université PARIS-SACLAY

M1 BIBS

Langages de scripts : Python

Séance 2

(Correction)

1 Objectifs :

- Savoir utiliser les dictionnaires
- Savoir déclarer et utiliser les fonctions
- Savoir mettre en place des méthodes de gestion d'erreur
- Savoir mettre en place des méthodes de test
- S'initier aux gestionnaires de versions et en particulier aux outils Git/GitHub.

Remarque :
Il y a quelques différences entre Python 2.x et Python 3.x. Notamment, l'utilisation de la fonction `print` nécessite dans la version 3 l'utilisation de parenthèses, sinon l'interpréteur signalera une erreur. La division "/" calcule aussi dans la version 3 le résultat directement en float, ce qui n'était pas le cas sous les versions antérieures. Dans le doute, pensez à transformer au moins un des membres de vos divisions en float ! Enfin, la fonction `range()` ne retourne plus une liste en 3.x, il faut alors utiliser la fonction `list()` pour faire la conversion (ex : `list(range(10))`. Lorsque vous aurez à transmettre un programme Python, pensez à indiquer dans un README.txt la version avec laquelle le programme fonctionne !

2 Exercice de rappels sur la séance 1

Écrivez un programme qui génère 10 points aléatoires dans un plan cartésien limité à [0,9] sur les 2 axes. Pour cela, généré une liste de listes de 2 éléments entiers entre 0 et 9. Générez une chaîne de caractères qui contient une représentation graphique de vos points. On peut par exemple utilisez le caractère de la touche 6 de votre clavier (la barre verticale) pour représenter l'axe vertical, le "_ " pour représenter l'horizontal et le "+ " pour représenter un point. Le caractère spécial de saut de ligne (antislash + n) permettra de faire des retours à la ligne. Essayez de mettre également les numéros sur les axes. Affichez cette chaîne de caractères.

Correction :

```
from random import randint

l_points = list()
    for i in range(10):
        l_points.append([randint(0,9),randint(0,9)])
    print l_points

    result=""
    for i in range(10):
        for j in range(10):
            x=j #Pour revenir en cartésien
            y=9-i #idem
            if [x,y] in l_points:
                if j==0 and i!=9: #On pourrait tester sur x et y aussi
                    result+="\n"+str(y)+"|* "
```

```
                elif j==0 and i==9:
                    result+="\n"+str(y)+"|*_"
                elif j!=0 and i==9:
                    result+="*_"
                else:
                    result+="* "
            else:
                if j==0 and i!=9:
                    result+="\n"+str(y)+"|  "
                elif j==0 and i==9:
                    result+="\n"+str(y)+"| _"
                elif j!=0 and i==9:
                    result+="_"
                else:
                    result+="  "
    result+="\n   0 1 2 3 4 5 6 7 8 9"

    print result
```

Cela devrait donner quelque chose comme ça :

```
9| *   *
8|    ***
7|
6|*    *
5|*
4|
3|
2|     *
1|
0|___*______*_
 0123456789
```

---

3 Les dictionnaires

Le dictionnaire est la dernière structure complexe que nous verrons dans cette UE. C'est aussi une structure très utile qui permettra de manipuler de nombreuses données trop complexes pour l'être avec des structures plus simples.

— Initialiser un dictionnaire : Pour initialiser un dictionnaire, il suffit d'utiliser l'une des syntaxes suivantes :

```
myDico1 = dict()
myDico2 = {}
```

— Rappelons que les listes contiennent des éléments quelconques auxquels on peut accéder en parcourant des indices ordonnés (i.e. des entiers allant de 0 à N). Les dictionnaires contiennent aussi des éléments quelconques (que l'on appelle "valeurs"), mais à la différence des listes, on y accède via n'importe quel objet non-mutables (int, float, string, tuple, ...) que l'on appelle "clés", qui peuvent être d'un type quelconque. Voici un exemple de dictionnaire :

```
d_logins = dict()
print d_logins
d_logins["Prenom"] = "Arya"
d_logins["Nom"] = "Stark"
d_logins["pwd"] = "Nymeria"
d_logins["age"] = 10
d_logins["freres"] = ["Jon Snow", "Robb Stark", "Bran Stark"]
print d_logins, d_logins["Prenom"]
```

Notez la syntaxe d'un dictionnaire renvoyée par le print, c'est aussi la syntaxe pour initialiser des dictionnaires non-vides. Ici `d_logins` contient plusieurs informations liées à Arya. On accède à l'information avec une "clé" précise. Par exemple, la clé "Prenom" (type = chaîne de caractères) renvoie également une chaîne de caractères, "Arya".

— La plupart du temps, on utilise une chaîne de caractères comme clé, mais on peut utiliser n'importe quel objet non-mutable du Python. Vous pouvez tester le code suivant pour vous en assurer :

```
d_logins[1] = "A Game of Thrones"
d_logins[2] = "A Clash of Kings"
d_logins[(1,2)] = 42
print d_logins
```

— A présent, comment peut-on parcourir un dictionnaire ? On peut pour cela utiliser les méthodes `keys()` et `values()` qui permettent respectivement d'accéder à toutes les clés et à toutes les valeurs d'un dictionnaire. Testez ce qui est retourné par ces méthodes :

```
print d_logins.keys(), type(d_logins.keys())
print d_logins.values(), type(d_logins.values())
```

La solution la plus courante est de parcourir la liste des clés et de récupérer les valeurs associées :

```
for key in d_logins.keys():
    print "key="+str(key)+" -> value="+str(d_logins[key])
```

Remarque : Lors des `print` précédents, vous aurez peut-être constaté que la liste des clés n'est pas vraiment ordonnée, à la différence des éléments d'une liste. Imaginons que nous utilisions un dictionnaire ayant 0, 1 et 2 comme clés et des valeurs quelconque, l'accès aux valeurs se fera de la même façon qu'avec une liste. Néanmoins, si vous supprimez la clé 2 par exemple, le dictionnaire ne réorganisera pas les indices comme dans les listes. Si vous voulez parcourir les clés d'une manière ordonnée, il vous faudra trier vous-même la liste des clés.

— Exercice : Soit la séquence suivante : ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG (c'est un exemple, si vous ne pouvez pas faire un copier-coller, ne perdez pas de temps à recopier exactement cette séquence !). En utilisant un dictionnaire, faites un programme qui répertorie tous les mots de 2 lettres qui existent dans la sequence (AA, AC, AG, AT, etc.) ainsi que leur nombre d'occurrences puis qui les affiche à l'écran.

Correction :

```
seq = "ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG"
d_freqMot2Lettres = dict()
for i in range(len(seq)-1):
    l2 = seq[i]+seq[i+1]
    if l2 in d_freqMot2Lettres.keys():
        d_freqMot2Lettres[l2] += 1
    else:
        d_freqMot2Lettres[l2] = 1
print d_freqMot2Lettres
```

4 Les fonctions en Python

Si vous suivez bien, il ne reste qu'un point important à aborder : les fonctions. Comme dans n'importe quel langage de programmation, la bonne pratique est de définir les fonctions de son programme dans des fichiers séparés du script principal (celui qui sera exécuté, i.e. l'équivalent de celui contenant le `main` en C par exemple).

— Définition de fonction : Pour définir une fonction en Python, il faut suivre cette syntaxe :

```
def NomDeMaFonction():
        # Bloc d'instructions

def NomDeMaFonctionAvecParam(a, b, c):
        # Bloc d'instructions
```

Comme partout en Python, on ne déclare pas le type des arguments. A la différence du C, cela a un impact sur la signature de la fonction : seul le nom de la fonction permet de la différencier d'une autre. Veillez donc bien à ne pas surcharger le nom de vos fonctions.

— Bien commencer sa fonction en la commentant : Avant même de commencer à écrire un bloc d'instructions, il est recommandé de mettre en petit commentaire sur ce qu'elle fera. La façon classique de le faire est :

```
def NomDeMaFonction():
        """ Ceci est une fonction vide sans argument.
        """

        # Bloc d'instructions
```

— Que doit-elle renvoyer ? Toujours avant de commencer à remplir le bloc d'instructions, on peut commencer à déclarer une variable qui sera modifiée puis renvoyée à la fin de la fonction par un classique `return`. En Python, il faut surtout penser à l'initialisation de cette variable en fonction de ce que l'on veut y mettre. Par exemple, si on veut une chaîne de caractères, on devrait commencer comme cela :

```
def NomDeMaFonction():
        """ Ceci est une fonction vide sans argument.
        Elle retourne une chaîne de caractères."""

        result = ""

        # Bloc d'instructions

        return result
```

Remarque : Naturellement, une fonction Python n'a aucune obligation de renvoyer un objet.

— Exercice : Reprenez le code de l'exercice de rappels et transformez-le en une fonction, qui prendra comme argument un nombre de points à générer aléatoirement et un symbole pour représenter les points. Elle retournera une chaîne de caractères semblable à celle générée précédemment.

Correction :

```
def ViewGenerator(nbPoints, symb):
    """ Gererate a string who represents a cartesian plan,
        with nbPoints points represented by a symbole symb.
        Limitation to [0,9] in both axes"""

    result = ""

    # Generate a list of nbPoints:
    l_points = list()
    for i in range(nbPoints):
        l_points.append([randint(0,9),randint(0,9)])
    print l_points

    # Generate the representation string:
    for i in range(10):
        for j in range(10):
            x=j #Pour revenir en cartésien
            y=9-i #idem
            if [x,y] in l_points:
                if j==0 and i!=9: #On pourrait tester sur x et y aussi
                    result+="\n"+str(y)+"|"+symb+" "
                elif j==0 and i==9:
                    result+="\n"+str(y)+"|"+symb+"_"
                elif j!=0 and i==9:
                    result+=""+symb+"_"
                else:
                    result+=""+symb+" "
            else:
                if j==0 and i!=9:
                    result+="\n"+str(y)+"|  "
                elif j==0 and i==9:
                    result+="\n"+str(y)+"| _"
                elif j!=0 and i==9:
                    result+=" _"
                else:
                    result+="  "
    result+="\n   0 1 2 3 4 5 6 7 8 9"

    return result

print ViewGenerator(20, "0")
```

Commentaires :

Amorcer le fait qu'il y a plein de problèmes dans cette fonction :
- Si symb fait plus d'un caractère -> décalage
- Si 2 points identiques -> se superposent et on ne les voit pas
- Si aucune message via print -> que se passe-t-il ?
- Si nombre de points énorme ?
- ...

---

5 La gestion d'erreur

Nous définirons ici la gestion des erreurs comme étant un moyen pour empêcher les utilisateurs de faire n'importe quoi avec votre programme, et bien entendu de les informer un minimum sur ce qu'ils doivent faire. Il y a des similitudes évidentes entre gestion des erreurs et tests (que nous verrons plus loin), mais à la différence des tests, la gestion des erreurs doit bien prendre en compte qu'elle s'adresse à des utilisateurs et non à des développeurs (bien qu'en pratique, les indications vous aideront aussi à déboguer).

— La fonction `raw_input` : Cette fonction permet de faire interagir votre programme avec son utilisateur. Essayez le code suivant :

```
userAge = raw_input("Saisissez votre âge : ")
print "Vous avez " + userAge + "ans !"
```

Lorsque l'interpréteur arrivera sur la ligne contenant la fonction raw_input, il va afficher le message entre parenthèses puis se mettre en pause tant que l'utilisateur n'aura pas rentré une valeur dans son terminal. Une fois cela fait, il placera la valeur saisie dans une variable de votre choix au format string. Cette fonction est l'occasion de mettre en relief une règle implicite de la programmation : toujours vérifier la validité des informations envoyées à votre programme par un utilisateur ! Ce n'est pas parce que vous savez comment utilisez votre programme que tout le monde le saura (et les comportements d'utilisateurs sont souvent imprévisibles !).

— Exercice 1 : Créez une nouvelle fonction qui, lorsqu'elle est exécutée dans un programme, permet à un utilisateur d'afficher autant de représentations de points dans un plan cartésien (cf. exercice précédent) qu'il le souhaite, en choisissant lui-même pour chacune de ces représentations le nombre de points à représenter ainsi que le symbole à utiliser.

Correction :

```
def userGeneratorPlan2D():
    """ Enable to an user to gererate a string who represents a cartesian plan,
    with number points of his choice, even the symbole of points."""

    print("Ce programme vous permet de générer des représentations de points dans un
        plan.")
    print("Vous pourrez choisir le nombre de points à générer ainsi que le symbole
        graphique de ceux-ci.")

    condArret = False
    num = 0
    while(condArret == False):
        num+=1
        print("\nGénération du graphe nÂ°"+str(num)+" :")
        nbPoints = int(raw_input("Combien de points voulez-vous générer ? (limité à
            500)"))
        symbole = raw_input("Quel symbole d'un unique caractère voulez-vous utiliser
            ?")
        view = ViewGenerator(nbPoints, symbole)
        print view

        condArretText = raw_input("Voulez-continuer ? (y or n)")
        if condArretText == "n":
            condArret = True

    print("Merci d'avoir utilisé ce programme !")

userGeneratorPlan2D()
```

Commentaires :
En profiter pour parler de la modularité. Découpez un programme en un ensemble de fonctions cohérente est important (réductionnisme, clarté, maintenabilité, ...).

— Exercice 2 : Modifiez la fonction précédente pour gérez de possibles erreurs que pourrait induire un utilisateur (et par "gérer", on entend aussi l'informer de ce qui se passe!). Conseil : regardez du côté de la fonction `isinstance()` et de l'instruction `continue`...

Correction : *Il suffit de remplacer une partie du code précédent par :*

```
nbPoints = int(raw_input("Combien de points voulez-vous générer ? (limité à 500)"))
        if isinstance(nbPoints, int):
            if nbPoints > 500:
                print("Vous avez rentré " + str(nbPoints) + "points, or le programme
                    est limité à 500.")
                num-=1
                print("Merci de rentrer de nouvelles informations")
                continue
        else:
            print("Il faut rentrer un nombre entier")
            num-=1
            print("Merci de rentrer de nouvelles informations")
            continue

        symbole = raw_input("Quel symbole d'un unique caractère voulez-vous utiliser
            ?")
        if len(symbole) != 1:
            print("Vous avez rentré plus d'un caractère. Un seul demandé...")
            num-=1
            print("Merci de rentrer de nouvelles informations")
            continue
```

— Certaines opérations sont plus compliquées à vérifier (par exemple : trop de cas à vérifier, cas inconnus, etc.) et risquent pourtant de faire stopper votre programme. Pour cela, Python, comme dans beaucoup d'autres langages, a à disposition l'instruction `try`. Il existe de nombreuses possibilités d'utilisation de cette instruction (en savoir plus), mais classiquement, voici sa forme basique :

```
try:
    # Bloc à risque
except:
    # Bloc qui sera exécuté en cas d'erreur
```

L'avantage de try, c'est qu'il permet de gérer un minimum une zone critique d'un programme, sans pour autant connaître à l'avance l'erreur et sans stopper l'exécution. Où pourrait-on placer ces instructions dans le code précédent d'après vous ?

Correction :

```
        try:
                nbPoints = int(raw_input("Combien de points voulez-vous générer ? (
                    limité à 500)"))
        except:
                print("Erreur : un nombre entier est demandé ici...")
                num-=1
                print("Merci de rentrer de nouvelles informations")
                continue
```

Commentaires :
Pour ne pas perdre de temps, pas d'exo spécifique ici. Le try ne s'applique en effet pas vraiment au programme précédent. Le seul endroit où on pourrait l'utiliser, c'est pour tester le caste du raw_input pour le nombre de points, qui est le seul endroit qui pourrait générer une erreur a priori...

---

6 Les tests (en particulier unitaires)

Un test unitaire est un test qui permet de vérifier le bon fonctionnement d'une fonction de votre programme : c'est à destination des développeurs. À chaque modification d'un programme, les tests unitaires associés permettent en effet de détecter de possibles problèmes.

— Les assertions : Comme en C, il existe une fonction `assert(conditionLogique)` en Python. Lors de l'exécution d'une assertion, si la valeur retournée par la condition testée est `False`, alors le programme s'arrêtera et vous signalera l'échec de votre test. Vous pouvez testé le code suivant pour comprendre le fonctionnement des assertions :

```
from math import pow
assert(pow(2,2)==4)
assert(pow(0,2)==0)
assert(pow(1,1)==10)
assert(pow(1,10)==1)
```

(en savoir plus)

— Le meilleur moyen d'utiliser des assertions, est d'en écrire plusieurs dans un script dédié qui sera exécuté avant toute validation de votre programme suite à des modifications. La grande question restante étant : comment choisir ce qui sera testé ? En effet en pratique, on ne peut naturellement pas tester toutes les possibilités. Il est alors conseillé de tester des valeurs aux limites, ainsi que quelques valeurs quelconques pour lesquelles on connaît également le résultat. Cela n'a pas vocation à être exhaustif. Mais en faire un minimum est primordial pour vérifier que son programme fait bien ce qu'on attend de lui et, dans le cas contraire, pour cibler rapidement le problème.

— Inconvénient des assertions : Le problème avec les assertions, c'est que votre programme de test s'arrêtera dès le premier assert trouvé avec une condition fausse. Si vous pensez que cela à son importance, vous pouvez toujours faire des tests en utilisant simplement des `if` à la place des `assert`. Cela règlera le problème cité et vous permettra de traiter plus finement vos tests. A l'inverse, cela nécessite d'écrire plus de code...

— Des tests pour mieux comprendre : Laisser des scénarios de test dans son programme peut également permettre à un autre développeur de mieux l'appréhender. Et ici, par test, on entend plutôt des démonstrations du fonctionnement de fonctions. Mais voilà, on ne veut pas que ces tests/démos se lancent lors de l'exécution principal du programme. Or, contrairement au C, Python n'a pas de fonction ou méthode main(). Quand on lance un script, tout le script est exécuté dans tous les cas. Cela pose donc un problème quand on a un script qui contient du code que l'on souhaite exécuter quand on lance le script directement, mais pas quand on l'importe dans un autre script. Pour contourner ce problème, il existe une instruction bien pratique :

```
if __name__ == '__main__':
        # code à exécuter si on exécute directement le script qui le contient
```

(en savoir plus)

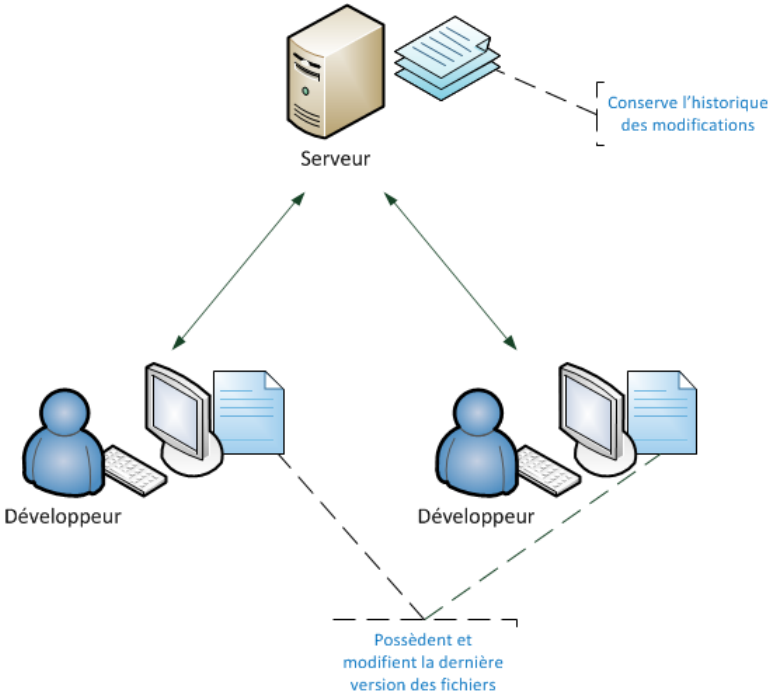
— Conclusion : A minima, un programme Python devrait contenir 3 scripts distincts : le script principal, le script de définition des fonctions et le script de test. Chacun de ces scripts doit respecter les consignes que nous avons vues aujourd'hui et durant la séance précédente. Et dans celui qui contient des fonctions, il doit également y avoir l'instruction vu précédemment pour pouvoir tester le comportement des fonctions qui s'y trouvent. Enfin, n'oubliez pas d'inclure un petit fichier texte README pour donner un minimum d'information sur votre programme (version d'interpréteur Python utilisée, OS testé, description du programme, auteur, etc.) !

7 Guide d'installation de Git/GitHub

Git est un gestionnaire de version (si ce n'est LE gestionnaire de version !), c'est-à-dire un logiciel qui permet de stocker des fichiers tout en conservant toutes les informations des modifications antérieures. C'est un outils basique pour tous les projets de développement informatique.

GitHub est un service web d'hébergement qui vient compléter votre gestion de versions en utilisant Git. Il offre en plus d'un espace de stockage de nombreuses fonctionnalités très utiles à la gestion de développement de logiciels (interface web, forum, wiki, bug-tracking, etc.).

Pour plus d'information, n'hésitez pas à référer au tuto suivant : https://openclassrooms.com/courses/gerer-son-code-avec-git-et-github/



source : https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git#/id/r-1234353

— Créez-vous un compte sur GitHub.

— Tapez simplement la commande `git` dans un terminal (quelque soit votre OS). Si vous voyez apparaître des informations en anglais sur les options de Git, c'est que Git est déjà installé sur votre ordinateur. Si ce n'est pas le cas, installez Git en suivant les démarches suivantes : https://openclassrooms.com/courses/gerer-son-code-avec-git-et-github/installer-git#/id/r-2448449

— Allez sur la page dédié à notre UE : https://github.com/M1-BIBS-Python, puis cliquez sur le "Repository" `Documents`. Vous devriez voir un icône `Clone or Download` en vert à droite de votre fenêtre. Cliquez dessus et copiez l'adresse `HTTPS`. A présent, créez un dossier dans votre dossier dédié à cette UE (exemple : GitRepo). Via le terminal, allez dans ce dossier, et tapez la commande suivante : `git clone lienHttpsFourniParGitHub`. Vous devriez avoir le dossier `Documents` ainsi que tous les fichiers/dossiers qu'il contient téléchargés en local sur votre ordinateur.

— Une 1ère commande `git add` : Dans votre dossier `Documents/testCommit/` local, créez un script Python contenant une fonction quelconque et respectant les conventions que nous avons vues. Donnez-lui un nom du type : VotrePrenomVotreNomTestCommit.py. Dans votre terminal, tapez `git status`. Vous devriez avoir un message vous indiquant que votre répertoire local contient des fichiers non suivis. En effet, pour ajouter un nouveau fichier à gérer dans votre projet, il va falloir l'indiquer à Git en faisant un : `git add VotreFichier.py`.

— Mon premier `commit` ! `commit` est une opération permettant de faire une sauvegarde EN LOCAL d'une version du code que vous avez en local également (et qui est bien identifiée par Git). Refaites un `git status` et vous devriez avoir un message vous disant qu'un nouveau fichier est prêt à être "commité". Il ne reste donc plus qu'à sauvegarder votre version en faisant :

`git commit -m "Un message informatif et succinct sur les modifications que vous avez effectué sur les fichiers que vous allez envoyer sur le serveur distant"`. Si vous refaites à nouveau un `git status`, vous verrez que vous n'avez plus rien à commiter. Remarque : on ne commite jamais une version non fonctionnelle ! (qui ne s'exécute même pas par exemple)

— Durant un développement, on peut récupérer toutes les versions des fichiers sur le serveur distant qui ont été sauvegardés à un instant t par un commit. Mais pour cela, il va falloir envoyer ses sauvegardes issues de vos commit sur le serveur distant. En effet, si vous allez voir sur le dossier concerné sur GitHub, vous ne devriez pas voir votre fichier dessus. Rien de plus simple (mais aussi dangereux) pour solutionner ça, faites juste : `git push origin master`. A présent, si vous retournez sur GitHub, votre fichier a dû apparaître (NB : si j'ai eu le temps de vous autoriser à écrire sur le répertoire naturellement...). Je disais "dangereux", car c'est à présent LA version courante sur le serveur distant, celle que tous vos collaborateurs téléchargeront. N'oubliez donc pas : Pour un commit, il faut déjà une version fonctionnelle, mais c'est encore plus important pour un push !

Remarque : Il ne faut pas faire de commit comme on fait un ctrl+s. Un commit permet de valider une avancée dans votre projet (régler un bug, ajouter une fonctionnalité, etc.). La taille des modifications d'un commit peut donc être très variable. L'idée est de valider chaque petite étape de progression de votre projet, donc minimum une fois par jour. Le push peut être beaucoup plus rare, une fois maximum par jour grosso modo.

— Commande `git pull` : Enfin, on voudrait à présent récupérer en local la version complète stockée sur le serveur distant (qui devrait contenir à présent les fichiers de chacun d'entre vous). Pour cela, il faut faire : `git pull`. Vous devriez avoir à présent tous les fichiers de vos camarades en local. Si des modifications sont pushées sur le serveur depuis votre dernier pull par quelqu'un d'autre, et que vous voulez pusher à votre tour vos modifications, vous obtiendrez un message d'erreur vous expliquant cette situation. Il suffit de faire à nouveau un `git pull`, et ne vous inquiétez pas : celui-ci ne devrait pas impacter vos modifications (il est tout de même conseillé de ne pas travailler sur les mêmes fichiers, en tout cas, pas au même endroit !), il mettra juste à jour vos propres fichiers sans écraser ceux modifiés par vos soins. Une fois cela fait, refaite un push et cela devrait fonctionner.

— Remarque 1 : Lors du `git clone`, notez que vous pouvez cloner votre repo avec deux options :

- L'option HTTPS qui est l'option la plus simple, qui demande de fournir vos identifiants GitHub (nom d'utilisateur et mot de passe) à chaque fois que vous faites un git clone.

- L'option SSH qui est plus pratique car elle ne vous demande pas vos identifiants à chaque fois. Par contre, pour l'utiliser, il faut générer une clé SSH, ce qui est un peu plus compliqué... Si cela vous intéresse, il y a un tuto GitHub ici.

— Remarque 2 : Si les lignes de commandes ne vous paraissent pas attrayantes, il existe des interfaces graphiques pour faire les mêmes opérations. Sous Windows par exemple, livré avec l'installation de Git, il suffit de faire clique-droit dans un dossier, et vous devriez voir une option "Git GUI".