3

La gestion des fichiers

Lorsqu'un programme est chargé en mémoire pour être exécuté, son environnement est volatile. Toutes les variables sont créées par le programme et sont détruites une fois l'exécution terminée. Les interactions avec le monde extérieur se font par un mécanisme d'entrées et de sorties : un programme peut lire et écrire des informations au moyen d'interfaces. On compte parmi ces interfaces :

- la console (entrée standard, sortie standard et sortie d'erreur),
- les tubes de communication (en anglais *pipe*),
- les *sockets* de communication réseau,
- et surtout les fichiers, supports de stockage textuels ou binaires.

Tous ces mécanismes d'interaction ont une structure similaire basée sur l'itération : le chargement de l'intégralité d'un fichier en mémoire pour le lire ou l'écrire étant souvent superflu, il est courant de lire et écrire celui-ci dans ces interfaces de manière séquentielle.

3.1. Le module `pathlib`

Python est un environnement multi-plateforme : le même programme peut s'exécuter quels que soient l'architecture de la machine, le modèle de processeur et le système d'exploitation, à condition qu'un interpréteur Python spécialement préparé (compilé) pour cette architecture y soit disponible.

Le formalisme de nommage des fichiers, différent entre les systèmes d'exploitation, peut être un frein à cette pratique. Le chemin d'accès est constitué d'une série de noms de répertoires à traverser pour accéder au fichier. Un séparateur (`/` sous Linux ou MacOS et `\` sous Windows) permet de séparer les différents noms. L'origine de ce chemin peut être absolue, exprimée par rapport à la racine de l'arborescence de fichiers, ou relative, par rapport au répertoire courant. Les répertoires `.` et `..` font respectivement référence au répertoire courant et au répertoire parent.

Le module `pathlib` permet de pallier ces différences et les écueils liés à l'échappement du caractère `\` en offrant une interface compatible entre les plateformes pour explorer l'arborescence de fichiers du système.

```
>>> from pathlib import Path
>>> current = Path(".")
>>> current
PosixPath('.')
>>> current.absolute()
PosixPath('/home/xo')
```

```
>>> docs = current / "Documents"
>>> docs
PosixPath('./Documents')
>>> (current / "Documents").absolute()
PosixPath('/home/xo/Documents')
```

Les méthodes de concaténation qui permettent de construire un chemin vers un fichier ne vérifient pas l'existence du fichier ni la cohérence du chemin. Des méthodes spécifiques permettent de vérifier l'existence d'un fichier, sa nature (fichier, répertoire, lien symbolique, etc.) et de créer ces chemins si nécessaire.

```
>>> livre = docs / "Livre Python"
>>> livre.exists()
False
```

```
>>> livre.mkdir()
>>> livre.exists()
True
>>> livre.is_dir()
True
```

Les attributs suivants permettent de manipuler différents attributs d'un chemin : le nom du fichier (`name`), sans son extension (`stem`), ou uniquement l'extension (`suffix`).

```
>>> todo = (livre / "todo.txt")
>>> todo.parent
PosixPath('Documents/Livre Python')
>>> todo.name
'todo.txt'
>>> todo.stem
'todo'
>>> todo.suffix
'.txt'
>>> todo.with_suffix(".docx")
PosixPath('Documents/Livre Python/todo.docx')
```

Quand les fichiers sont suffisamment petits pour être entièrement chargés sans saturer la mémoire de l'ordinateur, les méthodes `.read_text()` et `.write_text()` sont adaptées pour lire ou écrire l'intégralité du contenu textuel d'un fichier.

```
>>> contenu = "Liste des chapitres à écrire"
>>> todo.write_text(contenu)  # Écriture rapide dans un fichier
28
>>> todo.read_text()  # Lecture rapide dans un fichier
'Liste des chapitres à écrire'
>>> todo.is_file()
True
```

Ces méthodes qui manipulent des chaînes de caractères `str` (☞ p. 8, § **1.3**) sont réservées aux fichiers textuels. Dans le cas général (fichier textuel ou binaire) qui fait appel au type `bytes` (☞ p. 10, § **1.3**), les méthodes correspondantes sont `.read_bytes()` et `.write_bytes()`. Dans l'exemple suivant, on peut lire le contenu d'un fichier image PNG, dont la signature (les 8 premiers octets) est caractéristique.

```
>>> logo = p / "logo-python.png"
>>> content = logo.read_bytes()
>>> content[:8]
b'\x89PNG\r\n\x1a\n'
```

Nota bene Le standard définit cette même signature de 8 octets pour tous les fichiers PNG : le premier caractère notamment est situé en dehors de l'intervalle ASCII pour s'assurer qu'aucun fichier texte ne puisse être mépris pour un fichier PNG. Ainsi avec Python, l'ouverture d'un fichier PNG avec la méthode `.read_text()` lève une exception.

```
>>> content = logo.read_text()
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x89 [...]
```

Il est également possible de lister tous les fichiers d'une arborescence qui respectent un motif à l'aide de la méthode `.glob()` : le motif est défini à l'aide d'une expression régulière (☞ p. 30, § 2.4). On peut par exemple :

① lister dans le répertoire courant tous les fichiers de configuration (à l'extension `rc`) ;
② compter le nombre de fichiers dans toute l'arborescence du dossier temporaire `/tmp` : le motif `**/` parcourt tous les sous-dossiers du répertoire courant ;
③ accéder à des informations sur le fichier : permissions, utilisateurs, groupe, dates de création, de modification, etc.

```
>>> list(current.glob(".*rc"))  # ①
[PosixPath('.npmrc'), PosixPath('.wgetrc'), PosixPath('.zshrc'),
 PosixPath('.condarc'), PosixPath('.vimrc')]
>>> sum(1 for f in Path("/tmp").glob("**/*") if f.is_file())  # ②
86
>>> todo.stat()  # ③
os.stat_result(
    st_mode=33204, st_ino=14558483, st_dev=2050, st_nlink=1,
    st_uid=1001, st_gid=1001, st_size=30,
    st_atime=1586815012, st_mtime=1586815012, st_ctime=1586815012
)
```

Ce dernier exemple adapte la ligne ② pour utiliser la dernière date de modification du fichier : l'objectif ici est de compter le nombre de fichiers qui ont été modifiés il y a moins de 86400 secondes (24 heures).

```
>>> now = datetime.now()
>>> sum(
...     1 for f in Path("/tmp").glob("**/*")
...     if f.is_file() and now.timestamp() - f.stat().st_mtime < 86400
... )
5
```

3.2. Lecture et écriture séquentielles

Si la plupart des fichiers de petite taille peuvent être lus ou écrits avec les fonctions précédentes, ce mode de fonctionnement peut être surdimensionné dans certains cas :

— si l'information recherchée ne nécessite pas de stocker en mémoire tout le contenu du fichier, par exemple pour trouver la première ligne qui contient le caractère `#` ou pour compter le nombre de lignes d'un fichier textuel ;
— si le fichier est trop gros pour tenir dans la mémoire vive de l'ordinateur.

Dans ce cas, il convient de décomposer la manipulation :

— l'ouverture du fichier, à l'aide de la fonction `open()`, en précisant le mode d'ouverture (`'r'` pour la lecture seule, ☞ *read*, `'w'` pour l'écriture ☞ *write*, `'a'` pour l'écriture à la fin du fichier sans écraser le contenu existant ☞ *append*) et la nature du fichier à manipuler (par défaut textuel, sinon binaire avec l'option `'b'`) ;
— la lecture ou l'écriture d'une partie du fichier, à l'aide des fonctions `read()` et `write()` dans le cas général ;
— la fermeture du fichier.

En pratique, on utilise le schéma suivant :

④ le gestionnaire de contextes (avec le mot-clé `with` ☞ p. 236, § **16.3**) se charge de fermer correctement le fichier à la sortie du bloc, même si une exception est levée pendant l'exécution du bloc ;
⑤ pour les fichiers textuels, soit on utilise la méthode `readlines()` qui lit le fichier dans son intégralité pour le découper ligne par ligne ;
⑥ soit on utilise une simple itération qui charge les lignes en mémoire une par une ;
⑦ pour écrire dans un fichier, il faut ajouter manuellement les sauts de ligne.

```
p = Path("lorem_ipsum.txt")

# Le fichier est lu en entier puis découpé en une liste de chaînes
# de caractères en se basant sur le saut de ligne \n
with p.open('r') as fh:  # ④
    lines: list = fh.readlines()  # ⑤
    nb_lines = len(lines)

# On lit les lignes une par une pour compter le nombre de mots
num_words = 0
with p.open('r') as fh:  # ④
    for line in fh:  # ⑥
        num_words += len(line.split())

# On ouvre le fichier pour ajouter une ligne à la fin
with p.open('a') as fh:
    fh.write("# fin du fichier\n")  # ⑦
```

Pour les fichiers binaires, on utilise plutôt la méthode `.read()`, qui prend en argument un nombre d'octets à lire. Dans l'exemple ci-dessous, des fichiers images PNG ont malencontreusement été renommés avec l'extension `.jpg` : le programme fait la manipulation inverse. Ici aucun fichier n'est chargé en mémoire au-delà des 8 premiers octets, qui nous intéressent pour déterminer le type du fichier.

```
for fichier in Path(".").glob("*.jpg"):
    with fichier.open("rb") as fh:
        header = fh.read(8)

    if header == b'\x89PNG\r\n\x1a\n':
        fichier.rename(fichier.with_suffix(".png"))
```

3.3. Vérification de l'intégrité des fichiers

Lors du transfert d'un fichier produit par un tiers, une bonne pratique consiste à fournir en même temps que le fichier une chaîne de caractères hexadécimale, générée à l'aide d'une *fonction de hash*. C'est une mesure de protection simple contre la corruption de fichier, malfaisante ou non. Elle réduit le contenu d'un fichier à une chaîne de caractères, appelée *empreinte* (*hash* en anglais). La vérification consiste, une fois le fichier récupéré, à recalculer la fonction de hash et à comparer le résultat avec l'empreinte fournie. Si les deux valeurs correspondent, on considère que le fichier n'est pas corrompu.

Les *fonctions de hash* les plus communément utilisées sont MD5 (pour *Message Digest 5*) et les différentes versions de SHA (pour *Secure Hash Algorithm*). Ces *fonctions de hash* sont disponibles directement en Python, dans le module `hashlib`. Elles s'appliquent directement sur un objet de type `bytes`. La principale propriété voulue pour ces fonctions est de renvoyer des empreintes très différentes pour deux séquences `bytes` très proches.

```
>>> import hashlib
>>> hashlib.md5(b"Python!").hexdigest()
'b4fb1ac018715d026bcf69071f8919af'
>>> hashlib.md5(b"Python?").hexdigest()
'88eb397bcdc48f676d7008f765e5da1f'
```

Pour recalculer l'empreinte d'un fichier, on peut charger sa représentation binaire, et appliquer la même fonction.

```
>>> import sys
>>> from pathlib import Path
>>> sys.executable  # l'exécutable Python
'/usr/local/bin/python3.8'
>>> bytes_content = Path(sys.executable).read_bytes()
>>> hashlib.md5(bytes_content).hexdigest()
'a20563dd6d6256d1a285150b7309989c'
```

Pour de gros fichiers que l'on peut parcourir par morceaux, il est possible de construire l'empreinte de manière itérative à l'aide de la méthode `.update()`

```
p = Path("<gros fichier>")
h = hashlib.md5()

with p.open('rb') as fh:
    while True:
        data = fh.read(1024)  # lire par paquet de 1024 octets
        if data == b"":  # il n'y a plus rien à lire dans le fichier
            break
        # mise à jour du hash avec la nouvelle séquence d'octets
        h.update(data)

md5sum = h.hexdigest()
```

3.4. Sérialisation

La sérialisation est une opération qui permet de représenter un objet Python sous une forme qui puisse être enregistrée dans un fichier ou partagée avec d'autres processus. La difficulté de la sérialisation vient d'un compromis entre la compatibilité et la performance :

- une représentation binaire brute de l'objet permet d'écrire et de reconstruire un objet rapidement. En revanche, il n'est pas possible de contrôler le contenu de la représentation, ni de partager cet objet avec d'autres langages de programmation ;
- une représentation textuelle qui suit un formalisme de modélisation permet de fournir toutes les informations pour reconstruire l'objet en question dans n'importe quel langage de programmation. En revanche, la lecture et l'interprétation d'une telle représentation est plus coûteuse.

Le module `pickle` est spécifique au langage Python. Il est utilisé par le langage pour échanger des objets entre processus. La représentation binaire (le type `bytes`) d'un objet Python est alors écrite dans un fichier. Il est possible de partager ces fichiers entre ordinateurs mais la version de Python doit être la même : un fichier `pickle` écrit avec Python 3.8 ne pourra pas être ouvert avec la version 3.7 par exemple.

```
# écriture dans un fichier pickle
with Path("f1.pkl").open('wb') as fh:
    pickle.dump(elt1, fh)
    pickle.dump(elt2, fh)
```

```
# lecture des objets
with Path("f1.pkl").open('rb') as fh:
    elt1 = pickle.load(fh)
    elt2 = pickle.load(fh)
```

Le module `json` permet de lire et écrire des fichiers au format JSON (JavaScript Object Notation), un format de données textuel qui permet de représenter de l'information structurée. Des bibliothèques pour le format JSON existent dans la plupart des langages de programmation. Le format JSON se représente naturellement en Python à l'aide de dictionnaires, de listes et de valeurs génériques : chaîne de caractères, nombres (entiers, flottants) et booléens (en minuscule en JSON), et la valeur vide `None` (`null` en JSON). En revanche, le format n'accepte pas les commentaires.

```
pays = {
  'pays': [
    {'n': 'France', 'c': 'Paris'},
    {'n': 'Espagne', 'c': 'Madrid'},
    {'n': 'Italie', 'c': 'Rome'},
  ],
  'properties': {
    'n': 'nom', 'c': 'capitale'
  }
}
```

```
import json

with Path("pays.json").open('w') as fh:
    json.dump(pays, fh, indent=2)

with Path("pays.json").open() as fh:
    pays = json.load(fh)
```

✧ Bonnes pratiques

Une exception de type `TypeError` est levée si on souhaite sérialiser un objet Python classique. Une pratique courante consiste alors à sérialiser manuellement l'objet dans un dictionnaire avec tous les arguments qui permettent de le reconstruire.

```
exemple = {
    ...,
    # 'modification': datetime(2020, 1, 1, tz=timedelta(hours=1)),
    'modification': {'timestamp': 1577833200, 'timezone': '+01:00'}
}
```

Le module `base64` permet la conversion de données binaires (le type `bytes`) en chaîne de caractères qui utilise 64 caractères différents. L'intérêt de ce système est surtout de permettre de transmettre des données binaires courantes au sein d'un fichier textuel, au format JSON par exemple.

Dans l'exemple ci-dessous, l'utilisation du chemin vers un fichier PNG implique de partager les fichiers images en même temps que le fichier JSON, sans contrôle sur le contenu des fichiers images. Dans certains cas, il peut être préférable de partager un simple fichier JSON qui intègre les images en utilisant une représentation textuelle. Cette technique est couramment utilisée pour partager du contenu (image, son, police de caractères) sur des pages web.

```
import base64

with Path("drapeaux/fr.png").open('b') as fh:
    x = base64.encodebytes(fh.read())  # b'iVBORw0KGgoAAAANS [...]

# avec un chemin vers un fichier
france = {'n': 'France', 'c': 'Paris', 'd': 'drapeaux/fr.png'}
# avec la représentation base64: .decode() transforme en type str
france = {'n': 'France', 'c': 'Paris', 'd': x.decode() }
```

3.5. Flux de données

Les *sockets* sont des canaux de communication entre processus (programmes). Ces canaux peuvent être ouverts pour communiquer entre processus au sein d'une même machine ou *via* l'interface réseau, *sur Internet* par exemple. Différents protocoles de communication (TCP, UDP) existent : une explication détaillée déborde du cadre de cet ouvrage. Les applications que nous utilisons quotidiennement pour accéder à Internet (mail, navigation web HTTP, etc.) sont basées sur ces protocoles et fonctionnent à l'aide de *sockets*.

Le module `socket` permet de manipuler ces outils en Python. Prenons un cas d'utilisation très simple avec un programme Python qui renvoie l'heure quand on l'interroge.

```
from datetime import datetime, timezone
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:  # ①
    s.bind(("127.0.0.1", 12345))  # ②
    s.listen()  # ③
    conn, addr = s.accept()  # ④
    with conn:
        now: datetime = datetime.now(tz=timezone.utc)
        conn.sendall(str(now).encode())  # ⑤
```

Une *socket* est créée au sein d'un gestionnaire de contexte ①, elle gère automatiquement sa fermeture. Les paramètres (une explication complète dépasserait le cadre de cet ouvrage) correspondent à l'utilisation du protocole TCP/IP. La *socket* est alors rattachée à une adresse et un port ②. `127.0.0.1` correspond à l'adresse locale de chaque machine : le service proposé n'est alors accessible que depuis le même ordinateur. La *socket* est alors mise en attente de connexion ③. Une fois qu'une connexion est initiée et acceptée ④, on peut écrire n'importe quelle séquence de `bytes` dans la `socket` : le contenu sera reçu par le client.

Une fois le programme précédent lancé, il est possible de s'y connecter depuis un autre terminal à partir d'outils standards comme `netcat` [1] :

```
$ nc localhost 12345
2020-12-31 23:59:59.997960+00:00
```

Il est également possible d'utiliser une `socket` en mode client, comme le font `netcat` ou `telnet`. Pour cela, il suffit de lire de manière séquentielle le contenu de la `socket` ⑦. Le garde ⑧ permet de quitter le programme quand la réception est terminée. L'auteur recommande au lecteur de consulter la page https://www.telnet.org/htm/places.htm pour reproduire l'exemple ci-dessous avec des services similaires.

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:  # ⑥
    s.connect(("towel.blinkenlights.nl", 666))
    while True:
        data: bytes = s.recv(256)  # ⑦
        if len(data) == 0:  # ⑧
            break
        print(data.decode())  # ⑨
```

L'exemple précédent ne fait qu'afficher au fur et à mesure ce qui est reçu sur la `socket`. Le contenu binaire reçu comprend des caractères de contrôle pour effacer l'écran `b"\x1b[J"` ou pour placer le curseur en haut à gauche du terminal `b"\x1b[H"`.

Dans l'exemple suivant, nous n'allons afficher que ce qui est reçu après le dernier caractère de contrôle afin de laisser intacte l'apparence de notre terminal. Il pourrait y avoir plusieurs manières de procéder : stocker chaque `data` reçu dans une liste, puis concaténer tous les éléments reçus dans une nouvelle structure `bytes` ; ou alors écrire tous les `data` reçus dans un fichier binaire puis relire le contenu du fichier.

Il est possible de concaténer de manière efficace et séquentielle du contenu binaire à l'aide de flux de données dans le module `io`. Les structures les plus communes sont `io.BytesIO` pour le contenu binaire et `io.StringIO` pour le contenu textuel.

```
from io import BytesIO

content = BytesIO()  # création du flux de données binaires

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect(("towel.blinkenlights.nl", 666))
    while True:
        data: bytes = s.recv(1024)
        if len(data) == 0:
            break
        content.write(data)  # écriture séquentielle

content.seek(0)  # On se place au début du flux
data = content.read()  # puis on lit l'intégralité du flux

clear_idx = data.find(b"\x1b[")
```

1. Si la commande `nc` n'est pas accessible, il est possible de la remplacer par la commande `telnet` qu'il faut avoir préalablement activée sous Windows 10 à l'aide de l'instruction suivante à taper dans l'invite de commande : `pkgmgr/iu:"TelnetClient"`

```
while clear_idx != -1:
    # Effacement du contenu jusqu'au dernier caractère de contrôle
    data = data[clear_idx + 3 :]
    clear_idx = data.find(b"\x1b[")

print(data.decode())  # passage en chaîne de caractères
```

3.6. Compression et archivage

La compression et l'archivage sont des techniques efficaces couramment utilisées pour organiser, stocker ou partager de gros volumes de données. L'*archivage* est une opération qui permet de réunir un ou plusieurs fichiers, organisés en arborescence au sein d'un seul fichier ; la *compression* réduit le volume des archives produites.

Il est bien entendu toujours possible de compresser ou de décompresser les archives à l'aide d'outils tiers avant de manipuler les fichiers en Python. Lorsque cette option est fastidieuse, on peut faire appel à des bibliothèques Python qui permettent de lire et d'écrire directement des archives des formats les plus courants[2] : `zip`, `tar`, `gzip`, `bzip2`, ou `lzma`. Des bibliothèques externes sont également disponibles en support d'autres formats. Pour illustrer ce chapitre, nous nous contenterons de la bibliothèque `zipfile` mais la logique d'utilisation est la même quel que soit le format choisi.

Dans l'exemple suivant, nous utilisons une archive qui contient des images PNG des drapeaux des pays du monde. Après avoir téléchargé l'archive, il est possible d'explorer le contenu de l'archive avec le même motif de programmation que pour la lecture d'un fichier. La méthode `.infolist()` renvoi une structure semblable à un dictionnaire avec des informations sur les fichiers contenus dans l'archive : nom du fichier `filename`, taille du fichier une fois compressé `compress_size` et d'autres informations. On peut alors ouvrir les fichiers contenus dans l'archive à l'aide de la méthode `.open()`.

```
import json

from pathlib import Path
from zipfile import ZipFile

# Les fichiers sont téléchargeables sur flagpedia.net
# https://flagcdn.com/w2560.zip
# https://flagcdn.com/fr/codes.json

f_countries = Path("codes.json")
countries = json.loads(f_countries.read_text())
with ZipFile("w2560.zip", "r") as zf:
    all_files = []

    # On ouvre chaque fichier de l'archive
    for file_info in zf.infolist():
        with zf.open(file_info.filename, "r") as fh:
            # On récupère le nom du fichier, le nom du pays,
            # et la taille du PNG dans l'archive
```

2. https://docs.python.org/fr/3/library/archiving.html

```
            drapeau = {
                "fichier": file_info.filename,
                "taille_zip": file_info.compress_size,
                "pays": countries[file_info.filename[:-4]],
            }
            lire_png(fh, drapeau)  # définie plus loin
            all_files.append(drapeau)
```

On crée ici une liste avec un dictionnaire par fichier PNG présent dans l'archive. Pour finir d'illustrer ce chapitre, nous complétons le dictionnaire de métadonnées (nom du fichier, taille du drapeau dans l'archive, nom du pays) avec d'autres informations présentes dans le fichier.

Le format PNG est un format binaire de représentation des images : nous avons parlé précédemment de sa signature unique. La structure d'un fichier PNG est très formalisée : on y trouve des parties (appelés *chunks*), qui se décomposent toutes de la même manière : une taille (4 octets), un type (4 octets), des données (d'une longueur définie dans le champ de taille) et un code correcteur (4 octets). Il existe différents types de *chunks*, mais tous les fichiers contiennent *a minima*, un en-tête (type `IHDR`), des données compressées (type `IDAT`) et une marque de fin du fichier (type `IEND`).

Pour l'archive qui contient les drapeaux du monde, nous allons lire dans le fichier binaire :
— la taille de l'image (hauteur × largeur) ;
— la taille des données compressées de l'image [3].

```
def lire_entier(x: bytes) -> int:
    """Convertit une séquence bytes en entier.

    >>> lire_entier(b"\x01\x00")
    256
    """
    return int.from_bytes(x, byteorder="big")


def lire_png(fh, drapeau: dict) -> None:
    # Les 8 premiers bits sont la signature b"\x89PNG\r\n\x1a\n"
    signature = fh.read(8)

    # Le fichier est ensuite découpé en "chunks"
    chunk_type = b""

    while chunk_type != b"IEND":

        # Un chunk est constitué de 4 bits de taille, 4 bits de type,
        # puis des données, et enfin 4 bits d'un code correcteur d'erreur
        length = lire_entier(fh.read(4))
        chunk_type = fh.read(4)
        chunk_data = fh.read(length)
        crc = fh.read(4)
```

3. Le format PNG compresse la représentation d'une image au format *deflate*.

FIGURE 3.1 – Drapeaux les moins (première ligne) et les mieux compressés (deuxième ligne) par le format PNG

```
        # On récupère la taille de l'image dans le header (chunk IHDR)
        if chunk_type == b"IHDR":
            drapeau["largeur"] = lire_entier(chunk_data[:4])
            drapeau["hauteur"] = lire_entier(chunk_data[4:8])
            drapeau["L×h"] = drapeau["largeur"] * drapeau["hauteur"]

        # On récupère la taille de la partie compressée de l'image (chunk IDAT)
        if chunk_type == b"IDAT":
            drapeau["taille_png"] = length

    # Enfin, on calcule quelques ratios de compression
    drapeau["png_ratio"] = drapeau["L×h"] / drapeau["taille_png"]
    drapeau["zip_ratio"] = drapeau["taille_png"] / drapeau["taille_zip"]
```

On récupère enfin une liste qui contient un dictionnaire par pays. On peut alors trier cette liste par performance de compression des fichiers de drapeaux : nous pouvons sélectionner les drapeaux les mieux compressés par le format PNG d'une part, et les fichiers PNG qui sont les mieux compressés dans l'archive d'autre part.

```
sorted(all_files, key=itemgetter("png_ratio"))
```

	pays	png_ratio	zip_ratio
bl.png	Saint-Barthélemy	10.14	1.04
io.png	Territoire britannique de l'océan Indien	13.00	1.02
vi.png	Îles Vierges des États-Unis	13.32	1.07
pm.png	Saint-Pierre-et-Miquelon	13.80	1.04
mp.png	Îles Mariannes du Nord	14.42	1.04
se.png	Suède	2095.14	8.69
ch.png	Suisse	2848.15	7.64
pl.png	Pologne	3346.41	8.00
mc.png	Monaco	3362.98	9.93
lv.png	Lettonie	5041.23	3.96

Sans surprise, les drapeaux les mieux compressés par le format PNG sont alors des drapeaux bicolores très simples (Lettonie, Monaco, Pologne) alors que les moins compressés sont beaucoup plus stylisés, avec des armoiries complexes (Figure 3.1).

FIGURE 3.2 – Drapeaux les mieux compressés par le format ZIP

En revanche, les drapeaux les mieux compressés dans l'archive zip sont ceux qui suivent le motif tricolore le plus courant (Figure 3.2). Ces fichiers PNG ont beau avoir une représentation compressée cinq fois moins performantes que la Lettonie, ils sont très bien compressés dans l'archive ZIP parce que leurs structures sont semblables : ils ne diffèrent les uns des autres que par la couleur.

```
sorted(all_files, key=itemgetter("zip_ratio"))
```

	pays	png_ratio	zip_ratio
io.png	Territoire britannique de l'océan Indien	13.00	1.02
gb-wls.png	Pays de Galles	23.20	1.03
bl.png	Saint-Barthélemy	10.14	1.04
af.png	Afghanistan	31.80	1.04
pm.png	Saint-Pierre-et-Miquelon	13.80	1.04
fr.png	France	1071.58	17.65
ro.png	Roumanie	1071.58	17.89
gn.png	Guinée	1071.58	17.89
it.png	Italie	1071.85	18.28
be.png	Belgique	1074.05	22.04

En quelques mots...

Le langage Python permet une interaction facile avec le système de fichiers de l'ordinateur : lecture, écriture, compression, accès aux métadonnées. Il est possible d'automatiser en Python toutes les tâches de gestion des fichiers que l'on a l'habitude de réaliser manuellement avec le gestionnaire de fichiers de notre système d'exploitation.

Les programmes les plus simples lisent des données à partir d'un fichier, exécutent des opérations, puis retournent les résultats de manière structurée dans un autre fichier. Les autres modèles d'interaction avec le monde extérieur (entrée et sortie standard, sortie d'erreur, *sockets* de communication réseau) ont un mode de fonctionnement similaire à celui des fichiers, avec des fonctions de lecture et d'écriture de chaînes de caractères (mode texte) ou de `bytes` (mode binaire).