# Tests pour des programmes en ligne de commande

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 2 mars 2020

https://www.bortzmeyer.org/tests-commandes-shell.html

---

Tout développeur sait bien qu'il faut tester les programmes. Et il ou elle sait également qu'il doit s'agir de tests entièrement automatiques, pouvant être exécutés très souvent, idéalement à chaque changement du code ou de l'environnement. Il existe une quantité incroyable d'outils pour faciliter l'écriture de tels tests, pour tous les langages de programmation possibles. Souvent, ces outils sont même inclus dans la bibliothèque standard du langage, tellement les tests sont indispensables. Mais, lorsque j'ai eu besoin d'écrire un jeu de tests pour un programme en ligne de commande, je me suis aperçu qu'il n'existait pas beaucoup d'outils. Heureusement, Feth en a développé un, **test_exe_matrix**.

Les cadriciels existants pour écrire des tests sont typiquement spécifiques d'un langage de programmation donné. Le programme est structuré sous forme d'une ou plusieurs bibliothèques, dont les fonctions sont appelées par le programme de test, qui regarde si le résutat est celui attendu. Pour prendre un exemple simple, le langage Python a à la fois un cadriciel standard, unittest <https://docs.python.org/3/library/unittest.html>, et une alternative, que je trouve plus agréable, pytest <https://pytest.org/>. Pour illustrer, écrivons un `test_add.py` qui sera un jeu de tests de l'opération d'addition :

```
#!/usr/bin/env python3

import random

import pytest

@pytest.fixture(scope="module")
def generator():
    g = random.Random()
    yield g

def test_trivial():
    assert 2 + 2 == 4

def test_negative():
    assert 42 + (-42) == 0

def test_zero(generator):
    for i in range(0,10):
        x = generator.randint(0, 100)
        assert x + 0 == x
```

pytest va exécuter toutes les fonctions dont le nom commence par `test_`. Si l'assertion réussit, c'est bon, si elle échoue, pytest affiche le problème (en utilisant l'introspection, c'est un de ses avantages). Il suffit donc au développeur qui aurait modifié la mise en œuvre de l'opérateur + de lancer pytest et il verra bien si le code marche toujours ou pas. (Inutile de dire qu'un vrai jeu de test pour l'opérateur d'addition serait bien plus détaillé.)

Tout cela, c'est bien connu. Mais si je veux tester du code qui n'est pas une bibliothèque mais un exécutable en ligne de commande ? Il peut y avoir plusieurs raisons pour cela :

— C'est un programme privateur dont je n'ai pas les sources,
— Je veux tester l'analyse des arguments,
— Le programme n'est pas structuré sous forme d'une bibliothèque, tout est dans l'exécutable.

Étant à la fois dans le deuxième et dans le troisième cas, j'ai cherché une solution pour le tester. Mon cahier des charges était :

— Permettre de tester des programmes en ligne de commande Unix.
— Il faut pouvoir spécifier les arguments et le code de retour attendu.
— Il faut pouvoir spécifier des chaînes de caractères qui doivent être sur la sortie standard et l'erreur standard.

Comme indiqué plus haut, Feth Arezki `<https://framagit.org/feth>` a développé un outil qui correspond à ce cahier des charges, test_exe_matrix `<https://framagit.org/feth/test_exe_matrix>`. Il est bâti au dessus de pytest `<https://pytest.org/>` (ce qui est raisonnable, cela évite de tout refaire de zéro). Le principe est le suivant : on écrit un fichier en YAML décrivant les commandes à lancer et les résultats attendus. Puis on lance test_exe_matrix, et il indique si tout s'est bien passé ou pas, et en cas de problèmes, quels tests ont échoué. Voici un exemple de fichier de configuration avec un seul test, un test de la commande `echo` :

```
---
tests:
  - exe: '/bin/echo'
    name: "test echo stdout substring"
    args:
      - '-n'
      - 'coincoin'
    retcode: 0
    partstdout: 'inco'
    stderr: ''
```

La commande testée sera `/bin/echo -n coincoin`. Le paramètre `retcode` indique que le code de retour attendu est 0 (un succès, dans le shell Unix) et le paramètre `partstdout` indique qu'on attend cette chaine de caractères `inco` sur la sortie standard. Si on avait utilisé `stdout`, test_exe_matrix aurait exigé que l'entièreté de la sortie standard soit égale à cette chaîne. Testons maintenant :

```
% test_exe_matrix tmp.yaml
========================================= test session starts ==========================================
platform linux -- Python 3.6.9, pytest-5.3.1, py-1.8.1, pluggy-0.13.1
Tests from /home/stephane/src/Tests/CLI/test_exe_matrix/tmp.yaml
rootdir: /home/stephane
collected 2 items

../../../../.local/lib/python3.6/site-packages/test_exe_matrix/test_the_matrix.py ..               [100%]

========================================== 2 passed in 0.12s ===========================================
```

C'est parfait, tous les tests passent.

Un peu plus compliqué, avec la commande curl :

```
---
tests:
  - exe: '/usr/bin/curl'
    args:
      - 'https://www.bortzmeyer.org/'
    retcode: 0
    partstdout: 'Blog Bortzmeyer'
    timeout: 3 # Depuis Starbucks, c'est long
```

On a ajouté le paramètre `timeout` qui met une limite au temps d'exécution. Comme précédement, `test_exe_matrix tmp.yaml` va exécuter les tests et montrer que tout va bien.

Comme tous les bons cadriciels de test, test_exe_matrix permet aussi de tester des choses qui sont censées échouer, ici avec un domaine qui n'existe pas :

```
---
tests:
    - exe: '/usr/bin/curl'
      args:
          - 'https://www.doesnotexist.example/'
      retcode: 6
```

On oublie souvent de tester que, non seulement le logiciel marche bien dans les cas normaux, mais aussi qu'il échoue à juste titre lorsque la situation l'exige. Ici, on vérifie que curl a bien renvoyé le code de retour 6 et non pas 0. D'ailleurs, c'est l'occasion de montrer ce que fait test_exe_matrix lorsqu'un test échoue. Si je mets `retcode: 0` dans le fichier de configuration précédent :

```
% test_exe_matrix tmp.yaml
...
================================================ FAILURES ================================================
...
exetest = {'args': ['https://www.doesnotexist.example/'], 'exe': '/usr/bin/curl', 'retcode': 0, 'timeout': 1}
...
E           AssertionError: expected retcode is 0, but process ended withcode 6
E           assert 0 == 6
...
../../../../.local/lib/python3.6/site-packages/test_exe_matrix/test_the_matrix.py:127: AssertionError
=========================================== 1 failed, 1 passed in 0.14s ============================================
```

En utilisant l'introspection, test_exe_matrix a pu montrer exactement quel test échouait.

À l'heure actuelle, test_exe_matrix est en plein développement, et l'installation est donc un peu rugueuse. Si vous n'avez pas déjà Poetry <https://python-poetry.org/>, il faut l'installer, par exemple avec (oui, c'est dangereux, je sais) :

```
% curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py | python3
```

Une fois que vous avez Poetry, vous pouvez installer test_exe_matrix, par exemple avec :

```
% poetry build
% pip3 install dist/test_exe_matrix-0.0.18-py3-none-any.whl
```

Bien sûr, avant de suggérer le développement d'un nouveau logiciel, j'ai regardé ce qui existait, et ait été très surpris de ne rien trouver qui corresponde à mon cahier des charges, qui semblait pourtant modeste :

— Bats <https://github.com/sstephenson/bats> : il faut faire pas mal de shell bash pour l'utiliser.
— Btest <https://github.com/zeek/btest> est proche de ce que je cherche mais le test du code de retour n'est que binaire (0 ou pas 0), et on ne peut pas indiquer une chaîne qui doit être présente dans la sortie. Mettre un `| grep <chaine>` à la fin de la commande, ferait perdre le code de retour de la commande principale. (Dam_ned <https://mamot.fr/@Dam_ned> note qu'on peut peut-être s'en tirer avec `set -o pipefail` ou utiliser le tableau `$PIPESTATUS`.)
— Avec Lift <https://github.com/Malizor/lift>, je ne vois pas comment indiquer des chaînes de caractères à tester dans la sortie standard.
— Roundup <http://bmizerany.github.io/roundup/> : lui aussi semble exiger d'écrire pas mal de code shell.
— Quant à Aruba <https://github.com/cucumber/aruba/>, je n'ai tout simplement pas compris comment ça marchait.
— Testing-suite <https://github.com/asrospie/testing-suite> : très simple mais ne semble pas permettre de tester le code de retour.