Chapitre 3

Traitements des fichiers

Sommaire

Travailler directement avec l'interpréteur présente des avantages et inconvénients. Un des inconvénients majeur est que l'on perd toutes les manipulations réalisées à la fermeture du programme. On peut palier à cela en enregistrant le code dans un fichier sur le disque dur et y faire référence à chaque fois
Le fichier doit porter une extension particulière : `.py` et non .txt ou .doc. . . .

Exemple 2. Créons un fichier nommé `script1.py` et mettons-y les lignes de codes suivants :

```
1  import sys
2  print (sys.platform)
3  print (sys.version)
4  print (sys.copyright)
```

On peut l'exécuter depuis la ligne de commande grâce à l'instruction

```
$ python script1.py
```

On obtient le résultat à la console. On peut aussi, pour des besoins ultérieurs, stocker le résultat dans un fichier séparé. On peut dans ce cas faire une redirection du résultat sur un fichier `results.txt`.

```
$ python script1.py > results.txt
```

Vous pourrez dans ce cas ouvrir le fichier `results.txt` à tout moment
Pour ajouter des infos sur le fichier `results.txt`, dans la redirection, on utilise le symbole '≫' au lieu de '>'.

Nous allons présenter dans la suite, l'utilisation des fichiers ainsi que les mécanismes d'entrée-sortie correspondant en Python. Nous introduisons l'objet fichier : sa fonction intégrée, ses méthodes intégrées et ses attributs.

3.1 Objets de type fichier

Les objets de type fichier (ou objets fichiers) permettent d'accéder à des fichiers disque ordinaires, mais aussi à tout type de fichier utilisant cette abstraction. Une fois que l'O.S. dispose des extensions nécessaires, vous pouvez accéder à d'autres objets à l'aide d'interface de type fichier, exactement comme vous accéderiez à un fichier ordinaire.

Lorsque vous serez plus expérimenté en Python, vous rencontrerez de nombreux cas où vous aurez affaire à des objets analogues à des fichiers. Au nombre de ces exemples, vous trouverez l'ouverture d'une URL pour lire une page web en temps réel, ou l'exécution d'une commande dans un processus séparé en assurant une communication bidirectionnelle avec celui-ci comme s'il s'agissait de deux fichiers ouverts simultanément, l'un pour écrire et l'autre pour lire.

La fonction intégrée ci-dessous (`open()`) retourne un objet de type fichier qui est ensuite utilisé pour toutes les opérations qu'on souhaite exécuter sur le fichier en question.
Il faut juste voir les fichiers comme une suite d'octets contigus. Partout où l'on souhaite transmettre des données, on utilise en général un flux d'octets d'une sorte ou d'une autre, que ce flux soit constitué par des octets séparés ou par des blocs de données.

Python étant à l'origine conçu pour réaliser des opérations systèmes, il est naturellement pourvu d'outils très simples et très efficaces : en un mot très pythoniques. Ces outils ont été améliorés tout au long de l'évolution du langage.

3.2 Fonction intégrée `open()`

La fonction intégrée `open()` fournit une interface générale permettant d'initialiser le processus d'entrées-sorties sur un fichier. La fonction `open()` retourne un objet de type fichier si l'ouverture du fichier a réussi, ou produit une situation d'erreur. La syntaxe de base de la fonction `open()` est :

```
1 | objet_fichier = open(nom_fichier, mode_acces='r', bueffering=-1)
```

Le paramètre `nom_fichier` est une chaîne contenant le nom du fichier à ouvrir. Il peut s'agir d'un chemin d'accès relatif ou absolu. Le paramètre optionnel `mode_acces` est également une chaîne contenant un certain nombre d'indicateurs spécifiant le mode d'accès utilisé pour ouvrir le fichier. En général, les fichiers sont ouverts dans l'un des modes `'r'`, `'w'` ou `'a'`, représentant respectivement la lecture, l'écriture et la modification.
Un fichier ouvert en mode `'r'` doit déjà exister. Dans le cas d'un fichier ouvert en mode `'w'`, le contenu, s'il existe, sera d'abord effacé, puis le fichier sera recréé. Tout fichier ouvert en mode `'a'` sera en mode mise à jour et toutes les écritures

s'effectueront en fin de fichier, même si vous avez essayé d'y accéder à un autre endroit. Si le fichier n'existe pas, il sera créé de la même façon qu'avec `'w'`.

L'autre argument `bueffering` sert à indiquer quel regroupement est effectué sur les données lorsqu'on accède au fichier. La valeur 0 indique qu'il n'y a pas de regroupement (les caractères sont transmis un par un), la valeur 1 indique que les données sont transmises ligne par ligne et toute valeur supérieur à 1 indique que les données sont stockées dans un tampon (buffer), dont ce paramètre représente justement la taille. Si on indique pas de valeur, ou une valeur négative, on utilise le regroupement par défaut défini sur le système, qui est le mode ligne pour les périphériques de type télétype ou terminal alphanumérique (`tty`), ou un tampom de taille standard pour le reste.

Voici quelques exemples d'ouverture de fichiers :

```
fp = open('/etc/passwd')    #ouverture du fichier en lecture.
fp = open('test', 'w')   #ouverture du fichier en écriture.
fp = open('data', 'r+')  #ouverture du fichier en lecture-écriture.
```

Voici un exemple simple qui nous montre comment ouvrir un fichier :

```
>>> with open('test.txt','w') as f:
        type(f)
        dir(f)

<class '_io.TextIOWrapper'>
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__enter__',
 '__eq__', '__exit__', '__format__', '__ge__',
 '__getattribute__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__',
 '__lt__', '__ne__', '__new__', '__next__',
 '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '_checkClosed', '_checkReadable',
 '_checkSeekable', '_checkWritable', '_finalizing',
 'buffer', 'close', 'closed', 'detach', 'encoding',
 'errors', 'fileno', 'flush', 'isatty', 'line_buffering',
 'mode', 'name', 'newlines', 'read', 'readable',
 'readline', 'readlines', 'seek', 'seekable', 'tell',
 'truncate', 'writable', 'write', 'writelines']
```

La variable `f` est une structure qui correspond à une entrée/sortie (descripteur de fichier) et qui peut être utilisé comme un générateur. Nous verrons comment utiliser une telle variable pour répondre à différents besoins.

Remarque. Notons qu'il existe des modes binaires pour lire et écrire : `'rb'` et `'wb'`.

3.2.1 Gestion universelle du séparateur de lignes

Lorsque vous utilisez l'option `'U'` (comme mode d'accès) pour ouvrir un fichier, et cela quelque soit la méthode de lecture, Python retourne tous les séparateurs de ligne sous la forme d'un caractère `\n` (le mode `'rU'` est également fourni pour faire pendant à l'option `'rb'`). Il est possible de gérer de cette manière des fichiers possédant plusieurs types de séparateurs de lignes.

3.2.2 Méthodes intégrées s'appliquant aux fichiers

Une fois que la méthode `open()` s'est terminée avec succès et on a retourné un objet de type fichier, tous les accès ultérieurs à ce fichier s'effectuent à l'aide de ce pointeur, ou handle. Il existe quatre méthodes s'appliquant aux fichiers :

`Entrées : readline()` lit une ligne du fichier ouvert. La ligne y compris le ou les caractères de fin de ligne est retournée sous forme de chaîne.
`readlines()` lit toutes les lignes restantes et les retourne sous forme d'une liste de chaînes.

`Sorties :` La méthode `write()` effectue le traitement inverse de readline(). Elle prend une chaîne de caractères qui peut contenir une ou plusieurs lignes de données et écrit ces données dans le fichier.
La méthode `writelines()` opère sur une liste, tout comme `readlines()` mais accepte une liste de chaîne et les écrit dans un fichier.

```
>>>f = open("Fichiertexte", "w")
>>>f.write("Ceci est la ligne un\nVoici la ligne deux\n")
>>>f.write("Voici la ligne trois\nVoici la ligne quatre\n")
>>>f.close()
-----------------------------------------------------------------
>>> f = open('Fichiertexte','r')
>>> t = f.readline()
>>> print (t)
Ceci est la ligne un
>>> print (f.readline())
Voici la ligne deux
-----------------------------------------------------------------
>>> f = open('Fichiertexte','r')
>>> donnees=f.readlines()
>>> print (donnees)
['Ceci est la ligne un\n', 'Voici la ligne deux\n',
'Voici la ligne trois\n', 'Voici la ligne quatre\n']
>>>
-----------------------------------------------------------------------
```

La méthode readlines() permet donc de lire l'intégralité d'un fichier en une instruction seulement. Cela n'est possible toutefois que si le fichier à lire n'est pas trop gros : puisqu'il est copié intégralement dans une variable, c'est-à-dire dans la mémoire vive de l'ordinateur, il faut que la taille de celle-ci soit suffisante. Si vous devez traiter de gros fichiers, utilisez plutôt la méthode readline() dans une boucle par exemple.

Le code qui suit permet de recopier un fichier en éliminant les lignes de commentaires.

```
def filtre(source, destination):
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while True:
        txt = fs.readline()
        if txt =='':
            break
        if txt[0] != '#':
            fd.write(txt)
    fs.close()
    fd.close()
    return
```

Test. *En guise d'exercice, améliorer ce filtre pour qu'il supprime purement et simplement tous les commentaires contenus dans le document.*

Remarque. Lorsqu'on lit des lignes à partir d'un fichier, en utilisant une méthode de lecture, Python ne supprime pas les caractères de fin de ligne. C'est à vous de le faire. On peut par exemple procéder comme suit :

```
1   f= open("Fichiertexte", 'r')
2   donnees = [line.strip() for line in f.readlines()]
3   f.close()
```

De même que les méthodes de sortie `write` il faudra penser à rajouter les caractères de fin de ligne pour les renvois à la ligne.

Notons qu'on peut utiliser aussi la méthode `print` avec l'option `file` pour écrire dans un fichier.

Exemple 3.

```
1 with open("Fichiertexte", 'w') as f:
2     print("Ceci est un test", file=f)
```

Sauver et restituer des valeurs de types différents :

La fonction `write` prend comme argument une chaîne de caractères qu'il enregistre dans le fichier sur lequel on l'appelle. Cela pose un problème : à la lecture, pourrions nous faire la différence entre les chaînes de caractères et les nombres : par exemple

```
1  >>> x=1
2  >>> y=23.4
3  >>> z=56.789
4  ------------------------------------------------
5  >>>test=open('Fichiertexte','w')
6  >>> test.write(str(x))
7  >>> test.write(str(y))
8  >>> test.write(str(z))
9  >>> test.close()
10 >>> test=open('Fichiertexte','r')
11 >>> test.readlines()
12 ['123.456.789']
```

Ce nombre là n'est pas ce qu'on espérait avoir ('1' '23.4' '56.789').

La solution à ce problème nous vient de plusieurs solutions parmi lesquelles : le module `pickle` (conserver) et le module `json`! que nous allons étudier.

Pickle

La bibliothèque `pickle` est un module permettant de rendre persistantes très facilement des données simplement en les écrivant sur le disque dur sous une représentation très aisément lisible et facile à écrire.

```
1 >>> import pickle as pk
2 >>> f = open('Fichiertexte', 'wb')
3 >>> pk.dump(x, f)
4 >>> pk.dump(y, f)
5 >>> pk.dump(z, f)
```

```
>>> f.close()
>>> f = open('Fichiertexte', 'rb')
>>> t=pk.load(f)
>>> print (t, type(t))
1 <class 'int'>
>>> t=pk.load(f)
>>> print (t, type(t))
23.4 <class 'float'>
>>> t=pk.load(test)
>>> print (t, type(t))
56.789 <class 'float'>
```

La fonction `load()` effectue le travail de la fonction inverse de `dump()` à savoir restituer les valeurs sauvées par `dump`, une à une avec leur type correspondant.

Remarque.
On pourra utiliser la syntaxe avec `with` comme au début de la section en lieu et place de l'affectation explicite de l'objet fichier à une variable.

```
>>> x,y,z=1, 23.4, 56.789
```

json

Les chaînes de caractères peuvent facilement être lues et écrites dans des fichiers. Pour les nombres c'est un peu plus compliqué. Puisque la méthode `read` renvoie toujours une chaîne de caratères, nous devrons donc nous servir de fonctions telles que `int()` qui prendrait $'123'$ pour renvoyer 123. Si l'on veut sauver des données plus complexes comme des listes, des dictionnaires, les choses deviennent encore plus compliquées.
Python nous permet de nous servir du populaire format déchange de données `json (JavaScript Object Notation)`. Le module `json` nous permet de `serialiser` des données python i.e les convertir en chaînes de caractères de `json` et les `deserialiser` : l'opération inverse i.e reconstruire les données python à partir des chaînes de caratères.
Entre la `serialisation` et la `deserialisation`, les chaînes de caractères représentant l'objet peuvent être sauvées dans un fichier, des données, ou envoyées à travers le réseau à une machine distante.

Remarque. Le format `json` est très utilisé dans les applications modernes pour l'échange de données. Beaucoup de programmeurs sont familiers avec `json` ce qui en fait un bon choix pour l'interopérabilité.

Si on a un objet `x`, on peut donner sa représentation en chaîne de caractères `json` comme suit :

```
>>> import json
>>> json.dumps([1, 'bonjour', ['a','b','c']])
   '[1, ''bonjour'', [''a'', ''b'', ''c'']]'
```

`json` dispose aussi d'une autre fonction `dump()` qui `serialise` un objet dans un fichier texte. Donc si f est un objet fichier ouvert en écriture, on peut faire :

```
json.dump(x,f)
```

Pour décoder l'objet à nouveau, si f est un fichier objet ouvert en lecture, on peut faire

```
x = json.load(f)
```

3.3 Type de fichiers

L'extension d'un fichier aide le système d'exploitation à définir le programme qui sera associer à ce fichier. Par exemple sous windows, un fichier `test.docx` sera associé à MS Word.
L'extension de fichier indique souvent le type du fichier ou son format.
Attention à ne pas confondre l'extension d'un ficier et le format de ce fichier même si le plus souvent ils réfèrent la même chose. L'extension est donc juste la chaîne de caractères qui suit le "point" après le nom du fichier.
Python prend en charge deux types de fichiers : les fichiers textes et les fichiers binaires.
Les fichiers binaires sont des séquences d'octets ordonnées. Les formats binaires peuvent contenir plusieurs types de données dans le même fichier, par exemple : `audio, image, vidéo, etc.`. Ces fichiers peuvent s'ouvrir avec des applications dédiées mais s'interpréteront différemment avec un éditeur de texte par exemple.
On peut facilement reconnaître un fichier binaire d'un fichier texte par son extension.
Par convention les extensions reflètent le format du fichier en question.

Exemple 4.

Formats binaires — Images : `jpg, png, gif, bmp, tiff, psd, ...`
- — Vidéos : `mp4, mkv, avi, mov, mpg, vob, ...`
- — Audio : `mp3, aac, wav, flac, ogg, mka, wma, ...`
- — Documents : `pdf, doc, docx, xls, xlsx, ppt, pptx, odt, ...`
- — Archives : `zip, rar, 7z, tar, iso, ...`
- — Executables : `exe, dll, class, ...`

Formats textes — Web : `html, xml, css, svg, json`
- — Codes sources : `c, cpp, , cs, js, py, java, rb, pl, php, sh, ...`
- — Documents : `txt, tex, markdown, asciidoc, rtf, ...`
- — Configuration : `ini, cfg, rc, reg, ...`
- — Tableur : `csv, tsv, ...`

Exercice 1.
Ecrivez un programme qui lit et affiche chaque ligne d'un fichier `text.txt` donné.

Exercice 2.
Utilisez maintenant la methode `read` pour lire le contenu du fichier `test.txt` Essayez de comprendre la différence entre `read, readline et readlines`

Exercice 3.
Ecrivez un programme prenant un fichier source .py contenant des commentaires et de supprimer seulement le caractère permettant de commenter à savoir le '#'.

Exercice 4.
Ecrivez un programme python qui renverse tous les mots d'un fichier texte donné.

Exercice 5.
Ecrivez un programme capable de compter toutes les occurrences de chaque mot du texte et qui donne à la fin le nombre de mots du texte donné.

Exercice 6.
Ecrivez un programme qui détermine le mot le plus long d'un texte donné.

3.4 Cas pratiques

3.4.1 Lecture d'un dictionnaire

Un fichier présent dans notre ordinateur qui nous aide dans la correction automatique est le fichier `words` dont le chemin sous linux ou MacOS est

```
/usr/share/dict/words
```

Sous windows probablement ce sont les fichiers .dic du répertoire

```
%AppData%\Microsoft\Spelling\EN-US
```

Sinon faîtes une recherche de fichiers avec l'extension `.dic`.
Nous allons afficher les mots et le nombre de mots de 15 lettres contenus dans le `words`.

Exercice 7.
Créer des dictionnaires formés de mots de même longueur $(4, 5, 6, 7, 8)$ à partir d'un dictionnaire de votre ordinateur.
Créer des dictionnaires formés de 4 chiffres, 5 chiffres, 6 chiffres, 7 chiffres et 8 chiffres.

3.5 Archivage et Compression

3.5.1 Archivage : `zip` et `tar`

Les formats d'archives avec lesquels nous allons travailler sont les formats standards courants : `zip` et `tar`.

Archives `zip`

Le format d'archive `zip` est un standard d'archivage et de compression très utilisé. Le module de base du format d'archive `zip` dans `Python` est `zipfile`. Ce module fournit des outils efficaces pour créer, lire, écrire, ajouter et lister un fichier zippé. Le module `zipfile` définit les classes :

`ZipFile :` qui permet de lire et d'écrire des fichiers `zip`.

`is_zipfile :` qui renvoie `True` si le fichier est une archive `zip` valide, sinon elle renvoie `False`.

`PyZipFile :` permet de créer des archives `zip` contenant des libraires `Python`.

Nous allons principalement nous intéresser à la classe `ZipFile`.
Syntaxe :
`ZipFile(file, mode='r', compression='ZIP_STORED', allowZip64=True)`.
Rappelons que `ZipFile` est une classe du module `zipfile`. Pour l'utiliser, il faudra donc l'importer comme suit :

```
1 from zipfile import ZipFile
```

ou sinon le préfixer avec `zipfile` en important seulement ce dernier.

```
1 import zipfile
```

L'argument `mode` devra être `'r'` pour lire un fichier existant, `'w'` pour créer (éventuellement écraser s'il existe déjà) un nouveau fichier `file`, `'a'` pour ajouter à un fichier existant, ou `'x'` pour une nouvelle création (exclusive) d'un nouveau fichier. A noter qu'avec le mode `'x'`, si le fichier existe déjà, une exception `FileExistError` est soulevée.
`compression` est la méthode de compression de `zip` utilisée lors de l'écriture de l'archive. Elle devra être choisie parmi les méthodes suivantes : `ZIP_STORED, ZIP_DEFLATED, ZIP_BZIP2, ou ZIP_LZMA`. Par défaut, c'est `ZIP_STORED` qui est utilisée.
Lorsque `allowZip64` est définie à `True` (valeur par défaut), `zipfile` pourra créer des archives `zip` qui utilisent les extensions `ZIP64` lorsque le `zipfile` pèse plus de 4Go. Notons que `ZipFile` peut s'utiliser aussi avec `with`.

Exemple 5.

```
1 from zipfile import ZipFile
2
3 with ZipFile('test.zip', 'w') as zpw:
4     zpw.write('test1.txt')
5     zpw.write('test2.txt')
```

En exécutant ce code, on crée une archive `zip` : `test.zip` qui contient nos deux fichiers `test1.txt` et `test2.txt`.

A noter que si on utilise pas `with`, il faudra fermé le `zipfile` correctement avec la commande `ZipFile.close()`.

Puisqu'une archive `zip` contient le plus souvent plusieurs fichiers, on listera d'abord l'ensemble des fichiers contenus dans l'archive avant de lire un fichier donné.

```
import zipfile
with zipfile.ZipFile("test.zip", 'r') as zp:
    zp.printdir()
    prem = zp.infolist()[0]
    with zp.open(prem) as f:
        text = f.read()
        print(text)
```

Dans cet exemple nous avons ouvert une archive `zip` en mode lecture.

`with zipfile.ZipFile(''test.zip'', 'r') as zp:`

Nous avons afficher tous les fichiers de l'archive

`zp.printdir()`

Pour accéder aux informations de tout fichier contenu dans l'archive, on utilise la fonction `infolist()` de notre archive qui renvoie une liste de tous les fichiers membres dans l'ordre qu'on les a archivés.
Ainsi l'indice [0] correspond au premier fichier, ainsi de suite :

`prem = zp.infolist()[0]`

Maintenant qu'on a accès à notre fichier, on peut le lire normalement :

```
    with zp.open(prem) as f:
        text = f.read()
        print(text)
```

Cependant si l'archive `zip` est protégé par mot de passe, nous ne pourrons pas accéder au contenu et on aura un message d'erreur. Nous allons voir sous peu, comment on pourrait contourner cela.

Extraction de fichiers d'un `zip`

Il y a principalement deux commandes :

`extract :` Sa syntaxe est la suivante :

Syntaxe : `ZipFile.extract(member, path=None, pwd=None)`.
Elle permet d'extraire un fichier membre de l'archive `zip` dans l'espace de travail courant.
`member` doit être son nom complet.
`path` spécifie un répertoire d'extraction différent.
`pwd` est le mot de passe utilisé pour les fichiers cryptés.
Cette fonction renvoie le fichier membre avec l'arborescence nécessaire (fichier ou répertoire).

`extractall :` Sa syntaxe est la suivante :

Syntaxe : `ZipFile.extractall(path=None, members=None, pwd=None)`.
Extrait tous les membres d'une archive `zip` dans le répertoire de travail courant.
`path` spécifie un répertoire d'extraction différent.
`members` est optionnel et doit être un sous-ensemble de la liste renvoyée par `ZipFile.namelist()` [1]
`pwd` est le mot de passe utilisé pour les fichiers cryptés.

1. renvoie la liste des membres de l'archive par leur nom

Exercice 8.

1. Si vous n'avez pas encore créé d'archive `zip`, faîtes-le maintenant (une archive contenant plusieurs fichiers). Choisissez maintenant un fichier de l'archive et extrayez-le dans un répertoire différent.
2. Cette fois-ci extrayez tous les fichiers d'un coup. Changez de répertoire d'extraction.

Archive `zip` protégée par mot de passe

`ZipFile.setpassword(pwd)` définit un mot de passe pour l'extraction des fichiers zippés. On l'utilise pour décrypter et non pour crypter. Pour crypter on fera appel à d'autres outils appropriés de Python par exemple (gnupg).

Attaque par force brute

Pour accéder aux fichiers d'une archive zip protégée par mot de passe, nous allons ici procéder à une attaque par force brute. Pour ce faire, nous avons besoin du fichier `words` que nous avions lu dans un exemple précédent

`/usr/share/dict/words`

ou tout autre dictionnaire pertinent que vous avez dans votre machine.

Exercice 9.

Supposons que notre archive `test.zip` est maintenant protégée par mot de passe, essayons de retrouver le mot de passe et afficher le premier fichier de l'archive.

```
1  import zipfile
2
3  import zlib
4
5  reference = "/usr/share/dict/words"
6
7  with zipfile.ZipFile("testeur.zip", "r") as zp:
8      prem = zp.infolist()[0]
9
10     with open(reference,'r') as f:
11         for l in f:
12             mdp = l.strip().encode("utf-8")
13             try:
14                 with zp.open(prem, 'r', pwd=mdp) as fp:
15                     text = fp.read()
16                 print("Mot de Passe", mdp)
17                 #print(text)
18                 break
19             except (RuntimeError, zlib.error, zipfile.BadZipFile):
20                 pass
```

Exercice 10.
Personnaliser l'exemple précédent pour extraire tous les fichiers d'un zip ou quelques fichiers.

Ligne de commande

Le module `zipfile` dispose d'une interface en ligne de commande pour manipuler des archives `zip`

```
python -m zipfile -c test.zip test1.txt test2.txt
```

L'option `-c` permet de créer l'archive `zip`. On donne d'abord le nom de l'archive `zip` à créer, ici `test.zip`, puis la liste de fichiers à inclure dans l'archive. A noter qu'on peut inclure des répertoires si on le souhaite.
Si on veut extraire une archive `zip` dans un répertoire `dossierZip` donné, on utilisera l'option `-e` comme suit :

```
python -m zipfile -e test.zip dossierZip/
```

Si l'on souhaite seulement lister une archive `zip`, on utilisera l'option `-l` :

```
python -m zipfile -l test.zip
```

On a l'option `-t` pour tester si une archive `zip` est valide ou non :

```
python -m zipfile -t test.zip
```

Archivage `tar`

Le module `tarfile` permet de créer et lire des archives `tar`. Il permet de traiter aussi les archives compressées avec `gzip`, `bzip2`, ou `lzma`, de traiter aussi les formats :

- `GNU tar` format par défaut
- `ustar` de `POSIX.1-1988`,
- `pax` de `POSIX.1-2001`.

Sa syntaxe est la suivante :

Syntaxe :

```
tarfile.open(name=None,mode='r',fileobj=None,bufsize=10240,**kwargs)
```

Il renvoie un objet de type `TarFile` pour le chemin `name`.
`mode` doit être sous la forme `mode[:compression]`. Les valeurs possibles sont les suivantes :

'r' ou 'r :*' overture en lecture avec une compression transparente

'r :' ouverture en lecture sans compression.

'r :gz' ouverture en lecture avec une compression `gzip`.

'r :bz2' ouverture en lecture avec une compression `bzip2`.

'r :xz' ouverture en lecture avec une compression `lzma`.

'x ou x :' Création exclusive d'une archive `tar` sans compression. Soulève une exception `FileExistsError` si le fichier existe déjà.

'x :gz' Création exclusive d'une archive `tar` avec une compression `gzip`. Soulève une `FileExistsError` si le fichier existe.

'x :bz2' Création exclusive d'une archive `tar` avec une compression `bzip2`. Soulève une `FileExistsError` si le fichier existe.

'x :xz' Création exclusive d'une archive `tar` avec une compression `lzma`. Soulève une `FileExistsError` si le fichier existe.

'a' ou 'a :' ouvre une archive `tar` en mode `ajout` sans compression. Le fichier est créé s'il n'existe pas.

'w' ou 'w :' ouvre une archive `tar` en mode `création` sans compression.

'w :gz' Création d'une archive `tar` avec une compression `gzip`.

'w :bz2' Création d'une archive `tar` avec une compression `bzip2`.

'w :xz' Création d'une archive `tar` avec une compression `lzma`.

Remarque.

- Notez que `'a:gz', 'a:xz', 'a:bz2'` n'existent pas.
- Si la lecture échoue avec un `mode` donné, une exception `ReadError` est soulevée.
- Si la méthode de compression renseignée n'est pas supportée, une exception `CompressionError` est soulevée.
- Avec les modes `'w:gz', 'r:gz', 'w:bz2', 'r:bz2', 'x:gz', 'x:bz2'`, `tarfile.open()` peut prendre en argument `compresslevel` (par défaut c'est 9) pour spécifier le niveau de compression du fichier.
- Notons que `tarfile.open` est une raccourci de `TarFile.open`

Quelques méthodes utiles

`TarFile.getmember(name)` **:** Renvoie un objet `TarInfo` contenant les informations du membre `name` (type de fichier, taille, dates, permissions, propriétaire, etc.)

`TarFile.getmembers()` **:** Renvoie la liste d'objets des membres de l'archive. A noter que cette liste conserve l'ordre de l'archive.

`TarFile.getnames()` **:** Renvoie la liste de noms des membres de l'archive. A noter que cette liste conserve aussi l'ordre de l'archive.

`TarFile.next()` **:** Renvoie le membre suivant des objets de l'archive lorsque cette dernière est ouverte en lecture. Elle renvoie `None` s'il y en a plus.

`TarFile.extractall(path='.', members=None,*, numeric_owner=False)` **:** Extraie tous les membres de l'archive dans le répertoire courant. Si l'argument optionnel `members` est renseigné, il doit être une partie de la liste renvoyée par `getmembers()`.

`TarFile.extract(member, path='', set_attr=True, * , numeric_owner=False)` **:** Extraie un membre de l'archive dans le répertoire courant en utilisant son nom complet. Les informations du fichiers sont extraites de manière aussi précise que possible. `member` peut être un nom de fichier ou un objet archive. On peut spécifier un répertoire d'extraction différent en utilisant `path`.

NB : Dans la plupart du temps on se contente de `extractall()`.

`TarFile.extractfile(member)` **:** Extraie un membre de l'archive en tant que objet fichier. `member` peut être un nom de fichier ou un objet archive (TarInfo).

`TarFile.add(name,arcname=None,recursive=True,exclude=None,*,filter=None)` **:**

`TarFile.addfile(tarinfo, fileobj=None)`**:**

`TarFile.close()`**:**

Exemple 6.

```
1 import tarfile
2 with tarfile.open('test/test.tar', 'w') as fw:
3     fw.add('concoursL1.pdf')
4     fw.add('Adroite.py')
```

On procède presque de la même façon que pour les fichiers classiques. Pour maintenant lire le fichier créé, on utlisera le paramètre `'r'`.

Exemple 7.

```
1 with tarfile.open('test/test.tar', 'r') as f:
2     f.extractall('test/resulat')
```

Remarque. L'archive crée l'arborescence complète des fichiers.
Attention à ne pas écraser des fichiers importants et à compromettre la sécurité du système.
Il est donc important de vérifier le contenu d'une archive avant de l'extraire, surtout si elle provient d'une source inconnue.

Pour obtenir les informations sur les chemins et les membres d'une archive `tar`, on procède comme suit :

```
1 with tarfile.open('test/test.tar', 'r') as f:
2     print(f.getnames())
3     print(f.getmembers())
```

Pour plus d'informations,

```
1 with tarfile.open('test/test.tar') as f:
2     f.list()
```

```
?rw-r--r-- udiouf/udiouf      89138 2018-05-25 20:11:11 concoursL1.pdf
?rw-rw-r-- udiouf/udiouf         67 2018-06-01 17:38:41 Adroite.py
```

On peut aussi faire :

```
1 with tarfile.open('test/test.tar') as f:
2     f.list(False)
```

```
concoursL1.pdf
Adroite.py
```

Ligne de commande

Nouveau avec la version 3.4.
Le module `tarfile` fournit une interface de ligne de commande simple pour interagir avec les archives `tar`.
Si l'on veut créer une nouvelle archive `tar`, on spécifie son nom après l'option `-c` puis on liste les fichiers à inclure dans l'archive.

```
python -m tarfile -c test.tar test1.txt test2.txt
```

A noter qu'on peut également lui fournir un répertoire.

```
python -m tarfile -c teste.tar Documents/
```

Si on veut extraire une archive dans le répertoire courant, on utilise l'option `-e`

```
python -m tarfile -e test.tar
```

Si on veut extraire une archive tar dans un répertoire différent :

```
python -m tarfile -e test.tar repertoire-different/
```

Voici quelques options utiles :

— l'option `-l` liste les fichiers dans l'archive
— l'option `-t` teste si l'archive est valide ou pas.

3.5.2 Compression

Nous allons décrire ici les formats de compression `zlib`, `gzip`, `bzip2` et `lzma`.

Le module `zlib`

Ce module permet de compresser et décompresser des données en utilisant la librairie `zlib`.
Pour lire et créer des fichiers `.gz` on utilisera surtout le module `gzip`.
Voici quelques fonctions utiles :

`zlib.adler32`**(data [,value**]) : Calcule la somme de contrôle (empreinte) ou `checksum` en anglais d'`Adler-32` des données. A noter qu'une empreinte `Adler-32` est presque aussi fiable que le `CRC32` mais surtout est plus rapide à calculer.
Le résultat est un entier de 32-bits non signé. Si l'option `value` est renseignée, elle est utilisée comme valeur initiale de la somme de contrôle. Dans le cas contraire l'initialisation est faite avec 1 par défaut.
Cet algorithme n'est pas fait pour des besoins cryptographique et ne devrait donc pas être utilisé pour de l'authentification ou de la signature numérique. Puisqu'elle est destinée surtout pour la somme de contrôle, il n'est pas recommendé en général comme algorithme de hachage.

`zlib.compress`**(data, level=-1) :** Compresse les octets dans `data` et retourne un objet d'octets contenant les données compressées.
`level` est un entier compris entre 0 et 9 ou -1 déterminant le niveau de compression :
— Une valeur de 1 (`Z_BEST_SPEED`) la plus rapide mais produit la pire compression.
— Une valeur de 9 (`Z_BEST_COMPRESSION`) est la plus lente mais fournit la meilleur compression.

— Une valeur de 0 (`Z_NO_COMPRESSION`) signifie sans compression.
— La valeur par défaut −1 (`Z_DEFAULT_COMPRESSION`) représente un compromis entre vitesse et qualité de compression (actuellement équivalent au niveau 6).

`zlib.decompress(data, wbits=MAX_WBITS, bufsize=DEF_BUF_SIZE)` : decompresse les octets dans data en renvoyant un objet d'octets contenant les données décompressées. Le paramètre `wbits` dépend du format de données `data`. Il contrôle la taille du buffer.
Si `bufsize` est renseigné, il est utilisé comme valeur initiale de la taille du flux de sortie.

Le module `gzip`

Ce module fournit une interface simple pour compresser et décompresser des fichiers comme on le feraient les programmes de `GNU` : `gzip` et `gunzip`. La compression de données est fournit par le module `zlib`.

Le module `gzip` fournit une classe `GzipFile` et aussi des fonctions très pratiques : `open(), compress()` et `decompress()`. Ce module permet de lire et de créer des fichiers au format `gzip` en les compressant ou décompressant à la volée de sorte que cela ressemble à une lecture ou écriture de fichiers ordinaires.

Voici quelques fonctions utiles :

`gzip.open(filename, mode='rb', compresslevel=9, encoding=None, newline=None)` :
Permet d'ouvrir un fichier compressé au format `gzip` en mode texte ou binaire en retournant un objet fichier.
`filename` peut être le fichier actuel (une chaîne de caractères ou objet binaire), ou un fichier objet existant à lire ou à écrire.
Le paramètre `mode` peut être parmi `'r', 'rb', 'a', 'ab', 'w', 'x', 'xb'` pour le mode binaire ou `'rt', 'at', 'wt', 'xt'` pour le mode texte. A noter que la valeur par défaut est `'rb'`.
Le paramètre `compresslevel` est un entier entre 0 et 9, comme pour le constructeur `GzipFile`.
NB : Les modes `'x', 'xb'` et `'xt'` ont été ajoutés à partir de la version 3.4.

`gzip.GzipFile(filename=None,mode=None,compresslevel=9,fileobj=None,mtime=None)` :
C'est le constructeur de la classe `GzipFile`. Au moins l'une des valeurs parmi `filename` et `fileobj` doit être renseignée. La nouvelle classe est basée sur le paramètre `fileobj` lequel peut être un fichier ordinaire, ou tout autre objet pouvant représenter un fichier. Sa valeur par défaut est `None` auquel cas, `filename` est ouvert pour fournir un fichier objet.
Lorsque `fileobj` n'est pas `None`, le paramètre `filename` est simplement utilisé pour être inclus dans l'entête du fichier `gzip` : ce qui peut inclure le nom du fichier original non compressé.
Le paramètre `compresslevel` est un entier de 0 à 9 contrôlant le niveau de compression ; 1 est le plus rapide mais de qualité moindre tandis que 9 est le plus lent mais la meilleure qualité possible. 0 représente pas de compression. La valeur par défaut est 9
Le paramètre `mtime` est un entier (timestamp) à mettre dans la date de dernière modification dans le flux lors de lacompression. Il doit seulement être fourni

en mode compression. S'il est omis ou est défini à `None`, l'"heure" courante est utilisée.

`gzip.compress(data, compresslevel=9)` : Compresse `data` en renvoyant un objet d'octets contenant les données compressées. Le paramètre `compresslevel` a la même signification que dans le module `GzipFile`.

`gzip.decompress(data)` : Décompresse `data` en renvoyant un objet d'octets contenant les données décompressées.

Exemples

Création d'un fichier compressé avec `gzip`

```
1  import gzip
2  contenu = b'Nam dui ligula, fringilla a, euismod sodales,
3  sollicitudin vel, wisi. Morbi auctor lorem non justo.
4  Nam lacus libero, pretium at, lobortis vitae, ultricies et,
5  tellus. Donec aliquet, tortor sed accumsan bibendum,
6  erat ligula aliquet magna, vitae ornare odio metus a mi.
7  Morbi ac orci et nisl hendrerit mollis.'
8
9
10 with gzip.open('/home/udiouf/Documents/test.txt.gz', 'wb') as fw:
11     fw.write(contenu)
```

Leture d'un fichier compressé avec `gzip`

```
1  with gzip.open('/home/udiouf/Documents/test.txt.gz', 'rb') as fr:
2      contenu = fr.read()
3      print(contenu)
```

Compresser un fichier existant avec `gzip`

```
1  import gzip
2  import shutil
3  with open('/home/udiouf/Documents/test.txt', 'rb') as f:
4      with gzip.open('/home/udiouf/test.txt.gz', 'wb') as fz:
5          shutil.copyfileobj(f, fz)
```

Compresser une chaîne de caratères binaire avec `gzip`

```
1  import gzip
2  cc = b'Voici du contenu binaire par exemple.'
3  cz = gzip.compress(cc)
```

3.5.3 `bzip2 - bz2`

Le module `bz2` fournit une interface de compression de données en utilisant l'algorithme de compression `bzip2`. Il propose un taux de compression meilleur que `gzip`, mais plus lent à la compression et à la décompression. Une alternative à `bz2` est `lzma` qui offre une compression encore meilleure, des temps plus courts, mais une consommation mémoire plus élevée.
Le module `bz2` contient les fonctions suivantes :

— La fonction `open()` et la classe `BZ2File` pour lire et écrire des fichiers compressés.
— Les fonctions `compress` et `decompress` pour compresser et décompresser.
— On notera la possibilité d'une compression et décompression incrémentale avec les fonctions `BZ2Compressor` et `BZ2Decompressor`.

Syntaxes :

```
bz2.open(filename,mode='r',compresslevel=9,encoding=None,errors=None,newline=None)
```

Exemple de création d'un fichier `.tar.bz2`

```
1 import tarfile
2 with tarfile.open('/home/udiouf/test.tar.bz2', 'w:bz2') as f:
3     f.add('/home/udiouf/test1.txt')
4     f.add('/home/udiouf/test2.txt')
```

3.5.4 `lzma`

A partir de la version 3.3 de python.
Son utilisation est très proche de celle de `bz2`.

Syntaxes :

```
lzma.open(filename,mode='rb',format=None, check=-1, preset=None, filters=None, encoding=
```

Exemple d'utilisation

Création d'un fichier compressé `.xz`

```
1 import lzma
2 with lzma.open('fichier.xz', 'w') as fw:
3     fw.write(donnees)
```

Lecture d'un fichier compressé

```
1 import lzma
2 with lzma.open('fichier.xz') as f:
3     contenu = f.read()
```

Compresser des données dans la mémoire

```
1 import lzma
2 entree = b'Ceci constitue des donnees en entree'
3 sortie = lzma.compress(entree)
```

Compression incrémentale

```
1 import lzma
2 lzc = lzma.LZMACompressor()
3 cc1 = lzc.compress(b'Ceci constitue des donnees\n')
4 cc2 = lzc.compress(b'Ceci constitue des donnees suplementaires\n')
5 cc3 = lzc.compress(b'Et Ceci constitue aussi des donnees\n')
6 cc1 = lzc.flush()
7
8 result = b''.join([cc1, cc2, cc3, cc4])
```

Ecrire des données compressées dans un fichier déjà ouvert

```
import lzma

with open('fichier.xz', 'wb') as f:
    f.write(b'Ces donnees ne seront pas compressees\n')
    with lzma.open(f, 'w') as lzw:
        lzw.write(b'Celles-ci seront par contre compressees\n')
    f.write(b'Et celles la non !\n')
```