



Outils numériques et programmation



Martin VÉROT

En tant que scientifiques, le rôle des outils numériques jouera un rôle prépondérant dans votre carrière. En effet, à l'aide de ces outils, il sera possible de :

- traiter et automatiser un grand nombre de tâches. Cela joue un rôle essentiel dans la productivité et la capacité à produire des données exploitables.
- produire des données « digérées » à partir de données brutes (graphiques, modèles, exploitations, etc)
- créer des données numériques via des méthodes de simulation.

Ce cours est un cours introductif qui vise à vous initier à ces différents aspects. Il ne s'agit pas d'un cours de programmation pure :

- nous ne ferons pas d'algorithmique pour optimiser des processus;
- nous n'aborderont des problématiques plus globales de *design pattern*, test unitaire, etc.
- la conception d'interface graphique ne sera pas évoqué
- Ici, l'accent sera mis essentiellement sur la programmation fonctionnelle et pas sur la programmation orientée objet (bien qu'en python, tout soit objet).

Il ne s'agit pas non plus d'un cours sur les bases de données qui ne seront pas du tout abordées, ni de programmation web.

Le choix du langage et des outils s'est tourné du côté des systèmes UNIX (Linux et assimilé) qui forment est un environnement de travail fréquent dans le monde scientifique, gratuit, documenté et très riche. Pour la programmation, c'est le langage Python qui a été choisi car c'est un des langages de programmation couramment utilisé dans le monde scientifique, de nombreuses librairies y sont disponibles, la communauté est active et mature. S'il a ses limites, il a également de nombreuses qualités, en particulier le fait que ce soit un langage de haut niveau, très expressif et qui limite les besoins d'aller « mettre les mains dans le cambouis ». Cela correspond à la volonté du cours dont le mantra pourrait être : « *get things done* ».

La compréhension et l'appétence pour les choses numériques étant variables, une partie de ce qui sera écrit ici sera peut-être du charabia pour certains, des évidences pour d'autres. Mais le but est de forger une culture numérique minimale qui permette à tout le monde de progresser pour éviter de finir à passer des journées entières à ouvrir des fichiers, lire une donnée, la mettre dans un fichier excel pour ensuite la tracer et la modéliser à la main.^a

L'activité de programmation est ingrate, surtout au début : cela demande du temps, de la patience, un peu d'abnégation pour faire des choses qui sont basiques. Mais la pratique aide : les heures passées à travailler cette année peuvent servir de base pour gagner une productivité et un confort de travail énorme sur une carrière complète. Un bon programmeur sait exploiter sa fainéantise au maximum pour laisser son programme faire le travail à sa place. Mais la fainéantise a un coût : il faut passer le temps nécessaire à construire un programme pour qu'il corresponde à ses besoins spécifiques. Le retour sur investissement sera d'autant plus important que vous aurez pris l'habitude de programmer. L'expérience et l'habitude permettent de ne pas faire les mêmes erreurs, et il est fréquent de ré-utiliser des morceaux de code ou idées déjà mises en œuvre ailleurs. Le maître mot est donc de programmer pour faire ce dont vous avez besoin, quel que soit le sujet, le but, le moyen.

a. Toute ressemblance avec des situations rencontrées en thèse par l'auteur serait purement fortuite ... ou pas.

J'espère que vous arriverez à prendre goût à cette activité pour réussir à faire avancer vos programmes, idées, concepts et votre science là où elle n'aurait pu aller sans l'aide d'outils numériques.

Ce cours est le successeur de celui donné par Christophe Winisdoerffer. Il s'en inspire donc largement. Je tiens d'ailleurs à le remercier chaleureusement pour m'avoir permis à l'époque de progresser moi-même et pour les nombreuses discussions que nous avons eu sur le sujet. Plusieurs exercices proposés pour ce cours sont directement tirés de ceux proposés les années précédentes. Les différents intervenants du module ont également contribué à l'enrichir. Enfin, je tiens également à remercier chaleureusement le CBP pour nous accueillir en ses murs, et plus particulièrement Emmanuel Quemener qui se charge d'en assurer le bon fonctionnement en toutes circonstances.


Table des matières

1	La philosophie du cours	7
I	Quelques notions sur le système et la ligne de commande	9
II	Pour aller plus loin	11
2	Les expressions régulières	13
3	Tableaux non numériques	15
4	Les gestionnaires de version : git/github	17
4.1	Vocabulaire	18
4.2	Principe général des actions sur git	19
4.3	Opérations au niveau local	20
4.3.1	Installation	20
4.3.2	Initialisation et création d'un dépôt Git	20
4.3.3	Ajout/suppression/déplacement de fichiers dans la <i>staging area</i>	20
4.3.4	Mise à jour effective du dépôt : commit	21
4.3.5	Branch/Merge : créer des variations du dépôt	22
4.3.6	Se déplacer dans l'historique : reset	24
4.4	Interaction avec un serveur distant	24
4.4.1	Création du dépôt sur GitHub	24
4.4.2	Envoyer et recevoir des données : Push/Pull	25
4.4.2.1	Pull puis Push initial	26
4.4.2.2	Pull ou push en temps normal	27
4.5	Ce qu'il faut retenir	28
5	Calcul symbolique	29
6	Jupyter	31
7	La programmation orientée objet	33
8	Interaction avec Excel	35
9	Deep learning	37

Chapitre 1






La philosophie du cours

Vous aurez différentes ressources mises à disposition :

1. ce *polycopié* pour lire des concepts, idées méthodes qui ne peuvent pas forcément être intuités ;
2. des *cahiers Jupyter* pour que vous puissiez voir des portions de code en action, avec des exemples, erreurs, choses à compléter. Le tout pour avoir un retour rapide et un cycle d'apprentissage court. Les cahiers jupyter sont accessibles directement en cliquant sur les icônes . Sinon, ils sont tous disponibles [sur github](#) avec 'ch' suivi du numéro de chapitre correspondant dans le polycopié comme préfixe.
3. des *exercices plus complets* qui correspondent à des tâches complexes qui demandent de combiner des idées, fonctions, portions de code. Ils sont tous disponibles [sur github](#) avec le préfixe 'exo-'.

Les deux premiers types de ressources constitueront la partie de travail individuel à fournir. Les exercices complets seront pour leur part ce sur quoi nous nous concentrerons lors des séances. Pour que ces heures soient productives, il faudra absolument avoir travaillé sur les ressources précédentes.

Dans le cours, certains paragraphes seront étiquetés avec les symboles suivants :

-  pour les notions qui ne demandent pas d'effort particulier.
-  pour les notions un peu plus complexes qui nécessitent en général au moins un exemple pour comprendre
-  pour les notions plus complexes qui demandent de s'appropriier la notion. En général, le concept sera maîtrisé avec l'écriture d'un script.
-  Pour indiquer les petites astuces ou points de détail utiles.
-  Pour indiquer de la documentation spécifique.

Première partie

Quelques notions sur le système et la ligne de commande

Deuxième partie
Pour aller plus loin

Chapitre 2

Les expressions régulières

re

Chapitre 3

Tableaux non numériques

pandas

Chapitre 4

Les gestionnaires de version : git/github

Une des préoccupations courantes pour la programmation est d'avoir un suivi de version. Cela est indispensable pour pouvoir gérer l'introduction de nouvelles fonctionnalités, la correction de bugs, la prise en compte de remarque, etc. De manière analogue, le suivi de version est également un outil puissant pour le travail collaboratif sur des documents où il y a un grand nombre d'intervenant.

Il existe différentes combinaisons de logiciels et services. Le choix est ici de présenter la combinaison Git/Github. Elle a différents avantages et inconvénients mais correspond à une combinaison répandue. Dans les gestionnaires de version alternatifs, il existe également SVN (Subversion), tout aussi puissant, mais dont l'utilisation est moins courante.

Il est important d'être conscient que cette combinaison correspond à 2 outils distincts.

- Git est le gestionnaire de version au niveau local (client). C'est lui qui va gérer les opérations de base du suivi de version. C'est un logiciel libre, gratuit et open source.
- Github est le gestionnaire pour l'hébergement, côté serveur. Il est basé sur Git pour la gestion de version. La plateforme est utilisable gratuitement mais pas open-source. Une des pré-occupations actuelles majeures est liée à l'utilisation des données déposées sur la plateforme qui est une propriété de Microsoft. Il est manifeste que les codes déposés sur la plateforme sont ré-utilisés par les IA et donc exploitées pour des finalités avec des débouchés commerciaux. GitLab fait partie des alternatives courante (il existe encore une variété d'autres logiciels, voir figure 4.1).

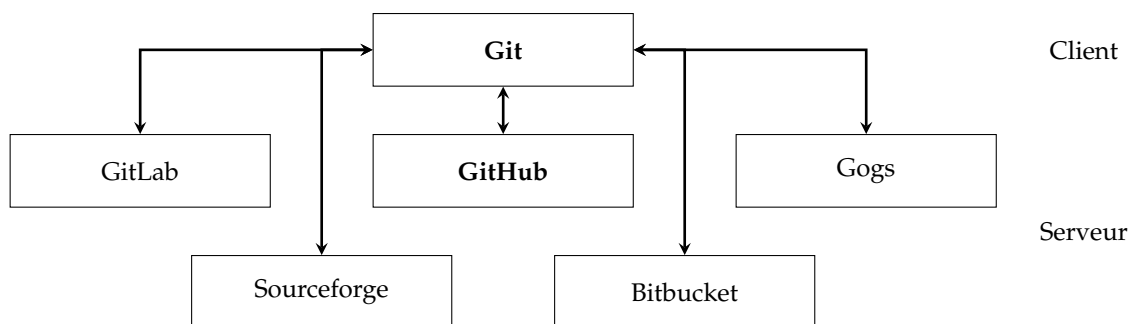


FIGURE 4.1 – Schéma de principe de la solution Git/GitHub. Quelques solutions alternatives à Github pour le côté serveur sont également indiquées.

4.1 Vocabulaire

Le principe général des gestionnaires de version est toujours similaire mais fait appel à un vocabulaire spécifique qui est parfois un peu abscon.

Appellation	Schéma	Commentaire
dépôt/ repo/ repository push	<div> <div> <div>fichier 1</div> <div>fichier 2</div> <div>fichier 3</div> </div> <div> <div>Dépôt local</div> <div>envoi</div> <div>Serveur distant</div> </div> </div>	<p>Ensemble de fichiers correspondant à un projet, sera géré avec Git et synchronisé sur GitHub</p> <p>Envoi du code de votre dépôt, le plus souvent vers la partie serveur</p>
pull	<div> <div>Dépôt local</div> <div>téléchargement</div> <div>Serveur distant</div> </div>	<p>Téléchargement du code, le plus souvent depuis la partie serveur</p>
commit	<div> <div>version XX</div> <div></div> <div>version XX+1</div> </div>	<p>Mise à jour du dépôt, en général en local dans un premier temps</p>
rollback	<div> <div>version XX</div> <div></div> <div>version XX-n</div> </div>	<p>Remise en l'état à une version antérieure</p>
branch	<div> <div>Dépôt principal</div> <div></div> <div>Dépôt principal</div> <div>Copie "temporaire" de dépôt</div> </div>	<p>Création d'une version parallèle, en général pour implémenter une petite fonctionnalité sans affecter le développement ou les évolutions globales du projet</p>
merge	<div> <div>Dépôt principal</div> <div>Copie "temporaire" de dépôt</div> <div>Dépôt principal</div> <div>Dépôt principal</div> </div>	<p>Fusion d'une branche avec le projet principal ou un autre branche</p>
fork	<div> <div>Dépôt principal</div> <div></div> <div>Dépôt principal</div> <div>Nouveau dépôt par copie</div> </div>	<p>Opération de clonage d'un dépôt, en général pour ajouter une fonctionnalité de manière indépendante du projet initial</p>

TABLEAU 4.1 – Quelques termes associés au gestionnaire de version git.

4.2 Principe général des actions sur git

Git place/catégorise les différents fichiers dans différents endroits en fonction de leur statut :

- **Les fichiers non suivis (*Untracked files*)** : ce sont les fichiers qui sont sur votre disque mais qui ne font pas partie du dépôt.
- **Les fichiers modifiés (*Modified files*)** : ce sont les fichiers qui ont été changés depuis leur dernier ajout au dépôt mais n'ont pas été mis à jour dans le dépôt.
- **Les fichiers en phase de « Staging »** : on les a placés dans une « zone tampon » et ils sont prêts à être mis à jour avec une action appelée `commit`.
- Les fichiers dans le dépôt qui sont à jour.

La synchronisation entre les fichiers en local et le dépôt se fait donc en deux phases (figure 4.2) :

- Le staging où on va mettre « en attente » les fichiers qui doivent être modifiés avant de faire une synchronisation ;
- Le commit où on va synchroniser les fichiers placés dans la zone de staging pour mettre à jour le dépôt à proprement parler.

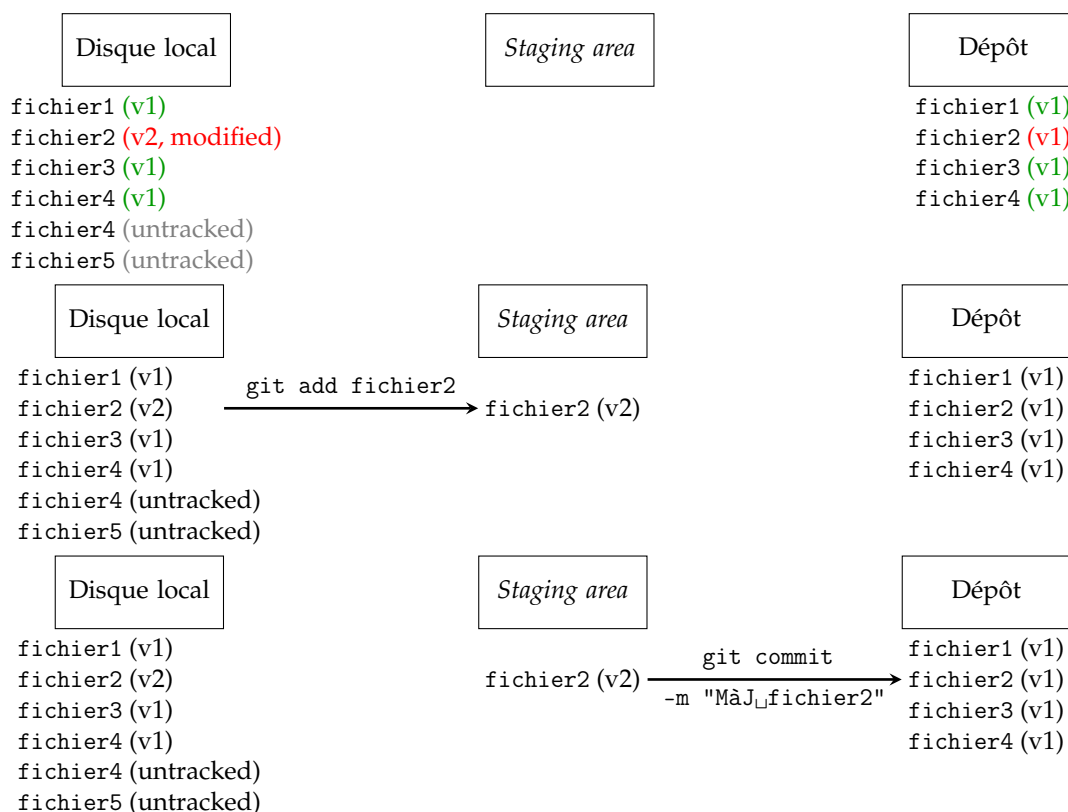


FIGURE 4.2 – Les différentes zones et un exemple de situation courante. Les fichiers 4 et 5 ne font pas partie du dépôt, les fichiers 1, 3 et 4 sont à jour mais le fichier 2 n'est pas à jour dans le dépôt. La synchronisation du dépôt va se faire en deux temps : une phase de staging et une phase de commit.

4.3 Opérations au niveau local

Nous allons ici donner les commandes à taper en ligne de commande. Il existe des logiciels visuels fournis de base comme Git-Gui ou Gitk (téléchargeables depuis le site de Git ou les dépôts). Il existe également une myriade d'autres logiciels pour rendre les opérations plus visuelles, mais une partie non négligeable de ceux-ci sont payants, propriétaires et pas forcément multi-plateforme.

4.3.1 Installation

Il faut aller sur le site de [git](#) pour télécharger la version correspond à votre système d'exploitation. Ensuite, il est possible de définir votre nom d'utilisateur et identifiant par défaut pour vos projets avec les commandes suivantes :

```
git config --global user.name "Nom_Complet"
git config --global user.email "votreemail@ens-lyon.fr"
git config --global init.defaultBranch master
```

Pour définir les valeurs pour chaque dépôt, il suffit d'utiliser l'option `--local` au lieu de `--global`.

Il est possible de vérifier les propriétés globales avec :

```
git config -l
```

4.3.2 Initialisation et création d'un dépôt Git

Pour créer un dépôt, il faut simplement se placer dans le dossier contenant le projet et taper la commande

```
git init
```

Cette commande va initialiser la création d'un dépôt. Lors de cette étape, il y a création d'un dossier caché « `.git` » qui va contenir les données sur le dépôt et son historique.

💡 Initialement, le dépôt créé est totalement vide et ne contient aucun des fichiers du dossier dans lequel vous avez placé la commande `git init`.

4.3.3 Ajout/suppression/déplacement de fichiers dans la *staging area*

Les commandes suivantes ne seront effectives sur le dépôt qu'après avoir effectué un `commit`.

Ajout de fichiers `git add` Pour ajouter des fichiers, il suffit d'utiliser la commande

```
git add fichier1.txt autre fichier.dat *.py
```

suivi d'un ou plusieurs noms de fichiers, les wildcard « `*` » sont également autorisés.

💡💡💡 Attention, dans la plupart des cas, **vous ne voulez PAS tout ajouter à votre projet, en particulier les fichiers contenant des mots de passe ou des identifiants de connexion** (aussi bien à Github qu'à des serveurs). Par conséquent, la commande « `git add *` » est à proscrire à moins d'ensuite supprimer les fichiers contenant lesdits mots de passe. Mais

même comme cela, il est courant d'avoir un sous-ensemble de fichiers qu'il n'est pas indispensable de partager (copies temporaires, fichiers de sauvegardes d'éditeurs, fichiers logs, etc). Pour cette raison, *il est toujours fortement conseillé de faire un dossier séparé dans lequel tous les mots de passe sont stockés*, puis de les utiliser en faisant des `import`, `include` ou assimilé en fonction du langage de programmation.

💡 Si vous avez ajouté par mégarde un fichier, il est possible de le supprimer de la *staging area* avec la commande `git reset file` qui agit dans ce cas là comme l'opposé de la commande `git add file`.

Suppression de fichiers `git rm --cached` Pour supprimer des fichiers, il existe plusieurs alternatives aux conséquences différentes :

- `rm file` qui va supprimer le fichier en local, *mais pas du dépôt*.
- `git rm file` qui va supprimer le fichier du dépôt ET en local. **Attention : cette commande ne permet PAS de supprimer un fichier uniquement du dépôt.**
- `git rm --cached file` qui permet de supprimer le fichier *uniquement du dépôt* mais de le conserver en local.

Il est tout de même possible de rattraper l'utilisation malencontreuse de `git rm` au lieu de `git rm --cached` (section 4.3.6). Le plus simple reste quand même d'être bien conscient des différences entre les deux commandes.

Déplacement de fichiers `git mv` Il suffit d'utiliser la commande `git mv` de manière analogue à la commande `mv` pour agir simultanément sur le dépôt et en local. Comme pour la commande `rm`, l'oubli du préfixe `git` n'agira qu'en local sans agir sur le dépôt.

4.3.4 Mise à jour effective du dépôt : commit

Statut des fichiers `git status` Jusqu'à présent, nous avons ajouté des fichiers en *staging area* (figure 4.2), mais nous n'avons pas encore de moyens pour contrôler le statut des fichiers. Cela peut se faire avec la commande

```
git status
```

Cette commande retourne 3 blocs de fichiers :

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   cours_python.pdf

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   exo/exo2a-algebre.ipynb

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    cours_python-agreg.aux
    cours_python-agreg.bbl
    cours_python-agreg.blg
    [...]
```

```
cours_python-agreg.toc
```

Elle retourne trois catégories de fichiers :

- Les fichiers ajoutés mais mis en *staging* dans la section `Changes to be committed`.
- Les fichiers qui ont été modifiés depuis le dernier commit mais qui ne sont pas encore synchronisés avec le dépôt ni dans la *staging area* dans la section `Changes not staged for commit`.
- Les fichiers non suivis dans la section `Untracked files`

L'option `git status -uno` permet de ne pas lister les fichiers non suivis (*untracked*).

Liste des fichiers dans le dépôt `git ls-files` Par défaut, la commande `git status` ne liste pas les fichiers à jour dans le dépôt. La commande

```
git ls-files
```

Permet de lister tous les fichiers présents dans le dépôt.

Effectuer un commit `git commit` Le *commit* va permettre de synchroniser à un instant donné les fichiers du dépôt avec ceux mis en *staging area*.


Il est **impératif** de fournir un message pour expliciter le contenu du commit. Sans l'option `-m`, la commande `git commit` ouvre un éditeur en ligne de commande pour pouvoir rédiger un message plus long et complet sur le commit. Sinon avec l'option `-m`, on peut indiquer fournir un court message pour indiquer des évolutions de version, de contenu, le fait qu'il s'agisse de modifications mineures, etc. Sans message, le commit est automatiquement refusé.

```
git commit
git commit -m "Texte_explicitant_les_modifications"
```

La commande retourne un message analogue à celui ci-dessous :

```
[master db0ca1d] Mise à jour git/github
2 files changed, 2 insertions(+)
create mode 100644 testbis.md
```

Sur la première ligne, la branche est indiquée (voir section 4.3.5), ici `master`, puis les caractères de la fin, `db0ca1d`, donnent les derniers caractères de l'identifiant du commit. Puis les modifications principales sont indiquées (modification de fichier, ajout, etc).

 Lorsqu'il y a eu uniquement des modifications de fichiers et que l'on veut toutes les inclure au dépôt, cela pourrait être fastidieux de le faire à la main. L'ajout de l'option `-a` permet d'automatiser la tâche en faisant directement l'ajout des fichiers modifiés et le commit. Les deux commandes ci-dessous sont équivalentes.

```
git commit -a -m "Mise_à_jour_de_tous_les_fichiers"
git commit -am "Mise_à_jour_de_tous_les_fichiers"
```

4.3.5 Branch/Merge : créer des variations du dépôt

Par la suite, pour les différents schémas, on va symboliser un commit par un point. Une branche est définie comme une ligne continue de commit successifs. La flèche sera une flèche du temps pour indiquer la succession des commits.



FIGURE 4.3 – Illustration de la notion de branche. Il s'agit d'une succession de commit effectués les uns après les autres. Ici, pour la branche "master", on a effectué 5 commits successifs.

La branche principale est généralement appelée *main* ou *master*. Elle est usuellement porteuse de toutes les modifications importantes de code. Dans certains cas, il peut être intéressant de créer des variations du dépôt initial pour pouvoir faire des modifications temporaires et personnelles sur la branche principale : par exemple introduire une nouvelle fonction non encore déboguée dans un code, introduire des modifications dans un document LaTeX alors que nous ne sommes pas responsable de l'article, etc. Dans ce cas, plutôt que de continuer la branche principale, il est possible de créer une branche secondaire qui permettra de plus facilement revenir en arrière si jamais l'évolution s'avère être infructueuse. Cela permettra également de développer une petite évolution d'un dépôt plus important partagé avec des collègues sans affecter la branche principale. L'opération de création correspond à un *branch* tandis que l'opération inverse de fusion de deux branches s'appelle un *merge*.

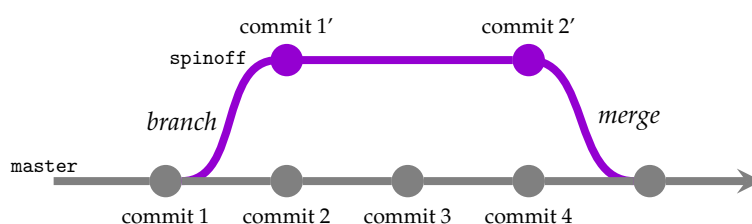


FIGURE 4.4 – Illustration de la notion de *branch* et de *merge* avec la convention du dessin précédent. On crée une branche *spinoff* dans laquelle on peut faire évoluer librement les fichiers du dépôt, puis il est possible de fusionner cette branche avec la branche *master* qui pourra elle aussi avoir évolué indépendamment.

Création d'une branche Pour la création d'une branche, il existe différentes commandes pour créer une nouvelle branche où *****newbranch***** doit être remplacé par le nom de la nouvelle branche :

```
git switch -c **newbranch**
git branch **newbranch**
```

Avec `switch -c`, on crée la branche et on s'y déplace automatiquement, tandis qu'avec `branch`, on crée la branche mais sans s'y déplacer.

La création d'une branche va copier tous les fichiers du dépôt dans leur version correspondant au dernier commit de la branche depuis laquelle on a effectué la commande.

Se déplacer entre les branches La commande `git branch` permet de lister toutes les branches existantes du dépôt pour ensuite pouvoir se déplacer entre elles. Elle indique également la branche sur laquelle nous sommes. Pour se déplacer il suffit ensuite d'utiliser la commande :

```
git checkout **otherbranch**
```

pour se déplacer sur la branche **otherbranch**. Il est alors possible de faire évoluer indépendamment chacune des branches *pour les fichiers faisant partie du dépôt*.

Fusionner des branches Pour fusionner des branches il faut utiliser la commande :

```
git merge **branche1** **branche2**
```

où **branche1** et **branche2** est le nom des deux branches à fusionner.

Bien que la commande soit brève et efficace, en pratique, il y a couramment des conflits à surmonter pour gérer la fusion des branches. Il y a plusieurs cas typiques :

- Le même fichier a été créé indépendamment sur les deux branches. Dans ce cas, il n’y a aucun moyen de déterminer automatiquement quel est le bon fichier à conserver.
- Le même fichier a été modifié sur chaque branche.

FIGURE 4.5 – •

```
git branch
git merge
git checkout
git log --merge
git diff
```

4.3.6 Se déplacer dans l’historique : reset

Maintenant que nous avons pu faire les différentes opérations de base en local, nous allons pouvoir voir l’intérêt d’un gestionnaire de version. En particulier comment faire pour pouvoir revenir en arrière, gérer son historique, etc.

```
git log
git reset --hard --soft --mixed
```

[git reset](#)

4.4 Interaction avec un serveur distant

4.4.1 Création du dépôt sur GitHub

Jusqu’à présent, toutes les actions ont été effectuées en local. Si cela peut être suffisant pour de la gestion de version sur des fichiers personnels, une des forces du couple git/github est de pouvoir ensuite partager ses fichiers avec une communauté plus large et éventuellement pouvoir faire du travail collaboratif sur un même projet.

Pour cela, il faut faire appel à une plateforme pour héberger les dépôts (voir figure 4.1). Comme annoncé en introduction du chapitre, nous allons nous concentrer sur l’utilisation de la plateforme github. Pour cela, il faut commencer par se créer un compte sur le site [github](#).

Une fois votre compte créé, il faut aller en haut à droite (figure 4.6) cliquer sur l’icône de votre compte et aller dans l’onglet « Repositories », puis cliquer sur « New ».

Il faut ensuite :

- définir un nom de dépôt,

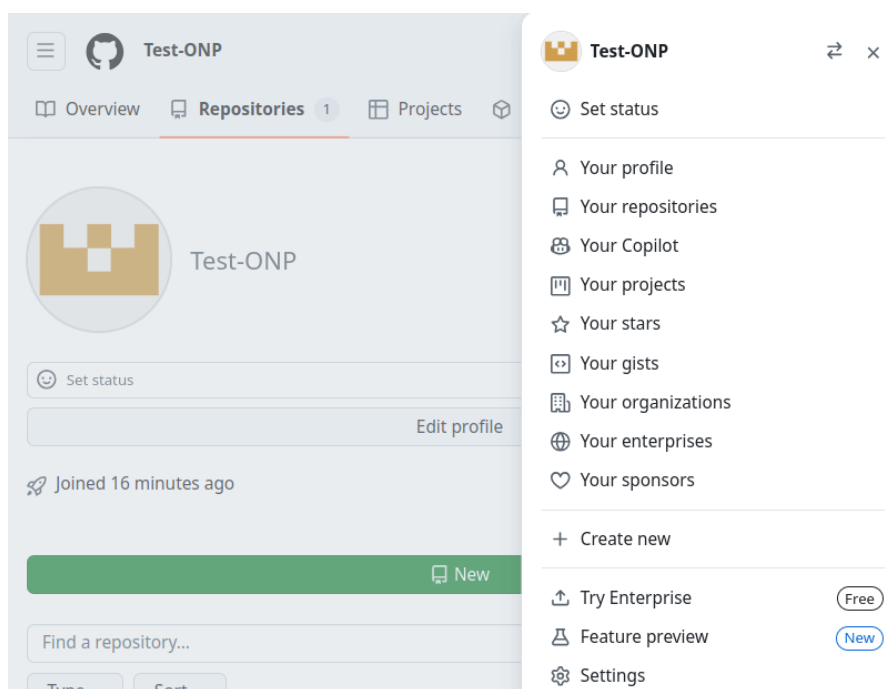


FIGURE 4.6 – Création d'un dépôt sur GitHub.

- une courte description,
- choisir si le dépôt sera public ou privé
- ajouter un fichier README dont le contenu apparaîtra par défaut en tant que première page du repository.
- créer un fichier `.gitignore` qui indique les fichiers à ne pas ajouter par défaut avec la commande `git add`
- fournir une licence par défaut à votre projet. **C'est extrêmement important pour protéger votre travail.**

Une fois le dépôt créé, il faut configurer les accès. Pour cela, il faut toujours cliquer sur son icône (figure 4.6), « Settings » puis « Developer Settings », « Personal Access Tokens », « Tokens (classic) », « Generate New Token ». Y remplir les champs nécessaire, mettre la date d'expiration à « No expiration » (pour qu'elle soit permanente, si c'est ce que vous souhaitez). Choisir les droits qui vous conviennent (normalement, il faut à minima cocher la section « repo » pour pouvoir faire des modifications sur vos propres dépôts). Vous aurez alors accès à un identifiant, qui commence probablement par « ghp_ ». **Comme indiqué, notez le précieusement et surtout gardez-le secret!**

💡 Github analyse les fichiers déposés sur les dépôts pour vérifier que vous ne partagez pas par inadvertance votre « Personal Access Token » en publiant un fichier qui le contient. Si jamais c'est le cas, alors il révoque automatiquement le token pour empêcher tout piratage du compte ou du dépôt. Il faut tout de même rester extrêmement vigilant au fait de ne pas divulguer par inadvertance vos autres mots de passe (voir l'avertissement donné section 4.3.3).

4.4.2 Envoyer et recevoir des données : Push/Pull

Dans le jargon git/github, un push correspond à l'envoi de données locales vers un serveur distant et le pull est la récupération du dépôt sur un serveur distant en local.

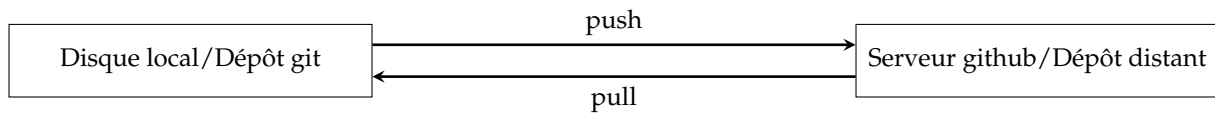


FIGURE 4.7 – Notion de pull (synchronisation local -> distant) et de push (synchronisation distant -> local).

4.4.2.1 Pull puis Push initial

La première synchronisation est la plus compliquée, on va commencer par importer le contenu depuis le dépôt distant, cette opération s'appelle un « *pull* ». Puis on va ensuite envoyer le contenu local vers le dépôt distant avec un « *push* ».

Une fois le dépôt créé, il faut synchroniser le contenu distant et local. Il peut y avoir des soucis divers et variés à ce moment là :

- par défaut github utilise comme branche principale "main" tandis que git utilise plutôt "master". Il est possible de faire en sorte d'avoir le même nom de branche principale (pour faciliter les opérations) :
 - soit via github : Settings (généraux) » Repositories » puis mettre *Repository default branch* à master. Dans ce cas la branche principale sera appelée master des deux côtés.
 - soit côté git :

```
git config --global init.defaultBranch main
```

Dans ce cas la branche principale sera appelée main des deux côtés.

- De plus, il faut commencer par faire une première synchronisation entre la partie distante et la partie locale, mais il y a un conflit initial car les deux dépôts n'ont à priori pas le même contenu. Il y a de nombreux moyens de faire (rebase, merge, synchronisation à la main, ...) Nous allons ici supposer qu'il n'y a pas de conflit important sur le contenu des deux dépôts vu que le dépôt sur github est quasiment vide (peut-être un fichier README.md, un fichier de licence et un fichier .gitignore). On va d'abord commencer par récupérer les données sur le serveur :

```
git push --set-upstream https://**login**@github.com/**login**/**
repository** master
```

Où il faut remplacer les deux ****login**** par votre login github, ****repository**** par le nom de votre repository. On a ici supposé que les branches principales s'appellent toutes les deux master. Il faut ensuite indiquer votre token pour procéder à la synchronisation avec le site.

Il y a alors probablement un message d'erreur qui s'affiche :

```
* branch          master    -> FETCH_HEAD
hint: You have divergent branches and need to specify how to reconcile
      them.
hint: You can do so by running one of the following commands sometime
      before
hint: your next pull:
hint:
hint:  git config pull.rebase false # merge
hint:  git config pull.rebase true  # rebase
```

```

hint: git config pull.ff only    # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a
      default
hint: preference for all repositories. You can also pass --rebase, --no
      -rebase,
hint: or --ff-only on the command line to override the configured
      default per
hint: invocation.

```

Qui est là pour vous indiquer que le contenu des deux dépôts diffère et donc que git n'a pas (encore) procédé à la synchronisation.

Il est possible de *rebase* le dépôt. Cette opération va procéder à une fusion qui n'est pas un *merge* (voir section 4.3.5). La portée finale de cette opération est de faire en sorte de lever les conflits pour ensuite procéder à la synchronisation.

```
git pull --rebase
```

Va alors faire une levée de conflit et vous devriez voir apparaître en local les fichiers qui étaient sur le dépôt distant (s'il y avait des fichiers dessus).

4.4.2.2 Pull ou push en temps normal

Une fois la première synchronisation effectuée, vous pouvez utiliser les transferts dans un sens comme dans l'autre.

```

git push --set-upstream https://**login**@github.com/**login**/**repository
** master
git pull --set-upstream https://**login**@github.com/**login**/**repository
** master

```

Il sera nécessaire de fournir son mot de passe à chaque commande. Il est possible de faire en sorte d'enregistrer le mot de passe avec gcm (git credential manager), mais il n'y a pas de solution qui soit à la fois universelle et sécurisée. Une page de documentation sur le sujet est disponible sur le wiki de git-credential-manager [Credential stores](#).

💡 Il est aussi possible de créer un petit script en bash pour éviter d'avoir à taper l'une ou l'autre des commandes.

4.5 Ce qu'il faut retenir

Commande	Utilité
Opérations de base	
<code>git_init</code>	Créer un dépôt (section 4.3.2)
<code>git_add_file</code>	Mettre les fichiers en <i>staging</i> (section 4.3.3)
<code>git_rm_file</code>	Supprimer les fichiers en local et dans le dépôt (section 4.3.3)
<code>git_rm --cached_file</code>	Supprimer les fichiers du dépôt uniquement (section 4.3.3)
<code>git_mv_file_dest</code>	Déplacer le fichier en local et dans le dépôt
Commit	
<code>git_ls-files</code>	Lister tous les fichiers dans le dépôt (section 4.3.4)
<code>git_status</code>	Voir le statut des différents fichiers (section 4.3.4)
<code>git_commit -m "MàJ"</code>	Effectuer un commit sur le dépôt (section 4.3.4)
<code>git_commit -a -m "MàJ"</code>	Ajouter en staging tous les fichiers modifiés et faire un commit (section 4.3.4)
Branches/Merge	
<code>git_branch</code>	Création d'une nouvelle branche (section 4.3.5)
<code>git_merge</code>	Fusion de deux branches (section 4.3.5)
Dépôt distant	
<code>git_push</code>	Synchroniser en envoyant les données locales vers le serveur distant (section 4.4.2)
<code>git_pull</code>	Synchroniser en recevant les données du serveur distant vers le dépôt local (section 4.4.2)

TABLEAU 4.2 – Commandes de base pour Git.

Chapitre 5

Calcul symbolique

sage

Chapitre 6

Jupyter

Pour avoir des graphiques interactifs : [Interaction sous Jupyter](#).

Chapitre 7

La programmation orientée objet

Chapitre 8

Interaction avec Excel

openpyxl

Chapitre 9

Deep learning

scikit learn