Chapitre 11

Manipulation de fichiers

Les données que l'on peut souhaiter manipuler (textes, images, ...) sont souvent fournies sous forme de fichiers ; l'objectif de ce chapitre est d'apprendre à effectuer un certain nombre de tâches les concernant, en particulier :

- ouvrir un fichier ;
- lire le contenu d'un fichier ;
- modifier le contenu d'un fichier ;
- fermer un fichier.

Interaction avec le système d'exploitation

Dans le premier chapitre de ce cours, nous avons appris qu'un système d'exploitation gère les fichiers et répertoires sous forme arborescente, et qu'au sein de cette arborescence se trouve le répertoire *courant*, c'est-à-dire celui dans lequel seront *par défaut* sauvegardés et recherchés les fichiers que nous manipulerons. Il se peut que ce répertoire ne corresponde pas à celui dans lequel vous souhaitez travailler ; il est alors nécessaire de préciser à l'interprète PYTHON le chemin de votre répertoire de travail.

Les instructions permettant à l'interprète de dialoguer avec le système d'exploitation ne sont pas directement accessibles ; elles font partie d'un module additionnel, nommé os, qu'il va falloir commencer par importer :

```
>>> import os
```

Obtenir le répertoire courant

La première commande de ce module que nous allons utiliser est la fonction `getcwd()`, qui indique le répertoire courant :

```
>>> os.getcwd()
'/home/bob'
```

Dans l'exemple ci-dessus, le répertoire courant est celui de l'utilisateur Bob [1] (dans cet exemple le chemin est indiqué suivant les conventions UNIX).

Changer le répertoire courant

La fonction `chdir` permet de changer le répertoire courant, en indiquant en argument (sous la forme d'une chaîne de caractères) le nouveau répertoire souhaité. Quel que soit votre système d'exploitation, vous pouvez utiliser les conventions lexicales du monde UNIX, le module os se chargeant automatiquement de la conversion. Si on reprend l'exemple simplifié donné au chapitre 1 de ce cours, on fait du répertoire travail de Bob le répertoire courant en écrivant :

```
>>> os.chdir('/home/bob/travail')
```

Obtenir la liste des fichiers et répertoires

Enfin, la fonction `listdir` permet d'obtenir la liste des fichiers et répertoires contenus dans un répertoire dont le nom a été passé en argument (sous la forme d'une chaîne de caractères). Là encore, quel que soit votre système d'exploitation, vous pouvez utiliser les conventions lexicales du monde UNIX. Poursuivons le même exemple, en visualisant le contenu du répertoire courant :

```
>>> os.listdir('.')
['doc1']
```

Nous avons ici utilisé la description *relative* du répertoire à partir du répertoire courant (représenté, rappelons-le, par un point), mais nous aurions pu évidemment utiliser sa description absolue pour obtenir le même résultat :

1. voir le chapitre 1 de ce cours.

```
>>> os.listdir('/home/bob/travail')
['doc1']
```

Il existe dans le module os bien d'autres fonctions permettant d'interagir avec le système d'exploitation mais dont nous n'aurons pas l'usage par la suite.

1. Lecture et écriture dans un fichier texte

Désormais, nous supposerons que dans le répertoire courant se trouve un fichier nommé `exemple.txt` contenant le texte suivant :

```
Am, stram, gram,
Pic et pic et colégram,
Bour et bour et ratatam,
Am, stram, gram.
```

La première chose à faire est d'ouvrir ce fichier, à l'aide de la commande **open**. Cette fonction prend deux arguments : le premier est une chaîne de caractères décrivant (sous forme relative ou absolue) le chemin menant au fichier à ouvrir, le second indiquant le mode d'ouverture : `'r'` (comme *read*) pour lire le contenu du fichier, `'w'` (comme *write*) pour écrire dans ce fichier, `'a'` (comme *append*) pour ajouter du texte à la suite de ce fichier, etc.
Pour l'instant, nous voulons seulement lire son contenu, donc nous écrivons :

```
>>> comptine = open('exemple.txt', 'r')
```

Nous venons de créer une objet nommé `comptine` faisant référence au fichier `exemple.txt` :

```
>>> comptine
<_io.TextIOWrapper name='exemple.txt' mode='r' encoding='UTF-8'>
```

Nous allons désormais faire agir des *méthodes* sur cet objet, qui se répercuteront sur le fichier lié.

1.1 Lecture complète ou partielle

L'objet que nous venons de créer est ce qu'on appelle un *flux* : les caractères sont lisibles uniquement les uns après les autres, sans possibilité de retour en arrière ni de saut en avant. Ce n'est guère pratique, aussi notre première tâche sera de convertir ce flux en chaîne de caractères.
Le moyen le plus simple est d'utiliser la méthode `read()`, prise sans argument, qui lit le flux dans son entier et le convertit en chaîne de caractères :

```
>>> comptine.read()
'Am, stram, gram,\nPic et pic et colégram,\nBour et bour et ratatam,\nAm,
 stram, gram.'
```

Rappelons que dans une chaîne de caractères, le passage à la ligne est représenté par le caractère spécial `\n`.

Cette méthode est la plus simple, mais n'est évidemment adaptée que si le fichier ne contient pas un texte trop long. Si nécessaire, on peut préciser en argument de la méthode `read` le nombre de caractères à lire. Voici par exemple une façon de procéder pour lire les caractères par groupe de 10 :

```
>>> comptine = open('exemple.tex', 'r')
>>> lst = []
>>> while True:
...     txt = comptine.read(10)
...     if len(txt) == 0:
...         break
...     lst.append(txt)
>>> lst
['Am, stram,', ' gram,\nPic', ' et pic et', ' colégram,', '\nBour et b',
 'our et rat', 'atam,\nAm, ', 'stram, gra', 'm.']
```

• Lecture par lignes

Plus intéressant, la méthode `readline()` permet de lire une ligne de texte (en incluant le caractère de fin de ligne), et surtout la méthode `readlines()`, qui fournit la liste des lignes du texte [2] :

```
>>> comptine = open('exemple.tex', 'r')
>>> comptine.readlines()
['Am, stram, gram,\n', 'Pic et pic et colégram,\n', 'Bour et bour et
 ratatam,\n', 'Am, stram, gram.']
```

Mieux encore, un parcours par énumération est possible, l'énumération du fichier se faisant ligne par ligne :

```
>>> n = 0
>>> for l in comptine:
...     n += 1
...     print('{} :'.format(n), l, end='')
1 : Am, stram, gram
2 : Pic et pic et colégram,
3 : Bour et bour et ratatam,
4 : Am, stram, gram.
```

Fermeture d'un fichier
Enfin, une fois le fichier lu, n'oublions pas de le refermer, afin qu'il soit disponible de nouveau pour tout autre usage : c'est le rôle la méthode `close()` :

```
>>> comptine.close()
```

1.2 Fichiers CSV

De nombreuses données scientifiques se présentent sous forme de tableaux ; une façon simple de les transmettre est de les représenter par un fichier CSV (pour *comma-separated value*) : il s'agit d'un simple fichier texte, chaque ligne de texte correspondant à une ligne du tableau et un caractère spécial (virgule ou point-virgule le plus souvent) aux séparations entre les colonnes. Par exemple, le tableau suivant :

Planète	rayon (en km)	gravité (en m/s^2)	période de révolution (en jours)
Mercure	2439	3,7	88
Vénus	6052	8,9	225
Terre	6378	9,8	365
Mars	3396	3,7	687

sera représenté par le fichier `planetes.csv` contenant le texte suivant [3] :

```
Mercure, 2439, 3.7, 88
Vénus, 6052, 8.9, 225
Terre, 6378, 9.8, 365
Mars, 3396, 3.7, 687
```

Ce format est facile à générer et à intégrer, c'est pourquoi de nombreuses données publiques de l'*Open Data* sont diffusées sous ce format.

Nous allons maintenant nous intéresser à la manière d'intégrer ce fichier de données au sein d'un environnement Python.
On commence par découper le texte en lignes :

```
>>> planetes = open('planetes.csv', 'r')
>>> lignes = planetes.readlines()
>>> planetes.close()
```

et chaque ligne doit ensuite être découpée en colonnes. On utilise pour se faire la méthode `split` qui découpe une chaîne de caractères en une liste de sous-chaînes, le séparateur de ces sous-chaînes étant indiqué en paramètre. Dans le cas de notre fichier CSV cela donne :

2. À condition bien entendu que le texte ne soit pas démesurément grand.
3. Pour simplifier les explications, les en-têtes des colonnes ont été omis.

```
>>> tab = []
>>> for chn in lignes:
...     tab.append(chn.split(','))
```

À cette étape, tab est une liste de listes égale à :

```
[['Mercure', ' 2439', ' 3.7', ' 88\n'], ['Vénus', ' 6052', ' 8.9', ' 225\n'],
 ['Terre', ' 6378', ' 9.8', ' 365\n'], ['Mars', ' 3396', ' 3.7', ' 687\n']]
```

Il reste à convertir le deuxième et le quatrième terme de chacune de ces listes en un entier et le troisième en un flottant :

```
>>> for lst in tab:
...     lst[1] = int(lst[1])
...     lst[2] = float(lst[2])
...     lst[3] = int(lst[3])
```

et la liste tab est maintenant prête à être utilisée :

```
[['Mercure', 2439, 3.7, 88], ['Vénus', 6052, 8.9, 225],
 ['Terre', 6378, 9.8, 365], ['Mars', 3396, 3.7, 687]]
```

1.3 Écrire dans un fichier

Deux modes d'ouverture sont possibles pour écrire dans un fichier : le mode `'w'` (*write* pour écrire) et le mode `'a'` (*append* pour ajouter). Le premier crée un nouveau fichier (s'il existe déjà un fichier du même nom, ce dernier sera effacé) et l'écriture commencera au début du fichier, tandis que le second ajoutera à la suite des données existantes celles que nous allons lui fournir. Dans les deux cas, la méthode `write` permet d'enregistrer les chaînes de caractères passées en argument les unes à la suite des autres.
Par exemple, pour ajouter au fichier `planetes.csv` des données supplémentaires, on procède ainsi :

```
>>> planetes = open('planetes.csv', 'a')
>>> planetes.write('Jupiter, 71492, 24.8, 4335\n')
>>> planetes.write('Saturne, 60268, 10.4, 10757\n')
>>> planetes.close()
```

(Ne pas oublier de fermer le fichier pour enregistrer les modifications.)

2. Encodage d'un fichier texte

2.1 Le jeu de caractères ASCII

À l'origine du développement de l'informatique, il a été décidé de coder un caractère sur un octet, et plus précisément sur 7 bits, le premier bit étant fixé à 0. On disposait ainsi de $2^7 = 128$ caractères différents, qui constituent le jeu de caractères ASCII (*American Standard Code for Information Interchange*). Tout nombre compris entre 0 et 127 s'écrivant en base 16 sous la forme $(xy)_{16}$ avec $x \in \{0,1,\ldots,6,7\}$ et $y \in \{0,1,\ldots,e,f\}$, on représente souvent la correspondance entre code ASCII et caractères par le tableau de la figure 1.
Les cases grisées correspondent à des caractères spéciaux (tabulation, passage à la ligne, etc.) qui peuvent dépendre du système ; le caractère correspondant au code ASCII $(20)_{16} = 32$ est l'espace inter-mot.

En PYTHON, on obtient le code ASCII d'un caractère à l'aide de la fonction `ord` ; à l'inverse, la fonction `chr` retourne le caractère dont le code ASCII est donné en paramètre :

```
>>> ord('A')
65
>>> chr(97)
'a'
```

En effet, $65 = (41)_{16}$ et $97 = (61)_{16}$.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0																
1																
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

FIGURE 1 – Table des caractères ASCII.

Malheureusement, ce codage simple est insuffisant pour pouvoir représenter la diversité des caractères des langues autres que l'anglais, aussi le huitième bit a été utilisé pour ajouter au jeu de caractères ASCII standard 128 autres caractères codés entre $128 = (80)_{16}$ et $255 = (ff)_{16}$. Cependant, chaque langue ayant des besoins spécifiques, ces extensions sont nombreuses et non compatibles entre elles : la norme LATIN-1 permet par exemple d'encoder les langues d'Europe occidentale (en partie tout du moins, puisque le caractère *œ* manque pour le français), la norme LATIN-2 pour les langues d'Europe centrale, etc. Pas moins de 16 variantes existent pour le seul standard ISO 8859. Et encore ne s'agit-il ici que de représenter les caractères des langues alphabétiques, car les écritures idéographiques comme le chinois nécessitent plusieurs milliers de caractères et ne peuvent donc être codées sur un seul octet.

Bref, le seul intérêt résiduel de ces anciennes normes réside dans leur simplicité : chaque caractère typographique est représenté par un seul octet, et une chaîne de caractères n'est rien d'autre qu'une séquence d'octets.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
8																
9																
a		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
b	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
c	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
d	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
e	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
f	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIGURE 2 – Extention LATIN-1 de la norme ASCII.

2.2 La norme Unicode

Ces différentes contraintes ont poussé à l'adoption d'une norme universelle, la norme Unicode, qui attribue un identifiant numérique différent à chacun des milliers de caractères nécessaires à la transcription des différentes langues mondiales. En revanche, il importe de comprendre que cette norme ne précise pas sous quelle forme cet identifiant doit être encodé par le système informatique. Il existe donc plusieurs normes d'encodages différentes en fonction des besoins, mais chacune a en commun d'associer à chaque caractère le même identifiant numérique.

L'encodage le plus fréquent est l'UTF-8 ; c'est la norme utilisée par défaut par PYTHON. Elle utilise entre 1 et 4 octets pour encoder un caractère et présente l'avantage d'assurer une parfaite compatibilité avec les textes encodés en ASCII.

Revenons un instant sur le fichier `exemple.txt` qui contient la comptine utilisée au début de ce chapitre ; il s'agit d'un fichier encodé au format UTF-8, encodage qui a été choisi par défaut par PYTHON lors de l'ouverture :

```
>>> comptine = open('exemple.txt', 'r')
>>> print(comptine.read())
Am, stram, gram
Pic et pic et colégram,
Bour et bour et ratatam,
Am, stram, gram.
```

Essayons maintenant de l'ouvrir avec un mauvais encodage :

```
>>> comptine = open('exemple.txt', 'r', encoding='latin1')
>>> print(comptine.read())
Am, stram, gram
Pic et pic et colÃ©gram,
Bour et bour et ratatam,
Am, stram, gram.
```

On peut observer que le caractère non ASCII `"é"` a été mal interprété à cause du mauvais encodage.

Caractères non occidentaux

À titre de curiosité, notons pour finir qu'il est possible d'accéder aux caractères non latins à l'aide de la fonction `chr` dont nous avons déjà parlé, qui non seulement retourne le caractère ASCII dont l'identifiant a été passé en paramètre mais plus généralement le caractère unicode associé à son identifiant, pour peu que le système soit capable de l'afficher.
Par exemple, les caractères minuscules de l'alphabet grec ont un identifiant compris entre 945 et 969 ; le script qui suit permet de les afficher :

```
>>> for i in range(945, 970):
...     print(chr(i), end=' ')
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ υ φ χ ψ ω
```

3. Lecture et écriture dans un fichier image

Les images publiées sur un site internet, les photographies prises avec un téléphone portable, sont des exemples d'images numériques. Il est possible de représenter ce type d'image par une matrice. Par exemple, L'image ci-dessous peut être représentée par une matrice 35×35 dont les éléments, des 0 ou des 1, indiquent la couleur du pixel : 0 pour le noir et 1 pour le blanc.

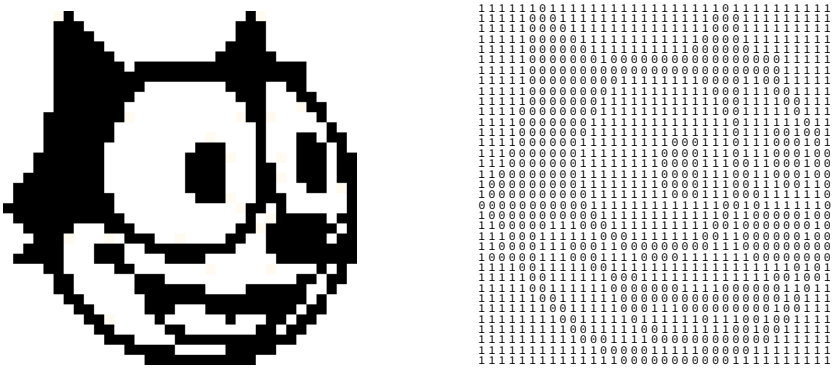

FIGURE 3 – Une image binaire et la matrice qui lui est associée.

Les images en niveau de gris peuvent aussi être représentées par des matrices, mais cette fois chaque élément détermine la luminance du pixel correspondant. Pour des raisons pratiques, la majorité des images en gris utilisent un entier compris entre 0 (pour le noir) et 255 (pour le blanc), autrement dit par un entier non signé codé sur 8 bits (un octet).

Enfin, les images en couleur peuvent être représentées par trois matrices, chacune déterminant la quantité respective de rouge, de vert et de bleu qui constitue l'image. Ce modèle de couleur est appelé RGB (*red-green-blue*) mais d'autres modèles existent, par exemple la quadrichromie (CMJN) en imprimerie.

Figure 4 – Huit niveaux de gris différents.

Les éléments de ces matrices sont des nombres entiers compris entre 0 et 255 (des entiers non signés sur 8 bits) qui déterminent l'intensité de la couleur de la matrice pour le pixel correspondant. Le modèle RGB permet donc de représenter $256^3 = 2^{24} = 16\,777\,216$ couleurs différentes en théorie.

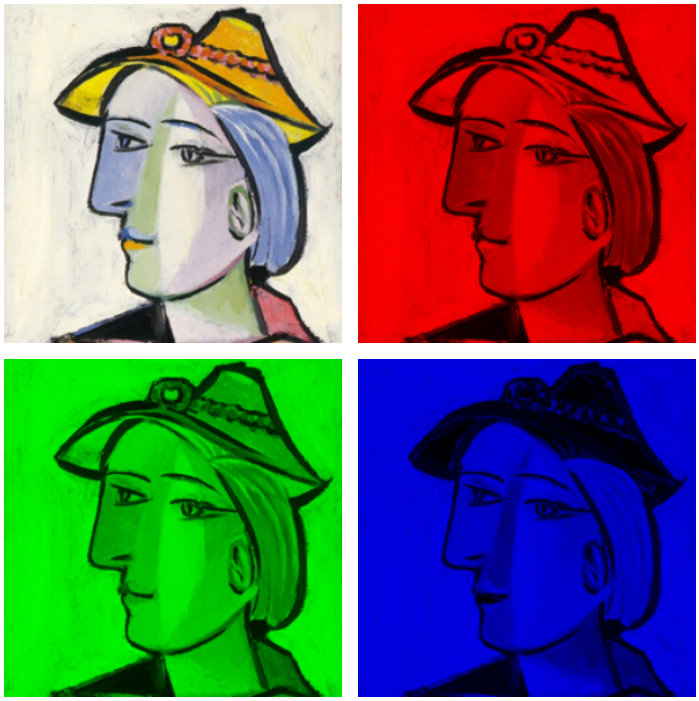

Figure 5 – Une image en couleur. Les trois autres images ont été obtenues en ne conservant qu'une des trois composantes RGB qui la composent.

Traitement d'images en Python

À moins d'installer sur votre système la bibliothèque Pillow spécialisée dans le traitement d'image sous Python il faudra se contenter du module `matplotlib.image` qui propose quelques fonctions basiques pour importer une image, essentiellement :

- `imread('fichier.png')` qui prend en argument le nom d'un fichier image au format png et retourne un tableau numpy ;
- `imsave('fichier.png', tableau)` qui sauvegarde un tableau numpy représentant une image sous la forme d'un fichier png.

À ces deux fonctions il peut être intéressant d'associer la fonction `imshow` du module `matplotlib.pyplot` qui affiche à l'écran une représentation imagée d'un tableau numpy.

Attention, la fonction `imrerad` convertit une image $m \times n$ en niveau de gris en un tableau $m \times n$ dont les valeurs sont non pas au type *uint8* comme on pourrait s'y attendre (entiers non signés sur 8 bits) mais au type *float32* (flottants codés sur 32 bits) et s'échelonnent entre `0.` (le noir) et `1.` (le blanc).
De même, une image en couleur de taille $m \times n$ sera transformée en un tableau tri-dimensionnel $m \times n \times 3$, chaque pixel étant associé à un triplet RGB au type *float32*.
À titre d'exemple, l'image précédente a été produite à l'aide du script :

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

img = mpimg.imread('picasso.png')

rouge = np.zeros_like(img)
vert = np.zeros_like(img)
bleu = np.zeros_like(img)

for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        rouge[i, j, 0] = img[i, j, 0]
        vert[i, j, 1] = img[i, j, 1]
        bleu[i, j, 2] = img[i, j, 2]

fig = plt.figure(figsize=(6, 6), frameon=False)
plt.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=.05, wspace=.05)

plt.subplot(2, 2, 1)
plt.axis('off')
plt.imshow(img)

plt.subplot(2, 2, 2)
plt.axis('off')
plt.imshow(rouge)

plt.subplot(2, 2, 3)
plt.axis('off')
plt.imshow(vert)

plt.subplot(2, 2, 4)
plt.axis('off')
plt.imshow(bleu)

plt.show()
```