Chapitre 1-4
Installer son environnement de travail

1. Introduction

Il ne s'agit ici que de CPython, l'implémentation de référence de Python, et non de PyPy ou Jython.
Quel que soit votre système d'exploitation, vous pouvez installer Python en lisant ce chapitre puis, dans un second temps, installer des bibliothèques tierces au gré de vos besoins (cf. section Installer une bibliothèque tierce) et vous pourrez créer des environnements virtuels (cf. section Créer un environnement virtuel).
Si vous souhaitez installer d'un seul coup Python ainsi que Jupyter (anciennement IPython) et la plupart des bibliothèques scientifiques ou d'analyse de données, vous pouvez aller directement à la section Installer Anaconda, pour installer celui-ci en lieu et place de Python. Vous disposerez alors d'autres méthodes pour gérer les environnements virtuels et pour installer des bibliothèques tierces.

2. Installer Python

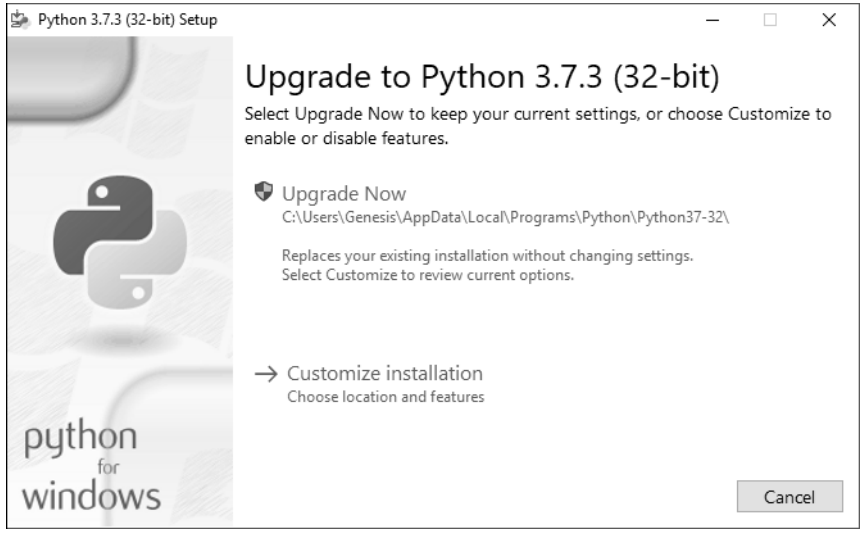
2.1 Pour Windows

Le système d'exploitation Windows requiert usuellement l'utilisation d'un installateur pour pouvoir installer un logiciel quel qu'il soit. Si vous disposez de Windows, vous devriez en avoir l'habitude. Python ne déroge pas à la règle.
Pour installer Python, vous devez donc aller sur le site officiel (https://www.python.org/downloads/) pour télécharger l'installateur adéquat. Comme vous pourrez le constater, on vous met en avant un accès rapide à la dernière version (au moment où ces lignes sont écrites, la 3.11.0), puis un accès aux dernières versions encore actives (actuellement la version 3.10 qui reçoit encore des corrections d'anomalies, puis les versions 3.9 à 3.7 qui reçoivent des corrections de sécurité uniquement).

Il est également possible de télécharger la toute dernière version de la branche 2.7 qui est en fin de vie (elle n'est plus mise à jour), car il existe encore de nombreux projets n'ayant pas encore migré.
Le support correctif dure 2 ans après la première sortie de la version et le support de sécurité dure 5 ans.
Pour notre part, nous vous conseillons la dernière 3.x, mais vous êtes libre d'installer celle que vous souhaitez ou même d'en installer plusieurs suivant vos contraintes, il n'y a pas d'objection à cela.
Une fois le téléchargement effectué, vous devez lancer l'installateur (et éventuellement passer quelques protections de votre système qui vous demande d'accorder votre confiance à cet installateur), pour observer l'écran suivant :

Comme vous pouvez le constater, il est possible de personnaliser l'installation en choisissant le chemin d'installation du logiciel ou en choisissant de ne pas sélectionner quelques fonctionnalités, mais nous ne le conseillons pas.
Nous vous recommandons en revanche de cocher la case **Add python.exe to PATH** afin de configurer la variable PATH du terminal pour rendre Python accessible plus facilement.

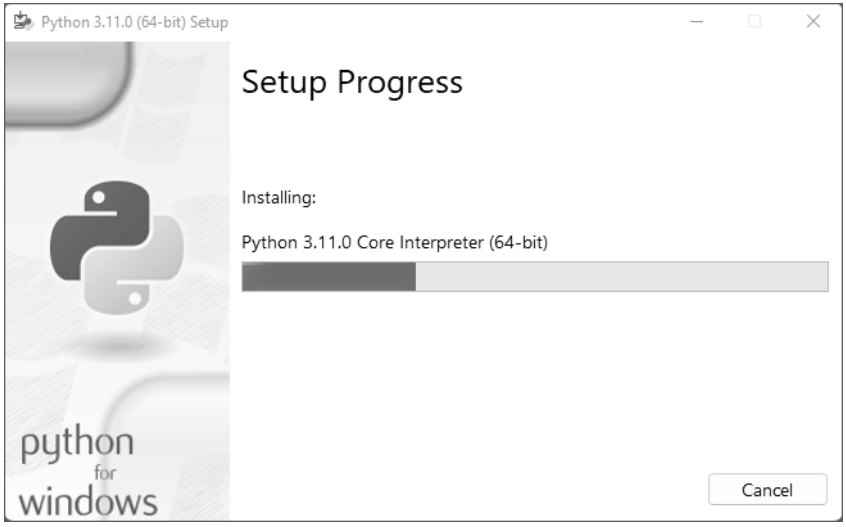
Si vous avez déjà une ancienne version de Python installée de la même branche (dans cet exemple, Python 3.7.2 est déjà installé), vous pourrez la mettre à jour à l'aide du même installeur :



Par contre, si vous avez déjà la version 3.7.1 et que vous installez la version 3.11, cette dernière ne viendra pas remplacer la précédente, mais s'installera à côté. Si vous souhaitez remplacer, il vous faudra donc désinstaller proprement la toute dernière version installée, ce qu'il est possible de faire en relançant l'installeur d'origine.

Nous vous encourageons à garder les installeurs sur votre PC, car ils pourraient devenir indisponibles au téléchargement si trop vieux.

Quel que soit le scénario, vous arriverez devant un écran vous montrant la progression de l'installation et vous n'aurez qu'à fermer la fenêtre une fois celle-ci terminée :



Vous êtes maintenant prêt à utiliser Python.

2.2 Pour Mac

Il faut savoir qu'une version de Python est déjà préinstallée sur Mac, car Mac OS X l'utilise pour ses propres besoins et Python est intégré à son propre cycle de développement. Cependant, si vous souhaitez une version différente de celle qui est déjà présente, vous pouvez l'installer, sachant qu' il n'y a pas de contre-indication à posséder plusieurs versions de Python sur la même machine.

Pour installer Python sur Mac OS X, la procédure à suivre est similaire à celle pour Windows. Il faut donc se rendre sur le site officiel (https://www.python.org/downloads/mac-osx/), télécharger un installateur correspondant à sa configuration et suivre les étapes.

Pour les utilisateurs de Mac, sachez que Python dispose d'une bonne intégration de ses spécificités, en particulier vis-à-vis de Objective-C, le langage de programmation avec lequel est développé Mac OS X, et Cocoa, interface de programmation de Mac OS X.

2.3 Pour GNU/Linux et BSD

Les différentes distributions libres utilisent nativement Python, notamment pour des parties sensibles. Python y est donc tout naturellement déjà installé, généralement sous la dernière version de la branche 2.x. Cependant, ici comme ailleurs, il n'y a pas d'objections à utiliser plusieurs versions de Python.
Le plus simple reste d'utiliser votre gestionnaire de paquets, ce qui peut se faire via un outil graphique, comme Synaptic pour Debian :

Il suffit alors de faire une recherche sur le mot-clé **`python`** pour voir les différentes versions (sur une ancienne Debian, ce sont Python 2.6, 2.7 et 3.2).
Par contre, tous les paquets python3-xxxxx que vous pouvez voir ici sont des bibliothèques tierces et non Python lui-même. Nous en parlerons plus tard dans ce chapitre.
Une fois les paquets souhaités sélectionnés, il ne manque plus qu'à les installer en cliquant sur le bouton **Appliquer**.
Notez que tout ceci peut se faire par la simple ligne de commande, toujours en utilisant votre gestionnaire de paquets qui peut être apt-get, aptitude, yum, emerge, pkg_add ou autre.
Voici par exemple pour une distribution Debian ou Ubuntu :

```
$ sudo aptitude install python3
```

Ceci ne permet cependant pas de choisir la version que l'on souhaite, à moins d'aller trouver des sources alternatives. Si l'on veut avoir la toute dernière version de Python, il faudra la plupart du temps passer par la compilation.

2.4 Par la compilation

Compiler Python n'est pas en soi une tâche très complexe. C'est par contre souvent une tâche imposée lorsque l'on ne travaille pas avec des conteneurs. En effet, en entreprise, on développe souvent des applications qui sont destinées à être hébergées. Il est alors impératif de travailler sur votre propre poste avec une version de Python qui soit la même que celle existante sur la machine de production.

Sous GNU/Linux, mais aussi sous d'autres systèmes, il est possible de compiler la version de Python que l'on souhaite. Après tout, Python n'est rien d'autre qu'un programme écrit en C. Pour ce faire, il faut aller télécharger le code source (https://www.python.org/downloads/source/), qui prend la forme d'une archive, puis décompresser celle-ci, se placer dans le répertoire ainsi obtenu et taper ces quelques commandes :

```
$ ./configure –prefix=/path/to/my/python/directory
$ make
$ sudo make altinstall
```

Notez que dans cette dernière ligne, nous n'utilisons pas la commande **`make install`**, qui aurait pour effet de remplacer votre Python système par le Python que vous compilez, ce qui pourrait avoir des conséquences indésirables voire désastreuses.

Notez également que vous choisissez lors de la configuration le chemin dans lequel vous placerez vos bibliothèques Python. En général, l'usage veut que l'on utilise **`/opt`**, mais il n'y a pas de règle, tout dépend des pratiques de votre entreprise ou votre expérience en la matière.

Si vous venez d'installer Python 3.5 par cette méthode, vous aurez alors maintenant accès à ce programme en l'appelant ainsi, depuis votre terminal :

```
$ python3.5
```

Par cette même méthode, vous pouvez installer les dernières versions (https://www.python.org/download/pre-releases/) de Python qui ne sont pas encore sorties (alphas ou betas), ce qui vous permet de les tester en avant-première !

Notons que, par cette méthode, toutes les bibliothèques de Python ne fonctionneront pas. En effet, lorsqu'elles ont besoin d'autres bibliothèques C, il faut effectuer des compilations croisées et utiliser les différents en-têtes de ces bibliothèques. C'est le cas par exemple pour faire fonctionner Curses, ReportLab (génération de fichiers PDF) ou encore PyUSB (accès aux ports matériels USB).

Dans ce cas-là, la commande **`./configure`** devra recevoir des arguments supplémentaires et vous devrez trouver un tutoriel en ligne pour vous indiquer la démarche, laquelle peut être plus ou moins complexe.

2.5 Pour un smartphone

Installer une machine virtuelle Python sur un smartphone est possible. Pour Android, la procédure est assez simple puisqu'il existe un produit dédié (http://qpython.com/), tout comme sur Windows Phone (https://apps.microsoft.com/store/detail/python-39/9P7QFQMJRFP7). Pour iOS, c'est une autre paire de manches (https://github.com/linusyang/python-for-ios) étant donné que l'utilisateur est enfermé dans un système sur lequel il n'a aucun contrôle.

3. Installer une bibliothèque tierce

Remarque

Si vous abhorrez le terminal, sachez que vous pouvez installer une bibliothèque tierce depuis votre IDE, ce qui sera probablement plus aisé pour vous.

3.1 À partir de Python 3.4

Pour installer une bibliothèque tierce, vous devez simplement connaître son nom. Celui-ci est généralement assez intuitif. Par exemple, la bibliothèque permettant de communiquer avec un serveur Redis s'appelle redis.
Il peut y avoir des variations. Par exemple, la bibliothèque de référence pour traiter du XML est lxml et, plus complexe, celle pour BeautifulSoup est bs4. En recherchant comment répondre à un besoin sur le Net ou sur PyPi (https://pypi.python.org/pypi), vous trouverez rapidement une bibliothèque de référence.
Sur des sujets plus confidentiels, il vous arrivera de trouver plusieurs petites bibliothèques. Vous pourrez alors les tester et choisir celle que vous utiliserez pour votre projet.
Sachez que vous pouvez aussi conduire une recherche directement depuis votre terminal :

```
$ pip search xml
$ pip search soup
```

Cela vous donnera une liste de bibliothèques accompagnée d'une courte description, à la manière de ce que font les gestionnaires de paquets sous Linux (lesquels sont écrits en Python, au passage).
Sachez que **`pip`** existe quel que soit votre système d'exploitation (vous devez être familier avec le terminal de votre système, cependant) et que depuis la version 3.4 de Python, il est installé automatiquement avec celui-ci. Si ce n'est pas votre cas, consultez la section suivante : Pour une version inférieure à Python 3.4.
`pip` est un outil formidable. Si vous utilisez une version de Python qui est celle du système, vous utiliserez alors la commande **`pip`** pour gérer les bibliothèques. Si vous utilisez une autre version, telle que Python 3.5, alors vous utiliserez la commande **`pip-3.5`**. Pour Python 3.3, ce sera **`pip-3.3`**. Dans les exemples suivants, il vous faudra prendre en compte cette particularité.
Cet outil vous permettra d'installer une bibliothèque à sa dernière version ainsi que toutes les bibliothèques dépendantes. En effet, il n'est pas rare qu'une bibliothèque de Python ait besoin d'une autre bibliothèque (ou de plusieurs) pour fonctionner. Par exemple, l'installation de redis se fait par cette commande :

```
$ pip install redis
```

On peut aussi choisir la version à installer :

```
$ pip install -Iv redis==2.10.5
```

Ou mettre à jour la bibliothèque à une version précise :

```
$ pip install -U redis==2.10.5
```

Ou à la dernière version :

```
$ pip install -U redis
```

Et on peut la désinstaller :

```
$ pip uninstall redis
```

Une fonctionnalité très importante permet d'obtenir la liste des bibliothèques déjà installées (quelle que soit la manière dont elles ont été installées) :

```
$ pip freeze
```

Ce que l'on peut mettre dans un fichier :

```
$ pip freeze > requirements.txt
```

Pour installer tous les paquets ainsi listés, il faut procéder ainsi :

```
$ pip install -r requirements/base.txt
```

Cette méthode est particulièrement utile dans le cadre d'un environnement virtuel ; nous y reviendrons.
Il est possible de retrouver des informations sur un paquet déjà installé :

```
$ pip show django-redis
---
Name: django-redis
Version: 4.3.0
Location: /path/to/my/env/lib/python3.4/site-packages
Requires: redis
```

On voit ici que le paquet **`django-redis`** a une dépendance vers **`redis`** : en l'installant, on installe automatiquement **`redis`**.
Mettre à jour ce paquet met à jour automatiquement les dépendances :

```
$ pip install -U django-redis
```

Si on ne veut pas mettre à jour les dépendances, on peut procéder ainsi :

```
$ pip install -U --no-deps django-redis
```

On peut aussi installer plusieurs bibliothèques en même temps :

```
$ pip install django-redis==4.3.0 bs4 lxml
```

Cette commande installera donc automatiquement redis s'il n'est pas installé, car il est déclaré comme dépendance.
Cette commande a cependant des limites. En effet, si vous installez une bibliothèque tierce qui utilise une bibliothèque C, vous devrez disposer des en-têtes C correspondants (paquets **`dev`** pour Debian ou **`devel`** pour Fedora). Il faut donc avoir un peu de pratique dans ce genre de situation pour savoir déjouer ces pièges.

Chapitre 3
Les tests unitaires en Python

1. Pourquoi les tests unitaires ?

Le lecteur peut être surpris de commencer son apprentissage par les tests unitaires avant de s'exercer aux tests fonctionnels. Malgré la différenciation de ces deux types de tests, ils n'en restent pas moins des tests. Ils se basent donc sur la même logique et les mêmes éléments de syntaxe.

Effectivement, en Python, un test fonctionnel s'écrit à travers la structure d'un test unitaire : quel que soit l'élément que nous testons, une fonction en test unitaire ou un comportement en test fonctionnel, nous posons une assertion à propos de cet élément et nous en vérifions la validité par la suite. Vous pouvez voir le principe d'assertion comme la condition d'une instruction conditionnelle, dans le sens où elle est soit vraie, i.e. le test est accepté, ou fausse, i.e. le test est rejeté.

2. Assertions

Nous pourrions penser qu'un test est une simple instruction conditionnelle `if`, car son résultat est un booléen, le test est validé ou non. Avec cette logique, l'écriture des tests serait rébarbative, non différenciée de l'écriture du code, et nécessiterait une convention interne de code pour bien organiser le code et les tests ainsi que leur validation. Il nous faudrait indiquer ce que nous devons afficher lorsqu'un test réussit, lorsqu'il échoue, indiquer un nouveau fichier principal pour lancer les tests, etc. Un réel bazar qui ne serait pas pratique donc pas maintenable.

Python intègre une instruction spécifique pour valider un test : `assert`.

```
assert test_conditionnel, [message]
```

L'instruction `assert` est directement suivie par le test et peut contenir un message à afficher lorsque le test est rejeté. L'instruction représentant le test doit toujours pouvoir être interprétée comme un booléen.

Si le test est valide, i.e. une instruction retournant `True`, l'instruction `assert` laisse l'interpréteur Python continuer l'exécution du test. Dans le cas contraire, elle renvoie une exception `AssertionError` qui stoppe l'exécution du reste du script.

```
def divide(a, b):
    assert b > 0, "Division par zéro impossible !!!"
    return a / b

if __name__ == '__main__':
    print("1 divisé par 2 :", divide(1, 2))
    print("1 divisé par 0 :", divide(1, 0))
    print("3 divisé par 2 :", divide(3, 2))
```

L'exemple précédent permet de tester qu'une division n'a pas de diviseur nul, ce qui n'est pas autorisé en mathématiques. Le premier appel à la fonction `divide` contenant l'assertion se déroule correctement, mais le deuxième appel lance une exception et stoppe l'exécution du script avec l'affichage suivant :

```
1 divisé par 2 0.5
Traceback (most recent call last):
  File "codeChapitre3.py", line 7, in <module>
```

```
    print("1 divisé par 0", divide(1,0))
                             ^^^^^^^^^^^
  File "codeChapitre3.py", line 2, in divide
    assert b > 0, "Division par zéro impossible !!!"
           ^^^^^
AssertionError: Division par zéro impossible !!!
```

Nous retrouvons bien le message que nous avons défini en argument de l'assertion dans la *traceback* de l'interpréteur Python. Si nous n'avions pas complété l'assertion avec ce message, nous n'aurions eu aucune information sur la raison du rejet du test, nous aurions simplement eu la ligne du script ayant soulevé l'exception, comme le montrent le code suivant et son affichage console :

```
def divide(a, b):
    assert b > 0
    return a / b

if __name__ == '__main__':
    print("1 divisé par 2", divide(1,2))
    print("1 divisé par 0", divide(1,0))
    print("3 divisé par 2", divide(3,2))

# Sortie console
1 divisé par 2 0.5
Traceback (most recent call last):
  File "codeChapitre3.py", line 7, in <module>
    print("1 divisé par 0", divide(1,0))
                             ^^^^^^^^^^^
  File "codeChapitre3.py", line 2, in divide
    assert b > 0
           ^^^^^
AssertionError
```

Remarque

L'argument représentant le message à afficher dans l'instruction `assert` de Python a une importance capitale pour des tests maintenables et clairs pour les développeurs.

3. Léger tour d'horizon des tests unitaires

Python étant un langage libre, ouvert et surtout avec une grande communauté très active, il existe une multitude de librairies de tests pour écrire des tests unitaires. Dans cette section, nous ne présenterons que les plus connues et/ou utilisées pour la rédaction de tests unitaires.

3.1 Script utilisé pour le comparatif

Afin de présenter les différentes librairies de tests en Python, nous allons implémenter un script représentant deux fonctions simples d'une calculette :

– une division entre deux numériques ;

– une soustraction entre deux numériques.

```
# Fonction de la division
def divide(a, b):
    return a / b

# Fonction de la soustraction
def substract(a, b):
    return a - b
```

3.2 Librairie Unittest

La librairie Unittest est historiquement la première librairie en Python permettant d'automatiser les tests unitaires.

Remarque

Pour le lecteur connaissant le langage Java, cette librairie s'inspire de l'API JUnit.

La librairie Unittest est installée avec l'installation de Python, nous pouvons donc la voir comme la version native des tests unitaires de Python.

Pour écrire un test avec cette librairie, nous devons l'importer dans le script, puis créer une nouvelle classe héritant de `TestCase` et représentant tous les tests que nous voulons effectuer. La convention est de déclarer une méthode par test : chaque méthode n'aura donc qu'une seule instruction d'assertion. Cette convention nous permet de mieux comprendre les tests qui sont rejetés et donc de corriger notre code plus rapidement et surtout de manière plus cohérente.

La librairie Unittest utilise ses propres instructions d'assertion, dont :

– La méthode `assertEqual(paramètre1, paramètre2)` testant que le premier paramètre retourne bien la valeur du deuxième paramètre.
– La méthode `assertTrue(condition)` qui lance une exception lorsque la condition est fausse.
– La méthode `assertFalse(condition)` qui est l'inverse de la méthode `assertTrue`.

Le lancement des tests de la classe se fait par l'appel de l'instruction `unittest.main()`.

```
import unittest
import calculette_fonctions

# Déclaration de la classe contenant tous les tests
class Test(unittest.TestCase):

    # une méthode par test
    # => un seul assert par méthode
    def testDivide(self):
          self.assertEqual(calculette_fonctions
.divide(5, 5), 1)

    def testSubstract(self):
          self.assertEqual(calculette_fonctions
.substract(5, 5), 0)

if __name__ == '__main__':
    # Lancement de tous les tests de la classe Test
    unittest.main()
```

Nos deux tests sont valides et nous obtenons l'affichage console suivant :

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Ajoutons maintenant un test qui sera rejeté en déclarant que *5 – 5* doit retourner 10.

```
import unittest
import calculette_fonctions


# Déclaration de la classe contenant tous les tests
class Test(unittest.TestCase):

    # une méthode par test
    # => un seul assert par méthode
    def testDivide(self):
          self.assertEqual(calculette_fonctions
.divide(5, 5), 1)

    def testRejectedSubstract(self):
          self.assertEqual(calculette_fonctions
.substract(5, 5), 10)

    def testSubstract(self):
          self.assertEqual(calculette_fonctions
.substract(5, 5), 0)

if __name__ == '__main__':
    # Lancement de tous les tests de la classe Test
    unittest.main()
```

L'un des avantages de la librairie Unittest, comparativement à la simple instruction `assert` de Python, vient du fait qu'un test rejeté n'interrompt pas les autres tests. Il est juste indiqué sur la sortie console que l'assertion est fausse.

```
.F.
======================================================================
FAIL: testRejectedSubstract (__main__.Test.testRejectedSubstract)

----------------------------------------------------------------------
Traceback (most recent call last):
  File "/ unittest-exemple.py", line 13, in testRejectedSubstract
    self.assertEqual(calculette.substract(5,5),10)
AssertionError: 0 != 10

----------------------------------------------------------------------
Ran 3 tests in 0.001s
```

3.3 Librairie DocTest

La librairie DocTest est également intégrée à la distribution officielle de Python, il n'y a donc pas d'installation à faire.

Elle se base sur la documentation des scripts avec les docstrings. En plus de donner une description de la fonction (paramètres et retour) dans la docstring, nous devons fournir des exemples d'interprétation par Python.

```
# Fonction de la division
def divide(a, b):
    """
          paramètre a, b : number
          retour : number
          Exemples :
          >>> divide(1, 2)
          0.5
          >>> divide(2, 1)
          1
    """
    return a / b

# Fonction de la soustraction
def substract(a, b):
```

```
    """
          paramètre a, b : number
          retour : number
          Exemples :
          >>> substract(1, 2)
          -1
          >>> substract(2, 1)
          1
    """
    return a - b

if __name__ == '__main__':
    # Lancement des tests décrits dans les docstrings
    import doctest
    doctest.testmod()
```

Nous indiquons les tests à exécuter ainsi que le résultat attendu dans la docstring grâce aux triples chevrons représentant l'appel de l'interpréteur Python et le saut de ligne pour indiquer l'affichage console attendu lors du test.

Remarque

Lors de l'exécution des tests unitaires avec DocTest, l'interpréteur Python compare, non pas la valeur, mais ce qui est affiché, ce qui peut être déroutant.

Pour lancer les tests unitaires avec DocTest, nous devons simplement exécuter le script avec l'interpréteur Python.

```
**********************************************************************
File "doctest-exemple.py", line 9, in __main__.divide
Failed example:
    divide(2,1)
Expected:
    1
Got:
    2.0
**********************************************************************
1 items had failures:
   1 of   2 in __main__.divide
***Test Failed*** 1 failures.
```

Comme avec la librairie Unittest, le test rejeté n'est pas bloquant, tous les tests sont bien exécutés. Nous remarquons également que, pour un test qui a échoué, DocTest affiche l'instruction, le résultat attendu et le résultat obtenu.

3.4 Librairie Testify

La librairie Testify est le successeur de la librairie Unittest. Les tests sont écrits dans les méthodes d'une classe héritant de `TestCase`. L'avantage majeur de Testify est de pouvoir lancer des méthodes automatiques selon le cycle de vie de la classe, par exemple avant ou après tous les tests.

Pour installer Testify, nous devons utiliser la ligne de commande pip :

```
pip install testify
```

Pour lancer les tests, nous utilisons l'instruction `run()`.

```
from testify import *
import calculette_fonctions

# Déclaration de la classe contenant tous les tests
class Test(TestCase):

    # une méthode par test
    # => un seul assert par méthode
    def testDivide(self):
           assert_equal(calculette_fonctions
.divide(5, 5), 1)

    def testRejectedSubstract(self):
          assert_equal(calculette_fonctions
.substract(5, 5), 10)

    def testSubstract(self):
          assert_equal(calculette_fonctions
.substract(5, 5), 0)

if __name__ == '__main__':
    # Lancement de tous les tests de la classe Test
    run()
```