



Outils numériques et programmation



Martin VÉROT

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

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

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

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

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

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

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

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

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

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

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

Table des matières

1	La philosophie du cours	9
I	Quelques notions sur le système et la ligne de commande	11
2	L'utilisation du terminal au XXI^e siècle	13
2.1	Avantages et inconvénients des interfaces graphiques	14
2.2	Avantages et inconvénients de la ligne de commande	14
2.3	Et au final, on fait quoi ?	15
3	Commandes de base	17
3.1	▲ Trouver les utilitaires en ligne de commande et LA commande maîtresse : man	17
3.2	Système de fichier et arborescence	18
3.2.1	▲ Mieux connaître l'arborescence : la commande ls	18
3.2.2	Les dossiers et fichiers particuliers	18
3.2.3	▲ Se déplacer avec la commande cd	19
3.2.4	▲ Création, modification, suppression de fichiers ou dossiers	21
3.2.5	▲ Trouver des fichiers	22
3.2.6	▲ Droits	23
3.2.6.1	Exécuter un programme/fichier	23
3.2.6.2	Connaître les droits	23
3.2.6.3	Modifier les droits	24
3.3	▲ Aller plus vite dans le terminal	24
3.4	■ Les liens	25
3.5	Redirection de la sortie	26
3.6	Quelques utilitaires	26
3.6.1	Connaître l'espace disque et la taille de certains dossiers	26
3.6.2	Les fichiers texte	27
3.6.2.1	▲ Trouver une ligne contenant une chaîne de caractère dans plusieurs fichiers : grep	27
3.6.2.2	Voir la fin d'un fichier : tail	28
3.6.2.3	Concaténer des fichiers : cat	28
3.6.2.4	Comparer des fichiers : diff et wdiff	28
3.6.2.5	■ ♦ Faire des modifications de fichiers (remplacement de chaîne de caractère, etc) : sed et awk	28
3.6.3	Les fichiers pdf	29
3.6.4	Reconnaissance de caractères : tesseract	29
3.6.5	Les images : ImageMagick	30

3.6.6	Les vidéos : ffmpeg	30
3.6.7	Compression et décompression	30
3.7	■ Enchaînement de commande : le pipe	31
3.8	■ Les alias et le fichier .bashrc	31
3.9	■ Environnement : PATH	32
3.10	■ Travail sur des machines distantes	33
3.10.1	Connexion sur des machines distantes : ssh	33
3.10.2	Au CBP : x2go	34
3.10.3	■ Copie à distance : scp	34
3.11	Gestion des processus	35
3.11.1	Fermer un processus qui buggue ou ralentit le système	35
3.11.2	Gérer l'exécution des processus dans le terminal	36
3.12	Scripts et personnalisation	37
3.13	Éditeurs de texte : IDE versus les deux indémodables vi et emacs	37
3.14	Ce qu'il faut retenir	39

II Programmation en python 41

4	Avant de se lancer	43
4.1	Décortiquer un problème avant de commencer	43
4.2	▲ Nommage des variables	44
4.3	Trouver de l'aide	44
4.3.1	Le facile : l'appel à un ami	44
4.3.2	Le moins plaisant (mais plus complet) : « RTFM »	45
4.3.3	Le pénible : consulter le code source	45
4.4	Déboguer un programme	45
4.4.1	Compartimenter/factoriser	45
4.4.2	Lire les messages d'erreur	46
4.4.3	Commenter son code (avec du texte)	46
4.4.4	Afficher ses variables et leur type	46
4.4.5	Revenir en arrière jusqu'à revenir à un programme fonctionnel	46
4.4.6	Faire un appel à l'aide	46
4.5	Commenter son code	47
4.6	Structure d'un script python	48
4.7	Ce qu'il faut retenir	48
5	Les bases	49
5.1	Types de base	49
5.1.1	Nombres	49
5.1.1.1	◆ Représentation flottante, comparaison et précision	49
5.1.2	Chaînes de caractère	50
5.1.3	Listes	51
5.1.4	Tuples	51
5.1.5	Dictionnaires	51
5.1.6	Fonctions	52
5.2	◆ Objets mutables et immutables	53
5.2.1	◆ Première conséquence : copie d'objets mutables/imbriqués : « shallow copy » versus « deep copy »	54

5.2.2	◆ Deuxième conséquence : éléments mutables et fonctions	56
5.2.2.1	◆ Modification d'un élément mutable au sein d'une fonction	56
5.2.2.2	◆ Élément mutable comme argument optionnel	56
5.3	Espaces de nommage	56
5.3.1	◆ les différents espaces de nommage	56
5.3.2	■ Portée des variables	57
5.4	Structures de bases : boucles et conditions	57
5.4.1	Boucles	58
5.4.1.1	Boucles for	58
5.4.1.2	Boucles while	58
5.4.2	Conditions if, elif, else	58
5.4.2.1	Instructions break, pass et continue	59
5.5	Gestion des fichiers	59
5.6	Évaluer la performance de son code	60
5.7	Ce qu'il faut retenir	60
6	Modules, librairies, packages et frameworks	61
6.1	Un peu de vocabulaire	61
6.2	Au fait, une librairie ça sert à quoi ?	61
6.3	Import d'une librairie ou d'un module	62
6.3.1	Import spécifique	63
6.3.2	Import « à la sauvage »	63
6.4	Quelques librairies courantes	64
7	La manipulation de tableaux avec Numpy	65
7.1	Broadcasting, slicing, axes	65
7.2	Fonctions analytiques, numériques	65
7.2.1	Dérivation	65
8	Faire des graphiques avec Matplotlib	67
8.1	Principes généraux	67
8.2	Graphiques unique	67
8.3	Graphiques multiples	67
8.4	Graphiques à trois dimensions	67
8.5	Graphiques animés	67
9	Quelques problèmes numériques courants	69
9.1	Recherche de zéros	69
9.2	Ajustement de courbe	69
9.3	Intégration	69
9.4	Équations différentielles	69
9.5	Transformée de Fourier	69
9.6	Arrangement, combinaison	69
III	Pour aller plus loin	71
10	Les expressions régulières	73
11	La programmation orientée objet	75

12 Interaction avec Excel	77
13 Tableaux non numériques	79
14 Calcul symbolique	81
15 Deep learning	83
16 Les gestionnaires de version : git/github	85

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
3. des *exercices plus complets* qui correspondent à des tâches complexes qui demandent de combiner des idées, fonctions, portions de code.

Les deux premiers types de ressources constitueront la partie de travail individuel à fournir. Cela sera évalué avec des questionnaires réguliers pour voir si vous avez compris les concepts centraux. Les exercices complets seront pour leur part ce sur quoi nous nous concentrerons lors des séances. Pour que ces heures soient productives, il faudra absolument avoir travaillé sur les ressources précédentes.

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

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

Première partie

Quelques notions sur le système et la ligne de commande

Chapitre 2

L'utilisation du terminal au XXI^e 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.

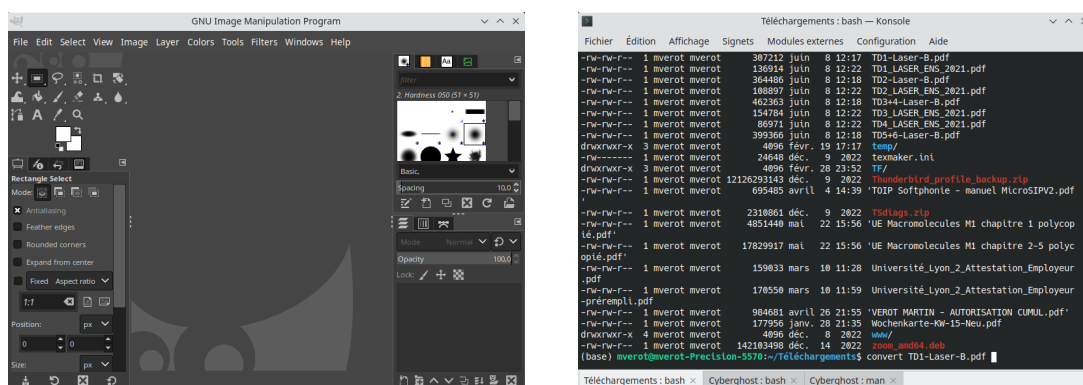


FIGURE 2.1 – Logiciel à interface graphique à gauche, console avec invite de commande à droite.

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 économe 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).

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS				2 MONTHS	2 WEEKS	1 DAY
	1 DAY					8 WEEKS	5 DAYS

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 lettres « 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  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 au à l'endroit « / »

Ensuite, chaque nouveau dossier correspond à une nouvelle branche ou ramification 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 important 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. 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 décompresse 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 éléments
run	20/07/2023 16:53	50 éléments
bin	20/07/2023 14:05	3 322 éléments
var	04/01/2023 18:46	14 éléments
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 éléments
Téléchargements	20/07/2023 17:13	145 éléments
Images	20/07/2023 15:19	104 éléments
Bureau	20/07/2023 11:41	3 éléments
Dropbox	19/07/2023 23:27	95 éléments
Cours	20/07/2023 17:24	23 éléments
python	22/06/2023 11:22	82 éléments
Cours-python	20/07/2023 17:58	12 éléments
images	20/07/2023 17:53	14 éléments
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 ▲ Création, 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^e 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.^a 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.^b
3. N’utilisez pas d’espace dans vos noms de fichiers, parce que c’est vraiment source de problème.^c
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, ect). 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 attendrez 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.^d

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 », voire 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'espace 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 ligne de caractère 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
<code>pdftotxt</code>	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 <code>pdf2txt</code> est similaire
<code>pdfseparate</code>	sépare les pages d'un pdf <code>pdfseparate cours_python.pdf %d-cours-python.pdf</code>
<code>pdfunite</code>	réunit plusieurs pdf
<code>pdfimages</code>	extraît les images d'un pdf
<code>pdfjam</code>	pour mettre deux pages par feuille, et faire bien d'autres choses <code>pdfjam --nup 2x1 --landscape -o out-nup.pdf fichier-input.pdf</code>
<code>pdfbook2</code>	pour avoir une impression en format livret (pour une reliure au centre)
<code>gs</code>	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, redimensionne, 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. Pour cela, il est possible de créer des alias qui sont de nouvelles commandes personnalisées. Pour cela, 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 un 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 rendre moins vulnérable aux attaques les machines qu'elle protège.

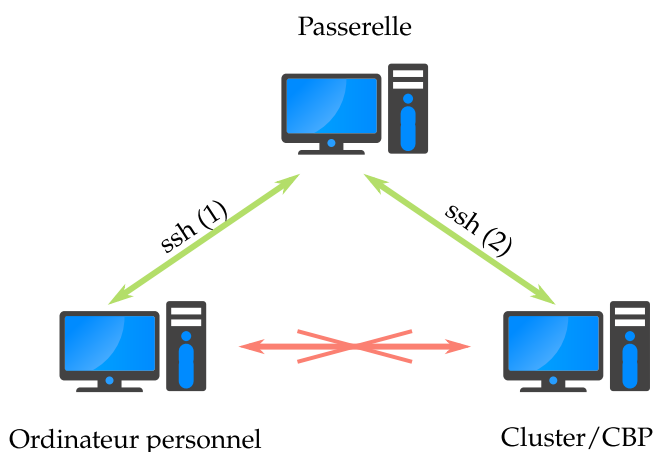


FIGURE 3.2 – Schéma traditionnel d'une connexion `ssh` via une passerelle pour accéder à un cluster. Il n'est pas possible de se connecter directement en `ssh` au cluster et il est nécessaire de passer par une passerelle.

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 est le même que le 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 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, il existe plusieurs outils accessibles en étant membre de l'ENS :

- un serveur de transfert de fichier interne à l'établissement : <https://filesender.ens-lyon.fr>
- un autre plus générique pour les membres du réseau RENATER dont l'ENS fait partie <https://filesender.renater.fr/>.

Dans les deux cas, les fichiers 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 complétion 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 veut 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
```

3.11 Gestion des processus

3.11.1 Fermer un processus qui bugue 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 en général 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)

commande `nohup`

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 un 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 logiciels 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 fonctionnements 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> <code>ls</code>	Emplacement dans l'arborescence pour donner le répertoire courant 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> <code>cp</code> <code>rm</code> <code>mkdir</code>	déplacer ou renommer un fichier ou des répertoires copier un fichier ou des répertoires supprimer un fichier ou des répertoires 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> <code>tail</code> <code>sed</code>	trouver les lignes contenant une chaîne de caractère précise dans un fichier afficher la fin d'un fichier faire des recherches/remplacer dans un fichier
<code>zip</code> / <code>unzip</code> <code>tar</code>	compression et décompression au format zip compression et décompression au format tar.gz « -xvfz » pour l'extraction, « -cvzf » pour la création
<code>df -h</code> <code>du -h</code>	connaître l'espace disque connaître la taille d'un dossier
<code>ssh</code> <code>exit</code> <code>scp</code>	connexion à une machine distante fermeture d'une connexion ssh transfert de fichier d'un ordinateur à un autre
<code>top</code> / <code>kill</code> & <code>Ctrl</code> + <code>C</code>	lister les processus en cours d'exécution/tuer un processus pour rendre le processus non bloquant 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 répondre 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 lecteur ;
- 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 solutions 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 bibliothèques ou fonctions. Cela nécessite par contre d'avoir une idée de la fonction ou bibliothèque 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 approprier 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'expliquer 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.^a

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 votre. 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'indices, 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.

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 simple.
- 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 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 fraction 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 moment. 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 bibliothèques `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.^a

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>"{:<10}".format('blabla')</code>	blabla_____	justifié à gauche sur (au moins) 10 caractères
<code>"{:>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</code> , <code>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 ??), 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.

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.

Les fonctions lambda À rédiger

Unpacking À rédiger

splat lien avec args et kwargs

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 peut 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).

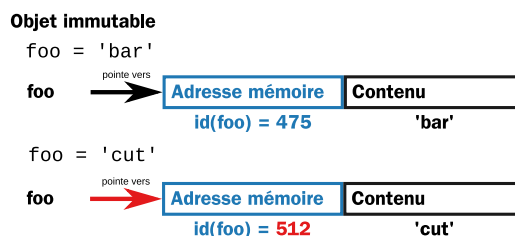


FIGURE 5.2 – Pour un objet immutable, en changer le contenu correspond en fait à 1) créer un nouvel objet en mémoire avec le nouveau contenu 2) mettre à jour la référence de la variable pour pointer vers l'objet que l'on vient de créer.

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 immutable c'est directement le lien avec la variable, pour un objet mutable, il s'agit d'une référence interne.

Objet mutable

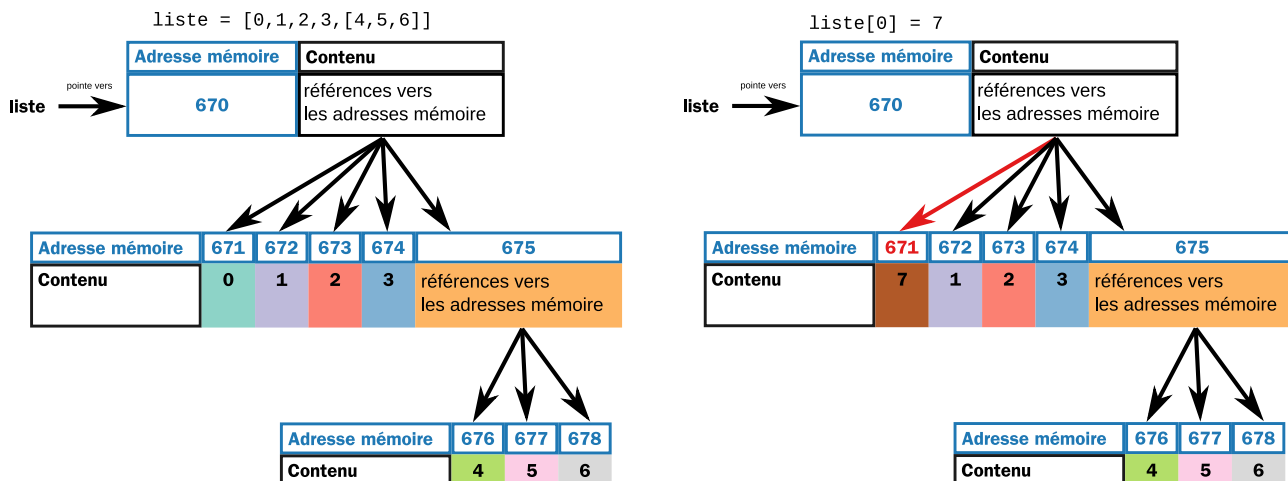


FIGURE 5.3 – Pour un objet mutable, en changer le contenu correspond en fait à 1) créer un nouvel objet en mémoire avec le nouveau contenu 2) mettre à jour la référence en **interne**.

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
int, float, complex	list, dict
string	ndarray (numpy)
bool	Dataframe (pandas)
frozenset	set
bytes	bytearray
tuples*	

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

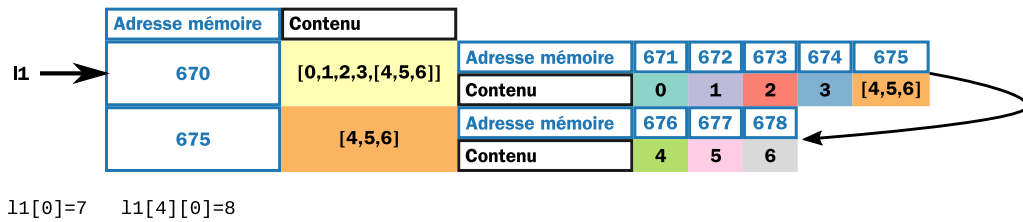
5.2.1 Première conséquence : copie d'objets mutables/imbriqués : « *shallow copy* » versus « *deep copy* »

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.



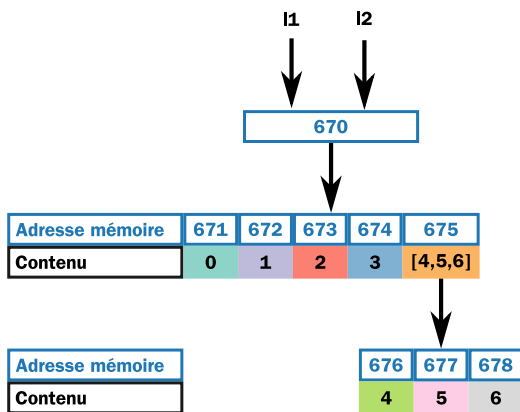
Sur l'exemple 5.4, la liste 1 contient une suite d'adresses (671 à 675) et chacune de ces adresses contient la valeur (soit les entiers 0,1,2,3, soit une liste d'entier [4,5,6]. Lors de la modification d'un élément de liste, on change le contenu de l'adresse mémoire en interne pour pointer vers un nouvel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



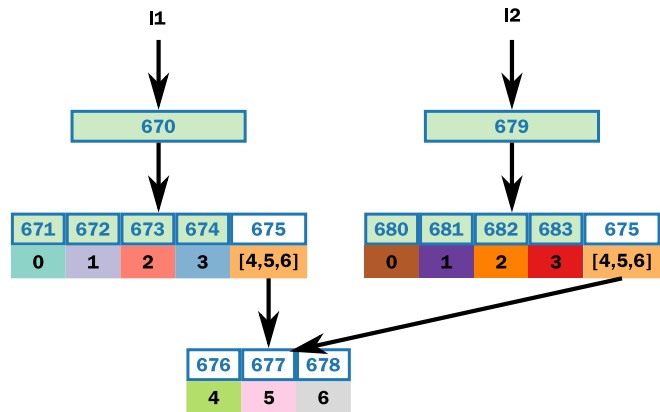
Changer un élément de liste revient à changer une case colorée sans changer l'adresse mémoire vers laquelle pointe la liste.

Shallow copy : `l2 = l1`



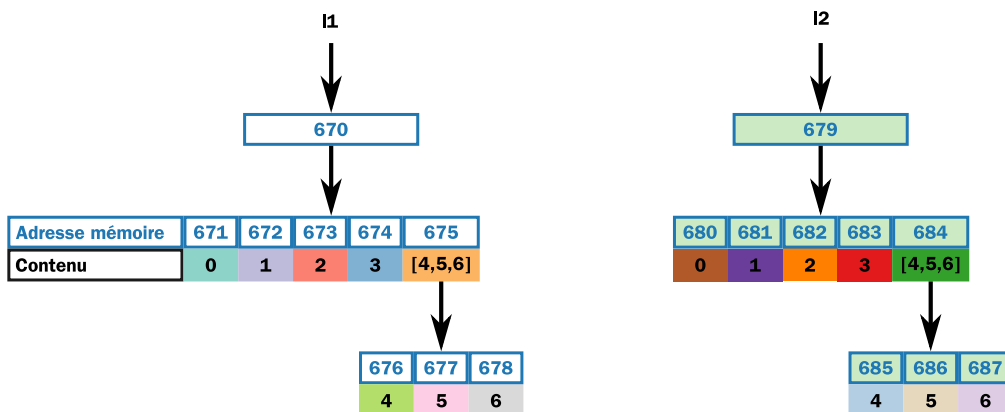
Modifier une liste modifie l'autre car on utilise dans les deux cas les mêmes adresses mémoire

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 = 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.^b

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 crée un nouvel espace de nommage lors de l'appel de la fonction et le maintient 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 ??).^c 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.

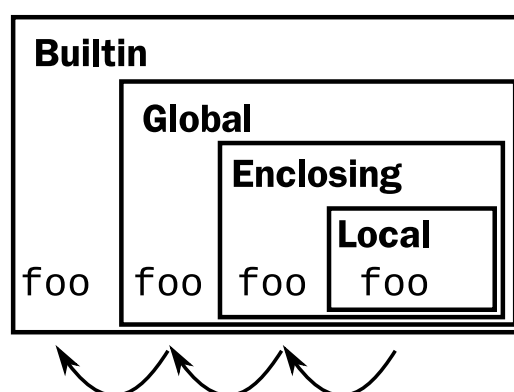


FIGURE 5.5 – Espaces de nommage et portée pour les différentes variables

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

💡 Pour rappel (section 5.2.2.1), les objet mutables peuvent être modifiés au sein d'une fonction, cela veut donc dire que si une variable n'existait 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 spécificité liées au langage.

c. Attention, dans d'autres langages de programmation, les espaces de nommages 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 indix in range(6):
    _print(indix)
```

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, `isinstance` 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` ^①. Le premier argument est le nom du fichier, le deuxième le type d'opérations qui pourront être effectuées avec le fichier (tableau 5.4).^d


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, 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.

5.6 Évaluer la performance de son code

complexité,timeit

5.7 Ce qu'il faut retenir

- Les types de bases utilisables sous python (`str`, `int`, `float`, `list`, `tuple`, `dict`^e)
- 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,`
 - `for index,element in enumerate(liste)`
 - `for key,value in dict.items()`
- Savoir ouvrir et fermer un fichier, le lire ou y écrire.

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

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 différents 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 milliers 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 être 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'utiliserai 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 uns 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 ^a

 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. ^b

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 coûte 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_librairie
```

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

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

7.1 Broadcasting, slicing, axes

masking

7.2 Fonctions analytiques, numériques

7.2.1 Dérivation

Chapitre 8

Faire des graphiques avec Matplotlib

8.1 Principes généraux

Toutes les moyennes Tufte, colorbrewer

8.2 Graphiques unique

plot, scatter

8.3 Graphiques multiples

grid, set_

8.4 Graphiques à trois dimensions

line,

8.5 Graphiques animés

Chapitre 9

Quelques problèmes numériques courants

9.1 Recherche de zéros

9.2 Ajustement de courbe

9.3 Intégration

9.4 Équations différentielles

9.5 Transformée de Fourier

9.6 Arrangement, combinaison

itertools

Troisième partie
Pour aller plus loin

Chapitre 10

Les expressions régulières

re

Chapitre 11

La programmation orientée objet

Chapitre 12

Interaction avec Excel

openpyxl

Chapitre 13

Tableaux non numériques

pandas

Chapitre 14

Calcul symbolique

sage

Chapitre 15

Deep learning

scikit learn

Chapitre 16

Les gestionnaires de version : git/github