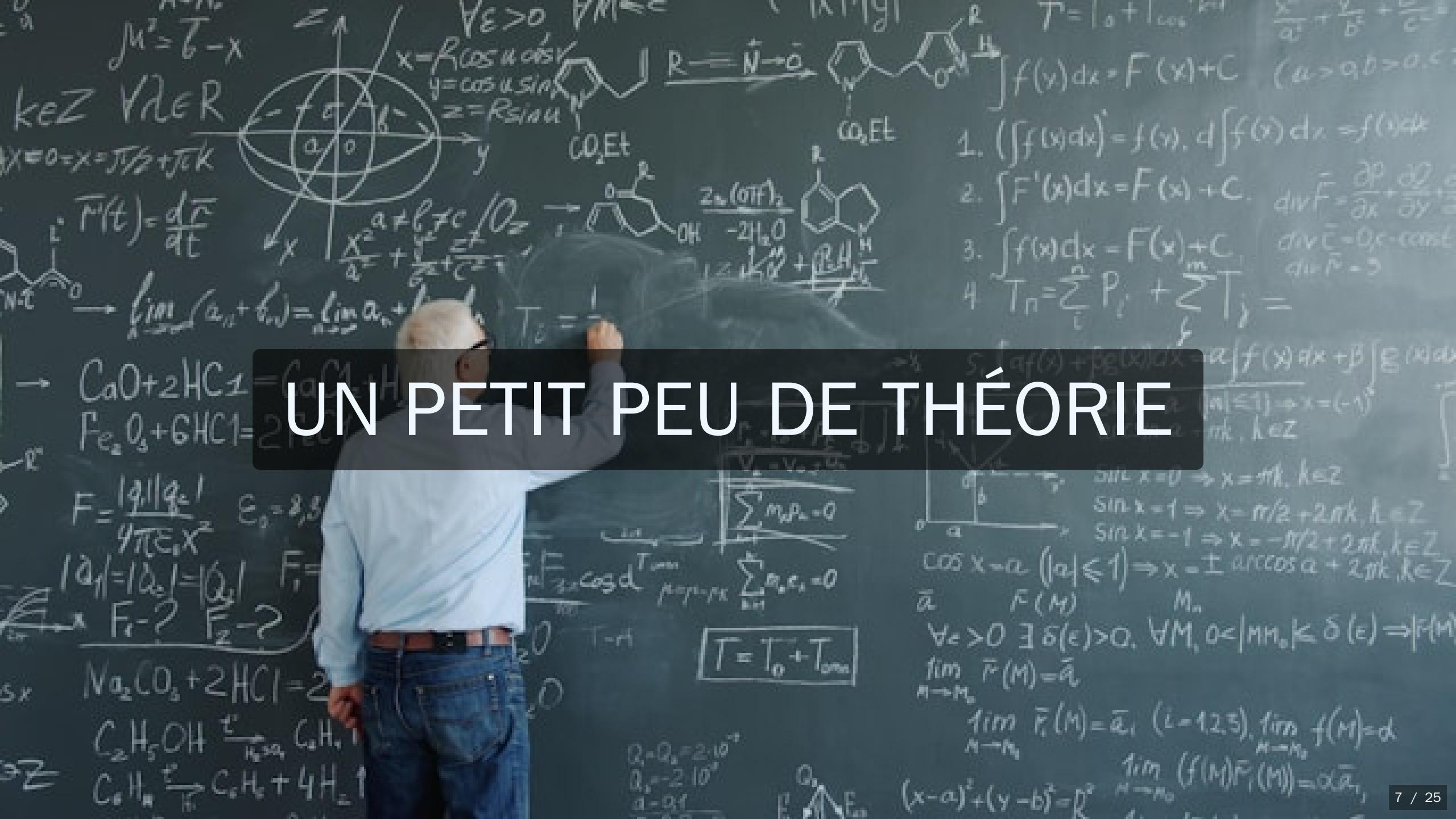
SONDAGE

1. qui ne connaît pas du tout et veut découvrir ?
2. qui a déjà eu du mal à les utiliser et veut s'améliorer ?
3. qui maîtrise déjà et va un peu s'ennuyer ?

OBJECTIF

- Comprendre ce qu'est un mock
- Connaître les problèmes qu'il résout
 - et qu'il cause (!)
- Savoir les appliquer
 - sans trop galérer
 - et donc savoir les débugger !

UN PETIT PEU DE THÉORIE



A QUOI ÇA RESSEMBLE ?

```
from unittest.mock import Mock
```

```
my_mock = Mock()  
str(my_mock) # <Mock id='127122938802736'>
```

```
my_mock.toto # pas d'erreur !  
# <Mock name='mock.toto' id='127122938803408'>
```

```
my_mock.toto.titi().tutu # toujours pas d'erreur !  
# <Mock name='mock.toto.titi().tutu' id='127122938804080'>
```

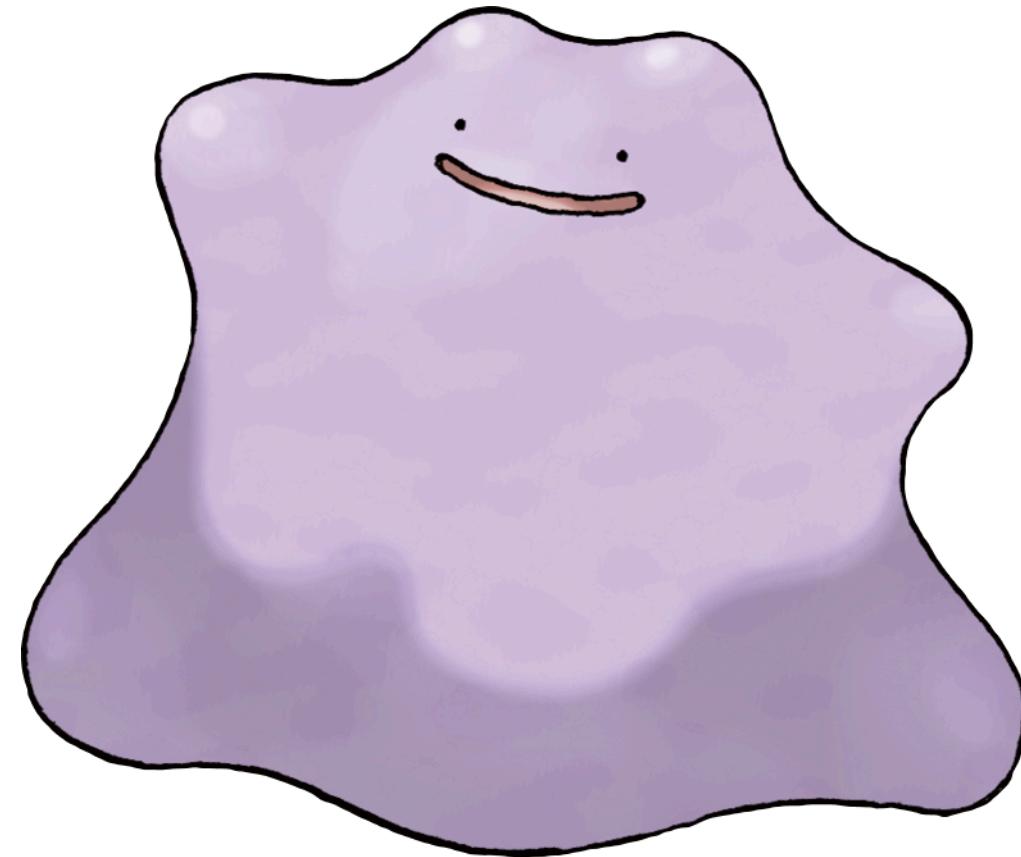
```
from unittest.mock import MagicMock

len(Mock())    # TypeError: object of type 'Mock' has no len()
len(MagicMock())  # 0
Mock()[0]      # TypeError: 'Mock' object is not subscriptable
MagicMock()[0]  # <MagicMock name='mock.__getitem__()' id='127122939480112'>
```

```
from unittest.mock import NonCallableMock

NonCallableMock()  # <NonCallableMock id='127122938805760'>
NonCallableMock().__() # TypeError: 'NonCallableMock' object is not callable
```

DES MÉTAMORPHES !



A QUOI ÇA SERT ?

- à tester
 - rappel : c'est --TRÈS-- important de tester !
 - "améliorer" des tests
 - FIRST : Fast, Independant, Repeatable, Self-sufficient, Timely
- .
1. vérifier ou bloquer des interactions à l'intérieur du code
 2. rendre pilotable certaines parties du code
 3. isoler des dépendances que je ne contrôle pas
- rendre testable du code qui ne l'est pas tellement / pas du tout

Quelques exemples de ce qui rend difficile de tester :

- le code interagit avec une API, dont j'ai pas le login
- ou l'API est payante
- ou l'API est très lente
- ou l'API est instable
- ou l'API n'existe même pas encore !
- ou l'API renvoie parfois des erreurs
- ou le code dépend de l'heure actuelle
- ou dépend de l'aléatoire
- ou écrit son résultat sur stdout
- ...

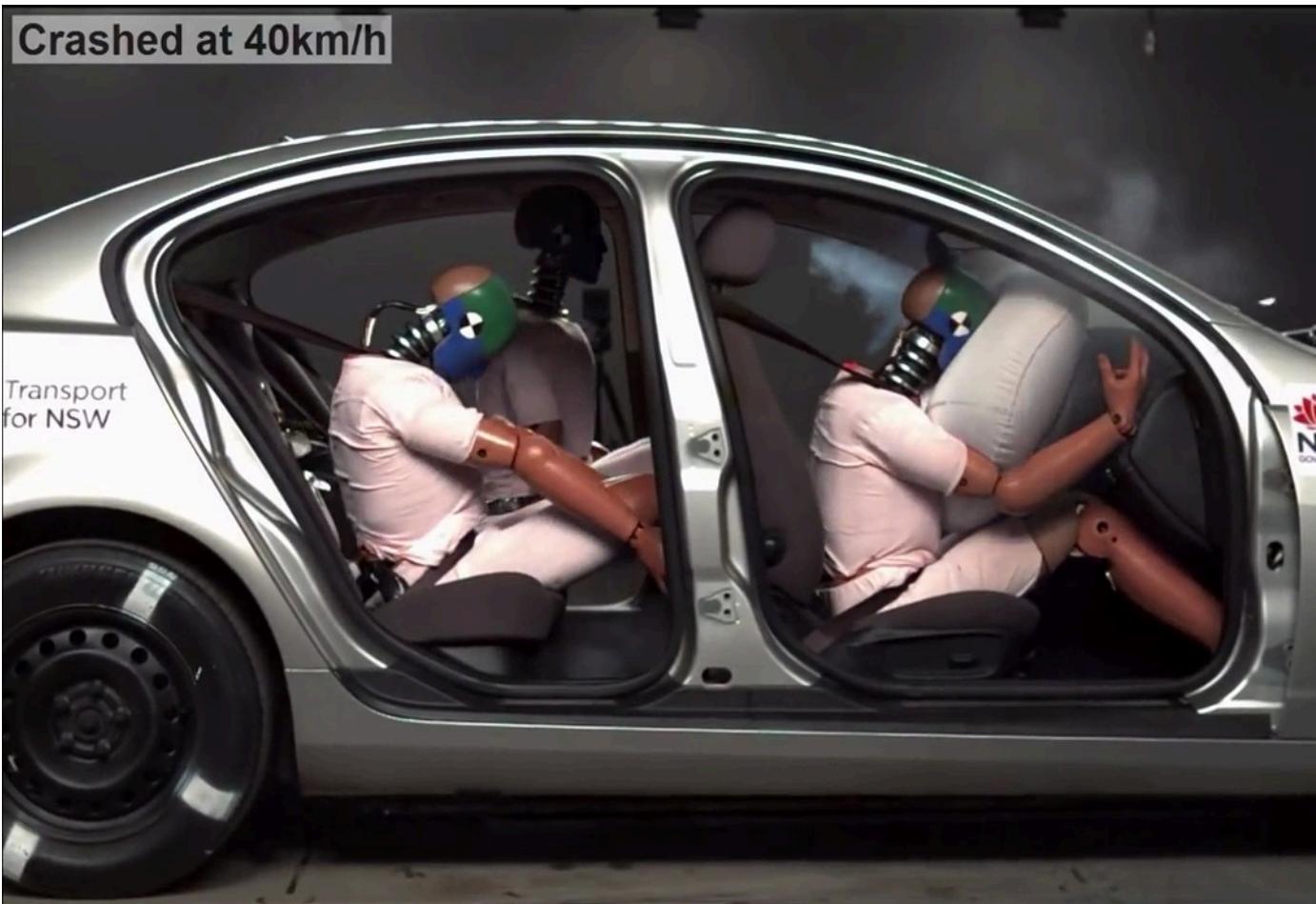
Solution : tricher, modifier le programme

DES DOUBLURES ?



- dummy ou stub ou spy ou fake ou mock ?
- en Python, on ne fait que des Mock !

Crashed at 40km/h



MISE EN PRATIQUE !

MÉTÉO INCONTRÔLABLE !



CONCLUSION

CONCLUSION

- les mocks c'est bien : isolation, perf, économies, ...
- les mocks c'est mal : isolation, fragilité, complexité, ...
- repenser l'architecture et la stratégie de test :
 - design feedback
 - adapter et facade patterns (narrow width)
 - inversion de dépendance (SOLID)
 - architecture hexagonale etc...
 - fakes --> simulateurs
 - TestContainers
 - contract testing
 - ...
 - mock en dernier recours

POUR ALLER + LOIN

- concernant les mocks :
 - [unittest.mock](#) — mock object library
 - [unittest.mock](#) — getting started
 - [Julien Lenormand](#) - Tout comprendre des mocks Python
 - [Martin Fowler](#) - Mocks Aren't Stubs
 - [Hynek Schlawack](#) - “Don’t Mock What You Don’t Own” in 5 Minutes
 - [Philippe Bourgau](#) - Careless Mocking Considered Harmful
 - [Ned Batchelder](#) - Why your mock doesn’t work
 - [mock_open](#), [fakefs](#), fakes locaux Cloud, [responses](#), [VCR.py](#), [io.StringIO](#), ...
- sur les tests auto :
 - [Julien Lenormand](#) et [Jonathan Gaffiot](#) - L'enfer des tests autos, bientôt en replay
 - [Brett Schuchert](#) et [Tim Ottinger](#) - tests "FIRST"
 - [Robert C. Martin \(Uncle Bob\)](#) - Test Contra-variance
- sur l'architecture :
 - [Julien Lenormand](#) - Une approche différente de l'architecture hexagonale et replay vidéo
 - [adapter](#) ou [facade](#) design patterns
 - programmation fonctionnelle (fonctions pures)

CRÉDITS PHOTOS

- Степана on Unsplash - Brown rabbit on window during daytime
- Godfrey Nyangechi on Unsplash - Black flat screen computer monitor
- Vitaly Gariev on Unsplash - Professor writing complex mathematical equations on a chalkboard
- Sebastian Coman Photography on Unsplash - Small appetizers arranged on a silver plate
- Internaute.com - Les acteurs et leurs doublures
- pokepedia.fr - Métamorph
- wikimedia.org - Lion tamer
- wikimedia.org - Crash test with airbag and safety belts
- Valerii Mishanov - The trainer puts his head in the lion's mouth

REMERCIEMENTS

- les sponsors, qui ont rendu cet évènement possible et accessible
- Lucie Anglade pour la proposition inespérée !
 - (et la personne qui s'est décommandée)
- Bas Terwijn pour la lib [memory_graph](#)
- Kaizen Solutions pour m'avoir sponsorisé pour l'écriture de mon premier article



QUESTIONS

LinkedIn : Julien Lenormand

Slides : <https://github.com/Lenormju/talk-mocking-python/>



EXTRA (EN VRAC)

1. lib `mock` est pour Python < 3.3 (2012, ajout à la stdlib),  Michael Foord
2. donner un `name` aux mocks, pour faciliter le debug
3. école *mockist* de London versus école *classicist* de Detroit
4. Dependency Inversion != Dependency Injection
5. Pytest = assert efficace, pas besoin de `class`, des fixtures, des plugins
6. Comment mesurer la fiabilité des tests ? Et leur maintenabilité ?
7. Définition de "test unitaire" ? Structurel, technique, fonctionnel ?
8. Définition de "legacy" ?
9. Quel intérêt au "collaboration testing" (tester le "how") ?

ABSTRACT

Si vous avez déjà essayé de mocker, vous avez sûrement eu du mal à bien les appliquer. Si vous en avez plein vos tests, vous avez sûrement remarqué qu'ils sont faillibles et fragiles. Mais ils peuvent aussi être très puissants, ce qui les rend très utiles pour le test.

Afin de les utiliser efficacement, il faut d'abord comprendre vraiment comment ils fonctionnent, leur interaction avec le système d'import, le modèle objet de Python, et l'architecture des programmes. En particulier, il faudra apprendre à discerner les "références" que Python utilise partout (des sortes de "pointeurs"), et les concepts d'immutabilité et de passage-par-copie/référence. Avec tous ces savoirs, vous pourrez prédire le comportement des mocks, et réussir à les utiliser sereinement. Vous confronterez alors cette compréhension à des exercices concrets de comment appliquer des mocks. Enfin, vous aurez des conseils d'outils à utiliser pour tirer un maximum de vos mocks, mais surtout de quand ne pas les utiliser.