

# III 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'algorithme 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.<sup>a</sup>

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

<b>1 La philosophie du cours</b>	<b>11</b>
<b>I Quelques notions sur le système et la ligne de commande</b>	<b>13</b>
<b>2 L'utilisation du terminal au XXI<sup>e</sup> siècle</b>	<b>15</b>
2.1 Avantages et inconvénients des interfaces graphiques . . . . .	16
2.2 Avantages et inconvénients de la ligne de commande . . . . .	16
2.3 Et au final, on fait quoi ? . . . . .	17
<b>3 Commandes de base</b>	<b>19</b>
3.1 ▲ Trouver les utilitaires en ligne de commande et LA commande maîtresse :	
man . . . . .	19
3.2 Système de fichier et arborescence . . . . .	20
3.2.1 ▲ Mieux connaître l'arborescence : la commande ls . . . . .	20
3.2.2 Les dossiers et fichiers particuliers . . . . .	20
3.2.3 ▲ Se déplacer avec la commande cd . . . . .	21
3.2.4 ▲ Création, modification, suppression de fichiers ou dossiers . . . . .	23
3.2.5 ▲ Trouver des fichiers . . . . .	24
3.2.6 ▲ Droits . . . . .	25
3.2.6.1 Exécuter un programme/fichier . . . . .	25
3.2.6.2 Connaître les droits . . . . .	25
3.2.6.3 Modifier les droits . . . . .	26
3.3 ▲ Aller plus vite dans le terminal . . . . .	26
3.4 ■ Les liens . . . . .	27
3.5 Redirection de la sortie . . . . .	28
3.6 Quelques utilitaires . . . . .	28
3.6.1 Connaître l'espace disque et la taille de certains dossiers . . . . .	28
3.6.2 Les fichiers texte . . . . .	29
3.6.2.1 ▲ Trouver une ligne contenant une chaîne de caractère dans plusieurs fichiers : grep . . . . .	29
3.6.2.2 Voir la fin d'un fichier : tail . . . . .	30
3.6.2.3 Concaténer des fichiers : cat . . . . .	30
3.6.2.4 Comparer des fichiers : diff et wdiff . . . . .	30
3.6.2.5 ■ ♡ Faire des modifications de fichiers (remplacement de chaîne de caractère, etc) : sed et awk . . . . .	30
3.6.3 Les fichiers pdf . . . . .	31
3.6.4 Reconnaissance de caractères : tesseract . . . . .	31
3.6.5 Les images : ImageMagick . . . . .	32

3.6.6	Les vidéos : ffmpeg . . . . .	32
3.6.7	Compression et décompression . . . . .	32
3.7	■ Enchaînement de commande : le pipe . . . . .	33
3.8	■ Les alias et le fichier .bashrc . . . . .	33
3.9	■ Environnement : PATH . . . . .	34
3.10	■ Travail sur des machines distantes . . . . .	35
3.10.1	Connexion sur des machines distantes : ssh . . . . .	35
3.10.2	Au CBP : x2go . . . . .	36
3.10.3	■ Copie d'un fichier à distance : scp . . . . .	36
3.10.4	■ Copie d'un répertoire à distance : rsync . . . . .	37
3.11	Gestion des processus . . . . .	38
3.11.1	Fermer un processus qui buggue ou ralentit le système . . . . .	38
3.11.2	Gérer l'exécution des processus dans le terminal . . . . .	39
3.12	Scripts et personnalisation . . . . .	39
3.13	Éditeurs de texte : IDE versus les deux indémodables vi et emacs . . . . .	40
3.14	Ce qu'il faut retenir . . . . .	41
<b>II</b>	<b>Programmation en python</b>	<b>43</b>
<b>4</b>	<b>Avant de se lancer</b>	<b>45</b>
4.1	Décortiquer un problème avant de commencer . . . . .	45
4.2	▲ Nommage des variables . . . . .	46
4.3	Trouver de l'aide . . . . .	46
4.3.1	Le facile : l'appel à un ami . . . . .	46
4.3.2	Le moins plaisant (mais plus complet) : « RTFM » . . . . .	47
4.3.3	Le pénible : consulter le code source . . . . .	47
4.4	Déboguer un programme . . . . .	47
4.4.1	Compartimenter/factoriser . . . . .	47
4.4.2	Lire les messages d'erreur . . . . .	48
4.4.3	Commenter son code (avec du texte) . . . . .	48
4.4.4	Afficher ses variables et leur type . . . . .	48
4.4.5	Revenir en arrière jusqu'à revenir à un programme fonctionnel . . . . .	48
4.4.6	Faire un appel à l'aide . . . . .	48
4.5	Commenter son code . . . . .	49
4.6	Structure d'un script python . . . . .	50
4.7	Ce qu'il faut retenir . . . . .	50
<b>5</b>	<b>Les bases</b>	<b>53</b>
5.1	Types de base . . . . .	53
5.1.1	Nombres . . . . .	53
5.1.1.1	◆ Représentation flottante, comparaison et précision . . . . .	53
5.1.2	Chaînes de caractère . . . . .	54
5.1.3	Listes . . . . .	55
5.1.4	Tuples . . . . .	55
5.1.5	Dictionnaires . . . . .	55
5.1.6	Fonctions . . . . .	56
5.1.6.0.1	Paramètres, arguments optionnels, args, kwargs . . . . .	56
5.1.6.0.2	Les fonctions anonymes ou fonctions lambda . . . . .	56

5.1.6.0.3	Packing et unpacking . . . . .	57
5.2	◆ Objets mutables et immutables . . . . .	58
5.2.1	◆ Première conséquence : copie d'objets mutables/imbriqués : « <i>shallow copy</i> » versus « <i>deep copy</i> » . . . . .	59
5.2.2	◆ Deuxième conséquence : éléments mutables et fonctions . . . . .	61
5.2.2.1	◆ Modification d'un élément mutable au sein d'une fonction . . . . .	61
5.2.2.2	◆ Élément mutable comme argument optionnel . . . . .	61
5.3	Espaces de nommage . . . . .	61
5.3.1	◆ les différents espaces de nommage . . . . .	61
5.3.2	■ Portée des variables . . . . .	62
5.4	Structures de bases : boucles et conditions . . . . .	62
5.4.1	Boucles . . . . .	63
5.4.1.1	Boucles <i>for</i> . . . . .	63
5.4.1.2	Boucles <i>while</i> . . . . .	63
5.4.2	Conditions <i>if, elif, else</i> . . . . .	63
5.4.2.1	Instructions <i>break, pass et continue</i> . . . . .	64
5.5	Gestion des fichiers . . . . .	64
5.6	Évaluer la performance de son code . . . . .	65
5.7	Ce qu'il faut retenir . . . . .	65
<b>6</b>	<b>Modules, librairies, packages et frameworks</b> . . . . .	<b>67</b>
6.1	Un peu de vocabulaire . . . . .	67
6.2	Au fait, une librairie ça sert à quoi ? . . . . .	67
6.3	Import d'une librairie ou d'un module . . . . .	68
6.3.1	Import spécifique . . . . .	69
6.3.2	Import « à la sauvage » . . . . .	69
6.4	Quelques librairies courantes . . . . .	70
6.5	Ce qu'il faut retenir . . . . .	71
<b>7</b>	<b>La manipulation de tableaux avec Numpy</b> . . . . .	<b>73</b>
7.1	Les <i>ndarray</i> . . . . .	73
7.1.1	Propriétés essentielles . . . . .	73
7.1.2	Fonctions pour la création . . . . .	74
7.2	Sélection d'éléments et <i>slicing</i> . . . . .	75
7.3	Opérations sur les tableaux, <i>broadcasting</i> et notion d'axe . . . . .	76
7.3.1	◆ <i>Broadcasting</i> ⓘ . . . . .	76
7.3.2	◆ Opération sur les tableaux, notion d'axe . . . . .	77
7.4	Points divers . . . . .	77
7.4.1	Dérivation numérique <i>versus</i> dérivation analytique . . . . .	77
7.4.2	Nombres pseudo-aléatoires . . . . .	77
7.5	Ce qu'il faut retenir . . . . .	78
<b>8</b>	<b>Faire des graphiques avec Matplotlib</b> . . . . .	<b>79</b>
8.1	Principes généraux . . . . .	79
8.1.1	Bibliographie . . . . .	80
8.2	Utilisation de matplotlib . . . . .	81

<b>9 Scipy et numpy pour résoudre quelques problèmes numériques courants</b>	<b>87</b>
9.1 Constantes numériques . . . . .	87
9.2 « Philosophie » des fonctions scipy . . . . .	87
9.3 Ajustement de courbe . . . . .	88
9.4 Transformée de Fourier . . . . .	88
9.5 Recherche de zéros . . . . .	89
9.5.1 Fonctions unidimensionnelles . . . . .	89
9.5.2 Fonctions vectorielles à plusieurs variables . . . . .	89
9.6 Minimisation de fonction . . . . .	90
9.7 Intégration . . . . .	90
9.8 Équations différentielles ordinaires . . . . .	91
9.8.1 Équation différentielle scalaire du premier ordre . . . . .	91
9.8.2 Équation différentielle vectorielle du premier ordre . . . . .	91
9.8.3 Équation différentielle du second ordre . . . . .	91
9.9 Fonctions statistiques . . . . .	92
9.10 Arrangement, combinaison . . . . .	92
 <b>III Pour aller plus loin</b>	 <b>93</b>
<b>10 Les expressions régulières</b>	<b>95</b>
10.1 Fonctionnement général des expressions régulières . . . . .	96
10.1.1 Allure usuelle d'une expression régulière . . . . .	96
10.1.1.0.1 Séparateur pour les langages de programmation autres que python . . . . .	96
10.1.2 Le <i>backtracking</i> . . . . .	96
10.2 Classes de caractère . . . . .	97
10.3 Quantificateurs . . . . .	98
10.3.1 Comportement <i>greedy</i> ou non . . . . .	98
10.4 Les parenthèses . . . . .	99
10.4.1 Les parenthèses capturantes . . . . .	99
10.4.2 Les parenthèses « autres » . . . . .	100
10.5 Caractères spéciaux . . . . .	101
10.6 Modificateurs flags . . . . .	101
10.7 Opérateurs . . . . .	101
10.7.1 Trouver un motif . . . . .	101
10.7.2 Rechercher/remplacer un motif . . . . .	102
 <b>11 Tableaux non numériques</b>	 <b>103</b>
 <b>12 Les gestionnaires de version : git/github</b>	 <b>105</b>
12.1 Vocabulaire . . . . .	106
12.2 Principe général des actions sur git . . . . .	107
12.3 Opérations au niveau local . . . . .	108
12.3.1 Installation . . . . .	108
12.3.2 Initialisation et création d'un dépôt Git . . . . .	108
12.3.3 Ajout/suppression/déplacement de fichiers dans la <i>staging area</i> . . . . .	108
12.3.4 Mise à jour effective du dépôt : commit . . . . .	109
12.3.5 Se déplacer dans l'historique : reset, revert, checkout . . . . .	111
12.3.5.1 Notion d'historique . . . . .	111

12.3.5.2 Les trois sections de Git . . . . .	111
12.3.5.2.1 L'espace de travail/ <i>Working Directory</i> . . . . .	111
12.3.5.2.2 La <i>Staging area/Staged snapshot</i> . . . . .	111
12.3.5.2.3 L'historique des commit/ <i>Commit History</i> . . . . .	111
12.3.5.3 Visualiser l'historique . . . . .	112
12.3.5.3.1 Le HEAD, son sens et comment se déplacer dans l'historique . . . . .	112
12.3.5.3.2 Revenir dans l'état d'un commit antérieur avec <code>restore</code> , <code>revert</code> et <code>reset</code> . . . . .	113
12.3.6 Branch/Merge : créer des variations du dépôt . . . . .	114
12.4 Interaction avec un serveur distant . . . . .	117
12.4.1 Création du dépôt sur GitHub . . . . .	117
12.4.2 Envoyer et recevoir des données : Push/Pull . . . . .	118
12.4.2.1 Pull puis Push initial . . . . .	118
12.4.2.2 Pull ou push en temps normal . . . . .	119
12.5 Ce qu'il faut retenir . . . . .	121
<b>13 Calcul symbolique</b>	<b>123</b>
<b>14 Jupyter</b>	<b>125</b>
<b>15 La programmation orientée objet</b>	<b>127</b>
<b>16 Interaction avec Excel</b>	<b>129</b>
<b>17 Deep learning</b>	<b>131</b>



# 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'approprier 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**



# Chapitre 2

## L'utilisation du terminal au XXI<sup>e</sup> siècle

À l'âge des apps, de l'ergonomie, de l'UX design et autres joyeusetés, la confrontation à l'utilisation des lignes de commandes peut sembler totalement désuète pour le novice (figure 2.1). L'écran noir avec l'unique capacité de taper quelques caractères peut sembler limitative voire rétrograde. Et pourtant, la puissance de l'écrit continue à surpasser le « clic souris », en particulier lorsqu'il s'agit de mélanger l'action de différents programmes. Là où les programmes à interface graphique communiquent en général très mal entre eux, les lignes de commande vont pouvoir permettre d'automatiser des actions sur des fichiers, processus ou autre. La force de la ligne de commande qui la rend encore indétrônable réside dans sa simplicité : pas besoin d'aller dans le douzième sous-menu qui aura changé de place à la version suivante du logiciel. Ici, tout est écrit, ré-utilisable, finement paramétrisable et surtout ré-utilisable. En effet, surtout pour les scientifiques, il est courant d'avoir à répéter l'extraction de données d'une manipulation. Pour les enseignants, il n'est pas moins courant d'avoir à répéter des tâches plus ou moins basiques.

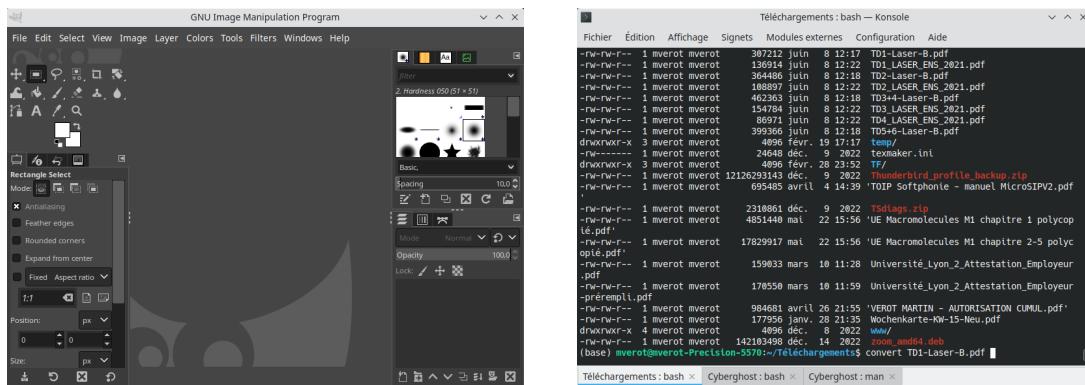


FIGURE 2.1 – Logiciel à interface graphique à gauche, console avec invite de commande à droite.

Or quoi de plus lassant que de traiter sa vingtième page scannée pour avoir un fond blanc, ouvrir son centième fichier pour trouver la valeur de l'énergie totale de son système. Trouver la position du maximum d'une courbe et j'en passe. Au troisième millénaire, les deux mondes ont encore totalement leur place et chacun présente des avantages et inconvénients du point de vue du profane.

## 2.1 Avantages et inconvénients des interfaces graphiques

### Avantages

- Simple d'utilisation en apparence
- Rétroaction immédiate sur les actions entreprises
- Permet d'enchaîner des actions indépendantes et peu reproductibles

### Inconvénients

- Limité à quelques programmes
- Une proportion non négligeable des programmes les plus connus avec interface graphique sont payants
- En général très peu adapté au traitement répétitif ou de masse,
- Difficulté à combiner différents programmes
- Pour le travail sur des machines à distance, généralement impossible
- Pour certains logiciels, chaque nouvelle version amène son lot de chamboulement dans les menus et sous-menus qui changent de place.

## 2.2 Avantages et inconvénients de la ligne de commande

### Avantages

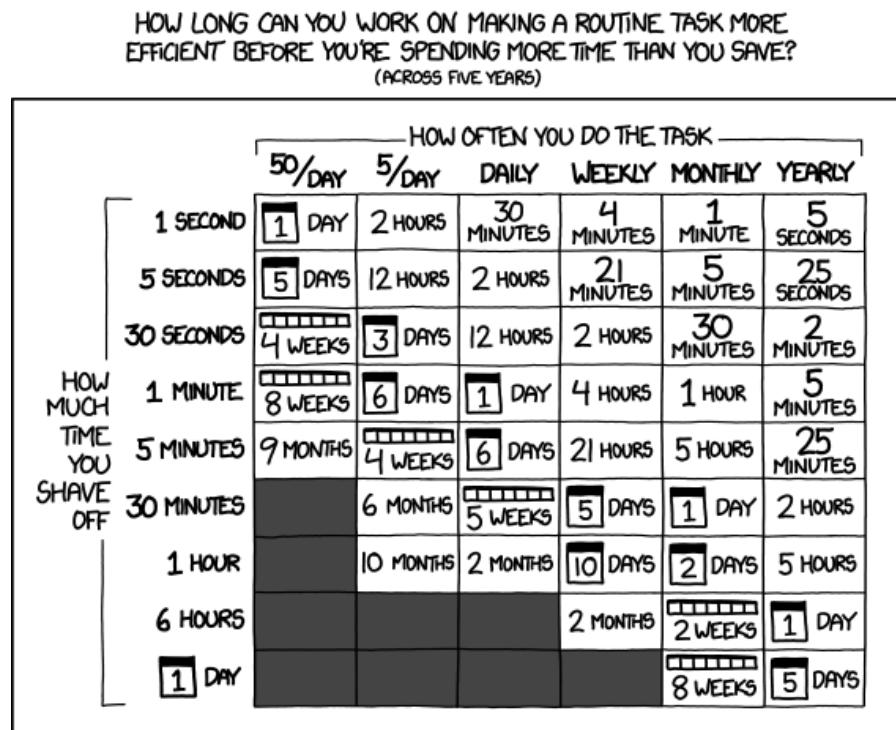
- Beaucoup de programmes capables de faire des actions très spécifiques.
- Une documentation généralement fournie
- La possibilité de « finetuner » les paramètres du programme pour faire exactement ce qui est souhaité
- Énorme capacité à répéter une séquence d'action spécifique
- Plus économique en ressources sur des environnements partagés

### Inconvénients

- Beaucoup plus aride initialement
- Mémorisation pas toujours facile des options et sous-options
- Absence de rétro-action immédiate qui peut engendrer de la frustration voire des erreurs difficilement réparables
- Une courbe d'apprentissage initiale plus élevée que pour un logiciel à interface graphique
- La compréhension des options et possibilités n'est pas toujours évidente
- Il est parfois difficile de savoir qu'il existe déjà des programmes tout fait pour faire des actions utiles

## 2.3 Et au final, on fait quoi ?

Le but n'est pas de jeter le bébé avec l'eau du bain : il faut savoir utiliser chaque outil lorsqu'il est le plus adapté. Mais pour tout ce qui est tâches répétitives, le temps passé à optimiser les choses peut très rapidement être rentabilisé (figure 2.2).



**FIGURE 2.2** – Temps que l'on peut passer à automatiser la tâche en fonction du temps économisé au final et de la fréquence à laquelle on effectue la tâche. Sur 5 ans, pour une tâche répétée 5 fois par jour, si on gagne 5 minutes par opération, alors tant qu'on met moins de 4 semaines à l'automatiser, on reste gagnant! ©xkcd <https://xkcd.com/1205>



# Chapitre 3

## Commandes de base

Cette partie vise à introduire le terminal et ses avantages. Sans tomber dans une liste à la Prévert des commandes utilisables et de leurs options, le but est ici de fournir quelques bases et pistes pour avoir un bagage minimal. Encore une fois, la pratique, les habitudes et les besoins se chargeront de vous faire progresser dans l'utilisation de certaines commandes, utilitaires ou pratiques. En particulier, ici, on décrit l'utilisation de bash, environnement utilisé au CBP de l'ENS de Lyon et sur la plupart des distributions courantes. La programmation en bash sera évoquée mais là encore, le sujet est trop vaste pour pouvoir être traité en intégralité.

 Dans ce chapitre, les commandes seront indiquées sur fond grisé comme ceci : `mkdir`. De plus, le symbole « \$ » en début de ligne est le prompt et indique qu'il s'agit d'une ligne de commande tapée dans le terminal.

### 3.1 ▲ Trouver les utilitaires en ligne de commande et LA commande maîtresse : `man`

Nous allons ici parler de différentes commandes de base. Cela peut suffire pour créer un sentiment d'overdose pour le néophyte. Cependant, il faut se dire que :

- la pratique est la meilleure maîtresse pour connaître lesdites commandes ;
- chacun connaît son petit lot de trucs et astuces, comme pour les applications sur smartphone, la commande jugée indispensable par certains sera totalement inutile pour d'autres ;
- internet est notre meilleur ami, en particulier avec une question posée en anglais de la forme « *What is the command line for [votre besoin du moment] under Linux ?* »

Une fois l'utilitaire trouvé, il peut y avoir un nombre d'options qui peuvent s'ajouter. Celles-ci correspondent généralement au fait d'ajouter des « -a », « -r », « -o », etc qui peuvent être plus ou moins cryptiques. Plutôt que de ré-inventer la roue, il est très souvent bien plus utile d'utiliser ces options que de recoder quelque chose qui utilise mal ledit outil. Encore une fois : « Inutile de ré-inventer la roue. » Surtout quand ce sera probablement moins bien fait que par des gens dont c'est le métier. Il arrive parfois que les forums indiquent la bonne option, mais pas toujours.

Pour continuer avec les dictons, en informatique, une des règles maîtresse tient en quatre lettre « RTFM » pour *Read That Fucking Manual*. L'écriture d'un manuel étant pénible, les gens qui se sont embêtés à la faire ne l'ont pas fait par plaisir mais pour que les gens utilisent correctement un programme à son plein potentiel. La commande `man` suivie du nom du

programme. La liste des options, et le manuel du programme est alors donné. Bien que le manuel puisse être long et/ou écrit dans un langage un peu cryptique, cela reste la meilleure source d'information disponible. Quitte à ensuite préciser sa question sur internet pour avoir des exemples illustrés utilisant ladite option. La touche `q` permet ensuite de fermer l'aide.

## 3.2 Système de fichier et arborescence

Si les nouveaux systèmes d'exploitation brouillent les cartes en essayant de cacher cela, sur un ordinateur, toutes les données enregistrées sont stockées dans un système de fichier. Et quoi qu'il se passe, ce système de fichier possède deux types d'objets principaux : les dossiers et les fichiers.

Ces deux structures forment l'arborescence de fichier. Cette arborescence fonctionne comme un système de coordonnées sur le disque dur.

Il y a une racine unique qui est le point de départ de toute l'arborescence. Ce point est l'origine. Sous Linux, cette racine correspond à l'endroit « / »

Ensuite, chaque nouveau dossier correspond à une nouvelle branche ou ramifications de l'arbre. Il est ainsi possible de spécifier de proche en proche où sont les choses. Le chemin correspond aux « coordonnées » de l'endroit. Le chemin est donné avec un caractère « / » pour indiquer un nouveau dossier.

Ainsi, le fichier situé au point suivant de l'arborescence :

```
/home/mverot/Téléchargements/Logiciels/test.pov
```

sera dans le dossier « Logiciels » placé dans le dossier « Téléchargements » lui-même dans le dossier « mverot » du dossier « home » situé dans la racine.

### 3.2.1 ▲ Mieux connaître l'arborescence : la commande ls

Pour pouvoir se déplacer le long de l'arborescence, il faut déjà la connaître. Pour cela, une des commandes les plus utilisées est `ls` (pour *list*). En indiquant un répertoire après la commande, alors celle-ci est exécutée pour le répertoire correspondant.

Dans sa version la plus basique, la commande va afficher tous les noms des dossiers et des fichiers. Certaines options vont permettre d'afficher plus ou moins d'informations (tableau 3.1). Ainsi, il est fréquent d'utiliser `ls -l` ou `ls -lrt`.

La commande `tree` permet de faire un listing analogue à celui obtenu avec la commande `ls -R`.

### 3.2.2 Les dossiers et fichiers particuliers

**Le « home »** Sous Linux, les dossiers particuliers n'ont pas la même structure que sous Windows. Un des dossiers les plus importants est le « home ». Ce dossier correspond à votre dossier personnel et se situe au sein du dossier « /home ». Comme l'indique son nom : ce dossier est votre maison et n'appartient qu'à vous. Ce dossier a le même nom que votre identifiant sur la machine. Par exemple, pour moi, il s'agit de `/home/mverot`. Il y aura un `home` par utilisateur sur la machine. Et chaque utilisateur a tous les droits sur ce dossier (voir section 3.2.6). Ce répertoire est par contre privé et les autres utilisateurs ne peuvent pas voir ce qu'il y a chez vous sauf si vous les autorisez. C'est l'équivalent du dossier personnel sur Windows (« Mes Documents » ou « Ce PC » ou « C:\Users\ »).

Option	Effet
-l	version détaillée qui donne la taille des fichiers, leur dernière date de modification ainsi que les droits sur le fichier (section 3.2.6)
-t	classe les fichiers par dernière date de modification, utile pour avoir les derniers fichiers à la fin
-r	pour intervertir l'ordre (alphabétique ou chronologique en fonction des autres options utilisées)
-a	liste tous les fichiers, y compris les fichiers cachés qui commencent par un « . »
-R	lister les fichiers et dossiers de manière récursive (dans tous les sous-dossiers du répertoire courant)
-h	Indique la taille des fichiers sous forme plus lisible (Ko, Mo, etc)

TABLEAU 3.1 – Quelques options courantes pour la commande `ls`.

**dossiers « /bin » « /usr/bin »** Ce dossier contient en général les programmes installés. Le dossier « /bin » contenant les programmes essentiels pour le système tandis que « /usr/bin » contient les programmes non essentiels. C'est un peu l'équivalent de « C :\Program Files » sous Windows

**dossier « /media »** Ce dossier contient les points de montage des médias amovibles (téléphone, disque dur, clé USB)

**dossiers et fichiers cachés** Les dossiers et fichiers commençant par un point « . » sont des fichiers cachés qui ne sont pas affichés par défaut par la commande `ls` ou dans l'explorateur de fichier.

### 3.2.3 ▲ Se déplacer avec la commande cd

Pour manipuler des fichiers, il est souvent nécessaire de se déplacer. Il est avant tout nécessaire de savoir où l'on est. Pour cela, il est possible d'utiliser la commande `pwd` (*print working directory*) qui indique le lieu où nous sommes sur l'arborescence.

LA commande pour se déplacer est `cd` pour *change directory*.

Il est possible de se déplacer de deux manières dans l'arborescence :

- soit en « **chemin absolu** » c'est à dire en partant de la racine à chaque fois. C'est comme utiliser des coordonnées GPS pour trouver un point : tout est référencé par rapport à l'origine de l'arborescence qui est la racine. Dans ce cas, il faut forcément commencer par la racine et donc le caractère « / ». L'avantage de la navigation absolue est qu'elle ne dépend pas de l'endroit où on est.
- soit en « **chemin relatif** » c'est à dire par rapport à l'endroit où vous êtes (l'équivalent de « vous prenez la prochaine à gauche »). Dans ce cas, on indique le chemin par rapport au point de l'arborescence où l'on est. Le déplacement relatif est avantageux car :
  - si le déplacement est petit, en général, la commande correspondante est plus courte donc plus rapide à taper ;
  - le déplacement relatif permet d'être indépendant de l'arborescence complète mais juste du « paysage local ». Donc par exemple si on décomprime un fichier chez

soi, il est possible de se déplacer de manière similaire si on part du même point de départ.

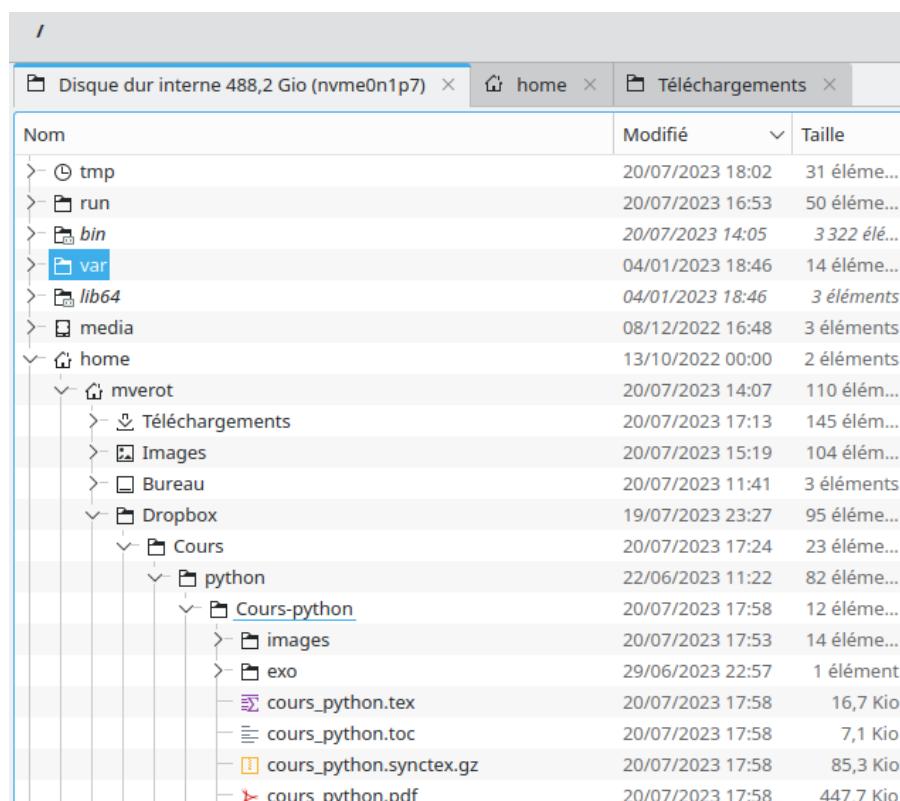
💡 Le home étant un dossier où il est courant de se déplacer, le symbole tilde « ~ » est équivalent à avoir tapé le chemin correspondant.

💡 Si on veut aller dans des sous-dossiers, il n'y a pas de problème particulier. S'il faut remonter dans l'arborescence, alors il faut utiliser « .. / » pour remonter d'un dossier, « ... / .. / » pour remonter de deux dossiers, etc..

💡 La touche tabulation  permet de faire de la complétion automatique. Si vous appuyez dessus, le système essaiera de compléter autant que possible le nom du fichier. Si jamais il y a plusieurs possibilités, en appuyant de nouveau sur la touche, le terminal indique les différentes possibilités restantes.

💡 Sans argument supplémentaire, la commande `cd` vous emmène dans votre home.

💡 « ./ » désigne le dossier courant.



Nom	Modifié	Taille
tmp	20/07/2023 18:02	31 élém...
run	20/07/2023 16:53	50 élém...
bin	20/07/2023 14:05	3 322 élém...
var	04/01/2023 18:46	14 élém...
lib64	04/01/2023 18:46	3 éléments
media	08/12/2022 16:48	3 éléments
home	13/10/2022 00:00	2 éléments
mverot	20/07/2023 14:07	110 élém...
Téléchargements	20/07/2023 17:13	145 élém...
Images	20/07/2023 15:19	104 élém...
Bureau	20/07/2023 11:41	3 éléments
Dropbox	19/07/2023 23:27	95 élém...
Cours	20/07/2023 17:24	23 élém...
python	22/06/2023 11:22	82 élém...
Cours-python	20/07/2023 17:58	12 élém...
images	20/07/2023 17:53	14 élém...
exo	29/06/2023 22:57	1 élément
cours_python.tex	20/07/2023 17:58	16,7 Kio
cours_python.toc	20/07/2023 17:58	7,1 Kio
cours_python.synctex.gz	20/07/2023 17:58	85,3 Kio
cours_python.pdf	20/07/2023 17:58	447,7 Kio

FIGURE 3.1 – Morceau d'arborescence.

Ainsi, pour l'arborescence donnée figure 3.1, toutes les commandes suivantes sont équivalentes :

```
$ pwd
/home/mverot
$ cd Dropbox/Cours/python/Cours-python/
$ cd ./Dropbox/Cours/python/Cours-python/
-----
$ pwd
/home/mverot/Dropbox/Cours/python/Cours-python/images
$ cd ../
```

```
$ cd /home/mverot/Dropbox/Cours/python/Cours-python/
-----
$ cd ~/Dropbox/Cours/python/Cours-python/
```

### 3.2.4 ▲ Crédit, modification, suppression de fichiers ou dossiers

**Les noms de fichier : les choses à faire et à ne pas faire** Même si nous sommes au 21<sup>e</sup> siècle, que l'UTF-8 existe, la manipulation de fichiers reste une opération de bas niveau, qui peut facilement être perturbée et mener à des catastrophes monumentales si mal maîtrisée. Bien qu'il soit tout à fait possible de contourner ces recommandations, il y aura **toujours** un moment où les avoir outrepassé vous mènera à des soucis plus ou moins importants.<sup>a</sup> La liste ci-dessous n'est pas forcément exhaustive, mais vous aidera à avoir une arborescence propre et facilement manipulable.

1. Ne mettez **JAMAIS** d'espaces dans vos noms de fichiers : en ligne de commande, un espace indique que l'on donne un nouvel argument au programme. Si les caractères ne sont pas échappés avec un « \ », alors **beaucoup** de choses peuvent se passer en fonction de la commande.
2. Ne mettez JAMAIS d'espaces dans vos noms de fichiers, parce que ça peut vraiment mener à une catastrophe.<sup>b</sup>
3. N'utilisez pas d'espace dans vos noms de fichiers, parce que c'est vraiment source de problème.<sup>c</sup>
4. Évitez les caractères non ASCII (de base) dans vos noms de fichiers. En particulier les caractères accentués, guillemets, de ponctuation (sauf le . avant l'extension de fichier), etc.
5. Pour les fichiers numérotés, mettez du padding (des zéros) pour que l'ordre de classement corresponde à l'ordre numérique (0001, 0002, 0003, etc plutôt que 1,2,3,4, etc). Car l'ordre de tri courant est un tri alphabétique qui place le fichier « 2-output.out » après le fichier « 10-output.out ».
6. Pour les fichiers avec des dates dedans, préférez l'ordre année-mois-jour (2023-07-21 pour le 21 juillet 2023) plutôt que l'ordre mois-jour-année (21-07-2023). Encore une fois car avec l'ordre année-mois-jour les fichiers seront naturellement classé par ordre chronologique. Ce qui ne sera pas le cas sinon.
7. Si possible, mettez une extension à votre fichier pour indiquer explicitement à vos utilisateurs le type de fichier fourni. (oui, ce n'est pas nécessaire, mais c'est gentil pour les autres !) Et ce n'est pas parce que Windows a décidé de masquer cette information dans son navigateur que c'est une bonne idée !

💡 Il est possible d'utiliser des jokers ou *wildcards* pour les noms de fichier afin de rendre les opérations plus souples (tableau 3.2).

- a. Si si, surtout quand vous vous y attendez le moins et que vous aurez fait une grosse bêtise.
- b. Oui, c'est suffisamment important pour être dit deux fois.
- c. Et même une troisième fois !

Joker	Rôle	Exemples
?	un unique caractère	cat?.png / cata.png cat0.png mais pas cathy.png
*	n'importe quel caractère	cat* / cathy.png catapulte.png cata.txt mais pas cabale.png
[xyz]	n'importe quel caractère indiqué entre les crochets une fois	test[abc] / testa testb testc mais pas testab ni test

TABLEAU 3.2 – Caractères et wildcards.

**Déplacer ou renommer un fichier ou un dossier : la commande mv** La commande `mv` (*move*) permet de déplacer un fichier il faut indiquer le fichier source en premier et la destination ensuite.

```
$ mv ancien-nom.txt nouveau-nom.txt
$ mv fichier-a-deplacer.txt nouveau-dossier/nouveau-nom.txt
$ mv fichier-a-deplacer.txt nouveau-dossier/
```

**Copier un fichier ou un dossier : la commande cp** La commande `cp` (*copy*) permet de copier les fichiers ou dossiers, sa syntaxe est analogue à celle de la commande `mv`. Pour copier des dossiers intégralement, il faut utiliser l'option « `-r` » ou « `-R` ».

**Supprimer un fichier ou un dossier : la commande rm** La commande `rm` (*remove*) permet de supprimer les fichiers ou dossiers, pour supprimer un dossier, il faut utiliser l'option « `-r` » ou « `-R` » (sinon, il existe aussi la commande spécifique `rmdir` pour supprimer les dossiers vides uniquement).

💡 **La commande rm est TRÈS dangereuse**, en particulier quand on la combine avec des options et des jokers. Ainsi, la commande « `rm -rf *` » peut supprimer tous les fichiers et sous-dossiers du répertoire courant sans aucun avertissement. Pour peu qu'une commande `cd` ait emmené dans le mauvais répertoire, alors on peut se retrouver à effacer le mauvais répertoire et tout ce qu'il contient !

**Créer un dossier : la commande mkdir** La commande `mkdir` (*make directory*) permet de créer un dossier.

**Créer un fichier : la commande touch** La commande `touch` permet de créer des fichiers vides, (elle permet également de mettre à jour la date de dernière modification d'un fichier)

### 3.2.5 ▲ Trouver des fichiers

Pour trouver des fichiers, les navigateurs restent malgré tout peu efficaces. Dans le terminal, la commande `find` est extrêmement efficace. Par défaut, la recherche s'effectue à partir du répertoire courant mais il est possible de préciser un autre répertoire. L'option « `-name` » permet de faire une recherche sur le nom de fichier et l'option « `-iname` » pour une recherche insensible à la casse (aux majuscules et minuscules).

Pour trouver un fichier contenant le mot `python` dans son home :

```
$ find ~ -name '*python*'
```

Pour trouver un fichier contenant le mot Python dans le répertoire courant :

```
$ find . -iname '*python*'
```

Les options de la recherche sont très nombreuses, la lecture du manuel de la commande indique comment faire des recherches plus spécifiques : profondeur de la recherche, permissions, par droits, avec des expressions régulières (section 10), etc.<sup>d</sup>

### 3.2.6 ▲ Droits

#### 3.2.6.1 Exécuter un programme/fichier

Pour exécuter un programme, il faut le lancer en précisant le nom du fichier pour le différencier d'une ligne de commande.

```
$ ./script.py
```

permettra ainsi d'exécuter le fichier "script.py" pour peu que celui-ci ait un shebang et soit exécutable (voir ci-après, sections 3.12 et 4.6).

#### 3.2.6.2 Connaître les droits

Comme les ordinateurs peuvent être partagés, Les systèmes UNIX assurent un compartimentage strict des données. Il y a ainsi plusieurs catégories d'utilisateurs (tableau 3.3).

Lettre	Groupe
u	l'utilisateur ou <i>user</i> propriétaire du fichier, cette information est donnée par la commande <code>ls -l</code> par la première colonne après le premier nombre et avant le deuxième nombre (qui est la taille du fichier)
g	le groupe ou <i>group</i> , c'est à dire un groupe d'utilisateurs, cette information est donnée par la commande <code>ls -l</code> par la deuxième colonne après le premier nombre et avant le deuxième nombre (qui est la taille du fichier)
o	les autres ou <i>other</i> , tous les autres utilisateurs
a	tout le monde, soit les trois groupes précédents

TABLEAU 3.3 – Les différents groupes d'utilisateurs.

Chaque groupe d'utilisateurs a ainsi des droits spécifiques sur chaque fichier et chaque dossier. Cela évite ainsi que n'importe qui puisse consulter ce qu'il y a dans votre home.

Il y a également plusieurs types d'autorisation listés dans le tableau 3.4.

La commande `ls -l` permet d'indiquer les droits de chaque groupe sur le fichier.

```
$ ls -l
-rw-rw-r-- 1 mverot mverot      525 avril 18 13:32 fireflyFileOne.inp
drwxr-xr-x 9 mverot mverot    4096 févr. 25 21:53 Zotero
```

d. Bonne lecture pour la description de la centaine d'options disponibles !

Lettre	Chiffre	Droit
r	4	Droit de lecture : permet d'accéder à un fichier mais pas de le modifier.
w	2	Droit d'écriture : permet d'écrire ou modifier le fichier.
x	1	Droit d'exécution : permet de lancer le fichier en tant que programme.

TABLEAU 3.4 – Les différents droits possibles.

- La première lettre indique s'il s'agit d'un dossier (lettre « d »), d'un lien (lettre « l », voir [3.4](#)) ou d'un fichier (avec un tiret « - »)
- Les trois lettres suivantes (2 à 4) indiquent les droits du propriétaire. Ici, le propriétaire a le droit de lecture et modification sur le fichier "fireflyFileOne.inp" et tous les droits sur le dossier "Zotero".
- Les trois lettres suivantes (5 à 7) indiquent les droits du groupe. Ici, droit de lecture et modification sur le fichier "fireflyFileOne.inp" et droit de lecture et d'exécution sur le dossier "Zotero".
- Les trois lettres suivantes (8 à 10) indiquent les droits des autres utilisateurs. Ici, droit de lecture sur le fichier "fireflyFileOne.inp" et droit de lecture et d'exécution sur le dossier "Zotero".

### 3.2.6.3 Modifier les droits

Il est possible de changer les droits avec la commande `chmod` (*change mode*). Cette commande attend en deuxième argument le changement de droits à appliquer puis pour finir le fichier ou dossier sur lequel appliquer le changement.

Pour indiquer le changement de droit, il faut indiquer le groupe concerné (voir tableau [3.3](#)). Ensuite si on veut ajouter (+), définir (=) ou retirer (-) certains droits et pour finir les droits concernés (tableau [3.4](#)) – pour cela, on peut soit indiquer les lettres, soit un chiffre compris entre 0 et 7 correspondant à la somme des chiffres des droits concernés.

Ainsi, la commande suivante retirera le droit de lecture et d'écriture à tout le monde sur le fichier "secretfile.txt".

```
$ chmod a-rw secretfile.txt
```

Tandis que la commande suivante donnera accès aux autres utilisateurs pour qu'ils puissent exécuter le programme "pythonkiller.py"

```
$ chmod o+x pythonkiller.py
```

 *En pratique, nous nous servirons beaucoup de la commande `chmod +x truc.py` pour rendre nos scripts pythons exécutables. En effet, sans cette étape, il sera impossible de les exécuter.*

## 3.3 ▲ Aller plus vite dans le terminal

Les lignes à taper dans le terminal peuvent être plus ou moins longues, pour s'y déplacer plus rapidement, il y a quelques astuces listées tableau [3.5](#). Une version plus complète est disponible sur la page suivante [Handy Keyboard Shortcuts for the Linux Bash Terminal](#).

Commande	Effet
+	Aller au début de la ligne
+	Aller à la fin de la ligne
+	Effacer après le curseur
+	Effacer avant le curseur
+  /	se déplacer d'un mot vers la gauche ou vers la droite

TABLEAU 3.5 – Raccourcis au sein du terminal.

## 3.4 ■ Les liens

Outre les fichiers et les dossiers, il existe un troisième type d'objet dans l'arborescence : les liens. Cela permet de faire ... un lien sur un fichier sans le copier intégralement.

Cela peut-être utile si :

- on veut utiliser un fichier dans un répertoire d'analyse alors que celui-ci se trouve dans un répertoire de production de données ;
- on veut éviter de copier ou déplacer un fichier si celui-ci est très volumineux.
- un même fichier est utilisé à plein d'endroits différents. Il est alors plus avantageux que les modifications de ce fichier soient immédiatement répercutées partout où il est utilisé.

Il s'agit presque d'un raccourci mis à part que l'environnement du lien reste celui de son répertoire.

Pour créer un lien, la commande est `ln source lien` où "source" indique le fichier sur lequel on va faire un lien et "lien" sera le nom du lien. Il existe deux types de liens différents :

- les liens en dur *hard links* qui ont l'avantage de continuer à pointer sur le fichier source même si celui-ci est déplacé ou renommé (moyennant quelques restrictions : on ne peut pas faire de lien en dur sur un dossier ou sur un autre disque).
- les liens symboliques ou *soft links* qui peuvent servir pour des dossiers mais qui deviennent cassés si jamais la source est déplacée ou renommée.

La seule différence entre les deux est que pour créer un lien symbolique, il faut utiliser l'option « `-s` ».

```
$ ln fireflyFileOne.inp ./lien-dur
$ ln -s fireflyFileOne.inp ./lien-symbolique
$ ls -l
-rw-rw-r-- 2 mverot mverot      525 avril 18 13:32 lien-dur
-rw-rw-r-- 2 mverot mverot      525 avril 18 13:32 fireflyFileOne.inp
lrwxrwxrwx 1 mverot mverot  18 juil. 22 01:59 lien-symbolique ->
fireflyFileOne.inp
```

On peut voir que le lien dur a exactement les mêmes caractéristiques que le fichier original (taille comprise) alors que le lien symbolique a une date de modification différente et est beaucoup plus petit en taille. Il est également indiqué avec une flèche pour indiquer le fichier auquel il fait référence.

## 3.5 Redirection de la sortie

Beaucoup de commandes affichent des choses dans le terminal. Cependant, il peut être utile de stocker le résultat de ces commandes dans un fichier. Cela est permis grâce aux opérateurs de redirection qui sont les suivants :

- `>` redirige la sortie vers le fichier indiqué en **écrasant** le contenu du fichier si jamais il existe déjà.
- `>>` redirige la sortie vers le fichier indiqué en **ajoutant** le contenu à la fin du fichier si jamais il existe déjà.
- `<` permet à l'inverse de prendre un fichier en tant qu'argument d'une commande.

## 3.6 Quelques utilitaires

### 3.6.1 Connaître l'espaces disque et la taille de certains dossiers

Pour les futurs utilisateurs de centre de calcul, le respect par chacun des utilisateurs des différents espaces disque est **CRUCIAL**. C'est une source de tensions entre utilisateurs et de dysfonctionnement importante uniquement liée à l'activité humaine. En général, chaque utilisateur a droit à différentes tailles d'espace : un certain quota individuel dans le home qui est relativement petit, un espace plus grand et partagé dans un répertoire de travail appelé "scratch".

La commande `df -h` (l'option "`-h`" permettant d'avoir les tailles écrites en format plus facilement lisible) liste tous les disques et leur utilisation.

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs          3,2G  2,6M  3,1G  1% /run
/dev/nvme0n1p7  480G 274G 182G 61% /
tmpfs          16G  352M  16G  3% /dev/shm
tmpfs          5,0M  4,0K  5,0M  1% /run/lock
/dev/nvme0n1p1  286M 119M 168M 42% /boot/efi
tmpfs          3,2G 132K  3,2G  1% /run/user/1000
```

Dans l'exemple, on peut ainsi voir que le disque est utilisé à 61% avec encore 182 Go de libre pour un disque d'une capacité de 480 Go.

La commande `du -h` permet de connaître la taille d'un dossier. En l'exécutant depuis son home, il est alors possible de vérifier si on dépasse son quota ou non.

**Règles d'usage sur des clusters de calcul** En général, sur un cluster, le home est censé contenir les fichiers source de ses calculs et les fichiers de sortie condensés (indiquant des énergies ou résultats principaux). Cet espace est généralement « le nerf de la guerre » car il contient les fichiers importants des utilisateurs, il est donc généralement archivé, sauvegardé avec différentes options. Il est généralement limité à quelques dizaines de Giga-octets. Sa taille réduite et le fait qu'il soit sauvegardé implique la règle suivante :

**« On ne calcule JAMAIS dans son home. »**

Cela permet :

- d'éviter de transférer des fichiers volumineux avec un faible débit;

- d'éviter de consommer de l'espace disque partagé avec d'autres utilisateurs, ce qui pourrait compromettre le bon fonctionnement global du cluster (tous les utilisateurs utilisant leur home) ;
- d'éviter de consommer des ressources pour la duplication et l'archivage des données du home ;
- de ralentir le remise en route du cluster en cas de saturation de l'espace disque : les administrateurs systèmes ne pouvant pas faire de ménage dans le home sans risquer de faire de dégâts, alors qu'ils peuvent plus facilement libérer de l'espace sur des dossiers partagés mais non sauvegardés.

De plus, il faut veiller à faire régulièrement le ménage dans tous ses dossiers, y compris dans les dossiers de calcul et ne pas hésiter à archiver certains fichiers de données volumineux (pouvant parfois faire plusieurs tera-octets). L'espace de stockage n'est jamais gratuit, même s'il a tendance à toujours augmenter avec le temps.

 Dans le cadre de l'UE, les calculs que nous allons faire étant brefs, nous exécuterons nos fichiers dans le home. Mais dès que l'on a des opérations dont la durée totale dépasse la minute, il faut calculer sur l'espace dédié adapté. (Qu'il s'agisse d'un unique gros calcul sur plusieurs nœuds de plusieurs heures ou de milliers de calculs très brefs.)

### 3.6.2 Les fichiers texte

En tant que scientifiques, il sera courant de manipuler des fichiers de données créés par des programmes donc à la structure relativement fixe. Il existe toute une série de petits utilitaires qui peuvent servir pour manipuler les fichiers texte. Sans chercher à être exhaustif, le but est ici d'en donner une description succincte pour savoir où chercher en cas de besoin.

#### 3.6.2.1 Trouver une ligne contenant une chaîne de caractère dans plusieurs fichiers : grep

La commande `grep` est une commande extrêmement utile pour afficher le contenu d'une ligne contenant une suite de caractères bien précise. Cela peut aussi bien être une valeur pour une pression donnée, une énergie calculée, etc. Il est possible d'utiliser des expressions régulières (10), ainsi que de nombreuses options, bien évidemment toutes décrites dans le manuel.

Par exemple, pour trouver toutes les lignes contenant la chaîne de caractère "E(RAM1)" dans les fichiers avec l'extension ".log" dans les sous-dossiers du répertoire courant. On a donc récupéré une valeur dans 6 fichiers différents en une seule commande !

```
$ grep 'E(RAM1)' */*.log
1-4-opt/1-4-opt.log: SCF Done: E(RAM1) = -0.135738260762
1-5-opt/1-5-opt.log: SCF Done: E(RAM1) = -0.154419935035
1-8-opt/1-8-opt.log: SCF Done: E(RAM1) = -0.154062741345
2-3-opt/2-3-opt.log: SCF Done: E(RAM1) = -0.145644211402
2-6-opt/2-6-opt.log: SCF Done: E(RAM1) = -0.150934360839
2-7-opt/2-7-opt.log: SCF Done: E(RAM1) = -0.150202328553
```

### 3.6.2.2 Voir la fin d'un fichier : tail

Pour les personnes faisant des simulations, il est également courant de chercher à voir si elles sont finies ou non. En général, les programmes utilisés écrivent une ligne particulière en toute fin de fichier lorsque le calcul est fini. La présence de cette ligne en fin de fichier ou non permet alors de vérifier l'état du calcul. La commande `tail` permet ainsi de voir la fin du fichier sans avoir à ouvrir de programme tierce.

Ci-dessous, un exemple pour afficher la dernière ligne des fichiers en .log dans les sous-dossiers du répertoire courant. Dans ce cas, tous les calculs se sont correctement finis vu qu'il est indiqué "Normal termination".

```
$ tail -n 1 -q */*.log
Normal termination of Gaussian 09 at Mon Apr 11 23:40:49 2022.
Normal termination of Gaussian 09 at Mon Apr 11 23:40:56 2022.
Normal termination of Gaussian 09 at Mon Apr 11 23:41:05 2022.
Normal termination of Gaussian 09 at Mon Apr 11 23:41:16 2022.
Normal termination of Gaussian 09 at Mon Apr 11 23:41:22 2022.
Normal termination of Gaussian 09 at Mon Apr 11 23:41:27 2022.
```



La commande `head` fait l'inverse en affichant le début d'un fichier.

### 3.6.2.3 Concaténer des fichiers : cat

La commande `cat` (*catenate*) permet de mettre plusieurs fichiers textes bout à bout. Elle permet aussi de passer le contenu d'un fichier dans la sortie du terminal (eventuellement pour pouvoir enchaîner des commandes voir 3.7).

Pour concaténer deux fichiers dans le fichier "out.txt", il faut utiliser la commande suivante :

```
cat fichier1.txt fichier2.txt > out.txt
```

### 3.6.2.4 Comparer des fichiers : diff et wdiff

Les commandes `diff` (*difference*) et `wdiff` (*word difference*) permettent de comparer des fichiers. La commande `diff` compare des lignes complètes alors que `wdiff` compare mot à mot. La commande `colordiff` peut permettre de mettre en évidence les différences avec de la couleur (si elle est disponible). Cela peut être très utile pour savoir quel fichier correspond à la dernière version ou analyser les différences (par exemple pour voir les changements effectués par un collaborateur sur un article ou un programme).

### 3.6.2.5 🟠 🔴 Faire des modifications de fichiers (remplacement de chaîne de caractère, etc) : sed et awk

Il peut être très utile d'automatiser la création de fichier. Par exemple pour faire varier un paramètre de manière discrète. En général, cela se traduit par le fait de modifier un chiffre ou une valeur sur une unique ligne d'un fichier d'entrée. Plutôt que de créer ces fichiers à la main, il peut être (beaucoup) plus simple de partir d'un fichier et faire des modifications. Pour cela, les commandes `sed` et `awk` sont des outils de choix. Il existe des livres complets destinés à l'utilisation de ces deux commandes. Pour simplifier les choses, `sed` est un peu plus limité que `awk` mais est aussi relativement plus facile à maîtriser pour les opérations

simples et courantes. Les deux programmes sont capables d'utiliser les expressions régulières pour faire des opérations de remplacement (section 10). `awk` est également capable de faire des traitements plus sophistiqués sur des données tabulaires, faire des opérations mathématiques sur des données, etc.

Sans rentrer dans les détails, quelques exemples sont donnés pour faire des remplacements basiques avec la commande `sed`. Ce dernier fonctionne par défaut sur des lignes uniquement.

Pour remplacer la chaîne de caractère "temperature=24" par "temperature=30" dans le fichier "input24.txt" :

```
sed -e 's/temperature=24/temperature=30/' input24.txt > input30.txt
```

L'option "-e" indique d'exécuter la commande donnée entre guillemets, le "s" indique que l'on fait une substitution(remplacement) les trois slash "/" séparent d'un côté le motif recherché et de l'autre le motif de substitution. Par défaut, `sed` n'effectue qu'un remplacement par ligne, si on veut remplacer toutes les occurrences, il faut ajouter l'option "g" après le "/" final. De même, si on veut limiter la substitution à certaines lignes ou une gamme de ligne, alors il faut l'indiquer avant le "s" initial :

```
sed -e '24s/é/e/g' input24.txt > input30.txt
```

Remplacera ainsi tous les caractères "é" en "e" à la ligne 24.

Il est aussi possible de supprimer des lignes, en ajouter, utiliser des plages de valeur, etc. Un bon tutoriel sur le sujet vous aidera à progresser si jamais vous en avez l'utilité. On verra qu'il est possible de faire des choses similaires en python, mais cela pourra nécessiter plus d'efforts.

Pour `awk`, n'étant pas expert dans le domaine, je vous renvoie également à un bon tutoriel si jamais vous en avez besoin.

### 3.6.3 Les fichiers pdf

Le tableau 3.6 vous liste quelques commandes qui peuvent s'avérer utiles pour manipuler des documents pdf.

Il peut être utile d'utiliser l'utilitaire `ghostscript` pour réduire la taille des pdf contenant des images haute résolution – comme un rapport de stage. La commande suivante peut permettre de se ramener à une taille plus raisonnable :

```
gs -sDEVICE=pdfwrite -dCompatibilityLevel=1.4 -dPDFSETTINGS=/printer -dNOPAUSE -dQUIET -dBATCH -sOutputFile="fichier-compresse.pdf" fichier-enorme-car-plein-dimages.pdf
```

L'option /ebook à la place de /printer permet une réduction encore plus importante de la taille (au détriment de la résolution des images).

### 3.6.4 Reconnaissance de caractères : tesseract

Pour tout ce qui concerne la reconnaissance de caractère, la commande `tesseract` est très robuste. Il est possible d'indiquer la langue dans laquelle la reconnaissance de caractère doit être effectuée. Pour de la reconnaissance de caractère sur des textes contenant des formules mathématiques, je n'ai par contre pas de solution miracle à proposer.

Pour faire de la reconnaissance de caractère en français sur le fichier "file.png" :

Commande	Utilisation
pdftotxt	convertit un pdf en fichier texte (sans faire de reconnaissance de caractère!), l'option "-layout" permet d'essayer de garder la mise en page originale (surtout pour les tableaux), la commande pdf2txt est similaire
pdfseparate	sépare les pages d'un pdf pdfseparate cours_python.pdf %d-cours-python.pdf
pdfunite	réunit plusieurs pdf
pdfimages	extrait les images d'un pdf
pdfjam	pour mettre deux pages par feuille, et faire bien d'autres choses pdfjam --nup 2x1 --landscape -o out-nup.pdf fichier-input.pdf
pdfbook2	pour avoir une impression en format livret (pour une reliure au centre)
gs	ghostscript, utilitaire très puissant pour modifier les pdf, les convertir en image, etc

TABLEAU 3.6 – Quelques commandes liées à la manipulation de documents pdf

```
$ tesseract "file.png" "outputText" -l fra
```

### 3.6.5 Les images : ImageMagick

Pour tout ce qui concerne la manipulation d'image, l'utilitaire ImageMagick permet d'effectuer une large palette d'opérations. la commande correspondante est convert. Il est possible de couper, redimensionner, gérer la transparence, faire des rotations, enlever le fond d'une image, faire de la détection de contours, etc. Encore une fois, plutôt que de longuement décrire l'utilisation de ce programme, ce sera plutôt de l'aide en ligne qui vous indiquera la bonne combinaison d'options adaptée à vos besoins.

### 3.6.6 Les vidéos : ffmpeg

Le programme ffmpeg est l'équivalent d'ImageMagick pour les vidéos, les options sont tout aussi nombreuses et permettent de faire de l'extraction, de la conversion, etc.

### 3.6.7 Compression et décompression

Pour compresser en zip des fichiers, la commande est de la forme :

```
$ zip archive *
```

Pour mettre tous les fichiers du répertoire courant dans le fichier "archive.zip". La commande pour dézipper est unzip.

Pour la compression d'un ensemble de fichier, si la taille est un enjeu, il faut mieux utiliser le « format » .tar.gz qui compresse plus efficacement un ensemble de fichier alors que zip comprime les fichiers individuellement.

Pour décompresser une archive, la commande à utiliser est la suivante :

```
tar -xvzf archive.tar.gz
```

Et pour compresser :

```
tar -cvfz archive *
```

Les options "-x" et "-c" indiquant l'extraction ou la création respectivement. Les options "-v" (*verbose*) sont là pour afficher les fichiers concernés, l'option "-z" indique le format de compression ou décompression (gzip) et l'option "-f" indique que le fichier est donné en paramètre.

## 3.7 ■ Enchaînement de commande : le pipe

Si chaque commande peut être individuellement puissante, il est possible de faire des choses encore plus puissantes en combinant plusieurs d'entre elles. Le caractère | ou pipe permet de faire cela. Dans ce cas, il est possible d'enchaîner des commandes, par exemple trouver des fichiers avec la commande `find` pour ensuite les déplacer avec la commande `mv`. En fonction des situations, la commande `xargs` permet d'utiliser la sortie d'une commande précédente en tant qu'argument pour la commande suivante.

Par exemple, pour trouver la dernière occurrence d'une chaîne de caractère, il est possible de combiner les commandes `grep` et `tail` :

```
$ grep 'E(RAM1)' anthracene-2-7-opt.log
SCF Done: E(RAM1) = -0.145473375628 A.U. after 16 cycles
SCF Done: E(RAM1) = -0.149406876134 A.U. after 13 cycles
SCF Done: E(RAM1) = -0.150154480084 A.U. after 12 cycles
SCF Done: E(RAM1) = -0.150196887686 A.U. after 11 cycles
SCF Done: E(RAM1) = -0.150202328553 A.U. after 10 cycles
$ grep 'E(RAM1)' anthracene-2-7-opt.log | tail -n 1
SCF Done: E(RAM1) = -0.150202328553 A.U. after 10 cycles
```

La commande `tail` utilise ainsi le résultat de la commande `grep` pour ne garder que la dernière ligne et donc afficher le résultat final de l'énergie après optimisation de la géométrie (qui est le seul chiffre intéressant).

Ainsi, la combinaison de deux commandes basiques peut rapidement mener à des résultats très puissants. D'autant plus qu'il est possible d'utiliser plusieurs pipe de suite et donc d'encore démultiplier la puissance des différentes commandes.

## 3.8 ■ Les alias et le fichier .bashrc

Il arrive d'avoir à taper régulièrement des commandes avec certaines options. Pour cela, il est parfois utile de se faire des raccourcis personnels. C'est le rôle des alias qui sont de nouvelles commandes personnalisées. Le mécanisme passe par un fichier dans lequel on définit ces alias : le fichier ".bashrc" qui se trouve dans la racine du home. Par défaut, ce fichier est caché. Il faut donc commencer par ouvrir ce fichier avec les commandes suivantes :

```
$ cd
$ gedit .bashrc &
```

Le programme `gedit` permet alors de modifier le fichier pour ajouter ses alias. Pour cela, il suffit d'ajouter une ligne commençant par `alias`, puis sa définition.

Par exemple, pour avoir directement l'affichage des données sous forme complète classées par ordre chronologique, il est possible d'utiliser la ligne suivante :

```
#mes alias
alias ll='ls -lrt'
```

Il faut ensuite enregistrer le fichier puis forcer le système à le recharger pour qu'il prenne en compte l'alias nouvellement défini. Cela se fait avec la commande `source .bashrc`. Il est alors possible d'utiliser directement votre nouvel alias.

 Le fichier `.bashrc` contient également des informations importantes, et s'il est mal configuré, cela peut mener à des catastrophes. Il est donc de bon ton de **toujours** faire une copie de sauvegarde avant de le modifier pour pouvoir revenir en arrière.

 Le fichier `.bashrc` est ce qui permet de personnaliser l'environnement, il est généralement utile d'en garder une archive pour revenir à un terminal configuré aux petits oignons en cas de changement de machine ou d'utilisation d'un nouvel environnement de travail.

## 3.9 Environnement : PATH

Le système va chercher les commandes disponibles dans certains répertoires particuliers du système. L'ensemble de ces dossiers s'appelle le PATH. Pour le visualiser, il est possible d'utiliser la commande suivante :

```
$ echo $PATH
/home/mverot/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Chaque répertoire utilisé est séparé par le symbole ":". Il est possible de savoir exactement où se trouve chaque programme avec la commande `which`.

Sur un environnement partagé, en général, si l'administrateur installe un logiciel, il fait en sorte que ce dernier soit accessible à tous les utilisateurs en s'assurant que le dossier d'installation fasse partie du PATH. Cependant, il arrive également que certains utilisateurs aient à installer des programmes pour leurs besoins personnels. Il est alors intéressant de les ajouter au PATH pour y avoir accès quel que soit le répertoire courant.

Pour faire l'ajout de manière temporaire en ajoutant par exemple le dossier "Scripts" de son home :

```
export PATH="$PATH:$HOME/Scripts"
```

 Attention, la modification du PATH, si elle est mal faite peut amener à une catastrophe : on peut par exemple perdre l'accès à tous les programmes si on a enlevé les répertoires importants. Il faut donc être minutieux et rigoureux lors de ces opérations.

Il est également possible de faire l'ajout de manière permanente en modifiant le fichier `.bashrc` :

```
$ cd
$ gedit .bashrc &
##ajouter la ligne suivante au fichier puis l'enregistrer
PATH="$PATH:$HOME/Scripts"
$ source .bashrc
```

 Le fichier `.bashrc` contient également des informations importantes, et s'il est mal configuré, cela peut mener à des catastrophes. Il est donc de bon ton de **toujours** faire une copie de sauvegarde avant de le modifier pour pouvoir revenir en arrière.

## 3.10 ■ Travail sur des machines distantes

Pour le calcul de grande ampleur, il est maintenant systématique d'avoir recours à des clusters de calcul. Pour cela, il faut travailler sur des ordinateurs distants. Le fait d'avoir à travailler à distance fait qu'en général, il n'est pas possible de s'y connecter avec une interface graphique. En effet, cela est couteux en bande passante et pour des interfaces partagées avec plusieurs centaines d'utilisateurs, cela demanderait trop de ressources. Un des outils privilégiés est le protocole ssh. Ce protocole est sécurisé et robuste.

Le protocole ssh permet de se connecter sur une machine distante. Cependant, en général, plutôt que d'aller directement sur les machines d'intérêt, il faut passer par une passerelle (*gateway*) (figure 3.2). Cette passerelle sert à protéger les machines d'attaques.

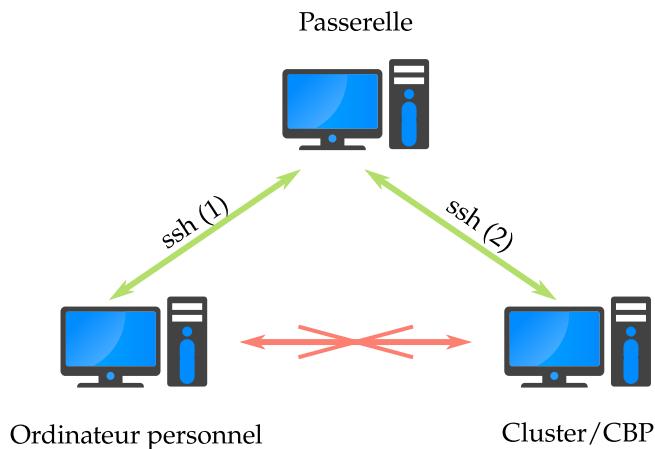


FIGURE 3.2 – Schéma traditionnel d'une connexion ssh via une passerelle pour accéder à un cluster. Il n'est pas possible de se connecter directement en ssh au cluster et il est nécessaire de passer par une passerelle.

### 3.10.1 Connexion sur des machines distantes : ssh

La commande pour se connecter en ssh à une machine est ... `ssh` il faut indiquer son identifiant en préfixe, suivi d'un "@" puis le nom de la machine. Si jamais son identifiant est le même sur les deux machines, alors il n'est pas nécessaire de le préciser.

Pour l'ENS de Lyon, il faut commencer par activer son accès ssh via l'ENT : <https://intranet.ens-lyon.fr/ent> (connexions réseaux > Activer l'accès SSH).

La procédure donnée ci-après indique comment se connecter à une machine du CBP depuis l'extérieur de l'ENS de Lyon. Depuis l'ENS, il n'est pas nécessaire de passer par la passerelle.

1. Pour se connecter à la passerelle de l'ENS de Lyon, il faut se connecter avec son nom d'utilisateur sur la machine `ssh.ens-lyon.fr` :

```
$ ssh mverot@ssh.ens-lyon.fr
```

Puis ensuite, il faut taper son mot de passe qui est le même que celui de l'ENS de Lyon pour le webmail.

2. Ensuite, pour se connecter au CBP, il faut choisir une des machines depuis la page <https://www.cbp.ens-lyon.fr/python/forms/CloudCBP> (accessible elle-même depuis <https://www.cbp.ens-lyon.fr/doku.php?id=ressources:ressources>) et lui ajouter le suffixe ".cbp.ens-lyon.fr" :

```
ssh mverot@c8220air7.cbp.ens-lyon.fr
```

Encore une fois, le mot de passe à utiliser est le même que votre mot de passe général de l'ENS de Lyon.

Il est possible d'enchaîner les deux commandes sur une seule ligne (noter l'ajout de l'option "-t") :

```
ssh -t mverot@ssh.ens-lyon.fr ssh -t mverot@c8220air7.cbp.ens-lyon.fr
```

Dans ce cas, deux mots de passe sont demandés : un pour se connecter sur la passerelle et un pour se connecter au CBP. Il s'agit ici du même mot de passe dans les deux cas, mais dans certains cas, il peut s'agir de deux mots de passe différents.

Lors de ces différentes manipulations, rien n'a changé en apparence (si ce n'est l'apparition de quelques messages pour indiquer la connexion sur une nouvelle machine). Cependant, nous avons bien changé de station de travail ! Pour le vérifier, il est possible d'utiliser la commande `hostname` puis `hostname -d` qui permet de donner le nom de la machine sur laquelle nous sommes et son nom de domaine. Pour fermer la connexion ssh, il suffit de taper la commande `exit` deux fois d'affilé (une fois pour fermer la connexion entre la passerelle et le CBP et la deuxième fois pour fermer la connexion entre l'ordinateur actuel et la passerelle).



Sous Windows, l'utilitaire putty permet de se connecter en ssh.

### 3.10.2 Au CBP : x2go

Au CBP, il est possible d'avoir une connexion avec une interface graphique complète via l'utilitaire x2go. Pour cela, toutes les informations sont disponibles sur la page suivante : <https://www.cbp.ens-lyon.fr/doku.php?id=ressources:x2go4cbp>. On travaille alors à distance « comme si on y était ». (ou presque : certains programmes ne s'ouvrent pas à distance à cause de contraintes techniques)

### 3.10.3 Copie d'un fichier à distance : scp

S'il est nécessaire d'utiliser des centres de calcul pour réaliser des simulations, il est en général très utile ou nécessaire de faire des étapes de post-traitement sur son ordinateur personnel (par facilité, pour avoir accès à certains programmes ou pour pouvoir utiliser certains programmes avec des interfaces graphiques). Il faut alors faire des transferts de fichier.

En ayant accès à un navigateur web, en étant membre de l'ENS :

- pour les membres du réseau RENATER dont l'ENS fait partie <https://filesender.renater.fr/>.

Les fichiers à transférer peuvent faire plusieurs dizaines de giga-octets.

Lorsque l'accès à un navigateur n'est pas possible, il est possible d'utiliser la commande `scp`. Cette commande permet de faire une copie d'une machine A à une machine B. La première partie de la commande indique le fichier source et la deuxième partie la destination. Comme la commande est exécutée depuis la machine A (ou la machine B) il est en général plus facile d'écrire la partie de la commande concernant la machine locale. Pour la machine locale, il suffit de préciser le chemin (absolu ou relatif) et pour la machine distante, il faut préciser les informations de connexion (comme pour la commande `ssh`) puis utiliser le caractère ":" pour préciser le chemin sur la machine distante. Encore une fois, il faut rentrer son mot de passe sur la machine distante pour que le transfert soit effectif.

 Malheureusement, sur la machine distante, il n'est pas possible d'utiliser la compléition automatique vu que le terminal n'a pas accès à l'arborescence sur celle-ci.

Pour un utilisateur "mverot" et les deux machines A et B s'appelant respectivement "calvin.reseauA" et "hobbs.reseauB", si on cherche à transférer un fichier "data.out" dans le répertoire "Simulation" situé dans son home sur la machine A vers le répertoire "Analyse" dans le home de la machine B.

Si on est sur la machine A et que l'on veux copier sur la machine B :

```
$ pwd
/home/mverot/Simulation
$ hostname
calvin
$ scp data.out mverot@hobbs.reseauB:~/Analyse
$ scp ~/Simulation/data.out mverot@hobbs.reseauB:~/Analyse
```

Si on est sur la machine B et que l'on veut copier depuis la machine A :

```
$ pwd
/home/mverot/Analyse
$ hostname
hobbs
$ scp mverot@calvin.reseauA:~/Simulation/data.out .
$ scp mverot@calvin.reseauA:~/Simulation/data.out ~/Analyse
```

 Pour copier en passant par une passerelle à l'ENS :

```
scp -oProxyCommand="ssh -W %h:%p login@ssh.ens-lyon.fr" login@nomdemachine.
cbp.ens-lyon.fr:~/emplacement-fichier .
```

Où il faut adapter le login, le nom de la machine et l'emplacement du fichier.

### 3.10.4 Copie d'un répertoire à distance : rsync

La commande `scp` permet de copier un unique fichier mais si on veut synchroniser des dossiers, cela n'est pas adapté, ou demande de faire une archive, faire un `scp` puis décompresser l'archive. La commande `rsync` permet de directement synchroniser des dossiers entre deux machines distantes. Dans le cas de l'ENS, il faut passer par la passerelle.

```
rsync -avz --progress --stats -e "ssh -J user@ssh.ens-lyon.fr" user@machine.
cbp.ens-lyon.fr:remote-dir local-dir
```

en remplaçant "user" par votre nom d'utilisateur, "machine" par le nom de la machine du CBP qui vous intéresse, "remote-dir" par le chemin qui mène au dossier distant et local-dir par le chemin qui mène au dossier là où voulez faire la synchronisation (en général le répertoire courant donc ".").

 L'utilisation de `rsync` permet également de faire des sauvegardes, en particulier si utilisé avec `crontab`, je vous laisse faire des recherches avec ces mots-clés si vous êtes intéressés.

## 3.11 Gestion des processus

### 3.11.1 Fermer un processus qui buggue ou ralentit le système

Comme toujours, il peut arriver de faire des bêtises, avoir un programme qui plante ou autre. Pour cela, il est possible d'ouvrir un gestionnaire de processus (équivalent du bon vieux Ctrl+Alt+Suppr sous Windows). Pour cela, on utilise généralement la commande `top` qui liste les processus les plus gourmands en ressources. Il est souvent utile de lister les processus qui nous appartiennent car il est en général impossible d'agir sur les autres. Pour cela, il faut utiliser l'option "-u" avec son nom d'utilisateur.

```
top - 11:29:32 up 5 days, 12:03, 4 users, load average: 1,30, 0,88, 0,90
Tasks: 448 total, 1 running, 447 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1,8 us, 1,1 sy, 0,0 ni, 97,1 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
MiB Mem : 31763,1 total, 17084,9 free, 8225,0 used, 6453,1 buff/cache
MiB Swap: 2048,0 total, 1191,2 free, 856,8 used. 21814,6 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2377227	mverot	20	0	1131,1g	216860	99108	S	16,9	0,7	13:31.52	brave
2362307	mverot	9	-11	1689300	16596	6720	S	5,0	0,1	8:25.29	pulseaudio
2362409	mverot	20	0	1950748	69960	43452	S	4,7	0,2	0:22.92	konsole
2375439	mverot	20	0	32,4g	39452	29644	S	4,3	0,1	8:34.79	brave
2362051	mverot	20	0	3771764	50736	35436	S	4,0	0,2	12:18.70	kwin_x11
2375432	mverot	20	0	33,1g	148328	85188	S	2,3	0,5	15:19.85	brave
56021	mverot	20	0	4139756	75580	1136	S	2,0	0,2	16:39.96	cryfs
2378253	mverot	20	0	1131,1g	822696	87644	S	1,0	2,5	1:48.56	brave
2378612	mverot	20	0	1131,1g	823212	88236	S	1,0	2,5	1:46.16	brave
2377907	mverot	20	0	32,9g	25844	24616	S	0,7	0,1	1:48.60	brave
2378797	mverot	20	0	1131,1g	824040	87796	S	0,7	2,5	1:46.24	brave
2379065	mverot	20	0	1131,1g	823844	87992	S	0,7	2,5	1:43.75	brave
2949277	mverot	20	0	1222456	161596	79952	S	0,7	0,5	1:54.29	inkscape
768457	mverot	20	0	16884	5840	4608	R	0,3	0,0	0:00.09	top

On peut alors voir l'utilisation du processeur (colonne "CPU") et de la RAM ("%MEM") pour chacun des processus. La colonne la plus utile est la colonne "PID" qui permet d'avoir l'identifiant du processus.

Une fois le processus fautif identifié, il est possible de le clore avec la commande `kill` suivie du nom du processus. En cas de processus récalcitrant à la commande précédente, l'option "-9" indiqué avant l'identifiant du processus fermera de manière plus agressive le processus.

```
$ kill 2377227
$ kill -9 2377227
```

### 3.11.2 Gérer l'exécution des processus dans le terminal

Par défaut, le lancement d'une commande bloque le terminal tant que le processus n'est pas fini. Si jamais la tâche dure longtemps ou que la commande ouvre une fenêtre, le terminal est alors inutilisable. Pour éviter ou contourner cela, il y a plusieurs moyens :

- Le premier est d'ajouter une esperluette "&" après la commande. Par exemple `$ gedit &` permet de lancer l'éditeur de texte gedit sans bloquer le terminal.
- `[ctrl]+[Z]` permet de suspendre le processus, on reprend la main sur le terminal, mais le programme est arrêté. Pour reprendre le processus mis en pause, il faut utiliser la commande `fg`.
- Pour arrêter le processus qui bloquait le terminal : `[ctrl]+[C]` (c'est l'équivalent du `kill` vu précédemment mais sans avoir besoin de chercher l'identifiant du processus)
- La commande `nohup` en tant que préfixe permet également de ne pas bloquer le terminal mais elle fait également en sorte que le processus tourne toujours y compris si le terminal est fermé.

## 3.12 Scripts et personnalisation

Jusqu'à présent, nous n'avons utilisé que des commandes qui tenaient sur une seule ligne, mais il peut être intéressant d'enchaîner des commandes sur un fichier (sans avoir besoin du pipe), utiliser des variables pour faire certaines opérations, etc. Pour cela, il est possible de créer des scripts en bash. L'extension correspondante est ".sh". Le fichier doit être exécutable et commencer par une ligne spéciale :

```
#!/bin/bash
```

Cette ligne est appelée "shebang" et permet d'indiquer au système le langage utilisé pour pouvoir l'interpréter correctement.

Il est possible de définir des variables avec le symbole "=" sans espace avant ni après. Pour les réutiliser, il faut ajouter un "\$" devant. Ainsi le script suivant enregistre le répertoire courant dans une variable "CURR\_DIR" (*current directory*), puis crée un répertoire de travail (*working directory*) indiqué dans la variable "WORK\_DIR", s'y déplace, y crée un fichier "testfile" avant de retourner dans le répertoire de départ.

```
CURR_DIR=$PWD
WORK_DIR=/scratch/mverot/
mkdir -p WORK_DIR
cd $WORK_DIR
touch testfile
cd $CURR_DIR
```

Il est possible d'utiliser des boucles sur des fichiers :

```
for f in *.png
do
```

```
echo $f
done
```

permettra ainsi d'afficher le nom de tous les fichiers ayant l'extension png (équivalent de `ls *.png`) .

Dans l'exemple ci-dessus, le fichier sans son extension est accessible avec la commande suivante :  `${f%.png}`

### 3.13 Éditeurs de texte : IDE versus les deux indémodables vi et emacs

Pour éditer un fichier, il existe différents utilitaires. Sous Linux, il existe `gedit` , `nano` et bien d'autres équivalents. Les options sont plus ou moins évoluées, avec éventuellement de la coloration syntaxique, des aides à la saisie, etc. Pour éditer des scripts pythons, les IDE Spyder, pyzo sont également légion. Chaque logiciel a ses avantages et inconvénients propre.

Au-delà de ces éditeurs spécialisés, il existe deux programmes phares pour l'édition sans interface graphique dans le terminal : `vi` / `vim` et `emacs` . Là encore, la préférence pour l'un ou l'autre est une histoire de querelle de chapelle. Emacs utilise la syntaxe LISP et demande en général abondamment d'utiliser la touche `[ctrl]` . Vi utilise pour sa part des modes capable de faire différents type d'édition mais il est possible de s'y perdre. Dans les deux cas, la maîtrise d'un seul de ces logiciel est suffisamment technique pour qu'on s'y consacre pleinement. Par contre, pour de l'édition de fichier, ils sont tous les deux généralement bien plus puissants que bon nombre d'IDE. IDE qui n'hésitent d'ailleurs pas à implémenter des fonctionnement directement inspirés de l'un ou l'autre pour permettre des modifications plus efficaces ou perfectionnées.

L'explication du fonctionnement de l'un ou l'autre de ces deux programmes est le sujet de nombreux livres, sites, tutoriels. Là encore, il faut s'en servir quotidiennement pour se perfectionner progressivement afin d'être capable de faire ce que l'on souhaite. La courbe d'apprentissage est raide mais extrêmement rentable sur le long terme pour ceux qui passeront beaucoup de temps à programmer ou lancer des simulations.

### 3.14 Ce qu'il faut retenir

- Les commandes données tableau 3.7
- Savoir qu'il faut éviter les espaces dans les noms de fichiers
- Savoir modifier le fichier .bashrc et le recharger avec la commande `source .bashrc` pour définir un alias ou modifier son PATH.
- Connaître le principe des passerelles pour les connexions en ssh.

Commande	Utilisation
<code>man</code>	Manuel de la commande
<code>pwd</code>	Emplacement dans l'arborescence pour donner le répertoire courant
<code>ls</code>	Lister les fichiers et les sous-dossiers, « -l » pour une version exhaustive, « -t » pour trier par date, « -r » pour inverser l'ordre, « -a » pour afficher les fichiers cachés
<code>cd</code>	Déplacement dans l'arborescence. « ../ » pour remonter d'un dossier, « ./ » dossier courant « / » racine, « ~ » home.
<code>mv</code>	déplacer ou renommer un fichier ou des répertoires
<code>cp</code>	copier un fichier ou des répertoires
<code>rm</code>	supprimer un fichier ou des répertoires
<code>mkdir</code>	créer un répertoire
<code>find</code>	trouver un fichier ou un dossier « -name » pour indiquer le nom du fichier
<code>chmod</code>	changer les droits sur un fichier, « +x » pour rendre exécutable
<code>grep</code>	trouver les lignes contenant une chaîne de caractère précise dans un fichier
<code>tail</code>	afficher la fin d'un fichier
<code>sed</code>	faire des recherches/remplacer dans un fichier
<code>zip</code> / <code>unzip</code>	compression et décompression au format zip
<code>tar</code>	compression et décompression au format tar.gz « -xvfz » pour l'extraction, « -cvzf » pour la création
<code>df -h</code>	connaître l'espace disque
<code>du -h</code>	connaître la taille d'un dossier
<code>ssh</code>	connexion à une machine distante
<code>exit</code>	fermeture d'une connexion ssh
<code>scp</code>	transfert de fichier d'un ordinateur à un autre
<code>top</code> / <code>kill</code>	lister les processus en cours d'exécution/tuer un processus
&	pour rendre le processus non bloquant
<code>[ctrl] + [C]</code>	arrêter le processus en cours d'exécution dans le terminal

TABLEAU 3.7 – Commandes de base à retenir.



# **Deuxième partie**

## **Programmation en python**



# Chapitre 4

## Avant de se lancer

La programmation demande de se forger des habitudes. Ces habitudes sont généralement dictées par le bon sens et l'expérience de programmeurs confirmés. Elles peuvent sembler pénibles et contraignantes initialement, mais elles sont là pour faciliter les choses sur le long terme.

Lors de la programmation, il existe plusieurs phases :

1. la phase de programmation initiale lors de laquelle on développe son code pour correspondre à nos besoins ;
2. les phases de test et débogage qui ont lieu lors de la phase de programmation ;
3. la phase de polissage du code : rédaction des commentaires, de la documentation, renommage des variables, etc ;
4. la phase d'utilisation initiale ;
5. la phase d'extension lors de laquelle on a tendance à ajouter de la complexité au programme initial pour lui ajouter des fonctionnalités
6. la phase de maintien du code pour le maintenir fonctionnel sur le long terme

Les phases qui sont souvent le plus chronophages sont celles de débogage et de maintien du code ou de légère amélioration. Les conseils qui vont suivre viseront à les rendre moins pénibles.

### 4.1 Décortiquer un problème avant de commencer

💡 La liste qui suit n'est pas une obligation mais elle donne des pistes sur la manière d'attaquer un problème.

- Avant tout, il faut commencer par chercher si quelqu'un n'a pas déjà fait ce que vous cherchez à faire. Dans le meilleur des cas, vous aurez économisé quelques heures à programmer, la personne qui l'aura fait l'aura mieux fait que vous et vous pourrez passer directement à autre chose. Au pire, vous aurez vu des bouts de code ou cela vous aura donné des idées sur ce que votre programme devrait faire.
- Ensuite, faites-vous une petite liste de ce que vous voulez faire, cela n'a pas forcément besoin d'être écrit proprement quelque part. Ça peut être griffonné sur un bout de papier ou même rester dans votre tête.
- Laissez mijoter un peu votre idée, le temps de laisser les choses décanter, affiner vos besoins, penser à une fonctionnalité intéressante que vous auriez pu oublier.

- Une fois le besoin exprimé, fractionnez-le en une succession d'étapes simples de manière procédurale. Réfléchissez à comment vous feriez les choses si vous aviez à les faire à la main. Inutile de chercher à optimiser les choses.
- Pour chacune des étapes, cherchez de l'aide sur internet pour résoudre le problème (chatGPT, stackoverflow, vos anciens scripts, vos amis, vos collègues sont vos meilleurs amis pour vous guider!)
- Normalement, à ce stade, vous allez pouvoir commencer à programmer.

Rappelez-vous : la programmation est d'autant plus facile que vous aurez de l'expérience. Donc commencez par programmer des choses simples, qui vous font envie, et surtout forcez-vous à programmer !

## 4.2 ▲ Nommage des variables

**Pour les noms de variable, privilégiez des noms explicites.**

Avoir des variables qui ont des noms totalement abstrait :

- rend le code illisible pour un relecteur ;
- augmente la charge cognitive du programmeur ;
- augmente la probabilité de finir par utiliser deux fois le même nom de variable pour deux choses différentes (ce qui mène à des catastrophes !)

Tant que la compacité du code n'est pas une exigence, il est toujours préférable de rendre vos variables explicites. Par contre, il faut faire attention à ne pas tomber dans l'excès inverse pour que le nom des variables puisse toujours s'appliquer malgré les évolutions du code.

Pour le nommage, il existe plusieurs conventions, le plus important est d'essayer de maintenir la même convention de nommage au sein d'un même programme.

💡 Normalement, pour les variables et les fonctions, il est recommandé d'utiliser des minuscules avec des "\_" pour séparer les mots.

💡 Essayez d'utiliser des noms de variable en anglais : si jamais vous partagez votre code, un russe a beaucoup moins de chance de comprendre votre code s'il est écrit avec des noms de variable français plutôt qu'en anglais.

💡 N'utilisez JAMAIS de caractères accentués dans vos noms de variable : un américain (ou un chinois) n'arrivera pas à taper un "é" sur son clavier.

## 4.3 Trouver de l'aide

### 4.3.1 Le facile : l'appel à un ami

Pour trouver de l'aide, cela peut tout à fait commencer par une recherche internet. En général, il faut mieux taper sa question en anglais pour augmenter la probabilité d'avoir une réponse. Les forums comme stackoverflow permettent souvent d'avoir des propositions de réponse ou des solution approchantes. Cependant, attention aux copier-coller trop hâtifs : ils faut prendre le temps de comprendre la ou les solutions proposées. De plus, il peut arriver que la solution ou la question ne corresponde pas exactement à ce que vous souhaitez faire. Dans ces cas là, inutile de jeter le bébé avec l'eau du bain. Cela peut constituer une piste de départ et permettre d'affiner le nom de votre recherche avec le nom d'une fonction.

Vous pouvez également trouver de nombreux tutoriels sur l'utilisation de certaines librairies ou fonctions. Cela nécessite par contre d'avoir une idée de la fonction ou librairie à utiliser. N'hésitez pas à vous en servir.

Il est maintenant également possible de demander à une intelligence artificielle d'écrire un morceau de code à votre place. Même si ce n'est pas forcément une solution miracle (le code peut buguer d'entrée). Cela peut vous aider à construire un morceau de code ou vous donner des pistes.

### 4.3.2 Le moins plaisant (mais plus complet) : « RTFM »

Lorsque les solutions précédentes ne suffisent pas (voire tout le temps), il faut également en passer par la case certes moins agréable mais néanmoins nécessaire de la lecture du manuel ou de la documentation. Comme en section 3.1, l'adage « RTFM » pour *Read That Fucking Manual* est de rigueur. La documentation est la première source d'information. C'est elle qui fait référence et souvent, c'est là que vous trouverez les options miracles des différentes fonctions pour faire en sorte de faire exactement ce que vous souhaitez.

Si la lecture du manuel peut parfois être aride, surtout pour un néophyte vu qu'il peut y avoir des termes très abscons. C'est une étape essentielle pour pouvoir progresser.

Pour les plus grosses librairies, le manuel peut se trouver en ligne sur un site dédié. Mais pour les librairies plus confidentielles, il peut être nécessaire d'aller lire la documentation ou y accéder dans le code source. Certaines librairies peuvent vous y aider.

### 4.3.3 Le pénible : consulter le code source

Même si je ne le souhaite à personne, il peut arriver d'avoir à aller lire dans le code source directement du code non documenté. Si jamais ça vous arrive, posez-vous la question de savoir si vous avez vraiment besoin de le faire (car normalement, tout code doit être commenté, voir section 4.5). C'est en général un mauvais signe de la part de la personne qui a conçu le programme.

Mais quand il n'y a pas le choix... Il faut alors vous appropier le code de quelqu'un d'autre à la volé, vous verrez que l'exercice est généralement extrêmement fastidieux et chronophage et reviens presque à repartir de zéro.

## 4.4 Déboguer un programme

Un bon programmeur est avant tout quelqu'un capable de tester et déboguer son programme. Un programme qui ne fonctionne pas est un programme inutile. Aussi bien pour l'utilisateur que pour le programmeur (et vous allez souvent être dans les deux rôles!).

Cependant, un des énormes avantages de la programmation est de pouvoir tester généralement à peu de frais son code en condition réelles ou approchant. Il est **indispensable** de tester son code. Il faut cependant avoir certains réflexes pour rendre l'opération plus facile.

Il existe toute un formalisme associé à la manière de concevoir un programme pour le déboguer au mieux : les tests unitaires. Cependant, n'étant pas informaticiens de métier, nous allons aborder les choses sous un angle plus pragmatique et simpliste.

### 4.4.1 Compartimenter/factoriser

Une des manières les plus efficaces pour déboguer un programme est de concevoir et mettre au point de petits blocs « indépendants ». Cela va généralement de pair avec la partie 4.1. On peut en général déboguer chaque sous-partie du problème exprimé.

De plus, une des très bonnes manières de compartimenter les choses est de créer des fonctions. En effet, chaque fonction constitue un bloc indépendant. De plus, si elles sont bien créées, les variables d'entrée sont naturellement listées. Et pour finir, cela permet de mettre à jour et déboguer son code à un unique endroit plutôt qu'à tous les endroits où on utilise un code identique.

#### 4.4.2 Lire les messages d'erreur

Pour tous les bugs qui affichent une erreur explicite, les messages d'erreurs permettent généralement de faciliter les choses. En effet, ils affichent généralement la chaîne qui a déclenché l'erreur. Il faut alors aller chercher la ligne qui est indiquée. De plus, un copier coller du message d'erreur sur internet pourra permettre de l'expliciter en langage courant.

 Il arrive que la ligne d'erreur indiquée soit la mauvaise à quelques lignes près, par exemple en cas d'oubli de parenthèse. Si jamais vous ne voyez pas d'erreur à l'endroit indiqué, n'hésitez pas à remonter de quelques lignes.

#### 4.4.3 Commenter son code (avec du texte)

On verra bientôt l'intérêt de commenter son code (section 4.5). Mais dès à présent : un code bien commenté vous indiquera les points sensibles et sources d'erreur potentielles. C'est un levier très puissant pour indiquer comment vous avez géré des particularités au moment de l'écriture initiale du code. Un code bien commenté indiquera ce qu'il ne faut pas faire ou à quoi sert une ligne de code qui semble inutile ou abstraite.<sup>a</sup>

#### 4.4.4 Afficher ses variables et leur type

En cas de bug, il est souvent utile d'afficher le contenu des variables. Cela peut souvent permettre de voir leur contenu, quelle modification peut les avoir affecté. De même, leur type peut parfois permettre de voir s'il y a des incompatibilité d'opérations sur des objets ayant le mauvais type.

#### 4.4.5 Revenir en arrière jusqu'à revenir à un programme fonctionnel

Pour trouver la source de l'erreur, il peut être judicieux de commenter des morceaux de code pour désactiver temporairement les lignes ajoutées. Si jamais vous avez écrit beaucoup de ligne d'un coup, cela permettra de restreindre la partie à déboguer. De plus en commentant sélectivement quelques parties de votre code, cela permettra de circonscrire la zone à corriger. Commencez par commenter largement, puis réduisez petit à petit la zone commentée pour en venir au nœud du problème progressivement.

#### 4.4.6 Faire un appel à l'aide

En cas d'impossibilité à résoudre son bug, il faut savoir se résoudre à demander de l'aide. Si jamais vous avez quelqu'un avec les bonnes compétences sous la main (ou si vous comptez faire appel à un forum), il faut tout de même montrer patte blanche et que vous avez vous-même fait des efforts pour déboguer votre programme.

---

a. [Un exemple](#) avec une faille dans le protocole SSL.

Il faut commencer par lister ce que vous souhaitez faire, puis proposer un code minimal qui ne fonctionne pas (*Minimal (Not) Working Example*). Cela commencera par montrer vos efforts, de plus cela indiquera rapidement ce qu'il faut regarder et surtout, cela facilitera l'appropriation de votre code.

 Dites vous que lire le code de quelqu'un d'autre, c'est à peu près quatre fois plus difficile que de lire le vôtre. En effet, il faut comprendre l'intention, la méthode, la syntaxe et le cheminement. Si des gens prennent beaucoup de temps pour vous aider, le minimum est donc de prendre soin de leur gentillesse.

## 4.5 Commenter son code

Une des plus grosses différences entre un programmeur débutant et un programmeur plus expérimenté sera le nombre et la qualité des commentaires laissés au sein du code. En effet, parmi les différentes étapes listées en début de chapitre 4, un débutant passera surtout du temps sur la première phase : rédiger un code qui fait ce qu'il souhaite. Un programmeur plus expérimenté passera pour sa part beaucoup plus de temps sur la troisième phase qui consiste à polir son code.

En effet, bien commenter son code permet de gagner ÉNORMÉMENT de temps sur les deux dernières phases de vie du programme. Avec l'expérience, on apprend généralement avec grande peine que ce sont les phases qui sont les plus chronophages. Se replonger dans un code demande un effort qui peut parfois être proche de celui nécessaire pour repartir de zéro.

Il est donc **crucial** de commenter son code. En particulier :

- toutes les petites astuces qui vous ont pris du temps (corrections d'indice, tri, fonction trouvée sur le net, etc)
- les fonctions : pour cela, la documentation doit être placée entre triple double quote « """ » il faut en préciser le but (en une ligne), les arguments et leur type, ainsi que donner l'argument retourné par la fonction
- indiquer le rôle des différentes portions de code
- préciser l'origine d'un code emprunté (lien, article, etc) pour pouvoir retrouver l'accès à des précisions supplémentaires
- écrire le code au fur et à mesure : cette activité est longue mais nécessaire. Elle est cependant d'autant plus facile à accomplir qu'elle est faite au fur et à mesure.

Cependant, il est tout aussi important de :

- garder des commentaires en adéquation avec le code, y compris lors de modifications : des commentaires erronés peuvent être plus dommageables qu'aucun commentaire du tout.
- être simple et concis : il est rarement utile de commenter des portions de code triviales ou compréhensibles. Tout comme il faut aller au plus simple pour que le lecteur comprenne le plus vite possible votre intention.

Pour un logiciel complet, il peut aussi être utile de mettre en place une vraie documentation rédigée pour expliquer plus en détail certains points du code qui sont trop complexes pour être expliqués dans le code.

## 4.6 Structure d'un script python

Pour toute la suite du cours, on utilisera systématiquement la structure suivante pour les programmes utilisés :

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
5 Descriptif du fichier
"""

# Importation des librairies

10 # Definition des fonctions

# Programme principal
if __name__ == "__main__":
    pass

```

- La première ligne `#!/usr/bin/env python3` est le shebang : c'est la ligne où vous indiquez à l'ordinateur le langage qu'il doit utiliser pour interpréter votre script. Cette ligne est **cruciale** pour que votre script soit correctement exécutée. C'est ce qui permet à votre script d'être exécuté en tapant `./script.py` au lieu de `python3 script.py`.
- La description du fichier est indispensable pour avoir une idée globale du but du programme c'est aussi l'occasion d'y mettre une licence, les sources du code utilisé, etc.
- Les librairies doivent être importées au début du fichier afin d'être accessibles à tout instant dans votre programme, y compris au sein des fonctions.
- Les fonctions doivent également être définies au début de votre programme, après l'import des librairies pour être ré-utilisables aussi bien au sein de votre programme que depuis un autre programme.
- La ligne `if __name__ == "__main__":` est tout aussi importante : elle permet de faire en sorte de n'exécuter le code principal que si vous exécutez le script directement (en tapant `./script.py` par exemple). Ainsi, le programme principal du script ne sera PAS exécuté si vous faites un simple import de fonction. Un exemple avec et sans utilisation de la ligne `if __name__ == "__main__":` est disponible sur [github](#) dans le fichier `ifname.zip`. Lors de l'exécution du fichier `call_lib_withorwithout_ifname.py` deux librairies sont importées : une avec et l'autre sans cette ligne. Avec la ligne, lors de l'import, il n'y a aucun code d'exécuté alors que c'est le cas si la ligne n'a pas été écrite.

## 4.7 Ce qu'il faut retenir

- Savoir découper un problème complexe en superposition de problèmes plus simples. Il faut donc toujours prendre le temps de planifier son programme pour le découper en morceaux plus simples.
- Utiliser des fonctions est un excellent moyen de compartimenter son code.
- Toutes les variables nécessaires à une fonction doivent être appelées lors de l'exécution de la fonction.
- Pour nommer une variable, elle doit être explicite, suivre une convention de nommage et être utilisable par le plus grand nombre (pas d'accents ni de noms français !)
- Il faut prendre le temps de déboguer un code régulièrement.

- Apprendre à lire les erreurs qui empêchent l'exécution d'un script.
- Pour déboguer : penser à afficher les variables, leur type, commenter des portions de code.
- Il est **indispensable de commenter son code**. En particulier les fonctions.



# Chapitre 5

## Les bases

En complément du polycopié, des cahiers jupyter interactifs pour montrer certains points du polycopié sont mis à disposition. À chaque fois qu'un concept est illustré dans le cahier jupyter, une icône  qui sera un lien cliquable sera présent pour voir le concept illustré en pratique.

### 5.1 Types de base

Python est un langage typé mais qui utilise une déclaration de type implicite. Ainsi, il ne sera pas possible de convertir directement un entier en nombre flottant sans explicitement le demander. Il faut donc également faire très attention lors de l'initialisation ou de la déclaration des variables sous peine de créer un objet qui n'aura pas le comportement attendu.

#### 5.1.1 Nombres

Pour les nombres, il y a essentiellement deux types couramment utilisés : les flottants et les entiers  . Python accepte la surcharge d'opérateur donc sera capable de faire des opérations comme la division pour les deux types. Cependant, si on veut faire une division entière pour obtenir uniquement le quotient, il faut utiliser l'opérateur `//`  . Le reste peut être obtenu avec l'opérateur `%`.

De plus, python gère très bien la notation scientifique, ainsi il est possible de définir le rayon de Bohr  $a_0 = 5,2917721 \cdot 10^{-11}$  m avec l'expression suivante : `a0 = 5.2917721e-11`.

##### 5.1.1.1 Représentation flottante, comparaison et précision

**Ce paragraphe est extrêmement important pour comprendre certains comportements qui peuvent sembler totalement erratiques si l'on n'est pas conscient du problème.**

Les nombres flottants sont stockés physiquement sous une forme particulière dans la mémoire qui engendre des « problèmes d'arrondis ». Ainsi, si on demande à python si  $0,9$  est égal à  $0,3 + 0,3 + 0,3$ , sa réponse est .. NON!  Une explication détaillée est fournie dans la documentation python : [Floating Point Arithmetic: Issues and Limitations](#).

Le problème n'est pas lié à python mais à la manière dont les nombres sont stockés en mémoire. Ils correspondent à des fractions mais en base binaire au lieu d'utiliser une base décimale. Il est **indispensable** d'en être conscient pour éviter d'avoir de mauvaises surprises à certains moments. Les modules `decimal` et `fractions` peuvent permettre d'identifier et corriger les erreurs liées. 

Si les écarts peuvent sembler anodins, les calculs scientifiques effectuent en général de très nombreuses opérations qui peuvent aboutir à des résultats pour lesquels l'écart entre la valeur théorique et la valeur calculée peut devenir importante. Les librairies `numpy` et `scipy` ont été implémentées de manière à minimiser et pouvoir fournir une estimation des erreurs numériques. Erreurs qui peuvent survenir à cause du calcul en nombre flottant.

De plus, si possible, il faut mieux *tester des inégalités que des égalités* pour éviter d'avoir des conditions non satisfaites.

### 5.1.2 Chaînes de caractère

Python accepte encore une fois la surcharge d'opérateur et accepte l'opérateur `+` pour les chaînes de caractère. Cela peut avoir des conséquences étranges en cas de confusion entre chaîne de caractère correspondant à un nombre et vrai nombre flottant. ☀ Cependant, python génère une erreur de compilation si on mélange les deux types pour limiter la casse.

**Formatage** Les chaînes de caractère sont le plus souvent utilisées pour afficher leur contenu ou le contenu de certaines variables. Python offre de très nombreuses options. Les plus courantes sont listées dans le tableau 5.1. Il est généralement préférable d'utiliser la fonction `format` qui est dédiée au fait de formater correctement des variables plutôt que de recourir à des fonctions ou artifices alambiqués qui feront moins bien que cette dernière. <sup>a</sup>

Option	Résultat	Commentaire
<code>"{} {}".format('a',3.14)</code>	a 3.14	les variables sont placées séquentiellement
<code>"{1} {0}".format('a',3.14)</code>	3.14 a	les variables sont placées en fonction du nombre indiqué entre accolades
<code>"{truc} {bidule}".format(truc='a',bidule=3.14)</code>	3.14 a	les variables à placer sont nommées
<code>"{:&lt;10}" .format('blabla')</code>	blabla_____	justifié à gauche sur (au moins) 10 caractères
<code>"{:&gt;10}" .format('blabla')</code>	_____blabla	justifié à droite sur (au moins) 10 caractères
<code>"{:d}" .format(1)</code>	1	Pour les entiers
<code>"{:3d}" .format(1)</code>	__1	... sur (au moins) 3 caractères
<code>"{:03d}" .format(1)</code>	001	en complétant par des zéros si nécessaire
<code>"{:f}" .format(3.14)</code>	3.140000	fixed-point number, 6 chiffres après la virgule (par défaut)
<code>"{:,.3f}" .format(3.14)</code>	3.140	3 chiffres après la virgule
<code>"{:10.3f}" .format(3.14)</code>	_____3.140	sur (au moins) 10 caractères
<code>"{:e}" .format(3.14)</code>	3.140000e+00	notation scientifique, 6 chiffres après la virgule (par défaut)
<code>"{:,.3e}" .format(3.14)</code>	3.140e+00	3 chiffres après la virgule
<code>"{:10.3e}" .format(3.14)</code>	_____3.140e+00	sur (au moins) 10 caractères

TABLEAU 5.1 – Quelques option de formatage pour mettre sous forme de chaîne de caractère le contenu de variables. ☀ Documentation officielle.

Il existe également l'option d'utiliser `f""` pour insérer directement des variables `f"la variable truc vaut {truc} et bidule vaut {bidule}"`. Cette option est moins verbeuse mais ne propose pas autant de contrôle que la fonction `format`. Il est également possible de faire en

a. Ou à un enchaînement stérile de guillemets et symboles « + » – qui en plus de cela sont peu souples.

sorte de ne pas avoir à échapper les backslash \ en utilisant `r'''`. Cela est utile pour taper du LaTeX plus facilement.

Pour utiliser les caractères « » et « » en même temps que la fonction `format`, il faut répéter le caractère correspondant. Il est également possible d'aller à la ligne dans une chaîne de caractère en utilisant le caractère « \ ».

### 5.1.3 Listes

Les listes sont numérotées en commençant par un élément d'indice 0. Elles sont ordonnées et mutables. Il est possible d'utiliser plusieurs méthodes pour modifier ou manipuler les listes (5.2)

Méthode	Effet
<code>l.append(x)</code>	Ajoute l'élément <code>x</code> à la fin de la liste
<code>l.extend(iterable)</code>	Ajoute chacun des éléments de <code>iterable</code> à la fin de la liste
<code>l.insert(i, x)</code>	Ajoute <code>x</code> en position <code>i</code> à la liste
<code>l.pop([i])</code>	Enlève l'élément à la position <code>i</code> , ou le dernier élément de la liste si aucun argument n'a été indiqué
<code>l.remove(x)</code>	Enlève le <i>premier</i> élément égal à <code>x</code> dans la liste
<code>l.clear()</code>	Enlève tous les éléments de la liste
<code>l.index(x[, start[, end]])</code>	Retourne l'indice du premier élément de la liste égal à <code>x</code> . <code>start, end</code> servent à délimiter les bornes des indices cherchés
<code>l.count(x)</code>	Renvoie le nombre d'éléments de la liste. (équivalent à <code>len(l)</code> )
<code>l.sort(*, key=None, reverse=False)</code>	Trie les éléments de la liste, <code>key</code> permet d'indiquer une fonction à utiliser avant la comparaison
<code>l.reverse()</code>	Intervertir les éléments de la liste
<code>l.copy()</code>	Effectue une copie « shallow » (superficielle) de la liste

TABLEAU 5.2 – Méthodes utilisables sur les listes. ☰

### 5.1.4 Tuples

Les tuples correspondent à des listes non modifiables, une fois déclarés, ils ne peuvent donc plus être changés. Alors qu'une liste vide peut être initialisée avec des crochets [], un tuple est déclaré avec des accolades (). Les tuples servent essentiellement pour stocker des listes de variables qui doivent impérativement rester inchangées au cours du temps dans le programme.

En pratique, il est possible de modifier le contenu d'un tuple si celui-ci contient un objet mutable (voir 5.2). Il est donc déconseillé d'utiliser des objets mutables au sein de tuples pour éviter ce comportement qui est rarement voulu pour un tuple.

### 5.1.5 Dictionnaires

Les dictionnaires permettent d'associer une « clé » (`key`) à une « valeur » (`value`). Les clés sont uniques. Alors qu'une liste vide peut être initialisée avec des crochets [], un dictionnaire

vide est déclaré avec des accolades {}.

Pour créer des listes de dictionnaires, il faut toujours privilégier le fait de faire des structures homogènes et reproductibles pour pouvoir à chacune des valeurs associées à une clé. ☺. De même, les clés doivent à priori être données en anglais et ne doivent pas contenir de caractères non ASCII – dans les deux cas pour un souci de portabilité du code.

**À partir de python 3.7, les dictionnaires deviennent des objets ordonnés** (la liste de clé est toujours dans le même ordre) alors que ce n'est PAS le cas dans les versions précédentes.

### 5.1.6 Fonctions

Pour rappel (voir 4), une fonction se doit de :

- avoir un nom explicite, sans caractère spécial
- toujours être commentée : but, arguments d'entrée, valeur de retour ;
- avoir une valeur de retour autant que possible

Une fonction commence toujours avec le mot clé def ensuite, il faut indiquer le nom de la fonction puis entre parenthèse les arguments de la fonction.

💡 **Il faut toujours passer EXPLICITEMENT les arguments d'une fonction à cette dernière.** Sinon, un jour, ça se passera mal, voire très mal, voire très très mal. (section 5.3.1)

💡 Toutes les variables internes d'une fonction n'existent que lors de l'exécution de la fonction. Il est donc possible d'utiliser le même nom de variables dans des fonctions différentes.

**5.1.6.0.1 Paramètres, arguments optionnels, args, kwargs** Il est possible de rendre certains arguments optionnels. Ils sont précisés après les arguments positionnels. Dans ce cas, dans la définition il faut les définir avec un nom explicite et fournir une valeur par défaut (voir section 5.2.2.2 pour les objets mutables). ☺

```
def func(x,y,z=0,liste = None, extended = True):
    __pass
```

Si un utilisateur souhaite changer une des valeurs par défaut, il pourra le faire en précisant la variable qu'il souhaite changer. Si le nom n'est pas précisé, les variables sont prises dans l'ordre de la déclaration de la fonction.

💡 En python, on passe les paramètres par référence, cela veut dire qu'une fonction peut, en son sein modifier les arguments qui lui ont été fournis – si ceux-ci sont mutables (voir section 5.2). Il faut donc faire attention à la manière dont sont manipulés les arguments au sein des fonctions pour éviter d'avoir des comportements non désirés (voir section 5.2.2.1).

Dans la documentation, il est fréquent de trouver dans les définitions des fonctions des \*args OU \*\*kwargs

- \*args correspond à un ensemble d'arguments optionnels mais non nommés ;
- \*\*kwargs correspond à un ensemble d'arguments optionnels mais nommés ;

L'avantage de ces deux solutions est de pouvoir rendre plus souple la définition de vos fonctions.

**5.1.6.0.2 Les fonctions anonymes ou fonctions lambda** Les fonctions lambda sont des fonctions qui n'ont pas de nom. Elles sont utiles lorsqu'elles servent d'argument à une fonction qui attend une fonction et qu'elles sont trop peu utilisées pour mériter un nom propre.

Le code suivant :

```
my_Func = lambda a, b : a * b
```

est équivalent à :

```
def my_Func(a,b):
    """
    Fonction qui retourne le produit des deux arguments
4    """
    return a * b
```

En pratique, on utilise la fonction anonyme dans des cas où il n'est justement pas nécessaire de lui donner de nom, contrairement à ce qui a été fait dans l'exemple ci-dessus. Mais sinon, il n'aurait pas été possible de lui donner un code équivalent avec une définition de fonction classique. Il est à noter l'absence du mot clé `return` dans la fonction `lambda`.

**5.1.6.0.3 Packing et unpacking** Lorsqu'une fonction retourne une liste, il est possible de faire du « *unpacking* » : cela consiste à affecter directement les variables en tant qu'élément de liste.

Il est possible de ne garder qu'un sous-ensemble d'arguments en « jetant » les éléments non souhaités avec le caractère `_` ☺.

De même, il est possible de transformer directement une liste en suite d'arguments pour une fonction grâce au splat `*`.

Ce mécanisme est utilisé pour les `*args` et `**kwargs` que l'on peut voir régulièrement dans la documentation des fonctions (cf paragraphe 5.1.6.0.1)

## 5.2 ♦ Objets mutables et immutables

L'explication qui suit peut être complétée par la lecture de cet article : [Python's Mutable vs Immutable Types: What's the Difference?](#)

Pour le langage Python, il existe deux types d'objets : les objets mutables et les objets non mutables.

- un élément mutable est un objet qui peut être changé après sa création tout en gardant la même adresse mémoire ;
- un élément immutable est un objet qui ne peut pas être changé sans changer l'adresse mémoire dans laquelle il est stocké ;

**Différence entre variables et objets** Une variable n'a pas intrinsèquement de type, mais c'est l'objet vers lequel elle pointe qui en a un. En effet une variable ne fait que pointer en mémoire vers où est stocké l'objet auquel elle fait référence (figure 5.1).



FIGURE 5.1 – Une variable ne fait que pointer vers un objet qui a une adresse mémoire. C'est l'objet référencé qui impose le type de la variable.

**Objet immutable et changement de variable** pour un objet immutable, si on en change le contenu, on va en fait avoir créé un nouvel objet avec une nouvelle adresse mémoire et la variable va maintenant pointer vers cette nouvelle référence d'adresse mémoire (figure 5.2).

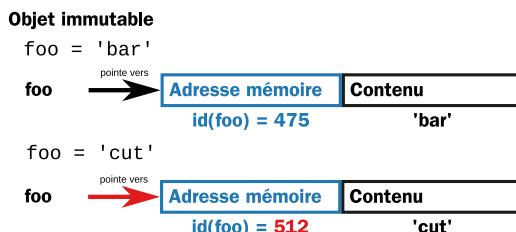


FIGURE 5.2 – Pour un objet immuable, en changer le contenu correspond en fait à 1) créer un nouvel objet en mémoire avec le nouveau contenu 2) mettre à jour la référence de la variable pour pointer vers l'objet que l'on vient de créer.

**Objet mutable et changement de variable** Pour les objets mutables, il est possible d'en changer le contenu sans que celui-ci change d'adresse en mémoire. En effet, on change en général une référence interne à une adresse mémoire sans avoir à créer de nouvel objet pour la variable (figure 5.3).

La différence vient donc de la référence vers l'adresse mémoire changée : pour un objet immuable c'est directement le lien avec la variable, pour un objet mutable, il s'agit d'une référence interne.

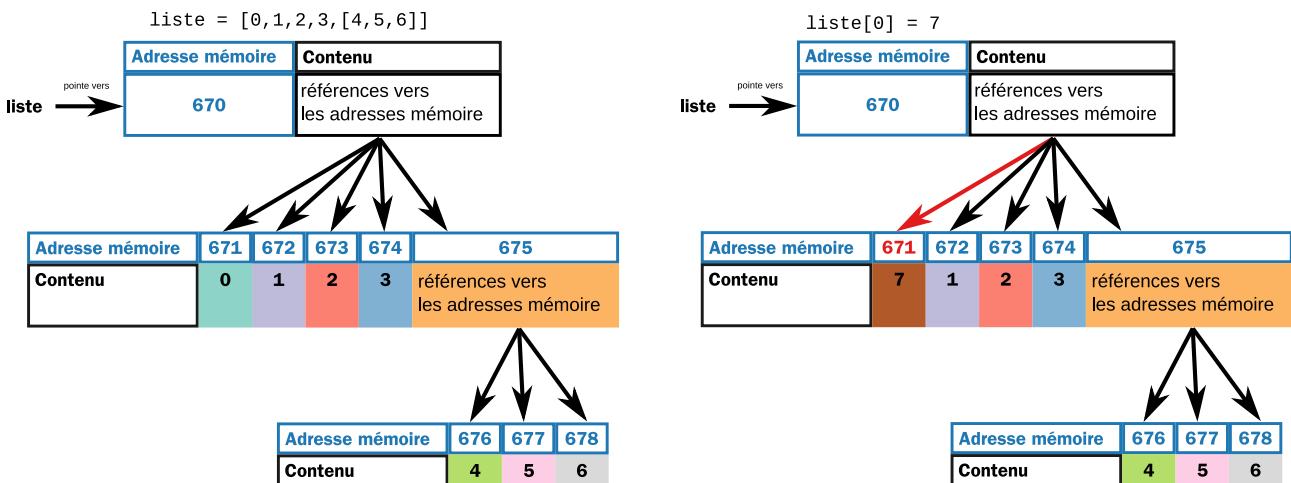
**Objet mutable**

FIGURE 5.3 – Pour un objet mutable, en changer le contenu correspond en fait à 1) créer un nouvel objet en mémoire avec le nouveau contenu 2) mettre à jour la référence en interne.

Le tableau 5.3 indique quelles catégories de variables sont mutables ou non en python. On peut voir que les variables mutables sont en général des objets contenant des objets imbriqués.

Immutable	Mutable
<code>int, float, complex</code>	<code>list, dict</code>
<code>string</code>	<code>ndarray (numpy)</code>
<code>bool</code>	<code>Dataframe (pandas)</code>
<code>frozenset</code>	<code>set</code>
<code>bytes</code>	<code>bytearray</code>
<code>tuples*</code>	

TABLEAU 5.3 – Variables mutables et immutables. \*Les tuples sont immutables mais seulement s'ils ne contiennent pas d'éléments mutables.

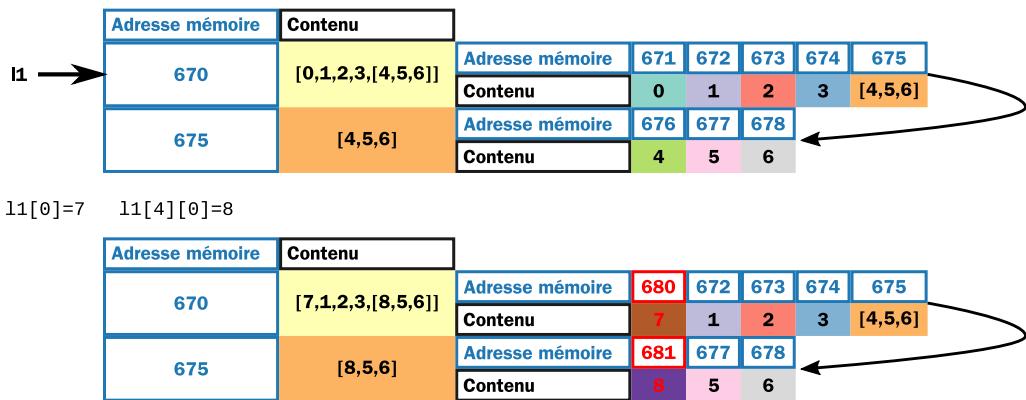
### 5.2.1 ◆ Première conséquence : copie d'objets mutables/imbriqués : « *shallow copy* » versus « *deep copy* »

Pour des éléments imbriqués (liste, tuple, dictionnaire, ndarray, etc), ils sont stockés à un endroit de la mémoire et ces éléments pointent sur chacun des éléments qu'ils contiennent.



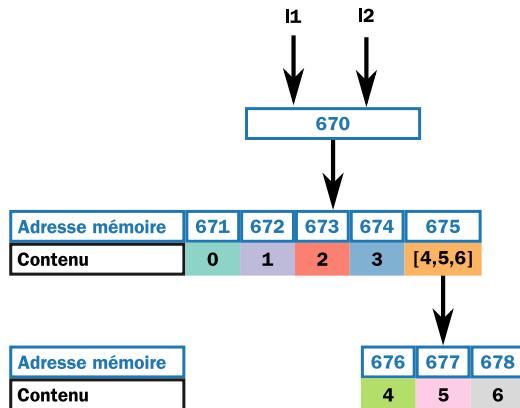
Sur l'exemple 5.4, la liste 1 contient une suite d'adresses (671 à 675) et chacune de ces adresses contient la valeur (soit les entiers 0,1,2,3, soit une liste d'entier [4,5,6]). Lors de la modification d'un élément de liste, on change le contenu de l'adresse mémoire en interne pour pointer vers un nouvel objet, mais uniquement cela. La liste pointe toujours vers la même adresse mémoire.

Dans le cas d'une *shallow copy* (copie superficielle), la copie se fait directement au niveau de l'adresse mémoire principale. Comme les deux variables correspondent à la même adresse mémoire, la modification d'une liste correspond automatiquement à la modification de l'autre également !



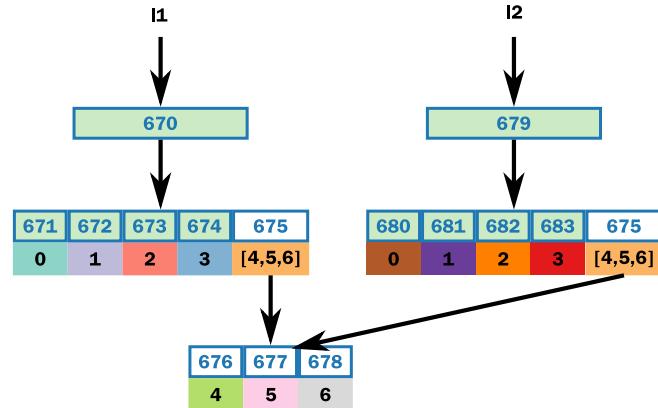
Changer un élément de liste revient à changer une case colorée sans changer l'adresse mémoire vers laquelle pointe la liste.

**Shallow copy :**  $l2 = l1$



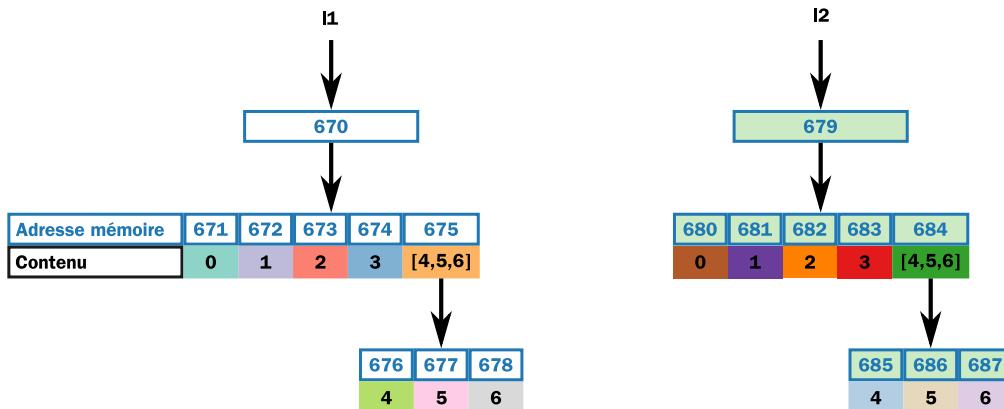
Modifier une liste modifie l'autre car  
on utilise dans les deux cas les mêmes adresses mémoire

**Copie moins shallow**  $l2 = l1.copy()$



Modifier une liste ne modifie pas l'autre (au premier rang d'imbrication uniquement)  
car toutes les cases vertes créées lors de la copie sont indépendantes

**Deep copy :**  $l2 = \text{copy.deepcopy}(l1)$



Les deux listes sont totalement indépendantes

**FIGURE 5.4 – Mécanisme de stockage des listes et profondeur de copie.**

Dans le cas d'une copie avec la méthode `.copy()` on pointe vers deux adresses différentes donc les listes peuvent être modifiées indépendamment. De plus, les références de premier niveau sont différentes et peuvent donc être modifiées indépendamment.

Dans le cas d'une copie avec la fonction `deepcopy()` de la librairie `copy` tous les éléments sont totalement indépendants. La modification d'une liste à un niveau quelconque est totalement indépendante et ne modifie donc pas l'autre liste. Par contre, on occupe... deux fois plus d'espace en mémoire.<sup>b</sup>

## 5.2.2 🟥 Deuxième conséquence : éléments mutables et fonctions

### 5.2.2.1 🟥 Modification d'un élément mutable au sein d'une fonction

Si un des arguments d'une fonction est mutable, alors le modifier au sein de la fonction le modifie également en dehors de la fonction. Pour éviter cela, il faut faire une copie ou créer un nouvel élément mutable au sein de la fonction. ☺ Pour un objet immuable, le changer au sein de la fonction n'a pas d'influence en dehors de celle-ci.

### 5.2.2.2 🟥 Élément mutable comme argument optionnel

Si on utilise un élément mutable comme argument optionnel, alors à chaque appel de la fonction, ce sera l'élément mutable optionnel créé lors de la définition de la fonction qui sera changé. ☺ Ce comportement peut être souhaité.. ou pas. Pour l'éviter, il faut initialiser la variable à `None`, puis après un test, créer la variable mutable lors de l'exécution de la fonction.

## 5.3 Espaces de nommage

### 5.3.1 🟥 les différents espaces de nommage

Les espaces de nommage correspondent à des lieux abstraits dans lesquels les variables sont accessibles. Ils permettent également de trancher quel variable sera utilisée en cas d'homonymie (portée). Enfin, ils délimitent une temporalité d'existence des variables lors de l'exécution d'un programme.

Dans python, il existe quatre espaces de nommage principaux (*namespaces*) qui sont tous imbriqués « en poupée russe » les uns dans les autres :

- L'espace de nommage **built-in** est celui qui est accessible partout et tout le temps. C'est celui dans lequel les fonctions, constantes, types et exceptions de python sont définies. Il est possible d'avoir leur liste avec la commande `dir(__builtins__)`. `len`, `float`, `True` appartiennent à cet espace de nommage par exemple.
- L'espace de nommage **global** qui contient les variables et fonctions accessibles à l'ensemble du programme. Il est possible de voir le contenu avec la commande `globals()`.
- L'espace de nommage **local** qui correspondent aux variables accessibles au sein d'une fonction. L'interpréteur créé un nouvel espace de nommage lors de l'appel de la fonction et le maintien tant que la fonction n'est pas exécutée. Il est possible de voir le contenu avec la commande `locals()`.
- L'espace de nommage **enclosing** : si une fonction `inner_func` est définie à l'intérieur d'une fonction `outer_func`, les deux fonctions ont leur propre espace de nommage.

---

b. En pratique, les choses sont un tout petit peu plus compliquées sur le plan technique mais ça ne change rien sur le plan conceptuel.

Mais l'espace de nommage de `outer_func` est accessible depuis `inner_func`. Il s'agit de l'espace de nommage **enclosing** pour `inner_func`

### 5.3.2 Portée des variables

Lorsque l'on va faire appel à une variable, python va aller chercher le contenu dans le premier espace de nommage où la variable est définie (figure ??).<sup>c</sup> Ainsi, si on cherche à utiliser la variable `foo`, python va :

- d'abord aller la chercher dans l'espace *local*, si elle existe à cet endroit, alors elle utilise celle-ci.
- sinon, elle va aller chercher dans l'espace *enclosing*, si elle existe à cet endroit, alors elle utilise celle-ci.
- sinon, elle va aller chercher dans l'espace *global*, si elle existe à cet endroit, alors elle utilise celle-ci.
- sinon elle va aller chercher dans l'espace *built-in*, si elle existe à cet endroit, alors elle utilise celle-ci.
- si la variable n'a pas été trouvée, alors il y a un message d'erreur.

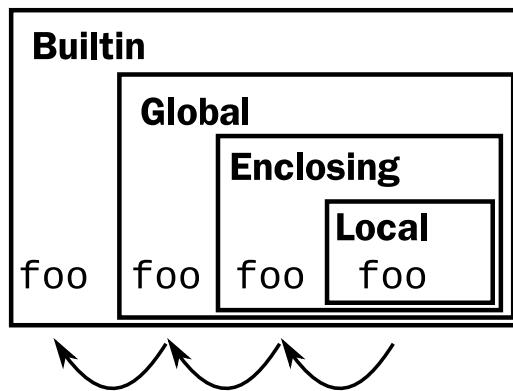


FIGURE 5.5 – Espaces de nommage et portée pour les différentes variables

Cet effet est illustré au sein du cahier jupyter suivant [C](#).

**💡** Pour rappel (section 5.2.2.1), les objets mutables peuvent être modifiés au sein d'une fonction, cela veut donc dire que si une variable n'existe pas dans l'espace de nommage local de la fonction et que l'on a utilisé une variable globale. Alors c'est la variable globale qui sera modifiée. Traquer les bugs dus à ce comportement peut être extrêmement difficile !

## 5.4 Structures de bases : boucles et conditions

Comme tout langage de programmation usuel, python utilise toutes les structures de boucles et condition usuelles : boucle `for`, `if`, `while`, etc. Il y a par contre quelques spécificités liées au langage.

c. Attention, dans d'autres langages de programmation, les espaces de noms sont différents. En particulier en Javascript, les espaces de nommage et les règles associées : portée des variables, sont très différentes de tous les autres langages de programmation. Je vous recommande chaudement la lecture de l'ouvrage *Javascript, the good parts* de Douglas Crockford pour mieux connaître toutes les « particularités » de ce langage si jamais vous avez à vous y frotter.

## 5.4.1 Boucles

### 5.4.1.1 Boucles for

La structure la plus basique est d'utiliser une itération sur une suite d'entiers. Ici, il faut comprendre que la fonction `range` retourne un objet itérable qui est une liste d'entier. De plus, l'argument donné est l'entier strictement supérieur au plus grand entier de la liste, il ne fera donc pas partie des entiers retournés.

Il est possible de spécifier l'indice départ et la pas.

```
for index in range(6):
    print(index)
```

**Parcourir une liste** Pour parcourir les listes, il existe différentes façon d'itérer :

- utiliser les indices des éléments de la liste `for i in len(liste);`
- itérer sur les éléments de la liste directement avec `for .. in :;`
- utiliser simultanément les éléments et les indices `for .. in enumerate(liste);`

**Parcourir un dictionnaire** Pour un dictionnaire, de base, on itère uniquement sur les clés, mais il est également possible d'itérer sur les paires clé/valeur :

- `for key in dictionnaire:` permet d'accéder aux valeurs des clés existant au sein du dictionnaire (rappel : les clés ne sont ordonnées qu'à partir de python 3.7);
- `for key,value in dictionnaire.items():` permet d'accéder aux paires clé/valeur;

### 5.4.1.2 Boucles while

Lors des processus itératifs, plutôt que d'avoir une boucle de longueur finie et connue à l'avance, il est possible d'avoir recours à une boucle qui se répète tant qu'une condition n'est pas satisfaite (une condition de convergence en général).

**Attention**, à cause des erreurs d'arrondi en virgule flottante, il est généralement préférable de vérifier une inégalité qu'une égalité qui ne sera peut-être jamais vérifiée.

## 5.4.2 Conditions if, elif, else

La structure la plus connue est `if, elif, else` qui permet de faire des tests. Comme dans le cas des boucles while, il faut faire attention aux conditions d'égalités entre deux variables s'il s'agit de nombres flottant. Il est généralement préférable d'utiliser la fonction `isclose` de la librairie `math` ou `numpy`. Il est normalement préférable de mettre les conditions dans leur ordre de probabilité décroissante.

Il est possible de combiner des tests avec les opérateurs `or`, `and`. Tout comme il est possible de tester la négation avec l'opérateur `not`.

**Comparateurs** Les comparateurs numériques autorisés sont les suivants : `<`, `>`, `<=`, `>=`, `==` (égalité), `!=`. De plus, python demande toujours d'utiliser un double `==` pour tester les égalités.

Il est aussi possible d'utiliser les comparateurs suivants :

- `is` vérifie l'égalité stricte entre objets pour savoir s'il s'agit du même objet à la même adresse mémoire. La comparaison est donc plus stricte que pour l'opérateur `==`. `is not` est l'analogie mais pour la négation de `is`.

- `in` pour savoir si un élément est présent dans un objet (dans une liste en général)
- `isnumeric` pour tester si la variable est numérique, `instance` pour savoir si une variable a le type attendu, etc.

#### 5.4.2.1 Instructions `break`, `pass` et `continue`

Au sein des boucles, il est possible d'utiliser les instructions spéciales :

- `pass` permet d'avoir un début de bloc sans rien avoir à faire au sein du bloc, cela permet donc essentiellement de tester la validité de l'en-tête du bloc (condition, boucle) ou alors d'éviter une erreur d'exécution si jamais le bloc est encore vide pour l'instant.
- `break` permet de sortir totalement du bloc d'instruction, ainsi, si l'instruction `break` est exécutée, alors on continue à partir de la fin du bloc
- `continue` permet de sauter pour terminer la boucle courante et aller directement au début de la boucle suivante

## 5.5 Gestion des fichiers

Il est courant d'avoir à manipuler des fichiers de donnée, aussi bien en lecture qu'en écriture. Les fichiers manipulés peuvent être particulièrement volumineux : il est possible d'atteindre plusieurs dizaines de Go pour certains types de calcul. Il faut donc particulièrement être vigilant à l'usage de la mémoire : il sera impossible de charger intégralement d'aussi gros fichiers en mémoire.

L'ouverture d'un fichier se fait avec la fonction `open` ⓘ. Le premier argument est le nom du fichier, le deuxième le type d'opérations qui pourront être effectuées avec le fichier (tableau 5.4). Ⓣ

Caractère	Signification
'r'	ouvre en lecture (par défaut)
'w'	ouvre en écriture, en effaçant le contenu du fichier
'x'	ouvre pour une création exclusive, échouant si le fichier existe déjà
'a'	ouvre en écriture, ajoutant à la fin du fichier s'il existe
'b'	mode binaire
't'	mode texte (par défaut)
'+'	ouvre en modification (lecture et écriture)

TABLEAU 5.4 – Modes d'ouverture possibles pour les fichiers.

Normalement, tout fichier ouvert doit être fermé lorsque son utilisation est finie avec la méthode `.close()`, sinon, il se peut que son contenu ne soit pas enregistré. Il est possible de rendre la fermeture automatique en utilisant l'instruction `with` :

```
with open(filename, 'r') as f:
    pass
```

À la fin du bloc `with`, le fichier sera alors automatiquement fermé.

Pour lire un fichier, il existe plusieurs méthodes, soit pour le lire en entier, soit pour le charger morceaux par morceaux, cela est illustré dans le script de démonstration suivant Ⓣ.

d. Il est préférable d'ouvrir le fichier avec des droits les plus restreints possibles. Cela pourra par exemple permettre d'éviter d'écrire par inadvertance par dessus un fichier de sortie résultant de plusieurs jours/semaines de calculs.

 Lorsque c'est possible, il est préférable d'utiliser des chemins relatifs pour les fichiers afin que le code soit le plus portable possible (indépendant de l'ordinateur sur lequel il est exécuté).

## 5.6 Évaluer la performance de son code

Pour évaluer la performance de son code, il est nécessaire de faire des tests d'exécution, ce qui implique différentes phases. Pour faire des mesures, il est courant d'utiliser la librairie `timeit` qui permet de chronométrier l'exécution d'une portion de code un grand nombre de fois pour que cela soit représentatif car cela n'est pas toujours strictement reproductible en fonction des autres processus en cours sur la machine au moment de l'exécution. Il est courant de faire des comparaisons d'algorithmes en fonction de la taille et d'avoir des résultats non monotones (un algorithme sera plus performant dans certains cas, moins dans d'autres). Pour des cas complexes (entraînement de modèle d'IA, parallélisation), un même code peut avoir des temps d'exécutions et une efficacité très différents en fonction de l'architecture complète de la machine. Ainsi, pour des codes qui durent longtemps et /ou consomment beaucoup de ressources, il peut être crucial de faire des tests de performance pour avoir un code suffisamment efficace pour mener la tâche correctement. Pour des codes ayant des temps d'exécution très court, c'est moins nécessaire.

Il faut cependant toujours garder à l'esprit qu'un code qui marche à petite échelle (phase de développement, de test) peut très rapidement devenir totalement inutilisable à grande échelle (agrandissement du système d'étude, utilisation abusive d'un modèle). Un code « bien pensé » dès le début sera en général plus robuste. Malheureusement, il n'y a pas de règle systématique et absolue pour pouvoir résister lors du changement d'échelle...

Un exemple pour illustrer l'utilisation de `timeit` et l'importance de ne PAS utiliser de boucle `for` avec `numpy` est disponible sur le github [C](#).

## 5.7 Ce qu'il faut retenir

- Les types de bases utilisables sous python (`str, int, float, list, tuple, dict`<sup>e</sup>)
- Les options de formatage accessibles pour mettre en forme des variables.
- Que les nombres sont stockés sous forme de flottants, et que cela peut entraîner des erreurs numériques. Entre autre, soit parce que la valeur réellement utilisée n'est pas strictement égale à la valeur écrite par le programmeur, soit parce que l'on mélange de grands et de petits nombres.
- Il existe deux catégories d'objets en python : les objets mutables et les objets immutables.
- Le caractère mutable ou immutable a des conséquences importantes lors de la copie d'objets (*deep copy* versus *shallow copy*).
- Le caractère mutable ou immutable a des conséquences importantes lors de l'exécution de fonctions : une variable globale mutable peut être modifiée par une fonction.
- Comprendre la notion d'espace de nommage et les conséquences en terme de portée des variables.
- Savoir parcourir des listes, dictionnaires, etc. En sachant utiliser, en particulier les structures suivantes :
  - `for element in liste,`

e. Il existe aussi le type `set`, qui dérive des listes avec certaines particularités et méthodes spécifiques.

- `for index,element in enumerate(liste)`
- `for key,value in dict.items()`
- Savoir ouvrir et fermer un fichier, le lire ou y écrire.

# Chapitre 6

## Modules, librairies, packages et frameworks

### 6.1 Un peu de vocabulaire

Sous python, on peut parler de module, librairie, package ou framework. Bien que légèrement différents, ces différentes termes regroupent tous une simple et même idée : ce sont des choses qui regroupent des ensembles de fonction, classes et objets pour faciliter la programmation ou ré-utiliser des portions de code. Cela peut aller d'un code écrit modestement dans son coin à de grosses machineries de plusieurs dizaines de millier de lignes de code maintenues par une communauté.

- Un module correspond à un fichier qui contient les fonctions que l'on cherche à utiliser.
- Un package est un ensemble de modules construits de manière cohérente, cela correspond à un répertoire de modules qui sont organisés.
- Une librairie est à visée encore plus large qui correspond généralement à un ensemble de package. Parfois, le terme peut d'ailleurs est utilisé de manière interchangeable avec celui de package.
- Les workflows sont en général plus complexes car en plus de fournir des objets, fonctions ou classe, ils imposent en général une manière de les faire interagir entre eux. Ce qui force en général certaines formes de structuration du code qui les utilise.

Ici, par la suite j'utilisera essentiellement le terme de librairie qui est le plus générique en programmation.

### 6.2 Au fait, une librairie ça sert à quoi ?

Les librairies vont en général avoir plusieurs rôles tous aussi importants les un que les autres.

**Factoriser des morceaux de code** Les librairies et modules permettent de ré-utiliser des fonctions. C'est leur principal rôle. En définissant les fonctions à un unique endroit, cela implique que les corrections sont alors immédiatement effectives partout où elles sont utilisées. Par contre, cela peut également apporter son lot de nouveaux bugs ou casser la compatibilité du code. Normalement, il faut toujours veiller autant que possible à maintenir une compa-

tibilité ascendante<sup>a</sup>

 Ainsi, il est systématiquement préférable de maintenir ses fonctions à un UNIQUE endroit pour éviter des divergences de fonctionnalités, niveaux de débogage. Même si en retour cela peut parfois demander de vérifier que les fonctions d'une librairie sont utilisables dans divers contextes.

**Avoir à disposition des fonctions robustes, déjà écrites et testées** La plupart des librairies phares ont plusieurs dizaines d'années de recul. Elles ont été testées, pensées et réfléchies pour répondre aux besoins, être finement paramétrables, etc. Plutôt que de ré-inventer la roue, il est en général plus simple de s'en servir pour faire quelque chose de plus compliqué. Cela permet également de gagner énormément de temps : il n'y a pas à coder les fonctions, et surtout pas à les déboguer.<sup>b</sup>

**Bénéficier d'un espace de nommage dédié** Les librairies bénéficient d'un espace de nommage dédié. Pour des codes qui deviennent complexes, cela permet d'éviter les conflits de noms qui peuvent mener à des catastrophes avec des objets mutables. Ainsi, on peut avoir des fonctions qui s'appellent `max` dans plusieurs modules différents et qui ont un comportement différent. On peut alors facilement différencier des fonctions en leur assignant des espaces de nommage différent plutôt qu'en cherchant à avoir des noms de fonctions différents. De plus, cela permet d'être plus libre dans l'espace de nommage global en n'ayant pas à se pré-occuper de l'ensemble des noms de fonctions contenus dans une librairie.

## 6.3 Import d'une librairie ou d'un module

**Import d'un module local** Il est possible d'importer un module local. Dans ce cas, il est possible de préciser le chemin du fichier qui contient le module. Celui-ci **doit** avoir une extension en `.py` et lors de l'import, il ne faut **pas** ajouter l'extension.

```
import top_module as tm
```

Permet alors d'importer le fichier `top_module.py` placé dans le même répertoire. Au passage, la directive « `as` » permet d'assigner un espace de nommage `tm`. Cela revient conceptuellement à faire un copier coller du fichier importé à l'endroit où le module est importé. Chacune des fonctions ou classes définies dans le module sera alors accessible si on utilise comme préfixe du nom de la fonction l'espace de nommage correspondant. Ainsi, si le fichier `top_module.py` contient la fonction `myFunc()`, alors celle-ci pourra être utilisée avec `tm.myFunc()` sans rentrer en conflit avec l'espace de nommage global du script. Il sera donc également possible de créer une fonction `myFunc()` au sein du script.

**Import d'une librairie générique** Pour une librairie générique, la syntaxe est rigoureusement identique, la différence étant qu'au lieu d'aller chercher dans le dossier du script, python va généralement chercher dans un ensemble de répertoires prédéfinis la librairie.

a. C'est à dire que les évolutions de la librairie ajoutent des fonctionnalités tout en gardant le code écrit avant ces modifications toujours fonctionnel. Cela peut sembler naturel et simple, mais en pratique, c'est souvent plus compliqué.

b. Attention, pour des librairies obscures, récentes, ou peu utilisées, il se peut que vous découvriez des bugs et que le temps de les découvrir vous coutera plus que le temps de faire un code fonctionnel. Il y a toujours une estimation du bénéfice-risque à avoir lors de l'utilisation d'une librairie spécifique.

Cela demande bien évidemment d'avoir installé la librairie au préalable. Si on dispose des droits administrateurs sous Linux, cela se fait le plus souvent avec la commande :

```
pip install ma_librarie
```

Il est ensuite possible d'importer la librairie comme pour un module local avec la commande import :

```
import ma_librarie # ou import ma_librarie as ml
```

Par contre, l'emplacement de cette dernière peut être plus difficile à trouver. La librairie inspect peut permettre de trouver le fichier et donc consulter le fichier source pour trouver de la documentation ou le code source.<sup>c</sup> De plus, la librairie inspect permet d'afficher la documentation de la fonction très facilement  :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
5 Programme python pour trouver le fichier contenant une fonction donnée
"""

import inspect

10 import scipy
import scipy.optimize as optimize

if __name__ == "__main__":
    func = optimize.least_squares # Nom de la fonction cherchée dans l'espace de nommage
    approprié
15 files = inspect.getfile(func) # Fonction pour localiser le fichier contenant ladite
    fonction
    doc = inspect.getdoc(func)
    print(files) # Affiche l'emplacement de la fonction
    print(doc) # Affiche la documentation de la fonction

20 #retourne /usr/lib/python3/dist-packages/scipy/optimize/_lsq/least_squares.py au CBP
```

### 6.3.1 Import spécifique

Plutôt que d'importer une librairie complète, il est parfois plus judicieux de n'importer que quelques fonctions de ladite librairie plutôt que l'intégralité de celle-ci. La forme de l'import est alors très légèrement différente :

```
from numpy import sqrt
```

Dans ce cas là, la fonction est ajoutée à l'espace de nommage global et pas à celui d'une librairie. Il n'y a donc plus à utiliser de préfixe pour utiliser ladite fonction. 

### 6.3.2 Import « à la sauvage »

Sinon, il existe également une méthode à ne PAS utiliser :

```
from numpy import *
```

---

c. voir sections 4.3.2 et 4.3.3.

Dans ce cas là, TOUTES les fonctions de la librairies sont ajoutée à l'espace de nommage global. Ce qui peut engendrer des conflits avec des variables ou fonctions définies au sein du script. ☺

## 6.4 Quelques librairies courantes

Il existe un nombre quasiment infini de librairies et modules en python. Les lister tous n'aurait strictement aucun sens. Le tableau 6.1 en présente quelques unes qui peuvent servir.

Nom de la librairie	But
<code>datetime</code>	permet de manipuler des dates
<code>os</code>	Permet d'utiliser des lignes de commande au sein de python
<code>sys</code>	Accès à certaines données système
<code>pathlib, glob</code>	opérations sur le système de fichier et les chemins
<code>argparse</code>	permet d'avoir accès des options pour le script comme s'il s'agissait d'une ligne de commande
<code>inspect</code>	Pour avoir accès à des données d'une librairie
<code>re</code>	Utilisation des expressions régulières (voir 10)
<code>openpyxl</code>	Création de fichiers excel sous python
<code>black</code>	uniformisation du code
<code>tkinter</code>	Génération d'interface graphique
<code>django</code>	Framework pour le développement web
<code>matplotlib / seaborn</code>	Création de graphiques scientifiques
<code>math</code>	Fonctions et objets mathématiques de base
<code>itertools</code>	Fonctions pour la combinatoire
<code>numpy</code>	Calcul sur des tableaux et opérations mathématiques « de base » usuelles
<code>scipy</code>	Outils scientifiques usuelles (statistiques, équations différentielles, FFT, recherche de zéro, fit, etc)
<code>pandas</code>	Gestion de tableaux de donnée
<code>scikit-learn,</code> <code>tensorflow, pyTorch</code>	Machine learning sous python
<code>PySerial</code>	Utilisation du port série sous Python pour communiquer avec des instruments
<code>trackpy</code>	Analyse d'images pour suivre des objets
<code>h5py</code>	lecture de fichiers HDF5 (format de données scientifiques)
<code>sage / sympy</code>	Calcul analytique/symbolique sous python (dérivation analytique, etc)

TABLEAU 6.1 – Quelques librairies utiles. Par la suite, nous nous focaliserons sur numpy, scipy et matplotlib.

## 6.5 Ce qu'il faut retenir

- Savoir importer un module en local ou une librairie ;
- Comprendre l'importance du fait d'avoir un espace de nommage pour chaque module ;
- Savoir faire différents types d'imports, et aussi ne pas faire d'import sauvage dans l'espace de nommage global.
- Savoir lire la documentation d'une librairie.



# Chapitre 7

## La manipulation de tableaux avec Numpy

Pour faire des simulations ou du traitement de données, il est courant de faire appel à des données sous forme de tableaux. L'utilisation de ces tableaux en programmation « bas-niveau » se traduit généralement par l'utilisation répétée de boucle `for`. La bibliothèque numpy créé un nouveau type de variable : le type *N-dimensional array* ou *ndarray*. Ces tableaux sont donc multidimensionnels comme annoncé dans leur nom et ont pour particularité :

- de ne pouvoir contenir qu'un unique type d'éléments (par exemple : que des `int`, que des `float`, etc);
- d'être forcément homogènes : il doit toujours y avoir le même nombre d'éléments dans chacune des dimensions.

La force de la bibliothèque numpy va être de pouvoir effectuer des opérations courantes sur ces tableaux (filtrage, multiplication, dérivation, concaténation, etc) avec une rapidité bien supérieure à celle obtenue via l'utilisation de boucles `for`. De plus, les fonctions proposées sont en général capable d'agir sur tout le tableau ou uniquement certaines dimensions ce qui rend les opérations d'autant plus faciles. ☀

### 7.1 Les `ndarray`

#### 7.1.1 Propriétés essentielles

Un `ndarray` est caractérisé par différentes grandeurs. Il est crucial de comprendre chacune de ces caractéristiques pour pouvoir exploiter pleinement la puissance de la bibliothèque numpy.

Ces caractéristiques sont résumées dans le tableau 7.1. ☀

Caractéristique	Fonction	Explication
Dimension	<code>data.ndim</code>	Dimension du tableau. 1 correspond à un vecteur, 2 à une matrice, 3 à un parallélépipède, 4 un hypercube, etc
Shape	<code>data.shape</code>	Indique la longueur de chacune des dimensions sous forme de tuple
Taille	<code>data.size</code>	Nombre total d'éléments dans le tableau, il s'agit donc du produit de chacune des valeurs présentes dans <code>data.shape</code>
Type	<code>data.dtype</code>	Types d'éléments présents dans le <code>ndarray</code>

TABLEAU 7.1 – Caractéristiques de base d'un `ndarray` appelé `data`.

**Affichage d'un ndarray** Pour l'affichage (figure 7.1) :

- le **dernier** axe est toujours affiché de gauche à droite;
- l'**avant-dernier** axe est toujours affiché de haut en bas;
- pour les autres, ils sont affichées de haut en bas avec un espace pour séparer chaque « tranche ».

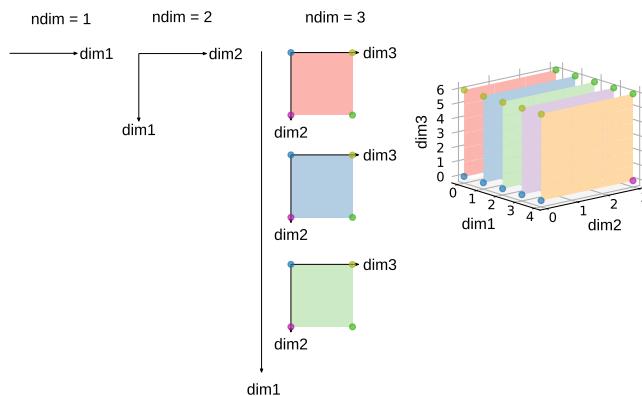


FIGURE 7.1 – Sens d'affichage des ndarray.

### 7.1.2 Fonctions pour la création

Sans chercher à être exhaustif, il existe différentes manières de créer des ndarray. La plus simple est de les déclarer à la main avec `np.array`, sinon, `np.asarray` permet de convertir l'objet fourni en ndarray. La fonction `np.copy` fait une *shallow-copy* d'un tableau.<sup>a</sup>

Sinon, les fonctions présentées tableau 7.2 permettent également de créer des tableaux automatiquement à partir de leur shape. Il est également possible de créer des matrices diagonales, tridiagonales, etc. ☺

Fonction	Commentaire
<code>empty</code>	Créé un tableau non initialisé, les valeurs sont donc quelconques
<code>zeros</code>	Créé un tableau rempli avec des zéros
<code>ones</code>	Créé un tableau rempli avec des uns
<code>full</code>	Créé un tableau rempli avec une unique valeur

TABLEAU 7.2 – Fonctions usuelles pour la création de ndarray. Toutes ces fonctions existent également avec le suffixe `_like` pour créer un tableau avec un shape identique à celui du tableau d'origine indiqué.

**Tableaux unidimensionnels** Il est également courant de créer des tableaux unidimensionnels qui contiennent des suites de nombre espacés régulièrement (tableau 7.3).

**Coordonnées multidimensionnelles** Il peut être parfois utile de travailler sur des tableaux qui contiennent des coordonnées multidimensionnelles au lieu d'éléments simples. Pour cela, la fonction `np.meshgrid` est indiquée. ☺

a. Il faut donc être très vigilant si jamais le ndarray contient des objets mutables (voir section 5.2).

Fonction	Commentaire
<code>np.arange</code>	suite des entiers
<code>np.linspace</code>	suite d'éléments régulièrement espacés compris entre deux bornes
<code>np.logspace</code>	suite d'éléments espacés logarithmiquement compris entre deux bornes, se différencie de la fonction précédente essentiellement par la manière dont sont données les bornes

TABLEAU 7.3 – Fonctions pour créer des tableaux unidimensionnels.

## 7.2 Sélection d'éléments et *slicing*

Les tableaux  $N$ -dimensionnels permettent de stocker des données sous une forme très pratique où chaque axe correspond à un type de données. Par exemple, pour suivre une particule au cours du temps, on peut décider qu'un axe va représenter un pas de temps et qu'un deuxième axe va permettre de stocker les coordonnées  $x,y,z$ .

Il est alors important de pouvoir choisir certains éléments du tableau de manière pratique : par exemple, sélectionner toutes les valeurs  $x(t)$  ou à l'inverse la position de la particule  $(x(t_i),y(t_i),z(t_i))$  à un instant  $t_i$  donné. ☺

### Accès simple à un élément ou un groupe d'éléments

- Un entier indique une coordonnée précise ;
- Un symbole « `:` » indique tous les éléments correspondants à cette dimension. Si jamais il y a moins d'indices de fournis que la dimension du tableau, alors c'est équivalent à rajouter des « `:` » jusqu'à atteindre la dimension du tableau.
- Les entiers négatifs correspondent au fait de compter à partir de la fin. `-1` indique le dernier élément dans une dimension, `-2` l'antépénultième, etc.

Groupe d'indice	Résultat
<code>data[2,3,4]</code>	On fournit <code>data.dim</code> coordonnées : sélectionne l'élément placé aux indices fournis
<code>data[2,:,:]/data[2]</code>	Utilisation du caractère « <code>:</code> » pour sélectionner tous les éléments correspondants. On aura ici un tableau de dimension <code>data.ndim-1</code> et de shape <code>(data.shape[1], data.shape[2],)</code>

TABLEAU 7.4 – Différentes méthodes pour accéder à des éléments d'un tableau. ☺

💡 À la différence de liste imbriquées, on peut voir que pour un `ndarray`, la sélection d'éléments se fait avec une unique paire de crochets.

**Accès à un élément ou un groupe d'éléments spécifique : *slicing*** Cela va correspondre au fait de faire du *slicing* (tranchage) du tableau.

Pour faire du slicing, on va indiquer les éléments à sélectionner sous la forme :

```
start:stop:step
```

où `start` est l'indice de départ, `stop` est le premier indice exclu, `step` est le pas entre deux indices. ☺

- si `step` est manquant, il vaut 1 ;
- si on indique seulement `start` suivi de ":" on prend tous les éléments à partir de l'indice de départ inclus ;
- si on indique seulement ":" suivi de `stop` on prend tous les éléments jusqu'à celui indiqué par `stop` exclu ;
- par défaut, `start` vaut 1 et `stop` vaut le *shape* de la dimension correspondante.

Si jamais le slicing doit être dynamique (dépendre de certaines variables), il est possible d'utiliser un oslice : `slice(start, end, step)`. Si un des indices vaut `None`, alors il prend la même valeur que par défaut.

**Sélection conditionnelle d'éléments** Il est possible de faire du *masking* pour sélectionner des éléments uniquement s'ils vérifient une condition. Par exemple, sur un tableau d'entier, on peut sélectionner tous les nombres pairs avec la condition suivante :

```
data[data % 2 == 0]
```

Attention, dans ce cas, le résultat retourné est toujours un élément à une seule dimension vu que le résultat n'a aucune raison d'être un tableau de type ndarray (avec toujours autant d'éléments dans chacune des dimensions).

## 7.3 Opérations sur les tableaux, *broadcasting* et notion d'axe

La librairie numpy propose de nombreuses opérations possibles. Les opérations de base sont celles **élément par élément** (addition, multiplication, soustraction, etc).

### 7.3.1 *Broadcasting* ⓘ

En cas de manipulation de tableaux qui n'ont pas exactement le même shape, alors il peut y avoir des incompatibilités. Les règles utilisées par numpy sont celles du *broadcasting* qui permet de faire en sorte de se ramener systématiquement à des tableaux ayant une forme analogue.

**Règle du broadcasting** En partant de la dernière dimension, on se ramène à des tableaux ayant le même shape de la manière suivante :

- si les dimensions sont égales, alors on ne fait rien ;
- si une des deux dimensions est égale à 1, alors on « étire » le tableau dans la direction correspondante jusqu'à avoir deux dimensions égales ;
- Si on a « épuisé » les dimensions dans un des deux tableaux, alors on ajoute une nouvelle dimension « par la gauche » et on étire le tableau dans la dimension correspondante.
- sinon, si pour une dimension les dimensions sont différentes sans qu'une des deux soit égale à 1, il y a une erreur.

Puis on continue ainsi de droite à gauche pour toutes les dimensions jusqu'à avoir des dimensions égales pour les deux tableaux. ⓘ

 Les situations de grand danger sont celles où il peut y avoir broadcasting mais qu'il soit « mauvais » (ne correspond pas à ce que l'on veut faire). Par exemple si on veut multiplier un vecteur de shape  $(3,)$  avec une matrice  $3 \times 3$  mais selon l'avant dernière dimension et pas la dernière dimension. ⓘ

### 7.3.2 ⚡ Opération sur les tableaux, notion d'axe

Pour les tableaux, il est possible d'utiliser un grand nombre de fonctions : somme, moyenne, médiane, dérivée, etc. Cependant, il n'est pas rare d'être intéressé par une opération mais uniquement selon certaines dimensions. Ces dimensions correspondent à des axes du tableau. ☺ Cela permet d'être plus spécifique pour les opérations à mener et évite généralement d'avoir à faire des opérations compliquées pour obtenir un résultat équivalent. En général, la/les dimensions sur lesquelles s'effectue l'opération n'apparaissent plus dans le résultat (sauf si cela est demandé explicitement). ☺

## 7.4 Points divers

### 7.4.1 Dérivation numérique *versus* dérivation analytique

La bibliothèque numpy n'est PAS une bibliothèque pour faire du calcul symbolique/analytique mais pour faire du calcul numérique sur des tableaux. Il faut donc bien comprendre les limites des différents calculs effectués. Ainsi, toutes les opérations de dérivation, intégration et autre vont avoir des limitations par rapport à des opérations analytiques :

- Les formules utilisées sont en général de bonnes approximations numériques mais ne sont pas exactes ;
- De plus, le calcul en nombre flottant (voir section 5.1.1.1) introduit des erreurs dans certains cas.

**Dérivation** Pour calculer la dérivée, la fonction `np.gradient` utilise une formule centrée d'ordre 2 si l'espacement entre les points est régulier.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad (7.1)$$

On peut montrer qu'il y a deux domaines en fonction de la valeur de  $h$  (figure 7.2) :

- Dans un premier temps, lorsque  $h \rightarrow 0$ , il y a généralement une amélioration du résultat car l'approximation est de mieux en mieux vérifier avec le passage à la limite.
- Cependant, au-delà d'un certain seuil, (en général vers  $h \approx 10^{-6}$ ) le résultat se dégrade à cause du calcul en nombre flottant (voir la section 5.1.1.1)

De plus, aux bords, par défaut, numpy utilise non pas une formule d'ordre deux, mais une formule d'ordre 1, ce qui peut conduire à des comportements « anormaux » pour les premiers points. Avec l'option `edge_order = 2`, il est possible de se ramener à une formule adaptée qui est d'ordre deux comme pour l'ensemble des autres points. ☺

Il existe des formules d'ordre supérieur à deux, tout comme des fonctions spécifiques pour calculer des dérivées d'ordre supérieur, mais dans ce cas, il faut aller chercher du côté de librairies spécifiques (`findiff` ou `numdifftools` par exemple) ou programmer soi-même ces implémentations particulières du calcul de la dérivée. En particulier l'utilisation répétée de l'opération `np.gradient` pour calculer les dérivées successives peut aggraver les différents problèmes numériques.

### 7.4.2 Nombres pseudo-aléatoires

Numpy propose une série de fonctions pour générer des nombres aléatoires selon des lois variées. ① Au-delà des aspects techniques, il faut surtout se rappeler que les nombre

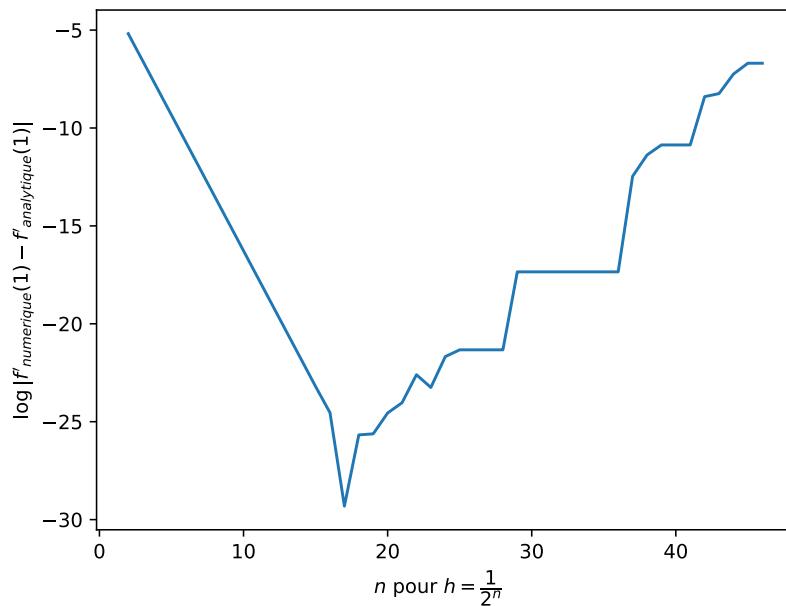


FIGURE 7.2 – Pour la fonction  $f(x) = \sin(x)$ , logarithme de la valeur absolue de la différence entre la valeur de la dérivée calculée numériquement avec la fonction `np.gradient` de numpy et la valeur analytique pour différent valeur de  $n$  avec  $h = \frac{1}{2^n}$  au point  $x = 1$ .

aléatoires ne le sont que partiellement : au bout d'un certain temps, ils se répètent, peuvent ne pas suivre la loi qu'ils sont censé représenter pour un grand nombre de tirages, etc. Comme la plupart des générateurs de nombre aléatoires, numpy fait appel à un *seed* qui va déterminer la suite de nombre aléatoire qui sera tirée. S'il n'est pas possible de connaître la valeur de ce seed après un tirage, il est au contraire possible de le fixer à priori avec une valeur. Cela est très utile si on cherche à tester son programme et effectuer des comparaisons sur des nombres aléatoires mais toujours identiques. Par contre, si jamais on se retrouve par hasard avec deux seed identiques, les deux séries de nombres aléatoires qu'on peut tirer se retrouvent alors étroitement corrélées.

Il existe un module spécifique dédié (`secrets` ⓘ) pour le tirage des nombres aléatoires utilisables pour de la cryptographie si jamais l'aspect pseudo-aléatoire n'est pas suffisant. Il faut ainsi retenir que numpy propose la génération de nombres aléatoires suffisants pour faire de la modélisation numérique, mais que ce n'est PAS adapté pour des aspects cryptographiques.

## 7.5 Ce qu'il faut retenir

- Pouvoir créer un tableau de type `ndarray`;
- Comprendre la notion de dimension et de `shape` pour les `ndarray`;
- Savoir sélectionner un élément ou un groupe d'élément avec du `slicing`;
- Connaître et comprendre les règles de `broadcasting`.
- Savoir utiliser des opérations selon un axe ou un groupe d'axes.
- Être conscient des limites sur le calcul numérique de la dérivée.
- Être conscient des limites de la génération de nombres aléatoire et comprendre la notion de `seed`.

# Chapitre 8

## Faire des graphiques avec Matplotlib

La création de figure représentative de résultats expérimentaux est cruciale en science pour réussir à montrer à son lecteur les causes, effets, subtilités et résultats expérimentaux. Un bon graphique peut réussir à faire passer un message impossible à synthétiser en une page. Tout comme il peut expliciter des points techniques d'un modèle.

En temps normal, la création d'un bon graphique peut ainsi prendre une voire plusieurs journées. D'autant plus lorsque la quantité de données est importante.

### 8.1 Principes généraux

Réussir à donner un ensemble de règles à suivre pour la création d'un bon graphique est une gageure. De plus cela est un sujet à part entière. L'expérience rendra les choses plus faciles au fur et à mesure. Je liste ci-dessous quelques règles génériques qui ne sont exhaustives ni toujours à appliquer.

**Évitez les camemberts et toute représentation utilisant un angle pour comparer deux valeurs numériques** : nos yeux sont extrêmement mauvais pour comparer des angles, il faut donc éviter à tout prix les représentations qui sont basées sur ce principe. Une comparaison faisant appel à une comparaison de longueur est généralement bien plus pertinente. Et pour des valeurs simples, un bon tableau peut être plus pertinent si jamais il est bien présenté. ☺

Le même principe s'applique pour la comparaison de dérivées ou pour faire la différence entre deux courbes. ☺

**Évitez les courbes en 3D pour tout ce qui est données numériques** Encore une fois, notre œil a du mal à comparer des données dans un espace tridimensionnel projeté sur 2 dimensions. Les représentations 3D peuvent être pertinentes pour représenter des données spatiales, mais beaucoup moins pour des données numériques. [site Fundamentals of Data Visualization](#)

**Faites des graphiques où les données principales ressortent** Il faut mieux rendre les axes, légendes etc un peu moins prégnantes pour faire ressortir les données. De même, il peut être intéressant de souligner des points où il faut regarder, passer les axes en gris au lieu du noir par défaut, etc. De même, réduisez au maximum tout ce qui pourrait distraire l'œil des données (gridline/grilles, surabondance de ticks, etc).

**Réfléchissez au fait d'inclure le zéro ou pas** Si vous cherchez à comparer des données absolues, n'oubliez pas de donner une référence et donc à priori, indiquez la position du zéro pour montrer si la variation est importante ou non. À l'inverse, si vous êtes intéressés par les variations plutôt que par la valeur absolue, alors ne pas inclure le zéro peut permettre d'utiliser au maximum l'espace pour les souligner. ☺

**Un choix de couleur harmonieux** Il est généralement très difficile de trouver un lot de couleurs harmonieuses. Matplotlib propose différents jeux de couleur [①](#) et utilise par défaut les couleurs tab10 pour différencier les lignes. Pour plus de variations, j'utilise personnellement le site [ColorBrewer](#) qui propose de nombreuses options, indique si le schéma résiste à la photocopie ou au passage en noir et blanc, etc. Au delà de quelques couleurs, il est rapidement très difficile de faire quelque chose qui convienne à la fois aux personnes ayant des déficiences visuelles (daltonismes en particulier), qui résiste à la photocopie, au passage en noir et blanc et soit facilement rendu aussi bien à l'impression que sur écran... Les couleurs criardes/saturées par défaut dans les logiciels de dessin sont tout de même généralement déconseillées, en particulier le jaune sur fond blanc qui ressort souvent très mal.

**Ne pas être avare en figure** Il est généralement judicieux de multiplier des figures mettant en valeur plusieurs manière de présenter les données pour décrire au mieux un jeu de données complexe que de surcharger une unique figure avec toutes les données.

**Allez visiter la galerie** La [galerie](#) regorge d'exemples qui peuvent vous aider à construire vos graphiques. Ceux-ci peuvent vous aider à trouver de l'inspiration. Cependant, ce n'est pas toujours parce que vous pouvez faire quelque chose que c'est judicieux de le faire. Et à l'inverse, ce qui doit vous guider est la manière dont vous voulez mettre en valeur vos résultats, pas la bibliothèque python que vous utilisez pour le faire. Ainsi, il est parfois très difficile de faire des choses esthétiques, mais le temps passé à le faire vous fera progresser et contribuera à faire en sorte que votre message soit convaincant.

### 8.1.1 Bibliographie

Avant de commencer, je vous recommande très chaudement de passer du temps dans la lecture des quelques ressources bibliographiques suivantes :

- [Fundamentals of Data Visualization](#) de Claus O. Wilke : de nombreux exemples illustrés avec de bonnes et mauvaises pratiques. Couvre de nombreux exemples, balaye le sujet tout en restant concis.
  - Les livres d'Edward Tufte, ce dernier ayant eu une influence significative sur la présentation des données :
    - *Envisioning information*, Edward R. Tufte, 9780961392116
    - *The visual display of quantitative information* Edward R. Tufte, 9780961392147
    - *Visual explanations : images and quantities, evidence and narrative* Edward R. Tufte, 9780961392123
    - *Beautiful evidence* Edward R. Tufte, 9780961392178
- Son dernier livre : *Seeing with Fresh Eyes : Meaning, Space, Data, Truth* est par contre une réelle déception et ne vaut pas le coup d'œil.

## 8.2 Utilisation de matplotlib

Le plus simple est probablement de commencer par aller voir les [antisèches sur matplotlib](#) faites par Nicolas Rougier. Ces fiches expliquent toutes les options les plus courantes de matplotlib. Le reste des options se fera au cas par cas en fonction des besoins spécifiques.

**Subplots** Au-delà des multiples options, le plus important à retenir est que certaines fonctions qui marchent pour un unique graphique (`plt.title`, `plt.xlabel`, etc) ne marchent pas pour des sous-graphiques. La différence principale étant en général le fait d'utiliser un préfixe `set_` pour les sous-figures.

Plutôt que l'utilisation de `plt.subplots` ⓘ, je vous recommande plutôt d'utiliser `gridspec` ⓘ qui offre plus de souplesse – au moins dans sa syntaxe. Une troisième alternative est `subplot_mosaic` ⓘ

**Coordonnées** Matplotlib permet également de placer des éléments sur un graphique. Il existe alors plusieurs manières de décrire les coordonnées :

- soit en fonction des données du graphique (*data*), cela est pratique pour légendrer certains points précisément;
- soit en fonction de la position absolue dans le cadre (*display*), ce qui est plus pratique pour un placement absolu. Attention, les valeurs en coordonnées absolues ne sont calculées qu'au départ et peuvent donc être fausse lors du redimensionnement de la fenêtre.

Pour plus d'informations, il faut aller voir la documentation. ⓘ.

**Enregistrement, mise en page** Lors de la création des figures, il est recommandé de préciser la taille avec l'argument `figsize` pour éviter de faire des redimensionnements hasardeux et inesthétiques :

```
fig = plt.figure(figsize=(10, 8))
```

La taille indiquée est en pouce.

Il y a intérêt à enregistrer les données (avec la fonction `savefig`) au format `svg` pour pouvoir les retoucher si nécessaire (avec `inkscape` par exemple). Par contre, si le nombre de données est extrêmement grand, alors des options comme `rasterized` évitent d'avoir des images trop lourdes.

De plus, pour ajuster les marges, la fonction `tight_layout()` permet usuellement d'ajuster correctement les espaces pour éviter des chevauchements.

Par défaut, matplotlib affiche un cadre complet autour de chaque figure, cependant, il est possible d'en enlever certaines parties avec `ax.spines[] .set_visible(False)` où l'argument de `spines` peut être au choix parmi `top, bottom, left, right`.

**Personnalisation** Il est possible de changer un très grand nombre de paramètres par défaut, il est possible de les définir via `matplotlib.rcParams` ou via un fichier `matplotlibrc` dans le répertoire courant. ⓘ On pourra noter en particulier la possibilité d'utiliser LaTeX dans sa version complète avec :

```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif"      #'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'
})
```

# Matplotlib for beginners

Matplotlib is a library for making 2D plots in Python. It is designed with the philosophy that you should be able to create simple plots with just a few commands:

## 1 Initialize

```
import numpy as np  
import matplotlib.pyplot as plt
```

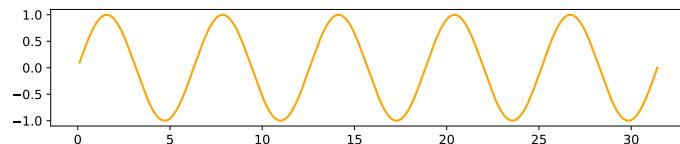
## 2 Prepare

```
X = np.linspace(0, 4*np.pi, 1000)  
Y = np.sin(X)
```

## 3 Render

```
fig, ax = plt.subplots()  
ax.plot(X, Y)  
plt.show()
```

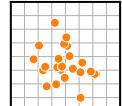
## 4 Observe



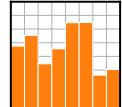
## Choose

Matplotlib offers several kind of plots (see Gallery):

```
X = np.random.uniform(0, 1, 100)  
Y = np.random.uniform(0, 1, 100)  
ax.scatter(X, Y)
```



```
X = np.arange(10)  
Y = np.random.uniform(1, 10, 10)  
ax.bar(X, Y)
```



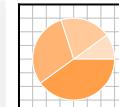
```
Z = np.random.uniform(0, 1, (8,8))  
ax.imshow(Z)
```



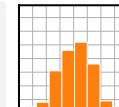
```
Z = np.random.uniform(0, 1, (8,8))  
ax.contourf(Z)
```



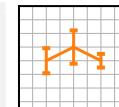
```
Z = np.random.uniform(0, 1, 4)  
ax.pie(Z)
```



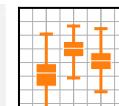
```
Z = np.random.normal(0, 1, 100)  
ax.hist(Z)
```



```
X = np.arange(5)  
Y = np.random.uniform(0, 1, 5)  
ax.errorbar(X, Y, Y/4)
```



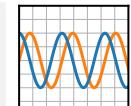
```
Z = np.random.normal(0, 1, (100,3))  
ax.boxplot(Z)
```



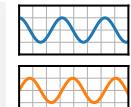
## Organize

You can plot several data on the same figure, but you can also split a figure in several subplots (named Axes):

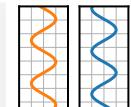
```
X = np.linspace(0, 10, 100)  
Y1, Y2 = np.sin(X), np.cos(X)  
ax.plot(X, Y1, X, Y2)
```



```
fig, (ax1, ax2) = plt.subplots(2,1)  
ax1.plot(X, Y1, color="C1")  
ax2.plot(X, Y2, color="C0")
```

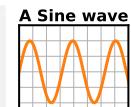


```
fig, (ax1, ax2) = plt.subplots(1,2)  
ax1.plot(Y1, X, color="C1")  
ax2.plot(Y2, X, color="C0")
```

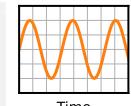


## Label (everything)

```
ax.plot(X, Y)  
fig.suptitle(None)  
ax.set_title("A Sine wave")
```



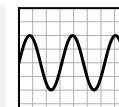
```
ax.plot(X, Y)  
ax.set_ylabel(None)  
ax.set_xlabel("Time")
```



## Tweak

You can modify pretty much anything in a plot, including limits, colors, markers, line width and styles, ticks and ticks labels, titles, etc.

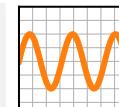
```
X = np.linspace(0, 10, 100)  
Y = np.sin(X)  
ax.plot(X, Y, color="black")
```



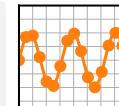
```
X = np.linspace(0, 10, 100)  
Y = np.sin(X)  
ax.plot(X, Y, linestyle="--")
```



```
X = np.linspace(0, 10, 100)  
Y = np.sin(X)  
ax.plot(X, Y, linewidth=5)
```



```
X = np.linspace(0, 10, 100)  
Y = np.sin(X)  
ax.plot(X, Y, marker="o")
```



## Explore

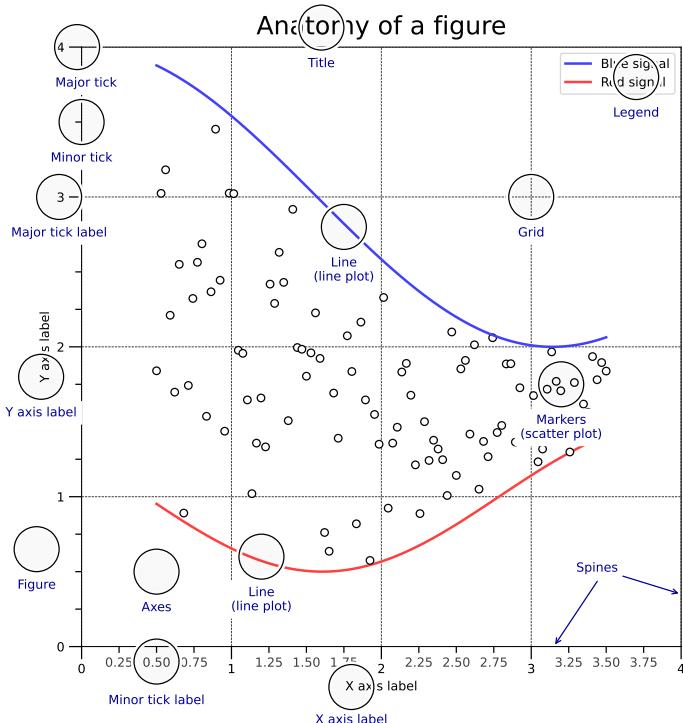
Figures are shown with a graphical user interface that allows to zoom and pan the figure, to navigate between the different views and to show the value under the mouse.

## Save (bitmap or vector format)

```
fig.savefig("my-first-figure.png", dpi=300)  
fig.savefig("my-first-figure.pdf")
```

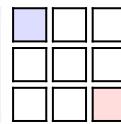
# Matplotlib for intermediate users

A matplotlib figure is composed of a hierarchy of elements that forms the actual figure. Each element can be modified.

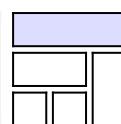


## Figure, axes & spines

```
fig, axs = plt.subplots(3,3)
axs[0,0].set_facecolor("#dddddff")
axs[2,2].set_facecolor("#fffffd")
```



```
gs = fig.add_gridspec(3, 3)
ax = fig.add_subplot(gs[0, :])
ax.set_facecolor("#dddddff")
```



```
fig, ax = plt.subplots()
ax.spines["top"].set_color("None")
ax.spines["right"].set_color("None")
```

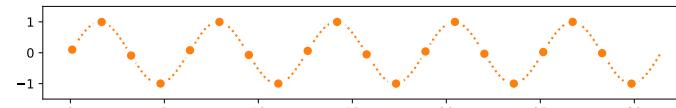


## Ticks & labels

```
from mpl.ticker import MultipleLocator as ML
from mpl.ticker import ScalarFormatter as SF
ax.xaxis.set_minor_locator(ML(0.2))
ax.xaxis.set_minor_formatter(SF())
ax.tick_params(axis='x', which='minor', rotation=90)
```

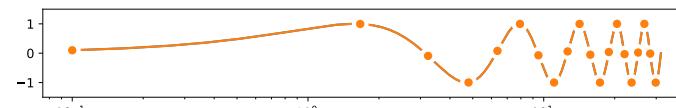
## Lines & markers

```
X = np.linspace(0.1, 10*np.pi, 1000)
Y = np.sin(X)
ax.plot(X, Y, "C1o:", markevery=25, mec="1.0")
```



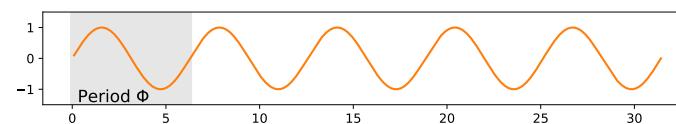
## Scales & projections

```
fig, ax = plt.subplots()
ax.set_xscale("log")
ax.plot(X, Y, "C1o-", markevery=25, mec="1.0")
```



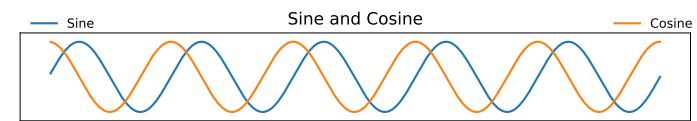
## Text & ornaments

```
ax.fill_between([-1,1],[0],[2*np.pi])
ax.text(0, -1, r"Period $\Phi$")
```



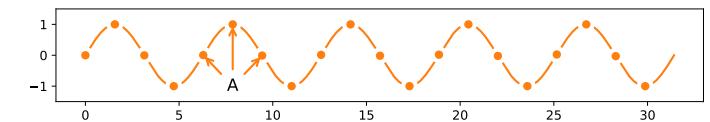
## Legend

```
ax.plot(X, np.sin(X), "C0", label="Sine")
ax.plot(X, np.cos(X), "C1", label="Cosine")
ax.legend(bbox_to_anchor=(0,1,1,.1), ncol=2, mode="expand", loc="lower left")
```



## Annotation

```
ax.annotate("A", (X[250],Y[250]),(X[250],-1),
ha="center", va="center",arrowprops =
{"arrowstyle" : "->", "color": "C1"})
```



## Colors

Any color can be used, but Matplotlib offers sets of colors:

C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9

## Size & DPI

Consider a square figure to be included in a two-columns A4 paper with 2cm margins on each side and a column separation of 1cm. The width of a figure is  $(21 - 2*2 - 1)/2 = 8\text{cm}$ . One inch being 2.54cm, figure size should be  $3.15 \times 3.15$  in.

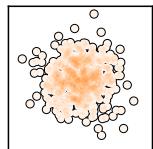
```
fig = plt.figure(figsize=(3.15,3.15), dpi=50)
plt.savefig("figure.pdf", dpi=600)
```

# Matplotlib tips & tricks

## Transparency

Scatter plots can be enhanced by using transparency (alpha) in order to show area with higher density. Multiple scatter plots can be used to delineate a frontier.

```
X = np.random.normal(-1, 1, 500)
Y = np.random.normal(-1, 1, 500)
ax.scatter(X, Y, 50, "0.0", lw=2) # optional
ax.scatter(X, Y, 50, "1.0", lw=0) # optional
ax.scatter(X, Y, 40, "C1", lw=0, alpha=0.1)
```



## Rasterization

If your figure has many graphical elements, such as a huge scatter, you can rasterize them to save memory and keep other elements in vector format.

```
X = np.random.normal(-1, 1, 10_000)
Y = np.random.normal(-1, 1, 10_000)
ax.scatter(X, Y, rasterized=True)
fig.savefig("rasterized-figure.pdf", dpi=600)
```

## Offline rendering

Use the Agg backend to render a figure directly in an array.

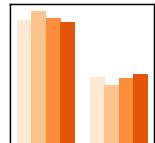
```
from matplotlib.backends.backend_agg import FigureCanvas
canvas = FigureCanvas(Figure())
... # draw some stuff
canvas.draw()
Z = np.array(canvas.renderer.buffer_rgba())
```

## Range of continuous colors

You can use colormap to pick from a range of continuous colors.

```
X = np.random.randn(1000, 4)
cmap = plt.get_cmap("Oranges")
colors = cmap([0.2, 0.4, 0.6, 0.8])

ax.hist(X, 2, histtype='bar', color=colors)
```



## Text outline

Use text outline to make text more visible.

```
import matplotlib.path_effects as fx
text = ax.text(0.5, 0.1, "Label")
text.set_path_effects([
    fx.Stroke(linewidth=3, foreground='1.0'),
    fx.Normal()])
```



## Multiline plot

You can plot several lines at once using None as separator.

```
X, Y = [], []
for x in np.linspace(0, 10*np.pi, 100):
    X.extend([x, x, None]), Y.extend([0, sin(x), None])
ax.plot(X, Y, "black")
```



## Dotted lines

To have rounded dotted lines, use a custom linestyle and modify dash\_capstyle.

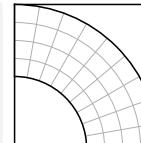
```
ax.plot([0,1], [0,0], "C1",
        linestyle = (0, (0.01, 1)), dash_capstyle="round")
ax.plot([0,1], [1,1], "C1",
        linestyle = (0, (0.01, 2)), dash_capstyle="round")
```



## Combining axes

You can use overlaid axes with different projections.

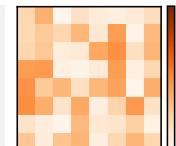
```
ax1 = fig.add_axes([0,0,1,1],
                   label="cartesian")
ax2 = fig.add_axes([0,0,1,1],
                   label="polar",
                   projection="polar")
```



## Colorbar adjustment

You can adjust a colorbar's size when adding it.

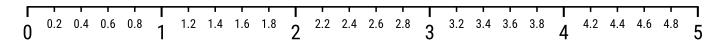
```
im = ax.imshow(Z)
cb = plt.colorbar(im,
                  fraction=0.046, pad=0.04)
cb.set_ticks([])
```



## Taking advantage of typography

You can use a condensed font such as Roboto Condensed to save space on tick labels.

```
for tick in ax.get_xticklabels(which='both'):
    tick.set_fontname("Roboto Condensed")
```



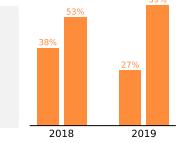
## Getting rid of margins

Once your figure is finished, you can call `tight_layout()` to remove white margins. If there are remaining margins, you can use the `pdfcrop` utility (comes with TeX live).

## Hatching

You can achieve a nice visual effect with thick hatch patterns.

```
cmap = plt.get_cmap("Oranges")
plt.rcParams['hatch.color'] = cmap(0.2)
plt.rcParams['hatch.linewidth'] = 8
ax.bar(X, Y, color=cmap(0.6), hatch="/")
```



## Read the documentation

Matplotlib comes with an extensive documentation explaining the details of each command and is generally accompanied by examples. Together with the huge online gallery, this documentation is a gold-mine.





# Chapitre 9

## Scipy et numpy pour résoudre quelques problèmes numériques courants

La bibliothèque `scipy` s'appuie lourdement sur `numpy` en interne. Il existe plusieurs sous-modules spécifiques pour certaines utilisations. À la différence de `numpy`, il faut préciser les sous-modules utilisés. Ainsi il faudra en général au minimum 2 lignes pour faire l'import :

```
1 import scipy
    import scipy.stats #statistiques
    import scipy.integrate #intégration
    import scipy.optimize #ajustement de courbe
```

La liste des différents sous-modules est disponible dans la documentation [\(i\)](#).

### 9.1 Constantes numériques

Je vous conseille très très TRÈS fortement de toujours utiliser les constantes numériques du module `scipy.constants` [\(i\)](#).

1. Avant tout car cela évite les *fautes de frappe* et donc un débogage très difficile : il n'y a rien de plus difficile que de trouver qu'un "8.314" s'est transformé en "8.134".
2. De plus, cela évite généralement tout problème numérique lié à la troncature de constantes.
3. Cependant les valeurs utilisées peuvent dépendre de la version de la librairie, par exemple, aujourd'hui, il s'agit des valeurs du CODATA 2018 mais cela peut être amené à évoluer. Pour du maintien sur le long terme, cela peut amener à une instabilité numérique des résultats. Mais cela assure un maintien avec des valeurs relativement récentes des constantes.

### 9.2 « Philosophie » des fonctions `scipy`

De manière générique, un grand nombre de problèmes va demander de faire appel à des fonctions qui opèrent sur un grand nombre de variables et paramètres :

$$f(x_1, x_2, \dots, x_n; \alpha, \beta, \gamma, \dots) \quad (9.1)$$

Et il sera courant de vouloir faire des actions uniquement par rapport à certaines variables ou paramètres. Par exemple, calculer l'intégrale de la fonction mais uniquement par rapport à la variable  $x_1$ . Il faut donc faire comprendre à `scipy` qui sera considéré en tant que paramètre, et qui sera considéré en tant que variable.

Pour cela, il faut en général :

1. mettre en premier les variables qui seront l'objet des fonctions complexes de python pour la fonction sur laquelle on veut agir (variables d'intégration, etc)
2. appeler la fonction complexe de `scipy` (par exemple `integrate` pour faire une intégration) en lui donnant comme argument :
  - la fonction sur laquelle on veut agir
  - les *paramètres* considérés comme fixes avec l'option `args = ()` suivie d'un tuple contenant les valeurs des paramètres.

Ainsi pour faire l'intégration de la fonction  $f$  par rapport à  $x_1$ , la structure globale de l'appel à `scipy` pour faire l'intégration ressemblera à ☰ :

```
int = scipy.integrate.quad(f, [...], args=(x_2, ..., x_n, alpha, beta, gamma, ...))
```

## 9.3 Ajustement de courbe

Les fonctions accessibles sont nombreuses. La syntaxe peut être plus ou moins simple en fonction de ce qui est souhaité. Pour tout ce qui est régression linéaire ou polynomiale, la fonction `np.polyfit` permet d'avoir accès aux paramètres et de faire des régressions pondérées si jamais l'incertitude sur la grandeur observable est connue. ☰

Il est également possible d'utiliser `optimize.curve_fit` et `optimize.least_squares` pour faire des régressions non-linéaires. Les deux fonctions ont des syntaxes assez différentes même si leur finalité est similaire. ☰ `least_squares` permet de fixer des bornes pour les paramètres à trouver (et utilise une syntaxe similaire à celle décrite 9.2) tandis que `curve_fit` utilise des algorithmes particuliers pour chercher les paramètres optimaux et fournit directement la covariance entre les paramètres trouvés.

 Dans tous les cas, il est recommandé d'avoir au moins une dizaine de points par paramètre à optimiser.

 De plus, pour les régressions non linéaires, il est **crucial** d'essayer de donner une estimation, même grossière des paramètres. En effet, les algorithmes d'optimisation, aussi sophistiqués soit-ils peuvent très facilement tomber dans des minimums relatifs qui ne correspondent pas forcément au valeur optimales des paramètres. On peut également tenter de faire une optimisation avec différents paramètres de départ pour essayer de voir si les paramètres finaux trouvés sont stables.

 Attention, les paramètres trouvés sont rarement indépendants, il faut donc bien prendre en compte leur covariance pour exploiter les incertitudes associées.

## 9.4 Transformée de Fourier

Numpy et `scipy` sont tous les deux capables de faire des *fast fourier transforms*. Cependant, pour un signal réel, la transformée est complexe. Il faut donc soit se ramener à un domaine réel en utilisant `np.abs` pour avoir le module et `np.angle` pour avoir la phase, soit utiliser les

fonctions dédiées pour contourner les « difficultés » associées, qui sont en général préfixées avec « `r` ». ☺

💡 La transformée de Fourier donnera la valeur moyenne du signal pour une fréquence nulle. Si cela n'est pas intéressant pour le traitement, il est alors judicieux de soustraire la composante moyenne du signal pour ne pas être « parasité » par la composante continue du signal.

## 9.5 Recherche de zéros

### 9.5.1 Fonctions unidimensionnelles

Il est également possible de trouver des paramètres permettant de satisfaire une égalité pour des équations non linéaires. Les fonctions possibles sont multiples. Dans tous les cas, on se ramène à un problème de recherche de zéro :

$$g(x) = \alpha \quad (9.2)$$

$$\Rightarrow f(x) = g(x) - \alpha = 0 \quad (9.3)$$

`optimize.root_scalar` est la fonction générique qui peut faire appel à différentes méthodes plus spécifiques. Le plus simple est de commencer par la méthode de Newton-Raphson. ⓘ Si non, la méthode `brentq` est une alternative parmi d'autres. ⓘ

Pour la plupart des méthodes :

- on suppose la fonction continue
- il est nécessaire de fournir deux points  $a$  et  $b$  pour lesquels  $f(a) < 0$  et  $f(b) > 0$  (ou vice-versa).

La complexité des algorithmes et les propriétés sont discutées dans la documentation ⓘ . La convergence n'est pas assuré pour la méthode de Newton-Raphson alors qu'elle l'est dans les autres cas. ☺

💡 Le conditionnement du problème peut parfois faciliter la résolution.

### 9.5.2 Fonctions vectorielles à plusieurs variables

Pour les fonctions multidimensionnelles, il est beaucoup plus difficile d'assurer la convergence de la méthode.

Si on a une fonction vectorielle dont on cherche des zéros :

$$f(x_1, \dots, x_n) = \begin{pmatrix} f_1(x_1, \dots, x_n) \\ f_2(x_1, \dots, x_n) \\ \vdots \\ f_m(x_1, \dots, x_n) \end{pmatrix} \quad (9.4)$$

alors la plupart des méthodes exploitent le jacobien qui est la matrice définie par le fait que chaque colonne est la dérivée de chacune des composantes de la fonction par rapport à

chaque variable. On a donc une matrice à  $m$  lignes et  $n$  colonnes :

$$jac = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \quad (9.5)$$

 **Il est toujours préférable de donner le jacobien de la fonction s'il est accessible.** Cela permet d'économiser beaucoup de temps de calcul et facilite énormément la convergence.

 Il existe de nombreuses méthodes d'optimisation, la convergence et la rapidité de celle-ci dépendent fortement du problème considéré.

 Comme pour l'optimisation de courbes, il est préférable de donner une estimation, même grossière d'une solution. Que ce soit par une estimation grossière ou des méthodes plus sophistiquées pour assurer la convergence de la solution vers une solution numérique acceptable. ☺

## 9.6 Minimisation de fonction

Quel que soit le type de minimisation (unidimensionnelle ou multidimensionnelle). Il est difficile d'être certains que les algorithmes trouvent le minimum absolu de la fonction. En effet, ils peuvent finir dans des minimum locaux.

Comme pour l'ajustement de courbe, il est d'autant plus facile de trouver le minimum que la fonction est connue pour donner un point de départ raisonnable. ☺

 Pour la minimisation de fonction à plusieurs variables, parmi les différents algorithmes disponibles, il est en général préférable (voire nécessaire) de donner le jacobien pour faciliter la convergence de l'algorithme, voir de donner le Hessien qui est l'analogie du jacobien, mais pour la dérivée seconde.

## 9.7 Intégration

Pour l'intégration, il existe plusieurs variantes en fonctions du nombre d'intégrations à effectuer (intégrale simple, double, triple, etc). IL est possible d'utiliser les fonctions quad et dérivées si jamais l'expression analytique de la fonction est connue ⓘ . Dans ce cas, scipy donne une estimation de l'erreur sur l'intégrale. ☺

Il est également possible de faire l'intégration de fonctions numériques, dans ce cas, on fait généralement une intégration de Simpson ou de Romberg ⓘ . Par contre, dans ce cas, il n'est pas possible d'avoir une estimation de l'erreur commise lors de l'intégration.

## 9.8 Équations différentielles ordinaires

### 9.8.1 Équation différentielle scalaire du premier ordre

Pour une équation différentielle du premier ordre de la forme :

$$\frac{dy}{dt} + \frac{1}{\tau}y = f(t)$$

Il faut ré-écrire l'équation différentielle sous la forme :

$$\frac{dy}{dt} = \underbrace{f(t) - \frac{1}{\tau}y}_{=g(t,y)}$$

Il faudra ensuite créer une fonction  $g(t,y)$  qui donne la dérivée et scipy va s'occuper de faire l'intégration pour trouver  $y(t)$ . La fonction pour faire la résolution de l'équation différentielle doit obligatoirement suivre la structure suivante :

- le premier argument de la fonction doit être la variable d'intégration
- Le deuxième argument de la fonction doit être la fonction sur laquelle porte l'intégration  $y(t)$ ,

Un exemple est présenté dans le cahier jupyter associé. ☀

💡 Les coefficients peuvent ne pas être constants pour l'équation homogène.

### 9.8.2 Équation différentielle vectorielle du premier ordre

Dans ce cas, la démarche est rigoureusement identique, la différence principale étant que la fonction  $g(t,y)$  sur laquelle porte l'intégration est vectorielle. Il y aura donc autant de composantes que la taille du vecteur.

💡 Les équations peuvent coupler différentes composantes du vecteur.

### 9.8.3 Équation différentielle du second ordre

Pour une équation différentielle du second ordre, il faut transformer l'équation scalaire en équation vectorielle à 2 composantes.

Ainsi, on va transformer l'équation différentielle du second ordre de la forme :

$$\ddot{u} + \frac{\omega_0}{Q}\dot{u} + \omega_0^2 u = f(t)$$

en équation vectorielle qui porte sur le vecteur  $\mathbf{v} = \begin{pmatrix} u \\ \dot{u} \end{pmatrix}$  sous la forme :

$$\begin{pmatrix} \frac{d u}{d t} \\ \frac{d \dot{u}}{d t} \end{pmatrix} = \begin{pmatrix} \frac{d \mathbf{v}[0]}{d t} \\ \frac{d \mathbf{v}[1]}{d t} \end{pmatrix} = \begin{pmatrix} \dot{u} \\ \ddot{u} \end{pmatrix} = \begin{pmatrix} -\frac{\omega_0}{Q} \underbrace{\dot{u}}_{\mathbf{v}[1]} & \mathbf{v}[1] \\ -\omega_0^2 \underbrace{u}_{\mathbf{v}[0]} & +f(t) \end{pmatrix}$$

L'astuce étant d'utiliser la deuxième coordonnée du vecteur comme valeur de la dérivée de la première composante. On fait donc « glisser » la dérivée première, qui est la deuxième coordonnée de  $\mathbf{v}$  en tant que coordonnée de la dérivée de la première composante du vecteur.

💡 Les coefficients peuvent ne pas être constants pour l'équation homogène.

## 9.9 Fonctions statistiques

à faire : cdf, ppf, pdf

## 9.10 Arrangement, combinaison

itertools

**Troisième partie**

**Pour aller plus loin**



# Chapitre 10

## Les expressions régulières

Les expressions régulières (*Regular expressions* ou *RegExp*) sont un système de rechercher/remplacer très poussé, leur puissance fait que le principe de base est quasiment systématiquement implémenté dans les différents langages de programmation récents (Perl, PHP, Javascript, Python, Go, Ruby, etc). Leur puissance fait que ce système est également implémenté dans de nombreux IDE/éditeurs de texte.

Il y a donc deux choses à apprendre :

- la syntaxe générale pour les expressions régulières (généralement commune à tous les langages de programmation sur le fonctionnement de base)
- la librairie qui les gère sous python (module « `re` » comme Regular Expression)

L'objectif des expressions régulières est de trouver/remplacer une recherche stricte ("toto" devient "tata") par une recherche sur des motifs qui peuvent prendre une forme beaucoup plus subtile et évoluée, par exemple, il est possible de demander :

- de remplacer un point par une virgule uniquement si il y a un chiffre placé avant et/ou après. (pour remplacer le séparateur décimal anglo-saxon par celui utilisé en typographie française par exemple). Un rechercher remplacer sur un texte complet serait susceptible de remplacer également les points utilisés en tant que ponctuation dans le texte.
- de transformer les puissances de 10 en notation python "1.3e-03" en équivalent LATEX `1.3 \cdot 10^{-3}`, et ce quel que soit la puissance de 10, en UNE SEULE fois.
- trouver une énergie dans différents fichiers, mais que si le mot-clé qui le précède est donné pour la stocker dans un nouveau fichier sous une forme plus compacte.

 Il est parfois possible d'utiliser d'autres outils que les expressions régulières pour des finalités proches (comme les programmes `awk`, `grep` ou `sed`, voir les exercices en bash du cours). Mais leur apprentissage peut parfois être long. Ainsi, même si les expressions régulières sont coûteuses en ressource et qu'il peut exister des alternatives plus efficaces. La maîtrise de ces solutions alternatives ne justifie pas toujours un investissement qui soit toujours rentable.

 Comme tout outil puissant, les expressions régulières ont un coût en temps de calcul non négligeable, il sera donc TOUJOURS préférable de faire une recherche simple si elle suffit et si jamais elle ne suffit pas, de faire en sorte d'avoir des expressions les plus directives et restreintes possibles pour gagner autant que possible sur le temps de calcul correspondant (j'anticipe, mais éviter les « `.*` » à outrance). **Si un rechercher/remplacer simple suffit, alors il faut l'utiliser.**

 Les expressions régulières étant très utiles et pratique elles sont implémentées dans beaucoup de langage de programmation mais l'implémentation technique précise peut légèrement différer d'un langage à l'autre, il faut donc vérifier la compatibilité de la syntaxe,

en particulier en cas d'utilisation de fonctionnalités « exotiques » (assertion *lookahead*, nommage des groupes, *flags*, etc)

Il est également fréquent d'avoir des programmes (excel, éditeurs de texte, IDE python, etc) qui proposent un mode spécifique pour pouvoir faire des rechercher/remplacer avec des expressions régulières. Il n'est donc pas toujours nécessaire de passer par python ou un langage de programmation pour utiliser des expressions régulières.

## 10.1 Fonctionnement général des expressions régulières

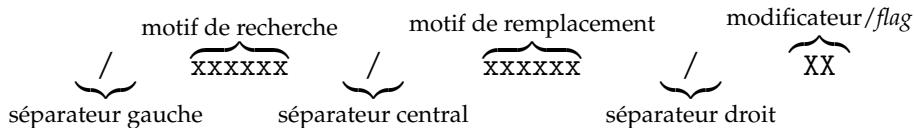
### 10.1.1 Allure usuelle d'une expression régulière

Dans leur syntaxe la plus courante, les expressions régulières ont toujours la même structure :

- (un délimiteur gauche)
- un motif de recherche
- (un délimiteur central)
- (un motif de remplacement)
- (un délimiteur droit)
- un modificateur (*flag*) pour préciser le mode de recherche

Python utilise une syntaxe où chaque élément (motif de recherche, motif de remplacement et *flag*) est précisé séparément et il n'y a pas de séparateur donné explicitement.

On aura donc une expression régulière qui aura le plus souvent la forme suivante dans différents langages de programmation :



**10.1.1.0.1 Séparateur pour les langages de programmation autres que python** Pour les autres langages de programmation, le séparateur peut être quelconque et le plus souvent on trouve le slash /, mais si le séparateur apparaît très souvent dans la chaîne de recherche (par exemple le / si on travaille sur du code HTML) alors il est possible de mettre autre chose comme un \$ ou un !. Les délimiteurs gauche, central et droit doivent être identiques.

### 10.1.2 Le *backtracking*

Pour les expressions régulières, le moteur va rechercher progressivement le motif indiqué pour vérifier si l'expression recherchée est présente dans la chaîne de caractère sur laquelle on l'exécute. Cependant, comme les motifs sont plus souples, il se peut que le moteur ait des fois à revenir en arrière pour faire un nouvel essai pour voir si en interprétant légèrement différemment la recherche parmi les différentes possibilités il est possible de trouver une correspondance. Le fait de revenir en arrière est appelé le *backtracking*. Le fait d'avoir à potentiellement faire beaucoup d'essais pour trouver une correspondance peut rendre l'exécution de l'expression régulière extrêmement longue. Ainsi, plus le motif recherché pourra être vaste (. \* en particulier, qui correspond à rechercher un enchaînement d'une longueur quelconque de n'importe quel caractère autre qu'un saut de ligne) peut rendre très longue et lente l'expression régulière. L'optimisation des expressions régulières peut ainsi prendre un temps non négligeable si jamais il est nécessaire d'en faire beaucoup et cet aspect ne doit pas être pris à la légère !

Il faut ainsi veiller à ne tenter des reconnaissances via expression régulière que dans un contexte le plus restreint possible pour gagner du temps d'exécution par exemple en ne faisant pas la recherche par expression régulière dans une portion de fichier dont on est sûr qu'elle ne donnera rien.

## 10.2 Classes de caractère

La puissance des expressions régulières vient en partie de leur capacité à faire une recherche non pas sur un caractère strict mais sur des classes de caractères. Les classes peuvent être personnalisées ou génériques.

Pour définir une classe de caractères personnalisée, il faut employer des crochets [] et mettre entre crochets tous les caractères ou classe de caractère recherchés. Par exemple [aeiouy] définit une classe de caractère qui correspond à toutes les voyelles de la langue française. C'est la syntaxe la plus basique. Il existe également des classes pré-définies, celles-ci sont fournies dans le tableau 10.1.

 Les caractères spéciaux comme +, ( ne sont plus spéciaux à l'intérieur des crochets, il n'est pas nécessaire de les échapper.

 Il est possible d'utiliser le tiret - pour définir des intervalles, soit de lettres, soit de chiffres. Par exemple [a-e] est équivalent à [abcde].

---

### Classe Correspondance

---

#### Caractères numériques

---

- \d      usuellement, cela correspond à la classe [0-9], mais en pratique, cela inclue également quelques autres caractères unicode associés à des chiffres dans des langues non latines
  - \D      négation de \d donc équivalent à [^\d], ou de manière un peu moins rigoureuse à [^0-9]
- 

#### Caractères « blancs » ou d'espacement

---

- \s      correspond à tous les caractères de séparation (espaces, retours à la ligne (\r, \n), tabulations (\t,\v) et saut de page \f) correspond à [ \t\n\r\f\v]
  - \S      correspond à la négation de la classe précédente équivalent à [^\s] ou [^ \t\n\r\f\v]
- 

#### Caractères alphanumériques

---

- \w      Caractères alphanumériques, équivalent à [a-zA-Z0-9\_]
  - \W      Caractères non alphanumériques, équivalent à [^a-zA-Z0-9\_]
- 

#### Classe universelle

---

- .      n'importe quel caractère autre qu'un retour à la ligne \n
- 

TABLEAU 10.1 – Quelques classes de caractère couramment utilisées pour les expressions régulières. Attention, pour des classes de caractère exotiques (\b, \B), il peut y avoir de légères variations en fonction des langages de programmation.

## 10.3 Quantificateurs

Il est possible de spécifier le nombre d'occurrences lors de la recherche,  $\{n,m\}$  indiquera que le caractère ou la classe de caractère doit apparaître un nombre de fois compris entre  $n$  et  $m$  inclus. Il est ensuite possible d'avoir plus de souplesse pour préciser un nombre précis d'occurrences, une valeur minimale sans valeur maximale ou l'inverse.

Quantificateur	Explication
$\{n,m\}$	$n \geq x \geq m$ avec $x$ le nombre d'occurrences, comportement <i>greedy</i> (voir section 10.3.1)
$\{n,\}$	$n \geq x$ avec $x$ le nombre d'occurrences
$\{,m\}$	$0 \geq x \geq m$ avec $x$ le nombre d'occurrences
$\{m\}$	$x = m$ avec $x$ le nombre d'occurrences
$\{n,m\}^+$	$n \geq x \geq m$ avec $x$ le nombre d'occurrences, essaye de capturer le maximum possible sans mécanisme de retour en arrière (backtracking) pour maximiser la probabilité de réussite de reconnaissance du motif (peut permettre un gain de temps d'exécution)
*	Équivalent de $\{0,\}$ , comportement <i>greedy</i> (voir section 10.3.1)
+	équivalent de $\{1,\}$ , comportement <i>greedy</i> (voir section 10.3.1)
?	équivalent de $\{0,1\}$ , comportement <i>greedy</i> (voir section 10.3.1)
$?; +?; *?; \{n,m\}?$	idem que ci-dessus mais comportement non <i>greedy</i> (voir section 10.3.1)
$?+; ++; **$	idem que ci-dessus mais comportement <i>greedy ne permettant pas le backtracking</i> .

TABLEAU 10.2 – Quelques quantificateurs possibles pour les expressions régulières sous Python.

### 10.3.1 Comportement *greedy* ou non

Pour les expressions régulières, il existe deux comportements différents :

- Le comportement *greedy* qui va essayer de prendre le plus de caractères possibles, c'est le comportement par défaut;
- Le comportement *non greedy* qui va essayer de prendre le moins de caractères possibles.

Pour rendre les quantificateurs non *greedy*, il faut leur ajouter un `?` après.

Par exemple, pour la chaîne de caractère `aaaaaa`, la différence entre les deux quantificateurs sera que `a*` prendra « `aaaaaa` » si possible alors que `a*?` ne prendra ... rien du tout !

💡 Le `+` ajouté après un quantificateur est un opérateur *greedy*, mais en empêchant le backtracking, cela peut aboutir à un comportement qui impacte la reconnaissance ou non de l'expression régulière complète, ainsi, pour la chaîne de caractère `aaaaaab` :

- `a*b` aboutit à un succès de la reconnaissance de l'expression régulière ;
- `a*ab` aboutit à un succès de la reconnaissance de l'expression régulière car le moteur est capable de revenir en arrière pour ne prendre que quatre « `a` » pour ensuite prendre le `ab` restant ;
- `a*+ab` aboutit à un échec car `a*+` a déjà reconnu et utilisé `aaaaa` il n'y a alors plus de « `a` » supplémentaire disponible ce qui entraîne donc un échec car il n'est pas possible de revenir en arrière pour essayer de ne prendre que `aaaa` pour essayer de maximiser la reconnaissance du motif.

## 10.4 Les parenthèses

### 10.4.1 Les parenthèses capturantes

Une fois la recherche effectuée, il est possible de « capturer » une partie du motif. Cela va permettre de pouvoir replacer le contenu capturé dans le motif de remplacement. Par exemple, pour passer de la notation scientifique dans Python à la notation équivalente sous L<sup>A</sup>T<sub>E</sub>X il faut passer de  $7.34e-06$  à  $7,34\cdot 10^{-6}$ . Il faut donc capturer quatre morceaux : le 7, le 34, le  $-$  et le 6 pour ensuite placer une virgule entre le 7 et le 34, la partie  $\cdot 10^{-6}$  entre le 34 et le  $-$ , puis le 6 puis  $\cdot$ . C'est ici que les parenthèses capturantes vont jouer leur rôle pour indiquer quel morceau capturer et où les placer.

Comme le nom de la section l'indique, l'action de capture va se faire via des ... parenthèses.<sup>a</sup> Ainsi, il suffit de placer des parenthèses autour du motif à capturer. Ensuite, pour le remplacement, le plus généralement, on utilise  $\backslash 1$ ,  $\backslash 2$ , etc où le chiffre indique l'ordre de la parenthèse capturante par ordre d'ouverture.

 Si jamais c'est nécessaire, il est possible de donner des noms plus explicites en le précisant avec la syntaxe `(?P<nomDuGroupe>)` qui permettra d'utiliser `(?P=nomDuGroupe)` à l'intérieur du motif en lui-même, où `\g<nomDuGroupe>` pour un remplacement (à la place de  $\backslash 1$ ). Cependant, cette syntaxe est spécifique à Python et ne marche pas forcément pour tous les langages de programmation.

---

a. Oui, le suspens était insoutenable!

### 10.4.2 Les parenthèses « autres »

Il est aussi possible de donner d'autres sens aux parenthèses. Ces utilisations sont beaucoup moins courantes que les parenthèses courantes. Les différentes syntaxes sont données dans le tableau 10.3 à titre indicatif en cas de besoin.

Syntaxe	Explication
(?P<nomDuGroupe>)	Parenthèses capturantes pour donner un nom explicite au groupe capturé
(?P=nomDuGroupe)	Référence à un groupe déjà capturé en amont dans la même expression régulière avec (?P<nomDuGroupe>)
<i>lookahead</i>	
(?=motif)	Permet une affirmation <i>lookahead</i> : permet de vérifier si le motif <code>motif</code> est placé <i>après</i> la partie correspondante de l'expression régulière, mais la parenthèse n'est <b>pas</b> capturante de <code>motif</code> . <code>/e(?=\\d)/</code> permettra ainsi de repérer uniquement les "e" immédiatement suivis d'un chiffre.
(?!motif)	Permet une affirmation <i>negative lookahead</i> : permet de vérifier si le motif <code>motif</code> n'est <b>pas</b> placé <i>après</i> la partie correspondante de l'expression régulière, mais la parenthèse n'est <b>pas</b> capturante de <code>motif</code> . Ainsi, <code>/e(?!\\d)/</code> permettra ainsi de repérer uniquement les "e" immédiatement suivis par autre chose qu'un chiffre.
<i>lookbehind</i>	
(?<=motif)	Permet une affirmation <i>lookbehind</i> : permet de vérifier si le motif <code>motif</code> est placé <i>avant</i> la partie correspondante de l'expression régulière, mais la parenthèse n'est <b>pas</b> capturante de <code>motif</code> . <code>/(?&lt;=\\d)e/</code> permettra ainsi de repérer uniquement les "e" immédiatement précédés d'un chiffre.
(?<!\motif)	Permet une affirmation <i>negative lookbehind</i> : permet de vérifier si le motif <code>motif</code> n'est <b>pas</b> placé <i>avant</i> la partie correspondante de l'expression régulière, mais la parenthèse n'est <b>pas</b> capturante de <code>motif</code> . <code>/(?&lt;!\\d)e/</code> permettra ainsi de repérer uniquement les "e" immédiatement précédés par autre chose qu'un chiffre.
(?:motif)	Permet de faire des parenthèses non capturantes.

TABLEAU 10.3 – Les différents types de parenthèse possibles.

Il est aussi possible de spécifier les *flags* via des parenthèses à l'intérieur de l'expression régulière (voir section 10.6).

## 10.5 Caractères spéciaux

Caractère	Sens
-	intervalle au sein d'une classe de caractères définie avec des crochets
[]	à l'intérieur d'une classe [] : négation de ce qui suit
{}	délimiteurs pour une classe de caractère personnalisée
{}	délimiteurs pour la quantification du nombre d'occurrences acceptées
+ * ?	quantificateurs (voir section 10.3)
	opérateur « ou » : a b correspond à « a » ou « b »
~ ou \A	Le début de la chaîne de caractère
\$ ou \z ou \z	La fin de la chaîne de caractère (avant le retour à la ligne si applicable)
()	Parenthèses capturantes ou pour des assertions de type <i>lookahead</i> ou <i>lookbehind</i> (section 10.4).
\	Annonce un groupe capturé auparavant comme \1 (section 10.4) ou une classe de caractère.

TABLEAU 10.4 – Quelques caractères spéciaux pour l'utilisation d'expressions régulières.

Pour des chaînes qui contiennent des \, il est nécessaire de les échapper une première fois pour qu'ils ne soient pas pris comme annonciateurs d'une classe par le compilateur puis échappés une nouvelle fois dans la chaîne de caractères pour qu'ils puissent être pris en compte lors de la compilation. Cela veut dire qu'il faut utiliser \\\" pour échapper un seul backslash dans le motif à rechercher. Il est pour cela possible d'utiliser r"\\" qui ne demandera qu'un unique échappement r"\\" au lieu de "\\\".

## 10.6 Modificateurs/flags

Pour les différentes fonctions, il est possible de spécifier un *flag* pour préciser ou altérer légèrement le comportement de l'expression régulière. Dans d'autres langages de programmation, il arrive de mettre le *flag* à la fin de l'expression régulière. On peut utiliser les versions abrégées via re.z ou z correspond au *flag* utilisé.

## 10.7 Opérateurs

Sans forcément rentrer dans toutes les possibilités offertes par la librairie re, il y a deux utilisations principales :

- trouver un motif;
- remplacer un motif par un autre.

### 10.7.1 Trouver un motif

Il existe différentes variantes :

- re.match() qui ne cherchera que les motifs qui commencent à partir du début de la chaîne de caractère, cela correspond à avoir ~ ou \A au début du motif recherché.

<i>flag</i>	Explication
ASCII, A	Pour différentes classes de caractère, seuls les caractères de type ASCII seront reconnus (au lieu des caractères unicode). Cela peut permettre d'accélérer légèrement l'exécution de la recherche, mais fait perdre un peu de souplesse.
DOTALL, S	fait en sorte que le caractère « . » englobe également un retour à la ligne.
IGNORECASE, I	Rendre la recherche insensible à la casse. (majuscules/minuscules)
LOCALE, L	Faire une reconnaissance qui prend en compte les caractères correspond à la langue définie en tant que LOCALE. peu usité. MULTILINE, M
VERBOSE, X	Autorise des expressions régulières plus verbeuses, qui peuvent être organisées sur plusieurs lignes pour être plus lisibles et compréhensibles.

TABLEAU 10.5 – Quelques flags disponibles sous Python.

- `re.search` qui cherchera le motif quel que soit sa position. C'est donc une version plus souple que `match`.
- `re.findall` et `re.finditer` qui permettent de retourner *tous* les motifs trouvables, `re.findall` ne retourne qu'une liste des chaînes de caractères trouvées qui correspondent au motif tandis que `re.finditer` retourne un objet itérable qui contient lui-même des objets de type `Match` qui sont des objets correspondant à des motifs trouvés par expression régulière et sont donc plus « complets ».

### 10.7.2 Rechercher/remplacer un motif

Il est possible d'utiliser :

- `re.sub` qui effectue le remplacement, par défaut, le remplacement est global (toutes les occurrences du motif sont remplacées). Mais il est possible de limiter le nombre de remplacement.
- `re.subn` qui est totalement similaire mais retourne la chaîne de caractère finale et le nombre de remplacement effectués.

# **Chapitre 11**

## **Tableaux non numériques**

pandas



# Chapitre 12

## Les gestionnaires de version : git/github

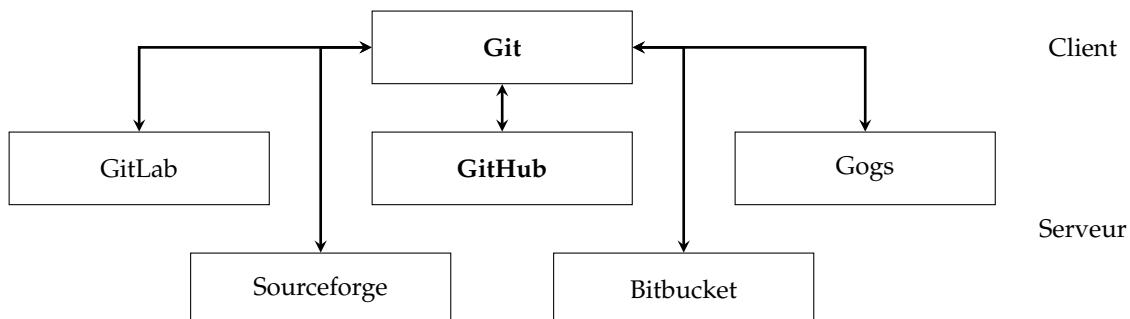
*La rédaction de ce tutoriel a largement utilisé les tutoriels de la compagnie Atlassian <https://www.atlassian.com/git/tutorials>.*

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-utilisées 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 12.1).



**FIGURE 12.1 – Schéma de principe de la solution Git/GitHub. Quelques solutions alternatives à GitHub pour le côté serveur sont également indiquées.**

## 12.1 Vocabulaire

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

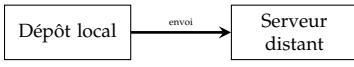
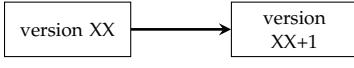
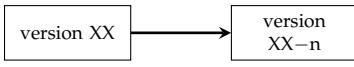
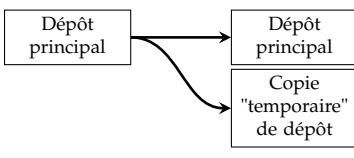
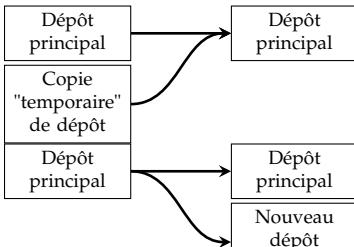
Appellation	Schéma	Commentaire
dépôt/ repo/ repository	fichier 1 fichier 2 fichier 3	Ensemble de fichiers correspondant à un projet, sera géré avec Git et synchronisé sur GitHub
push		Envoi du code de votre dépôt, le plus souvent vers la partie serveur
pull		Téléchargement du code, le plus souvent depuis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

TABLEAU 12.1 – Quelques termes associés au gestionnaire de version git.

## 12.2 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 fichier en local et le dépôt se fait donc en deux phases (figure 12.2) :

- 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.

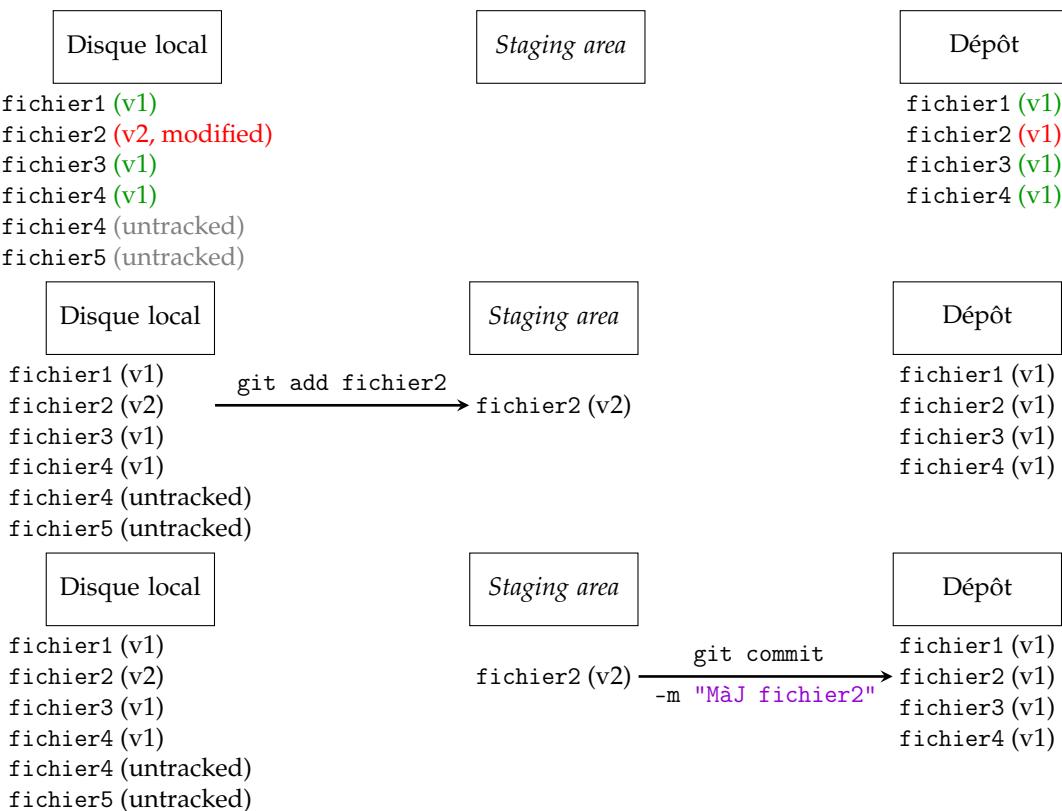


FIGURE 12.2 – Les différentes zones et un exemple de situation courante. Les fichiers 4 et 5 ne font pas partie du dépôt, les fichiers 1, 3 et 4 sont à jour mais le fichier 2 n'est pas à jour dans le dépôt. La synchronisation du dépôt va se faire en deux temps : une phase de staging et une phase de commit.

## 12.3 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.

### 12.3.1 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
```

### 12.3.2 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`.

### 12.3.3 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 autrefichier.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`. La commande `git reset --keep` permet de remettre à zéro la *Staging area*.

 Pour remettre dans la *staging area* tous les fichiers mis à jour, il est possible d'utiliser `git ls-files --modified | xargs git add`, pour également enlever les fichiers supprimé en plus de ceux modifiés : `git add -u`

**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 12.3.5). 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.

#### 12.3.4 Mise à jour effective du dépôt : commit

**Statut des fichiers** `git status` Jusqu'à présent, nous avons ajouté des fichiers en *staging area* (figure 12.2), 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)
4       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)
9       modified:   exo/exoZa-algebre.ipynb
```

```
Untracked files:
```

14

```
(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(+)
 3 create mode 100644 testbis.md
```

Sur la première ligne, la branche est indiquée (voir section 12.3.6), 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"
```

Cependant, il faut faire attention à ne pas ajouter des fichiers contenant des mots de passe ou informations sensibles. Pour cela, il est possible de spécifier dans un fichier appelé `.gitignore`

### 12.3.5 Se déplacer dans l'historique : reset, revert, checkout

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. Il existe de nombreux outils visuels pour simplifier les opérations, en particulier gitk qui est disponible sur différentes plateformes et permet de voir plus simplement l'historique d'un projet et d'y naviguer. Cette partie va mélanger les deux aspects : ligne de commande et interface graphique pour pouvoir faire le lien entre les deux.

#### 12.3.5.1 Notion d'historique

*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 12.3** – 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.

#### 12.3.5.2 Les trois sections de Git

Git utilise trois aspects différents pour la gestion des différents fichiers. Cela correspond plus ou moins à la vision de la figure 12.2. Il faut légèrement revenir sur les notions associées pour mieux comprendre comment se déplacer dans l'historique.

**12.3.5.2.1 L'espace de travail/Working Directory** Cela correspond aux fichiers tels qu'ils sont visibles sur le disque à un instant donné. Cela correspond aux fichiers listés par la commande `ls` avec leur contenu à l'instant où la commande est utilisée.

En naviguant dans l'historique ou les branches (section 12.3.6), Git peut modifier cet espace de travail en revenant à des « images » faites lors de différents commit.

**12.3.5.2.2 La Staging area/Staged snapshot** Cette zone correspond à la zone de mise en attente et est visible avec la commande `git status`. Elle permet de voir les fichiers qui feront partie du prochain commit.

**12.3.5.2.3 L'historique des commit/Commit History** L'historique des commit est l'endroit où git enregistre les « images » des fichiers du dépôt lors des commit. Il va être possible de naviguer au sein de cet historique et accéder à l'état/au contenu des fichiers lors de ces commit. En pratique, cet historique est donc bien plus riche que ce que laisse supposer la figure 12.2 car il y a une dimension temporelle supplémentaire du contenu du dépôt à différents instant.

### 12.3.5.3 Visualiser l'historique

Pour visualiser l'historique, la commande la plus simple est

```
git log
```

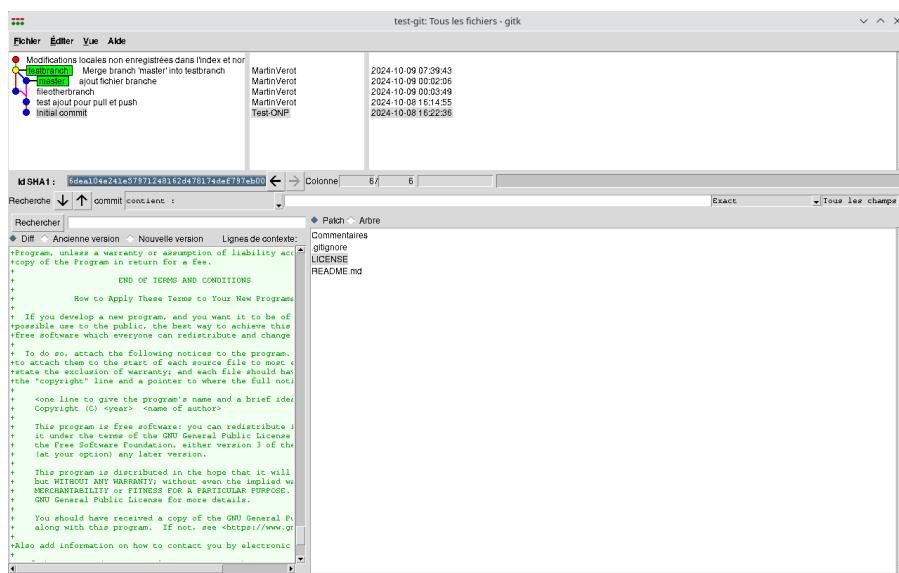
On a alors la liste des commit avec la date, l'auteur, les messages de commit et une clé qui permet d'identifier chaque commit. Il s'agit du "*commit-id*", qui est une clé SHA1 de certaines informations qui rend cet identifiant normalement unique pour une modification donnée. C'est la version la plus exhaustive.

Il est aussi possible d'utiliser la commande

```
git reflog
```

Qui précise les premiers caractères du *commit-id*, le commentaire et aussi la référence en position relative par rapport au HEAD actuel.

Avec le programme `gitk` (normalement installé par défaut avec git) ou équivalent, l'historique apparaît sous forme graphique avec les commit-id précisés. Il est également possible d'avoir simplement accès aux évolutions de chaque fichier concerné par le commit (figure 12.4). Cet outil permet d'avoir une navigation facilitée dans l'historique et peut permettre d'avoir accès aux diff ou aux fichiers concernés dans leurs différentes versions en quelques clics. À vous de voir si cela suffit pour ce que vous voulez faire.



**FIGURE 12.4 – Visualisation de l'historique des commits et des fichiers concernés dans le commit grâce au logiciel gitk.** L'historique est présenté sous forme graphique en haut à gauche, les infos sur les commits sont données en haut à droite. La partie en bas à gauche permet de voir les fichiers ou le diff entre fichiers. La partie en bas à droite permet de sélectionner les fichiers à regarder. Le commit-id est donné au milieu à gauche.

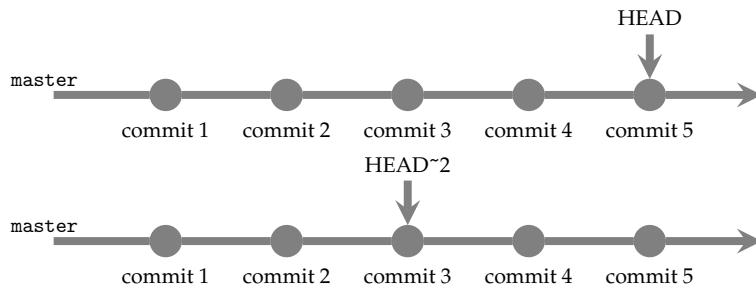
### 12.3.5.3.1 Le HEAD, son sens et comment se déplacer dans l'historique

Git utilise un nom particulier pour indiquer où nous sommes placés sur l'historique des commit : le « HEAD ». Le plus souvent, le HEAD sera placé sur l'endroit le plus récent de l'historique des commit pour refléter l'état le plus récent du dépôt. Lorsque ce n'est pas le cas, on parle de *detached HEAD*. Le passage en *detached HEAD* est en général un état transitoire pour pouvoir revenir en arrière ou voir l'état des fichiers dans un état passé.

Il est possible de naviguer dans l'historique de manière absolue ou relative par rapport à l'état où le HEAD est placé au bout de la branche (situation « normale »).

Pour un déplacement relatif, on précise de combien de commit on veut remonter avec le caractère « ~ ».

💡 Il est aussi possible d'utiliser le symbole « ^ » pour remonter le long de branches multiples, mais cela est en général plus rarement utilisé et plus souvent source de confusion. Pour plus de précisions, vous pouvez par exemple aller sur [cette discussion stackoverflow](#).



**FIGURE 12.5 – Notion de HEAD.** En haut, le HEAD est placé au niveau de la flèche verticale à la fin de la branche master, sur le dernier commit, c'est la situation la plus courante. En bas, Le HEAD (flèche verticale) est placé deux commit en amont du dernier commit, on se trouve alors dans une situation de "detached head". Pour déplacer le HEAD de la situation du dessus à cette nouvelle situation, on dira qu'on se positionne à HEAD~2 (le ~ indique qu'on remonte le long de l'historique de la branche, le 2 indique de combien de commit on remonte dans l'historique.)

Pour se déplacer dans l'historique, il est possible d'utiliser les commandes suivantes :

```
git checkout **commit-id complet (donné par git log)**
git checkout **commit-id court (donné par git reflog)**
git checkout HEAD@{XX}**chiffre donné par git reflog)**
```

<sup>5</sup> git checkout HEAD~XX \*\*Pour remonter de XX commit dans l'[historique](#)\*\*

💡 Si vous êtes dans l'état *detached head*, il est déconseillé de faire un nouveau commit car celui-ci ne sera rattaché à aucun branche. Vous ne perdrez pas l'historique des commits, mais intégrer proprement ce commit dans l'historique sera plus technique. Il est préférable de préalablement créer une nouvelle branche avec la commande `git switch -c new-branch` puis la fusionner ensuite avec la branche principale avec la commande `merge` (voir section [12.3.6](#)).

💡 Il est par exemple possible de copier de vieux fichiers lorsque vous êtes en *detached head* puis revenir au dernier commit si vous voulez récupérer des fichiers.

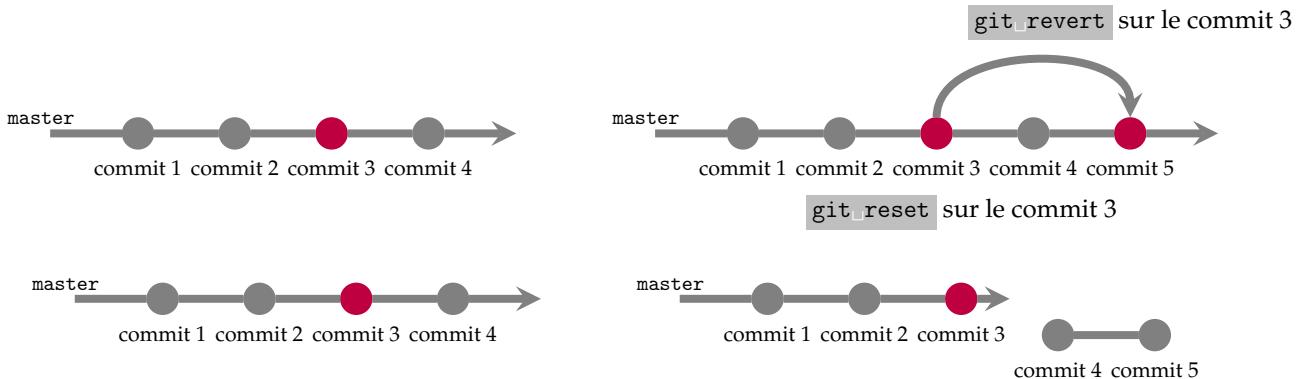
💡 Pour sortir de l'état de *Detached HEAD*, vous pouvez utiliser une des deux commandes ci-dessous :

```
git checkout master (ou le nom de votre branche)
git checkout -`
```

#### 12.3.5.3.2 Revenir dans l'état d'un commit antérieur avec `restore`, `revert` et `reset`

Il existe trois commandes pour ramener le dépôt à un état antérieur en cas de besoin :

- `git restore` : qui permet de rétablir un fichier ou groupe de fichier à partir d'un commit précédent.
- `git revert` correspond à un nouveau commit dans l'historique qui fait revenir le dépôt (et pas un unique fichier comme `git restore`) à un état antérieur mais poursuit l'historique des commit.
- `git reset` avec trois variantes différentes. `git reset` dans son utilisation la plus courante permet de revenir au dépôt dans un état antérieur dans l'historique **en mettant de côté tous les commit ayant eu lieu après**.



**FIGURE 12.6 – En haut :** action d'un `git revert` : on revient à la situation du dépôt lors du commit 3 via un nouveau commit dans l'historique, le contenu du commit 4 sera donc toujours (facilement) accessible. **En bas,** action d'un `git reset` : on est revenu à l'état du commit 3 et tous les commits ultérieurs sont devenus orphelin et donc à minima difficilement accessibles (et à terme automatiquement supprimés).

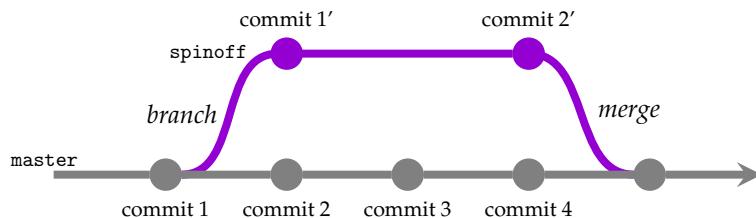
**Les différentes variantes de `git reset`** Par défaut, la commande `git reset` utilise l'option `--mixed` mais existe dans trois grandes versions :

- `--soft` Dans ce cas, on revient au commit précédent, mais toutes les modifications effectuées entre-temps sont dans le *working directory* et sont également en *staging area*. Donc si on refait un commit juste après un `reset --soft`, on se retrouve dans l'état initial.
- `--mixed` Dans ce cas, on revient à l'état précédent pour le dépôt, mais sans avoir gardé en *staging area* les modifications depuis ce commit. Les modifications sont par contre gardées dans le *working directory*. Pour revenir à l'état initial après un `reset --mixed`, il faudrait ré-ajouter tous les fichiers modifiés avec `git add` puis faire un commit.
- `--hard` Dans ce cas, tous les fichiers sont remis dans l'état dans lequel ils étaient lors du commit spécifié. Donc toutes les modifications du dépôt effectuées dans le *Working Directory* et toutes les modifications ultérieures **sont perdues**. De par la nature destructive de cette opération, elle est donc déconseillée lorsque vous débutez.

### 12.3.6 Branch/Merge : créer des variations du dépôt

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 12.7 – Illustration de la notion de *branch* et de *merge* avec la convention du dessin précédent.** On créé 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 les commandes :

```
git checkout **branche1** (optionnel si vous y êtes déjà)
git merge **branche2**
```

où `**branche1**` est la branche à garder et `**branche2**` est le nom de la branche à fusionner.

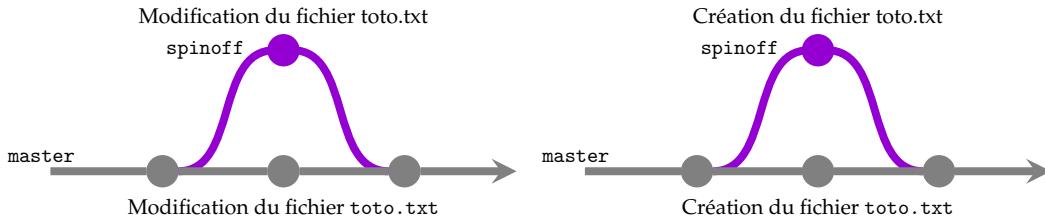
💡 Il est conseillé de faire un commit sur les différentes branches à fusionner avant le merge pour éviter toute perte de donnée.

💡 Après un merge réussi, il est possible de supprimer la branche fusionnée avec la commande

```
git branch -d **branche2**
```

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 aucun moyen de déterminer automatiquement quel est le bon fichier à conserver.
- Le même fichier a été modifié sur chaque branche.



**FIGURE 12.8 – Exemple de situation de conflit lors d'un merge.** À gauche : on a deux modifications concourantes du même fichier sur deux branches distinctes. À droite, création de deux fichiers ayant le même nom mais pas le même contenu.

Dans ces différents cas, il faut résoudre les conflits. Git ne peut pas intuiter la où les bonnes modifications à garder dans les différentes branches. Git va alors modifier les fichiers pour lesquels il y a des conflits en précisant les endroits où il y a des conflits.

```
git merge **branch**
```

Va normalement sortir un message d'erreur de la forme

```
error: Entry '**fileName**' would be overwritten by merge. Cannot merge. (Changes in staging area)
```

Le mieux est alors de lancer la commande :

```
git status
```

Qui va lister les fichiers à l'origine du conflit entre branches. Il est alors possible d'ouvrir les fichiers concernés dans un éditeur de texte. À chaque endroit responsable de la divergence, il y aura les délimiteurs suivants :

```
<<<<< HEAD **branche sur laquelle on va faire le merge**
```

```
Contenu de la branche principale
```

```
=====
```

<sup>4</sup> Contenu de la branche distante à fusionner

```
>>>>> **nom de la branche à fusionner**
```

Il faudra donc résoudre le conflit en précisant le contenu à conserver et répéter l'opération sur tous les fichiers concernés. Il faut ensuite faire un git add des fichiers suivi d'un git commit.

💡 En cas de souci lors du merge, il est toujours possible de revenir en arrière avec la commande git merge --abort. De même, la commande git log --merge permet de lister tous les commits qui ont généré des conflits sur les différentes branches.

💡 Il est possible de favoriser certaines branches avec des options : --ours va privilégier la branche sur laquelle nous sommes tandis que --theirs fera l'inverse. Mais cela peut être source de perte d'information et est donc plus dangereux qu'une résolution de conflit plus classique.

## 12.4 Interaction avec un serveur distant

### 12.4.1 Crédation 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 12.1). 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 12.9) cliquer sur l'icône de votre compte et aller dans l'onglet « Repositories », puis cliquer sur « New ».

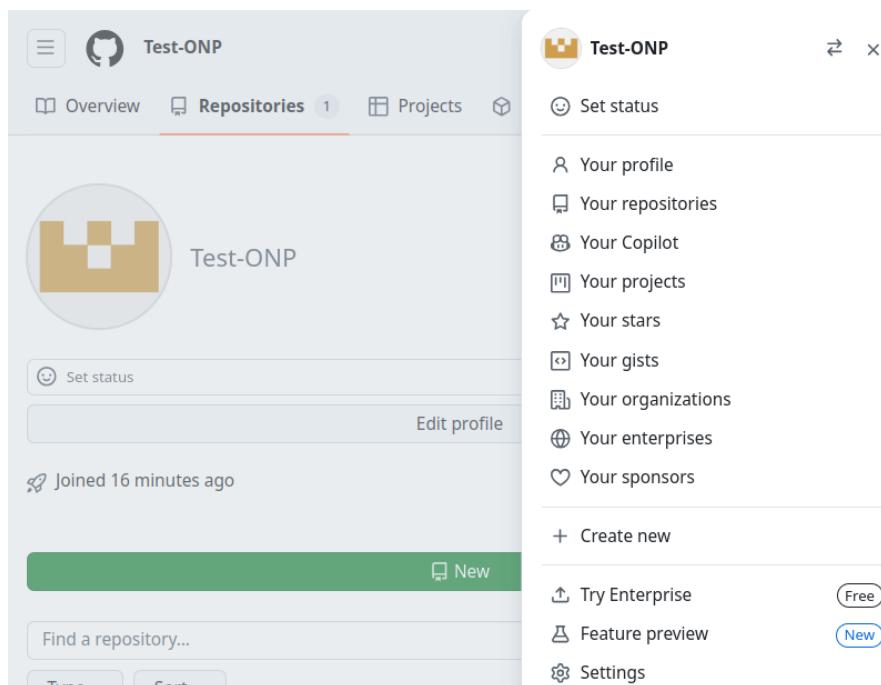


FIGURE 12.9 – Crédation d'un dépôt sur GitHub.

Il faut ensuite :

- définir un nom de dépôt,
- 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 12.9), « Settings » puis « Developer Settings », « Personal Access Tokens », « Tokens (classic) », « Generate New Token ». Y remplir les champs nécessaires, 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 12.3.3).

## 12.4.2 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.

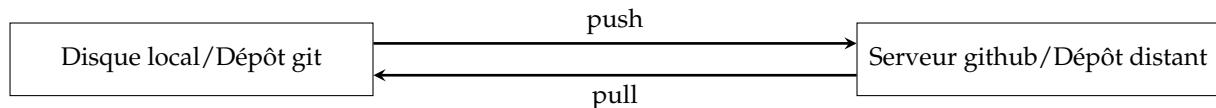


FIGURE 12.10 – Notion de pull (synchronisation local -> distant) et de push (synchronisation distant -> local).

### 12.4.2.1 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:
5 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:
10 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 12.3.6). La portée finale de cette opération est de faire en sorte de lever les conflits pour ensuite procéder à la synchronisation.

```
git pull --rebase
```

2

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).

#### 12.4.2.2 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
```

en remplaçant \*\*login\*\* par votre login et \*\*repository\*\* par les valeurs correspondantes, de même, on a supposé que la branche principale distante est nommée `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.a](#)

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

## 12.5 Ce qu'il faut retenir

Commande	Utilité
<b>Opérations de base</b>	
<code>git init</code>	Créer un dépôt (section 12.3.2)
<code>git add file</code>	Mettre les fichiers en <i>staging</i> (section 12.3.3)
<code>git rm file</code>	Supprimer les fichiers en local <b>et</b> dans le dépôt (section 12.3.3)
<code>git rm --cached file</code>	Supprimer les fichiers du dépôt <b>uniquement</b> (section 12.3.3)
<code>git mv file dest</code>	Déplacer le fichier en local et dans le dépôt
<b>Commit</b>	
<code>git ls-files</code>	Lister tous les fichiers dans le dépôt (section 12.3.4)
<code>git status</code>	Voir le statut des différents fichiers (section 12.3.4)
<code>git commit -m "MàJ"</code>	Effectuer un commit sur le dépôt (section 12.3.4)
<code>git commit -a -m "MàJ"</code>	Ajouter en staging tous les fichiers modifiés et faire un commit (section 12.3.4)
<b>Historique</b>	
<code>git log</code>	Historique des commits détaillé (section 12.3.5.3)
<code>git reflog</code>	Historique bref (section 12.3.5.3)
<code>git revert</code>	Récupération de l'état antérieur du dépôt (section 12.3.5.3.2)
<code>git reset</code>	Retour en arrière dans l'historique (section 12.3.5.3.2)
<b>Branches/Merge</b>	
<code>git branch</code>	Création d'une nouvelle branche (section 12.3.6)
<code>git merge</code>	Fusion de deux branches (section 12.3.6)
<code>git checkout</code>	Déplacement sur une autre branche
<b>Dépôt distant</b>	
<code>git push</code>	Synchroniser en envoyant les données locales vers le serveur distant (section 12.4.2)
<code>git pull</code>	Synchroniser en recevant les données du serveur distant vers le dépôt local (section 12.4.2)

TABLEAU 12.2 – Commandes de base pour Git.



# **Chapitre 13**

## **Calcul symbolique**

sage



# **Chapitre 14**

## **Jupyter**

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



# **Chapitre 15**

## **La programmation orientée objet**



# **Chapitre 16**

## **Interaction avec Excel**

openpyxl



# **Chapitre 17**

## **Deep learning**

scikit learn