Rédaction mathématique et informatique

Cette semaine

LaTeX :

- retour sur l'inclusion de code
- insertion de code
- environnements flottants

Cette semaine

LaTeX :

- retour sur l'inclusion de code ⟵ LaTeX exécute le code
- insertion de code ⟵ LaTeX affiche le code
- environnements flottants

Cette semaine

LaTeX :

- retour sur l'inclusion de code ⟵ LaTeX exécute le code
- insertion de code ⟵ LaTeX affiche le code
- environnements flottants

git :

- retour sur le statut d'un dépôt
  - notion d'index
  - retour sur le fichier `.gitignore`
- branches et fusions de versions (aperçu)
  - notion de branche
  - fusion (*merge*) automatique et conflictuelle

Cette semaine

LaTeX :

- retour sur l'inclusion de code ⟵ LaTeX exécute le code
- insertion de code ⟵ LaTeX affiche le code
- environnements flottants

git :

- retour sur le statut d'un dépôt
  - notion d'index
  - retour sur le fichier `.gitignore`
- branches et fusions de versions (aperçu)
  - notion de branche
  - fusion (*merge*) automatique et conflictuelle

Questions : → réponses

Inclure du contenu LaTeX depuis un autre fichier

Inclure du contenu depuis un autre fichier (rappels)

La commande `\input{chemin-vers-fichier}`
inclut le contenu du fichier indiqué par chemin-vers-fichier
dans le code LaTeX (à l'endroit où est placé la commande)

subfile.tex

```
blabla


blabla
```

mainfile.tex

```
bla
\input{subfile}
bla
```

≡

"ce que voit LaTeX"

```
bla
blabla


blabla
bla
```

But :

- isoler des parties complexes de code
- structurer le code (*e.g.*, chaque section dans un fichier différent)
- réutiliser des morceaux de code dans différents document
- collaborer efficacement (fusions automatique assurées avec git)

Insérer du code

But: Afficher des bouts de code (programmation), *e.g.* :

1 Programmation

Que fait le code PYTHON suivant ?

```
message = "Je vous le dis: \"ceci n'est pas un code\"."
my_modulo = 16
max_value = 32
#construction des dictionnaires
d = [
{ 'mod': i%my_modulo, 'value': i }
for i in range(max_value+1)
]
assert d[5]['mod'] == d[21]['value']%my_modulo
```

But: Afficher des bouts de code (programmation), *e.g.* :

1 Programmation

Que fait le code PYTHON suivant ?

```
message = "Je vous le dis: \"ceci n'est pas un code\"."
my_modulo = 16
max_value = 32
#construction des dictionnaires
d = [
{ 'mod': i%my_modulo, 'value': i }
for i in range(max_value+1)
]
assert d[5]['mod'] == d[21]['value']%my_modulo
```

Problème: Les caractères spéciaux doivent être *échappés* :

code LaTeX	rendu pdf
\textbackslash	\
\%	%
\{	{
_	_
...	...

Nécessité

Sur un exemple bref, ça peut encore aller, *e.g.*,

le code : Faire `\texttt{\textbackslash LaTeX}` pour `\LaTeX`.

donne le résultat : Faire \LaTeX pour LaTeX.

Nécessité

Sur un exemple bref, ça peut encore aller, *e.g.*,

le code : Faire `\texttt{\textbackslash LaTeX}` pour `\LaTeX`.

donne le résultat : Faire \LaTeX pour LaTeX.

mais sur un exemple plus grand, . . .

Nécessité

Sur un exemple bref, ça peut encore aller, *e.g.*,

le code : Faire `\texttt{\textbackslash LaTeX}` pour `\LaTeX`.

donne le résultat : Faire \LaTeX pour LaTeX.

mais sur un exemple plus grand, . . . quelle horreur ! quel enfer !

```
\section{Programmation}
Que fait le code \textsc{Python} suivant?

{%
  \ttfamily%
  message = "Je vous le dis: \textbackslash"ceci
  n\textquotesingle est pas un code\textbackslash"."\newline
  my\_modulo = 16\newline
  max\_value = 32\newline
  \#construction des dictionnaires\newline
  d = [\newline
  \{ \textquotesingle mod\textquotesingle: i\%my\_modulo,
  \textquotesingle value\textquotesingle: i \}\newline
  for i in range(max\_value+1)\newline
  ]\newline
  assert d[5][\textquotesingle mod\textquotesingle] ==
  d[21][\textquotesingle value\textquotesingle]\%my\_modulo
}
```

Verbatim

Le mode *verbatim* permet de définir des blocs où LaTeX n'interprète rien.

Verbatim

Le mode *verbatim* permet de définir des blocs où LaTeX n'interprète rien.

De base :

- la commande `\verb` (une ligne)

- l'environnement `verbatim` (plusieurs lignes)

Verbatim

Le mode *verbatim* permet de définir des blocs où LaTeX n'interprète rien.

De base :

- la commande `\verb` (une ligne)

bloc délimité par un caractère spécial

```
Faire \verb+\LaTeX+ pour \LaTeX.
```
→ Faire \LaTeX pour LaTeX.

- l'environnement `verbatim` (plusieurs lignes)

Verbatim

Le mode *verbatim* permet de définir des blocs où LaTeX n'interprète rien.

De base :

- la commande `\verb` (une ligne)

bloc délimité par un caractère spécial au choix

Faire `\verb|\LaTeX| pour \LaTeX.`	→	Faire \LaTeX pour LaTeX.

- l'environnement `verbatim` (plusieurs lignes)

Verbatim

Le mode *verbatim* permet de définir des blocs où LaTeX n'interprète rien.

De base :

- la commande `\verb` (une ligne)

bloc délimité par un caractère spécial au choix

```
Faire \verb|\LaTeX| pour \LaTeX.
```
→ Faire \LaTeX pour LaTeX.

- l'environnement `verbatim` (plusieurs lignes)

bloc délimité par `\begin{verbatim}`/`\end{verbatim}`

```
ah   bon?  %interprété
\begin{verbatim}
  ah   bon?  %affiché
  \section{Un titre}
  \label{sec/exemple}
\end{verbatim}
```
→
```
ah bon?
ah   bon?  %affiché
\section{Un titre}
\label{sec/exemple}
```

Verbatim

Le mode *verbatim* permet de définir des blocs où LaTeX n'interprète rien.

De base :

- la commande `\verb` (une ligne)

bloc délimité par un caractère spécial au choix

```
Faire \verb|\LaTeX| pour \LaTeX.
```
→
Faire \LaTeX pour LaTeX.

- l'environnement `verbatim` (plusieurs lignes)

bloc délimité par `\begin{verbatim}`/`\end{verbatim}`

```
ah   bon?  %interprété
\begin{verbatim}
  ah   bon?  %affiché
  \section{Un titre}
  \label{sec/exemple}
\end{verbatim}
```
→
```
ah bon?
ah   bon?  %affiché
\section{Un titre}
\label{sec/exemple}
```

Les retours à la ligne et les espaces sont préservés.

Exemple

```
\section{Programmation}
Que fait le code \textsc{Python} suivant?
\begin{verbatim}
  message = "Je vous le dis: \"ceci n'est pas un code\"."
  my_modulo = 16
  max_value = 32
  #construction des dictionnaires
  d = [
    { 'mod': i%my_modulo, 'value': i }
    for i in range(max_value+1)
  ]
  assert d[5]['mod'] == d[21]['value']%my_modulo
\end{verbatim}
```

le code LaTeX

le résultat pdf

1 Programmation

Que fait le code PYTHON suivant ?

```
message = "Je vous le dis: \"ceci n'est pas un code\"."
my_modulo = 16
max_value = 32
#construction des dictionnaires
d = [
{ 'mod': i%my_modulo, 'value': i }
for i in range(max_value+1)
]
assert d[5]['mod'] == d[21]['value']%my_modulo
```

Peut-on faire mieux ?

Peut-on faire mieux ?

Oui.

Peut-on faire mieux ?

Oui. Avec des paquets prévus pour.

Peut-on faire mieux ?

Oui. Avec des paquets prévus pour.

Notamment avec,

- le paquet `listing`
- le paquet `minted` + outils extérieurs (`pygments`)

Listings

Le paquet `listings` permet d'inclure du code
et d'appliquer des styles selon le langage de programmation

Listings

Le paquet `listings` permet d'inclure du code
et d'appliquer des styles selon le langage de programmation

Il définit :

- la commande `\lstinline` (analogue de `\verb`)
- l'environnement `lstlisting` (analogue de `verbatim`)

Listings

Le paquet `listings` permet d'inclure du code
et d'appliquer des styles selon le langage de programmation

Il définit :

- la commande `\lstinline` (analogue de `\verb`)
- l'environnement `lstlisting` (analogue de `verbatim`)
- la commande `\lstinputlisting` : pour lire le code depuis un fichier

Listings

Le paquet `listings` permet d'inclure du code
et d'appliquer des styles selon le langage de programmation

Il définit :

- la commande `\lstinline` (analogue de `\verb`)
- l'environnement `lstlisting` (analogue de `verbatim`)
- la commande `\lstinputlisting` : pour lire le code depuis un fichier

```
\lstinputlisting{./projets/helloworld.py}
```

Listings

Le paquet `listings` permet d'inclure du code
et d'appliquer des styles selon le langage de programmation

Il définit :

- la commande `\lstinline` (analogue de `\verb`)
- l'environnement `lstlisting` (analogue de `verbatim`)
- la commande `\lstinputlisting` : pour lire le code depuis un fichier

```
\lstinputlisting{./projets/helloworld.py}
```

Il offre des options :

- spécification du langage

```
\lstinline[language=python]{print("Hello World")}
```

- numérotation des lignes
- échappement (pour quand-même interpréter des commande LaTeX)
- définition de style, ajouts de mot-clés...

Listings

Le paquet `listings` permet d'inclure du code
et d'appliquer des styles selon le langage de programmation

Il définit :

- la commande `\lstinline` (analogue de `\verb`)
- l'environnement `lstlisting` (analogue de `verbatim`)
- la commande `\lstinputlisting` : pour lire le code depuis un fichier

```
\lstinputlisting{./projets/helloworld.py}
```

- la commande `\lstset` pour définir des options globalement

Il offre des options :

- spécification du langage

```
\lstinline[language=python]{print("Hello World")}
```

- numérotation des lignes
- échappement (pour quand-même interpréter des commande LaTeX)
- définition de style, ajouts de mot-clés...

Insertion de code

Le paquet `upquote` (pour `\verb`/`verbatim`)
ou l'option `upquote` de `listing` (pour `\lstinline`/`lstlisting`)
pour des vrais *quotes* de code (`` ` ' `` plutôt que ‘ ’)

Insertion de code

Le paquet `upquote` (pour `\verb`/`verbatim`)
ou l'option `upquote` de `listing` (pour `\lstinline`/`lstlisting`)
pour des vrais *quotes* de code (`` ` `` ' plutôt que ‘ ’)

Pour insérer des mot-clés de code dans un paragraphe,
on utilise les commandes `\texttt`/`\ttfamily` (`tt` pour ***t****eletype* ***t****ext*)
(la 1re s'applique à son argument obligatoire, la 2nde change la police courante)
c'est le style utilisé (par défaut) par `\verb` et cie

Insertion de code

Le paquet `upquote` (pour `\verb`/`verbatim`)
ou l'option `upquote` de `listing` (pour `\lstinline`/`lstlisting`)
pour des vrais *quotes* de code (`` ` `` ' plutôt que ‘ ’)

Pour insérer des mot-clés de code dans un paragraphe,
on utilise les commandes `\texttt`/`\ttfamily` (`tt` pour ***t****eletype* ***t****ext*)
(la 1re s'applique à son argument obligatoire, la 2nde change la police courante)
c'est le style utilisé (par défaut) par `\verb` et cie

Le paquet `minted` utilise un outil extérieur pour colorer du code
nécessite l'option de compilation `-shell-escape` ⚠ et `pygments`.

Exemple

Le tex avec listing (environnement lstlisting) :

```
\section{Programmation}
Que fait le code \textsc{Python} suivant?
\begin{lstlisting}[
    language=python, basicstyle=\scriptsize, tabsize=2, gobble=4
  ]
  message = "Je vous le dis: \"ceci n'est pas un code\"."
  my_modulo = 16
  max_value = 32
  #construction des dictionnaires
  d = [
    { 'mod': i%my_modulo, 'value': i }
    for i in range(max_value+1)
  ]
  assert d[5]['mod'] == d[21]['value']%my_modulo
\end{lstlisting}%
```

Exemple

Le `pdf` avec `listing` (environnement `lstlisting`) :

1 Programmation

Que fait le code PYTHON suivant ?

```
message = "Je vous le dis: \"ceci n'est pas un code\"."
my_modulo = 16
max_value = 32
#construction des dictionnaires
d = [
  { 'mod': i%my_modulo, 'value': i }
  for i in range(max_value+1)
]
assert d[5]['mod'] == d[21]['value']%my_modulo
```

(on pourrait améliorer le style)

Exemple

Le `tex` avec `listing` (commande `\lstinputlisting`) :

```
\section{Programmation}
Que fait le code \textsc{Python} suivant?
\lstinputlisting[%
  language=Python,
  basicstyle=\scriptsize,
  commentstyle=\color{NavyBlue},
  showstringspaces=false
]{verbatim-python.py}%
```

Exemple

Le `pdf` avec `listing` (commande `\lstinputlisting`) :

1 Programmation

Que fait le code PYTHON suivant ?

```
message = "Je vous le dis: \"ceci n'est pas un code\"."
my_modulo = 16
max_value = 32
#construction du dictionnaire
d = [
    { 'mod': i%my_modulo, 'value': i }
    for i in range(max_value+1)
]
assert d[5]['mod'] == d[21]['value']%my_modulo
```

(on pourrait améliorer le style)

Exemple

Le `tex` avec `minted` :

```
\section{Programmation}
Que fait le code \textsc{Python} suivant?
\inputminted{python}{verbatim-python.py}%
```

Exemple

Le pdf avec minted :

1 Programmation

Que fait le code PYTHON suivant ?

```
message = "Je vous le dis: \"ceci n'est pas un code\"."
my_modulo = 16
max_value = 32
#construction du dictionnaire
d = [
    { 'mod': i%my_modulo, 'value': i }
    for i in range(max_value+1)
]
assert d[5]['mod'] == d[21]['value']%my_modulo
```

Les environnements flottants

Rappel: espacement vertical

Rappel: espacement vertical

principe : LaTeX décide où placer les changements de page

Rappel: espacement vertical

principe : LaTeX décide où placer les changements de page
moyens : LaTeX fait varier la taille des espaces verticaux

Rappel: espacement vertical

principe : LaTeX décide où placer les changements de page
moyens : LaTeX fait varier la taille des espaces verticaux
problèmes : certaines portions du document ne doivent pas être coupées :

- tableaux
- graphiques
- images
- code
- algorithmes
- ...

Rappel: espacement vertical

principe : LaTeX décide où placer les changements de page
moyens : LaTeX fait varier la taille des espaces verticaux
problèmes : certaines portions du document ne doivent pas être coupées :
▶ tableaux ▶ graphiques ▶ images ▶ code ▶ algorithmes ▶ . . .
solution : on sort ces parties du flux du document,
on laisse LaTeX les placer au mieux (avant, après, à la fin ?)
et on y fait référence (*e.g.*, voir Figure 3)

Rappel: espacement vertical

principe : LaTeX décide où placer les changements de page
moyens : LaTeX fait varier la taille des espaces verticaux
problèmes : certaines portions du document ne doivent pas être coupées :
▶ tableaux ▶ graphiques ▶ images ▶ code ▶ algorithmes ▶ ...
solution : on sort ces parties du flux du document,
on laisse LaTeX les placer au mieux (avant, après, à la fin ?)
et on y fait référence (*e.g.*, voir Figure 3)

On parle d'élément **flottant**.

Rappel: espacement vertical

principe : LaTeX décide où placer les changements de page
moyens : LaTeX fait varier la taille des espaces verticaux
problèmes : certaines portions du document ne doivent pas être coupées :
▶ tableaux ▶ graphiques ▶ images ▶ code ▶ algorithmes ▶ ...
solution : on sort ces parties du flux du document,
on laisse LaTeX les placer au mieux (avant, après, à la fin ?)
et on y fait référence (*e.g.*, voir Figure 3)

On parle d'élément **flottant**.

Les environnements `figure` et `table` indiquent à LaTeX
qu'un bloc est une figure ou une table, à placer de manière flottante.

Rappel: espacement vertical

principe : LaTeX décide où placer les changements de page
moyens : LaTeX fait varier la taille des espaces verticaux
problèmes : certaines portions du document ne doivent pas être coupées :
▶ tableaux ▶ graphiques ▶ images ▶ code ▶ algorithmes ▶ ...
solution : on sort ces parties du flux du document,
on laisse LaTeX les placer au mieux (avant, après, à la fin ?)
et on y fait référence (*e.g.*, voir Figure 3)

On parle d'élément **flottant**.

Les environnements `figure` et `table` indiquent à LaTeX
qu'un bloc est une figure ou une table, à placer de manière flottante.

L'option `float` de l'environnement `lstlisting` indique à LaTeX
que l'insertion de code est à faire de manière flottante.

Légende

Les environnements flottants peuvent (et doivent) avoir une légende
le numéro à référencer est dans la légende

Légende

Les environnements flottants peuvent (et doivent) avoir une légende
le numéro à référencer est dans la légende

La commande `\caption` permet de définir la légende, *e.g.*,

Légende

Les environnements flottants peuvent (et doivent) avoir une légende
le numéro à référencer est dans la légende

La commande `\caption` permet de définir la légende, *e.g.*,

```
\begin{figure}
  \caption{Le Puy de Dôme}
  \label{fig/puy de dome}
  \includegraphics[
    width=\linewidth
  ]{
    pics/puydedome
  }
\end{figure}
```

Figure 1 – Le Puy de Dôme

Légende

Les environnements flottants peuvent (et doivent) avoir une légende
le numéro à référencer est dans la légende

La commande `\caption` permet de définir la légende, *e.g.*,

```
\begin{figure}
  \includegraphics[
    width=\linewidth
  ]{
    pics/puydedome
  }
  \caption{Le Puy de Dôme}
  \label{fig/puy de dome}
\end{figure}
```

Figure 1 – Le Puy de Dôme

Des environnements sémantiques

Objectif double :

1. Donner plus de liberté à LaTeX pour gérer les espaces verticaux ;
2. Indiquer un bloc sémantique (*“ceci est une figure”*).

Des environnements sémantiques

Objectif double :

1. Donner plus de liberté à LaTeX pour gérer les espaces verticaux ;
2. Indiquer un bloc sémantique (*“ceci est une figure”*).

Dans les documents longs (*e.g.*, rapport de stage),
il faut en général lister les figures et les tables
avec la table des matières (*i.e.*, liste des parties)

Des environnements sémantiques

Objectif double :

1. Donner plus de liberté à LaTeX pour gérer les espaces verticaux ;
2. Indiquer un bloc sémantique (“*ceci est une figure*”).

Dans les documents longs (*e.g.*, rapport de stage),
il faut en général lister les figures et les tables
avec la table des matières (*i.e.*, liste des parties)

Ceci peut être fait automatiquement avec

- `\listoffigures`
- `\listoftables`
- `\listoflistings`
- `\tableofcontents`

Placement des éléments flottants

Les environnements flottants prennent une option
qui indique où placer l'élément, de préférence :

h : Placer ici (***h****ere*)

t : Placer en haut (***t****op*) de page

b : Placer en bas (***b****ottom*) de page

p : Placer sur une page (***p****age*) particulière réservée aux flottants

! : Forcer le placement

On donne un ordre de préférence (*e.g.*, `\begin{figure}[ht]`).

Placement des éléments flottants

Les environnements flottants prennent une option
qui indique où placer l'élément, de préférence :

h : Placer ici (***h****ere*)

t : Placer en haut (***t****op*) de page

b : Placer en bas (***b****ottom*) de page

p : Placer sur une page (***p****age*) particulière réservée aux flottants

! : Forcer le placement

On donne un ordre de préférence (*e.g.*, `\begin{figure}[ht]`).

⚠ il est généralement préférable de laisser LaTeX gérer cela

Aller plus loin sur les figures

Le paquet `float` permet

1. de (re)définir des styles pour les environnements flottants ;
2. de définir de nouveaux environnements flottants ;
3. d'utiliser le placement `H` qui rend l'élément non-flottant.

Aller plus loin sur les figures

Le paquet `float` permet

1. de (re)définir des styles pour les environnements flottants ;
2. de définir de nouveaux environnements flottants ;
3. d'utiliser le placement `H` qui rend l'élément non-flottant.

Le paquet `subcaption` permet de faire des sous-figures.

Aller plus loin sur les figures

Le paquet `float` permet

1. de (re)définir des styles pour les environnements flottants ;
2. de définir de nouveaux environnements flottants ;
3. d'utiliser le placement `H` qui rend l'élément non-flottant.

Le paquet `subcaption` permet de faire des sous-figures.

Le paquet `wrapfig` par son environnement `wrapfigure`
permet au texte de continuer autour d'une figure.

git

Statut d'un dépôt git

Exemple

```
[user@machine:~/…/repogit]$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
        modifié :         a
        supprimé :        c
        modifié :         e
        nouveau fichier : g
        renommé :         b -> i

Modifications qui ne seront pas validées :
  (utilisez "git add/rm <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
        modifié :         a
        supprimé :        d
        modifié :         f

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
        h

[user@machine:~/…/repogit]$
```

Trois niveaux

git voit :

le répertoire :

l'état des fichiers (tracés ou non) dans le répertoire

git connait :

la version précédente :

l'état des fichiers tracés dans la dernière version

l'index :

l'état des fichiers à prendre en compte dans la prochaine version

Trois niveaux

git voit :

1. le répertoire :
   l'état des fichiers (tracés ou non) dans le répertoire

git connait :

2. la version précédente :
   l'état des fichiers tracés dans la dernière version
3. l'index :
   l'état des fichiers à prendre en compte dans la prochaine version

Cela définit donc **3 niveaux.**

Trois niveaux

git voit :

1. le répertoire :
   l'état des fichiers (tracés ou non) dans le répertoire

git connait :

2. la version précédente :
   l'état des fichiers tracés dans la dernière version
3. l'index :
   l'état des fichiers à prendre en compte dans la prochaine version

Cela définit donc **3 niveaux.**

l'index : sorte de pré-version qui sert à préparer la prochaine version.

Trois niveaux

git voit :

1. le répertoire :
   l'état des fichiers (tracés ou non) dans le répertoire

git connait :

2. la version précédente :
   l'état des fichiers tracés dans la dernière version
3. l'index :
   l'état des fichiers à prendre en compte dans la prochaine version

Cela définit donc **3 niveaux.**

l'index : sorte de pré-version qui sert à préparer la prochaine version.

`git add/rm/mv` : construit l'index

Trois niveaux

git voit :

1. le répertoire :
   l'état des fichiers (tracés ou non) dans le répertoire

git connait :

2. la version précédente :
   l'état des fichiers tracés dans la dernière version
3. l'index :
   l'état des fichiers à prendre en compte dans la prochaine version

Cela définit donc **3 niveaux.**

l'index : sorte de pré-version qui sert à préparer la prochaine version.

`git add/rm/mv` : construit l'index

`git reset` : déconstruit l'index

Statut

`git status`

git indique ce qu'il voit de chaque fichier (tracé ou du répertoire).

Statut

`git status`

git indique ce qu'il voit de chaque fichier (tracé ou du répertoire).

→ Est-il connu de git (*i.e.*, dans la version précédente) ?

si oui

→ A-t-il été altéré (modifié/supprimé/renommé) depuis la dernière version ?

si oui

→ A-t-il été placé dans l'index (*via* `git add/rm/mv`) ?

si non

→ A-t-il été ajouté à l'index (*via* `git add`) ?

Statut

`git status`

git indique ce qu'il voit de chaque fichier (tracé ou du répertoire).

→ Est-il connu de git (*i.e.*, dans la version précédente) ?

si oui

→ A-t-il été altéré (modifié/supprimé/renommé) depuis la dernière version ?

si oui

→ A-t-il été placé dans l'index (*via* `git add/rm/mv`) ?

si oui

→ A-t-il été modifié depuis qu'il a été placé dans l'index ?

si non

→ A-t-il été ajouté à l'index (*via* `git add`) ?

Statut

git status

```
[user@machine:~/…/repogit]$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
        modifié :         a
        supprimé :        c
        modifié :         e
        nouveau fichier : g
        renommé :         b -> i

Modifications qui ne seront pas validées :
  (utilisez "git add/rm <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
        modifié :         a
        supprimé :        d
        modifié :         f

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
        h

[user@machine:~/…/repogit]$
```

Statut

`git status`

git indique ce qu'il voit de chaque fichier (tracé ou du répertoire)

→ Est-il connu de git (*i.e.*, dans la version précédente) ?

si oui

→ A-t-il été altéré (modifié/supprimé/renommé) depuis la dernière version ?

si oui

→ A-t-il été placé dans l'index (*via* `git add/rm/mv`) ?

si oui

→ A-t-il été modifié depuis qu'il a été placé dans l'index ?

si non

→ A-t-il été ajouté à l'index (*via* `git add`) ?

Statut

`git status`

git indique ce qu'il voit de chaque fichier (tracé ou du répertoire)
sauf les fichiers ignorés.

→ Est-il connu de git (*i.e.*, dans la version précédente) ?

si oui

→ A-t-il été altéré (modifié/supprimé/renommé) depuis la dernière version ?

si oui

→ A-t-il été placé dans l'index (*via* `git add/rm/mv`) ?

si oui

→ A-t-il été modifié depuis qu'il a été placé dans l'index ?

si non

→ A-t-il été ajouté à l'index (*via* `git add`) ?

Retour sur le fichier `.gitignore`

Sert à indiquer à git, qu'il doit ignorer des fichiers, c'est à dire [a] :

- ne pas les lister dans `git status`
- refuser de les ajouter

a. sauf si on le force *via* une option, *e.g.*, `git add -f`/`git status --ignored`

Retour sur le fichier `.gitignore`

Sert à indiquer à git, qu'il doit ignorer des fichiers, c'est à dire [a] :

- ne pas les lister dans `git status`
- refuser de les ajouter

Tracer tous les fichiers est souvent non-nécessaire & contre-productif,
d'où le besoin d'ignorer des fichiers.

a. sauf si on le force *via* une option, *e.g.*, `git add -f`/`git status --ignored`

Retour sur le fichier `.gitignore`

Sert à indiquer à git, qu'il doit ignorer des fichiers, c'est à dire [a] :

- ne pas les lister dans `git status`
- refuser de les ajouter

Tracer tous les fichiers est souvent non-nécessaire & contre-productif,
d'où le besoin d'ignorer des fichiers.

Exemple de fichier `.gitignore` :

```
*.aux
*.log
*.out
*.toc
*.pdf
tmp/
exemple-squelette.tex
exemple-listing.tex
```

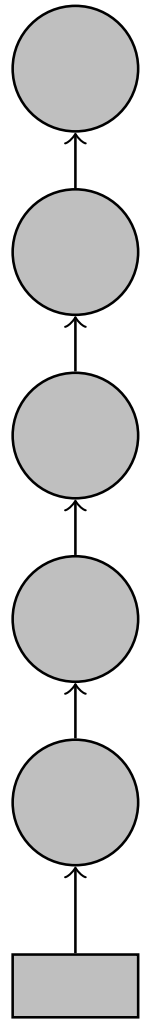
a. sauf si on le force *via* une option, *e.g.*, `git add -f`/`git status --ignored`

Branches git

aperçu avec des simplifications (tout n'est pas rigoureusement exact)

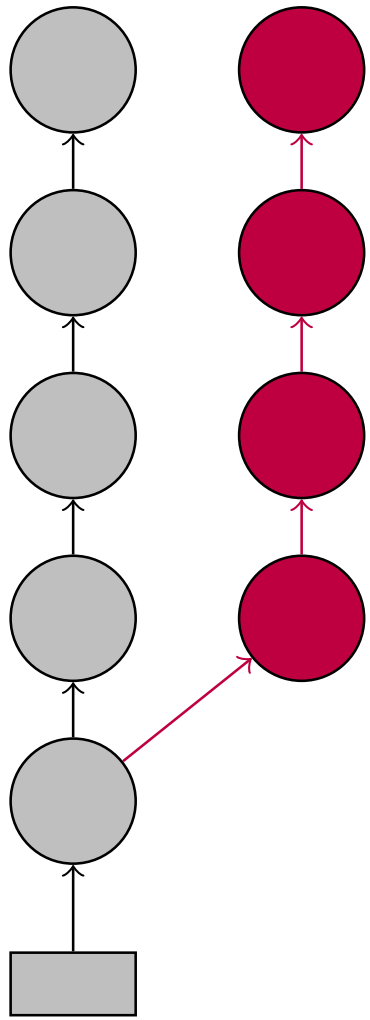
Branches

- chaque version a une version mère
  - sauf la pseudo-version racine (`git init`)



Branches

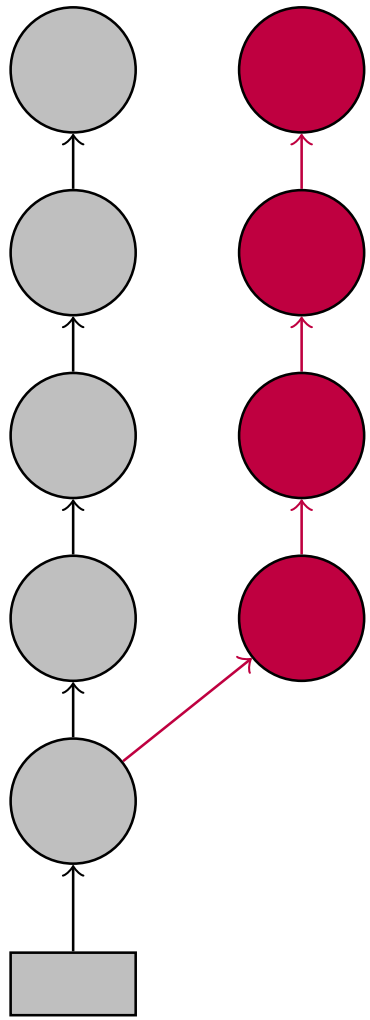
- chaque version a une version mère
  - sauf la pseudo-version racine (`git init`)
- mais une version peut avoir plusieurs versions filles



Branches

- chaque version a une version mère
  – sauf la pseudo-version racine (`git init`)
- mais une version peut avoir plusieurs versions filles

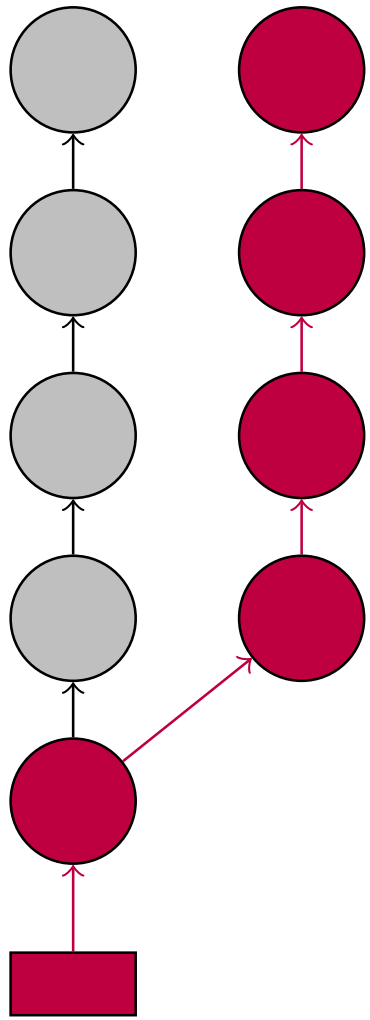
branche git : un chemin racine $\xrightarrow{\star}$ feuille.



Branches

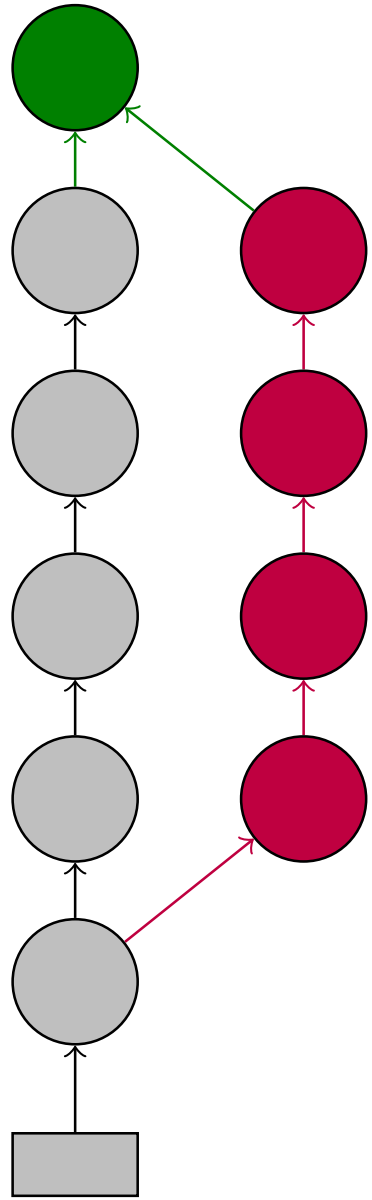
- chaque version a une version mère
  – sauf la pseudo-version racine (`git init`)
- mais une version peut avoir plusieurs versions filles

branche git : un chemin racine $\xrightarrow{\star}$ feuille.



Fusion

Un *merge* : une version qui a deux versions mères.

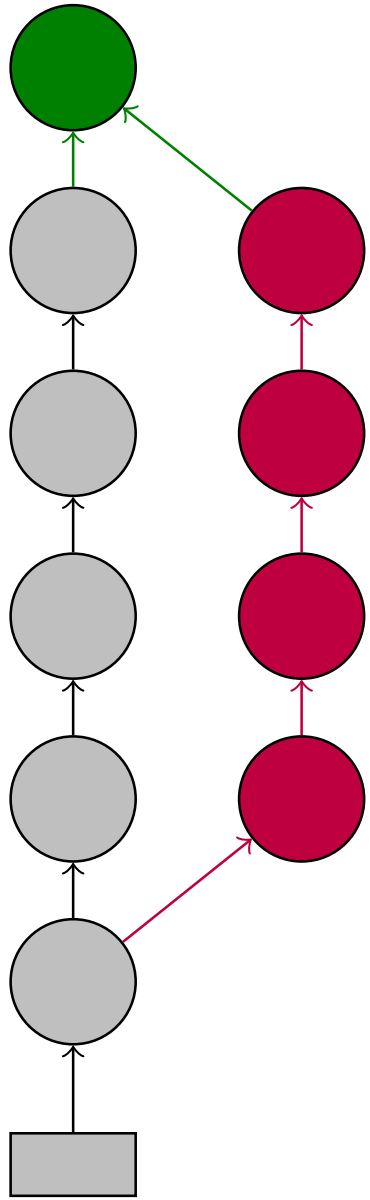


Fusion

Un *merge* : une version qui a deux versions mères.

s'obtient *via* `git merge` ⟨`other`⟩,

ou, `git pull`...



Fusion

Un *merge* : une version qui a deux versions mères.

s'obtient *via* `git merge` ⟨`other`⟩,

ou, `git pull`...

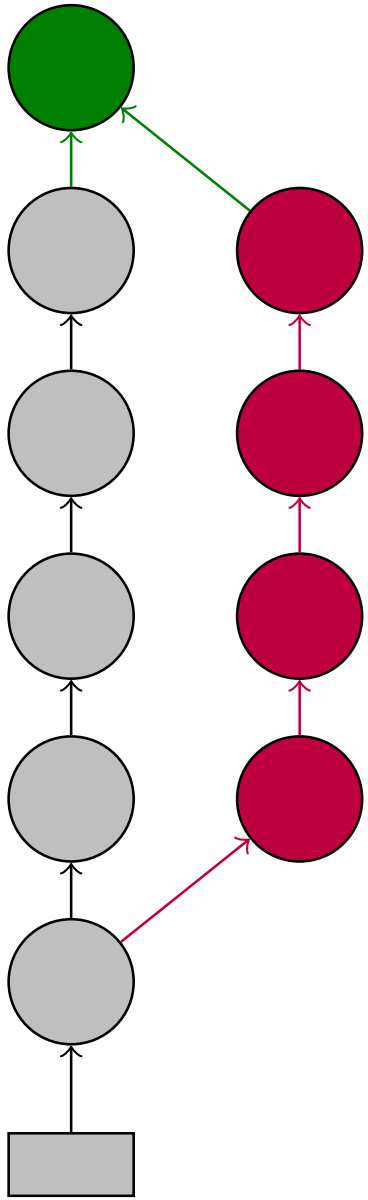
En cas de conflit, git indique :

- les fichiers problématiques
- et à l'intérieur, les zones problématiques par

  `<<<<<<< HEAD:` début de zone

  `=======:` séparateur

  `>>>>>>> other:` fin de zone



Fusion

Un *merge* : une version qui a deux versions mères.

s'obtient *via* `git merge ⟨other⟩`,

ou, `git pull...`

En cas de conflit, git indique :

- les fichiers problématiques
- et à l'intérieur, les zones problématiques par

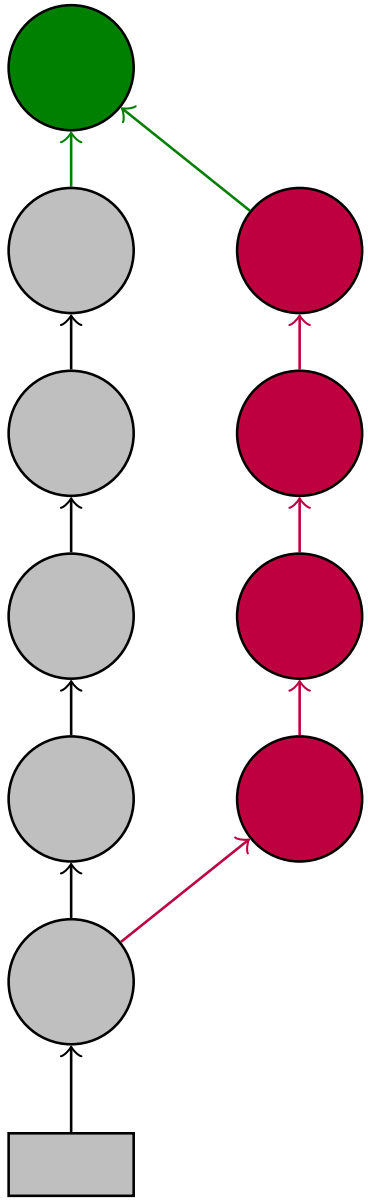
  `<<<<<<< HEAD:` début de zone

  (contenu dans la version de la branche courante)

  `=======:` séparateur

  (contenu dans la version version de l'autre branche)

  `>>>>>>> other:` fin de zone



Fusion

Un *merge* : une version qui a deux versions mères.

s'obtient *via* `git merge` ⟨`other`⟩,

ou, `git pull`...

En cas de conflit, git indique :

- les fichiers problématiques
- et à l'intérieur, les zones problématiques par

  `<<<<<<< HEAD:` début de zone

  (contenu dans la version de la branche courante)

  `=======:` séparateur

  (contenu dans la version version de l'autre branche)

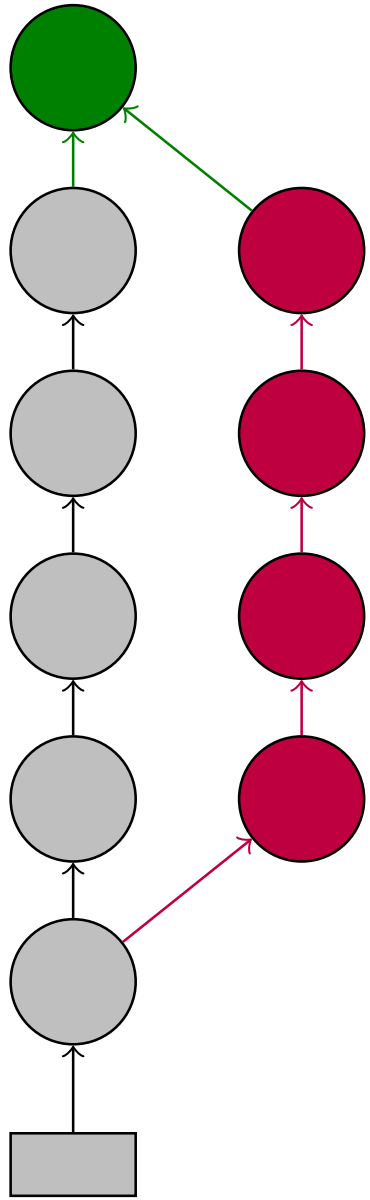
  `>>>>>>> other:` fin de zone

il faut alors :

- éditer les fichiers concernés
- modifier les zones problématiques

  souvent : choisir une version

  parfois, fusionner les deux, à la main
- terminer la fusion avec `git commit` (sans message)



En pratique

Dans ce cours, on se concentre sur une situation simple :

un dépôt local & un dépôt distant (gitlab)

le dépôt local n'a que 2 branches :

1. *main*[1] : la branche principale dans laquelle on travaille ;
2. *origin/main*[1] : la branche distante,
   copiée localement lors des `git pull`.[1]

Localement, on reste toujours dans la branche *main*.[2]

---

1. Plus exactement, lors des `git fetch` qui sont effectués lors des `git pull`.
2. Encore souvent nommée *master*, bien que ce terme jugé esclavagiste cède désormais la place au terme, plus neutre, *main*.

En pratique

Dans ce cours, on se concentre sur une situation simple :

un dépôt local & un dépôt distant (gitlab)

le dépôt local n'a que 2 branches :

1. *main*[1] : la branche principale dans laquelle on travaille ;
2. *origin/main*[1] : la branche distante,
   copiée localement lors des `git pull`.[1]

Localement, on reste toujours dans la branche *main*.[2]

Pas de `git merge`, mais des `git pull`

---

1. Plus exactement, lors des `git fetch` qui sont effectués lors des `git pull`.
2. Encore souvent nommée *master*, bien que ce terme jugé esclavagiste cède désormais la place au terme, plus neutre, *main*.

En pratique

Dans ce cours, on se concentre sur une situation simple :

un dépôt local & un dépôt distant (gitlab)

le dépôt local n'a que 2 branches :

1. *main*[1] : la branche principale dans laquelle on travaille ;
2. *origin/main*[1] : la branche distante,
   copiée localement lors des `git pull`.[1]

Localement, on reste toujours dans la branche *main*.[2]

Pas de `git merge`, mais des `git pull`
qui impliquent des fusions, avec des conflits possibles.

---

1. Plus exactement, lors des `git fetch` qui sont effectués lors des `git pull`.
2. Encore souvent nommée *master*, bien que ce terme jugé esclavagiste cède désormais la place au terme, plus neutre, *main*.

Aller plus loin sur les branches

git permet en fait d'avoir plusieurs branches locales :

on change de branche avec `git switch`

on les gère avec `git branch`

Aller plus loin sur les branches

git permet en fait d'avoir plusieurs branches locales :

on change de branche avec `git switch`

on les gère avec `git branch`

intérêt : plusieurs variantes d'un même projet, *e.g.* :

- une branche principale propre & fonctionnelle
- une branche pour chaque extension en cours d'élaboration
- une branche pour les corrections urgentes de bugs
- *etc. . .*

Conclusion

Questions

Posez vos questions (sur l'ensemble du module).

Aller plus loin avec git

git est un outil très riche.

3. on a vu `git commit --amend` qui modifie le dernier `commit` en TP !

Aller plus loin avec git

git est un outil très riche.

Nous n'avons (presque) pas vu :

3. on a vu `git commit --amend` qui modifie le dernier `commit` en TP !

Aller plus loin avec git

git est un outil très riche.

Nous n'avons (presque) pas vu :

- la gestion de branches multiples `git switch`/`git branch` (mentionnée à la *slide* précédente)

3. on a vu `git commit --amend` qui modifie le dernier `commit` en TP !

Aller plus loin avec git

git est un outil très riche.

Nous n'avons (presque) pas vu :

- la gestion de branches multiples `git switch`/`git branch`
  (mentionnée à la *slide* précédente)
- les retours dans le passé `git checkout`/`git restore`
  (voir l'état d'un fichier dans une version antérieure)

3. on a vu `git commit --amend` qui modifie le dernier `commit` en TP !

Aller plus loin avec git

git est un outil très riche.

Nous n'avons (presque) pas vu :

- la gestion de branches multiples `git switch`/`git branch`
  (mentionnée à la *slide* précédente)
- les retours dans le passé `git checkout`/`git restore`
  (voir l'état d'un fichier dans une version antérieure)
- les modifications du passé `git rebase`/`git commit --amend`[3]
  (modifier/corriger/oublier des versions passées)

3. on a vu `git commit --amend` qui modifie le dernier `commit` en TP !

Aller plus loin avec git

git est un outil très riche.

Nous n'avons (presque) pas vu :

- la gestion de branches multiples — `git switch`/`git branch` (mentionnée à la *slide* précédente)
- les retours dans le passé — `git checkout`/`git restore` (voir l'état d'un fichier dans une version antérieure)
- les modifications du passé — `git rebase`/`git commit --amend`[3] (modifier/corriger/oublier des versions passées)
- et bien d'autre chose — `git blame`/`git bisect` `git revert`/`git tag` *etc...*

3. on a vu `git commit --amend` qui modifie le dernier `commit` en TP !

Aller plus loin avec git

git est un outil très riche.

Nous n'avons (presque) pas vu :

- la gestion de branches multiples — `git switch`/`git branch` (mentionnée à la *slide* précédente)
- les retours dans le passé — `git checkout`/`git restore` (voir l'état d'un fichier dans une version antérieure)
- les modifications du passé — `git rebase`/`git commit --amend`[3] (modifier/corriger/oublier des versions passées)
- et bien d'autre chose — `git blame`/`git bisect` `git revert`/`git tag` *etc...*

Vous pouvez les découvrir par vous même

par exemple en jouant à ohmygit
mais surtout en pratiquant.

3. on a vu `git commit --amend` qui modifie le dernier `commit` en TP !

Aller plus loin avec LaTeX

LaTeX est un outil très riche.

Aller plus loin avec LaTeX

LaTeX est un outil très riche.

Nous n'avons pas vu :

- l'écriture avancée de macros, de paquets, d'environnements ;
- la gestion des dimensions, des compteurs, des boîtes (*box*) ;
- les classes orientées présentation (`beamer`)
  ou autres (*e.g.*, partitions musicales) ;
- les outils de dessins (*e.g.*, `tikz`, `pgf`, `pstricks`, ...).

Aller plus loin avec LaTeX

LaTeX est un outil très riche.

Nous n'avons pas vu :

- l'écriture avancée de macros, de paquets, d'environnements ;
- la gestion des dimensions, des compteurs, des boîtes (*box*) ;
- les classes orientées présentation (`beamer`)
  ou autres (*e.g.*, partitions musicales) ;
- les outils de dessins (*e.g.*, `tikz`, `pgf`, `pstricks`, ...).

Vous pourrez découvrir tout cela par vous même
en pratiquant.

Fin du module

- **dernier TP :**

  LaTeX (insertion de code et de figure) & git (fusions avec conflits)

Fin du module

- **dernier TP :**
  LaTeX (insertion de code et de figure) & git (fusions avec conflits)
- **TP noté** (semaine du 51, du 18 au 22 décembre) :
  tout est au programme (cours-td comme tp)
  les documents (supports, sujets, notes, et même web) seront autorisés

Fin du module

- **dernier TP :**
  LaTeX (insertion de code et de figure) & git (fusions avec conflits)
- **TP noté** (semaine du 51, du 18 au 22 décembre) :
  tout est au programme (cours-td comme tp)
  les documents (supports, sujets, notes, et même web) seront autorisés
- **Projet** (lancé prochainement) à réaliser :
  - par équipe de 2 (binôme) ou 3 (trinôme) d'un même groupe
  - sur git et en LaTeX
  - à propos d'un sujet scientifique (mathématique et/ou informatique)

Fin du module

- **dernier TP :**
  LaTeX (insertion de code et de figure) & git (fusions avec conflits)
- **TP noté** (semaine du 51, du 18 au 22 décembre) :
  tout est au programme (cours-td comme tp)
  les documents (supports, sujets, notes, et même web) seront autorisés
- **Projet** (lancé prochainement) à réaliser :
  - par équipe de 2 (binôme) ou 3 (trinôme) d'un même groupe
  - sur git et en LaTeX
  - à propos d'un sujet scientifique (mathématique et/ou informatique)

Suite du module

- module projet Python au S2 (gestion avec git, rapport en LaTeX) ;

Fin du module

- **dernier TP :**
  LaTeX (insertion de code et de figure) & git (fusions avec conflits)
- **TP noté** (semaine du 51, du 18 au 22 décembre) :
  tout est au programme (cours-td comme tp)
  les documents (supports, sujets, notes, et même web) seront autorisés
- **Projet** (lancé prochainement) à réaliser :
  - par équipe de 2 (binôme) ou 3 (trinôme) d'un même groupe
  - sur git et en LaTeX
  - à propos d'un sujet scientifique (mathématique et/ou informatique)

Suite du module

- module projet Python au S2 (gestion avec git, rapport en LaTeX) ;
- toute votre vie : git-ez vos projets, LaTeX-ez vos rapports.

Fin du module

- **dernier TP :**
  LaTeX (insertion de code et de figure) & git (fusions avec conflits)
- **TP noté** (semaine du 51, du 18 au 22 décembre) :
  tout est au programme (cours-td comme tp)
  les documents (supports, sujets, notes, et même web) seront autorisés
- **Projet** (lancé prochainement) à réaliser :
  - par équipe de 2 (binôme) ou 3 (trinôme) d'un même groupe
  - sur git et en LaTeX
  - à propos d'un sujet scientifique (mathématique et/ou informatique)

Suite du module

- module projet Python au S2 (gestion avec git, rapport en LaTeX) ;
- toute votre vie : git-ez vos projets, LaTeX-ez vos rapports.

Défi

Trouvez et exécutez des commandes BASH et git
vous permettant d'obtenir le statut de l'exemple donné en slide 22.