

Passeport Informatique Télécoms

2023

À lire attentivement avant de débuter

Ce module est composé d'une série d'activités, à **conduire en autonomie**, visant à développer un socle commun de compétences en informatique pour l'ensemble des étudiants de 3e année en Télécoms. Par autonomie, nous entendons que chaque étudiant effectue des recherches sur les moteurs de recherche ou via la commande `linux man`. **Ne posez aucune question sans avoir au préalable effectué des recherches en autonomie.**

Compétences ciblées

- C2.3 Spécifier, concevoir et modéliser des algorithmes et des programmes informatiques
 - Connaître les opérations basiques (déplacement, écriture, enregistrement) sous `vim`
 - Connaître les opérations standard (copier, coller, annuler) sous `vim`
 - Connaître les opérations avancées (split, fusion, recherche) sous `vim`
 - Connaître l'interpréteur de commandes `Python3`
 - Connaître les concepts basiques (listes, tuples, dictionnaires, structures conditionnelles, boucles) sous `Python3`
 - Connaître les concepts standard (modularité, exceptions, classes, manipulation de fichiers) sous `Python3`
 - Être capable d'exécuter un script `Bash`
 - Être capable d'exécuter des instructions simples dans l'interpréteur de commandes `Python3`
- C2.7 Mettre en oeuvre, réaliser, développer, déployer des programmes informatiques
 - Être capable d'afficher et manipuler des variables sous `Bash` et `Python3`
 - Être capable de naviguer dans un système de fichiers via les commandes `cd`, `ls`
 - Être capable de manipuler des fichiers d'un système de fichiers via les commandes `mkdir`, `ln`, `more`, `head`, `tail`
 - Être capable d'archiver des répertoires au sein d'un système de fichiers via les commandes `tar`, `zip`, `unzip`
 - Être capable de traduire un énoncé simple en un programme `Bash` ou `Python3` fonctionnel
 - Être capable de structurer la compilation d'un programme écrit dans un langage compilé, par exemple C.
- C3.2 Travailler, apprendre, évoluer de manière autonome
 - Être capable de réaliser un tutoriel en autonomie
 - Être capable de réaliser un mini projet en autonomie
 - Être capable de conduire une recherche sur un sujet technique en autonomie

Activités

Chacune des activités proposées se décompose comme suit :

- Un tutoriel : selon son niveau et son rythme chaque étudiant devra réaliser au moins un tutoriel associé à l'activité.
- Un ensemble d'exercices d'entraînement : au choix les étudiants peuvent soit effectuer le tutoriel puis l'ensemble des exercices ou les faire simultanément.
- Un quiz de validation des compétences pour l'activité : un QCM chronométré devra être réalisé par chaque étudiant afin d'évaluer le niveau de compétences pour l'activité.
- Au terme de ces activités, un projet individuel, mobilisant toutes les compétences acquises lors des activités précédentes, devra être réalisé.

Auto-évaluation

Bien qu'il est fortement conseillé de suivre les tutoriels et d'effectuer les exercices d'entraînement, il n'est pas requis que vous exploriez exhaustivement toutes les notions développées dans un tutoriel ni que vous effectuez tous les exercices proposés. Les quiz ne peuvent-être réalisés qu'une seule fois. Dès le démarrage vous n'aurez qu'une pognée de minutes pour les compléter. Il n'y a pas d'évaluation dans cet EC. Vous aurez à vous auto-évaluer suivant vos performances durant les quiz notamment.

Tri des étudiants

Nous souhaitons donner plus d'autonomie à ceux d'entre vous qui en ont besoin. Pour cela, deux tris auront lieu, permettant ainsi de sélectionner les étudiants sur lesquels notre attention se portera, et ceux qui pourront continuer en toute autonomie les activités de PIT.

1. **Lundi 24/10 de 17h :00mn à 17h :15mn** — Tout étudiant **des groupes G2 et G3** validant le [quiz](#) du premier tri, continuera en totale autonomie PIT. Le quiz est validé lorsque l'étudiant a obtenu dans la limite du temps imparti au moins 75% de bonnes réponses aux questions posées. Tous les autres étudiants seront appelés à continuer les activités sous l'encadrement des enseignants.
2. **Mardi 24/10 de 17h :00mn à 17h :15mn** — Tout étudiant **du groupe G1** validant le [quiz](#) du premier tri, continuera en totale autonomie PIT. Le quiz est validé lorsque l'étudiant a obtenu dans la

limite du temps imparti au moins 75% de bonnes réponses aux questions posées. Tous les autres étudiants seront appelés à continuer les activités sous l'encadrement des enseignants.

3. **Jeudi 27/10 de 11h :00mn à 11h :15mn** — Tout étudiant validant le [quiz](#) du deuxième tri, continuera en totale autonomie PIT. Le quiz est validé lorsque l'étudiant a obtenu dans la limite du temps imparti au moins 75% de bonnes réponses aux questions posées. Tous les autres étudiants seront appelés à continuer les activités sous l'encadrement des enseignants.
4. Ensuite, selon leur avancement dans les activités PIT, les étudiants pourront être excusés par la suite.

Encadrement pédagogique

Les encadrants ne seront pas toujours présents dans votre salle, ils sont à cheval sur plusieurs salles. Vous avez la possibilité de les contacter par email si vous avez besoin d'aide.

- jilles-steeve.dibangoye@insa-lyon.fr
- loiseau@creatidis.insa-lyon.fr

Table des matières

1 Faire connaissance	7
1.1 Informatique	7
1.2 Ordinateur	7
1.3 Système d'exploitation	8
1.4 Codage des informations	8
1.5 Fichier et système de fichiers	9
1.6 Interpréteur de commandes	9
1.7 Editeur de texte	10
1.7.1 Navigation générale	10
1.7.2 Manipulation de lignes	10
1.7.3 Marques	11
1.7.4 Commande = (indentation)	11
1.7.5 Commande . (point) (répétition)	11
1.7.6 Recherche/Remplacement d'une chaîne de caractères .	11
1.7.7 Manipulation de fichiers	12
1.7.8 Auto-évaluation	12
2 Passer aux commandes	13
2.1 Commandes	13
2.2 Fonctionnalités supplémentaires	13
2.3 Travail à distance	14
2.4 Programmer en shell	14
2.5 Exercices	14
3 Structurer la compilation	19
3.1 Pourquoi compiler ?	19
3.2 Une seule commande	20
3.3 Une commande par fichier source	20
3.4 Avec Make	20

3.5	Exercices	22
4	Programmation python	25
4.1	Faites vos premiers pas en Python	25
4.1.1	Compétences ciblées	25
4.1.2	Déclarer et initialiser une variable	26
4.1.3	Déclarer des procédures munies de paramètres	26
4.1.4	Définir un programme principal avec <code>sys.argv</code>	27
4.1.5	Contrôler le flux d'instructions	27
4.2	Manipulez des objets en Python	28
4.2.1	Compétences ciblées	28
4.2.2	Écrire et utiliser une fonction	29
4.2.3	Créer et éditer des objets	29
4.3	Créer des objets en Python	31
4.3.1	Compétences ciblées	31
4.3.2	Créez des modules	31
4.3.3	Manipulez un fichier	32
4.3.4	Créez et utilisez une classe	32
5	Git, Github	35
5.1	Git, Github	35
5.2	Exercices	35

Chapitre 1

Faire connaissance

Durée estimée : 2 à 4h

1.1 Informatique

La Société Informatique de France a proposé la définition suivante de l'informatique :

Definition 1. *L'informatique est la science et la technique de la représentation de l'information d'origine artificielle ou naturelle, ainsi que des processus algorithmiques de collecte, stockage, analyse, transformation, communication et exploitation de cette information, exprimés dans des langages formels ou des langues naturelles et effectués par des machines ou des êtres humains, seuls ou collectivement.*

L'informatique n'est donc pas juste la science des ordinateurs, c'est une science et une technique qui peut s'appliquer à des supports matériels très différents, dont l'[ordinateur](#), mais pas seulement. Comme entrée en matière, vous êtes invité à lire ce [billet](#) ainsi que les textes qui lui sont liés.

1.2 Ordinateur

L'ordinateur reste néanmoins une machine de prédilection en informatique. Vous allez certainement utiliser une telle machine pendant des heures, des jours, voire des années. Cela mérite bien de savoir, au moins sommairement, de quoi est fait un ordinateur. Après avoir visionné cette vidéo [sur ce qu'à un ordinateur dans le ventre](#) et vous être renseigné, éventuellement, sur

les termes employés, vous devriez savoir ce que sont des périphériques (interne/externe, entrée/sortie), des contrôleurs, des ports, des disques dur, des barrettes de RAM, une carte mère, un micro-processeur.

1.3 Système d'exploitation

Le système d'exploitation se situe à l'interface entre deux mondes : le logiciel et le matériel, composé essentiellement de mémoire, de processeur(s), de périphériques. Le rôle du système d'exploitation est d'assurer que ces éléments matériels, requis par les logiciels en cours d'exécution, soient utilisés de manière partagée, équilibrée et sûre. Cette vidéo explique le [rôle du système d'exploitation](#), notamment dans la gestion virtualisée de la mémoire, le contrôle de l'exécution via l'ordonnanceur et la médiation par les appels systèmes.

1.4 Codage des informations

Vous savez certainement que toutes les informations que l'ordinateur manipule sont codées en binaire, c'est-à-dire en suites de 0 et 1 :

1. les [entiers naturels et relatifs](#) bien sûr,
2. les nombres décimaux dits à [virgule flottante](#),
3. les caractères ([ASCII](#), [unicode](#)),
4. les données multimédias (son, image, vidéo, page web...),
5. ainsi que les instructions et programmes...

Lire ce billet introduisant le [codage binaire](#) est un minimum. Mais pourquoi le binaire ? C'est d'abord un compromis entre la quantité de symboles et la quantité de chiffres nécessaires pour représenter un nombre possible. En décimal, il nous faut 4 chiffres pour représenter 1024, c'est pas mal. Mais nous avons dix symboles possibles 0, 1, ... 9 pour chaque chiffre. C'est beaucoup et difficile à matérialiser avec une bonne résistance au bruit. En unaire, on n'a au contraire qu'un seul symbole. C'est facile à matérialiser (une pierre, une diode allumée, etc.), mais il nous faut 1024 chiffres pour représenter 1024 ! En binaire, on a deux symboles, généralement notées 0 et 1. C'est encore facile à matérialiser (tension inférieure ou supérieure à un seuil, courant électrique qui passe ou ne passe pas, etc.) et il ne faut pas que $\log_2(1024) = 10$ chiffres pour représenter 1024, ce qui est tout à fait raisonnable compte-tenu de la petitesse et la complexité des circuits électroniques actuellement construits.

1.5 Fichier et système de fichiers

Le terme [système de fichiers](#) désigne à la fois l'organisation des informations mémorisées sur les périphériques de stockage de l'ordinateur et la vue logique hiérarchique présentée à l'utilisateur. Les informations sont stockées dans des paquets appelés fichiers, écrits dans un certain format, c'est-à-dire selon une certaine manière d'encoder les informations en binaire. Ces fichiers peuvent être aussi bien du texte, une page web, un morceau de musique, une photo de vacance, un script, un logiciel, etc. Un répertoire regroupe des fichiers ou d'autres répertoires, ce qui aboutit à une hiérarchie de fichiers. C'est le [chemin d'accès](#) qui permet de localiser un fichier de la hiérarchie.

Faire 1. *Vous maîtrisez ce qu'est un fichier, un répertoire (= dossier), un chemin d'accès, un lien symbolique, le répertoire courant ? Si oui, passez à la suite. Si non, jouez à [Find your path](#) pour vous familiariser avec la représentation des chemins dans les environnements de type unix. Si vous avez joué déjà 20 minutes et que vous ne voyez pas le rapport avec ce qui précède, contactez l'enseignant.*

1.6 Interpréteur de commandes

La [console](#) est une interface textuelle qui permet à un utilisateur de demander à l'ordinateur de réaliser certaines tâches, uniquement à l'aide d'un écran et d'un clavier. Sur un serveur sans interface graphique la console est généralement directement accessible au démarrage. Sur une machine grand public, on utilise un [émulateur de terminal](#), c'est-à-dire un programme qui émule une console dans une interface graphique. L'utilisateur n'a qu'à écrire par le clavier la commande qu'il souhaite que le système d'exploitation exécute. Quand une ligne de commande est écrite, elle est passée à l'interpréteur de lignes de commande (=shell), qui la décortique et se charge de la faire exécuter en interaction avec le système d'exploitation. Il existe de nombreux interpréteurs de lignes de commandes, qui fonctionnent tous plus ou moins de la même façon. Nous considérerons que ce sera Bourne-Again Shell (bash) dans le reste du document. C'est le shell associé par défaut à un compte d'utilisateur dans Ubuntu. Si vous utilisez un autre système d'exploitation, voyez comment utiliser un émulateur de terminal, ainsi que bash, ou vous adapter à un shell similaire. Cette page de l'[astus](#) peut vous aider.

1.7 Editeur de texte

Vous allez bientôt écrire vous-mêmes des programmes. Ces programmes ne seront d'abord rien d'autre que du texte sauvegardé dans un fichier. Pour écrire du texte, vous avez besoin avant tout d'un éditeur de texte, un logiciel permettant d'écrire et modifier un texte. De même que vous êtes libres d'utiliser le système d'exploitation qui vous convient, vous êtes libres d'utiliser l'éditeur de texte qui vous convient (`emacs`, `atom`, etc), sous réserve que vous sachiez les utiliser bien sûr... Néanmoins, nous vous demandons de savoir utiliser, à un niveau débutant au moins, un éditeur particulier : `vi`. Pourquoi lui ? Il fait parti probablement de tous les systèmes d'exploitation de type unix, car il est léger, rapide, économique en ressource, puissant en fonctionnalité. Vous devrez peut-être un jour vous connecter à un serveur unix en mode console et modifier un fichier de configuration. Vous serez alors content de savoir ouvrir le fichier avec `vi`, passer en mode insertion et sauvegarder vos modifications. Vous vous remercierez d'avoir regarder attentivement les liens suivants :

- [tutoriel](#),
- [FAQ](#),
- [cheatsheet](#) (vous en trouverez pleins d'autres sur le web),
- [vi ou vim](#)? . Note : dans Ubuntu, comme dans d'autres systèmes, vim est le seul éditeur de type vi installé par défaut, et donc, vi démarre vim par défaut. Ce n'est pas un problème car tout ce qui est dans vi est disponible dans vim.

1.7.1 Navigation générale

- Créez un répertoire `lab7` dans lequel vous déposerez vos fichiers.
- Copiez le fichier `yenta.c` disponible sur la page PIT de Moodle dans ce répertoire
- Combien de lignes possède le fichier `yenta.c` ? **`wc -l yenta.c`**

1.7.2 Manipulation de lignes (**vim yenta.c**)

- Supprimer/couper des lignes
- Copier des lignes
- Couper/Supprimer les #mots à partir du curseur courant
- Couper/Supprimer les #caractères à partir du curseur courant
- Indiquez comment copier les 3 premiers mots de la ligne courante en 2 commandes. La première se positionne en début de ligne, la suivante

copie les 3 premiers mots.

1.7.3 Marques

- Indiquez la suite de commande permettant de se déplacer, en utilisant les marques, la fonction `static int yenta_get_socket(pci_socket t *socket, socket state t *state)` à la fin du fichier.
 - Joindre l'ensemble des lignes de cette dernière fonction.
- J -> joint 2 lignes ; %J -> avec espace ; gJ -> avec tabulation
10J -> 10 lignes avec espace**

on positionne la marque sous le curseur avec « m_ » où _ correspond à une lettre (donc 26 marques possible) et on y accède avec « '_ »

1.7.4 Commande = (indentation)

La commande = permet de ré-indenter automatiquement. Elle prend en paramètre une portée (présentée en détail plus loin dans le chapitre). Pour ré-indenter entièrement un fichier, taper "gg=G" :

- 'gg' place le curseur au début du fichier
- = demande la ré-indentation
- G demande d'exécuter l'action jusqu'à gg=G la fin du fichier
- Abîmez un peu l'indentation de `yenta.c` puis ré-indentez le automatiquement.

1.7.5 Commande . (point) (répétition)

- Allez au début du fichier et répétez votre dernière commande de jointure.
- Ajoutez un commentaire à la fin du fichier. **Ctrl + v**

1.7.6 Recherche/Remplacement d'une chaîne de caractères

- À quelle ligne se trouve la fin de la fonction `yenta_get_status` ? **Ligne 235**
 - Positionnez vous à la ligne 219 (2 1 9 G). En deux commandes, à quelle ligne se trouve la fin de la fonction `yenta_set_power` ? **230G**
 - Dans le fichier `yenta.c` remplacez toutes les tabulations par 2 espaces. Donnez la commande de substitution utilisée
 - Quelle est la commande de substitution équivalente à la commande J?
 - Replacez les accolades à la suite des fonctions et non pas à la ligne suivante.
- « /yenta_get_status » permet de placer le curseur sur ce mot et « :set nu » permet d'afficher les lignes à gauche (:set nu! pour les cacher)
- « :s/terme_recherche/terme_replacement/g » -> sur toute la ligne
 « :%s/terme_recherche/terme_replacement/g » -> sur tout le fichier
 « .,\$/terme_recherche/terme_replacement/g » -> de la ligne actuelle jusqu'à la fin du fichier

1.7.7 Manipulation de fichiers

Les commandes pour manipuler les fichiers sont :

- Pour sauver un fichier

1.7.8 Auto-évaluation

Faire 2. Faites le *quizz* de validation sur Moodle.

Bravo, vous êtes maintenant ceinture jaune de vi. Vous voulez devenir ceinture noire ? Utilisez-le pour écrire tous vos programmes.

Chapitre 2

Passer aux commandes

Durée estimée : 4h

2.1 Commandes

Definition 2. *Une commande est une instruction qu'un utilisateur envoie au système d'exploitation de l'ordinateur pour lui faire exécuter un programme.*

Un processus est une instance d'un programme en train de s'exécuter, autrement dit une tâche en cours. A chaque commande donnée, le programme correspondant est exécuté, un nouveau processus est créé. Il peut s'agir de supprimer des fichiers, d'accéder à des répertoires, de modifier des droits d'accès, etc. Il existe un grand nombre de commandes et les actions précises de chacune d'elles sont de plus conditionnées par un ensemble plus ou moins grand de paramètres.

1. [commandes basiques](#),
2. [commandes à connaître](#) (uniquement les sections 1, 2 et 3 à ce stade),

2.2 Fonctionnalités supplémentaires

1. [redirections et enchainements](#) (sections 2, 3 et 5),
2. [variables d'environnement](#),
3. gérer les [processus](#).

[Aide mémoire](#) sur l'utilisation des consoles bash et quelques commandes sous Unix.

2.3 Travail à distance

Enfin, en ces temps où le travail à distance prend de l'ampleur, nous vous demandons de connaître le protocole de communication **SSH** et de maîtriser les commandes ssh (section 2.1, 2.3) et scp (section 2.4). La documentation de l'astus peut vous aider à **vous connecter à des machines de l'INSA depuis chez vous**. D'abord, se connecter au **VPN**, puis lancer une session **SSH**.

2.4 Programmer en shell

Definition 3. *Un script shell est un programme informatique développé pour fonctionner dans un shell Unix, il permet d'automatiser une série d'opérations qui seront exécutées de manière séquentielle en utilisant les commandes que nous avons déjà vues, mais aussi des structures de contrôle (constructions conditionnelles et boucles).*

Les scripts shell peuvent aller de scripts simples à de véritables usines à gaz. Commencez donc par :

1. écrire un **script HelloWorld!** en shell
2. gérer les **entrées / sorties**
3. passer aux **structures de contrôle**
4. définir des **fonctions** en shell

2.5 Exercices

Faire 3. *Evaluez vos compétences sur les questions suivantes.*

1. *Quoi servent les commandes shell suivantes ?*

- **pwd** **chemin emprunté du répertoire racine au répertoire actuel**
- **ls** **liste les docs/dossiers du répertoire actuel**
- **ls *** **liste les docs/dossiers du répertoire actuel et leurs contenus**
- **ls -l /usr/bin** **liste les docs/dossiers du répertoire « bin » avec les détails**
- **ls /home** **liste les docs/dossiers du répertoire « home »**
- **ls ..** **liste les docs/dossiers du répertoire précédent le répertoire précédent**
- **ls /** **liste les docs/dossiers du répertoire racine**

2. *Flux (entrées / sorties)*

- Que fait la commande suivante : ‘> fictoto’ ?
- Que fait la commande **echo ‘yo’ > plop** exécutée deux fois ?
- Que fait la commande **echo ‘yo’ » plop** exécutée deux fois ?

Crée un fichier nommé « **fictoto** » dans le répertoire racine / redirige un fichier **fictoto** en sortie standard

2x(echo yo > plop) : crée un fichier nommé **plop** dans le répertoire racine contenant le texte « **yo** »

2x(echo yo >> plop) : crée un fichier nommé **plop** dans le répertoire racine contenant le texte « **yo /*ENTER*/ yo** »

Identifiant X pour :

- entrée standard (stdin) : X = 0
- sortie standard (stdout) : X = 1
- sortie d'erreur (stderr) : X = 2

- Que fait la commande ‘ls -l /etc | more’
- À quoi sert la redirection avec ‘2>’ ? **echo yo 2> plop : crée un fichier plop dans le répertoire racine ne contenant aucun texte car c'est stderr**
- 3. Variables d'environnement
 - Définir une variable d'environnement *TOTO*
 - > *export TOTO='toto'*
 - Définir une variable d'environnement *MYSELF* comme suit :
 - (a) *MYSELF='whoami'*
 - (b) *MYSELF=(whoami)*
 - Comment lister toutes les variables d'environnement ?
- 4. Variable d'environnement *\$PATH*
 - Que fait cette variable ?
 - Faire un script shell ‘~/Temp/yo.sh’ qui affine ‘Yo!’ sur la sortie standard.
 - Comment invoquer ‘yo.sh’ depuis :
 - (a) ‘~/Temp’
 - (b) ‘~’
 - (c) Etendre la variable *\$PATH* pour que ‘yo.sh’ soit disponible depuis n'importe quel répertoire
 - (d) Fermer le SHELL courant, puis en relancer un nouveau. La commande ‘yo.sh’ est-elle encore disponible ? Comment y remédier ?
- 5. Archives
 - Que fait :
 - (a) *echo 'Yo!' | bzip2 -?*
 - (b) *echo 'Yo!' | bzip2 > yo.bz2* ?
 - (c) *bunzip2 yo.bz2* ?
 - (d) *bzip2 fichier*
 - Même principe pour *gzip / gunzip*
- 6. Commande shell *grep*
 - Rechercher les fichiers dans le répertoire ‘/usr/bin’ avec une longueur de quatre caractères et se termine par un ‘r’
 - Rechercher toutes les lignes qui se terminent par ; dans le fichier *stdlib.h*
 - Une application particulière consiste à repérer les lignes vides dans un fichier.

- Rechercher les fichiers dans le répertoire `/usr/bin` avec une longueur de 4 caractères et se termine par un `r`
 - Rechercher les fichiers dans le répertoire `/usr/bin` dont les permissions pour le groupe sont `r x`
 - Sachant que `curl` permet de faire une requête HTTP et qu'en HTML la balise `<a>` indique un lien, que `wc` (*cf man wc*) compte des occurrences, combien y a-t-il de liens sur la page `http://www.insa-lyon.fr/?` (en une seule commande)
7. Placer les commandes qui suivent dans le fichier `~/temp/bonjour`, puis essayer le script :

```
#!/bin/bash
# Cette premiere ligne de commentaire est lue et
# sert a determiner quel shell sera appele pour executer
# ce script !
# pour appeler ce script : ./bonjour nom prenom
if test $# -eq 2; then
    echo "Bonjour $2 $1 et bonne journee !"
else
    echo "Syntaxe : $0 nom prenom"
fi
```

Dans ce script, on utilise la structure conditionnelle `if then else fi` du shell. Pour savoir si la condition est vraie ou fausse, le shell execute la commande parenthésée. Faites un `man test` pour comprendre comment s'utilise `test`.

8. Dans le shell, la variable prédefinie `$?` contient la valeur de sortie de la dernière commande. Par convention, la valeur 0 signifie qu'il n'y a pas eu d'erreur. Lorsqu'on utilise la commande `test`, cette variable est modifiée selon que la condition est vraie ou non. Essayer les commandes `test 0 -ne 1; echo $?` puis `test 1 -ne 1; echo $?`.
9. Ecrire un script qui affiche “fichier present” si l'argument de la commande correspond à un fichier qui existe (pour cela utiliser la commande `test`).
10. Ecrire un script qui affiche “repertoire present” si l'argument de la commande correspond à un répertoire qui existe.
11. Ecrire un script qui lance un processus en tâche de fond (par exemple `xclock &`), puis qui affiche le PID de ce fils en utilisant

la variable \$!. Utiliser la commande `wait $!` pour attendre la fin de ce fils au sein du script.

12. *Creer une commande qui, lorsqu'elle est appelee, affiche le nombre d'arguments qui lui sont fournis, ainsi que le premier de ces arguments, s'il y en a au moins un.*
13. *Que fait le script suivant :*

```
#!/bin/bash
((test $1 -lt $2) && (echo '$1 < $2')) || (echo '$2 < $1')
```

Apres avoir teste ce script, remplacez les apostrophes (') par des guillemets (""). Constatez que l'execuition de ce script produit un resultat different.

Recrivez ce script en utilisant une structure conditionnelle `if then else fi`.

14. *Que fait le script suivant ?*

```
#!/bin/bash
ls $1 2>/dev/null
if test $? -eq 0; then echo fichier existe;
else echo fichier inexistant; fi
```

Recrivez ce script en utilisant l'option `-e` de `test` et en supprimant l'appel a `ls`.

15. *Ecrire un script coupe qui prend trois arguments, dans l'ordre : un nom de fichier et deux entiers n et p, et qui affiche les lignes n a p du fichier. Pour cela, vous utiliserez les commandes `head` et `tail`.*
16. *Un processus shell possede des variables auxquelles sont affectees certaines valeurs. Voyons certaines commandes bash pour manipuler les variables. La liste des variables s'obtient avec la commande `set`. Pour affecter une valeur a une variable de type chaine de caractere, on utilise la syntaxe `VARIABLE="valeur"`. Lorsqu'une variable est definie, elle est accessible depuis le shell ou elle a ete definie. Pour qu'une variable soit accessible par des processus fils d'un certain shell, il faut exporter ces variables avec la commande `export VARIABLE`. La liste des variables exportees est accessible avec la commande `env`. Pour affecter le resultat d'une expression arithmetique a une variable, on peut utiliser `let` (commande de bash) de la maniere suivante : `let VAR2=$VAR1+1`. Tapez (dans cet ordre) les commandes suivantes, et comprenez le resultat :*

```
VAR1=11; VAR2=22; let VAR3=$VAR1+1; export VAR2
env | grep VAR.=
set | grep VAR.=
bash
env | grep VAR.=
set | grep VAR.=
exit
```

17. Ecrire un script affichant les nombres de 1 à jusqu'à la valeur passée en paramètre du script à l'aide d'une boucle `while do done`. Modifier enfin le script pour calculer la moyenne de ces nombres de tous ces nombres.
18. Examiner puis modifier le script qui suit. Vous ferez en sorte que lorsque l'utilisateur répond « o », alors :
- si `$file` est un répertoire alors on affiche le contenu de ce répertoire (commande `ls`) ;
 - si `$file` est un fichier de type lien on affiche ce lien ;
 - si `$file` est un fichier non executable on affiche le fichier (commande `less`).
- Si l'utilisateur répond « n », rien ne se passe, et s'il répond « q » le script doit s'interrompre. Vous utiliserez une structure `case in esac` pour considérer les trois réponses possibles : o, n et q.

Faire 4. Téléchargez et décompressez cette [archive](#), puis traitez les questions listées dans le fichier `instructions.md`.

Faire 5. Faites le [quizz](#) d'auto-évaluation bash sur Moodle.

Faire 6. Faites le [quizz](#) d'auto-évaluation ssh sur Moodle.

Bravo, vous êtes maintenant ceinture verte de bash. Vous voulez devenir ceinture noire ? Apprenez à maîtriser l'[art de la ligne de commande](#).

Chapitre 3

Structurer la compilation

Durée estimée : 2h

3.1 Pourquoi compiler ?

Lorsque vous implementez un algorithme dans votre langage de préférence, le code résultant est appelé **code source**. Il s'agit d'instructions écrites dans le langage de votre choix, intelligible pour l'homme. Malheureusement, ce code source n'est pas intelligible pour la machine. Il faut donc le transformer afin de fournir à la machine un code qu'il puisse comprendre. On distinguera deux types de langages informatiques intelligibles pour l'homme, les **langages interprétés** et les **langages compilés**. Contrairement aux premiers, ces derniers analysent et interprètent une fois pour toute le fichier source afin de générer un **fichier objet**, permettant en suite de construire un **fichier executable**. Le processus de génération d'un fichier executable à partir d'un ou plusieurs fichiers sources est appelé la compilation. Structurer cette opération permet de produire plus facilement les executables de gros projets.

La structuration de la compilation se fait au moyen d'un fichier **Makefile** via la commande **make**. Le fichier **Makefile** permet d'ordonnancer des tâches pour créer des fichiers ou d'associer un groupe d'instructions à une commande. Son utilisation est donc totalement agnostique du langage utilisé : il peut être utilisé pour faire autre chose que compiler des projets en C/C++, Java, etc.

Definition 4. *La commande **make** est outil très général permettant, entre autre, d'automatiser la compilation d'un projet.*

Supposons que le projet soit constitué des sources C suivants : main.c, structure.c, et operation.c . Il est possible de compiler ce projet de trois manières différentes, les parties suivantes précisent ces différentes étapes.

3.2 Une seule commande

```
gcc -o prog -Wall main.c structure.c operation.c
```

3.3 Une commande par fichier source

```
gcc -o prog -Wall main.c
gcc -o prog -Wall structure.c
gcc -o prog -Wall operation.c
```

Les commandes précédentes compilent chaque fichier source et créent les fichiers objets correspondants qui s'appellent : main.o, structure.o et operation.o. Un fichier objet est le résultat de la compilation d'un fichier source et contient les instructions machines associées à chaque fonction du fichier source, ainsi que la liste des fonctions appelées qui ne se trouvent pas dans le fichier source (par exemple, les fonctions des librairies standards printf, scanf, malloc, etc. et les fonctions que vous avez écrites mais qui se trouvent dans les autres fichiers sources (voir l'utilisation du mot clé extern)).

```
gcc -o prog main.o structure.o operation.o
```

Cette dernière commande crée l'exécutable prog en assemblant (cette partie de la compilation s'appelle l'édition de liens) les ensembles d'instructions associés à chaque fichier objet. C'est à ce moment que le compilateur vérifie qu'il connaît l'ensemble d'instructions associé à chaque fonction.

3.4 Avec Make

L'utilisation de make est relativement simple, une fois que l'on a compris que son rôle est de produire automatiquement la séquence de commandes permettant de construire un projet. Pour créer l'exécutable,

la première fois, il faut que make génère la séquence de commandes décrite dans la partie 2 :

```
gcc -c -Wall main.c
gcc -c -Wall structure.c
gcc -c -Wall operation.c
gcc -o prog main.o structure.o operation.o
```

De la même manière, après la modification de main.c, make doit générer la séquence de commandes suivante :

```
gcc -c -Wall main.c
gcc -o prog main.o structure.o operation.o
```

Le raisonnement qui nous a permis d'écrire cette séquence de commande est basé sur les dépendances (ou les relations) entre ces fichiers : `main.c` produit `main.o` et `main.o`, `structure.o` et `operation.o` permettent de construire le projet `prog`. Il suffit de décrire ces relations à `Make`, ainsi que les commandes associées pour qu'il puisse produire la séquence de commandes correcte. Ces relations sont décrites dans un fichier texte, nommé `Makefile` ou `makefile`. Ce fichier est constitué des descriptions des relations entre les fichiers sources et les fichiers objets ainsi que des relations entre les fichiers objets et le projet.

Une relation s'écrit de la manière suivante dans le fichier `makefile` :

```
produit : source
           commande
```

Cette règle indique plusieurs choses à make. Premièrement, que produit est créé à partir de source et que c'est commande qui permet de le faire. On peut donc décrire la relation entre `main.c` et `main.o`, qui est produit par la compilation de `main.c` :

```
main.o : main.c
gcc -c -Wall main.c
```

De même, la relation entre les fichiers objets et le projet s'écrit :

```
prog: main.o structure.o operation.o
gcc -o prog main.o structure.o operation.o
```

Pourachever la création du makefile, il ne reste plus qu'à écrire les règles pour `structure` et `operation` :

```

structure.o : structure.c
gcc -c -Wall structure.c

operation.o : operation.c gcc -c -Wall operation.c

```

Un dernier détail, comme le makefile est composé de plusieurs relations, il faut indiquer à make laquelle construire en priorité : par convention, c'est tout simplement la première. Il suffit donc d'écrire la règle décrivant la construction du projet au début du fichier makefile. Le makefile complet ressemblera donc à :

```

prog: main.o structure.o operation.o
gcc -o prog main.o structure.o operation.o

main.o : main.c
gcc -c -Wall main.c

structure.o : structure.c
gcc -c -Wall structure.c

operation.o : operation.c
gcc -c -Wall operation.c

```

Lors d'une modification, `make` se base sur la date de modification des fichiers pour déterminer les mises à jours à effectuer. Dans le scénario précédent, le projet complet est compilé et exécuté. Lors de ce premier appel à `make`, ni les fichiers objets, ni l'exécutable n'existent, ils sont donc créés en utilisant la commande associée à leur règle. Ensuite, `main.c` est modifié, `main.c` devient donc plus récent que `main.o` et `prog`. `Make` analyse les règles définies dans le makefile et vérifie les dates des fichiers. Comme `main.c` est plus récent que `main.o`, ce dernier ne peut être le produit de la version actuelle de `main.c`, il faut donc le recompiler, la commande associée à la règle `main.o` sera donc exécutée (`gcc -c -Wall main.c`). Après la compilation, la date de `main.o` est plus récente que celle de `prog`, il faudra donc récréer `prog` avec la commande correspondante (`gcc -o prog main.o structure.o operation.o`).

3.5 Exercices

Vous devez réaliser un petit programme qui prend deux paramètres, soit en ligne de commande via `argv`, soit en les demandant interactivement : un caractère `c` et un entier `i`. Le caractère `c` correspond à

un mot connu du programme et le programme affichera `i` fois ce mot.
Vous devez pouvoir afficher les mots suivants :

- `b` : bonjour
- `m` : merci
- `a` : au revoir

Exemple : si l'utilisateur tape '`b`' et `5`, alors votre programme affichera "bonjour bonjour bonjour bonjour bonjour". Vous devez impérativement structurer votre programme de la manière suivante :

- un fichier `fonctions.c` de fonctions qui contient (au moins) deux fonctions (et pas de `main`) : une qui prend en paramètre un caractère et affiche le mot correspondant, une qui prend en paramètre un caractère et un entier et qui appelle la fonction précédente le nombre de fois nécessaire
- un fichier `test.c`, contenant un `main`, qui fait quelques appels différents et successifs aux fonctions de `fonctions.c` pour vérifier qu'elles fournissent le résultat attendu
- un fichier `programme.c`, contenant un `main`, qui récupère les paramètres et appelle les fonctions de `fonctions.c`. Attention, un programme complet ne peut contenir qu'un seul `main`, il faut donc compiler ensemble soit `fonctions.c` et `test.c`, soit `fonctions.c` et `programme.c`.

Faire 7. Voici les tâches à réaliser.

1. Ecrivez les fichiers `.c`
2. Ecrivez les fichiers `.h`
3. Compilez les deux programmes avec des appels manuels à `gcc`
4. Ecrivez un Makefile pour automatiser la compilation. Vérifiez qu'une modification sur le fichier de `fonctions` provoque la recompilation des deux programmes. Cette structuration permet de dissocier un fichier contenant tous les tests d'un autre contenant le programme à livrer et nous attendons que vous fassiez cet effort lors de tout développement. Les fichiers de test réalisés feront partie intégrante de vos TP et pourront donc être présentés ou retournés.

Chapitre 4

Programmation python

Durée estimée : 8h à 12h

4.1 Faites vos premiers pas en Python

Vous allez maintenant apprendre à programmer dans un langage de haut niveau. Il a l'avantage d'être facile à prendre en main par un débutant et d'être puissant dans le sens où quelques lignes assez lisibles permettront au programmeur de faire effectuer un traitement complexe. Il est très utilisé, entre autres domaines, dans le web côté serveur (Django, Flask), en calcul scientifique (Numpy, Matplotlib, Sympy, SageMath), en machine learning (Pytorch), voire en administration système, remplaçant alors bash.

- [niveau débutant](#)
- niveau avancé, documentation officielle :
 - [tutoriel](#)
 - [HOWTOs](#)
 - [FAQ](#)

Attention, il y a différentes versions de python qui cohabitent encore ça et là. Python 2 est maintenant obsolète. Veillez à programmer en python 3 et vérifier la version des extraits de code que vous glanez sur le web.

4.1.1 Compétences ciblées

Savoirs :

1. Savoir la différence entre variables globales et locales

2. Savoir passer en paramètres d'une fonction des valeurs

Capacités :

1. Être capable de déclarer et initialiser une variable
2. Être capable de définir la portée d'une variable dans un programme
3. Être capable de définir une procédure munie de ses paramètres
4. Être capable de définir le programme principal
5. Être capable d'utiliser les structures conditionnelles.
6. Être capable d'utiliser les boucles
7. Être capable de gérer les exceptions

4.1.2 Déclarer et initialiser une variable

1. Variables réelles

- (a) Affectez les variables `temps` et `distance` par les valeurs 6.892 et 19.7 **`temps=6.892 et distance=19.7`**
- (b) Affichez la valeur de la `vitesse` **`vitesse=distance/temps`**
- (c) Améliorer la vitesse en imposant un chiffre après la point décimal **`round(vitesse, 1)`**

2. Interactions avec l'utilisateur

- Saisissez un `nom` et un `age` en utilisant l'instruction `input()`.
Puis les afficher. **`nom = input("Nom : ")`**
 `age = input("Age : ")`

4.1.3 Déclarer des procédures munies de paramètres

1. Écrire une procédure `table` avec quatre paramètres : `base`, `debut`, `fin` et `inc`. Cette procédure doit afficher la table des `base`, de `debut` à `fin`, de `inc` en `inc`. Tester la procédure par un appel dans le programme principal.
2. Écrire une fonction `cube` qui retourne le cube de son argument.
Écrire une fonction `volumeSphere` qui calcule le volume d'une sphère de `rayon r` fourni en argument et qui utilise la fonction `cube`. Tester la fonction `volumeSphere` par un appel dans le programme principal.

PROCEDURE :
def table(base, debut, fin, inc):
for x in (debut, fin, inc):
print(base[x])
print("Terminé")

APPEL :
table([3,5,23,'g','Hello'],0,4,1)

FONCTIONS :
from math import pi
def volumeSphere(r):
vol=(4*pi*cube(r))/3
return vol

APPEL :
volumeSphere(3)

def cube(n):
return n3**

4.1.4 Définir un programme principal avec sys.argv

1. Définir un programme principal qui affiche la vitesse puis le nom et l'âge saisis.
2. Écrire un programme qui liste les arguments passés en ligne de commande, cf. `sys.argv`
3. Déclarer la fonction `vitesse(temps, distance)`, qui affiche la vitesse
4. Déclarer la fonction `homme(nom, age)`, qui affiche le nom et l'âge
5. Écrire un programme qui affiche prend en paramètres le temps, distance, nom, et âge, et exécute les fonctions `vitesse(temps, distance)` et `homme(nom, age)`
6. Quelle est la portée de chacune des variables utilisées dans ce programme ? **variables globales : t, d, n, a**
variables locales : v

```
import sys
print (sys.argv)
```

```
def homme(nom, age):
    print("Nom : ",nom)
    print("Age : ",age)
```

```
vitesse=input()
nom = input()
age = input()
print(vitesse,nom,age)
```

```
def vitesse(temps, distance):
    v=distance/temps
    print("Vitesse : ",v)
```

```
t=2
d=6
n='Valentin'
a=21
vitesse(t,d)
homme(n,a)
```

4.1.5 Contrôler le flux d'instructions

Ecrire les programmes suivants dans des fonctions, et exécuter les dans un programme principal.

1. Saisissez un flottant. S'il est positif ou nul, affichez sa racine, sinon affichez un message d'erreur :
 - (a) d'abord en utilisant une boucle de contrôle `if / else`
 - (b) puis en utilisant les mots clés `try` et `except`
2. L'ordre *lexicographique* est celui du dictionnaire.
 - (a) Saisissez deux mots, comparez-les pour trouver le *plus petit* et affichez le résultat.
 - (b) Refaites l'exercice en utilisant l'instruction ternaire :
3. On désire sécuriser une enceinte préssurisée. On se fixe une pression seuil et un volume seuil : `pression_seuil = 2.3`, `volume_seuil = 7.41`. On demande de saisir la pression et le volume courant de l'enceinte et d'écrire un script qui simule le comportement suivant :
 - (a) si le volume *et* la pression sont supérieurs aux seuils : arrêt immédiat ;

```
from math import *
def racine_2():
    f = float(input("Nombre : "))
    try:
        root = sqrt(f)
        print("Sa racine vaut : ",root)
    except ValueError:
        print("Le nombre est négatif")
```

```
def racine():
    f = float(input("Nombre : "))
    if f >= 0:
        print("Sa racine vaut : ",f**0.5)
    else:
        print("Le nombre est négatif")
```

```
def lexico_2():
    mot_1 = str(input("Mot 1 : "))
    mot_2 = str(input("Mot 2 : "))
    petit = mot_2 if mot_1>mot_2 else mot_1
    print(petit,"se trouve avant dans le dictionnaire")
```

```
def lexico():
    mot_1 = str(input("Mot 1 : "))
    mot_2 = str(input("Mot 2 : "))
    if mot_1>mot_2:
        print(mot_2, "se trouve avant", mot_1)
    else:
        print(mot_1, "se trouve avant", mot_2)
```

```
pression_seuil = 2.3
volume_seuil = 7.41
p = input("Pression : ")
v = input("Volume : ")
if (p>pression_seuil) and (v>volume_seuil):
    print("Arrêt immédiat !")
elif (p>pression_seuil):
    print("Augmenter le volume de l'enceinte")
elif (v>volume_seuil):
    print("Diminuer la pression de l'enceinte")
else:
    print("OK")
```

```

a = 0
b = 10
while a<b:
    a+= 1
    print("a =", a)
while b>0:
    b-= 1
    if (b%2 != 0):
        print("b =", b)

```

```

while(1):
    a = int(input("Saisir un entier :"))
    if a>=1 and a<=10:
        print(a)
    else:
        print("Cet entier n'est pas entre 1 et 10")

```

```

mot = input()
for i in mot:
    print(i)

liste = [3,4,76,'f','mot',-8]
for i in liste:
    print(i)

```

```

for i in range(0,15,3):
    print(i)

```

```

for i in range(0,10):
    if i+1==5:
        break
    print(i+1)

```

```

for i in range(-3,4):
    try:
        print(sin(i)/i)
    except:
        print("Impossible pour i =",i)

```

- (b) si seule la pression est supérieure la pression seuil : demander d'augmenter le volume de l'enceinte ;
(c) si seul le volume est supérieur au volume seuil : demander de diminuer le volume de l'enceinte ;
(d) sinon déclarer que *tout va bien*.

Ce comportement sera implémenté par une alternative multiple.

4. Initialisez deux entiers : $a = 0$ et $b = 10$.
 - (a) Écrire une boucle affichant et incrémentant la valeur de a tant qu'elle reste inférieure celle de b .
 - (b) Écrire une autre boucle décrémentant la valeur de b et affichant sa valeur si elle est impaire. Bouclez tant que b n'est pas nul.
5. Écrire une *saisie filtrée* d'un entier dans l'intervalle 1 10, bornes comprises. Affichez la saisie.
6. Affichez chaque caractère d'une chaîne en utilisant une boucle **for**. Affichez chaque élément d'une liste en utilisant une boucle **for**.
7. Affichez les entiers de 0 15 non compris, de trois en trois, en utilisant une boucle **for** et l'instruction **range()**.
8. Utilisez l'instruction **break** pour interrompre une boucle **for** d'affichage des entiers de 1 10 compris, lorsque la variable de boucle vaut 5.
9. Utilisez l'instruction **continue** pour modifier une boucle **for** d'affichage de tous entiers de 1 10 compris, sauf lorsque la variable de boucle vaut 5.
10. Utilisez une **exception** pour calculer, dans une boucle évoluant de -3 3 compris, la valeur de $\sin(x)/x$.

```

for i in range(0,10):
    if i+1!=5:
        continue
    print(i+1)

```

4.2 Manipulez des objets en Python

Les questions identifiées par (*) sont difficiles et par conséquent réservées uniquement aux étudiants ayant terminés toutes les autres questions.

4.2.1 Compétences ciblées

Savoir :

1. écrire et utiliser une fonction
2. créer et éditer des objets

```
def table(base,debut,fin,inc):
    for i in range(debut,fin+1,inc):
        if i%base==0:
            print(i, "est un multiple de", base)
```

4.2. MANIPULEZ DES OBJETS EN PYTHON

29

4.2.2 Écrire et utiliser une fonction

- Écrivez une procédure `table` avec quatre paramètres : `base`, `debut`, `fin`, `inc`. Cette procédure doit afficher la table des multiples de `base`, allant de `debut` à `fin`, et construits de `inc` en `inc`. Testez la procédure par un appel dans le programme principal.
- Écrivez une fonction `cube` qui retourne le cube de son argument. Écrivez une fonction `volumeSphere` qui calcule le volume d'une sphère de rayon `r` fourni en argument et qui utilise la fonction `cube`. Testez la fonction `volumeSphere` par un appel dans le programme principal. (**Voir partie 4.1.3 question 2**)
- Écrivez une fonction `volMasseEllipsoide` qui retrouve le volume et la masse d'une ellipsoïde grâce à un tuple. Les paramètres sont les trois demi-axes et la masse volumique. On donnera ces quatre paramètres des valeurs par défaut. On donne : $v = \frac{3}{4}\pi abc$. Testez cette fonction par des appels avec différents nombres d'arguments.
- Écrivez une fonction `somme` avec un argument *tuple de longueur variable* qui calcule la somme des nombres contenus dans le tuple. Testez cette fonction par des appels avec différents tuples d'entiers ou de flottants.
- Écrivez une autre fonction `somme` avec trois arguments, et qui renvoie leur somme. Dans le programme principal, définir un tuple de trois nombres, puis utilisez la syntaxe d'appel à la fonction qui décompresse le tuple. Affichez le résultat.
- Écrivez une fonction `unDictionnaire` avec un argument *dictionnaire de longueur d'appel variable*, et qui affiche son argument. Dans le programme principal, définir un dictionnaire, puis utilisez la syntaxe d'appel à la fonction qui décompresse le dictionnaire. Affichez le résultat.

4.2.3 Créer et éditer des objets

- Définissez la liste : `liste = [17,38,10,25,72]`, puis effectuez les actions suivantes :
 - triez et affichez la liste ;
 - ajoutez l'élément 12 à la liste et affichez la liste ;
 - renversez et affichez la liste ;
 - affichez l'indice de l'élément 17 ;
 - enlevez l'élément 38 et affichez la liste ;

```
def somme(tuple):
    x = 0
    for i in range(0,len(tuple)):
        x = x+tuple[i]
    return x
```

```
from math import pi

def volMasseEllipsoide(a,b,c,p):
    v = 3/4*pi*a*b*c
    m = p*v
    return v,m
```

```
def somme(a,b,c):
    x = a+b+c
    return x

tuple = (2,4,5)
print(somme(tuple[0],tuple[1],tuple[2]))
```

- affichez la sous-liste du 2^e au 3^e élément ;
 - affichez la sous-liste du début au 2^e élément ;
 - affichez la sous-liste du 3^e élément à la fin de la liste ;
 - affichez la sous-liste complète de la liste ;
 - affichiez le dernier élément en utilisant l’indication négatif.
- Bien remarquer que certaines méthodes de listes ne retourne rien.
2. Initialisez `truc` comme une liste vide, et `machin` comme une liste de cinq flottants nuls. Affichez ces listes. Utilisez la fonction `range()` pour afficher :
 - les entiers de 0 à 3
 - les entiers de 4 à 7
 - les entiers de 2 à 8 par pas de 2.
 Définir `chose` comme une liste des entiers de 0 à 5 et testez l’appartenance des éléments 3 à 6 dans `chose`.
 3. Utilisez une liste en compréhension pour ajouter 3 à chaque élément d’une liste d’entiers de 0 à 5.
 4. Utilisez une liste en compréhension pour ajouter 3 à chaque élément d’une liste d’entiers de 0 à 5, mais seulement si l’élément est supérieur ou égal à 2.
 5. Utilisez une liste en compréhension pour obtenir la liste [`ad`', '`ae`', '`bd`', '`be`', '`cd`', '`ce`'] partir des chaînes `abc`' et '`de`'.
Indication : utilisez deux boucles `for` imbriquées.
 6. Utilisez une liste en compréhension pour calculer la somme d’une liste d’entiers de 0 à 9.
 7. Définissez deux ensembles : $X = \{a, b, c, d\}$ et $Y = \{s, b, d\}$, puis affichez les résultats suivants :
 - les ensembles initiaux ;
 - le test d’appartenance de l’élément ‘c’ à X ;
 - le test d’appartenance de l’élément à à Y ;
 - les ensembles $X - Y$ et $Y - X$;
 - l’ensemble $X \cup Y$ (union) ;
 - l’ensemble $X \cap Y$ (intersection).
 8. Écrivez une fonction `compterMots` ayant un argument (une chaîne de caractères) et qui renvoie une *dictionnaire* qui contient la fréquence de tous les mots de la chaîne entrée.
 9. Le type *dictionnaire* (ou tableau associatif) permet de représenter des tableaux structurés. En effet, chaque *clé* un *dictionnaire* associe une *valeur*, et cette valeur peut elle-même être une structure de données (liste, tuple ou un *dictionnaire* …).

Soit le tableau suivant représentant des informations physio-chimiques sur des éléments simples (température d'ébullition (T_e) et de fusion (T_f), numéro (Z) et masse (M) atomique :

Au	T_e/T_f	2970	1063
Au	Z/A	79	196.967
Ga	T_e/T_f	2237	20.8
Ga	Z/A	31	69.72

Affectez les données de ce tableau à un dictionnaire `dico` python de façon à pouvoir écrire par exemple :

```
>> print dico["Au"]["Z/A"][0] # affiche : 79
```

10. (*) Implémentez une pile LIFO avec une liste. Pour cela, définir trois fonctions :
 - `pile` : qui retourne une pile à partir d'une liste variable d'éléments passés en paramètres ;
 - `empile` : empile un élément en *haut* de la pile ;
 - `depile` : dépile un élément du *haut* de la pile.
11. (*) De la même manière, implémentez une queue FIFO avec une liste. Essayez d'ajouter un menu de manipulation de la queue.

4.3 Créer des objets en Python

4.3.1 Compétences ciblées

Savoir :

1. écrire et lire un fichier
2. créer un module
3. créer et utiliser une classe

4.3.2 Créez des modules

1. Écrire un module de calcul des racines du trinôme réel : ax^2+bx+c . Le module définit une fonction `trinome` avec trois paramètres du trinôme, a , b , et c . La fonction doit retourner une tuple dont le premier élément est le nombre de racines du trinôme (0, 1, ou 2), et les autres éléments sont les racines éventuelles.
Testez votre fonction avec trois jeux de valeurs suivantes : 1, -3, 2, 1 ; 1, -2, 1 et 1, 1, 1.

4.3.3 Manipulez un fichier

1. Créez un fichier `NbLigne` qui a pour paramtre un *nom de fichier existant* (.txt) et qui renvoie le nombre de lignes de ce fichier.
2. Créez un fichier texte nommé `La_mesure_de_lhomme.txt` et écrivez la chaîne de caractres suivante dedans :
“La mesure de l’homme.

Ce n est pas celui qui critique qui est important, ni celui qui montre du doigt comment l homme fort trébuche ou comment l homme d action aurait pu faire mieux.

L hommage est dû celui qui se bat dans l arme, dont le visage est couvert de poussire et de sueur, qui va de l avant vaillamment, qui commet des erreurs et en commettra encore, car il n y a pas d efforts humains sans erreurs et imperfections. C est lui ou elle qu appartient l hommage, celui u celle dont l enthousiasme et la dévotion sont grands, celui ou celle qui se consume pour une cause importante, celui ou celle qui, au mieux, connaîtra le triomphe du succès, et au pis, s il échoue, saura qu il a échoué alors qu il risquait courageusement.”

3. Écrivez le contenu du fichier `Demi.txt` (créer et éditer au préalable) dans le fichier `La_mesure_de_lhomme.txt`.
4. Affichez le contenu du fichier `La_mesure_de_lhomme.txt`.

4.3.4 Créez et utilisez une classe

1. Définissez la class `MaClasse` possédant les attributs suivants :
 - données : deux attributs de classes : $x = 23$ et $y = x + 5$.
 - méthode : une méthode `affiche` contenant un attribut d’instance $z = 42$ et les affichages de y et de z .

Dans le programme principal, instanciez un objet de la classe `MaClasse` et invoquez la méthode `affiche`.

2. Définir une classe `Vecteur2D` avec un constructeur fournissant les coordonnées par défaut d’un vecteur du plan (par exemple : $x = 0$ et $y = 0$). Dans le programme principal, instanciez un `Vecteur2D` sans paramtre, un `Vecteur2D` avec ses deux paramtres, et affichez-les.
3. Enrichissez la classe `Vecteur2D` précédent en lui ajoutant une méthode d’affichage et une méthode de surcharge d’addition de deux

vecteurs du plan. Dans le programme principal, instanciez deux `Vecteur2D`, affichez-les et affichez leur somme.

4. Définissez une classe `Rectangle` avec un constructeur donnant les valeurs (`longueur` et `largeur`) par défaut et un attribut `nom = "rectangle"`, une méthode d'affichage et une méthode `surface` renvoyant la surface d'une instance.
 - Définissez une classe `Carre` héritant de `Rectangle` et qui surcharge l'attribut d'instance : `nom = "carré"`. Dans le programme principal, instanciez un `Rectangle` et un `Carre` et affichez-les.
 - Définissez une classe `Point` avec un constructeur fournissant les coordonnées par défaut d'un point du plan (par exemple : `x=0.0` et `y = 0.0`).
5. Définissez une classe `Segment` dont le constructeur possde quatre paramtres : deux pour l'origine et deux pour l'extrémité. Ce constructeur définit deux attributs : `orig` et `extrem`, instances de la classe `Point`. De cette manire, vous concevez une classe *composite* : la classe `Segment` est composée de deux instances de la classe `Point`.
 - (a) Ajouter une méthode d'affichage.
 - (b) Enfin écrivez un auto-test qui affiche une instance de `Segment` initialisée par les valeurs 1, 2, 3 et 4.

Chapitre 5

Git, Github

Durée estimée : 4h

5.1 Git, Github

- [Git](#)
- [Github](#)

5.2 Exercices

Faire 8. Téléchargez cette [archive](#). Elle contient des fichiers à compléter pour réaliser un programme python capable de résoudre une grille de Sudoku. L'objectif est à la fois de coder un tel programme, mais surtout de développer à plusieurs (au moins deux!) et se familiariser ainsi avec git et github.

```
git init : initialise un projet      // à faire 1 seule fois
git clone : télécharge la version initiale d'un projet à partir d'une machine distante (ex : github.com)      // à faire 1 seule fois
git add <fichier> : ajouter un fichier et son état au projet (*.<extension>, --all...) et attend le prochain commit
git rm <fichier> : supprime un fichier et son état du projet (*.<extension>, --all...) et attend le prochain commit
git commit -m "nom_commit" : archive les modifications sous le nom "nom_commit"
git status : affiche l'état du projet
git log : affiche l'historique du projet
git diff <fichier> : affiche les différences entre un fichier et ce même fichier dans le commit précédent
git checkout . : indique que le répertoire courant '.' doit revenir à l'état du commit actuel (ex : si projet supprimé par erreur)
git checkout <id_commit> <fichier> : récupère un fichier dans un commit précis
git reset HEAD <fichier> : permet d'annuler l'intégration d'une des modifications dans le commit
git push -u origin main : sélection le commit pour le mettre dans la branche main
git remote add origin <lien github> : ajoute le projet au repository du lien choisi
```

POUR EXPORTER SUR GITHUB

35

- `git init`
- `git add <fichier>`
- `git commit -m "first commit"`
- `git remote add origin <lien>`
- `git push -u origin main`