Tests unitaires

Tests unitaires : Définitions

- Unité de code : ensemble d'instruction qui réalise une opération complète (typiquement une fonction)

```
# Code source
def ma_fonction(arg1, arg2):
    return arg1 + arg2
```

- Tests : appels de l'unité de code sur des jeux de données pour lesquels les « résultats » sont connus
  - ma_fonction(1, 1) retourne 2
  - ma_fonction(-1, 1) retourne 0
  - …

Tests unitaires : Objectifs

- Avoir des cas tests à la fois
  - Simples
  - Exhaustifs, i.e. qui couvrent différents type de configuration
  - Jouables automatiques
- Exemple

```
# Code de test
from codesource import ma_fonction

assert ma_fonction(1, 1)   == 2   # test canonique
assert ma_fonction(1, -1)  == 0   # test avec un chiffre négatif
assert ma_fonction(2, 3)   == 5   # test non canonique
assert ma_fonction(2.5, 3) == 5.5 # test avec des réels
```

Tests unitaires : Intérêts

- Réfléchir à la construction des briques de base
  - les Entrées/Sorties
  - Définition des API (Application Programming Interface)
  - Rigueur dans le développement
  - => les tests unitaires sont à penser en amont
- Prévenir la régression
  - Au cours du temps le code évolue
  - La réutilisation de certaines briques (factorisation) nécessite la modification de celles-ci
  - => les tests unitaires détectent rapidement les régressions

Régression : Exemple

- Evolution de `ma_fonction`

```
# Code source
def ma_fonction(arg1, arg2):
    return int(arg1) + int(arg2)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-7-b7361b809026> in <module>
      6 assert ma_fonction(1, -1)  == 0 # test avec un chiffre négatif
      7 assert ma_fonction(2, 3)   == 5 # test avec autre chose que des 1
----> 8 assert ma_fonction(2.5, 3) == 5.5 # test avec des réels

AssertionError:
```

unittest

- Module python de test unitaire
- Permet de définir des fonctions pour chaque test et de les jouer
- Joue tous les tests même si certains plantent

```
# Code de test
import unittest
from moncode import ma_fonction

class TestMaFonction(unittest.TestCase):
    def test_canonique(self):
        res = ma_fonction(1, 1)
        self.assertEqual(res, 2)

    def test_negatif(self):
        res = ma_fonction(-1, 1)
        self.assertEqual(res, 0)
```

```
# Code de test
...
# Ceci lance le test si on exécute le
script directement.
if __name__ == '__main__':
    unittest.main()
```

Git

Git Dépôt

Création d'un dépôt

- Créer un répertoire, dans ce répertoire

```
git init
```

Cloner (copier) un dépôt

- créer une copie d'un dépôt local

```
git clone /path/to/repository
```

- créer une copie d'un dépôt distant (sur un serveur)

```
git clone username@host:/path/to/repository
```

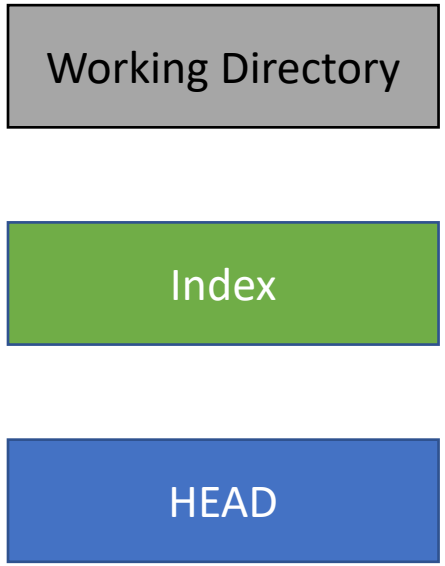
Git Arbres

dépôt local composé de trois "arbres" gérés par git :

1. **Espace de travail (Working Directory)** qui contient réellement les fichiers
2. **Index** qui joue un rôle d'espace de transit pour les fichiers
3. **HEAD** qui pointe vers la dernière validation (commit) faite.

Commande utile (état courant)

```
git status
```



Git : Ajouter *Commiter*

- proposer un changement (l'ajouter à l'**Index**)

```
git add <filename>
git add *
```

- 1ère étape dans un workflow git basique.
- Pour valider (*commiter*) ces changements (l'ajout à HEAD)

```
git commit -m "Message de validation"
```

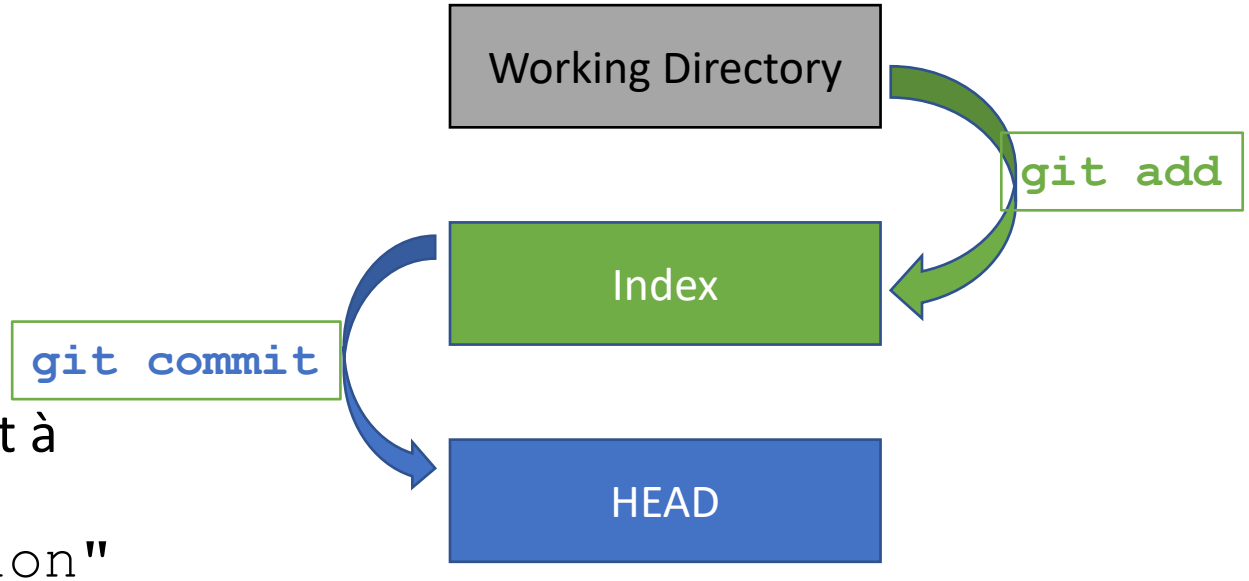
  - Le fichier est ajouté au **HEAD** pas au dépôt distant
  - Chaque commit est identifié par un numéro SHA

- Commande utile (historique)

```
git log
```

- Tagger une version

```
Git tag <nom_version> <SHA1>
```



Git branch

- Les branches sont utilisées pour développer des fonctionnalités isolées des autres.
- La branche *master* est la branche par défaut (création du dépôt).
- créer une nouvelle branche nommée *feature* (et s'y positionner)

```
git checkout -b feature
```

- retourner sur une branche

```
git checkout <branch>
```

- supprimer la branche

```
git branch -d feature
```

- une branche n'est *pas disponible pour les autres* tant qu'elle n'est pas envoyée vers le dépôt distant

Git Dépôt distant

- Pour les envoyer le **HEAD** au dépôt distant

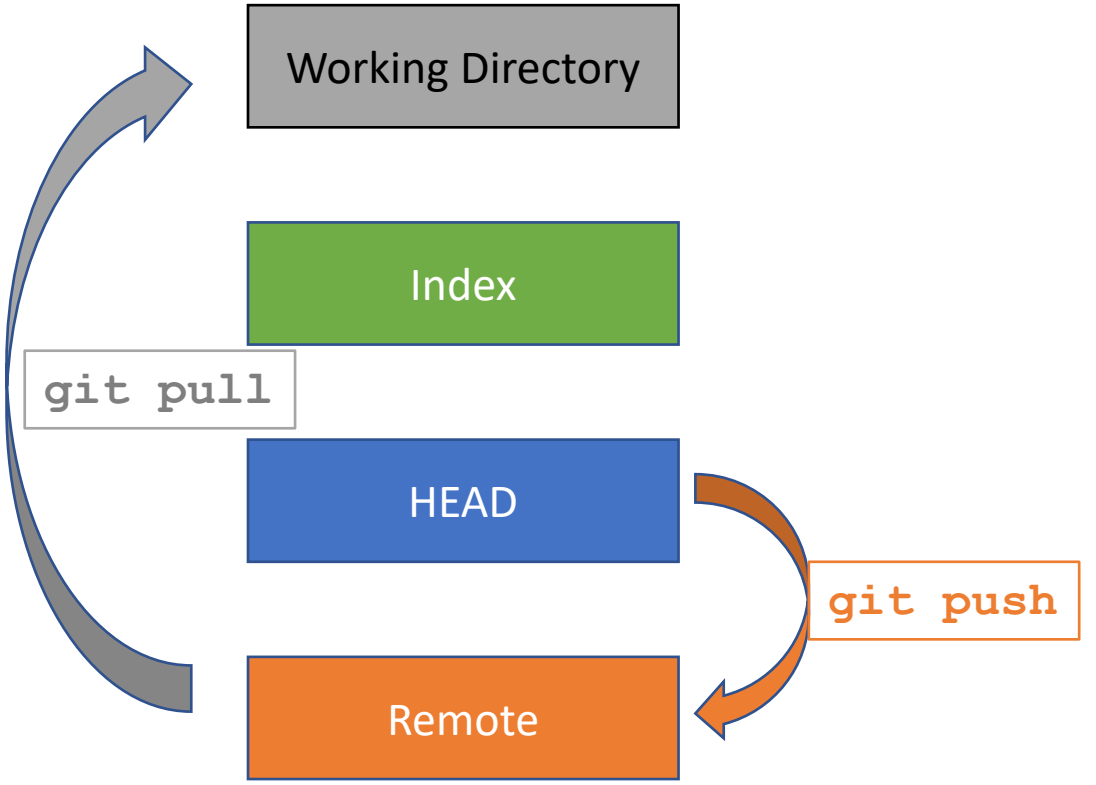
  `git push origin <branch>`

- Cloner dépôt local sur un serveur distant

  `git remote add origin <server>`

- Mettre à jour le dépôt local à partir du distant

  `git pull`



Git Fusion

- fusionner une autre branche avec la branche active

  `git merge <branch>`

- En cas de conflit : correction *a la mano* et ajout de la modification

  `git add`

- Aperçu des différences entre deux branches

  `git diff <source_branch> <target_branch>`

- Retour en arrière (sur modification locale)

  `git checkout -- <filename>`

  - Remplace le contenu du fichier `<filename>` par le contenu dans le HEAD