4

L'utilisation des fichiers

Les fonctions prédéfinies

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'informations plus importante.

Pour que cela devienne possible, nous devrons doter nos programmes de divers mécanismes permettant de créer des fichiers, d'y envoyer des données et de les récupérer par la suite.

Les langages de programmation proposent des jeux d'instructions plus ou moins sophistiqués pour effectuer ces tâches.

Travailler avec des fichiers

L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver, puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations, mais généralement vous ne faites pas les deux à la fois. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de pages. Vous lisez la plupart des livres en suivant l'ordre normal des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre.

Tout ce que nous venons de dire des livres s'applique également aux fichiers informatiques. Un fichier se compose de données enregistrées sur votre disque dur, sur une disquette, une clef USB ou un CD. Vous y accédez grâce à son nom lequel peut inclure aussi un nom de répertoire. En première approximation, vous pouvez considérer le contenu d'un fichier comme une suite de caractères, ce qui signifie que vous pouvez traiter ce contenu, ou une partie quelconque de celui-ci, à l'aide des fonctions servant à traiter les chaînes de caractères.

Liste des modules importés

Le dictionnaire modules du module « **sys** » contient l'ensemble des modules importés.

```
>>> import sys
>>> for m in sys.modules:
        print(m, " " * (14 - len(str(m))),sys.modules[m])


builtins        <module 'builtins' (built-in)>
sys             <module 'sys' (built-in)>
...
```

Lorsque le programme stipule l'import d'un module, Python vérifie s'il n'est pas déjà présent dans cette liste. Dans le cas contraire, il l'importe. Chaque module n'est importé qu'une seule fois.

Le dictionnaire « **sys.modules** » peut être utilisé pour vérifier la présence d'un module ou lui assigner un autre identificateur. Un module est un objet qui n'autorise qu'une seule instance.

Les attributs communs à tout module

Une fois importés, tous les modules possèdent cinq attributs qui contiennent des informations comme leur nom, le chemin du fichier correspondant, l'aide associée.

__all__	Contient toutes les variables, fonctions, classes du module.
__builtins__	Ce dictionnaire contient toutes les fonctions et classes inhérentes au langage Python utilisées par le module.
__doc__	Contient l'aide associée au module.
__file__	Contient le nom du fichier qui définit le module. Son extension est pyc.
__name__	Cette variable contient a priori le nom du module sauf si le module est le point d'entrée du programme auquel cas cette variable contient « __main__ ».

Ces attributs sont accessibles si le nom du module est utilisé comme préfixe. Sans préfixe, ce sont ceux du module lui-même.

```
>>>
import os
>>> print(os.__name__)
os
>>> print(os.__doc__)
OS routines for NT or Posix depending on what system we're on.

This exports:
  - all functions from posix or nt, e.g. unlink, stat, etc.
  - os.path is either posixpath or ntpath
  - os.name is either 'posix' or 'nt'
  - os.curdir is a string representing the current directory (always '.')
  - os.pardir is a string representing the parent directory (always '..')
  - os.sep is the (or a most common) pathname separator ('/' or '\\')
  - os.extsep is the extension separator (always '.')
  - os.altsep is the alternate pathname separator (None or '/')
  - os.pathsep is the component separator used in $PATH etc
  - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
  - os.defpath is the default search path for executables
  - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).

>>> if __name__ == "__main__" :
        print("ce fichier est le point d'entrée")
else :
        print("ce fichier est importé")

ce fichier est le point d'entrée
```

Le répertoire courant

os

Le module os regroupe plusieurs fonctions ou objets qui sont dans certains cas des alias vers des éléments d'autres modules.

On peut regrouper ces éléments en quatre sous-ensembles :

- opérations sur les descripteurs de fichiers ;
- manipulation des fichiers et répertoires ;
- manipulation des processus ;
- informations sur le système.

Pour simplifier les explications qui vont suivre, nous indiquerons seulement des noms simples pour les fichiers que nous allons manipuler. Si vous procédez ainsi dans vos exercices, les fichiers en question seront créés et/ou recherchés par Python dans le répertoire courant. Si vous travaillez avec IDLE, vous souhaiterez donc certainement forcer Python à changer son répertoire courant, afin que celui-ci corresponde à vos attentes. Même si vous travaillez sous Windows, vous pouvez utiliser « **/** » et non « **** » en guise de séparateurs. Python effectuera automatiquement les conversions nécessaires, suivant que vous travaillez sous Mac OS, Linux, ou Windows.

```
>>> from os import chdir,getcwd
>>> chdir("F:/PYT - Python programmation objet/Scripts")
>>> rep_cour = getcwd()
>>> print(rep_cour)
F:\PYT - Python programmation objet\Scripts
```

La première commande importe la fonction « **chdir** » du module « **os** ». La seconde commande provoque le changement de répertoire « **chdir** ».

- Vous avez également la possibilité d'insérer ces commandes en début de script, ou encore d'indiquer le chemin d'accès complet dans le nom des fichiers que vous manipulez, mais cela risque peut-être d'alourdir l'écriture de vos programmes.
- Choisissez de préférence des noms de fichiers courts. Évitez dans toute la mesure du possible les caractères accentués, les espaces et les signes typographiques spéciaux. Dans les environnements de travail de type Unix (Mac OS, Linux, BSD...), il est souvent recommandé de n'utiliser que des caractères minuscules.

Les deux formes d'importation

Les lignes d'instructions que nous venons d'utiliser sont l'occasion d'expliquer un mécanisme intéressant. Vous savez qu'en complément des fonctions intégrées dans le module de base, Python met à votre disposition une très grande quantité de fonctions plus spécialisées, qui sont regroupées dans des modules.

Pour utiliser les fonctions d'un module, il suffit de les importer. Mais cela peut se faire de deux manières différentes, comme nous allons le voir ci-dessous. Chacune des deux méthodes présente des avantages et des inconvénients.

```
>>> import os
>>> rep_cour = os.getcwd()
>>> print(rep_cour)
F:\PYT - Python programmation objet\Scripts
```

Nous pouvons même importer toutes les fonctions d'un module, comme dans :

```
from tkinter import *
```

Cette méthode d'importation présente l'avantage d'alléger l'écriture du code. Elle présente l'inconvénient (surtout dans sa dernière forme, celle qui importe toutes les fonctions d'un module) d'encombrer l'espace de noms courant. Il se pourrait alors que certaines fonctions importées aient le même nom que celui d'une variable définie par vous-même, ou encore le même nom qu'une fonction importée depuis un autre module. Si cela se produit, l'un des deux noms en conflit n'est évidemment plus accessible.

L'écriture séquentielle

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un objet-interface particulier, que l'on appelle objet-fichier. On crée cet objet à l'aide de la fonction intégrée « **open** ». Celle-ci renvoie un objet doté de méthodes spécifiques, qui vous permettront de lire et écrire dans le fichier.

L'exemple ci-après vous montre comment ouvrir un fichier en écriture, y enregistrer deux chaînes de caractères, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé automatiquement. Par contre, si le nom utilisé concerne un fichier préexistant qui contient déjà des données, les caractères que vous y enregistrerez viendront s'ajouter à la suite de ceux qui s'y trouvent déjà. Vous pouvez faire tout cet exercice directement à la ligne de commande :

```
>>> import os
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> print(os.getcwd())
F:\PYT - Python programmation objet\Scripts
>>> obFichier = open('Monfichier.txt','at')
>>> obFichier.write('Bonjour, fichier !')
18
>>> obFichier.write("Quel beau temps, aujourd'hui !")
30
>>> obFichier.close()
>>> print(os.listdir(path='.'))
['Monfichier.txt']
>>> print(os.path.isfile("F:/PYT - Python programmation
objet/Scripts/Monfichier.txt"))
True
>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
    print('Le dossier actuel est ' + repN)
    for fichier in fichiers:
        print('Fichier -- ' + repN + ': '+ fichier)
    print('')

Le dossier actuel est F:\PYT - Python programmation objet\Scripts
Fichier -- F:\PYT - Python programmation objet\Scripts: Monfichier.txt
```

La fonction « **open** » attend deux arguments, qui doivent tous deux être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, et le second est le mode d'ouverture.

La méthode « **write** » réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Ces données sont enregistrées dans le fichier les unes à la suite des autres. Chaque nouvel appel de « **write** » continue l'écriture à la suite de ce qui est déjà enregistré.

La méthode « **close** » referme le fichier. Celui-ci est désormais disponible pour tout usage.

Mode	Signification
r	Ouverture du fichier en lecture.
w	Ouverture du fichier en écriture.
x	Ouverture du fichier en création exclusive.

a	Ouverture du fichier en mise à jour (append). Si le fichier n'existe pas, il est créé. S'il existe déjà, les nouvelles informations sont ajoutées à la fin.
+	Ouverture du fichier en lecture et en écriture.
t	Ouverture du fichier en lecture et en écriture. Si le fichier n'existe pas, il est créé. S'il existe déjà, son contenu est écrasé.
b	Ouverture du fichier en lecture et en mise à jour. Si le fichier n'existe pas, il est créé. S'il existe déjà, les nouvelles informations sont ajoutées à la fin.

Lecture séquentielle d'un fichier

Vous allez maintenant ouvrir le fichier, mais cette fois en lecture, de manière à pouvoir y relire les informations que vous avez enregistrées dans l'étape précédente :

```
>>> import os
>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
    print('Le dossier actuel est ' + repN)
    for fichier in fichiers:
        print('Fichier -- ' + repN + ': '+ fichier)
    print('')

Le dossier actuel est F:\PYT - Python programmation objet\Scripts
Fichier -- F:\PYT - Python programmation objet\Scripts: Monfichier.txt
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> def lireFichier (nomFichier):
        ofi = open(nomFichier, 'r')
        t = ofi.read()
        print(t)
        ofi.close()

>>> lireFichier('Monfichier.txt')
Bonjour, fichier !Quel beau temps, aujourd'hui !
```

Comme on pouvait s'y attendre, la méthode « **read** » lit les données présentes dans le fichier et les transfère dans une variable de type « **string** ». Si on utilise cette méthode sans argument, la totalité du fichier est transférée.

La méthode « **read** » peut également être utilisée avec un argument. Celui-ci indiquera combien de caractères doivent être lus, à partir de la position déjà atteinte dans le fichier.

```
>>> ofi = open('Monfichier.txt', 'r')
>>> t = ofi.read(7)
>>> print(t)
Bonjour
>>> print(ofi.read(15))
, fichier !Quel
>>> print(ofi.read(1000))
 beau temps, aujourd'hui !
```

Si la fin du fichier est déjà atteinte, « **read** » renvoie une chaîne vide.

```
>>> t = ofi.read()
>>> print(t)

>>> if len(t) <= 0 : ofi.close()
>>> ofi.closed
True
```

Dans tout ce qui précède, nous avons admis sans explication que les chaînes de caractères étaient échangées telles quelles entre l'interpréteur Python et le fichier. En réalité, ceci est inexact, parce que les séquences de caractères doivent être converties en séquences d'octets pour pouvoir être mémorisées dans les fichiers. De plus, il existe malheureusement différentes normes pour cela. En toute rigueur, il faudrait donc préciser à Python la norme d'encodage que vous souhaitez utiliser dans vos fichiers.

La copie d'un fichier

Il va de soi que les boucles de programmation s'imposent lorsque l'on doit traiter un fichier dont on ne connaît pas nécessairement le contenu à l'avance. L'idée de base consistera à lire ce fichier morceau par morceau, jusqu'à ce que l'on ait atteint la fin du fichier.

```
>>> def copieFichier(source, destination):
        "copie intégrale d'un fichier"
        fs = open(source, 'r')
        fd = open(destination, 'w')
        while 1:
            txt = fs.read(50)
            if txt =="":
                break
            fd.write(txt)
        fs.close()
        fd.close()
        return

>>> copieFichier('Monfichier.txt', 'MonfichierC.txt')
>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
    print('Le dossier actuel est ' + repN)
    for fichier in fichiers:
        print('Fichier -- ' + repN + ': '+ fichier)
    print('')

Le dossier actuel est F:\PYT - Python programmation objet\Scripts
Fichier -- F:\PYT - Python programmation objet\Scripts: Monfichier.txt
Fichier -- F:\PYT - Python programmation objet\Scripts: MonfichierC.txt
>>> lireFichier ('MonfichierC.txt')
Bonjour, fichier !Quel beau temps, aujourd'hui !
```

Vous savez en effet que l'instruction « **while** » doit toujours être suivie d'une condition à évaluer ; le bloc d'instructions qui suit est alors exécuté en boucle, aussi longtemps que cette condition reste vraie. Or nous avons remplacé ici la condition à évaluer par une simple constante, et vous savez également que l'interpréteur Python considère comme vraie toute valeur numérique différente de zéro. Nous pouvons cependant interrompre ce bouclage en faisant appel à l'instruction « **break** », laquelle permet éventuellement de mettre en place plusieurs mécanismes de sortie différents pour une même boucle.

Les fichiers texte

Un fichier texte est un fichier qui contient des caractères « **imprimables** » et des espaces organisés en lignes successives, ces lignes étant séparées les unes des autres par un caractère spécial non imprimable appelé marqueur de fin de ligne.

Les fichiers texte sont donc des fichiers que nous pouvons lire et comprendre à l'aide d'un simple éditeur de texte, par opposition aux fichiers binaires dont le contenu est – au moins en partie – inintelligible pour un lecteur humain, et qui ne prend son sens que lorsqu'il est décodé par un logiciel spécifique. Par exemple, les fichiers contenant des images, des sons, des vidéos, etc. sont presque toujours des fichiers binaires. Nous donnons un petit exemple de traitement de fichier binaire un peu plus loin, mais dans le cadre de ce cours, nous nous intéresserons presque exclusivement aux fichiers texte.

Il est très facile de traiter des fichiers texte avec Python. Par exemple, les instructions suivantes suffisent pour créer un fichier texte de quatre lignes :

```
>>> import os
>>> nomFichier='Fichiertexte'
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> f = open(nomFichier, "w")
>>> f.write("Ceci est la ligne un\nVoici la ligne deux\n")
41
>>> f.write("Voici la ligne trois\nVoici la ligne quatre\n")
43
>>> f.write("#Voici la ligne quatre\n")
43
>>> f.close()
>>> f = open(nomFichier, 'r')
>>> print(f.read())
Ceci est la ligne un
Voici la ligne deux
Voici la ligne trois
Voici la ligne quatre
#Voici la ligne quatre

>>> f.close()
```

Notez bien le marqueur de fin de ligne « **\n** » inséré dans les chaînes de caractères, aux endroits où l'on souhaite séparer les lignes de texte dans l'enregistrement.

Lors des opérations de lecture, les lignes d'un fichier texte peuvent être extraites séparément les unes des autres à l'aide de la méthode « **readline** ».

La méthode « **readlines** » transfère toutes les lignes restantes dans une liste de chaînes. Ainsi cette méthode 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.

```
>>> f = open(nomFichier, 'r')
>>> print(f.readlines())
['Ceci est la ligne un\n', 'Voici la ligne deux\n', 'Voici la ligne
trois\n', 'Voici la ligne quatre\n', '#Voici la ligne quatre\n']
>>> f.close()

>>> def filtre(source,destination):
```

```
        "recopier un fichier en"
        "éliminant les lignes de remarques"
        fs = open(source, 'r')
        fd = open(destination, 'w')
        while 1:
            txt = fs.readline()
            if txt =='':
                break
            if txt[0] != '#':
                fd.write(txt)
        fs.close()
        fd.close()
        return

>>> filtre(nomFichier,'Fichiertexte.txt')
>>> f = open('Fichiertexte.txt', 'r')
>>> print(f.read())
Ceci est la ligne un
Voici la ligne deux
Voici la ligne trois
Voici la ligne quatre

>>> f.close()
```

La fonction « **open** » renvois un objet de type file, qui contient les méthodes suivantes.

`close()`	ferme le flux.
`flush()`	vide le tampon interne.
`fileno()`	renvoie le descripteur de fichier.
`isatty()`	renvoie vrai si le fichier est branché sur un terminal tty.
`next()`	renvoie la prochaine ligne lue, ou provoque une exception StopIteration.
`read([size])`	lit au plus size octets. Si size est omis, lit tout le contenu.
`readline([size])`	lit la prochaine ligne. Si size est fourni, limite le nombre d'octets lus.
`readlines([sizehint])`	appelle « **readline** » en boucle, jusqu'à la fin du flux. Si « **sizehint** » est fourni, s'arrête lorsque ce nombre est atteint ou dépassé par la ligne en cours.
`seek(offset[,whence])`	positionne le curseur de lecteur en fonction de la valeur d'offset. whence permet de faire varier le fonctionnement (0 : position absolue – valeur par défaut, 1 : relative à la position courante, 2 : relative à la fin du fichier).
`tell()`	renvoie la position courante.
`truncate([size])`	tronque la taille du fichier. Si size est fourni, détermine la taille maximum.
`write(str)`	écrit la chaîne « **str** » dans le fichier.
`writelines(sequence)`	écrit la séquence de chaînes.

Les objets de type file sont des itérateurs, qui peuvent donc être utilisés directement comme des séquences.

```
>>> import os
>>> os.chdir("F:\PYT - Python programmation objet\Scripts")
```

```
>>> mon_fichier = open('infos.txt', mode='w', encoding='utf-8')
>>> mon_fichier.write('1. première info\n')
17
>>> mon_fichier.write('2. deuxième info\n')
17
>>> mon_fichier.write('3. troixième info\n')
18
>>> mon_fichier.write('4. 圖形碼常用字次常用字\n')
14
>>> mon_fichier.close()
>>> mon_fichier = open('infos.txt', 'r')
>>> for line in mon_fichier:
        print(line)

1. premiÃ¨re info
2. deuxiÃ¨me info
3. troixiÃ¨me info
4. åœ–å½¢ç¢¼å¸¸ç”¨å—ç¬¬å¸¸ç”¨å—

>>> mon_fichier.close()
>>> print(mon_fichier.name,\
      mon_fichier.encoding, \
      mon_fichier.newlines.encode(), \
      mon_fichier.closed)
infos.txt cp1252 b'\r\n' True
>>> mon_fichier = open('infos.txt', mode='r', encoding='utf-8')
>>> for line in mon_fichier:
        print(line)

1. première info
2. deuxième info
3. troixième info
4. 圖形碼常用字次常用字

>>> mon_fichier.close()
>>> print(mon_fichier.name,\
      mon_fichier.encoding, \
      mon_fichier.newlines.encode(), \
      mon_fichier.closed)
infos.txt utf-8 b'\r\n' True
```

Le type file possède en outre un certain nombre d'attributs.

`Closed`	renvoie vrai si le fichier a été fermé.
`Encoding`	renvoie l'encoding utilisé par le fichier pour l'écriture. Si des chaînes unicode sont écrites dans le flux, elles sont encodées avec ce codec.
`Name`	renvoie le nom du fichier.
`Newlines`	renvoie le type de passage à la ligne utilisé (\r, \n, ou \r\n)

L'enregistrement et restitution de variables

L'argument de la méthode « **write** » utilisée avec un fichier texte doit être une chaîne de caractères. Avec ce que nous avons appris jusqu'à présent, nous ne pouvons donc enregistrer d'autres types de valeurs qu'en les transformant d'abord en chaînes de caractères « **string** ». Nous pouvons réaliser cela à l'aide de la fonction intégrée « **str** ».

```
>>> import os
>>> nomFichier='FichierFormate.txt'
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> f = open(nomFichier, 'w')
>>> x,d,s = 5.5,7,"caractères"
>>> res = "un nombre réel %f et un entier %d, une chaîne de %s, \n"\
         "un réel d'abord converti en  chaîne de  caractères %s\n" \
         %  (x,d,s, str(x+4))
>>> f.write(res)
121
>>> res = "un nombre réel " + str (x) + \
          " et un entier " + str (d) + \
          ", une chaîne de " + s + \
          ",\nun réel d'abord converti en chaîne de caractères "\
          + str(x+4)
>>> f.write(res)
114
>>> f.close()
>>> f = open(nomFichier, 'r')
>>> print(f.read())
un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
un réel d'abord converti en  chaîne de  caractères 9.5
un nombre réel 5.5 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
>>> f.close()
```

Si nous enregistrons les valeurs numériques en les transformant d'abord en chaînes de caractères, nous risquons de ne plus pouvoir les retransformer correctement en valeurs numériques lorsque nous allons relire le fichier.

Il existe plusieurs solutions à ce genre de problèmes. L'une des meilleures consiste à importer un module Python spécialisé : le module « **pickle** ».

```
>>> import pickle
>>> a, b, c = 27, 12.96, [5, 4.83, "René"]
>>> f =open('donnees_binaires', 'wb')
>>> pickle.dump(a, f)
>>> pickle.dump(b, f)
>>> pickle.dump(c, f)
>>> f.close()
>>> f =open('donnees_binaires', 'rb')
>>> j = pickle.load(f)
>>> k = pickle.load(f)
>>> l = pickle.load(f)
>>> print(j, type(j))
27 <class 'int'>
>>> print(k, type(k))
```

```
12.96 <class 'float'>
>>> print(l, type(l))
[5, 4.83, 'René'] <class 'list'>
```

Comme vous pouvez le constater dans ce court exemple, le module « **pickle** » permet d'enregistrer des données avec conservation de leur type.

Attention, les fichiers traités à l'aide des fonctions du module « **pickle** » ne seront pas des fichiers texte, mais bien des fichiers binaires. Pour cette raison, ils doivent obligatoirement être ouverts comme tels à l'aide de la fonction « **open** ». Vous utiliserez l'argument « **wb** » pour ouvrir un fichier binaire en, et l'argument « **rb** » pour ouvrir un fichier binaire en lecture.

La fonction « **dump** » du module « **pickle** » attend deux arguments : le premier est la variable à enregistrer, le second est l'objet fichier dans lequel on travaille. La fonction « **load** » effectue le travail inverse, c'est-à-dire la restitution de chaque variable avec son type.

De la même manière le module « **shelve** » vous permettra de écrire d'ajouter ou lire les données sauvegardées.

```
>>> import shelve
>>> shelfFile = shelve.open('shelveFichier')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
>>> shelfFile = shelve.open('shelveFichier')
>>> print(shelfFile['cats'],type(shelfFile))
['Zophie', 'Pooka', 'Simon'] <class 'shelve.DbfilenameShelf'>
>>> shelfFile.close()
>>> shelfFile = shelve.open('shelveFichier')
>>> fruits = ['apples', 'oranges', 'cherries', 'banana']
>>> pers = ['Alice', 'Bob', 'Carol', 'David']
>>> animaux = ['dogs', 'cats', 'moose', 'goose']
>>> shelfFile['fruits'] = fruits
>>> shelfFile['pers'] = pers
>>> shelfFile['animaux'] = animaux
>>> shelfFile.close()
>>> shelfFile = shelve.open('shelveFichier')
>>> list(shelfFile.keys())
['cats', 'fruits', 'pers', 'animaux']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon'], ['apples', 'oranges', 'cherries',
'banana'], ['Alice', 'Bob', 'Carol', 'David'], ['dogs', 'cats',
'moose', 'goose']]
>>> for categorie in shelfFile.keys() :
    print(shelfFile[categorie],type(shelfFile[categorie]))

['Zophie', 'Pooka', 'Simon'] <class 'list'>
['apples', 'oranges', 'cherries', 'banana'] <class 'list'>
['Alice', 'Bob', 'Carol', 'David'] <class 'list'>
['dogs', 'cats', 'moose', 'goose'] <class 'list'>
```

Après avoir exécuté le code précédent sur Windows, vous verrez trois nouveaux fichiers dans le répertoire de travail courant: shelveFichier.bak, shelveFichier.dat et shelveFichier.dir.

Les exceptions try – except – else

Les exceptions sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue, et un message d'erreur plus ou moins explicite est affiché.

```
>>> print(10/0)
Traceback (most recent call last):
  File "<pyshell#340>", line 1, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le type d'erreur, et ensuite une information spécifique de cette erreur.

Dans de nombreux cas, il est possible de prévoir à l'avance certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible d'associer un mécanisme de surveillance à tout un ensemble d'instructions, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.

Un mécanisme de ce type s'appelle en général mécanisme de traitement des exceptions. Celui de Python utilise l'ensemble d'instructions « **try - except - else** », qui permettent d'intercepter une erreur et d'exécuter une portion de script spécifique de cette erreur. Il fonctionne comme suit.

Le bloc d'instructions qui suit directement une instruction « **try** » est exécuté par Python sous réserve. Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction « **except** ». Si aucune erreur ne s'est produite dans les instructions qui suivent « **try** », alors c'est le bloc qui suit l'instruction « **else** » qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

```
>>> try:
        print(10/0)
    except:
        print("La division par zéro n'est pas acceptée")

La division par zéro n'est pas acceptée
```

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, lequel fichier étant destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se plante. Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
>>> def existe(fname):
        try:
            f = open(fname,'r')
            f.close()
            return 1
        except:
            return 0
```

```
>>> filename = input("Veuillez entrer le nom du fichier : ")
Veuillez entrer le nom du fichier : Il n existe pas
>>> if existe(filename):
        print("Ce fichier existe bel et bien.")
else:
        print("Le fichier-", filename, "-est introuvable.")

Le fichier-Il n existe pas-est introuvable.
```

Il est également possible de faire suivre l'instruction « **try** » de plusieurs blocs « **except** », chacun d'entre eux traitant un type d'erreur spécifique.

Il est aussi possible d'ajouter une clause qui sert de préfixe à une liste d'instructions qui ne sera exécutée que si aucune exception n'est déclenchée.

```
try:
# ... instructions à protéger
except:
# ... que faire en cas d'erreur
else:
# ... que faire lorsque aucune erreur n'est apparue
```

```
>>> def inverse(x):
...     y = 1.0 / x
...     return y
...
>>> try:
...     print(inverse(2)) # pas d'erreur
... except:
...     print("le programme a déclenché une erreur")
... else:
...     print("tout s'est bien passé")
...
0.5
tout s'est bien passé
```

Le type d'exception

Parfois, plusieurs types d'erreurs peuvent être déclenchés à l'intérieur d'une portion de code protégée. Pour avoir une information sur ce type, il est possible de récupérer une variable de type « **Exception** ».

```
>>> def inverse(x):
...     return 1.0 / x
...
>>> try:
...     print(inverse(2))
...     print(inverse(0))
... except Exception as exc:
...     print("exception de type ", exc.__class__)
...     # affiche exception de type exceptions.ZeroDivisionError
...     print("message", exc)
...     # affiche le message associé à l'exception
...
0.5
exception de type  <class 'ZeroDivisionError'>
message float division by zero
```

La variable « **exc** » est en fait une instance d'une classe d'erreur, « **__class__** » correspond au nom de cette classe. A l'aide de la fonction « **isinstance** », il est possible d'exécuter des traitements différents selon le type d'erreur.

```
>>> try:
...     print((-2.1) ** 3.1) # première erreur
...     print(inverse(2))
...     print(inverse(0)) # seconde erreur
... except Exception as exc:
...     if isinstance(exc, ZeroDivisionError):
...         print("division par zéro")
...     else:
...         print("erreur insoupçonnée :", exc.__class__)
...         print("message", exc)
...
(-9.48606594010979-3.0822096637057887j)
0.5
division par zéro
```

Une autre syntaxe plus simple permet d'attraper un type d'exception donné en accolant au mot-clé « **except** » le type de l'exception qu'on désire attraper.

```
try:
    # ... instructions à protéger
except type_exception_1:
    # que faire en cas d'erreur de
    #      type type_exception_1
except (type_exception_i, type_exception_j):
    # que faire en cas d'erreur de type
    #       type_exception_i ou type_exception_j
```

```
        except type_exception_n:
            # que faire en cas d'erreur de
            #      type type_exception_n
        except:
            # que faire en cas d'erreur d'un type
            #       différent de tous les précédents types
        else:
            # que faire lorsqu'aucune erreur n'est apparue
```

```
>>> try:
...     print((-2.1) ** 3.1)
...     print(inverse(0))
... except ZeroDivisionError:
...     print("division par zéro")
... except Exception as exc:
...     print("erreur insoupçonnée :", exc.__class__)
...     print("message ", exc)
...
(-9.48606594010979-3.0822096637057887j)
division par zéro
```

Les d'exceptions standards

Le langage Python propose une liste d'exceptions standards. Lorsqu'une erreur ne correspond pas à l'une de ces exceptions, il est possible de créer une exception propre à un certain type d'erreur. Lorsqu'une fonction ou une méthode déclenche une exception non standard, généralement, le commentaire qui lui est associé l'indique. Voici quelques types d'exception courantes :

Nom Erreur	Description
AttributeError	Une référence à un attribut inexistant ou une affectation a échoué.
OSError	Une opération concernant les entrées/sorties (Input/Output) a échoué. Cette erreur survient par exemple lorsqu'on cherche à lire un fichier qui n'existe pas.
ImportError	Cette erreur survient lorsqu'on cherche à importer un module qui n'existe pas.
IndentationError	L'interpréteur ne peut interpréter une partie du programme à cause d'un problème d'indentation. Il n'est pas possible d'exécuter un programme mal indenté mais cette erreur peut se produire lors de l'utilisation de la fonction compile.
IndexError	On utilise un index erroné pour accéder à un élément d'une liste, d'un dictionnaire ou de tout autre tableau.
KeyError	Une clé est utilisée pour accéder à un élément d'un dictionnaire dont elle ne fait pas partie.
NameError	On utilise une variable, une fonction, une classe qui n'existe pas.
TypeError	Erreur de type, une fonction est appliquée sur un objet qu'elle n'est pas censée manipuler.
UnicodeError	Erreur de conversion d'un encodage de texte à un autre.
ValueError	Cette exception survient lorsqu'une valeur est inappropriée pour une certaine opération, par exemple, l'obtention du logarithme d'un nombre négatif.

```
>>> try:
...     d = ["un", "deux"]
...     print(d[2])
...     print(inverse(0))
... except IndexError:
...     print("index erroné")
... except ZeroDivisionError:
...     print("division par zéro")
... except Exception as exc:
...     print("erreur insoupçonnée :", exc.__class__)
...
index erroné
```

Lancer une exception

Lorsqu'une fonction détecte une erreur, il lui est possible de déclencher une exception par l'intermédiaire du mot-clé « **raise** ». Cette exception est attrapée plus bas.

Il est parfois utile d'associer un message à une exception afin que l'utilisateur ne soit pas perdu. Le déclenchement d'une exception suit la syntaxe suivante :

```
raise exception_type(message)
```

```
>>> def inverse(x):
...     if x == 0:
...         raise ValueError(
...             "valeur nulle interdite, fonction inverse")
...     y = 1.0 / x
...     return y
...
>>> try:
...     print(inverse(0)) # erreur
... except ValueError as exc:
...     print("erreur, message :", exc)
...
erreur, message : valeur nulle interdite, fonction inverse
```

Les instructions try, except imbriquées

Comme pour les boucles, il est possible d'imbriquer les portions protégées de code les unes dans les autres. Dans l'exemple qui suit, la première erreur est l'appel à une fonction non définie, ce qui déclenche l'exception NameError.

```
>>> def inverse(x):
...     return 1.0 / x
...
>>> try:
...     try:
...         print(inverses(0))
...                # fonction inexistante --> exception NameError
...         print(inverse(0))
...                # division par zéro --> ZeroDivisionError
...     except NameError:
...         print("appel à une fonction non définie")
... except ZeroDivisionError as exc:
...     print("erreur", exc)
...
appel à une fonction non définie

>>> try:
...     try:
...         print(inverse(0))
...               # division par zéro --> ZeroDivisionError
...         print(inverses(0))
...               # fonction inexistante --> exception NameError
...     except NameError:
...         print("appel à une fonction non définie")
... except ZeroDivisionError as exc:
...     print("erreur", exc)
...
erreur float division by zero
```

Une autre imbrication possible est l'appel à une fonction qui inclut déjà une partie de code protégée.

```
>>> def inverse(x):
...     try:
...         y = 1.0 / x
...     except ZeroDivisionError as exc:
...         print("erreur ", exc)
...         if x > 0:
...             return 1000000000
...         else:
...             return -1000000000
...     return y
...
>>> try:
...     print(inverse(0))
...           # division par zéro --> la fonction inverse sait gérer
...     print(inverses(0))
...           # fonction inexistante --> exception NameError
```

```
... except NameError:
...     print("appel à une fonction non définie")
...
erreur  float division by zero
-1000000000
appel à une fonction non définie
```

Les fichiers compressés

Les fichiers « **zip** » ou « **gzip** » sont très répandus de nos jours et constituent un standard de compression facile d'accès quelque soit l'ordinateur et son système d'exploitation. Le langage Python propose quelques fonctions pour compresser et décompresser ces fichiers par l'intermédiaire du module « **zipfile** » respectif « **gzip** ».

Création d'une archive

Pour créer un fichier « **zip** », le procédé ressemble à la création de n'importe quel fichier. La seule différence provient du fait qu'il est possible de stocker le fichier à compresser sous un autre nom à l'intérieur du fichier « **zip** », ce qui explique les deux premiers arguments de la méthode « **write** ». Le troisième paramètre indique si le fichier doit être compressé « **zipfile.ZIP_DEFLATED** » ou non « **zipfile.ZIP_STORED** ».

```
>>> import os
>>> import zipfile
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> os.mkdir("F:/PYT - Python programmation objet/repCompress")
>>> def compressRepertoire(nomRepertoire) :
        file = zipfile.ZipFile (\
        "F:/PYT - Python programmation
objet/repCompress/compressRep.zip", "w")
        for repN, susRep, fichiers in os.walk(nomRepertoire):
            for fichier in fichiers:
                file.write (fichier,fichier,\
                    zipfile.ZIP_DEFLATED)
        file.close()

>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
        print('Le dossier actuel est ' + repN)
        for fichier in fichiers:
            print('Fichier -- ' + repN + ': '+ fichier)
        print('')

Le dossier actuel est F:\PYT - Python programmation objet\Scripts
Fichier -- F:\PYT - Python programmation objet\Scripts: Fichiertexte
Fichier -- F:\PYT - Python programmation objet\Scripts: Fichiertexte.txt
Fichier -- F:\PYT - Python programmation objet\Scripts: Monfichier.txt
Fichier -- F:\PYT - Python programmation objet\Scripts: MonfichierC.txt

>>> compressRepertoire(os.getcwd())
>>> os.chdir("F:/PYT - Python programmation objet/repCompress")
>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
        print('Le dossier actuel est ' + repN)
        for fichier in fichiers:
            print('Fichier -- ' + repN + ': '+ fichier)
        print('')

Le dossier actuel est F:\PYT - Python programmation objet\repCompress
Fichier -- F:\PYT - Python programmation objet\repCompress: compressRep.zip
```

Lecture d'une archive

L'exemple suivant permet par exemple d'obtenir la liste des fichiers inclus dans un fichier zip

```
>>> import os
>>> import zipfile
>>> os.chdir("F:/PYT - Python programmation objet/repCompress")
>>> file = zipfile.ZipFile ("compressRep.zip", "r")
>>> for info in file.infolist () :
    print(info.filename, info.date_time, info.file_size)

Fichiertexte (2017, 3, 25, 16, 16, 54) 112
Fichiertexte.txt (2017, 3, 25, 16, 22, 16) 88
Monfichier.txt (2017, 3, 25, 15, 29, 54) 48
MonfichierC.txt (2017, 3, 25, 15, 53, 8) 48
>>> file.close()
>>> def lireFichier(nomFichier,archive) :
        file = zipfile.ZipFile (archive, "r")
        print(file.read(nomFichier).decode())
        file.close()

>>> lireFichier("Fichiertexte","compressRep.zip")
Ceci est la ligne un
Voici la ligne deux
Voici la ligne trois
Voici la ligne quatre
#Voici la ligne quatre
```

La manipulation de fichiers

Le module « **os.path** » propose plusieurs fonctions très utiles qui permettent entre autres de tester l'existence d'un fichier, d'un répertoire, de récupérer diverses informations comme sa date de création, sa taille…

abspath(path)	Retourne le chemin absolu d'un fichier ou d'un répertoire.
commonprefix(list)	Retourne le plus grand préfixe commun à un ensemble de chemins.
dirname(path)	Retourne le nom du répertoire.
exists(path)	Dit si un chemin est valide ou non.
getatime(path) getmtime(path) getctime(path)	Retourne diverses dates concernant un chemin, date du dernier accès (getatime), date de la dernière modification (getmtime), date de création (getctime).
getsize(file)	Retourne la taille d'un fichier.
isabs(path)	Retourne True si le chemin est un chemin absolu.
isfile(path)	Retourne True si le chemin fait référence à un fichier.
isdir(path)	Retourne True si le chemin fait référence à un répertoire.
join(p1, p2, ...)	Construit un nom de chemin étant donné une liste de répertoires.
split(path)	Découpe un chemin, isole le nom du fichier ou le dernier répertoire des autres répertoires.
splitext(path)	Découpe un chemin en nom + extension.

La liste des fichiers

La fonction « **listdir** » du module « **os** » permet de retourner les listes des éléments inclus dans un répertoire (fichiers et sous-répertoires). Toutefois, le module « **glob** » propose une fonction plus intéressante qui permet de retourner la liste des éléments d'un répertoire en appliquant un filtre.

```
>>> def liste_fichier_repertoire (folder, filter):
        file,fold = [], []
        res = glob.glob (folder + "\\" + filter)
        rep = glob.glob (folder + "\\*")
        for r in rep :
            if r not in res and os.path.isdir (r) :
                res.append (r)
        for r in res :
            path = r
            if os.path.isfile (path) :
                file.append (path)
            else :
                fi,fo = liste_fichier_repertoire  (path,  filter)
                file.extend (fi)
                fold.extend (fo)
        return file,fold
```

```
>>> import os
>>> os.chdir("F:\PYT - Python programmation objet\Scripts")
>>> folder = os.getcwd()
>>> filter =  "*.txt"
>>> import glob
>>> file,fold =  liste_fichier_repertoire  (folder, filter)
>>> for f in file : print("fichier ", f)

fichier  F:\PYT - Python programmation objet\Scripts\FichierB.txt
fichier  F:\PYT - Python programmation objet\Scripts\FichierFormate.txt
fichier  F:\PYT - Python programmation objet\Scripts\Fichiertexte.txt
fichier  F:\PYT - Python programmation objet\Scripts\Monfichier.txt
fichier  F:\PYT - Python programmation objet\Scripts\MonfichierC.txt
fichier  F:\PYT - Python programmation
objet\Scripts\repCompress\Monfichier.txt
>>> for f in fold : print("répertoire ", f )
```

Liste non exhaustive des fonctionnalités offertes par les modules shutil et os.

`copy(f1,f2)`	« **shutil** » Copie le fichier f1 vers f2
`chdir(p)`	« **os** » Change le répertoire courant, cette fonction peut être importante lorsqu'on utilise la fonction system du module « **os** » pour lancer une instruction en ligne de commande ou lorsqu'on écrit un fichier sans préciser le nom du répertoire.
`getcwd()`	« **os** » Retourne le répertoire courant.
`mkdir(p)`	« **os** » Crée le répertoire p.
`makedirs(p)`	« **os** » Crée le répertoire p et tous les répertoires des niveaux supérieurs s'ils n'existent pas.
`remove(f)`	« **os** » Supprime un fichier.
`rename(f1,f2)`	« **os** » Renomme un fichier
`rmdir(p)`	« **os** » Supprime un répertoire

Les expressions régulières

Les expressions régulières sont prises en charge par le module « **re** ». Ainsi, comme avec tous les modules en Python, nous avons seulement besoin de l'importer pour commencer à les utiliser.

Même si les expressions régulières ne sont pas propres à un langage, chaque implémentation introduit généralement des spécificités pour leur notation. L'antislash « **** » tient un rôle particulier dans la syntaxe des expressions régulières puisqu'il permet d'introduire des caractères spéciaux. Comme il est également interprété dans les chaînes de caractères, il est nécessaire de le doubler pour ne pas le perdre dans l'expression.

```
>>> expression = "\btest\b"
>>> print(expression)
test
>>> expression = "\\btest\\b"
>>> print(expression)
\btest\b
>>> expression = r"\btest\b"
>>> print(expression)
\btest\b
```

Les chaînes de caractères peuvent éventuellement être précédées d'une lettre « **r** » ou « **R** » Ces chaînes sont appelées chaînes brutes et traitent l'antislash « **** » comme un caractère littéral.

La syntaxe des expressions régulières

La syntaxe des expressions régulières peut se regrouper en trois groupes de symboles :

- les symboles simples ;
- les symboles de répétition ;
- les symboles de regroupement.

Les symboles simples

Les symboles simples sont des caractères spéciaux qui permettent de définir des règles de capture pour un caractère du texte et sont réunis dans le tableau ci-dessous.

Symbole	Fonction
.	Remplace tout caractère sauf le saut de ligne.
^	Symbolise le début d'une ligne.
$	Symbolise la fin d'une ligne.
\A	Symbolise le début de la chaîne.
\b	Symbolise le caractère d'espacement. Intercepté seulement au début ou à la fin d'un mot. Un mot est ici une séquence de caractères alphanumériques ou espace souligné.
\B	Comme « **\b** » mais uniquement lorsque ce caractère n'est pas au début ou à la fin d'un mot.
\d	Intercepte tout chiffre.
\D	Intercepte tout caractère sauf les chiffres.
\s	Intercepte tout caractère d'espacement : horizontale « **\t** », verticale « **\v** », saut de ligne « **\n** », retour à la ligne « **\r** », form feed « **\f** ».
\S	Symbole inverse de \s
\w	Intercepte tout caractère alphanumérique et espace souligné.
\W	Symbole inverse de « **\w** ».
\Z	Symbolise la fin de la chaîne.

Voici quelques exemples :

```
>>> import re
>>> re.findall(r'.', ' test *')
[' ', 't', 'e', 's', 't', ' ', '*']
>>> re.findall(r'.', 'test\n')
['t', 'e', 's', 't']
>>> re.findall(r'.', '\n')
[]
>>> re.findall(r'^le', "c'est le début")
[]
>>> re.findall(r'^le', "le début")
['le']
>>> re.findall(r'mot$', 'mot mot mot')
['mot']
>>> re.findall(r'\Aparoles', 'paroles, paroles, paroles,\nparoles,
encore des paroooooles')
['paroles']
>>> re.findall(r'\bpar\b', 'parfaitement')
[]
>>> re.findall(r'\bpar\b', 'par monts et par veaux')
['par', 'par']
>>> re.findall(r'\Bpar\B', "imparfait")
['par']
```

```
>>> re.findall(r'\Bpar\B', "parfait")
[]
>>> re.findall(r'\d', '1, 2, 3, nous irons au bois (à 12:15h)')
['1', '2', '3', '1', '2', '1', '5']
>>> print(''.join(re.findall(r'\D', '1, 2, 3, nous irons au bois (à
12:15h)')))
, , , nous irons au bois (à :h)
>>> len(re.findall(r'\s', "combien d'espaces dans la phrase ?"))
5
>>> len(re.findall(r'\s', "latoucheespaceestbloquée"))
0
>>> phrase = """Lancez vous!"""
>>> len(re.findall(r'\s', phrase))
1
>>> len(re.findall(r'\S', "combien de lettres dans la phrase ?"))
29
>>> ''.join(re.findall(r'\w', '*!mot-clé_*'))
'motclé_'
>>> ''.join(re.findall(r'\W', '*!mot-clé_*'))
'*!-*'
>>> re.findall(r'end\Z', 'The end will come')
[]
>>> re.findall(r'end\Z', 'This is the end')
['end']
```

Le fonctionnement de chacun de ces symboles est affecté par les options suivantes :

- **(A)SCII** : les symboles « **\w** », « **\W** », « **\b** », « **\B** », « **\d** », « **\D** », « **\s** » et « **\S** » se basent uniquement sur le code ascii.
- **S** ou **DOTALL** : le saut de ligne est également intercepté par le symbole « **\b** ».
- **(M)ULTILINE** : dans ce mode, les symboles « **^** » et « **$** » interceptent le début et la fin de chaque ligne.
- **(U)NICODE** : les symboles « **\w** », « **\W** », « **\b** », « **\B** », « **\d** », « **\D** », « **\s** » et « **\S** » se basent sur de l'unicode.
- **(I)GNORECASE** : rend les symboles insensibles à la casse du texte.
- **X** ou **VERBOSE** : autorise l'insertion d'espaces et de commentaires en fin de ligne, pour une mise en page de l'expression régulière plus lisible.

```
>>> re.findall('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['a', 'B']
>>> chars = ''.join(chr(i) for i in range(256))
>>> " ".join(re.findall(r"\w", chars))              <--------UNICODE
'0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X
Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z ª ² ³ µ ¹
º ¼ ½ ¾ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ
ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ'
>>> " ".join(re.findall(r"\w", chars, flags=re.UNICODE))
'0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X
Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z ª ² ³ µ ¹
º ¼ ½ ¾ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ
ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ'
>>> " ".join(re.findall(r"\w", chars, flags=re.ASCII))
'0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X
Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z'
```

```
>>> re.findall(r"\w+", "这是一个 例子 是一", re.UNICODE)
['这是一个', '例子', '是一']
>>> re.findall(r"\w+", "这是一个 例子 是一")
['这是一个', '例子', '是一']
```

Les symboles de répétition

Les symboles simples peuvent être combinés et répétés par le biais de symboles de répétition.

Symbole	Fonction
`*`	Répète le symbole précédent de 0 à n fois (autant que possible).
`+`	Répète le symbole précédent de 1 à n fois (autant que possible).
`?`	Répète le symbole précédent 0 ou 1 fois (autant que possible).
`{n}`	Répète le symbole précédent n fois.
`{n,m}`	Répète le symbole précédent entre n et m fois inclus. n ou m peuvent être omis comme pour les tranches de séquences. Dans ce cas ils sont remplacés respectivement par 0 et *.
`{n,m}?`	Équivalent à {n,m} mais intercepte le nombre minimum de caractères.
`e1|e2`	Intercepte l'expression e1 ou e2. (OR)
`[]`	Regroupe des symboles et caractères en un jeu.

Voici quelques exemples :

```
>>> import re
>>> re.findall(r'pois*', 'poisson pois poilant poi')
['poiss', 'pois', 'poi', 'poi']
>>> re.findall(r'pois+', 'poisson pois poilant poi')
['poiss', 'pois']
>>> re.findall(r'pois?', 'poisson pois poilant poi')
['pois', 'pois', 'poi', 'poi']
>>> re.findall(r'pois{2}', 'poisson pois poilant poi')
['poiss']
>>> re.findall(r'pois{2,4}', 'poisssssssssssson pois poilant poi')
['poissss']
>>> re.findall(r'pois{,4}', 'poisssssssssssson pois poilant poi')
['poissss', 'pois', 'poi', 'poi']
>>> re.findall(r'pois{2,}', 'poisssssssssssson pois poilant poi')
['poissssssssssss']
>>> re.findall(r'pois{2,4}?', 'poisssssssssssson pois poilant poi')
['poiss']
>>> re.findall(r'pois{2,}?', 'poisssssssssssson pois poilant poi')
['poiss']
>>> re.findall(r'pois{,4}?', 'poisssssssssssson pois poilant poi')
['poi', 'poi', 'poi', 'poi']
>>> re.findall(r'Mr|Mme', 'Mr et Mme')
['Mr', 'Mme']
>>> re.findall(r'Mr|Mme', 'Mr Untel')
['Mr']
>>> re.findall(r'Mr|Mme', 'Mme Unetelle')
['Mme']
>>> re.findall(r'Mr|Mme', 'Mlle Unetelle')
[]
>>> re.findall(r'[abc]def', 'adef bdef cdef')
```

```
['adef', 'bdef', 'cdef']
```

Le regroupement de caractères accepte aussi des caractères d'abréviation, à savoir :

- **-** : définit une plage de valeurs. « **[a-z]** » représente par exemple toutes les lettres de l'alphabet en minuscules.
- **^** : placé en début de jeu, définit la plage inverse. « **[^a-z]** » représente par exemple tous les caractères sauf les lettres de l'alphabet en minuscules.

Les symboles de répétition « **?** », « ***** » et « **+** » sont dits gloutons ou greedy : comme ils répètent autant de fois que possible le symbole précédent, des effets indésirables peuvent survenir.

```
>>> telRegex = re.compile(r'''\d{2}[ ]\(\d\)\d
                [\.]\d{2}[\.]\d{2}
                [\.]\d{2}[\.]\d{2}''', re.VERBOSE)
>>> tel = telRegex.search(
    'Mobile 33 (0)6.85.20.70.68 Tél 33 (0)3.88.27.13.34')
>>> tel.group()
'33 (0)6.85.20.70.68'
>>> telRegex.findall(
    'Mobile 33 (0)6.85.20.70.68 Tél 33 (0)3.88.27.13.34')
['33 (0)6.85.20.70.68', '33 (0)3.88.27.13.34']
```

Dans l'exemple suivant, l'expression régulière tente d'extraire les balises html du texte sans succès : le texte complet est intercepté car il correspond au plus grand texte possible pour le motif. La solution est d'ajouter un symbole « **?** » après le symbole greedy, pour qu'il n'intercepte que le texte minimum.

```
>>> chaine = '<div><span>le titre</span></div>'
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search(chaine)
>>> mo.group()
'<div>'
>>> nongreedyRegex.findall(chaine)
['<div>', '<span>', '</span>', '</div>']
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search(chaine)
>>> mo.group()
'<div><span>le titre</span></div>'
>>> greedyRegex.findall(chaine)
['<div><span>le titre</span></div>']
```

Les symboles de regroupement

Les symboles de regroupement offrent des fonctionnalités qui permettent de combiner plusieurs expressions régulières, au-delà des jeux de caractères « **[ ]** » et de la fonction « **OR** », et d'associer à chaque groupe un identifiant unique. Certaines d'entre elles permettent aussi de paramétrer localement le fonctionnement des expressions.

Symbole	Fonction
(e)	Forme un groupe avec l'expression e. Si les caractères « **(** » ou « **)** » sont utilisés dans e, ils doivent être préfixés de « **** »
(?FLAGS)	Insère directement des flags d'options dans l'expression. S'applique à l'expression complète quel que soit son positionnement.
(?:e)	Similaire à (e) mais le groupe intercepté n'est pas conservé.
(?P<name>e)	Associe l'étiquette name au groupe. Ce groupe peut ensuite être manipulé par ce nom par le biais des API de « **re** », ou même dans la suite de l'expression régulière.
(?#comment)	Insère un commentaire, qui sera ignoré. Le mode « **verbose** » est plus souple pour l'ajout direct de commentaires en fin de ligne.
(?=e)	Similaire à (e) mais le groupe n'est pas consommé.
(?!e)	Le groupe n'est pas consommé et est intercepté uniquement si le pattern (le motif) n'est pas e. (?!e) est le symbole inverse de (?=e)
(?<=e1)e2	Intercepte e2 à condition qu'elle soit préfixée d'e1.
(?<!e1)e2	Intercepte e2 à condition qu'elle ne soit pas préfixée d'e1.
(?(id/name) e1|e2)	Rend l'expression conditionnelle : si le groupe d'identifiant id ou name existe, e1 est utilisée, sinon e2. e2 peut être omise, dans ce cas e1 ne s'applique que si le groupe id ou name existe. Dans l'exemple <123> et 123 sont interceptés mais pas <123.

Voici quelques exemples.

```
>>> re.findall(r'(\(03\))(80)(.*)','(03)80666666')
[('(03)', '80', '666666')]
>>> re.findall(r'(?i)AAZ*', 'aaZzzRr')
['aaZzz']
>>> re.findall(r'(?:\(03\))(?:80)(.*)','(03)80666666')
['666666']
>>> match = re.search(r'(03)(80)(?P<numero>.*)','0380666666')
>>> match.group('numero')
'666666'
>>> re.findall(r'(?# récupération des balises)<.*?>', \
    '<h2><span>hopla</span></h2>')
['<h2>', '<span>', '</span>', '</h2>']
>>> re.findall(r'John(?= Doe)','John Doe')
['John']
>>> re.findall(r'John(?= Doe)','John Minor')
[]
>>> re.findall(r'John(?! Doe)','John Doe')
```

```
[]
>>> re.findall(r'John(?! Doe)','John Minor')
['John']
>>> re.findall(r'(?<=John )Doe','John Doe')
['Doe']
>>> re.findall(r'(?<=John )Doe','John Minor')
[]
>>> re.findall(r'(?<!John )Doe','John Doe')
[]
>>> re.findall(r'(?<!John )Doe','Juliette Doe')
['Doe']
>>> re.match(r'(?P<one><)?(\d+)(?(one)>)', '<123')
>>> match = re.match(r'(?P<one><)?(\d+)(?(one)>)', '123')
>>> match.group()
'123'
>>> match = re.match(r'(?P<one><)?(\d+)(?(one)>)', '<123>')
>>> match.group()
'<123>'
```

Les fonctions et objets de re

Le module « **re** » contient un certain nombre de fonctions qui permettent de manipuler des motifs et les exécuter sur des chaînes :

Fonction	Description
compile(pattern[, flags])	compile le motif pattern et renvoie un objet de type « **SRE_Pattern** ».
escape(string)	ajoute un antislash « **** » devant tous les caractères non alphanumériques contenus dans string. Permet d'utiliser la chaîne dans les expressions régulières.
findall(pattern, string[, flags])	renvoie une liste des éléments interceptés dans la chaîne string par le motif pattern. Lorsque le motif est composé de groupes, chaque élément est un tuple composé de chaque groupe.
finditer(pattern, string[, flags])	équivalente à « **findall** », mais un itérateur sur les éléments est renvoyé. flags est un entier contenant d'éventuels flags, appliqués au motif complet.
match(pattern, string[, flags])	renvoie un objet de type « **MatchObject** » si le début de la chaîne string correspond au motif. flags est un entier contenant d'éventuels flags, appliqués au motif complet.
search(pattern, string[, flags])	équivalente à « **match** » mais recherche le motif dans toute la chaîne.
split(pattern, string[,maxsplit=0])	équivalente au « **split** » de l'objet string. Renvoie une séquence de chaînes délimitées par le motif pattern. Si maxsplit est fourni, limite le nombre d'éléments à maxsplit, le dernier élément regroupant la fin de la chaîne lorsque maxsplit est atteint.
sub(pattern, repl, string[, count])	remplace les occurrences du motif pattern de string par repl. repl peut être une chaîne ou un objet « **callable** » qui reçoit un objet « **MatchObject** » et renvoie une chaîne. Si count est fourni, limite le nombre de remplacements.
subn(pattern, repl, string[, count])	équivalente à « **sub** » mais renvoie un tuple (nouvelle chaîne, nombre de remplacements) au lieu de la chaîne.

```
>>> import re
>>> motif = re.compile('(Mr|Mme|Mlle)\s([A-Za-z]+)\s([A-Za-z]+)')
>>> print(motif.sub(r'Nom: \3, Prénom: \2', 'Mr John Doe'))
Nom: Doe, Prénom: John
>>> print(motif.sub(r'Mon nom est \g<3>, \g<2> \g<3>'\
          ,'Mr Jean Bon'))
Mon nom est Bon, Jean
```

Atelier 4

Exercice n° 1

Écrivez un script qui permette de créer et de relire aisément un fichier texte. Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite il lui proposera le choix, soit d'enregistrer de nouvelles lignes de texte, soit d'afficher le contenu du fichier.

L'utilisateur devra pouvoir entrer ses lignes de texte successives en utilisant simplement la touche « **`Enter`** » pour les séparer les unes des autres. Pour terminer les entrées, il lui suffira d'entrer une ligne vide.

L'affichage du contenu devra montrer les lignes du fichier séparées les unes des autres de la manière la plus naturelle (les codes de fin de ligne ne doivent pas apparaître).

Exercice n° 2

Considérons que vous avez à votre disposition un fichier texte contenant des phrases de différentes longueurs. Écrivez un script qui recherche et affiche la phrase la plus longue.

Écrivez un script qui compare les contenus de deux fichiers et signale la première différence rencontrée.

À partir de deux fichiers préexistants A et B, construisez un fichier C qui contienne alternativement un élément de A, un élément de B, un élément de A... et ainsi de suite jusqu'à atteindre la fin de l'un des deux fichiers originaux. Complétez ensuite C avec les éléments restant sur l'autre.

Écrivez un script qui permette d'encoder un fichier texte dont les lignes contiendront chacune les noms, prénom, adresse, code postal et no de téléphone de différentes personnes (considérez par exemple qu'il s'agit des membres d'un club).

Écrivez un script qui recopie le fichier utilisé dans l'exercice précédent, en y ajoutant la date de naissance et le sexe des personnes (l'ordinateur devra afficher les lignes une par une et demander à l'utilisateur d'entrer pour chacune les données complémentaires).