Initiation à la programmation

SL25Y031

Cours/TD – Chapitre 8

Université Paris Cité

Objectifs :

- Tester l'existence d'un fichier ou dossier.
- Lister le contenu d'un dossier.
- Lire un fichier texte.
- Écrire dans un fichier texte.

1 Système de fichiers

Le système de fichiers [Cours]

- Les données enregistrées sur un support physique (ex : un disque dur, une clef usb) sont généralement représentées sous forme d'une structure constituée de *dossiers* (ou « répertoires ») et de *fichiers*.
- Un fichier peut contenir tout type de données alors que les dossiers ne servent a priori qu'à l'organisation des fichiers. Un dossier peut être vu comme une liste (éventuellement vide) de fichiers et de dossiers formant son *contenu*. Excepté un dossier particulier, la *racine du système de fichiers*, tout dossier et tout fichier est *contenu* dans un unique dossier, son *dossier parent*.
- Les *descendants* d'un dossier sont (par récurrence) :
  - tout fichier ou dossier qu'il contient ;
  - tout descendant d'un dossier qu'il contient ;
- La structure obtenue est *arborée* :
  - elle ne contient pas de boucle, dans le sens où un dossier n'est jamais l'un de ses propres descendants ;
  - il existe un dossier (la racine), dont tous les autres éléments descendent.
- Tout dossier et tout fichier possède un *nom*, représenté en Python par une chaîne de caractères (ex : `"documents"`, `"exam.pdf"`). Le nom de la racine correspond généralement à la chaîne vide et tous les autres éléments du système de fichiers ont des noms non vides. Les éléments contenus dans un même dossier doivent avoir des noms tous distincts.
- Il est courant pour un processus interagissant avec le système de fichiers (ex : un terminal, un navigateur de fichiers) de donner (temporairement ou non) un statut particulier à un dossier, alors appelé « répertoire de travail actuel ». Lorsque Python exécute du code, un répertoire de travail actuel est défini. Le choix initial de ce dossier dépend de la manière dont le code a été exécuté.

Chemins [Cours]

- Un *chemin* est une chaîne de caractères non vide désignant (ou « pointant vers ») un élément existant ou fictif dans un système de fichiers. Un chemin désignant un fichier existant peut par exemple servir à lire ce fichier. Un chemin désignant un élement fictif peut par exemple servir à créer un élement à cet emplacement.
- Pour interpréter un chemin, il est nécessaire de définir trois valeurs de type `str` (qui peuvent varier d'un système à un autre) :
  - le *séparateur* (`"/"` dans les exemples qui suivent) ;

  - l'*abréviation de répertoire courant* (`"."` dans les exemples qui suivent) ;
  - l'*abréviation de répertoire parent* (`".."` dans les exemples qui suivent).

  Pour interpréter certains chemins, dit « relatifs », il est aussi nécessaire d'avoir une notion de répertoire de référence. C'est généralement le répertoire de travail actuel qui va servir de référence pour l'interprétation des chemins relatifs en PYTHON.
- Quelques exemples :
  - le chemin `"/home/guest/documents"` désigne un élément `"documents"` situé dans le dossier `"guest"` situé dans le dossier `"home"` de la racine ;
  - le chemin `"./../../exam.pdf"` désigne l'élément `exam.pdf` situé dans le dossier parent du dossier parent du répertoire de référence ;
  - le chemin `"../../exam.pdf"` désigne exactement le même élément que le chemin précédent ;
  - le chemin `"/"` désigne la racine du système de fichiers.
- Formellement, pour interpréter un chemin, il faut le décomposer en fonction des occurrences du séparateur. Nous reviendrons plus bas sur la méthode `split`, mais `path.split(sep)` est une liste de chaînes de caractères : la liste des chaînes obtenues en découpant `path` au niveau de chaque occurrence de `sep`. Par exemple, `"./../../exam.pdf".split("/")` vaut `[".", "..", "..", "exam.pdf"]`, `"/home/guest/documents".split("/")` vaut `["", "home", "guest", "documents"]` et `"/".split("/")` vaut `["", ""]`.
- On peut définir l'élément désigné par un chemin `path` étant donné le séparateur `sep` par récurrence sur la liste `path.split(sep)` :
  - cas de base :
    - `[""]` désigne la racine,
    - `["."]` désigne le répertoire de référence,
    - `[".."]` désigne le parent du répertoire de référence ou le répertoire de référence s'il s'agit de la racine,
    - pour tout autre chaîne de caractères `s` ne contenant pas `sep`, `[s]` désigne l'élément de nom `s` contenu dans le répertoire de référence ;
  - si la liste `l` désigne un dossier `d`, alors :
    - `l + [""]` désigne encore `d`,
    - `l + ["."]` désigne encore `d`,
    - `l + [".."]` désigne le parent de `d` ou `d` s'il s'agit de la racine,
    - pour tout autre chaîne de caractères `s` ne contenant pas `sep`, `l + [s]` désigne l'élément de nom `s` contenu dans `d`.
- Les chemins commençant directement par le séparateur (ex : `"/home/guest/documents"`) sont *absolus* : leur interprétation ne dépend pas du répertoire de référence. Les autres chemins sont les chemins relatifs (ex : `"../../exam.pdf"`), dont l'interprétation dépend du répertoire de référence.

2 Lecture du système de fichiers

Lecture de l'arborescence de fichiers [COURS]

- Un grand nombre de fonctions et autres valeurs utiles pour la manipulation du système de fichiers sont disponibles en PYTHON dans le module (≈ la bibliothèque) `os`. On peut accéder à ces fonctions et valeurs après avoir exécuté l'instruction suivante :

```
1 import os
```

- Il est possible de connaître à tout moment le chemin absolu du répertoire de travail actuel (sous forme d'une chaîne de caractères) en évaluant l'expression `os.getcwd()` :

```
1 print(os.getcwd()) # Affiche "/home/guest".
```

- Le nom de la fonction `getcwd` est une abréviation de « *get current working directory* ».
- La valeur du séparateur est assignée à la variable `os.sep`. La valeur de l'abréviation de répertoire courant est assignée à la variable `os.curdir`. La valeur de l'abréviation de répertoire parent est

assignée à la variable `os.pardir`.

```
print(os.curdir) # Affiche ".".
print(os.pardir) # Affiche "..".
print(os.sep) # Affiche "/".
```

— Comme la valeur des trois chaînes précédentes peut varier d'un système à l'autre, afin que votre code soit portable (c.-à-d. puisse s'exécuter facilement sur différentes machines), je vous demande de toujours passer par ces trois noms de variable et non vers leurs valeurs directement (ce qui serait un exemple de « codage en dur »).

— En pratique, `os.sep` est surtout utilisée comme chemin absolu vers la racine. Pour construire des chemins à partir d'autres chemins ou noms de fichier/dossier, il est possible d'utiliser la fonction `os.path.join`. Par exemple :

```
path = os.sep # Désigne la racine.
print(path) # Affiche "/".
path = os.path.join(path, "home", "guest")
print(path) # Affiche "/home/guest".
```

```
path = os.curdir # Désigne le répertoire de travail actuel.
print(path) # Affiche ".".
path = os.path.join(path, "documents", os.pardir, "videos", "
    films")
print(path) # Affiche "./documents/../videos/films".
```

— Il est possible de lister le contenu d'un dossier à l'aide de la fonction `os.listdir`. Si `path` est un chemin pointant vers un dossier `d`, alors `os.listdir(path)` vaut la liste des noms des éléments contenus dans `d`. Par exemple, les instructions suivantes, qui sont équivalentes, affichent le contenu du répertoire de travail actuel :

```
for e_name in os.listdir(os.getcwd()): print(e_name)
```

```
for e_name in os.listdir(os.curdir): print(e_name)
```

Notons que `os.listdir` lève une exception ($\approx$ le programme plante) si son argument pointe vers un fichier (plutôt qu'un dossier) ou vers un élément fictif.

— La fonction `os.path.exists` permet de savoir si son argument est un chemin pointant vers un élément existant. La fonction `os.path.isfile`, elle, ne vaut `True` que si le chemin pointe vers un fichier, alors que la fonction `os.path.isdir` ne vaut `True` que si le chemin pointe vers un dossier. Par exemple, si le système de fichiers contient à sa racine un dossier `"home"`, contenant un dossier `"guest"`, contenant un fichier `"chap_6.pdf"` :

```
path = os.path.join(os.sep, "home", "guest", "chap_6.pdf")
print(os.path.exists(path)) # Affiche "True".
print(os.path.isfile(path)) # Affiche "True".
print(os.path.isdir(path)) # Affiche "False".

path = os.path.join(os.sep, "home", "guest")
print(os.path.exists(path)) # Affiche "True".
print(os.path.isfile(path)) # Affiche "False".
print(os.path.isdir(path)) # Affiche "True".
```

Exercice 1 (Afficher les fichiers, $\star$)
Écrire une suite d'instructions affichant les noms des fichiers (et non des dossiers) contenus dans le répertoire de travail courant. □

Exercice 2 (Afficher les dossiers, ⋆)
Écrire une suite d'instructions affichant les noms des dossiers (et non des fichiers) contenus dans le répertoire de travail courant. □

Lecture de fichiers [Cours]

— La fonction `open` permet d'ouvrir un fichier afin d'y lire ou écrire du contenu. Cette fonction renvoie une *interface d'entrée/sortie* (« IO wrapper ») possédant des méthodes de lecture/écriture de contenu. Pour éviter certains effets indésirables, il faut *fermer* toute interface ouverte dès qu'elle n'est plus utile, avec la méthode `close`.

— Pour lire un fichier texte, il est possible de l'ouvrir en fournissant à `open` deux arguments :

1. un chemin pointant vers ce fichier ;
2. la chaîne `"r"` (pour « read »).

Le deuxième argument de `open` est appelé « mode d'ouverture ».

— Notons que `open` lève une exception (≈ le programme plante) son premier argument n'est pas un chemin vers un fichier texte existant.

— Python associe à toute interface d'entrée/sortie une position dans le fichier appelée « position courante ». Lorsque l'on lit ou écrit dans un fichier via une interface, on y lit ou écrit à la position courante. Lorsqu'un fichier est ouvert en mode `r`, la position courante est initialisée au tout début du fichier.

— La méthode `read` appelée sur l'interface ainsi obtenue renvoie, sous forme d'une chaîne de caractères, tout le contenu du fichier *à partir de la position courante*. La position courante est aussi déplacée par `read`, à la fin du fichier. Par exemple, si `filename` représente un chemin pointant vers un fichier texte :

```
f = open(filename, "r") # Ouverture pour lecture. Position
    courante initialisée au début du fichier.

print(f.read(), end="") # Affiche le contenu du fichier. La
    position courante est déplacée à la fin du fichier.

f.close() # Fermeture.
```

— Noter que tout fichier texte non vide *correctement formé* inclut un caractère de fin de ligne `"\n"` final. Il existe donc une distinction entre un fichier de texte vide, dont le contenu est la chaîne vide `""`, et un fichier de texte contenant une unique ligne vide, dont le contenu est `"\n"`.

— La position courante étant déplacée par `read` à la fin du fichier, si l'on appelle deux fois de suite cette méthode sur la même interface, le deuxième appel retourne nécessairement la chaîne vide (`""` ; il s'agit bien du contenu du fichier se situant entre la position du courante, se trouvant alors être la fin du fichier, et la fin du fichier).

— Plutôt que d'avoir à explicitement fermer le gestionnaire d'entrée/sortie avec la méthode `close`, il est possible d'ouvrir le fichier avec une construction en `with`. Par exemple, les deux blocs de code suivants sont équivalents :

```
f = open(filename, "r")

print("[début de la lecture]")
print(f.read(), end="")
print("[fin de la lecture]")

f.close() # Fermeture.
```

```
with open(filename, "r") as f: # f est automatiquement fermé
    après l'exécution du bloc de code introduit.
  print("[début de la lecture]")
  print(f.read(), end="")
  print("[fin de la lecture]")
```

— La méthode `readline` permet de lire un fichier texte ligne par ligne. Si la position courante se trouve être la fin du fichier, `readline` ne fait rien et retourne la chaîne vide. Sinon, `readline` déplace la position courante jusqu'après le prochain caractère de fin de ligne `"\n"` et retourne tout le texte se situant entre les deux. Dans ce cas, `readline` renvoie donc une *ligne de texte*, c.-à-d. une chaîne de caractères se terminant par un caractère de fin de ligne, et ne contenant aucun autre caractère de fin de ligne. (On peut se rappeler que la fonction `input` aussi retourne une ligne de texte, mais provenant d'une saisie clavier et non d'un fichier texte.) Par exemple, si `filename` représente un chemin pointant vers un fichier constitué d'exactement trois lignes de texte :

```
with open(filename, "r") as f:
  print(f.readline(), end="") # Affiche la première ligne du fichier.
  print(f.readline(), end="") # Affiche la seconde ligne du fichier.
  print(f.readline(), end="") # Affiche la troisième ligne du fichier.
  print(f.readline(), end="") # Aucun effet.
  print(f.readline(), end="") # Aucun effet.
```

— La méthode `readlines` fonctionne de manière similaire à `read`, dans le sens où elle lit le fichier de la position courante jusqu'à la fin et y déplace la position courante. Cependant, `readlines` ne retourne pas le contenu lu sous forme d'une unique chaîne de caractères, mais d'une liste de lignes de texte (chacune se terminant par le caractère de fin de ligne). Par exemple, si `filename` représente un chemin pointant vers un fichier constitué d'exactement trois lignes de texte :

```
with open(filename, "r") as f:
  l1 = f.readlines() # Liste de trois chaînes de caractères.
  print(len(l1)) # Affiche "3".
  l2 = f.readlines() # Liste vide.
  print(len(l2)) # Affiche "0".
```

Exercice 3 (Manipulation des méthodes de base, ⋆)
Considérer que `"/home/guest/documents/dialog.txt"` pointe vers un fichier texte contenant exactement les deux lignes suivantes (incluant les tirets) :

```
- Bonjour, comment allez-vous ?
- Très bien, et vous ?
```

Décrire très précisément (c.-à-d. notamment sans oublier les éventuelles lignes vides) l'affichage produit par l'exécution de chacune des suites d'instructions suivantes.

1.
```
import os
with open(os.path.join(os.sep, "home", "guest", "documents", "dialog.txt") as f:
  print(f.read())
  print(f.readline())
  print(f.readlines())
```

2.
```
import os
with open(os.path.join(os.sep, "home", "guest", "documents", "dialog.txt") as f:
  print(f.readline())
  print(f.readlines())
  print(f.read())
```

□

Exercice 4 (Réimplémentation de readlines, ⋆⋆)
Écrire une fonction `readlines` *réimplémentant la méthode du même nom, c.-à-d. prenant en argument* `f`*, le gestionnaire d'entrée/sortie obtenu à l'ouverture d'un fichier texte en lecture, et retournant la liste de toutes les lignes qui restent à y être lues. La fonction ne doit pas utiliser la méthode* `readlines` *mais plutôt* `readline`*.* □

Exercice 5 (Affichage de certaines lignes, ⋆⋆)

1. *Écrire une fonction* `f` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin pointant vers un fichier texte (existant), et affichant (sans saut de ligne supplémentaire) chacune des lignes ne commençant pas par* `"#"`*.*
2. *Modifier la fonction afin que rien ne se passe (en particulier, pas d'erreur) si l'argument* `filename` *passé ne pointe pas vers un fichier existant.*

□

Lecture de fichier et chargement en mémoire [Cours]

— Il peut être plus pratique d'utiliser `readlines` que `readline`. Cependant, alors qu'un appel à `readline` ne provoque la lecture et mise en mémoire que d'au plus une ligne de texte, un appel à `readlines` (comme à `read`) provoque la lecture et mise en mémoire de tout le contenu du fichier à partir de la position courante. Dans certaines situations, typiquement parce que ce contenu est trop volumineux ou parce que seule une ligne est utile, il est préférable voire nécessaire d'utiliser `readline`.

— Il est utile d'insister sur le fait que lorsqu'une fonction telle que `readline` lit une ligne vide, celle-ci est matérialisée par la chaîne `"\n"`. Cette chaîne est donc différente de la chaîne vide (`""`), retournée lorsque la position courante est déjà à la fin du fichier.

Exercice 6 (Fichier vide ?, ⋆)
Écrire une fonction `isEmpty` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin pointant vers un fichier texte (existant), et retournant* `True` *si ce fichier est vide,* `False` *sinon. Écrire une fonction ne nécessitant pas forcément la lecture de tout le fichier texte.* □

Fonctions utiles au traitement de fichiers texte [Cours]

— La méthode `lstrip` (« left strip ») appelée sans argument sur une chaîne de caractères `s` retourne la chaîne de caractères obtenue en supprimant de `s` tous les caractères d'espacement (ex : espace, tabulation) et de saut de ligne situés *à gauche* du premier caractère autre (c.-à-d. qui n'est ni un caractère d'espacement ni un saut de ligne). La méthode `rstrip` (« right strip ») a un comportement similaire mais supprime les caractères situés *à droite* du dernier caractère autre. Par exemple :

```
s1 = " \t\t  \n   Bonjour. Comment-allez vous ?   \t    \n"
s2 = s1.lstrip() # "Bonjour. Comment-allez vous ?   \t    \n"
s3 = s1.rstrip() # " \t\t  \n   Bonjour. Comment-allez vous ?"
```

— La méthode `strip` peut être vue comme une combinaison de `rstrip` et de `lstrip`. Appelée sans argument sur une chaîne de caractères `s`, elle retourne la chaîne de caractères obtenue en supprimant de `s` tous les caractères d'espacement et de saut de ligne situés à gauche du premier caractère autre et tous ceux situés à droite du dernier caractère autre. Par exemple :

```
s1 = " \t\t  \n   Bonjour. Comment-allez vous ?   \t    \n"
s2 = s1.strip() # "Bonjour. Comment-allez vous ?"
```

— Les méthodes `strip`/`rstrip` sont particulièrement utiles pour éliminer le caractère `"\n"` situé à la fin d'une ligne lue depuis un fichier texte.

— Par défaut, ces trois méthodes suppriment les caractères d'espacement et de saut de ligne, mais il est possible de spécifier l'ensemble des caractères à supprimer en leur passant comme argument une chaîne de caractères composée des caractères à supprimer. Par exemple :

```
s1 = "aabbabaccabcdddbaaab"
s2 = s1.strip("ab") # "ccabcddd"
s3 = s1.strip("abcd") # ""
```

— La méthode `split` appelée sur une chaîne de caractères `s` avec pour argument une chaîne de caractères `sep` retourne la liste de chaînes de caractères obtenue en découpant `s` au niveau de chaque occurrence de `sep`. Par exemple :

```
s = "bloup--blip-bloup--blop"
l = s.split("--") # ["bloup", "blip-bloup", "blop"]
```

```
s = "bloup--blip-bloup--blop"
l = s.split("-") # ["bloup", "", "blip", "bloup", "", "blop"]
```

— Il faut garder en tête que l'argument de `split` est interprété de manière très différente de l'argument de `strip` et ses variantes :
  — `s.split(sep)` découpe `s` au niveau de chaque occurrence (complète) de `sep`.
  — `s.strip(chars)` retourne une chaîne obtenue en supprimant de `s` des occurrences, non pas de `chars`, mais de n'importe quel caractère présent dans `chars`.

— La méthode `split` est la réciproque de `join`, dans le sens où si `s` et `sep` sont deux chaînes de caractères, `sep.join(s.split(sep))` vaut `s`.

— La méthode `split` est utile pour séparer (approximativement) les différents *tokens* (c.-à-d. occurrences de mots) d'une phrase en français. Par exemple :

```
s = "Le chat court après la souris."
l = s.split(" ") # ["Le", "chat", "court", "après", "la", "
    souris."]
```

— La méthode `split` accepte un argument supplémentaire indiquant un nombre maximum de coupes à effectuer. La méthode `split` appelée sur une chaîne de caractères `s` avec pour argument une chaîne de caractères `sep` et un entier `n` retourne la liste de chaînes de caractères obtenue en découpant `s` au niveau des **`n` premières occurrences de `sep`**. Par exemple :

```
s = "Le chat court après la souris."
l = s.split(" ", 3) # ["Le", "chat", "court", "après la souris
    ."]
l = s.split(" ", 0) # ["Le chat court après la souris."]
```

Exercice 7 (Découpage de noms, ⋆⋆)

1. *Écrire une fonction* `split_name` *prenant en argument une chaîne de caractères* `name` *supposée contenant un unique espace, et retournant la paire (un* `tuple`*) composée des deux sous-chaînes de* `name` *obtenues en la découpant au niveau de cet espace.*

   Contrat :
   `name = "George Sand"` → *retour :* `("George", "Sand")`

2. *Modifier la fonction proposée à la question précédente pour qu'elle renvoie* `None` *si* `name` *ne contient pas un unique espace.*

   Contrat :
   `name = "George Sand Michael"` → *retour :* `None`
   `name = "George"` → *retour :* `None`
   `name = "George Sand"` → *retour :* `("George", "Sand")`

3. *Modifier la fonction proposée à la question précédente pour qu'elle ne renvoie plus une paire* `(s1, s2)` *lorsque* `name` *contient un unique espace mais un dictionnaire de clefs* `"prénom"` *et* `"nom"`, `{"prénom": s1, "nom": s2}`.

   Contrat :
   `name = "George Sand"` → *retour :* `{"prénom": "George", "nom": "Sand"}`

□

Exercice 8 (Création de vocabulaire, ⋆⋆)
Écrire une fonction `words_set` *prenant en argument une chaîne de caractères* `filename` *supposée représenter le chemin d'un fichier texte, et retournant l'ensemble des mots apparaissant dans ce fichier. (Supposer que tous les tokens sont séparés par des espaces.)* □

3 Écriture du système de fichiers

Écriture de fichiers [Cours]

— Si l'on appelle la fonction `open` avec comme arguments
  — un chemin pointant vers un fichier fictif mais situé dans dossier existant et
  — le mode `"w"` (« write »),
  alors le fichier pointé par le chemin est créé (avec un contenu vide) et une interface d'entrée/sortie permettant d'y écrire du contenu est retournée. Par exemple, si le répertoire de travail actuel contient un dossier `"documents"` qui lui ne contient pas de fichier `"test.txt"`, alors l'exécution des instructions suivantes crée un tel fichier, entièrement vide :

```
with open(os.path.join(os.curdir, "documents", "test.txt"), "w
    ") as f: # Ouverture pour écriture.
  print("Empty file created.")
```

— Si le chemin passé comme premier argument à `open` avec le mode `"w"` pointe vers un fichier existant, alors ce fichier est *écrasé*, c.-à-d. que son contenu est effacé.
— Dans tous les cas, l'interface d'entrée/sortie retournée par `open` en mode `"w"` correspond donc à un fichier vide ; la position courante est définie au début du fichier, qui se trouve être aussi la fin.
— La méthode `write`, appelée sur une interface ouverte en écriture et avec pour argument une chaîne de caractères `s`, écrit `s` à partir de la position courante et déplace celle-ci jusqu'après le dernier caractère inséré. Par exemple, après exécution des instructions suivantes, le fichier pointé par `filename` contient exactement la ligne de texte `"Hello world!\n"` :

```
with open(filename, "r") as f: # Ouverture pour écriture.
  f.write("Hell")
  f.write("o world!")
  f.write("\n")
```

— Contrairement à `print` qui, par défaut, affiche à l'écran un saut de ligne supplémentaire après son argument, la méthode `write` n'écrit dans un fichier que les sauts de ligne explicitement contenus dans son argument. Par exemple, après exécution des instructions suivantes, le fichier pointé par `filename` n'est pas un fichier texte correctement formé car son contenu ne se termine pas par un caractère de saut de ligne :

```
with open(filename, "r") as f: # Ouverture pour écriture.
  f.write("Hello world!")
```

— Il est possible d'utiliser `open` avec le mode `"a"` (« append »). Ce mode fonctionne comme `"w"` dans le cas où le chemin pointe vers un fichier inexistant, mais n'écrase pas le fichier s'il existe. Dans ce dernier cas, le contenu du fichier n'est pas modifié et la position courante est initialisée à la fin du fichier. Par exemple, si `filename` pointe vers un fichier contenant une ligne `"Hello\n"`, l'exécution des instructions suivantes y ajoutent une seconde ligne `"word!\n"` :

```
with open(filename, "a") as f: # Ouverture pour écriture.
  f.write("world!\n")
```

Pour aller plus loin [Cours]

— Il est possible de créer des répertoires avec les méthodes `os.makedirs` et `os.mkdir`.
— Il est possible de télécharger des fichiers avec la méthode `urllib.request.urlretrieve`.

4 À faire chez soi

Exercice 9 (Parcours d'une branche, ⋆⋆⋆)
Écrire une fonction `listdir_rec` *prenant en argument une chaîne de caractères* `path` *et (i) retournant* `True` *et affichant les chemins de tous les fichiers descendants du dossier pointé par* `path` *si un tel dossier existe, et (ii) retournant* `False` *(sans rien afficher) sinon. La fonction doit être récursive, c.-à-d. que son fonctionnement repose sur le fait qu'un appel à* `listdir_rec` *apparaît dans le corps de* `listdir_rec` *elle-même.* □

Exercice 10 (Premières lignes, ⋆⋆)

1. *Écrire une fonction* `f` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin vers un fichier texte, et un entier positif* `n`*, et affichant (sans saut de ligne supplémentaire) les* `n` *premières lignes de ce fichier (ou moins si le fichier en contient moins).*
2. *Modifier la fonction de telle sorte que si l'argument* `n` *passé est strictement négatif, toutes les lignes soient lues.*

□

Exercice 11 (Création de chemins par jointure (I), ⋆⋆)
Dans cet exercice, on cherche à implémenter des fonctions implémentant de manière plus ou moins simplifiée le comportement de `os.path.join`*.*

1. *Écrire une fonction* `myJoin` *prenant en argument une liste de chaînes de caractères* `l` *et retournant la chaîne obtenue en concaténant, séparés par des occurrences de* `os.sep`*, les élements non vides de* `l`*. La fonction ne doit pas utiliser* `os.path.join`*, mais peut utiliser* `join`*.*

   Contrat :
   (En supposant que `os.sep = "/"`*.)*
   `l = ["..", "documents"]` → *retour :* `"../documents"`
   `l = ["a/b", ".", "c"]` → *retour :* `"a/b/./c"`
   `l = ["..", "", "documents", ""]` → *retour :* `"../documents"`
   `l = ["..", "", "documents", "", ""]` → *retour :* `"../documents"`
2. *Récrire* `myJoin` *de manière à ce que si un élément de l'argument* `l` *vaut ou commence par* `os.sep`*, tous les éléments précédents soient ignorés. Veiller à bien généraliser le comportement illustré dans le contrat, impliquant probablement de gérer les éléments égaux à* `os.sep` *différement des autres éléments commençant par* `os.sep`*.*

   Contrat :
   (En plus du contrat de la question précédente et toujours en supposant que `os.sep = "/"`*.)*
   `l = ["/", "home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["/home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["..", "/", "tmp", "/", "home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["..", "/", "tmp", "/home", "guest"]` → *retour :* `"/home/guest"`

□

Exercice 12 (Comptage d'occurrences, ⋆⋆)
Écrire une fonction `words_count` *prenant en argument une chaîne de caractères* `filename` *supposée être un chemin pointant vers un fichier texte, et retournant le dictionnaire associant à chaque mots apparaissant dans ce fichier son nombre d'occurrences. (Supposer que tous les tokens sont séparés par des espaces.)* □

Exercice 13 (Découpage, ⋆⋆⋆)
Écrire une fonction `split` *réimplémentant la méthode du même nom pour les séparateurs constitués d'un unique caractère, c.-à-d. prenant en argument une chaîne de caractères* `s` *et un caractère* `sep`*, et retournant la liste de chaînes de caractères obtenue en découpant* `s` *au niveau de chaque occurrence de* `sep`*. La fonction ne doit pas utiliser la méthode* `split`*. (Bien vérifier sur machine que la fonction proposée satisfait le contrat.)*

Contrat :
`s, sep = "abc.de..fgh."` → *retour :* `["abc", "de", "", "fgh", ""]`
`s, sep = "abc.de..fgh"` → *retour :* `["abc", "de", "", "fgh"]`

□

Exercice 14 (Création de chemins par jointure (II), ⋆⋆⋆)
Écrire une fonction `myJoin` *réimplémentant* `os.path.join`*, à l'unique différence près qu'alors que* `os.path.join` *accepte n'importe nombre d'arguments de type* `str`*,* `myJoin` *doit accepter un unique argument* `l`*, une liste de chaînes de caractères. Attention,* `os.path.join` *a un comportement qui n'est pas facile à décrire ; étudier attentivement le contrat et ne pas hésiter à faire d'autres tests sur machine pour comprendre précisément la valeur retournée par* `os.path.join`*. La fonction ne doit pas utiliser* `os.path.join`*, mais peut utiliser* `join`*. Indice : Construire une liste de chaînes de caractères* `tmp` *et retourner* `"".join(tmp)`*.*

Contrat :
(En supposant que `os.sep = "/"`*.)*

`l = ["..", "documents"]`	→	*retour :* `"../documents"`
`l = ["a/b", ".", "c"]`	→	*retour :* `"a/b/./c"`
`l = ["..", "", "documents", ""]`	→	*retour :* `"../documents/"`
`l = ["..", "", "documents", "", ""]`	→	*retour :* `"../documents/"`
`l = ["/", "home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/", "home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home/", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home//", "guest"]`	→	*retour :* `"/home/guest"`

□