

# Cours 2

---

Version 2, proposition de travail

1re année géomatique  
2027-2028

# Les fichiers du cours

Une archive par séance, un dossier par TD. Aujourd'hui, tout est dans `cours2/`.

```
cours2/                                un dossier par TD, dans l'ordre de la séance
├── 2a_vscode/
├── 2b_erreurs/
├── 2c_markdown/
└── 2d_depot_recette/
```

Dans chaque dossier, la feuille du TD et son guide détaillé, en PDF et en page web.

# Contenu de la séance

L'éditeur de code est présenté, puis configuré par toute la salle en même temps. Le reste de la séance s'y déroule.

| Partie                        | Nature                   | Durée |
|-------------------------------|--------------------------|-------|
| L'éditeur de code et Markdown | cours                    | 10'   |
| Configurer l'éditeur de code  | TD 2a, en classe entière | 25'   |
| Corriger trois programmes     | TD 2b                    | 10'   |
| Une recette en Markdown       | TD 2c                    | 12'   |
| Git local                     | cours et TD 2d           | 62'   |

Durées indicatives. Tout se fait dans VS Code, avec son terminal Git Bash.

---

# L'éditeur de code

Ce qu'un éditeur de code ajoute à un éditeur de texte.

# Les fonctions d'un IDE

IDE, pour *integrated development environment* : un logiciel qui réunit des fonctions pour aider à écrire, tester et partager du code.

| La fonction             | Ce que l'éditeur en fournit                                           |
|-------------------------|-----------------------------------------------------------------------|
| Écrire le code          | coloration, indentation, complétion, soulignement des fautes          |
| Le lancer et le tester  | un terminal intégré et un bouton d'exécution, sans quitter la fenêtre |
| Naviguer dans le projet | l'arborescence à gauche, la recherche dans tous les fichiers          |
| Déboguer                | exécuter pas à pas, arrêter sur une ligne, lire les variables         |
| Connaître le langage    | certain IDE n'en servent qu'un ; d'autres s'étendent par extensions   |

**Attention.** Un IDE ne contient pas forcément l'interpréteur ni le compilateur. Ils s'installent à part, et se configurent pour chaque langage et chaque système.

# Texte brut et règles du langage

Un langage a une syntaxe définie. L'éditeur la connaît, par une extension, et colore chaque catégorie de mot sans rien ajouter au fichier.

Enregistré par un traitement de texte

```
<text:p text:style-name="P1">
altitude = 128.4</text:p>
<text:p>print(altitude)</text:p>
```

Le fichier d'un programme

```
altitude = 128.4
print(altitude)
```

Affiché par l'éditeur de code

```
altitude = 128.4
print(altitude)
```

Il souligne de même ce qui ne suit pas ces règles, sans rien lancer.

```
altitudes = [128.4, 131.0]
for altitude in altitudes
    print(altitude)
```

Le deux-points manque. Sans l'éditeur, la faute n'apparaîtrait qu'au lancement : `SyntaxError: expected ':'`.

# Chasse fixe et chasse proportionnelle

Un éditeur de code emploie une police à chasse fixe, où toutes les lettres ont la même largeur. Le même programme dans la police d'un traitement de texte :

## Chasse fixe (éditeur de code)

```
def surface(longueur, largeur):  
    aire = longueur * largeur  
    if aire > 250:  
        categorie = "grande"  
    else:  
        categorie = "petite"  
    return aire, categorie
```

## Chasse proportionnelle (traitement de texte)

```
def surface(longueur, largeur):  
    aire = longueur * largeur  
    if aire > 250:  
        categorie = "grande"  
    else:  
        categorie = "petite"  
    return aire, categorie
```

Même texte, même taille, deux polices. À droite, rien ne dit de combien chaque ligne est décalée.

# L'indentation, en espaces ou en tabulation

Un espace et une tabulation sont deux caractères différents. Une tabulation vaut le nombre de colonnes que l'éditeur lui donne, et ce réglage change d'un éditeur à l'autre.

Tabulation réglée sur 4 colonnes

```
def surface(longueur, largeur):  
    ····aire = longueur * largeur  
→    return aire
```

les deux lignes semblent alignées

· marque un espace, → une tabulation, comme l'éditeur les dessine. Les octets du fichier sont les mêmes des deux côtés : seul le réglage de l'éditeur change.

Le même fichier, tabulation sur 8

```
def surface(longueur, largeur):  
    ····aire = longueur * largeur  
→        return aire
```

le décalage apparaît

# Markdown, un format texte pour les documents

Un fichier Markdown est du texte brut, comme un programme : il s'écrit dans l'éditeur de code, et git en compare les lignes. Ses modifications se comprennent sans programmer.

Ce qu'on écrit, `recette.md`

## ## Ingrédients

```
| Ingrédient | Quantité |  
| --- | --- |  
| Farine | 250 g |  
| Œufs | 4 |
```

## ## Préparation

1. Mélanger la farine et le sel.
2. Casser les œufs.

La recette du TD 2c sert ensuite de projet pour git. Le `README.md` d'un projet s'écrit de la même façon.

Ce que l'aperçu montre

## Ingrédients

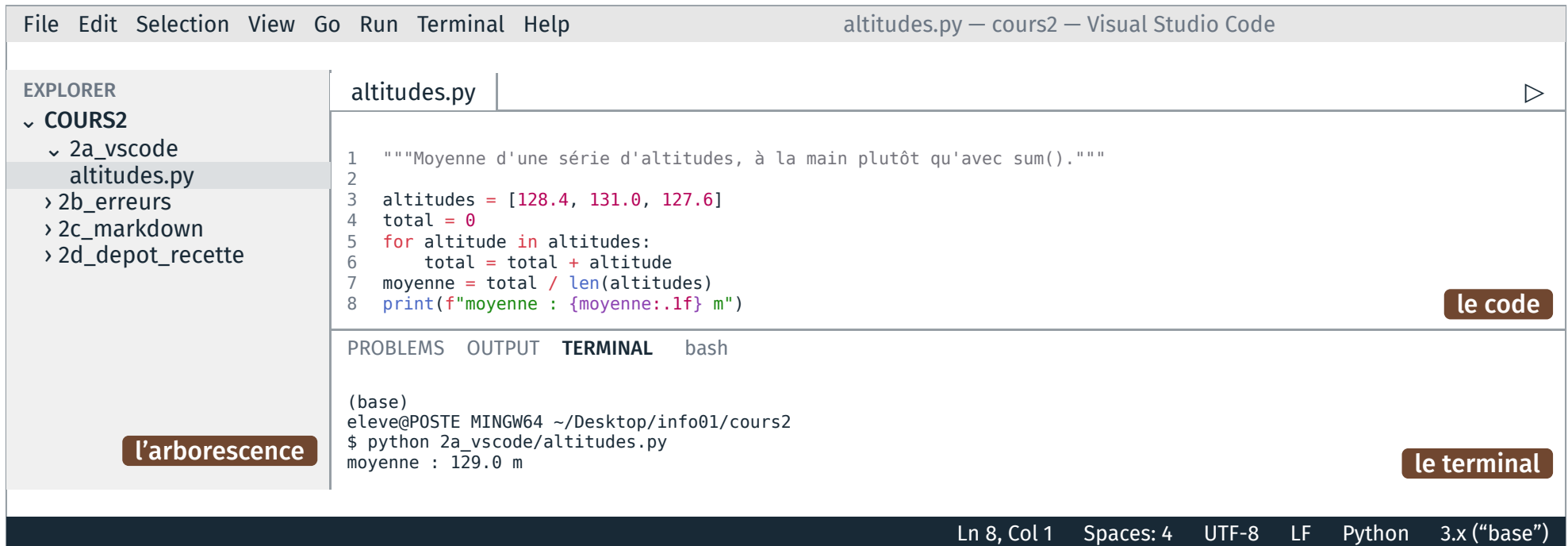
| Ingrédient | Quantité |
|------------|----------|
| Farine     | 250 g    |
| Œufs       | 4        |

## Préparation

1. Mélanger la farine et le sel.
2. Casser les œufs.

# Visual Studio Code

L'éditeur du module a trois zones : l'arborescence du dossier ouvert, le fichier, et le terminal, qui part de ce dossier.



## TRAVAUX DIRIGÉS

---

**TD 2a** Configurer VS Code  
cours2/2a\_vscode/

≈ 25'

## TRAVAUX DIRIGÉS

---

**TD 2b** Trois programmes fautifs  
cours2/2b\_erreurs/

≈ 10'

## TRAVAUX DIRIGÉS

---

**TD 2c** Une recette en Markdown, convertie par pandoc  
cours2/2c\_markdown/

≈ 12'

---

# Git local

Enregistrer les versions successives d'un projet, les comparer, travailler sur une variante et la fusionner.

# À quoi sert git

git enregistre les versions successives des fichiers d'un projet. Chaque version se relit, se compare et se restaure.

**Sans git : une copie par version**

```
recette.md  
recette_v2.md  
recette_v2_corrigee.md  
recette_finale.md  
recette_finale_OK.md
```

**Avec git : un fichier et son historique**

```
$ git log --oneline  
28cbb91 Ignore les fichiers produits par pandoc  
fe4fe8b Réduit les œufs à trois  
a16d497 Ajoute la recette des crêpes
```

Chaque ligne de droite est un commit : une version enregistrée, avec son identifiant et un message qui décrit ce qui a changé.

# Les systèmes de version

Un système centralisé garde l'historique sur un serveur. Dans un système distribué, comme git, chaque copie du projet contient tout l'historique.

Centralisé : CVS (1986), Subversion (2000)

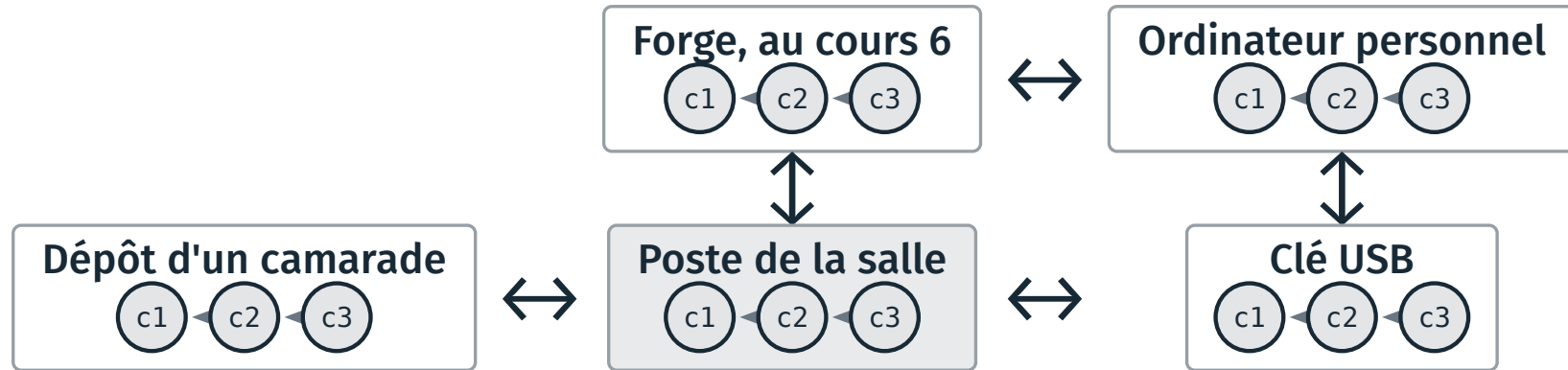
Distribué : git, Mercurial (2005)



À gauche, un poste n'a que la dernière version, et il faut le serveur pour faire un commit ou lire l'historique. À droite, chaque dépôt a tous les commits, et un commit se fait sans réseau.

# Les copies d'un dépôt

Une copie d'un dépôt git a tous ses commits. Deux copies échangent ensuite leurs nouveaux commits, sur un même poste, par une clé USB ou par un serveur.



`git clone` crée une copie. `git pull` récupère les nouveaux commits d'une autre copie, `git push` y envoie les siens.

# La commande git et ses sous-commandes

git est un programme en ligne de commande. Le mot qui suit `git` est une sous-commande, qui a ses propres options et ses propres arguments.

| Commande tapée                                 | Sous-commande       | Options                             | Arguments                           |
|------------------------------------------------|---------------------|-------------------------------------|-------------------------------------|
| <code>git init</code>                          | <code>init</code>   |                                     |                                     |
| <code>git add recette.md crepes.jpg</code>     | <code>add</code>    |                                     | <code>recette.md, crepes.jpg</code> |
| <code>git commit -m "Ajoute la recette"</code> | <code>commit</code> | <code>-m "Ajoute la recette"</code> |                                     |
| <code>git log --oneline</code>                 | <code>log</code>    | <code>--oneline</code>              |                                     |
| <code>git add -h</code>                        | <code>add</code>    | <code>-h</code>                     |                                     |

Forme : `git <sous-commande> [options] <arguments>`, celle des commandes du cours 1 avec un mot de plus. `-m` prend une valeur : le message.

# L'aide de git

`git --help` liste les sous-commandes courantes. `-h` après une sous-commande affiche ses options.

```
git --help
```

```
usage : git [-v | --version] [-h | --help] ...  
        ... <command> [<args>]
```

Ci-dessous les commandes Git habituelles dans diverses situations :

démarrer une zone de travail

|       |                                            |
|-------|--------------------------------------------|
| clone | Cloner un dépôt dans un nouveau répertoire |
| init  | Créer un dépôt Git vide ...                |

travailler sur la modification actuelle

|     |                                             |
|-----|---------------------------------------------|
| add | Ajouter le contenu de fichiers dans l'index |
| ... |                                             |

```
git add -h
```

```
usage : git add [<options>] [--] <chemin>...
```

|                    |                                       |
|--------------------|---------------------------------------|
| -n, --[no-]dry-run | simuler l'action                      |
| -v, --[no-]verbose | mode verbeux                          |
| ...                |                                       |
| -f, --[no-]force   | permettre l'ajout de fichiers ignorés |
| ...                |                                       |

`git add --help` ouvre la page complète du manuel : dans le navigateur sous Windows, dans le terminal sous Linux et macOS (q pour quitter).

# Créer un dépôt : git init

`git init` fait du dossier courant un dépôt : il y crée un dossier caché, `.git`, qui contiendra l'historique.

```
$ cd ~/Desktop/info01/cours2/2d_depot_recette/travail
$ pwd
/c/Users/eleve/Desktop/info01/cours2/2d_depot_recette/travail
$ git init
Dépôt Git vide initialisé dans ../2d_depot_recette/travail/.git/
$ ls -a
.
..
crepes.jpg
.git
recette.md
```

Les fichiers du dossier ne changent pas. Supprimer `.git` supprime l'historique, et laisse les fichiers.

# Trois zones : dossier, index, dépôt

Un commit enregistre ce que `git add` a placé dans l'index, et rien d'autre.



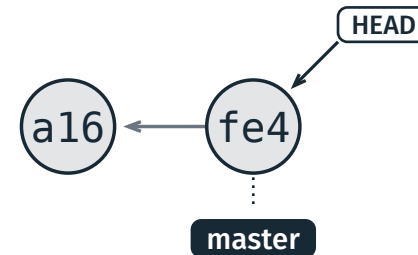
| Ce qu'affiche <code>git status</code>                                 | Zone                   |
|-----------------------------------------------------------------------|------------------------|
| « Fichiers non suivis », « Modifications qui ne seront pas validées » | dossier de travail     |
| « Modifications qui seront validées »                                 | index                  |
| « rien à valider, la copie de travail est propre »                    | tout est dans le dépôt |

`git status` affiche aussi la branche courante et les commandes qui s'appliquent. La taper avant toute autre, et avant de demander de l'aide.

# Enregistrer une version : add, puis commit

`git add` place un fichier dans l'index ; `git commit` enregistre l'index comme une nouvelle version, avec un message.

```
$ git add recette.md crepes.jpg
$ git commit -m "Ajoute la recette des crêpes"
[master (commit racine) a16d497] Ajoute la recette des crêpes
2 files changed, 24 insertions(+)
create mode 100644 crepes.jpg
create mode 100644 recette.md
...
$ git log --oneline
fe4fe8b Réduit les œufs à trois
a16d497 Ajoute la recette des crêpes
```



`a16d497` est le début de l'identifiant du commit, calculé à partir de son contenu. Chaque commit désigne son parent, d'où la flèche de `fe4` vers `a16` ; `master` désigne le dernier commit, et avance à chaque commit.

# Le message de commit

Le message décrit ce que le commit change, en une ligne, avec un verbe au présent. Un commit porte une seule modification.

| Message                                 | Ce qu'on en apprend dans six mois |
|-----------------------------------------|-----------------------------------|
| modif                                   | rien                              |
| corrections                             | rien : lesquelles ?               |
| Réduit les œufs à trois                 | la quantité d'œufs a changé       |
| Ignore les fichiers produits par pandoc | d'où vient le .gitignore          |

Le verbe complète « Ce commit... » : ajoute, corrige, remplace, ignore.

# Ce qui a changé : git diff

`git diff` montre les lignes modifiées depuis le dernier commit : en - l'ancienne version, en + la nouvelle, avec trois lignes de contexte.

```
$ git diff
diff --git a/recette.md b/recette.md
index aae4747..45bc5f0 100644
--- a/recette.md
+++ b/recette.md
@@ -9,7 +9,7 @@
| Ingrédient | Quantité |
|---|---|
| Farine | 250 g |
-| Œufs | 4 |
+| Œufs | 3 |
| Lait | 500 ml |
| Sel | 1 pincée |
| Beurre fondu | 50 g |
```

`@@ -9,7 +9,7 @@` : le passage commence à la ligne 9, et fait 7 lignes avant comme après. VS Code montre le même écart côte à côte.

# Annuler une modification non validée : git restore

`git restore` remet un fichier dans l'état du dernier commit. La modification est perdue : elle n'avait pas été enregistrée.

## Une ligne abîmée par erreur

```
$ git diff
@@ -21,4 +21,4 @@
    5. Laisser reposer une heure.
   -6. Cuire dans une poêle chaude, une minute par face.
   +6. Cuire.
```

## Remise en état

```
$ git restore recette.md
$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

Sortie de gauche réduite à la ligne @@ et aux lignes modifiées ; les quatre lignes d'en-tête du `diff` sont celles de la diapositive précédente.

# Ignorer des fichiers : .gitignore

Un fichier `.gitignore` liste des motifs de noms. `git status` n'affiche plus les fichiers qui y correspondent, et `git add` ne les ajoute pas.

| Ce qu'on ne versionne pas              | Pourquoi                                   | Motif    |
|----------------------------------------|--------------------------------------------|----------|
| les fichiers produits par une commande | ils se refont depuis leur source           | *.html   |
| les données de travail                 | chaque version resterait dans l'historique | donnees/ |
| les mots de passe et les clés          | toute copie du dépôt les contiendrait      | .env     |

`.gitignore`

`ls`

`git status`

```
*.html
*.odt
```

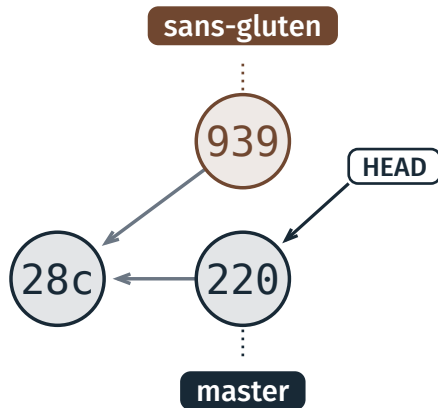
```
crepes.jpg
recette.html
recette.md
recette.odt
```

```
Sur la branche master
rien à valider, la copie de
travail est propre
```

`.gitignore` se versionne avec le projet.

# Les branches

Une branche est un nom posé sur un commit. Elle avance d'un commit à chaque commit fait sur elle ; HEAD désigne la branche courante.



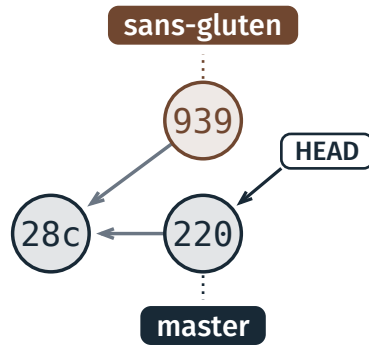
```
$ git checkout -b sans-gluten
Basculement sur la nouvelle branche 'sans-gluten'
...
$ git checkout master
...
$ git log --oneline --graph --all --decorate
* 22060f2 (HEAD -> master) Ajoute un conseil...
| * 9399bb0 (sans-gluten) Remplace la farine...
|/
* 28cbb91 Ignore les fichiers produits...
```

`git checkout -b nom` crée une branche et s'y place ; `git checkout nom` change de branche ; `git branch` les liste.

# Changer de branche change les fichiers

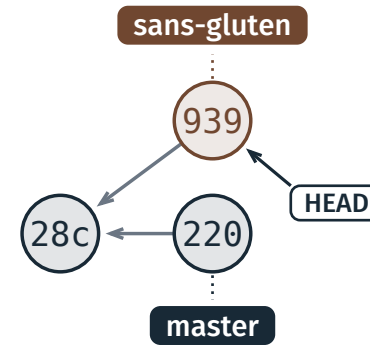
`git checkout` remplace les fichiers du dossier par ceux du dernier commit de la branche choisie.

Sur master



| Farine | 250 g |

Après `git checkout sans-gluten`



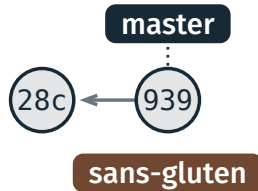
| Farine de sarrasin | 250 g |

Sous chaque graphe, la ligne de la farine dans `recette.md`, ouvert dans VS Code.

# Fusionner : git merge

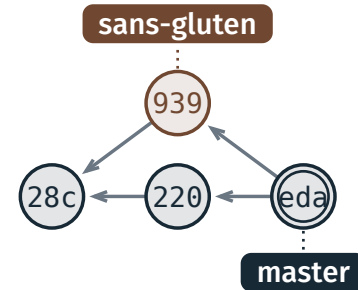
`git merge` apporte dans la branche courante les commits d'une autre branche. Quand les deux ont avancé, `git` crée un commit de fusion.

Seule `sans-gluten` a avancé



`master` avance jusqu'au commit de la branche (avance rapide).

Les deux branches ont avancé

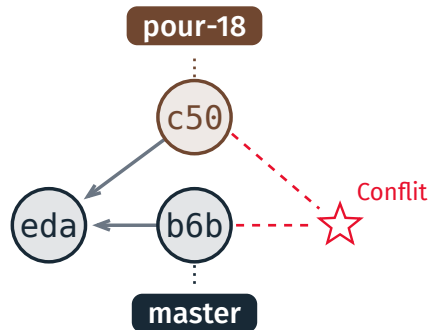


eda, le commit de fusion, a deux parents (double trait).

```
$ git merge sans-gluten -m "Fusionne la variante sans gluten"
Fusion automatique de recette.md
Merge made by the 'ort' strategy.
```

# Un conflit de fusion

Quand les deux branches ont modifié la même ligne, git arrête la fusion et écrit les deux versions dans le fichier.



| Commit       | La troisième ligne de <code>recette.md</code>     |
|--------------|---------------------------------------------------|
| eda, avant   | *Pour 12 crêpes – ..., 1 heure de repos.*         |
| c50, pour-18 | *Pour <u>18</u> crêpes – ..., 1 heure de repos.*  |
| b6b, master  | *Pour 12 crêpes – ..., <u>2 heures</u> de repos.* |

```
$ git merge pour-18
Fusion automatique de recette.md
CONFLICT (contenu) : Conflit de fusion dans recette.md
```

`pour-18` a changé le nombre de crêpes, `master` le temps de repos : la même ligne, modifiée des deux côtés depuis `eda`. La partie modifiée est soulignée, et « ... » abrège « 10 minutes de préparation ».

# Résoudre un conflit

Écrire la ligne voulue à la place des deux versions et des marqueurs, puis `git add` et `git commit` terminent la fusion.

recette.md **après** `git merge`

```
# Crêpes

<<<<<<< HEAD
*Pour 12 crêpes – 10 minutes de
préparation, 2 heures de repos.*
=====
*Pour 18 crêpes – 10 minutes de
préparation, 1 heure de repos.*
>>>>>>> pour-18
```

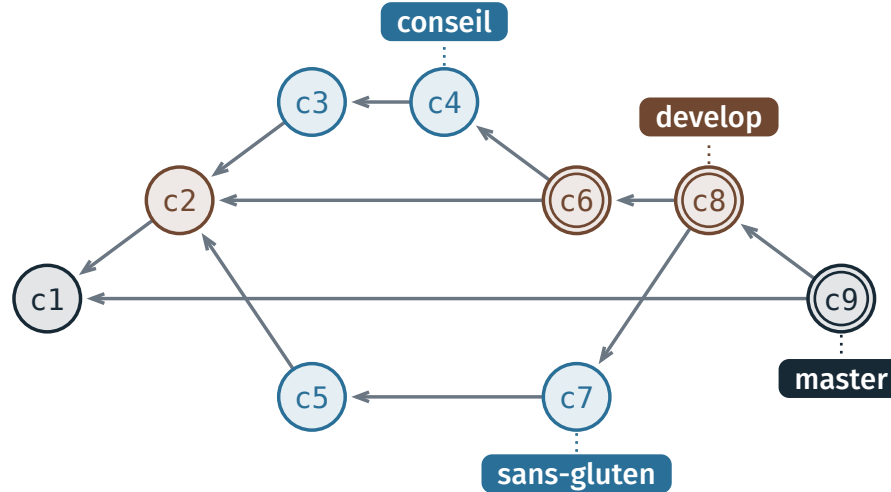
La ligne écrite, puis la fin de la fusion

```
*Pour 18 crêpes – 10 minutes de
préparation, 2 heures de repos.*
$ git add recette.md
$ git commit -m "Fusionne la version
pour 18 crêpes"
[master e87b68f] Fusionne la version
pour 18 crêpes
```

Entre `<<<<<<< HEAD` et `=====`, la ligne de la branche courante ; entre `=====` et `>>>>>>>`, celle de la branche fusionnée. La bonne ligne prend un morceau de chacune.

# L'organisation des branches à plusieurs

`master` ne reçoit que les versions terminées. `develop` réunit le travail en cours ; chaque tâche se fait sur sa propre branche, fusionnée dans `develop` quand elle est terminée.



Alice fait le conseil, Bruno la variante sans gluten, chacun dans sa copie du dépôt. c6 et c8 fusionnent les tâches dans `develop` ; c9 porte la version terminée dans `master`.

## TRAVAUX DIRIGÉS

---

**TD 2d**    Un dépôt git pour la recette  
cours2/2d\_depot\_recette/

≈ 40'

# À retenir

|                         |                                                                                       |
|-------------------------|---------------------------------------------------------------------------------------|
| Un éditeur de code      | colore, vérifie et lance le code ; son terminal est Git Bash                          |
| Un réglage              | User pour le compte, Workspace pour le dossier, écrit dans <code>settings.json</code> |
| Un commit               | une version enregistrée : ce que <code>git add</code> a mis dans l'index              |
| <code>git status</code> | la commande à taper avant toute autre                                                 |
| <code>.gitignore</code> | ce qui se refait ne se versionne pas                                                  |
| Une branche             | une variante, qui se fusionne ; un conflit se résout à la main                        |
| Une copie d'un dépôt    | tout l'historique ; les copies échangent leurs commits (cours 6)                      |