

Sérialisation et désérialisation

- La sérialisation consiste à transformer une structure de données en une suite d'octets afin de la stocker sur disque ou de l'envoyer sur le réseau. La désérialisation est l'opération inverse.
- Python dispose de plusieurs modules de sérialisation. Le plus utilisé est *pickle*, qui sérialise dans un format binaire uniquement lisible par Python.
- On peut également installer le module *yaml* qui sauvegarde les données dans un format lisible proche de XML ou le module *json* qui sauvegarde au format JSON (*JavaScript Object Notation*).
- Avec *pickle* et *json*, la sérialisation consiste à appeler les méthodes *dump* (pour écrire dans un fichier) ou *dumps* (pour écrire une chaîne d'octets) et la désérialisation consiste à appeler les méthodes *load* ou *loads*.
- Avec *yaml*, les méthodes *dump* et *load* traitent à la fois les fichiers et les chaînes d'octets.

Exemples

- Avec *pickle* :

```
import pickle

e1 = Ensemble(...)

# Sérialisation de e1 dans un fichier
with open('ensemble.pickle', 'wb') as f:
    pickle.dump(e1,f)                # ou : pickle.dump(e1, open('ensemble.pickle', 'wb'))

# Désérialisation
with open('ensemble.pickle', 'rb') as f:
    e1 = pickle.load(f)              # ou : e1 = pickle.load(open('ensemble.pickle', 'rb'))
```

- Avec YAML (il faut installer *PyYAML*) :

```
import yaml

e1 = Ensemble(...)

# Sérialisation de e1 dans un fichier
with open('ensemble.yaml', 'w') as f:
    yaml.dump(e1,f)                  # ou yaml.dump(e1, open('ensemble.yaml', 'w'))

# Désérialisation
with open('ensemble.yaml', 'r') as f:
    e1 = yaml.load(f)                # ou e1 = yaml.load( open('ensemble.yaml', 'r'))
```

Sérialisation d'un Ensemble

```
import pickle, yaml
(...)
s = pickle.dumps(ens)
print(s)
ens3 = pickle.loads(s)
print(ens3)

with open('ensemble.pickle', 'wb') as f:
    pickle.dump(ens, f)

with open('ensemble.pickle', 'rb') as f:
    ens3 = pickle.load(f)

s = yaml.dump(ens)
print(s)

ens4 = yaml.load(s)
print(ens3)

with open('ensemble.yaml', 'w') as f:
    yaml.dump(ens, f)

with open('ensemble.yaml', 'r') as f:
    ens4 = yaml.load(f)
```

```
# Sérialise dans un tableau d'octets
# b'\x80\x03censemble\nEnsemble\nq\x00)...

# 1, 2, 3, 12

# Sérialise dans un fichier binaire

# YAML n'a qu'une méthode dump
# !!python/object:ensemble.Ensemble
# _Ensemble__corps: {1: true, 2: true, 3: true, 12: true}

# 1, 2, 3, 12

# Sérialise dans un fichier texte
```

Fichiers texte

- La fonction `open()` permet d'ouvrir un fichier en lecture (mode `"r"` par défaut), en écriture (mode `"w"`) ou en ajout (mode `"a"`).
- Le mode peut être suivi de la lettre `b` pour indiquer une ouverture en mode binaire.
- En règle générale, il est préférable de considérer le fichier ouvert comme un itérateur classique (on peut alors utiliser une instruction `for`).
- Les méthodes `read()` et `readline()` renvoient une *chaîne* classique. La méthode `readlines()` renvoie un *tableau de chaînes* et n'ajoute pas de retour à la ligne.
- En lecture, la fin de fichier est détectée lorsque `read()` ou `readline()` renvoient une chaîne vide.
- La méthode `write()` écrit une chaîne dans le fichier ouvert en écriture. Elle renvoie le nombre de caractères écrits.
- On ferme un fichier avec la méthode `close()`.
- Si l'on a utilisé la construction `with`, le fichier ouvert est fermé automatiquement à la sortie du bloc.

Fichiers texte

Exemples :

```
fd = open("monfichier.txt")      # Ouverture en lecture par défaut
for ligne in fd:
    # faire quelque chose avec ligne
fd.close()                       # Fermeture explicite

for ligne in open("autre.txt"):
    # faire quelque chose avec ligne
# Ici le fichier est implicitement fermé car on est sorti de sa portée

fd = open("monfichier.txt")
contenu_complet = fd.read()      # une chaîne contenant tout le fichier

fd.seek(0)                      # on revient au début
prem = fd.readline()            # une chaîne contenant la 1ere ligne
sec  = fd.readline()            # une chaîne contenant la 2e ligne

fd.seek(0)
tab = fd.readlines()             # un tableau de lignes
print(tab[0])                   # première ligne...
```

Fichiers texte

Exemples :

```
fd = open("monfichier.txt", "w")      # Ouverture en écriture

# Si monfichier.txt existait, il a donc été écrasé !

fd.write("coucou\n")
fd.close()

with open("monfichier.txt") as fd:    # utilisation d'un bloc with
    # ici, fd est ouvert en lecture
# Sortie du bloc : fd est automatiquement fermé...

with open("monfichier.txt", "a") as fd:
    fd.write("au revoir\n")           # Ajout d'une ligne à la fin du fichier

from urllib.request import urlopen   # Gestion des URL comme des fichiers
import codecs

for ligne in urlopen("http://www.monsite.fr"):
    # on récupère des octets... donc il faut les décoder pour les transformer en caractères UTF-8
    print(codecs.decode(ligne))
```

Expressions régulières

Expressions régulières

- Les expressions régulières sont gérées via le module `re`, qu'il faut donc importer.
- Bien qu'elle ne soit pas nécessaire, la méthode `regex = re.compile(motif)` permet d'optimiser les recherches. Elle permet également de passer des options (`re.I` ou `re.IGNORECASE`, par exemple).
- Si le *motif* contient des caractères spéciaux, on peut utiliser une chaîne brute en la préfixant du caractère `r` : `r"ceci est une chaîne brute"` (fonctionne également avec `'`, `'''` et `"""`).
- La méthode `search` d'une expression régulière renvoie un objet « match » si une correspondance a été trouvée, `None` sinon.
- La méthode `match` renvoie un objet « match » si la chaîne passée en paramètre correspond exactement au motif, `None` sinon.
- La méthode `sub` renvoie une chaîne où la première occurrence du motif dans la chaîne initiale a été remplacée par une autre chaîne.

Expressions régulières

- La méthode *findall* renvoie la liste de toutes les chaînes correspondant au motif dans la chaîne. La méthode *finditer* fait de même, mais renvoie un itérateur. On peut paramétrer le début et la fin de la recherche dans la chaîne.
- Voir les documentations complètes de ces méthodes dans la documentation du module *re...*

Opérations sur un objet « match »

Un objet « match » renvoyé par les appels à *search* ou *match* dispose des méthodes suivantes :

- *group(grp)* renvoie une chaîne correspond au texte capturé par l'expression ou la chaîne capturée par le groupe passé en paramètre (le groupe 0 correspond à toute la capture). Si l'on a utilisé des groupes nommés, on peut passer les noms en paramètre.
- *groups* renvoie un tuple contenant toutes les groupes capturés, à partir du groupe 1.
- *groupdict* renvoie un dictionnaire de tous les groupes nommés qui ont été capturés. Les clés sont les noms des groupes.
- *start* et *end* renvoient l'indice de début et de fin de la capture.

Opérations sur un objet « match »

- Un groupe est délimité entre parenthèses. Un groupe est nommé par *?P<nom>*, (*?P<groupe>\d+*), par exemple.
- Le traitement d'un groupe nommé est plus lent que celui d'un groupe non nommé.
- Ce qui a été capturé par un groupe est représenté par *\i* pour le groupe *i* ou par (*?P=nom*) pour le groupe nommé *nom*.
- Si l'on ne veut pas mémoriser le groupe capturé, on utilise la notation (*?: ...*).

Exemple

On veut déterminer si un nom de fichier entré au clavier est un nom de fichier DOS valide, sachant que :

- Les noms de fichiers DOS ne sont pas sensibles à la casse
- Ils sont de la forme 8.3 : un nom principal et une extension, qui est facultative.
- Le nom principal doit commencer par une lettre et contenir des lettres des chiffres ou des blancs soulignés. Les lettres accentuées ne sont pas reconnues.
- L'extension ne contient que des lettres ou des chiffres.

Quelle expression régulière permettra de déterminer si la saisie est correcte et permettra d'isoler les deux parties du nom (nom principal et extension) ?

- Nom principal : `[a-zA-Z][a-zA-Z0-9_]{0,7}`
- Extension : `[a-zA-Z0-9]{1,3}`
- Nom complet, avec trois groupes pour capturer le nom principal et l'extension facultative :
`([a-zA-Z][a-zA-Z0-9_]{0,7})(\.[a-zA-Z0-9]{1,3})?`
- Idem, mais avec groupes nommés :
`(?P<nom>[a-zA-Z][a-zA-Z0-9_]{1,7})(\.(?P<ext>[a-zA-Z0-9]{1,3}))?`

Exemple

```
import re

nom_dos = re.compile(r"(?P<nom>[a-zA-Z][a-zA-Z0-9_]{1,7})(\.(?P<ext>[a-zA-Z0-9]{1,3}))?")
nom_fic = "blabla.txt"          # ou nom_fic = input("Nom du fichier : ")
result = re.match(nom_dos, nom_fic)
if result is not None:
    print(result.groups())      # ('blabla', '.txt', 'txt')
    nom, extension = result.group("nom"), result.group("ext")
    print(nom, extension)      # blabla txt
    nom, extension = result.group(1), result.group(3)
    print(nom, extension)      # blabla txt
else:
    print(nom_fic, "n'est pas un nom de fichier DOS")

nouveau_nom = re.sub(nom_dos, r"bllibli\2", nom_fic)
print(nouveau_nom)           # bllibli.txt

print(re.sub(r"(\w+)\s+(\w+)", r"\2 \1", "mots inversés"))    # inversés mots

while True:
    age = input("Entrez un âge : ")
    if re.match("\d+", age) is not None:    # Plus besoin de try: ...
        age = int(age)                     # Mais age pourrait être une valeur non correcte...
        break
```

Programmation d'interfaces graphiques
