Fichier read-me
Script Python de suppression automatique des fichiers temporaires

Notice rédigée par: Conor INNES.

Sommaire

I. Exécution du script

Ce script est codé en Python. Pour pouvoir l'exécuter, vous allez avoir besoin de l'IDLE Shell de Python (téléchargeable sur www.python.org). Vous pouvez également l'exécuter avec tout autre IDE capable d'exécuter des scripts Python, comme pyCharm (www.jetbrains.com/pycharm). Pour télécharger le programme de suppression des fichiers temporaires lui-même, rendez-vous sur la page GitHub du projet à l'adresse: https://github.com/ci22/Suppression-Fichiers-Temporaires .

Différentes versions de l'IDLE Shell de Python sont comptaibles avec différents systèmes d'exploitation. Cependant, ce script récupère les informations du runtime de Windows, il n'est donc conçu pour fonctionner que sous Microsoft Windows (il n'est pas comptabile avec MacOS ou Linux).

En outre, afin que ce script puisse fonctionner correctement, le module *Paramiko* doit-être installé sur le logiciel que vous utilisez pour exécuter ce programme (en effet, ce module est nécessaire pour l'exécution du script en mode réseau). Si vous souhaitez exécuter le script avec l'IDLE Shell, avant d'ouvrir l'IDLE, ouvrez une fenêtre de la ligne de commande de Windows. Naviguez ensuite vers le dossier où se trouve votre installation du logiciel Python, afin que nous puissions utiliser *Pip* (l'outil qui va nous permettre de télécharger Paramiko). Par exemple, si vous disposez de Python 39, saisissez *cd C:\Users\Utilisateur\AppData\Local\Programs\Python\Python39\scripts* pour changer de répertoire. Saisissez ensuite *pip install paramiko* pour télécharger les paquets nécessaires. Une fois l'installation terminée, vous pouvez fermer la ligne de commande.

Une fois le script (fichier *SuppressionFichiersTemporaires.py*) ouvert dans l'éditeur de code, cliquez sur «Run» puis sur «Run Module» pour l'exécuter avec l'IDLE Shell. Le programme va alors se lancer, et va proposer deux modes: le mode «local» et le mode «réseau». Si vous sélectionnez le mode local, le script va s'exécuter sur le poste que vous utilisez, et va purger le dossier C:\Windows\Temp pour supprimer l'ensemble des fichiers temporaires qu'il contient. Dans le contexte où vous sélectionnerez le mode réseau, le script va d'abord scanner l'ensemble des machines présentes sur un réseau local en pinguant un range d'adresses, et va ensuite confirmer pour chaque adresse si la machine concernée est joignable ou non. Le script demandera par la suite l'adresse de la machine cliente sur lequel vous souhaitez exécuter le script à travers le réseau, et purgera alors le répertoire contenant les fichiers temporaires de la même façon que pour le poste local.

Attention toutefois, puisque le pare-feu Windows peut être ammené à bloquer les communications de Paramiko pour l'éxécution du script à distance. Veillez donc à créer une règle autorisant les communications sur le port 22 à la fois sur le client et sur le serveur (voire à désactiver temporairement le pare-feu sur les deux machines).

Assurez-vous également que le service *OpenSSH Server* est bien activé sur le poste distant que vous souhaitez joindre.

II. Fonctionalités fournies

Tout d'abord, le script propose deux modes d'exécution: un mode d'exécution local qui permet de lancer le script sur le poste client sur lequel nous travaillons, et un mode d'exécution réseau qui permet de pousser le script à travers un réseau local pour l'exécuter à distance et sur plusieurs postes simultanément. Le mode réseau dispose lui-même d'un processus qui permet de scanner une plage d'adresses allant de 192.168.1.11 à 192.168.1.253. Même si cette plage est prérenseignée par défaut, il est possible de modifier manuellement cette partie du script pour l'adapter à la classe d'adressage de son réseau (A, B ou C), ou pour exclure la plage qui correspond aux serveurs, aux équipements d'interconnexion réseau, ou encore les téléphones IP par exemple (nous supposons que les serveurs occupent les machines de 1 à 11 et que la passerelle occupe l'adresse en 254). Ce scan automatique du réseau permet un repérage de toutes les machines allumées au moment où le script est lancé, et permet ainsi d'éviter tout message d'erreur dû à une éventuelle tentative d'exécution du script sur un poste client injoignable. Cette fonctionalité permet notamment à l'administrateur systèmes et réseaux de prendre note des machines sur lequel le script est exécuté, et lui permet ainsi de garder une trace des numéros des machines et de la date de leur dernier nettoyage périodique. Pour réaliser ce test, le script utilise le protocole ICMP (ping). Par ailleurs, le script affiche sur la console pendant le scan et pour chaque poste le résultat du test ping. En effet, en fonction de l'état de la réponse reçue par le poste qui pousse le script, trois cas de figure sont possibles: soit le poste est joignable, soit le poste n'a pas donné de réponse à la requête ICMP (la réponse du ping est peut-être bloquée par un pare-feu, par exemple), ou encore un message indique que le délai d'attente du ping est dépassé.

En outre, lorsque le script est exécuté en local sur le poste, le script récupère le *subprocess* (sous-processus) de l'environnement Windows. Le programme récupère et retourne ensuite, grâce au chemin d'accès de l'emplacement de stockage par défaut des fichiers temporaires de Windows, la taille totale de l'ensemble des fichiers stockés. Ceci permet à l'administrateur systèmes et réseaux d'avoir une vue globale sur l'espace de stockage libérée sur les postes des utilisateurs, une information particulièrement utile notamment si l'équipe technique a reçu des plaintes de la part des utilisateurs du système d'information liées à un ralentissement des postes clients. Lorsque le mode réseau est utilisé, le module Paramiko exécute à distance une commande sur l'invite de commande Windows qui permet la suppression de tous les fichiers temporaires du répertoire C:\Windows\Temp. Par ailleurs, lorsque le mode réseau est utilsé, en cas d'échec une sortie d'erreurs est effectuée sur la console pour informer l'administrateur d'un port SSH non ouvert, ou encore d'un défaut d'autentification par exemple.

De surcroît, l'extension de fichiers qui est renseigné dans le script est le .tmp, ce qui correspond à l'extension par défaut des fichiers temporaires de Windows. Ainsi, seuls les fichiers temporaires sont supprimés, évitant ainsi toute suppression accidentelle de fichiers importants (comme les fichiers de jounalisation, par exemple)

Pour finir, la console informe le technicien de l'état de suppression de chaque fichier, lorsque le script est exécuté en local: si la suppression a été réalisée avec succès, un message le confirme. Sinon, un message d'erreur indique la raison de l'échec de la suppression: manque de droits pour exécuter le programme, le chemin d'accès au fichier n'existe pas/n'existe plus, ou encore que le chemin d'accès n'est pas un répertoire ou un dossier.

III. Structuration du script

Dans cette partie du guide read-me, nous allons aborder le fonctionnement technique du script, en parcourant les lignes du code.

Tout d'abord, et dès le lancement du script, il commence par importer un certain nombre de processus. Ces processus, remontés grâce à la commande «import» permettent de récupérer diverses informations contenues dans le système d'exploitation du script.

```
import subprocess
import time
import paramiko
```

La variable d'environnement *subprocess* permet la récupération des processus interne du système d'exploitation, et il servira plus tard pour l'exécution de la commande ping. La commande *time* quand à elle, permet de récupérer les information liées au réglages horaire du poste, et sera utilisée pour le décompte en secondes des fonctions time.sleep. La fonction *Paramiko* est un équivalent de SSH en Python, et nous permettera de prendre la main à distance sur d'autres postes plus loin dans le script.

A l'étape d'après, le script indique que le programme est utilisable sur le poste local, et sur un autre poste client à travers le réseau. Le texte est donc affiché à l'aide d'une commande print, et la commande input qui suit permet le stockage du résultat saisi par l'utilisateur dans la variable c*hoixMode*.

```
print ("Ce programme est exécutable en local et en réseau")
choixMode = input("Pour exécution locale: saisir ''local'' - Pour exécution réseau: saisir ''reseau''")
```

Grâce à une commande *print* et au rappel de la variable *choixMode,* le programme confirme le choix de l'utilisateur:

```
print("Vous avez choisi le mode",choixMode)
```

Si l'utilisateur choisi le mode local et que ce terme figure dans la variable (une vérification avec la condition *if* est effectuée), l'utilisation de la commande *print* permet de confirmer que le répertoire C:\Windows\Temp va être chargé. Une commande *time.sleep* combiné à l'affichage de pointillés par le biais de la fonction *print* permet de montrer un processus de chargement du programme sur la console.

```
# Lecture du choix du mode d'exécution.
if choixMode == "local":
# Exécution poste local
    print("Chargement du répertoire C:\Windows\Temp")
    time.sleep(5)
    print(". . .")
```

La définition de la fonction *obtenirTaille* permet de récupérer puis d'afficher sur la console l'espace totale occupée en octets par les fichiers temporaires dans le dossier C:\Windows\Temp.

```
def fonctionPrincipale():
```

Nous donnons ensuite accès au système d'exploitation, afin que le script puisse récupérer les informations nécessaires et agir sur les fichiers. Nous enregistrons également le chemin d'accès où sont stockés les fichiers temporaires dans une variable.

```
import os
cheminAcces = 'C:\Windows\Temp'
fichierTemporaire = ".tmp"
```

Le script utilise ensuite un compteur qui permet de faire boucler la fonction 50 fois, afin que nous puissions nous assurer que tous les fichiers concernés soient supprimés.

```
for compteur in range(1, 50):
```

Deux conditions *if* sont ensuite utilisées pour vérifier que le chemin d'accès enregistré dans la variable est valide (ceci permet de vérifier que le chemin d'accès correspond bien à un répertoire et qu'il existe).

```
if os.path.exists(cheminAcces):
if os.path.isdir(cheminAcces):
```

Le programme parcourt alors le dossier racine, les sous-dossiers, et les fichiers en utilisant la fonctionalité *os.walk* et en passant la variable en paramètre.

```
for root_folder, folders, files in os.walk(cheminAcces):
```

Il parcourt ensuite les fichiers dans le dossier et les sous-dossiers.

```
for file in files:
```

On stocke ensuite le chemin d'accès du dossier racine et du fichier dans la variable *cheminFichier* à l'aide de la fonctionalité *os.path.join*.

```
cheminFichier = os.path.join(root_folder, file)
```

Le script va ensuite récupérer l'extension du fichier grâce à *os.path.splitext* avec la variable contenant le chemin du fichier passé en paramètre. On stocke ensuite le résultat dans la variable *extensionFichierTemp*.

```
extensionFichierTemp = os.path.splitext(cheminFichier)[1]
```

Une nouvelle comparaison est ensuite effectuée entre la variable stockant l'extension de fichier .tmp et le contenu de la variable *extensionFichierTemp*.

```
if fichierTemporaire == extensionFichierTemp:
```

Si la comparaison s'avère être juste, *os.remove* se charge de supprimer le fichier. Si la suppression fonctionne correctement, une sortie sur la console confirme que le fichier a été supprimé avec succès.

```
if not os.remove(cheminFichier):
    # Confirmation de la suppression
    print("Le fichier temporaire", cheminFichier, " a bien été supprimé")
```

En cas d'échec de la suppression, un message d'alerte s'affiche sur la console. L'utilisation de la variable dans la commande *print* permet de connaître précisément le fichier concerné. Le message suivant invite alors l'administrateur à vérifier qu'il dispose des droits nécessaires pour pouvoir supprimer le fichier (il est préférable d'exécuter le Shell de Python en mode administrateur).

```
else:
    # Message si erreur durant la tâche de suppression
```

```
        print("ERREUR! Le fichier temporaire", cheminFichier, " n'a pas pu être supprimé")
        # Le programme incite à vérifier que l'on est bien administrateur de son poste
        print("Veuillez vérifier que vous disposez des droits nécessaires pour exécuter ce programme, puis réessayez")
```

Les lignes suivantes correspondent à la partie *else* (sinon) des deux premières conditions *if*. Elles permettent respectivement de lister les fichiers et les dossiers qui ne sont pas des fichiers temporaires, et le cas échéant de préciser que le fichier en cours de suppression n'existe pas (dans le cas où il aura été supprimé ou renommé depuis le scan initial).

```
else:
        print("Le chemin d'accès - ", cheminFichier," n'est pas un répertoire/dossier ou un fichier temporaire")
# Si le test échoue, le chemin d'accès n'existe pas
else:
        print("Le chemin d'accès spécifé - ", cheminFichier, " - n'existe pas")
```

La partie suivante concerne l'exécution du script à travers le réseau, si l'utilisateur a choisi ce mode. La fonction *if* permet au programme de voir si l'utilisateur a choisi ce mode, en comparant le contenu de la variable *choixMode* avec le mot «reseau».

```
if choixMode=="reseau":
        print ("Scan du réseau local en cours...")
        print ("Cette opération peut prendre plusieurs minutes")
        time.sleep(3)
        print (". . .")
        time.sleep(3)
        print (". . .")
```

Le programme indique alors que le mode réseau va lancer un scan des postes présents sur l'infrastructure, et invite alors l'utilisateur à patienter pendant quelques instants. L'utilisation de la commande «time.sleep» associé à un temps d'attente de trois secondes, combinés à l'impression sur la console de pointillés permettent à l'administrateur réseau exécutant le programme de savoir que le programme est bien en cours d'exécution.

La fonction de scan du réseau local s'exécute ensuite. Elle utilise le protocole ICMP (ping) pour vérifier si la machine est en ligne ou non sur le réseau.

Par défaut, la variable *requetePing* parcourt un champ d'adresses réseau allant de 192.168.1.11 à 192.168.1.253. En effet, nous partons du principe que les machines présentes sur les dix premières addresses du réseau correspondent aux équipements réseau et aux serveurs, mais l'administrateur utilisant le script peut modifier cette partie pour l'adapter aux spécificités de son réseau.

```
 for requetePing in range(11,253):
            adresseIP = "192.168.1." + str(requetePing)
```

Cette partie du programme fonctionne en stockant le champ d'adresses allant de 11 à 253 dans la variable *requetePing*, puis en stockant dans la variable *adresseIP* les trois premiers octets du réseau, suivis par le contenu de la variable *requetePing* passée en chaîne de caractères.

La ligne suivante consiste à faire exécuter la requête ICMP en subprocess. L'exécution du subprocess stocke le résultat du ping dans une variable «reponse».

```
for Ping in range(11,253):
            adresseIP = "192.168.1." + str(Ping)
            reponse = subprocess.run(['Ping', '-n', '3', adresseIP],
shell=True)
```

Lors de l'exécution du programme, le résultat du ping pour chaque adresse est alors affichée sur la console.

Une fois le scan du réseau effectué, le programme va tenter d'ouvrir une connexion SSH avec un poste client distant avec Paramiko. Le script commence donc par déclarer une variable globale dans laquelle nous allons stocker l'adresse IP du poste à joindre, puis invite l'utilisateur du programme à saisir l'adresse.

```
global IPHote
        IPHote = input("Entrez l'adresse IP du poste sur lequel vous
voulez exécuter le script - ou annuler pour quitter")
```

Comme vous pouvez le constater, le script donne également l'option de quitter l'exécution du programme si nous ne souhaitons pas accéder à un poste à distance. La commande suivante va donc comparer la valeur de *IPHote* avec le mot «annuler», et si les deux valeurs correspondent, alors on appelle la commande *exit()* qui permet de fermer le shell, et ainsi d'arrêter l'éxécution du script. Dans le cas où l'utilisateur ne saisirai rien, le programme va de nouveau demander la saisie de l'adresse IP.

```
if IPHote == "annuler":
                exit()
        else:
                IPHote = input("Entrez l'adresse IP du poste sur lequel vous voulez exécuter le script - ou annuler pour quitter")
```

Une fonction est ensuite défine, elle permet l'ouverture de la connexion SSH à partir des informations saisies (la fonction demandera l'entrée de d'autres informations sur la console plus tard).

```
def fonctionSSH ():
```

Pour des raisons pûrement techniques, nous devons commencer par déclarer une variable qui s'appelle *NomPoste,* mais nous n'allons pas lui assigner de valeur pour l'instant. Ceci permettera d'éviter des dysfonctionnements du programme plus tard, car en effet le programme peut tenter d'utiliser une variable qui n'a pas été déclarée dans le script.

```
NomPoste = None
```

Nous allons également assigner la valeur *True* à la variable *fonctionSSH*.

```
fonctionSSH = True
```

Nous allons maintenant utiliser une condition *while*. On commencera par assigner une chaîne de caractères vide à la variable *nomUtilisateur*. De ce fait, nous allons pouvoir utiliser une autre boucle *while* pour faire en sorte que la console demande à saisir un nom d'utilisateur, jusqu'à ce que l'utilisateur en saisisse une.

```
while fonctionSSH:
# La fonction va boucler jusqu'à ce que l'on saisisse un nom d'utilisateur
       nomUtilisateur = ""
       while nomUtilisateur == "":
            nomUtilisateur = input("Saisissez le nom d'utilisateur du poste distant, ou annuler pour quitter".format(NomPoste))
```

De la même façon que pour l'adresse IP, nous donnons à l'utilisateur du programme l'option de quitter l'exécution, en comparant la variable avec le terme «annuler».

```
if nomUtilisateur == "annuler":
                    exit()
```

On effectue ce même type de comparaison pour la saisie du mot de passe, avec également l'option de quitter le programme une nouvelle fois (en cas de faute de frappe, par exemple). Ici, on utilise la variable *motDePasse*. Dans les deux cas, nous passons la variable *NomPoste* en paramètre.

```
motDePasse = ""
while motDePasse == "":
        motDePasse = input("Saisissez le mot de passe du poste
```

```
distant, ou annuler pour quitter".format(NomPoste))
        if motDePasse == "annuler":
                exit()
```

Une condition try est ensuite utilisée pour tenter l'ouverture d'une connexion SSH.

```
try:
```

Dans la variable *ssh*, nous allons stocker les informations à propos du client.

```
ssh = paramiko.SSHClient()
```

Parfois, il arrive qu'il manque une clé publique pour ouvrir une connexion SSH/Paramiko, ou qu'elle soit inconnue. Nous allons donc créer une politique qui gère les clés manquantes.

```
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

Paramiko se charge alors de démarrer la connexion SSH. Avant cela, une commande *print* averti l'utilisateur sur la console qu'une connexion SSH est en cours. Nous stockons également dans la variable *commandeDistante* la commande qui sera exécutée sur le poste distant. En effet, les commandes envoyées à travers Paramiko dans Windows sont exécutés dans la ligne de commande Windows. Ici, l'usage de la double esperluette (sigle du «et commercial») permet de faire passer deux commandes successives dans Paramiko. (Ceci indique à Paramiko qu'il doit traîter ces commandes successivement et sans ouvrir une nouvelle invite de commande à chaque fois).

```
commandeDistante = "cd C:\Windows\Temp && del *.tmp"
```

La commande *ssh.connect* est utilisée pour ouvrir la connexion SSH, avec les variables *IPHote*, *nomUtilisateur*, et *motDePasse* passées en paramètre. On stocke également les valeurs de *nomUtilisateur* et de *motDePasse* dans *username* et *password* pour que ces informations puissent être utilisés par Paramiko pour ouvrir la connexion.

```
print("Connexion SSH en cours...")
# Ouvrir la connexion avec le nom d'utilisateur et le mot de passe du poste distant
ssh.connect(IPHote, username=nomUtilisateur, password=motDePasse)
```

Une fois la connexion établie, nous demandons une saisie sur la console du poste distant, avec l'usage notamment de la fonction *exec_command*.

```
stdin, stdout, stderr = ssh.exec_command(commandeDistante)
```

Nous demandons ensuite une confirmation de l'exécution de la tâche, et l'affichage sur la console des informations qui proviennent de l'ordinateur distant.

```
print(stdout.read().decode())
print("Exécution de la commande distante en cours (via SSH)")
```

Nous utilisons ensuite la condition *False* que nous associons à la variable *fonctionSSH* pour fermer la connexion si nécessaire.

```
fonctionSSH = False
```

Il en va de même pour la condition *if* qui est utilisée pour fermer la connexion.

```
if ssh: ssh.close()
```

Les lignes suivantes permettent d'afficher les erreurs sur la console, afin d'informer

l'administrateur réseau de la clôture d'une connexion ou d'un défaut d'authentification (similaire à l'affichage des erreurs *Last Exit Code* dans Windows PowerShell).
NoValidConnectionsError est une fonctionalité qui nous permet d'être averti lorsque la connexion SSH a été interrompue (perte de connexion réseau, par exemple), alors que *AuthenticationException* nous indiquera que l'authentification de l'utilisateur SSH a été refusée, en cas d'erreur de saisie du nom d'utilisateur ou du mot de passe du compte utilisateur SSH. Les erreurs éventuels sont donc enregistrés dans une variable *erreurSSH*, puis affichés sur la console avec un message générique.

```
# Erreur de fermeture de connexion
    except      paramiko.ssh_exception.NoValidConnectionsError     as
erreurSSH:
                print("ERREUR - Connexion fermée! Voici le code
erreur rencontré:")
                print(erreurSSH)

# Erreur dûe à un défaut d'authentification
   except paramiko.ssh_exception.AuthenticationException as erreurSSH:
                print("ERREUR - Défaut d'authentification! Voici le
code erreur rencontré:")
                print(erreurSSH)
                Pass
```

Les trois dernières lignes du script concernent l'appel des fonctions. Les fonctions sont appelées ici afin de permettre leur exécution.
L'appel de la fonction d'ouverture de connexion SSH est réalisée à l'intérieur de la «boucle» de la fonction (l'indentation du code fait en sorte que `def fonctionSSH ():` et `fonctionSSH()` sont parfaitement alignés.

On appelle également les fonctions qui permettent d'obtenir l'espace de stockage occupé par les fichiers temporaires et de faire fonctionner le programme en mode local. L'appel de ces deux fonctions se fait à la ligne et sans laisser d'indentation supplémentaire, afin que l'appel de la fonction soit alignée avec la déclaration des fonctions `def obtenirTaille():` et `def fonctionPrincipale():` au début du script.

IV. Contributions et informations de contact

Il est possible de télécharger le script directement sur la page dédiée au projet sur GitHub. Lien direct vers la page du projet: https://github.com/ci22/Suppression-Fichiers-Temporaires

Ce script est publié sous la licence GPL. Vous êtes donc libre de le télécharger, de l'utiliser, et de le redistribuer. Vous êtes également libre d'effectuer des modifications ou d'adapter le programme et de redistribuer le code source à la communauté de développeurs, sur GitHub ou sur d'autres sites de partage de projets libres. Cependant, compte-tenu de la licence attribuée au programme, le développeur ne peut-être tenu responsable d'un mauvais fonctionnement du script ou de tout préjudice qui pourrait découler de son utilisation, et aucune garantie n'est fournie avec le script.

Pour obtenir des informations à propos du programme, pour partager vos idées, ou pour faire part de tout commentaire, vous pouvez envoyer un mail à l'adresse: conorinnes@gmail.com. Cette adresse mail vous permet notamment de signaler une erreur ou un bug dans le script. Vous pouvez également vous tenir informé de toute modification du script ou des fichiers qui y sont rattachés et des éventuels bugs retrouvés dans le programme directement sur la page GitHub du projet.