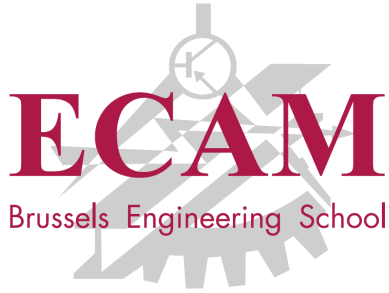


PI2T Développement informatique

Séance 1

Gestion de projet, versioning, debugging, testing et profiling

Sébastien Combéfis, Quentin Lurkin

1 février 2016

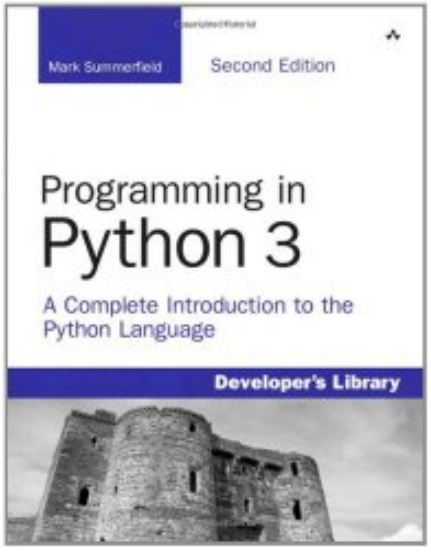
Informations générales (1)

- PI2T Développement informatique
  - Approfondissement du Python
  - 10 cours de 1h30 (15 heures)
- PI2L Projets de développement informatique
  - Développement d'une intelligence artificielle pour un jeu
  - 8 labos de 3h30 (28 heures)
- PN2L Application de méthodes numériques
  - Utilisation de libariries de calculs numériques
  - 4 labos de 3h30 (14 heures)

Informations générales (2)

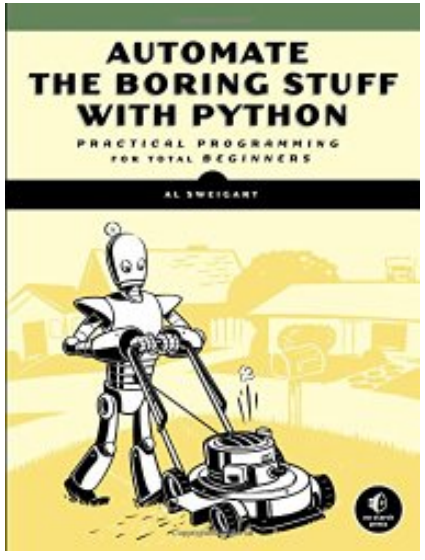
- Documents utilisés sont sur Eole (slides et énoncés des labos)
- Évaluation
  - Examen écrit (PI2T) : 30%
  - Labo (PI2L) : 45% (projets)
  - Labo (PN2L) : 25% (évaluation continue et test final)
- Enseignants
  - Sébastien Combéfis (s.combefis@ecam.be)
  - Francis Gueuning (f.gueuning@ecam.be)
  - André Lorge (a.lorge@ecam.be)
  - Quentin Lurkin (q.lurkin@ecam.be)

Livres de référence



ISBN

978-0-321-68056-3



ISBN

978-1-593-27599-0

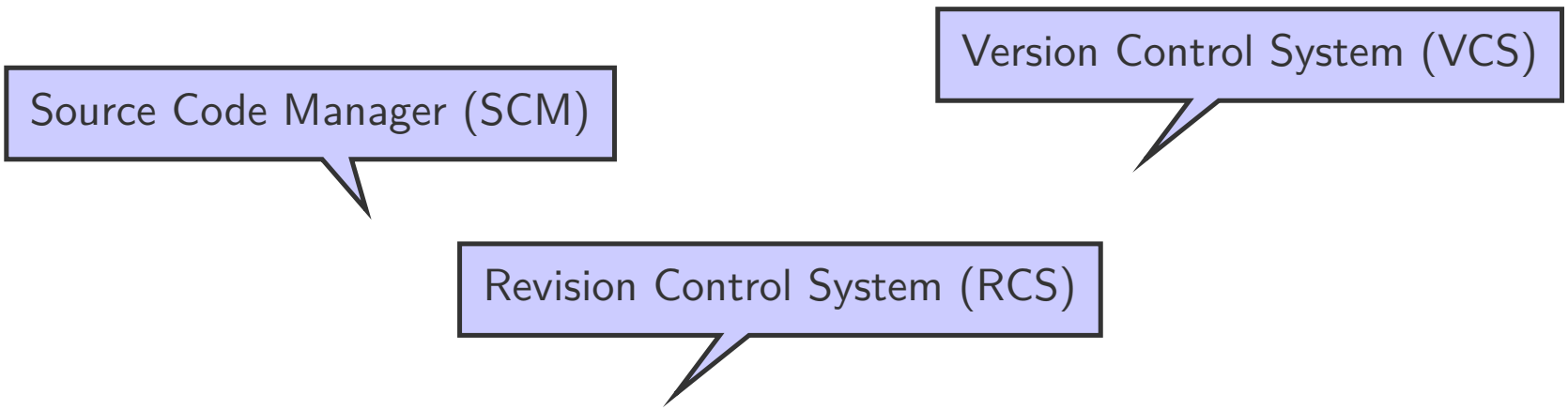
Objectifs

- Gestion du code source et déploiement
  - Versioning de code et exemple avec GitHub
  - Déploiement de code et exemple avec Heroku
- Test et analyse du code source
  - Debugging et comprendre les erreurs
  - Profiling et test de performance
  - Unit testing et exemple avec Travis

Versioning de code

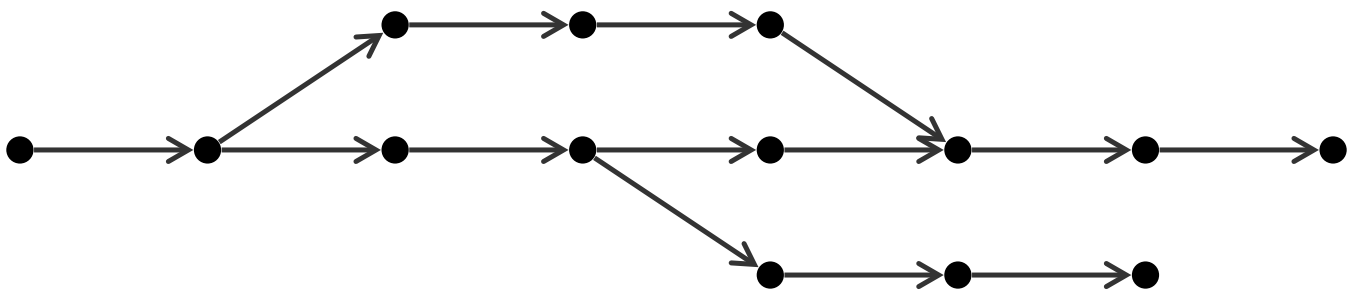
Source Code Management

- Pour tout projet informatique, il faut une stratégie de backup
- On ajoute souvent une gestion des versions
- Un développeur peut proposer plusieurs révisions par jour



Buts d'un gestionnaire de versions

- Gestion d'un projet de programmation
- Garder l'historique de toutes les modifications
- Travail en équipe
- Support de branches de développement



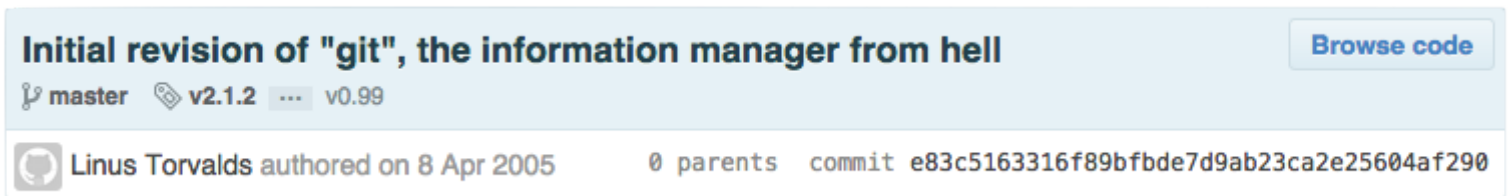
Git

- Système inventé par Linus Torvalds pour le kernel Linux
- Git a vu le jour en avril 2005

  Premier commit le 8 avril

- Logiciel de gestion de versions décentralisé

  Connexion internet uniquement pour les `pull` *et* `push`



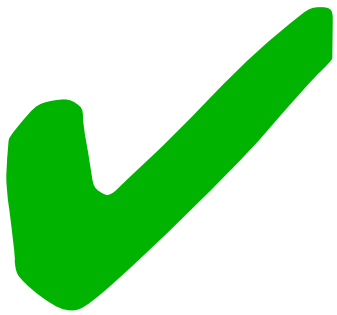
Prononciation

[gít]

[jít]

Prononciation

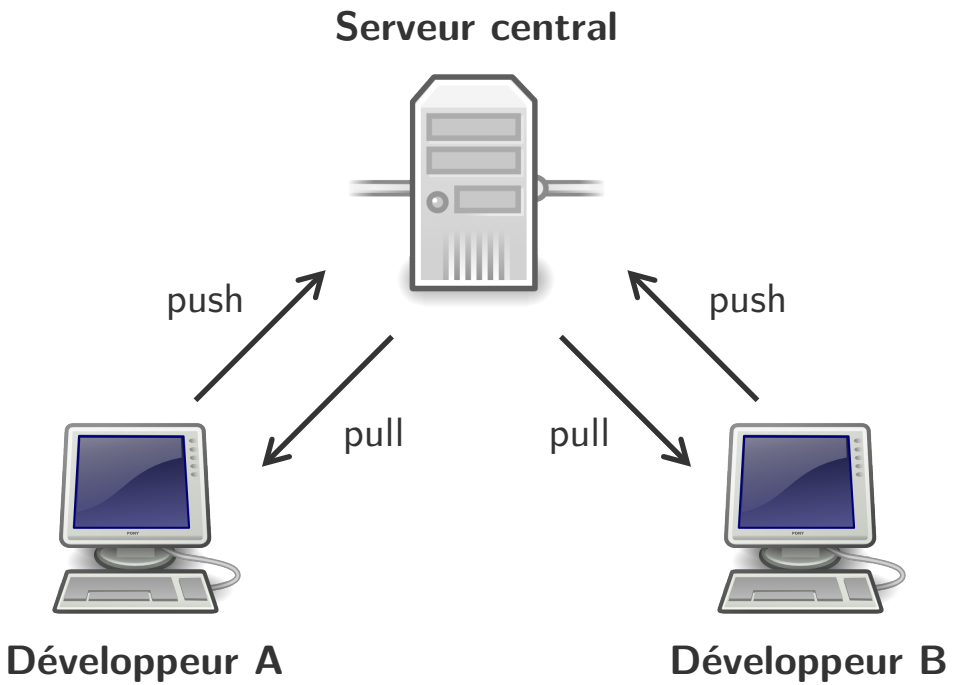
[gít]



[jít]

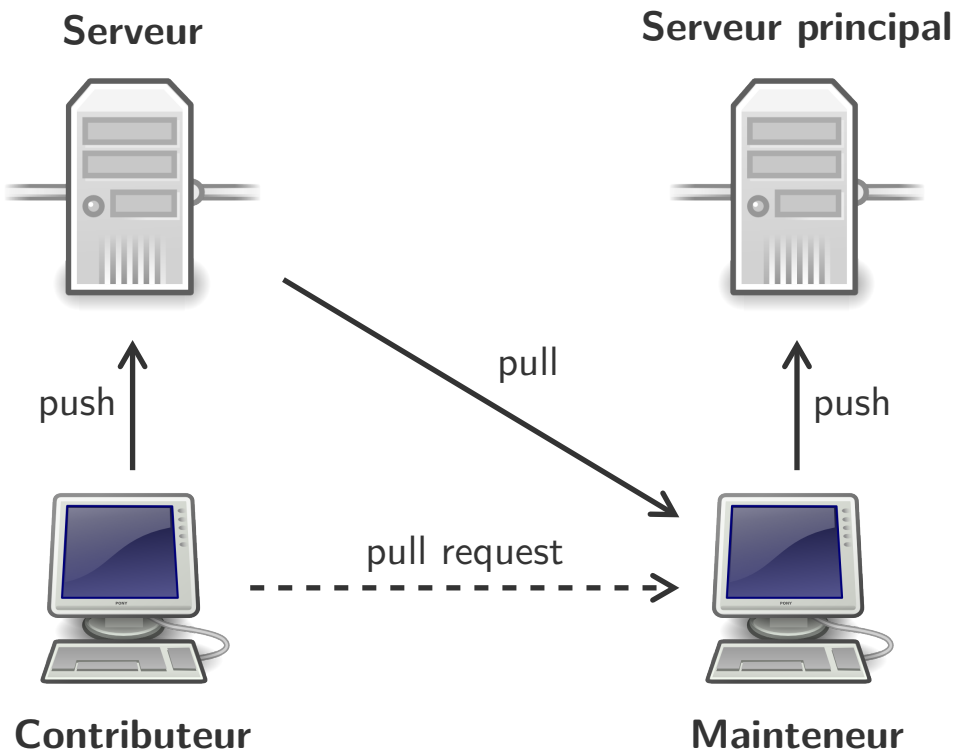
Git avec un serveur central

- Accès en écriture pour tous les développeurs



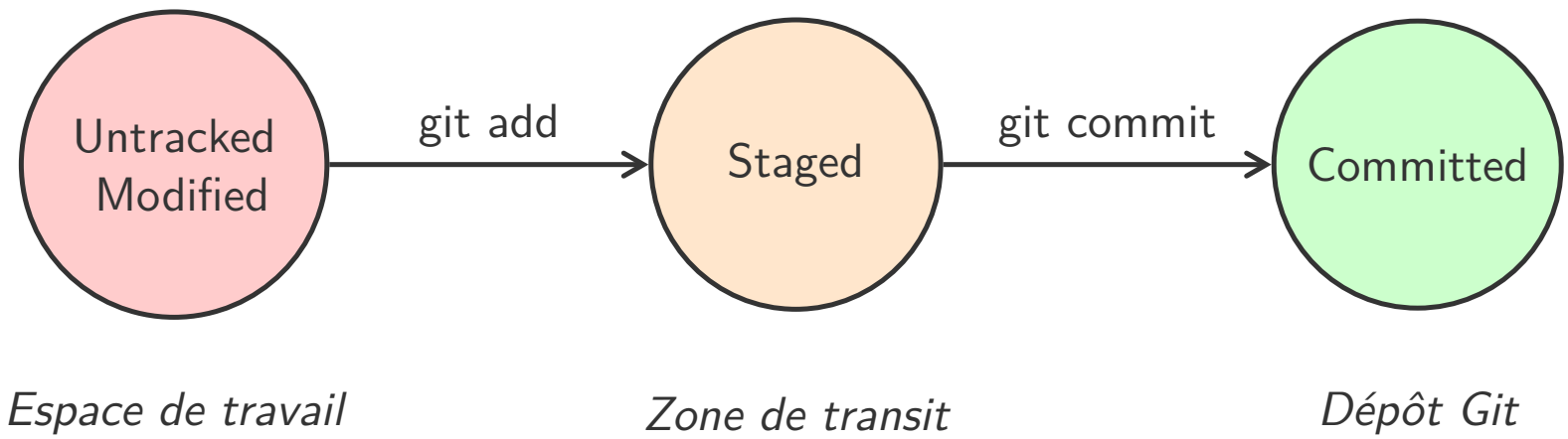
Git décentralisé

- Accès en écriture seulement pour les mainteneurs
- Les contributeurs font des *pull requests*



États des fichiers (1)

- Un fichier doit être explicitement ajouté au dépôt Git



États des fichiers (2)

- **Untracked/Modified**
  - Nouveaux fichiers ou fichiers modifiés
  - Pas pris en compte pour le prochain commit
- **Staged**
  - Fichiers ajoutés, modifiés, supprimés ou déplacés
  - Pris en compte pour le prochain commit
- **Unmodified/Committed**
  - Aucune modification pour le prochain commit

Commandes de base

- Ajouter un fichier dans la zone de transit

  git ***add*** *<fichier>*

- Obtenir l'état des fichiers

  git ***status***

- Valider les modifications en créant un commit

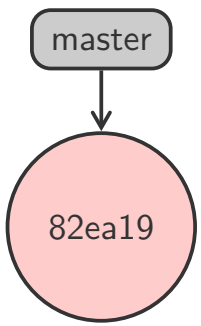
  git ***commit*** *-m "Titre du commit"*

- Obtenir l'historique des commits

  git ***log***

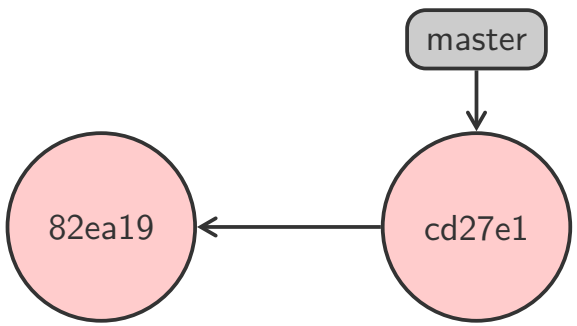
Le concept de branche

- Une branche pointe vers un commit
- À chaque nouveau commit, le pointeur de branche avance
- Un commit pointe vers le commit parent



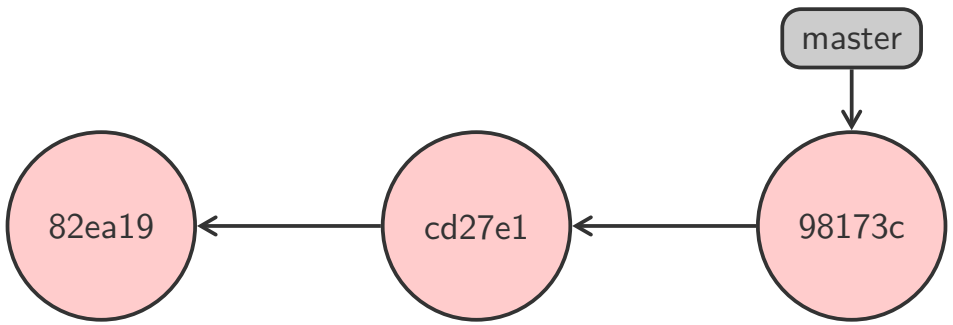
Le concept de branche

- Une branche pointe vers un commit
- À chaque nouveau commit, le pointeur de branche avance
- Un commit pointe vers le commit parent



Le concept de branche

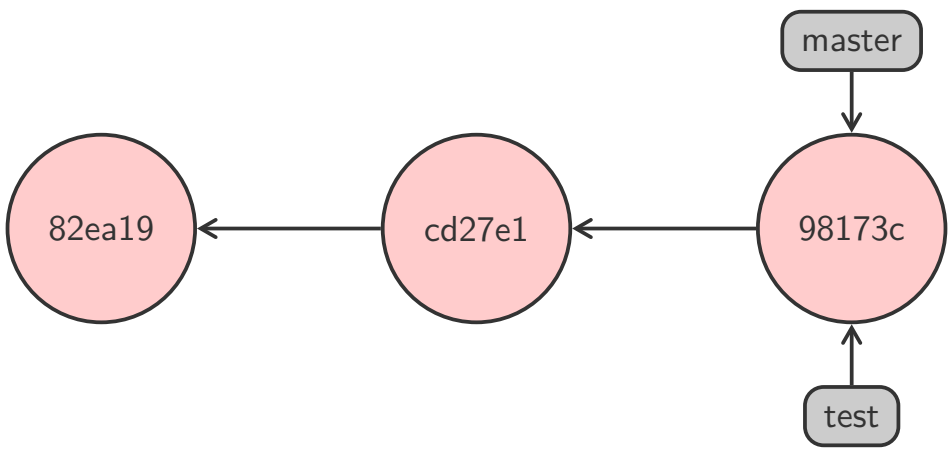
- Une branche pointe vers un commit
- À chaque nouveau commit, le pointeur de branche avance
- Un commit pointe vers le commit parent



Création d'une nouvelle branche

- Une nouvelle branche est créée avec « ***git*** *branch* <*name*> »

```
1 $ git branch test
```

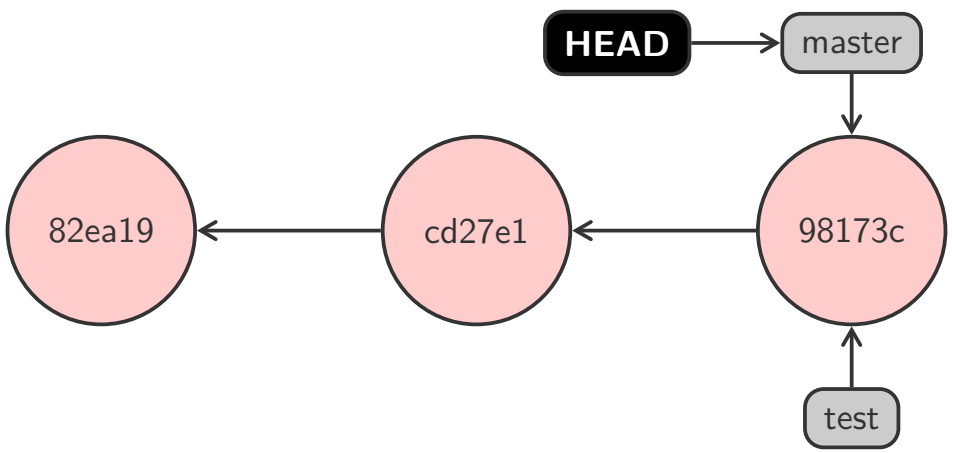


Branche courante

- La commande « ***git*** *branch* » liste les branches existantes

```
$ git branch
* master
  test
```

- La branche courante est identifiée par **HEAD**

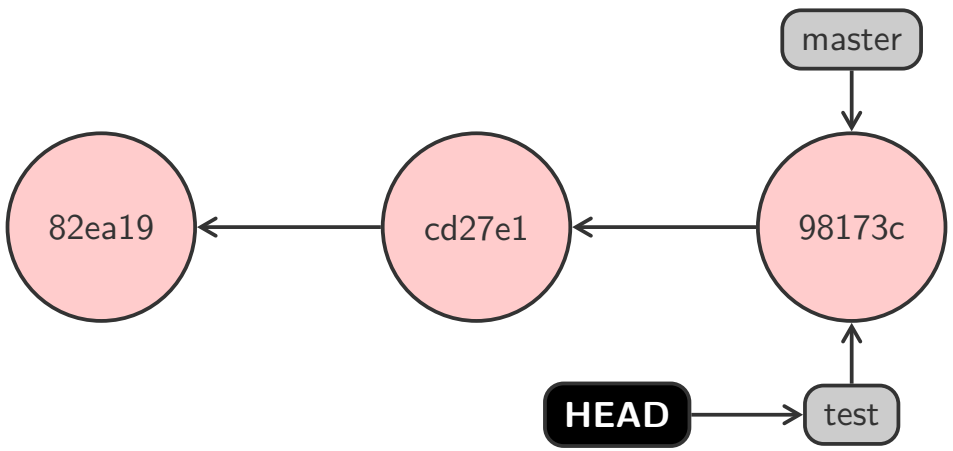


Changer de branche

- La commande « ***git*** *checkout <name>* » change de branche

```
1 $ git checkout test
2 Switched to branch 'test'
```

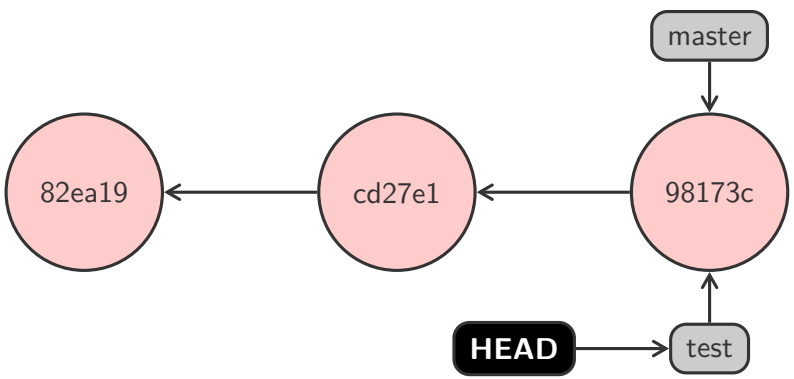
- La branche courante est identifiée par **HEAD**



Commit sur une branche

- Un commit va toujours se faire sur la branche courante

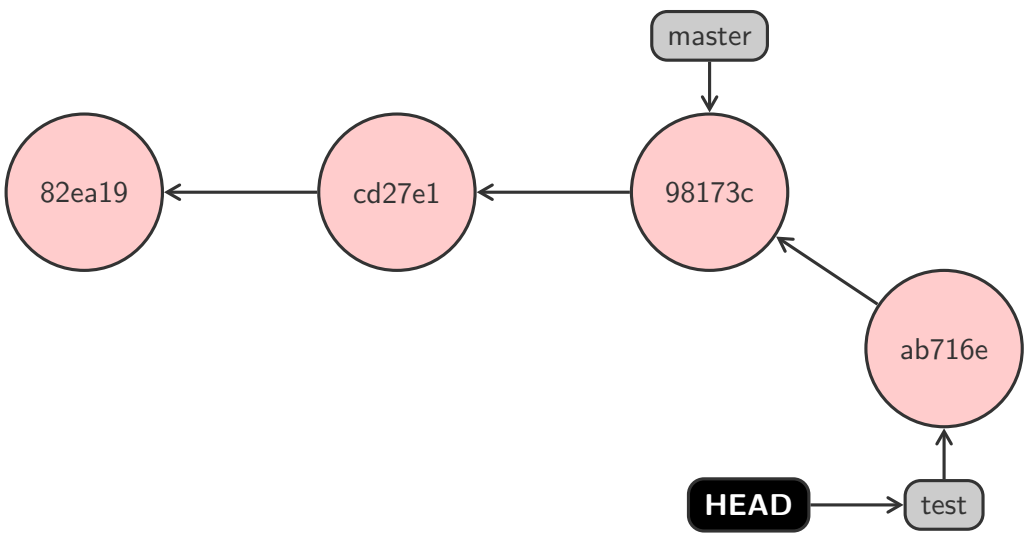
```
1  ...
2  $ git commit ...
3  $ git checkout master
4  ...
5  $ git commit ...
```



Commit sur une branche

- Un commit va toujours se faire sur la branche courante

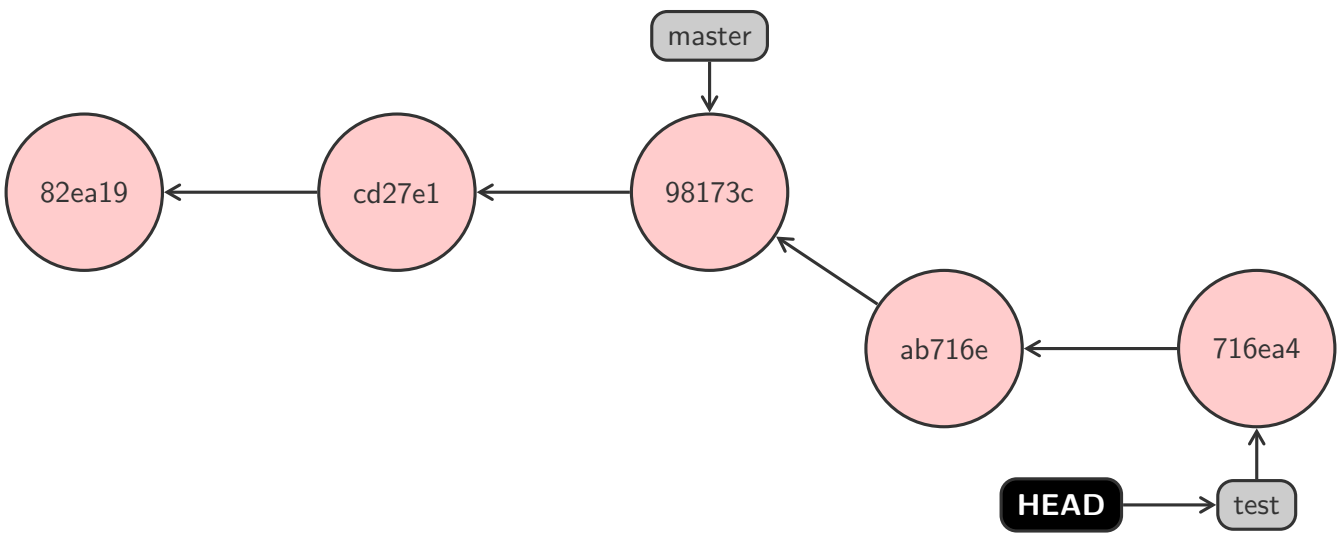
```
1  ...
2  $ git commit ...
3  $ git checkout master
4  ...
5  $ git commit ...
```



Commit sur une branche

- Un commit va toujours se faire sur la branche courante

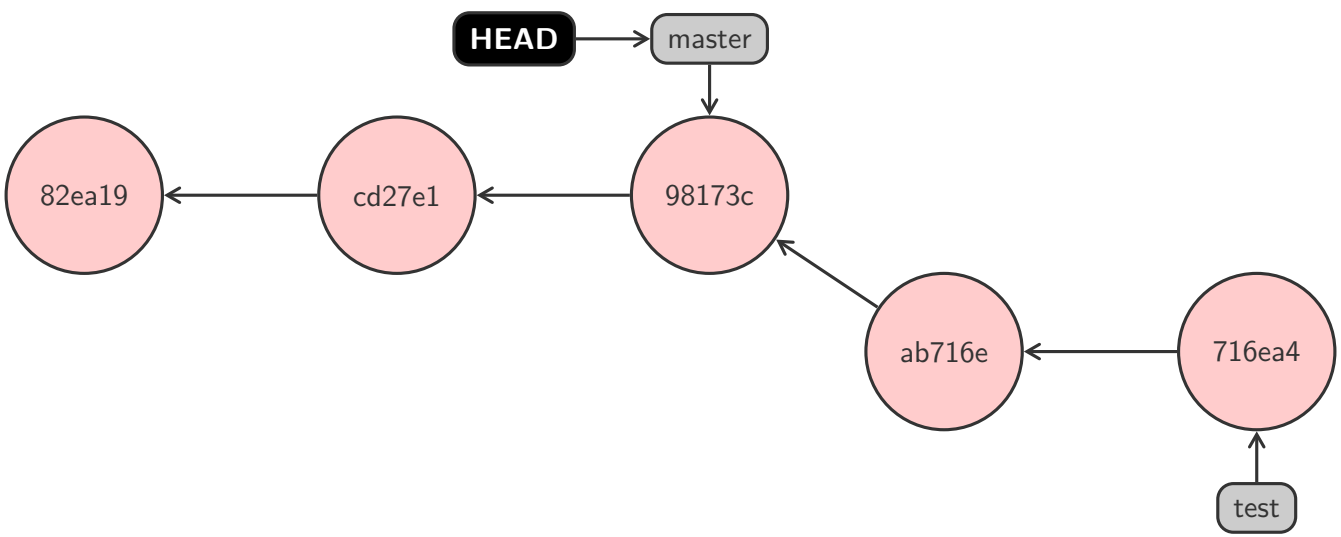
```
1  ...
2  $ git commit ...
3  $ git checkout master
4  ...
5  $ git commit ...
```



Commit sur une branche

- Un commit va toujours se faire sur la branche courante

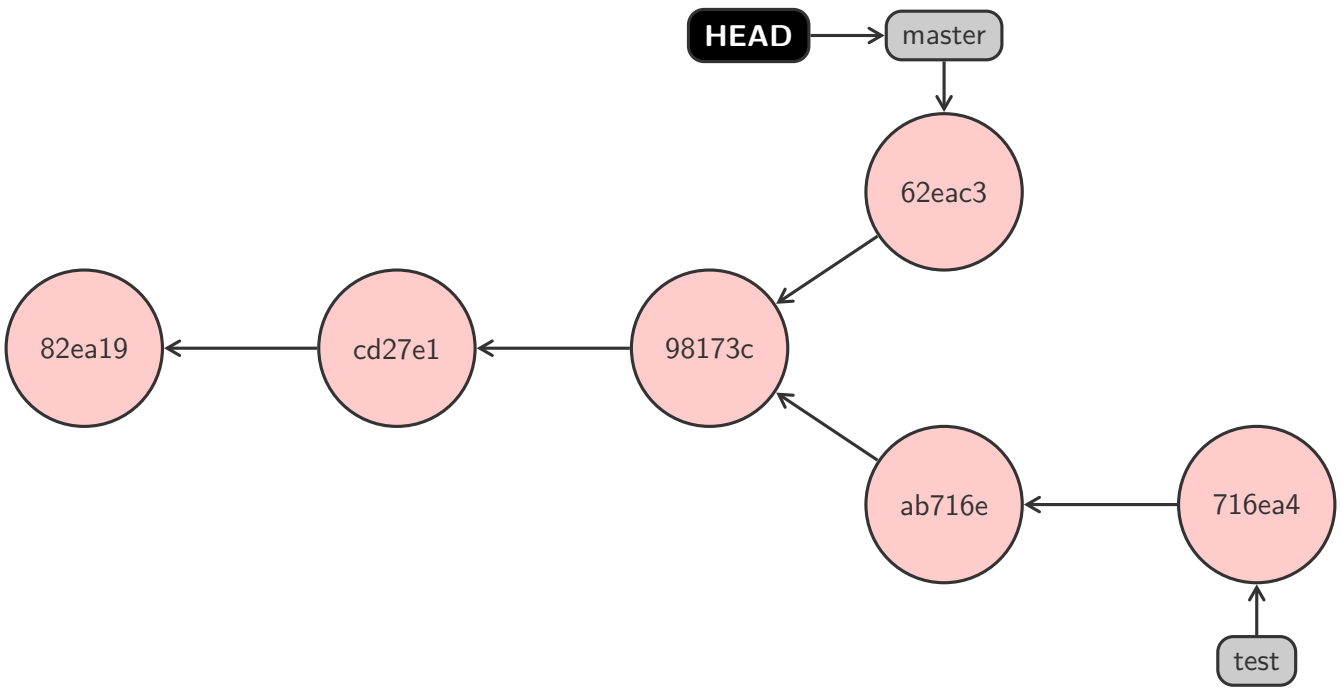
```
1  ...
2  $ git commit ...
3  $ git checkout master
4  ...
5  $ git commit ...
```



Commit sur une branche

- Un commit va toujours se faire sur la branche courante

```
1  ...
2  $ git commit ...
3  $ git checkout master
4  ...
5  $ git commit ...
```



Opérations de base sur une branche

- On peut supprimer une branche avec l'option `-d`

```
$ git branch -d test
Deleted branch test (was 617a041).
```

- On peut renommer une branche avec l'option `-m`

```
$ git branch
* master
  test
$ git branch -m test alternative
$ git branch
alternative
* master
```

Plateforme GitHub

- Plateforme d'hébergement de dépôts Git

  Serveur public permettant le partage de code

- Création gratuite d'un compte pour dépôts publics

  https://github.com/

Déploiement de code

Déploiement

- Installation, configuration et déploiement automatisé

  Sur base d'un dépôt Git

- Configuration de l'environnement d'exécution

- Définition du script de lancement

Plateforme Heroku

- Plateforme d'hébergement d'applications

  Déploiement d'applications web et serveur en ligne

- Création gratuite de dynos pour petites applications

  https://www.heroku.com

Debugging

Debugging

- Identification et correction de bugs
  - Erreur de syntaxe
  - Erreur d'exécution
- Importance de faire des backups réguliers
  - Backup des versions fonctionnelles avec versioning
  - Pouvoir revenir à une version fonctionnelle
  - Identifier le code qui a introduit le bug

Erreur de syntaxe

- Erreur de syntaxe décrite par trois éléments
  - Nom du fichier
  - Numéro de la ligne
  - La ligne contenant l'erreur avec un caret ^

```
1 def compute(n)
2     result = n
3     for i in range(n):
4         result += i
5     return result
```

```
$ python3 program.py
  File "program.py", line 1
    def compute(n)
                 ^
SyntaxError: invalid syntax
```

Erreur d'exécution (1)

- Arrêt immédiat de l'exécution en cas d'erreur

Affichage de la trace d'exécution montrant les erreurs non gérées

```
1  def mean(data):
2      total = 0
3      for elem in data:
4          total += elem
5      return total / len(data)
6
7  print(mean([1, 2, 3]))
8  print(mean([]))
```

```
$ python3 program.py
2.0
Traceback (most recent call last):
  File "program.py", line 8, in <module>
    print(mean([]))
  File "program.py", line 5, in mean
    return total / len(data)
ZeroDivisionError: division by zero
```

Erreur d'exécution (2)

- La trace d'exécution montre où l'erreur est apparue

  Et pas où elle s'est produite

- L'erreur peut trouver son origine dans la librairie standard

  Ou dans toute autre librairie utilisée

- Bon réflexe d'examiner les dernières lignes de la trace

  Ce sont les lignes relatives au code de l'utilisateur

Chasser et tuer un bug

- Méthode scientifique de debugging
  1. Reproduire le bug (*Reproduce*)
  2. Localiser le bug (*Locate*)
  3. Corriger le bug (*Fix*)
  4. Tester le correctif (*Test*)
- Utilisation des tests unitaires pour faciliter le processus

  Pour s'assurer que le correctif n'a pas introduit de nouveau bug

Debugguer

- Utilisation de la fonction `print` pour debugguer

  Pas pratique car pollue le code du programme

- Utilisation du module `pdb`
  - Exécution pas à pas du programme
  - Inspection de la mémoire

Module pdb

```
1  import pdb
2  pdb.set_trace()
3
4  a = 0
5  print(25 / a)
```

```
python3 program.py
> /Users/combefis/Desktop/program.py(4)<module>()
-> a = 0
(Pdb) s
> /Users/combefis/Desktop/program.py(5)<module>()
-> print(25 / a)
(Pdb)
ZeroDivisionError: division by zero
> /Users/combefis/Desktop/program.py(5)<module>()
-> print(25 / a)
(Pdb) p a
0
(Pdb) c
Traceback (most recent call last):
  File "program.py", line 5, in <module>
    print(25 / a)
ZeroDivisionError: division by zero
```

Profiling

Profiling

- Vérification de la consommation de ressources
  - Programme trop lent avec un goulot d'étranglement
  - Consommation excessive de mémoire avec fuites
- Plusieurs causes possibles
  - Choix d'un algorithme inapproprié
  - Choix d'une structure de données inadaptées

"Premature optimization is the root of all evil." — C.A.R. Hoare

Quelques tuyaux Python (1)

1. Préférez les tuples aux listes

   Lorsqu'il vous faut une séquence à utiliser en lecture seule

2. Utilisez des générateurs plutôt que de grosses séquences

   Permet une lazy evaluation des éléments de la séquence

3. Utilisez les structures de données prédéfinies de Python

   Dictionnaires (`dict`), listes (`list`), tuples (`tuples`)

Quelques tuyaux Python (2)

4. Créez les longues chaine de caractères avec `join`

   Accumuler les chaines dans une liste plutôt que de concaténer

5. Stockez une référence vers un objet souvent utilisé

   Pour une fonction depuis un module, une méthode sur un objet...

Mesure de performances

- Mesure de temps avec le module `timeit`

  Mesurer le temps d'exécution de petites portions de code

- Établissement du profil d'un programme avec `profile`

  Profiler les performances d'un programme

Module `timeit`

- Création d'un objet `timeit.Timer`

  Avec le code à mesurer et le code à exécuter avant

- Lancer l'exécution et la mesure avec la méthode `timeit`

  Avec le nombre de répétitions à faire en paramètre

```
1  import timeit
2
3  def compute(n):
4      result = n
5      for i in range(n):
6          result += i
7      return result
8
9  repeats = 1000
10 t = timeit.Timer("compute(2000)", "from __main__ import compute")
11 sec = t.timeit(repeats) / repeats
12 print('{} secondes'.format(sec))
```

`timeit` en ligne de commande

- Mesure du temps d'exécution en ligne de commande

  Pour éviter de devoir instrumenter son code

- Mesure le temps d'exécution de l'exécution d'un code

  Code à exécuter spécifié en paramètre

```
$ python3 -m timeit -n 1000 -s "from program import compute"
"compute(2000)"

1000 loops, best of 3: 170 usec per loop
```

Module `profile`

- Appel de la fonction `run` du module `profile`

  Il faut mettre la boucle directement dans l'appel

- Affichage du profil complet des appels

```
1  import profile
2
3  def compute(n):
4      result = n
5      for i in range(n):
6          result += i
7      return result
8
9  profile.run("for i in range(1000): compute(2000)")
```

`profile` en ligne de commande

- Établissement du profil de performance d'un programme

  Exécute un programme et mesure le temps passé dans les appels

```
$ python3 -m profile program.py
         1261 function calls (1258 primitive calls) in 0.155 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
[...]
        1    0.000    0.000    0.000    0.000 profile.py:104(Profile)
        1    0.000    0.000    0.000    0.000 profile.py:350(fake_code)
        1    0.000    0.000    0.000    0.000 profile.py:360(fake_frame)
        1    0.000    0.000    0.000    0.000 profile.py:43(_Utils)
        1    0.000    0.000    0.000    0.000 profile.py:9(<module>)
        1    0.000    0.000    0.155    0.155 profile:0(<code object <
        module> at 0x10a791c00, file "testE.py", line 1>)
        0    0.000             0.000          profile:0(profiler)
        1    0.002    0.002    0.155    0.155 program.py:1(<module>)
     1000    0.151    0.000    0.151    0.000 program.py:3(compute)
```


Test unitaire

Testing

- Nécessité de tester qu'un programme fait bien ce qu'il faut
  - Définir ce que le programme doit faire
  - Écrire un jeu de tests pour vérifier le programme
- Impossible de garantir l'exactitude d'un programme

  On ne peut pas tester tous les scénarios possibles
- Amélioration de la qualité de code

  Un jeu de tests bien choisi diminue le nombre de bugs potentiels

Types de test

- **Test utilisateur** (*usability testing*)

  Évaluer un programme par des tests utilisateurs (ergonomie...)

- **Test fonctionnel** (*functional* testing)

  Assurance qualité (QA) et test black-box sur les spécifications

- **Test d'intégration** (*integration* testing)

  Vérification des performances et de la fiabilité du programme

Test unitaire

- Test individuel d'une unité dans le code

  Une fonction, une classe ou une méthode

- Définition du test sur base d'une spécification du code

  Étant donné les préconditions, vérifier les postconditions

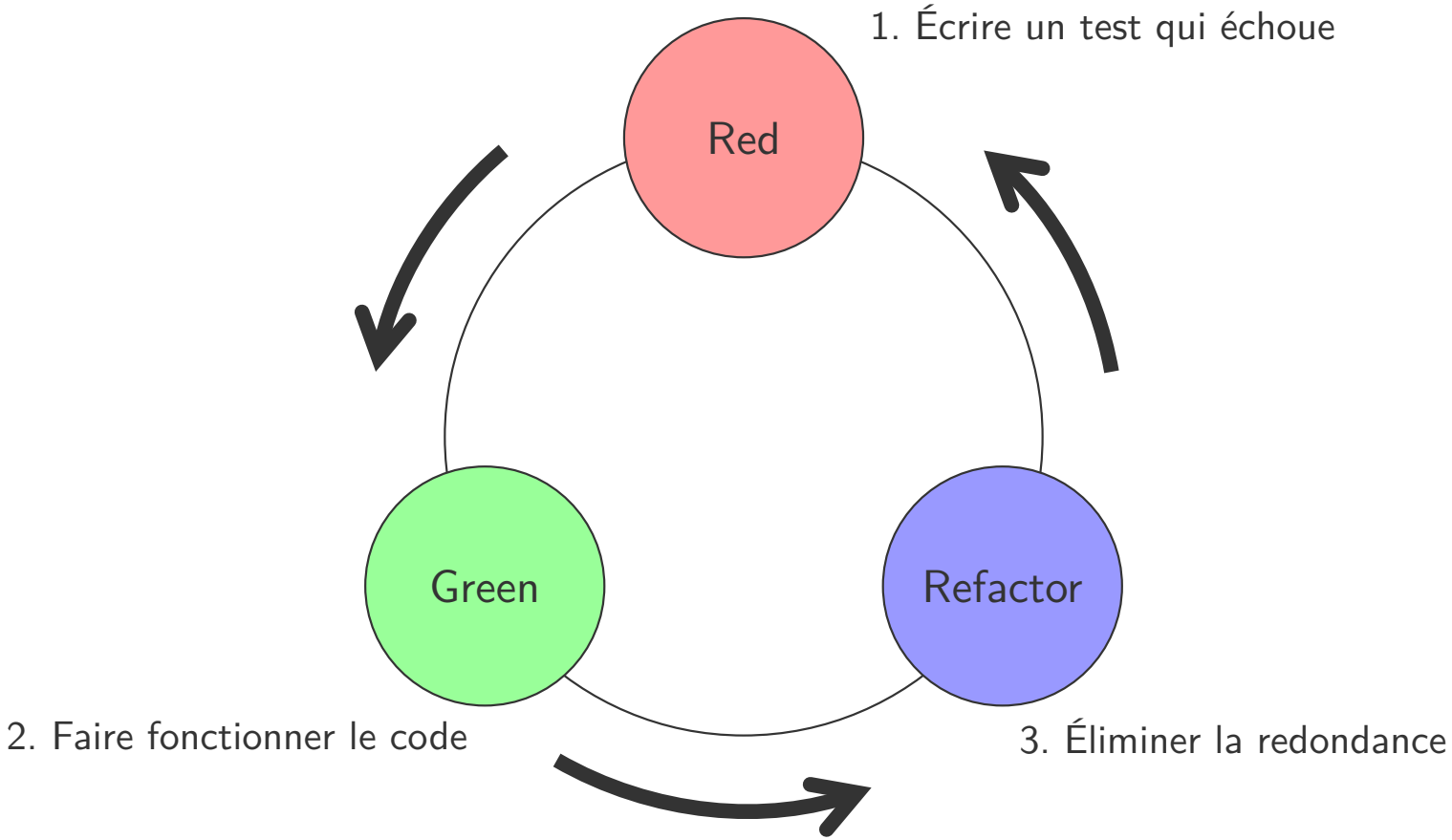
- Utilisé notamment en Test-Driven Development (TDD)

  Technique de développement de logiciel piloté par les tests

Cycle TDD

- Cycle en trois phases principales

  Red–Green–Refactor



Module doctest

- Permet de tester le programme à partir de sa documentation

  Test et solution attendue placé dans un docstring

- Tests écrits comme un appel dans l'interpréteur interactif

```
1  def compute(n):
2      """
3      >>> compute(0)
4      0
5      >>> compute(3)
6      6
7      """
8      result = n
9      for i in range(n):
10         result += i
11     return result
```

doctest en ligne de commande

- Exécution des tests avec résultats détaillés

  Bilan global des tests exécutés à la fin de la sortie de l'exécution

```
$ python3 -m doctest -v program.py
Trying:
    compute(0)
Expecting:
    0
ok
Trying:
    compute(3)
Expecting:
    6
ok
1 items had no tests:
    program
1 items passed all tests:
   2 tests in program.compute
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Exécution des `doctest` avec `unittest` (1)

- Exécuter les doctest directement en construisant un `unittest`
- Plusieurs étapes à suivre
  1. Création d'une suite de tests (`TestSuite`)
  2. Ajout d'un test à la suite, de type `doctest` (`DocTestSuite`)
  3. Création d'un exécuteur textuel (`TextTestRunner`)
  4. Exécution des tests (`run`)

Exécution des doctest avec unittest (2)

- Utilisation des modules doctest et unittest

  Programme à placer dans un fichier séparé

```
1  import doctest
2  import unittest
3  import program
4
5  suite = unittest.TestSuite()
6  suite.addTest(doctest.DocTestSuite(program))
7  runner = unittest.TextTestRunner()
8  print(runner.run(suite))
```

```
$ python3 test_testE.py
.
----------------------------------------------------------------------

Ran 1 test in 0.001s

OK
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

Module `unittest`

- Séparation claire entre les tests et le code à tester

  Utile lorsque code et tests pas rédigés par les mêmes personnes

- Quatre concepts clés

  1. **Test fixture** *pour initialiser et nettoyer un test*
  2. **Test suite** *est un ensemble de test cases*
  3. **Test case** *est l'unité de base des tests*
  4. **Test runner** *permet d'exécuter des suites de tests*

Classe de test (1)

- Une suite de tests se définit à partir d'une classe de test

  Classe « spéciale » construite sur base de `unittest.TestCase`

- Utilisation de méthodes prédéfinies pour exprimer les tests

  Expression de la valeur attendue d'exécution de code

Méthode	**Description**	**Test**
`assertTrue(x)`	Affirme que `x` est vrai	`bool(x) is True`
`assertEqual(a, b)`	Affirme que `a` et `b` sont égaux	`a == b`
`assertIs(a, b)`	Affirme que `a` et `b` sont identiques	`a is b`
`assertIsNone(x)`	Affirme que `x` est `None`	`x is None`
`assertIn(a, b)`	Affirme que `a` se trouve dans `b`	`a in b`
`assertIsInstance(a, b)`	Affirme que `a` est une instance de `b`	`isinstance(a, b)`

Et aussi `assertFalse`, `assertNotEqual`, `assertIsNot`, `assertIsNotNone`, `assertNotIn` et `assertNotIsInstance`...

Classe de test (2)

```python
import unittest
import program

class Test(unittest.TestCase):
    def test_compute(self):
        self.assertEqual(program.compute(0), 0)
        self.assertEqual(program.compute(-2), -1)

suite = unittest.TestLoader().loadTestsFromTestCase(Test)
runner = unittest.TextTestRunner()
print(runner.run(suite))
```

```
$ python3 test_program.py
F
======================================================================
FAIL: test_compute (__main__.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_program.py", line 7, in test_compute
    self.assertEqual(testE.compute(-2), -1)
AssertionError: -2 != -1

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
<unittest.runner.TextTestResult run=1 errors=0 failures=1>
```

Initialisation et nettoyage

- Initialisation avant et nettoyage après exécution de chaque test

  Via les méthodes `setUp` et `tearDown`

```
1  class Test(unittest.TestCase):
2      def setUp(self):
3          # Code exécuté avant chaque test
4
5      def tearDown(self):
6          # Code exécuté après chaque test
```

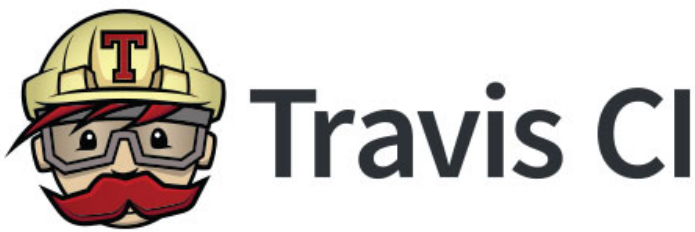
Plateforme Travis

- Plateforme d'exécution automatique de tests

  Code automatiquement rapatrié depuis GitHub par exemple

- Création gratuite d'un compte pour tester des dépôts publics

  https://travis-ci.org/



Configuration de Travis

- Création d'un fichier `.travis.yml` pour la configuration
  - Language de programmation
  - Version spécifique
  - Script à exécuter

```
1 language: python
2 python:
3   - "3.5"
4 script: python3 test.py
```

Crédits

- Photos des livres depuis Amazon
- https://www.flickr.com/photos/jwhitesmith/7363049912
- https://openclipart.org/detail/36565/tango-network-server-by-warszawianka
- https://openclipart.org/detail/34531/tango-computer-by-warszawianka
- https://www.flickr.com/photos/faisal_akram/8107449789
- https://www.flickr.com/photos/rachel_s/9243714784
- https://www.flickr.com/photos/110777427@N06/15632985383
- https://www.flickr.com/photos/nasamarshall/21064480196