

Tutorat M1 - Bash & Python

Amélie Gruel

Septembre 2019

1 Partie 1 - Bash

Pour commencer ce tutorat, on va d'abord créer à l'aide de commandes bash un répertoire sur le Bureau, intitulé `Tutorats`. Déplace toi dans ce répertoire, puis ajoutez y un répertoire `Tutorat1`.

Ensuite, balade toi dans tes répertoires. Petit conseil : l'idéal serait d'avoir un fichier `M1`, puis un sous fichier `S1` et des sous-sous-fichiers correspondant vos différents cours !

Si tu n'as pas encore organisé ton espace de travail, crée au moins un répertoire `M1` dans les Documents, et déplace-y donc le répertoire `Tutorats` qu'on a créé précédemment dans le Bureau.

Déplace toi à l'intérieur de ce fichier, puis dans celui `Tutorat1` : une fois à l'intérieur, crée un fichier `exercice1`.

Finalement, tu as changé d'avis et tu veux apporter de la joie dans ton terminal : renomme le dossier `Tutorats` en `TutoratParLesMeilleursM2AuMonde` (si si, ça met des paillettes dans ta vie je te jure).

Maintenant que tu as aménagé ton espace de travail de manière optimale, tu vas enfin pouvoir t'attaquer à Python : modifie le nom de ton fichier `exercice1` afin de pouvoir l'utiliser en script Python.



2 Partie 2 - Python : les bases

2.1 Exercice 1

Pour commencer à coder un script Python, ouvre le fichier `exercice1.py` dans un éditeur de code.

NB : il existe de nombreuses commandes pour ouvrir un script dans un éditeur particulier à partir du terminal. Entre autres :

- `code nomdefichier` pour ouvrir le script dans Visual Studio Code
- `atom nomdefichier &` pour ouvrir le script dans Atom
- `emacs nomdefichier &` pour ouvrir le script sur Emacs

Question a

Rédige un script qui demande un nombre entier à l'utilisateur, puis affiche successivement les entiers allant de 1 à ce nombre avec un pas de 1.

Le terminal affichera 1, 2, 3, etc jusqu'au nombre donné.

Question b

Reprends le script précédent en indiquant pour chaque chiffre si il est pair ou impair.

Petit indice : il faut utiliser une boucle if/elif.

2.2 Exercice 2

Question a

Écris un script qui définit au hasard un entier compris entre 1 et 10, puis demande à l'utilisateur un chiffre jusqu'à ce qu'il trouve le bon. Tant que l'utilisateur ne trouve pas le bon nombre, le terminal lui demandera un nouveau chiffre. Une fois le bon chiffre trouvé, le terminal affichera un message de félicitation.

Il faut utiliser une boucle while.

Question b

Reprends le script précédent, en le modifiant afin qu'après chaque essai raté le terminal indique à l'utilisateur si le chiffre trouvé est plus petit ou plus grand.

2.3 Exercice 3 - BONUS

La suite de Fibonacci est une suite réursive se comportant ainsi :

$$\begin{cases} U_{n+2} = U_{n+1} + U_n \\ U_0 = 0 \\ U_1 = 1 \end{cases}$$

Crée un programme qui demande à l'utilisateur pour quel n veut-il obtenir le U_n correspondant, puis lui retourne ce résultat.

3 Partie 3 - Python : y a qui dans le master déjà ?



3.1 Exercice 1

Crée une liste nommée `master`, contenant les noms de 5 M2.

Ensuite, ajoute un nom de professeur dans cette liste.

Finalement, tu te dis que les M2 tous seuls sont quand même plus cools, du coup enlève le professeur de ta liste ! Attention, fais le sans récrire entièrement ta liste.

Enfin, affiche le contenu de ta liste sur ton terminal ainsi que sa taille.

Chacune de ces actions doivent avoir lieu dans le même script, sur des lignes successives.

3.2 Exercice 2

Maintenant, on aimerait recenser tous les membres du Master, qu'ils soient en M1, M2, M3 et même les profs !

Mais que faudrait-il utiliser afin de stocker le nom des membres, et y associer leur statut (M1, M2, M3, prof)?

Utilise cette structure de données, que l'on nommera `master_complet`, afin de recenser au moins 6 personnes.

Ajoute y une personne après l'avoir initialisé.

Ensuite, affiche le statut de la personne de ton choix.

Enlève une personne de ton choix.

Maintenant, affiche le contenu de ta structure.

Enfin, affiche uniquement les élèves et professeurs, puis uniquement les statuts possibles en utilisant une méthode associée à ta structure de données.

Comme tout à l'heure, chacune de ces actions doivent avoir lieu dans le même script, sur des lignes de successives.

3.3 Exercice 3 - BONUS

L'utilisateur aimerait pouvoir recenser les membres de son Master à sa guise, en choisissant lui-même au fur et à mesure les actions qu'il veut effectuer. Rédige donc un programme contenant un menu, à l'aide duquel l'utilisateur pourra choisir entre :

1. créer un dictionnaire vide
2. y ajouter un membre du Master et son statut
3. enlever un membre de son choix
4. afficher le contenu du dictionnaire. Chaque nom sera affichée sur une ligne successive, avec le statut correspondant indiqué à la suite entre parenthèses.
5. obtenir le statut d'un certain membre. Le terminal devra alors demander à l'utilisateur d'entrer un nom, puis retournera le statut correspondant.
6. obtenir le nom des tous les membres ayant un certain statut. Le terminal devra alors demander à l'utilisateur d'entrer un statut, puis retournera tous les membres correspondant.
7. quitter le menu

Une fois que tu en es à ce point, préviens un M2 afin qu'il t'explique comment faire un menu.

4 Partie 4 - Python : un petit tour dans le CREMI

Ca y est, les premiers cours dans le CREMI ont eu lieu, tu commences enfin à te repérer à l'intérieur. Mais catastrophe !! il est 8h12, tu es à la bonne salle mais tu viens de recevoir un mail disant que finalement le cours a lieu 2 étages plus haut !

Il faut que tu te dépêches d'aller jusque là bas. Représente donc le CREMI à l'aide d'une liste de liste. Cette liste à deux dimensions génère une grille 10x10, dans laquelle tu devras te déplacer. Ta position initiale en $x=5$ et $y=5$ sera indiquée par un caractère (tel que "X").

Crée une fonction te permettant d'afficher le contenu de ta grille sur ton terminal. Une manière possible de l'afficher serait comme dans la figure 1.

```
(base) amelie@L-E7-SAMMY:~/Documents/M2/Tutorats$ python2 PetitTourDansLeCREMI.py
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 X 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Figure 1: Terminal affichant la grille représentant le CREMI. Les 0 indiquent les cases dans lesquelles tu peux te déplacer, le X la position de ton joueur. Ici il est en position $x=5$ et $y=5$.

Une fois que cela est fait, implémente une fonction te demandant dans quelle direction tu veux te diriger, puis effectuant ce déplacement. A la fin de chaque déplacement, la nouvelle grille devra être affichée sur ton terminal (tu peux pour cela appeler la fonction précédent).

Attention ! Tu ne peux pas te déplacer en dehors de la grille.



5 Partie 5 - Python : il était une fois

Télécharge le fichier "partie5.txt", qui contient le script d'un chef d'oeuvre du cinéma américain. Avant qu'un personnage parle, son nom est indiqué ainsi : "> NOMPERSO", sur une seule ligne. Enregistre ces noms dans une liste, afin de connaître quels sont les personnages prenant part au dialogue. Attention à ne pas en mettre en double !

On veut maintenant savoir qui est le personnage principal : pour cela, crée une fonction qui parcourt le fichier ligne par ligne et compte le nombre de fois où chacun des personnages trouvés ci-dessus parle. Le personnage principal sera donc celui dont le nom apparaît le plus souvent.

Pour faire cela, utilise un dictionnaire : la clé sera le nom du personnage, et la valeur sera augmentée de 1 à chaque mot lu correspondant à la clé.

Enfin, tu ne veux pas avoir à lire le dictionnaire afin de trouver qui parle le plus, mais tu veux que ton programme le fasse pour toi ! Rajoute donc une fonction à ton script afin qu'il identifie quel personnage parle le plus (c'est-à-dire quel personnage a la plus grande valeur associée dans le dictionnaire).

Maintenant que tu sais qui parle le plus, enregistre le contenu du dictionnaire obtenu précédemment dans un fichier `blablabla.txt`. Les personnages doivent apparaître selon le nombre de fois où chacun parle : celui qui parle le plus est sur la première ligne, et celui parlant le moins en dernier.



6 Partie 6 - Git/Github utiliser le versioning

6.1 Définitions

Blobs = Binary Large Object.

Chaque version d'un fichier est représenté par un Blob. Un blob garde les données du fichier. Son nom est le SHA1 du fichier. Git n'adresse pas les fichiers par nom mais par contenu de fichier.

Trees

Un arbre est un objet il représente un répertoire. Il conserve les BLOBs et les sous-répertoire. Il s'agit d'un binaire qui contient les références aux BLOBs et aux arbres qui sont aussi nommé selon le SHA1 du TREE.

Commits

Un commit contient l'état du dépôt (comme pour le reste il est nommé avec son SHA1). Un commit peut être considéré comme un noeud. Chaque objet possède un pointeur sur le commit parent (ce qui implique que l'on peut remonter dans l'historique des commits). Si un commit à plusieurs parents c'est qu'il est issu d'une fusion de branche.

Branches

Les branches sont utilisé pour créer une autre ligne de développement. Par défaut Git a une branche appelé Master (comme Subversion). Une branche correspond a une nouvelle fonctionnalité. Une fois que la fonctionnalité est mature on fusionne la branche avec la branche Master est on supprime la branche. Chaque branche est référencé dans le HEAD qui pointe sur le dernier commit de la branche (il est actualisé à chaque commit).

Tags

Un tag permet de donner un nom compréhensif à une version du dépôt. Il s'agit d'une branche immuable (non modifiable). Quand un tag est créé pour un commit il faut créer un autre commit car celui-ci n'est plus modifiable. En général les tags sont créé pour les versions officielle d'un programme.

Clone

Le clonage est une opération qui permet de créer un miroir du dépôt. Les dépôts peuvent ensuite être synchronisé.

Pull

Pull est une opération permet de copier les changements depuis un dépôt distant dans le dépôt local. Elle est utilisée afin de synchroniser deux dépôts. (correspond à update dans subversion)

Push

Push est opération qui copies les changements d'un dépôt local vers un dépôt distance. Il permet de conserver de manière permanente les changements dans un dépôt Git. (correspond à commit dans subversion)

HEAD

Le HEAD est stocké dans le répertoire !: `.git/refs/heads/`

```
• $ ls .git/refs/heads/  
  master  
  
• $ cat .git/refs/heads/master  
4e1243bd22c66e76c2ba9eddc1f91394e57f9f83
```

Revision

Revision représente la version du code source source. Les révisions sont représenter dans Git par les commits (leur SHA1)

URL

URL est la localisation du dépôt. L'URL est enregistré dans le fichier de configuration.

```
• $ cat .git/config  
[core]  
repositoryformatversion = 0  
filemode = true  
bare = false  
logallrefupdates = true  
[remote "origin"]  
url = https://github.com/axel-polin/Tutorat.git  
fetch = +refs/heads/*:refs/remotes/origin/*  
[branch "master"]  
remote = origin  
merge = refs/heads/master
```

6.2 Exercices

Suivre en live.

Site de référence : Tutorial's Point

Les lignes de commandes utilisées pour références seulement elles ne sont pas adaptées à tous les environnements :

```
295 git clone git@github.com:axel-polin/tutoratM1.git
296 cd tutoratM1/
297 ls
298 cd ..
299 git config --global user.name "axel-polin"
300 git config --global user.email "univ_apolin@protonmail.com"
301 git config --global color.status auto
302 git config --global color.branch auto
303 git config --global core.editor nano
304 git config --global merge.tool diff
305 git config --list
306 mkdir tutorat
307 cd tutorat
308 git init
309 ls
310 git status -s
311 echo '#This is my first README' > README.md
312 ls
313 git status -s
314 git add .
315 git status -s
316 git commit -m "This is my first commit"
317 git log
318 git remote add origin git@github.com:axel-polin/tutoratM1.git
319 git push origin master
320 git status -s
321 git log
322 ls
323 cd ..
324 ls
325 mkdir tutorat_clone
326 cd tutorat_clone/
327 git clone git@github.com:axel-polin/tutoratM1.git
328 ls
329 cd tutoratM1/
330 ls
331 touch script.py
332 nano script.py
333 git add script.py
334 git status -s
335 git commit -m 'My first code !'
336 git log
337 git show 111fd693275c026f8360e456bec8f04906e09360
338 git diff
339 nano script.py
340 git add script.py
```

```
341 git status -s
342 git commit --amend -m 'Hello message modified'
343 git log
344 git show cd85cd0ec97fa0bb7d5e2b1d297a7279992dcfc6
345 git push origin master
346 git pull
347 nano script.py
348 git status -s
349 git stash
350 git status -s
351 git stash pop
352 git status -s
353 git commit -m "Stash popped"
354 git add .
355 git commit -m "Stash popped"
356 git push origin master
357 mkdir src
358 git mv script.py src/
359 git status -s
360 git commit -m "Directory structure modified"
361 git push origin master
362 git mv src/script.py src/scriptRenamed.py
363 git commit -m "Script renamed"
364 git push origin master
365 history
366 ls
367 cd src
368 git rm scriptRenamed.py
369 status -l
370 status -s
371 git status -s
372 ls
373 cd ..
374 git status -s
375 git commit -a -m "Remove script"
376 git push origin master
377 ls
378 git add .
379 git commit -m "RAZ"
380 git push origin master
381 git log
382 nano src/script.py
383 git status -s
384 git checkout src/script.py
385 git status -s
386 cat src/hello.c
387 cat src/script.py
```

```

388 rm src/script.py
389 ls src/
390 git status -s
391 git checkout src/script.py
392 git status -s
393 ls -l src
394 nano src/hello.c
395 git status -s
396 git add src/hello.c
397 git status -s
398 git checkout HEAD -- src/hello.c
399 git status -s
400 cat .git/refs/heads/master
401 git log -2
402 git reset --soft HEAD~
403 cat .git/refs/heads/master
404 git log -2
405 ls -l src
406 git reset --hard 46553b44119f213a4217506d2b06f8a93664d7e2
407 ls -l src/
408 git tag -a 'Release_1_0' -m 'My first tag' HEAD
409 git push origin tag Release_1_0
410 git pull
411 git status -s
412 git tag -l
413 git show Release_1_0
414 git tag -d Release_1_0
415 ls -l
416 ls src/
417 nano src/hello.c
418 git add src/hello.c
419 git status -s
420 git commit -m "Added a new line in hlle.c"
421 git format-patch -1
422 l
423 ls
424 git push origin master
425 git commit -a -m "My first patch"
426 git add .
427 git commit -a -m "My first patch"
428 git push origin master
429 cd ..
430 git clone https://github.com/axel-polin/tutoratM1
431 cd ..
432 cd tutorat
433 cd ..
434 cd tutorat_clone/

```

```
435 rm -rf tutoratM1/
436 git clone https://github.com/axel-polin/tutoratM1
437 cd tutoratM1/
438 ls
439 git apply 0001-Added-a-new-line-in-hlle.c.patch
440 nano src/hello.c
441 git apply 0001-Added-a-new-line-in-hlle.c.patch
442 nano src/hello.c
443 git status -s
444 git branch new_branch
445 git branch
446 git checkout new_branch
447 git branch
448 git checkout -b test_branch
449 git branch
450 git checkout master
451 git branch -D test_branch
452 git branch
453 git branch -m new_branch branche1
454 git branch
455 git checkout branche1
456 nano src/hello.c
457 git status -s
458 git status -s
459 git add src/hello.c
460 git commit -m 'Add a line in hello.c'
461 git push origin branche1
462 git log origin/branche1 -2
463 git checkout master
464 git merge
465 git merge origin/branche1
466 git log -1
467 git branch
468 git checkout branche1
469 nano src/hello.c
470 git add src/hello.c
471 git commit -m "Add a new line 2"
472 git push origin branche1
473 git checkout master
474 nano src/hello.c
475 git diff
476 git add src/hello.c
477 git commit -m "Add a line in hello.c in master"
478 git push origin master
479 git checkout branche1
480 nano src/hello.c
481 git diff
```

```
482 git add src/hello.c
483 git commit -m 'Add a comment'
484 git push origin origin branch1
485 echo "resolve a conflict !!"
```