@IT
atmarkIT

解決！ Python
ファイル操作編

かわさきしんじ，Deep Insider 編集部［著］

※ 本 eBook の制作の都合上、Python コード中のシングルクオートやダブルクオート、バックスラッシュ（円マーク）などの記号類が、コードの実行確認に使用した Python 処理系ではシングルクオートやダブルクオート、バックスラッシュなどとして解釈されない文字となっていることがあります。コードをコピー&ペーストして使う際にはご注意ください。

テキストファイルの読み書きまとめ

テキストファイルに対する読み込みと書き込み、テキストファイルを読み書き両用でオープンする方法、エンコーディングの指定や検出の方法をまとめて紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 06 月 15 日）

ここでは「解決！Python」でこれまでに紹介してきたテキストファイルの読み書きの方法をまとめる。詳しい解説はコード例の後で紹介しているリンクを参照してほしい。

テキストファイルの読み込み

```
# test.txtファイルの内容
#atmark IT
#
#deep insider

# ファイルをオープンして、1行ずつその内容を読み込んで処理する
with open('test.txt') as f:
    for line in f:
        line = line.rstrip()  # 読み込んだ行の末尾には改行文字があるので削除
        print(line)
# 出力結果（4行目に空行が表示されるときとされないときがあるのを除き、以下同じ）
#atmark IT
#
#deep insider

# テキストファイルをオープンして、その内容を全て読み込み、クローズする
f = open('test.txt')  # f = open('test.txt', 'rt'):
s = f.read()  # ファイルの全内容が1つの文字列として返される
print(s)
f.close()

# with文と組み合わせると使い終わったとき（ブロック終了時）や
# 例外が発生したときにファイルが自動的にクローズされる
with open('test.txt') as f:
    s = f.read()

print(s)

# pathlibモジュールのPath.read_textメソッドを使う
from pathlib import Path

p = Path('test.txt')  # s = Path('test.txt').read_text()
s = p.read_text()  # ファイルのオープン、読み込み、クローズをまとめて実行
print(s)

# 改行を区切り文字として文字列を分割し、リストに格納
```

```
sl = s.split('¥n')
print(sl)  # ['atmark IT', '', 'deep insider', '']
for line in sl:
    print(line)

# readlineメソッドを使ってテキストファイルから1行ずつ内容を読み込む
with open('test.txt') as f:
    line = f.readline()
    while line:
        line = line.rstrip()
        print(line)
        line = f.readline()

# Python 3.8以降なら代入式を使ってシンプルに書ける
with open('test.txt') as f:
    while line := f.readline():
        line = line.rstrip()
        print(line)

# テキストファイルの内容をリストに読み込む
with open('test.txt') as f:
    sl = f.readlines()

print(sl)  # ['atmark IT¥n', '¥n', 'deep insider¥n']

# readlinesメソッドを使って各行を順次処理するループを形成
with open('test.txt') as f:
    for line in f.readlines():
        line = line.rstrip()
        print(line)
```

テキストファイルの読み込みについては「テキストファイルを読み込むには」を参照のこと。

テキストファイルへの書き込み

```
# ファイルを書き込み用にオープンして、ファイル先頭から文字列を書き込む
with open('test.txt', 'w') as f:  # 「with open('test.txt', 'wt') as f:」と同じ
    f.write('this is a test.¥n')  # 改行したければ改行文字を最後に付加
    sl = ['atmark IT¥n', 'deep insider¥n']
    f.writelines(sl)  # 文字列リストはwritelinesメソッドで書き込む
    x = 1
    f.write(str(x) + '¥n')  # テキストファイルに書き込めるのは文字列のみ

# 上で作成した内容の確認
from pathlib import Path

print(Path('test.txt').read_text(), end='')
# 出力結果:
```

```
#this is a test.
#atmark IT
#deep insider
#1

# with文を使わない場合
f = open('test.txt', 'w')
f.write('this is a test.¥n')  # 戻り値は書き込んだ文字数
f.close()

# pathlibモジュールのPathクラスが提供するwrite_textメソッドを使用
p = Path('test.txt')
p.write_text('foo¥nbar¥nbaz¥n')

print(p.read_text(), end='')
# 出力結果:
#foo
#bar
#baz

# ファイルを追記
with open('test.txt', 'a') as f:
    f.write('deep insider¥n')

print(Path('test.txt').read_text(), end='')
# 出力結果:
#foo
#bar
#baz
#deep insider

# ファイルを排他的に作成してオープン
with open('test.txt', 'x') as f:  # 既に存在しているのでFileExistsError例外
    pass

with open('test2.txt', 'x') as f:
    f.write('absolutely a new line!¥n')

print(Path('test2.txt').read_text(), end='')  # absolutely a new line!
```

テキストファイルへの読み込みについては「テキストファイルに書き込むには」を参照のこと。

テキストファイルを読み書き両用でオープン

```
with open('test.txt', 'w+') as f:  # ファイルを読み書き両用でオープン
    f.write('deep insider¥n')  # ファイル先頭から書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # deep insider:ファイル先頭から読み込んで表示

with open('test.txt', 'r+') as f:  # 内容を削除せずに読み書き両用でオープン
    f.write('atmark IT')  # ファイル先頭から書き込み(上書き)
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # atmark ITder:ファイル先頭から読み込んで表示

with open('test.txt', 'a+') as f:
    f.write('python¥n')  # ファイル末尾に書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # ファイル先頭から読み込んで表示
# 出力結果:
#atmark ITder
#python

with open('test.txt', 'x+') as f:  # FileExistsError例外
    pass

with open('moretest.txt', 'x+') as f:  # OK
    f.write('atmark IT¥ndeep insider¥n')  # ファイル先頭から書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # ファイル先頭から読み込んで表示
# 出力結果:
#atmark IT
#deep insider
```

モード	概要	ファイルの内容	ファイルポインタ
r+	ファイルを読み書き両用でオープンする。 指定した名前のファイルがなければFileNotFoundError例外が発生する	以前のファイルの内容を削除しない	ファイル先頭
w+	ファイルを読み書き両用でオープンまたは新規作成する	以前のファイルの内容を削除する	ファイル先頭
a+	ファイルを読み書き両用でオープンまたは新規作成する	以前のファイルの内容を削除しない	ファイル末尾
x+	ファイルを読み書き両用で排他的に新規に作成する	新規に作成	ファイル先頭

open 関数のモードに '+' を付加した場合の振る舞い

テキストファイルを読み書き両用でオープンする方法については「テキストファイルを読み書き両用にオープンするには」を参照のこと。

エンコーディングを指定して、シフト JIS などのファイルを読み書きする

```
# シフトJISエンコードのテキストファイルの読み込み
# sjis.txtの内容:このファイルはシフトJISでエンコードされています
with open('sjis.txt', encoding='shift_jis') as f:
    s = f.read()

print(s.rstrip())  # このファイルはシフトJISでエンコードされています

# UTF8エンコードのテキストファイルの読み込み
# utf8.txtの内容:このファイルはUTF-8でエンコードされています
with open('utf8.txt', encoding='utf-8') as f:
    s = f.read()

print(s.rstrip())  # このファイルはUTF-8でエンコードされています

# バイナリファイルとして読み込んだ後にエンコーディングを指定してデコード
with open('sjis.txt', 'rb') as f:
    b = f.read()

s = b.decode('shift_jis')
print(s.rstrip())  # このファイルはシフトJISでエンコードされています

# シフトJISエンコードでテキストファイルに書き込み
with open('sjis-2.txt', 'w', encoding='shift_jis') as f:
    f.write('このファイルもシフトJISでエンコードされています¥n')

with open('sjis-2.txt', encoding='shift_jis') as f:
    print(f.read().rstrip())

# UTF-8エンコードでテキストファイルに書き込み
with open('utf8-2.txt', 'w', encoding='utf-8') as f:
    f.write('このファイルもUTF-8でエンコードされています¥n')

with open('utf8-2.txt', encoding='utf-8') as f:
    print(f.read().rstrip())
```

エンコーディングの指定については「エンコーディングを指定して、シフト JIS などのファイルを読み書きするには」を参照のこと。

テキストファイルのエンコーディングを調べて、その内容を読み込む

```
# chardetによるエンコーディングの判定とテキストデータのデコード
# sjis.txtの内容:このファイルはシフトJISでエンコードされています
from chardet import detect  # 「pip install chardet」などでインストールしておく
```

```
with open('sjis.txt', 'rb') as f:  # バイナリファイルとしてファイルをオープン
    b = f.read()  # ファイルの内容を全て読み込む

print(b)  # b'¥x82¥xb1¥x82¥xcc¥x83t¥x83@¥x83C……¥x82¥xdc¥x82¥xb7¥r¥n'
enc = detect(b)  # chardet.detect関数を使ってエンコーディングを判定
print(enc)
# 出力結果:
# {'encoding': 'shift_jis', 'confidence': 0.99, 'language': 'Japanese'}

# 得られたエンコーディング情報を使ってファイルをオープンし直す
with open('sjis.txt', encoding=enc['encoding']) as f:
    s = f.read()

print(repr(s))  # 'このファイルはシフトJISでエンコードされています¥n'

# もしくは得られたエンコーディング情報を使ってバイト列をデコード
s = b.decode(encoding=enc['encoding'])
print(repr(s))  # 'このファイルはシフトJISでエンコードされています¥r¥n'

# ファイルサイズが大きい場合
from chardet.universaldetector import UniversalDetector

with open('sjis.txt', 'rb') as f:  # ファイルをバイナリファイルとしてオープン
    detector = UniversalDetector()  # UniversalDetectorオブジェクトを生成
    for line in f:  # 行末(¥n)またはEOFまでを読み込みながら、以下を繰り返す
        detector.feed(line)  # 読み込んだデータをfeedメソッドに渡す
        if detector.done:  # 判定できたらdone属性がTrueになるのでループを終了
            break
    detector.close()  # ループ終了時にUniversalDetectorオブジェクトをクローズ

print(detector.result)
# 出力結果:
# {'encoding': 'shift_jis', 'confidence': 0.99, 'language': 'Japanese'}

# UniversalDetectorオブジェクトもまとめてwith文で取り扱う
from chardet.universaldetector import UniversalDetector
from contextlib import closing

with open('sjis.txt', 'rb') as f, closing(UniversalDetector()) as detector:
    for line in f:
        detector.feed(line)
        if detector.done:
            break

print(detector.result)  # 結果を出力
```

エンコーディングの検出と、それを利用したファイルの読み込みについては「テキストファイルのエンコーディングを調べて、その内容を読み込むには（chardet パッケージ）」を参照のこと。

バイナリファイルの読み書きまとめ

バイナリファイルに対して文字列と整数を読み書きする方法、struct ／ pickle ／ shelve モジュールを使ってバイナリファイルに各種データを読み書きする方法をまとめて紹介。

かわさきしんじ，Deep Insider 編集部（2021 年 06 月 22 日）

ここでは「解決！Python」でこれまでに紹介してきたバイナリファイルの読み書きの方法をまとめる。詳しい解説はコード例の後で紹介しているリンクを参照してほしい。テキストファイルの読み書きについては「テキストファイルの読み書きまとめ」を参照されたい。

バイナリファイルの読み書き

```
# 文字列のバイナリファイルへの書き込み
with open('test.bin', 'wb') as f:
    s = 'ディープインサイダー'
    b = s.encode()  # 文字列もバイト列にエンコードする必要がある
    f.write(b)  # バイナリファイルにはバイト列しか渡せない

# 文字列のバイナリファイルからの読み込み
with open('test.bin', 'rb') as f:
    b = f.read()
    s = b.decode()

print(s)  # ディープインサイダー

# 整数のバイナリファイルへの書き込みと読み込み
from sys import byteorder

print(byteorder)  # littleもしくはbig
length = 4  # 4バイト長整数に変換する

with open('test.bin', 'wb') as f:
    n = 123456
    b = n.to_bytes(length, byteorder)  # nを4バイト長の符号なし整数に変換
    f.write(b)

# 整数のバイナリファイルからの読み込み
with open('test.bin', 'rb') as f:
    b = f.read(length)  # lengthだけバイナリファイルから読み込み
    n = int.from_bytes(b, byteorder)  # バイト列を整数に変換する

print(f'read data: {n}')  # read data: 123456

# 整数リストのバイナリファイルへの書き込みと読み込み
nlist = [1, 2, 3, 4, 5, 6]
length = 4  # 4バイト長整数
```

```
with open('test.bin', 'wb') as f:
    # 整数リストをバイト列リストに変換してファイルに書き込み
    blist = [n.to_bytes(length, byteorder) for n in nlist]
    f.writelines(blist)
    # 整数リストをバイト列に変換してファイルに書き込み
    #b = b''.join([n.to_bytes(length, byteorder) for n in nlist])
    #f.write(b)

# バイナリファイルからの整数の連続読み込み
with open('test.bin', 'rb') as f:
    result = []
    b = f.read(length)  # lengthだけバイナリファイルから読み込み
    while b:  # データがあるだけ以下を繰り返す
        n = int.from_bytes(b, byteorder)  # 読み込んだデータを整数に変換
        result.append(n)  # リストに追加
        b = f.read(length)  # lengthだけバイナリファイルから読み込み

print(result)  # [1, 2, 3, 4, 5, 6]

# バイト列を読み込んだ後にmemoryviewオブジェクトを得て、型変換を行う
with open('test.bin', 'rb') as f:
    b = f.read()

mv = memoryview(b)
result = mv.cast('i').tolist()
print(result)  # [1, 2, 3, 4, 5, 6]
```

バイナリファイルに対する読み書きについては「バイナリファイルを読み書きするには：文字列と整数編」を参照のこと。

structモジュールを使ったバイナリファイルの読み書き

```
# structモジュールを使ってバイナリファイルに書き込み
from struct import pack, unpack, calcsize, iter_unpack

person = ('かわさき', 120, 99.9)
fmt = '20sid'  # 長さ20のバイト列(20s)、整数(i)、倍精度浮動小数点(d)
b = pack(fmt, person[0].encode(), person[1], person[2])  # fmtに従ってバイト列化

with open('data.bin', 'wb') as f:
    f.write(b)

# データサイズの計算
data_size = calcsize(fmt)
print(data_size)  # 32(このデータは32バイト長)

# バイナリファイルから読み込んで、structモジュールを使って復元
with open('data.bin', 'rb') as f:
```

```
    b = f.read(data_size)

data = unpack(fmt, b)
data = (data[0].strip(b'¥x00').decode(), data[1], data[2])
print(data)  # ('かわさき', 120, 99.9)

# pathlibモジュールを使う
from pathlib import Path
p = Path('data2.bin')
b = pack(fmt, person[0].encode(), person[1], person[2])
p.write_bytes(b)

b = p.read_bytes()
data = unpack(fmt, b)
data = (data[0].strip(b'¥x00').decode(), data[1], data[2])
print(data)  # ('かわさき', 120, 99.9)

# 複数のデータをバイナリファイルに書き込み
p_list = [('かわさき', 120, 99.9),
          ('えんどう', 60, 68.3),
          ('いっしき', 25, 65.2)]

fmt = '20sid'
with open('data.bin', 'wb') as f:
    b = [pack(fmt, p[0].encode(), p[1], p[2]) for p in p_list]
    f.writelines(b)

# 複数のデータをバイナリファイルから読み込み
data_size = calcsize(fmt)

with open('data.bin', 'rb') as f:
    b = f.read()

result = [(d[0].strip(b'¥x00').decode(), d[1], d[2]) for d in iter_unpack(fmt, b)]

print(result)
```

書式指定文字	Pythonのデータ型	Cのデータ型	サイズ
x	パディング	–	–
c	長さ1のバイト列	char	1
b	整数	signed char	1
B	整数	unsigned char	1
i	整数	int	4
I	整数	unsigned int	4
f	浮動小数点数	float	4
d	浮動小数点数	double	8
s、p	文字列をバイト列に変換したもの	char[]	–

structモジュールで使われる書式指定文字(抜粋)

structモジュールを使ったバイナリファイルの読み書きについては「バイナリファイルを読み書きするには：structモジュール編」を参照のこと。

pickleモジュールを使ったバイナリファイルの読み書き

```
import pickle

favs = ['beer', 'sake']
mydata = {'name': 'かわさき', 'age': 999, 'weight': 123.4, 'favs': favs}

# pickle化してファイルに書き込み
with open('pickled.pkl', 'wb') as f:
    pickle.dump(mydata, f)

# 非pickle化
with open('pickled.pkl', 'rb') as f:
    mydata2 = pickle.load(f)
    favs2 = mydata2['favs']

print(mydata2)
# 出力結果
# {'name': 'かわさき', 'age': 999, 'weight': 123.4, 'favs': ['beer', 'sake']}

print(f'mydata2 == mydata: {mydata2 == mydata}')  # mydata2 == mydata: True
print(f'mydata2 is mydata: {mydata2 is mydata}')  # mydata2 is mydata: False

# クラスのインスタンスのpickle化
class Foo:
    def __init__(self, name, age):
        self.name = name
        self.age = age

foo = Foo('かわさき', 999)

with open('pickled.pkl', 'wb') as f:
    pickle.dump(foo, f)

# クラスのインスタンスの非pickle化
del foo  # インスタンスを削除
with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # 復元

print(f'name: {foo.name}, age: {foo.age}') # name: かわさき, age: 999

# 関数オブジェクトとクラスオブジェクトのpickle化
def hello():
    print('hello')
```

```python
with open('pickled.pkl', 'wb') as f:
    pickle.dump(Foo, f)  # 一つのファイルに複数のオブジェクトをpickle化できる
    pickle.dump(hello, f)

with open('pickled.pkl', 'rb') as f:
    Bar = pickle.load(f)  # FooクラスをBarクラスに復元
    greet = pickle.load(f)  # hello関数をgreet関数に復元

bar = Bar('bar', 101)
print(f'name: {bar.name}, age: {bar.age}')  # name: bar, age: 101
greet()  # hello

# Fooインスタンスの復元にはFooクラスが定義されている必要がある
foo = Foo('かわさき', 999)
with open('pickled.pkl', 'wb') as f:
    pickle.dump(foo, f)

del Foo, foo  # Fooクラスとそのインスタンスであるfooを削除
with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # FooクラスがないのでAttributeError例外

class Foo:  # 上とは別のFooクラスを定義してみる
    def __init__(self, a, b):
        self.a = a
        self.b = b

with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # 復元できてしまう

print(foo.a)  # AttributeError例外(復元したfooにはa属性はない)
```

pickle モジュールを使ったバイナリファイルの読み書きについては「バイナリファイルを読み書きするには：pickle 編」を参照のこと。

shelve モジュールを使ったバイナリファイルの読み書き

```
import shelve

data_file = 'mydata'
key = 'person_data'

person_data = [('kawasaki', 120), ('isshiki', 38)]

with shelve.open(data_file) as d:
    d[key] = person_data

with shelve.open(data_file) as d:
    data = d[key]
print(data)  # [('kawasaki', 120), ('isshiki', 38)]
# デフォルトでは読み書き両用でオープンされる
num_data = [1, 2, 3, 4, 5]

with shelve.open(data_file) as d:
    data = d[key]  # 読み込み
    print(data)  # [('kawasaki', 120), ('isshiki', 38)]
    d['num_data'] = num_data  # 書き込み
    nums = d['num_data']  # 読み込み
    print(nums)  # [1, 2, 3, 4, 5]

# writeback=False
more_data = ('endo', 45)

with shelve.open(data_file) as d:
    data = d[key]
    data.append(more_data)  # 読み出したデータに追加
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38)]
    d[key] = data  # 反映するには元のキーの値を置き換える必要がある
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38), ('endo', 45)]

# writeback=True
one_more_data = ('shimada', 50)

with shelve.open(data_file, writeback=True) as d:
    data = d[key]
    data.append(one_more_data)  # Shelfオブジェクトへの操作はキャッシュされる
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38), ('endo', 45), ('shimada', 50)]
    # closeメソッドかsyncメソッドの呼び出しで、キャッシュの内容が書き込まれる。
    # キャッシュサイズが大きくなると書き戻しに時間がかかる点には注意
```

shelve モジュールを使ったバイナリファイルの読み書きについては「バイナリファイルを読み書きするには：shelve 編」を参照のこと。

テキストファイルを読み込むには

open 関数や pathlib.Path クラスを使ってファイルをオープンし、その内容を読み込む方法、with 文と組み合わせる方法、テキストファイルを反復的に処理する基本パターンを紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 04 月 13 日）

```
# ファイルをオープンして、1行ずつその内容を読み込んで処理する
with open('test.txt') as f:
    for line in f:
        line = line.rstrip()  # 読み込んだ行の末尾には改行文字があるので削除
        print(line)
# 出力結果（4行目に空行が表示されるときとされないときがあるのを除き、以下同じ）
#atmark IT
#
#deep insider

# テキストファイルをオープンして、その内容を全て読み込み、クローズする
f = open('test.txt')  # f = open('test.txt', 'rt'):
s = f.read()  # ファイルの全内容が1つの文字列として返される
print(s)
f.close()

# with文と組み合わせると使い終わったとき（ブロック終了時）や
# 例外が発生したときにファイルが自動的にクローズされる
with open('test.txt') as f:
    s = f.read()

print(s)

# pathlibモジュールのPath.read_textメソッドを使う
from pathlib import Path

p = Path('test.txt')  # s = Path('test.txt').read_text()
s = p.read_text()  # ファイルのオープン、読み込み、クローズをまとめて実行
print(s)

# 改行を区切り文字として文字列を分割し、リストに格納
sl = s.split('¥n')
print(sl)  # ['atmark IT', '', 'deep insider', '']
for line in sl:
    print(line)

# readlineメソッドを使ってテキストファイルから1行ずつ内容を読み込む
with open('test.txt') as f:
    line = f.readline()
    while line:
        line = line.rstrip()
        print(line)
```

```
        line = f.readline()

# Python 3.8以降なら代入式を使ってシンプルに書ける
with open('test.txt') as f:
    while line := f.readline():
        line = line.rstrip()
        print(line)

# テキストファイルの内容をリストに読み込む
with open('test.txt') as f:
    sl = f.readlines()

print(sl)  # ['atmark IT¥n', '¥n', 'deep insider¥n']

# readlinesメソッドを使って各行を順次処理するループを形成
with open('test.txt') as f:
    for line in f.readlines():
        line = line.rstrip()
        print(line)
```

テキストファイルを読み込む基本

テキストファイルを読み込む基本的な流れは次のようになる。

1. テキストファイルをオープンする（open 関数）
2. テキストファイルの内容を読み込み、利用する
3. テキストファイルをクローズする（close メソッド）

open 関数の構文を以下に示す。以下の構文で省略されているパラメーターについては Python のドキュメント「open()」を参照されたい。

```
open(file, mode='r')
```

第 1 引数には、読み込みたいテキストファイルのファイル名を指定する。文字列でカレントディレクトリからの相対パスもしくは絶対パスでファイル名を指定するのが一般的だが、Python の pathlib.Path オブジェクト（path-like object）を渡してもよい。

第 2 引数には、ファイルをオープンするモードを指定する。省略した場合は「r」（読み込み）と「t」（テキストモード）が指定されたものとされる。そのため、テキストファイルを読み込み目的でオープンするのであれば、第 2 引数の指定は省略して、ファイル名を指定するだけでよい。

以下に「open 関数」→「ファイルからの読み込み（ファイルの利用）」→「ファイルのクローズ」という大まかな流れを示す。

```
f = open('test.txt')  # 「f = open('test.txt', 'rt')」と同じ
# …… ファイルを利用 ……
f.close()

from pathlib import Path

filepath = Path('test.txt')
f = open(filepath)  # path-like objectを使用
# …… ファイルを利用 ……
f.close()
```

ただし、ファイル操作時にはプログラム内外の現象を原因として例外が発生することもある。例外処理をきちんとしていないと、「open 関数」→「ファイルからの読み込み」→「ファイルのクローズ」という流れの途中でプログラムが終了してしまう可能性もある。with 文と組み合わせてファイルを扱うと、ブロック内の処理の終了時やブロック内で例外が発生したときには必ずファイルがクローズされるので、こちらを定型的に使うことをおすすめする。

```
with open('test.txt') as f:  # test.txtファイルをオープンして、変数fで扱う
    # …… ファイルを利用 ……
    pass

# with文が終われば、オープンしたファイルはクローズされる
```

テキストファイルの処理とは、open 関数でテキストファイルをオープンして、そのファイルを表すファイルオブジェクトを取得し、今度はそのファイルオブジェクトから何らかの形で各行のデータを取得し、それらを使って何らかの処理を行うということだ。

実は、ファイルオブジェクトはそれ自体が反復可能オブジェクトとなっているので、ファイルオブジェクトを for 文に与えるだけで、各行を反復的に処理できる。以下ではファイルオブジェクト自体を利用した反復処理、ファイルオブジェクトが持つ read ／ readline ／ readlines メソッドを使った処理を見ていく。このときには、以下のような内容のテキストファイル「test.txt」を例に使う。

```
atmark IT

deep insider
```

テキストファイルから 1 行ずつ内容を読み込む

ファイルオブジェクトを反復可能オブジェクトとして for 文に与えて、各行を反復的に処理する典型的なコードを以下に示す。なお、以降は with 文と組み合わせたファイル読み込みのコードを示す。

```
with open('test.txt') as f:
    for line in f:
        line = line.rstrip()
        print(line)
# 出力結果:
#atmark IT
#
#deep insider
```

この後に紹介する read メソッドや readlines メソッドは、ファイルの内容を全てメモリに読み込むのに対して、この方法では for 文のループが実行されるたびにファイルから 1 行だけがメモリへと読み込まれる。一度に全ての内容を読み込む必要がなければ、基本的にはテキストファイルを行ごとに処理するには、この方法を使うのがよいだろう。

注意すべき点としては、読み込んだ行の末尾には改行文字が付加されている点だ。そのため、上のコードでは文字列の rstrip メソッドを使って改行文字を削除している。この処理を省略した場合は次のようになる。各要素の末尾にある改行文字と print 関数が自動的に付加する改行文字により、以下のような結果になる。

```
with open('test.txt') as f:
    for line in f:
        print(line)
# 出力結果
#atmark IT
#

#

#
#deep insider
#
```

テキストファイルをオープンした場合、それを表すファイルオブジェクトには readline メソッドがある。これはファイルから 1 行だけを読み込むメソッドだ。これを使うと、上のコードは次のようにも記述できる。

```
with open('test.txt') as f:
    line = f.readline()  # readlineメソッドで上と同じことを行う
    while line:
        line = line.rstrip()
        print(line)
        line = f.readline()

with open('test.txt') as f:
    while line := f.readline():  # 代入式でシンプルに(Python 3.8以降)
        line = line.rstrip()
        print(line)
```

だが、こうした記述をするのであれば、先ほどのようにファイルオブジェクトを反復オブジェクトとして使う方がスッキリとするはずだ。なお、要素の末尾に改行文字が付加されるのが面倒に感じられるかもしれないが、テキストファイル中の空行には改行文字が付加される一方で、ファイル末尾以降を読み込もうとしたときには空文字列が返される。こうすることで、ファイル末尾までを読み込んだかそうでないかを判断できることは覚えておこう（上のコードを、よりシンプルにしようとして「f.readline().rstrip()」のようにすると、空行と空文字列の区別が付かなくなる）。

テキストファイルをオープンして、その内容を全て読み込み、クローズする

テキストファイルをオープンして、その内容を全て読み込み、クローズするときには read メソッドが使える。このメソッドは、テキストファイルの内容を全て読み込んで、単一の文字列として返送する。以下に例を示す。

```
with open('test.txt') as f:
    s = f.read()

print(s)
```

これにより、テキストファイルの内容全体が変数 s に代入される（改行文字を含む）。対話環境（REPL や Jupyter Notebook など）で変数 s の値を「s」のようにして評価すると、その値が見られる。

```
s  # 'atmark IT¥n¥ndeep insider¥n'
```

このように文字列全体の末尾にも改行文字があるので、「print(s)」のようにすると、最後に空行が表示される点には注意してほしい。

同様な処理は、pathlib モジュールの Path クラスのインスタンスメソッド read_text を使っても行える。ファイルのオープンやクローズを気にすることなく、テキストファイルの内容を読み込めるので、こちらの方法も覚えておくとよいかもしれない。

```
from pathlib import Path

p = Path('test.txt')  # s = Path('test.txt').read_text()
s = p.read_text()  # ファイルのオープン、読み込み、クローズをまとめて実行
print(s)
```

どちらの方法にしても、単一の文字列としてファイルの内容が返送されるが、その内容を行ごとに扱いたいのであれば、改行文字を区切り文字として文字列を分割する。

```
sl = s.split('¥n')
print(sl)  # ['atmark IT', '', 'deep insider', '']
for line in sl:
    print(line)
```

read メソッドで読み込んだ末尾に改行文字が付加されている場合、split メソッドで分割した結果（リスト）の最終要素が空文字列となる点には注意しよう。ただし、この手間を考えると最初に紹介した方法が簡潔だろう。各行を反復的に処理したいのではなく、ファイル全体の内容をさらに何かの関数やメソッドに渡して、処理したいのであれば、このメソッドを使うのがよい。

テキストファイルの内容をリストに読み込む

上の read メソッドの例で見たような、テキストファイル全体を読み込んでから、各行を分割して、それらを要素とするリストを作成したいのであれば、readlines メソッドを使う。readlines メソッドはまさにそうした処理をしてくれる。

以下に例を示す。

```
with open('test.txt') as f:
    sl = f.readlines()  # 「sl = list(f)」でもよい(説明は省略)

print(sl)  # ['atmark IT¥n', '¥n', 'deep insider¥n']
```

read メソッドで得たファイル全体の内容を split メソッドで分割したときには、改行文字が削除されていたが、こちらでは各要素の末尾に改行がある点には注意しよう。必要があれば、文字列の rstrip メソッドを呼び出すなどして、改行文字を削除する。

readlines メソッドの戻り値は各行を要素とするリストなので以下のように反復処理を行うことも可能だ。

```
with open('test.txt') as f:
    for line in f.readlines():
        line = line.rstrip()
        print(line)
```

ただし、これについても基本的にはファイルオブジェクトに対して反復処理を行うことで、同じことを実現できるので、あまり使う場面はないかもしれない。

テキストファイルに書き込むには

open 関数でファイルを書き込み用にオープンし、ファイルに文字列を書き込む方法や書き込みのモード、pathlib.Path クラスを使う方法などを紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 04 月 16 日）

```
# ファイルを書き込み用にオープンして、ファイル先頭から文字列を書き込む
with open('test.txt', 'w') as f:  # 「with open('test.txt', 'wt') as f:」と同じ
    f.write('this is a test.¥n')  # 改行したければ改行文字を最後に付加
    sl = ['atmark IT¥n', 'deep insider¥n']
    f.writelines(sl)  # 文字列リストはwritelinesメソッドで書き込む
    x = 1
    f.write(str(x) + '¥n')  # テキストファイルに書き込めるのは文字列のみ

# 上で作成した内容の確認
from pathlib import Path

print(Path('test.txt').read_text(), end='')
# 出力結果:
#this is a test.
#atmark IT
#deep insider
#1

# with文を使わない場合
f = open('test.txt', 'w')
f.write('this is a test.¥n')  # 戻り値は書き込んだ文字数
f.close()

# pathlibモジュールのPathクラスが提供するwrite_textメソッドを使用
p = Path('test.txt')
p.write_text('foo¥nbar¥nbaz¥n')

print(p.read_text(), end='')
# 出力結果:
#foo
#bar
#baz

# ファイルを追記
with open('test.txt', 'a') as f:
    f.write('deep insider¥n')

print(Path('test.txt').read_text(), end='')
# 出力結果:
#foo
#bar
#baz
```


→目次に戻る

```
#deep insider

# ファイルを排他的に作成してオープン
with open('test.txt', 'x') as f:  # 既に存在しているのでFileExistsError例外
    pass

with open('test2.txt', 'x') as f:
    f.write('absolutely a new line!¥n')

print(Path('test2.txt').read_text(), end='')  # absolutely a new line!
```

テキストファイルに書き込む基本

テキストファイルに書き込みを行う基本的な流れを以下に示す。

1. テキストファイルを書き込み用にオープンする（open 関数）
2. open 関数で取得したファイルオブジェクトを介して書き込みを行う（write ／ writelines メソッド）
3. テキストファイルをクローズする（close メソッド）

open 関数の構文を以下に示す。以下の構文で省略されているパラメーターについては、Python のドキュメント「open()」を参照されたい。

```
open(file, mode)
```

引数 file には書き込みを行う対象となるテキストファイルのファイル名を指定する。文字列でカレントディレクトリからの相対パスもしくは絶対パスでファイル名を指定するのが一般的だが、Python の pathlib.Path オブジェクト（path-like object）を渡してもよい（Path オブジェクトを open 関数に渡す例については、読み込み目的のオープンについての話ではあるが、「テキストファイルを読み込むには」を参照のこと）。

引数 mode には以下のいずれかを指定する。

- 'w'： 書き込み用にオープンする。既存のファイルをオープンしたときには、以前の内容は削除され、ファイルの先頭から新たに書き込みが行われる
- 'a'： 書き込み用にオープンする。既存のファイルをオープンしたときには、以前の内容の末尾に追加で書き込みが行われる
- 'x'： 書き込み用にファイルをゼロから新規に作成してオープンする。既存のファイルをオープンしようとしたときには、FileExistsError 例外を発生させる

テキストファイルをオープンすることを意味する 't' を上記に付加してもよい（'wt' など）。ただし、これは省略可能である（バイナリファイルをオープンするときの 'b' 指定は必須）。

以下に「open 関数」→「ファイルへの書き込み」→「ファイルのクローズ」という大まかな流れを示す。

```
f = open('test.txt', 'w')
f.write('this is a test.¥n')  # 戻り値は書き込んだ文字数
f.close()
```

ただし、ファイルへの書き込み時には、プログラム内外の現象を要因として例外が発生して、（例外処理をきちんとしていないと）ファイルをクローズする前にプログラムが終了してしまうこともある。特にファイル書き込み時にこうなると、ファイルに保存したはずのデータが保存されていないといったことも起こるかもしれない。そのため、with 文を使って、with 文のブロックの終了時やブロック内での例外発生時に必ずファイルがクローズされるようにすることをおすすめする。

```
with open('test.txt', 'w') as f:
    # ファイルへの書き込みを行う
    pass
```

open 関数の戻り値は、書き込み対象のファイルを表すファイルオブジェクトであり、このオブジェクトを介してファイルに書き込みを行う。

なお、本稿では、書き込んだファイルの内容を確認するのに、pathlib モジュールの Path クラスが持つ read_text メソッドを使用する（print 関数の引数 end に空文字列を指定しているのは、本稿の例では基本的に、改行文字付きでファイルに文字列を出力しているので、最後に余計な改行を表示しないようにするため）。

```
from pathlib import Path

print(Path('test.txt').read_text(), end='')
```

ファイルを書き込み用にオープンして、ファイル先頭から文字列を書き込む

ファイルを書き込み用にオープンし、（既存のファイルであれば、以前の内容を空にして）ファイル先頭から文字列を書き込むには、先ほども述べたように open 関数の引数 mode に 'w'（もしくは 'wt'）を指定して、ファイルをオープンすればよい。ファイルオブジェクトを得たら、write メソッドや writelines メソッドを使って書き込みを行う。

以下に例を示す。

```
with open('test.txt', 'w') as f:
    f.write('atmark IT¥n')

print(Path('test.txt').read_text(), end='')
# 出力結果:
#atmark IT
```

この例では、write メソッドでファイルに書き込みを行っている。write メソッドは文字列を引数として受け取り、ファイルに出力する。その戻り値は書き込みを行った文字数となる。上の例を見ると分かるが、write メソッドにより自動で改行文字が追加されるようなことはないので、必要なところで自分で改行文字を追加する必要がある。

複数の文字列をファイルに書き込むには writelines メソッドを使用する。

```
sl = ['atmark IT', 'deep insider']
with open('test.txt', 'w') as f:
    f.writelines([w + '¥n' for w in sl])  # 末尾に改行文字を追加して書き込み

print(Path('test.txt').read_text(), end='')
# 出力結果:
#atmark IT
#deep insider
```

この例では、2 つの文字列を要素とするリストの内容をファイルに書き込んでいるが、元の文字列リストには改行文字が付加されていないので、リスト内包表記を使って各要素の末尾に改行文字を追加している。

テキストファイルへの書き込みで重要なのは、テキストファイルには文字列型の値しか書き込めないことだ。以下の例では、整数値「1」をテキストファイルに書き込もうとしているが、これは文字列ではないので TypeError 例外が発生する。これらは何らかのロジックに従って文字列化してから書き込む必要がある（以下の 2 つ目の例では単純に str 関数で文字列化している）。

```
x = 1
with open('test.txt', 'w') as f:
    f.write(x)  # TypeError例外

with open('test.txt', 'w') as f:
    f.write(str(x) + '¥n')  # 文字列に変換して書き込み
```

そうではなく、「1」というデータそのものをファイルに書き込みたいのであれば、バイナリファイルを使う必要がある（これについては「バイナリファイルを読み書きするには：文字列と整数編」を参照されたい）。

なお、Python の pathlib モジュールが提供する Path クラスには write_text というインスタンスメソッドがある。これを使うと、ファイルのオープン／書き込み／クローズという処理をまとめて実行できるので、覚えておくとよいかもしれない。

```
p = Path('test.txt')
p.write_text('foo¥nbar¥nbaz¥n')

print(p.read_text(), end='')
# 出力結果:
#foo
#bar
#baz
```

write_text メソッドが受け取るのは文字列のみで、文字列以外の値（文字列リストや整数値など）を指定すると例外となる。よって、これはファイルをオープンして、write メソッドで書き込んで、クローズするという処理に相当するものと考えておこう。複数の文字列を書き込みたければ、以下のように改行文字を区切り文字として、文字列リストの要素を 1 個の文字列に連結する。

```
sl = ['atmark IT', 'deep insider']
s = '¥n'.join(sl)
p = Path('test.txt')
p.write_text(s)

print(p.read_text())
```

write_text メソッドを既存のファイルに対して呼び出した場合には、以前の内容は上書きされる。そのため、ファイルへの追記などの目的でこれを使用することはできない。

テキストファイルに追記

テキストファイルに追記するときには、open 関数の引数 mode に 'a'（または 'at'）を指定して、ファイルをオープンする。その後の書き込みは上と同様、write メソッドや writelines メソッドを使用する。

以下に例を示す。

```
Path('test.txt').write_text('atmark IT¥n')  # test.txtファイルに1行書き込み
print(Path('test.txt').read_text(), end='')
# 出力結果:
#atmark IT

with open('test.txt', 'a') as f:  # 追記用にオープン
    f.write('deep insider¥n')

print(Path('test.txt').read_text(), end='')
# 出力結果:
#atmark IT
#deep insider
```

この例では、最初に Path クラスのインスタンスに対して write_text メソッドを呼び出して、ファイルに書き込みを行った後、引数 mode に 'a' を指定して同じファイルをオープンし、そこに書き込みを行っている。write メソッドで書き込んだ内容が、追記されていることに注目しよう。

ファイルを排他的に作成して、書き込み用にオープン

open 関数の引数 mode に 'x' を指定した場合には、ファイルを排他的に作成して、書き込み用にオープンすることを意味する。ここでいう「排他的」とは、ファイルを作成する場所に、そのファイルが存在していないということだ（それまでに存在しなかったものを独占的に作成する、といった意味）。そのため、引数 mode に 'x' を指定して、既存のファイルをオープンしようとすると、FileExistsError 例外が発生する。

以下に例を示す。

```
with open('test.txt', 'x') as f:  # 既に存在しているのでFileExistsError例外
    pass

with open('test2.txt') as f:  # 存在しないのでFileNotFoundError例外
    pass

with open('test2.txt', 'x') as f:  # 上で存在していなかったのでOK
    f.write('absolutely a new file!¥n')

print(Path('test2.txt').read_text(), end='')  # absolutely a new file!
```

テキストファイルを読み書き両用にオープンするには

open 関数のモードに '+' を付加すると、ファイルを読み書き両用にオープンできる。'+' を付加したときの open 関数の振る舞いやコード例を紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 04 月 20 日）

```
with open('test.txt', 'w+') as f:  # ファイルを読み書き両用でオープン
    f.write('deep insider¥n')  # ファイル先頭から書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # deep insider:ファイル先頭から読み込んで表示

with open('test.txt', 'r+') as f:  # 内容を削除せずに読み書き両用でオープン
    f.write('atmark IT')  # ファイル先頭から書き込み(上書き)
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # atmark ITder:ファイル先頭から読み込んで表示

with open('test.txt', 'a+') as f:
    f.write('python¥n')  # ファイル末尾に書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # ファイル先頭から読み込んで表示
# 出力結果:
#atmark ITder
#python

with open('test.txt', 'x+') as f:  # FileExistsError例外
    pass

with open('test2.txt', 'x+') as f:  # OK
    f.write('atmark IT¥ndeep insider¥n')  # ファイル先頭から書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # ファイル先頭から読み込んで表示
# 出力結果:
#atmark IT
#deep insider
```

テキストファイルに読み書き両用にオープンするには

open 関数でファイルをオープンするときには 'r'（読み込み）、'w'（書き込み）、'a'（追記）、'x'（排他的に書き込み）のいずれかを指定するが、これに '+' を付加すると、「更新目的」として読み書き両用でファイルがオープンされる。例えば、'r' なら読み込み用にファイルがオープンされるが、'r+' なら読み込みに加えて、書き込みも行えるということだ。

なお、テキストファイルの読み込みについては「テキストファイルを読み込むには」を、テキストファイルへの書き込みについては「テキストファイルに書き込むには」を参照されたい。

'+' を付加した場合の open 関数の振る舞いがどのようになるかを以下の表に示す。

モード	概要	ファイルの内容	ファイルポインタ
r+	ファイルを読み書き両用でオープンする。 指定した名前のファイルがなければFileNotFoundError例外が発生する	以前のファイルの内容を削除しない	ファイル先頭
w+	ファイルを読み書き両用でオープンまたは新規作成する	以前のファイルの内容を削除する	ファイル先頭
a+	ファイルを読み書き両用でオープンまたは新規作成する	以前のファイルの内容を削除しない	ファイル末尾
x+	ファイルを読み書き両用で新規に作成する。 指定した名前のファイルが既に存在していればFileExistsError例外が発生する	新規に作成	ファイル先頭

open 関数のモードに '+' を付加した場合の振る舞い

以下に例を示す。

```
with open('test.txt', 'w+') as f:  # ファイルを読み書き両用でオープン
    f.write('deep insider¥n')  # ファイル先頭から書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # deep insider:ファイル先頭から読み込んで表示
```

この例では、open 関数のモードに 'w+' を指定してファイルをオープンしている。そのため、既存ファイルであればその内容は削除され、ファイルポインタはファイル先頭を指すようになる。

その後、write メソッドで書き込みを行い、seek メソッドでファイルポインタを先頭に移動しているので、最後の行の read メソッドではファイル先頭からファイルの内容が全て読み込まれる。その結果、ファイルに書き込んだ 'deep insider¥n' が画面に表示されている。

このように '+' を付加することで、ファイルに書き込みをしたり、ファイルから読み込みをしたりすることが可能になる。

```
with open('test.txt', 'r+') as f:  # 内容を削除せずに読み書き両用でオープン
    f.write('atmark IT')  # ファイル先頭から書き込み(上書き)
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # atmark ITder:ファイル先頭から読み込んで表示
```

この例では、open 関数のモードに 'r+' を指定してファイルをオープンしている。このときには、ファイルの内容は削除されず、ファイルポインタはファイル先頭を指している。そのため、ここで行っているように、'atmark IT' を write メソッドで書き込もうとすると、ファイルの内容が先頭から上書きされる。

その後、seek メソッドでファイルポインタを先頭に移動して、read メソッドで読み込みを行うと、'atmark ITder' と以前の内容に上書きをした結果が得られる。

```
with open('test.txt', 'a+') as f:
    f.write('python¥n')  # ファイル末尾に書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # ファイル先頭から読み込んで表示
# 出力結果:
#deep insider
#python
```

この例では、open 関数のモードに 'a+' を指定している。このときには、ファイルの内容は削除されず、ファイルポインタはファイル末尾を指している。そのため、write メソッドで書き込んだ内容は、ファイルの末尾に追記される。

```
with open('test.txt', 'x+') as f:  # FileExistsError例外
    pass
```

この例では、open 関数のモードに 'x+' を指定して、既存のファイル（test.txt）をオープンしようとしている。'x+' はファイルを「排他的にオープン」する（ファイルをオープンする場所に、その名前のファイルがない状態で新規に作成する）ので、この場合は FileExistsError 例外が発生する。

```
with open('test2.txt', 'x+') as f:  # OK
    f.write('atmark IT¥ndeep insider¥n')  # ファイル先頭から書き込み
    f.seek(0)  # ファイル先頭にファイルポインタを移動
    print(f.read(), end='')  # ファイル先頭から読み込んで表示
# 出力結果:
#atmark IT
#deep insider
```

この例では、同じく open 関数のモードに 'x+' を指定して、今度は存在しないファイルの名前を指定している。それまでファイルは存在していなかったので、ファイルのオープンに成功し、ファイルの内容が削除されるかどうかは関係なく、ファイルポインタはファイル先頭（＝ファイル末尾）を指す。

エンコーディングを指定して
シフト JIS などのファイルを読み書きするには

open 関数の encoding パラメーターでテキストファイルのエンコーディング方式を明示して、デフォルトエンコーディング以外の形式で符号化されているファイルを読み書きする方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2024 年 07 月 10 日）

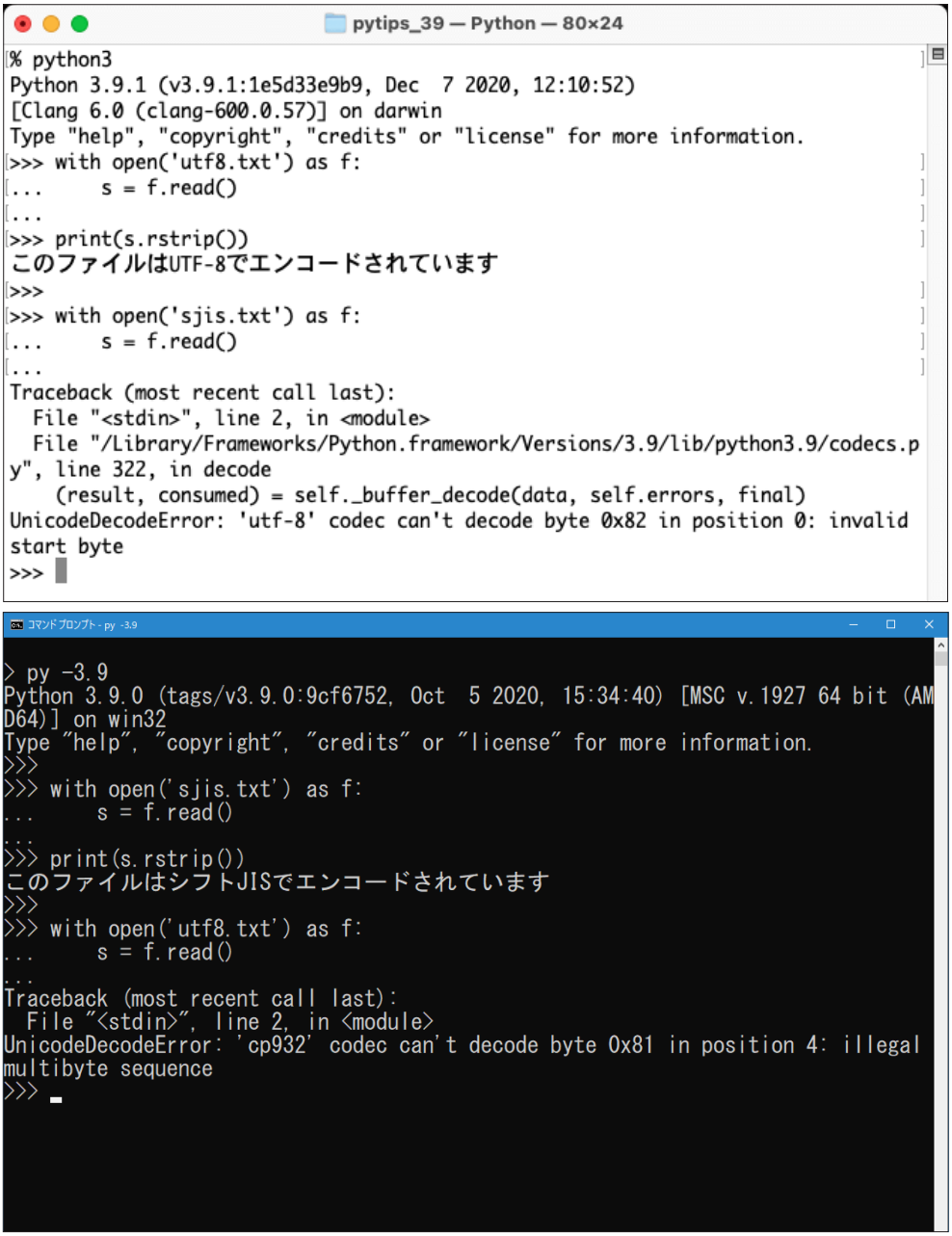
* 本稿は 2021 年 4 月 27 日に公開された記事を Python 3.12.4 で動作確認したものです（確認日：2024 年 7 月 10 日）。

```
# シフトJISエンコードのテキストファイルの読み込み
with open('sjis.txt', encoding='shift_jis') as f:
    s = f.read()

print(s.rstrip())  # このファイルはシフトJISでエンコードされています

# UTF8エンコードのテキストファイルの読み込み
with open('utf8.txt', encoding='utf-8') as f:
    s = f.read()

print(s.rstrip())  # このファイルはUTF-8でエンコードされています

# バイナリファイルとして読み込んだ後にエンコーディングを指定してデコード
with open('sjis.txt', 'rb') as f:
    b = f.read()

s = b.decode('shift_jis')
print(s.rstrip())  # このファイルはシフトJISでエンコードされています

# シフトJISエンコードでテキストファイルに書き込み
with open('sjis-2.txt', 'w', encoding='shift_jis') as f:
    f.write('このファイルもシフトJISでエンコードされています¥n')

with open('sjis-2.txt', encoding='shift_jis') as f:
    print(f.read().rstrip())

# UTF-8エンコードでテキストファイルに書き込み
with open('utf8-2.txt', 'w', encoding='utf-8') as f:
    f.write('このファイルもUTF-8でエンコードされています¥n')

with open('utf8-2.txt', encoding='utf-8') as f:
    print(f.read().rstrip())
```

open 関数とエンコーディング

Python の open 関数でテキストファイルをオープンする場合、特に指定をしない限り、コードを実行しようとしているプラットフォームごとに定められているエンコーディングを使って、そのファイルがオープンされる。例えば、macOS ならテキストファイルが UTF-8 でエンコードされていると見なされ、Windows（日本語版）なら CP932（≒シフト JIS）でエンコードされていると見なされる。

そのため、macOS から Python でシフト JIS のテキストファイルを読み込もうとすると例外が発生するし、Windows（日本語版）から Python で UTF-8 のテキストファイルを読み込もうとするとやはり例外が発生する。以下はその例だ。



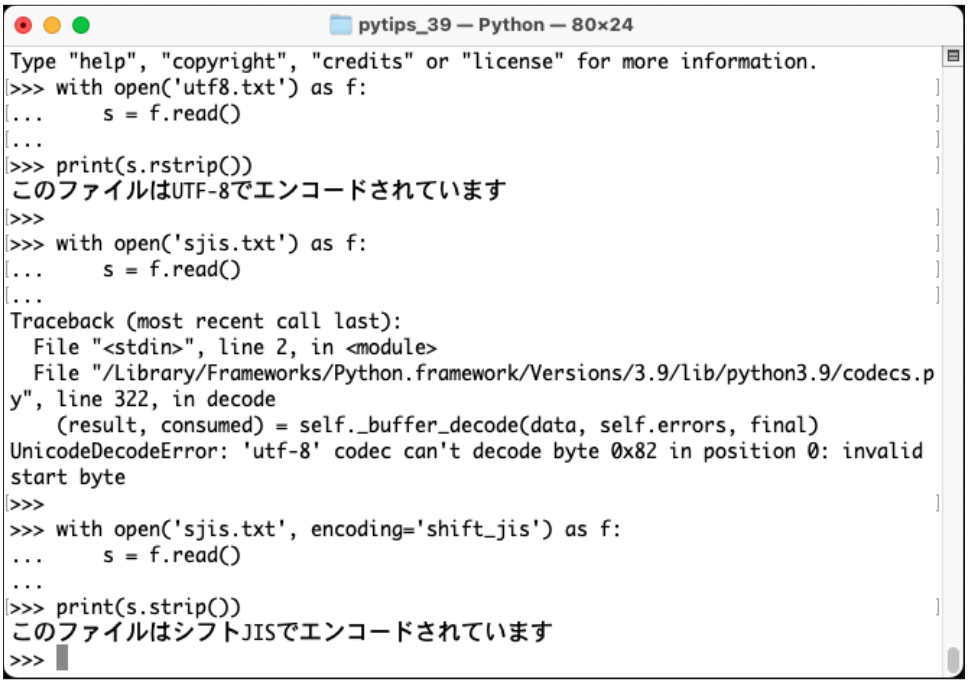
デフォルトのエンコーディングだと、macOS ではシフト JIS エンコードのテキストファイルを読み込めず、Windows では UTF-8 エンコードのテキストファイルを読み込めない

このように、手元のマシンとは別のマシンで作成されたテキストファイルが自分が普段使っているのとは異なるエンコーディングになっていることはよくある。オープンしようとしているテキストファイルのエンコーディングが分かっていれば、open 関数の encoding パラメーターで、それを明示することで、デフォルトのエンコーディングとは異なる形式のテキストファイルも読み込めるようになる。

例えば、テキストファイルを読み込む際に、それがシフト JIS でエンコードされていることを明示するには次のようにする。

```
with open('sjis.txt', encoding='shift_jis') as f:
    s = f.read()

print(s.strip())
```

これを macOS 上で実行した結果を以下に示す。

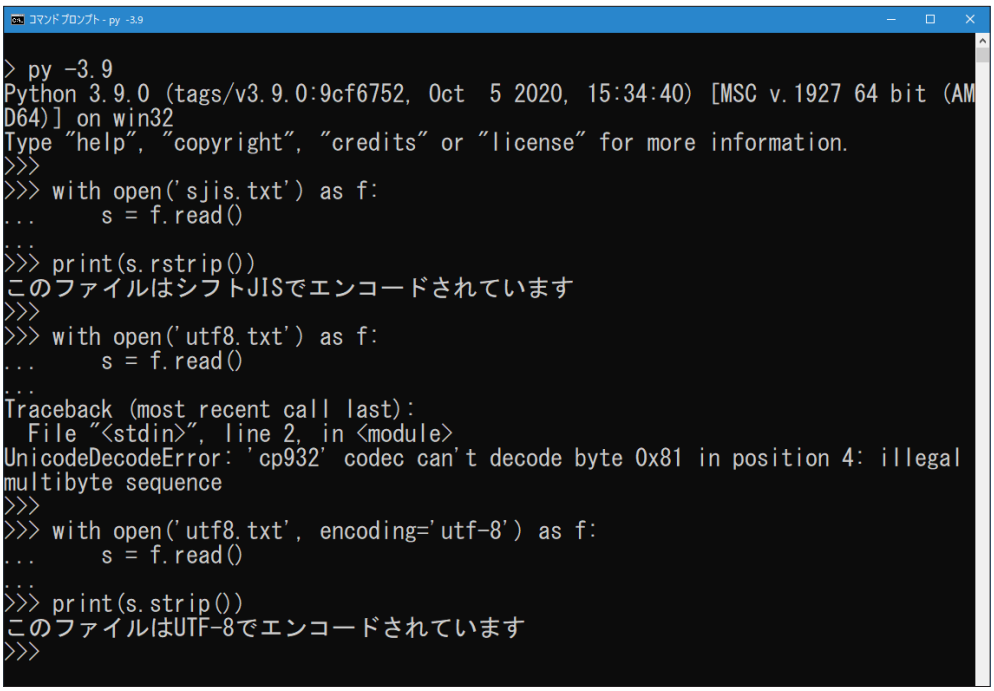


macOS からシフト JIS エンコーディングのテキストファイルをオープンできた

同様に UTF-8 エンコードされていることを明示して、ファイルをオープンするには次のようにする。

```
with open('utf8.txt', encoding='utf-8') as f:
    s = f.read()

print(s.strip())
```

これを Windows で実行した結果を以下に示す。



Windows から UTF-8 エンコーディングのテキストファイルをオープンできた

あるいはテキストファイルを「バイナリファイル」としてオープンした後に、エンコーディングを指定してデコードする方法もある。以下に例を示す。

```
with open('sjis.txt', 'rb') as f:
    b = f.read()

s = b.decode('shift_jis')
print(s.rstrip())
```

encoding パラメーターに指定できる値については「標準エンコーディング」を参照のこと。以下には一部を抜粋する（ハイフンの代わりにアンダースコア、またはその逆を記述可能。大文字小文字の区別はない）。

名前	別名
cp932	932、ms932、mskanji、ms-kanji
euc_jp	eucjp、ujis、u-jis
iso2022_jp	csiso2022jp、iso2022jp、iso-2022-jp
shift_jis	csshiftjis、shiftjis、sjis、s_jis
utf_32	U32、utf32
utf_16	U16、utf16
utf_8	U8、UTF、utf8、cp65001

open 関数でエンコーディングに指定可能な値（一部）

ファイルに書き込む場合も同様だ。以下に例を示す（実行結果は省略）。

```
with open('sjis-2.txt', 'w', encoding='shift_jis') as f:
    f.write('このファイルもシフトJISでエンコードされています¥n')

with open('sjis-2.txt', encoding='shift_jis') as f:
    print(f.read().rstrip())

with open('utf8-2.txt', 'w', encoding='utf-8') as f:
    f.write('このファイルもUTF-8でエンコードされています¥n')

with open('utf8-2.txt', encoding='utf-8') as f:
    print(f.read().rstrip())
```

ただし、ファイルを追記するようなときには、必ずテキストファイルを適切なエンコーディングでオープンするようにしよう。以下のようなコードは実行でき、ファイルを壊す。

```
# UTF-8エンコーディングのテキストファイルにシフトJISで追記
with open('utf8-2.txt', 'a', encoding='shift_jis') as f:
    f.write('シフトJIS¥n')

with open('utf8-2.txt', encoding='shift_jis') as f:
    print(f.read().rstrip())
```

テキストファイルのエンコーディングを調べて その内容を読み込むには（chardet パッケージ）

別環境で作られたテキストファイルの内容を読み込む際には、まずそのエンコーディングを調べる必要がある。chardet パッケージを使って、これを行う方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 05 月 11 日）

```
# chardetによるエンコーディングの判定とテキストデータのデコード
# sjis.txtの内容:このファイルはシフトJISでエンコーディングされています
from chardet import detect  # 「pip install chardet」などでインストールしておく

with open('sjis.txt', 'rb') as f:  # バイナリファイルとしてファイルをオープン
    b = f.read()  # ファイルの内容を全て読み込む

print(b)  # b'¥x82¥xb1¥x82¥xcc¥x83t¥x83@¥x83C……¥xa2¥x82¥xdc¥x82¥xb7¥r¥n'
enc = detect(b)  # chardet.detect関数を使ってエンコーディングを判定
print(enc)
# 出力結果:
# {'encoding': 'shift_jis', 'confidence': 0.99, 'language': 'Japanese'}

# 得られたエンコーディング情報を使ってファイルをオープンし直す
with open('sjis.txt', encoding=enc['encoding']) as f:
    s = f.read()

print(repr(s))  # 'このファイルはシフトJISでエンコーディングされています¥n'

# もしくは得られたエンコーディング情報を使ってバイト列をデコード
s = b.decode(encoding=enc['encoding'])
print(repr(s))  # 'このファイルはシフトJISでエンコーディングされています¥r¥n'

# ファイルサイズが大きい場合
from chardet.universaldetector import UniversalDetector

with open('sjis.txt', 'rb') as f:  # ファイルをバイナリファイルとしてオープン
    detector = UniversalDetector()  # UniversalDetectorオブジェクトを生成
    for line in f:  # 行末(¥n)またはEOFまでを読み込みながら、以下を繰り返す
        detector.feed(line)  # 読み込んだデータをfeedメソッドに渡す
        if detector.done:  # 判定できたらdone属性がTrueになるのでループを終了
            break
    detector.close()  # ループ終了時にUniversalDetectorオブジェクトをクローズ

print(detector.result)
# 出力結果:
# {'encoding': 'shift_jis', 'confidence': 0.99, 'language': 'Japanese'}

# UniversalDetectorオブジェクトもまとめてwith文で取り扱う
from chardet.universaldetector import UniversalDetector
from contextlib import closing
```

```
with open('sjis.txt', 'rb') as f, closing(UniversalDetector()) as detector:
    for line in f:
        detector.feed(line)
        if detector.done:
            break

print(detector.result)  # 結果を出力
```

chardet パッケージを使ってエンコーディングを検出して、その内容を読み込む

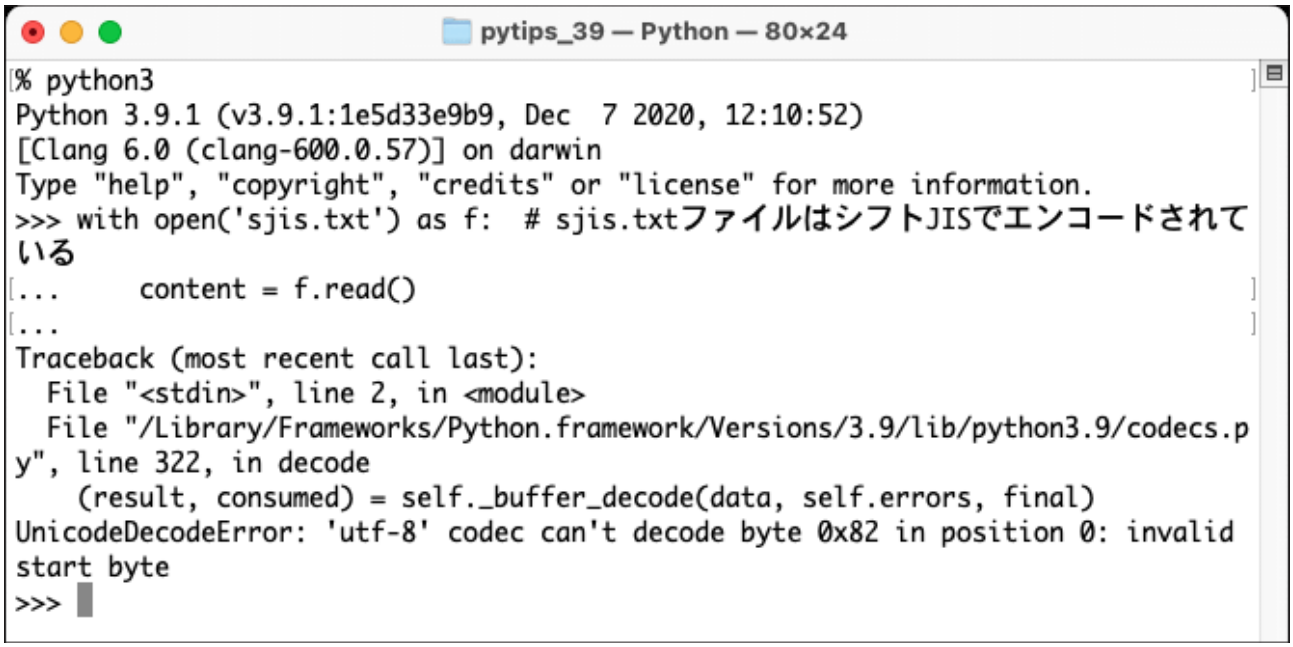
Python の open 関数でテキストファイルをオープンする場合、特に指定をしない限り、コードを実行しようとしているプラットフォームごとに定められているエンコーディングを使って、そのファイルがオープンされる。

ローカルマシンのハードディスクなどを対象として、決まったエンコーディングでファイルを読み書きしている分には問題はないが、別のマシンで作られたテキストファイルを読み書きしたり、ネットワーク経由で手に入れたテキストデータを扱ったりする際には、エンコーディングが手元のマシンのそれとは異なる場合がある。

例えば、以下はシフト JIS でエンコードされたテキストファイル（sjis.txt）の内容を macOS で読み込もうとしているところだ（ファイルの内容は「このファイルはシフト JIS でエンコーディングされています ¥n」）。

```
with open('sjis.txt') as f:  # sjis.txtファイルはシフトJISでエンコードされている
    content = f.read()
```

これを実行した結果を以下に示す。



シフト JIS 形式のファイルの読み込みに失敗したところ

この通り、macOS 上で特に指定をせずにシフト JIS エンコーディングのテキストファイルの内容を読み込もうとすると失敗する（ファイルはオープンできるが、read メソッドで例外が発生する）。Windows 環境（デフォルトのエンコーディングは「cp932」≒シフト JIS）で UTF-8 エンコーディングのテキストファイルの内容を読み込もうとしたときにも同様なエラーが発生する。

テキストファイルで使われているエンコーディングが分かっていれば、open 関数の encoding パラメーターに 'shift_jis'（や 'utf-8'）などを渡すことでファイルの内容を読み込めるようになるが（「エンコーディングを指定して、シフト JIS などのファイルを読み書きするには」を参照）、エンコーディングが分からないときには、それを判定する必要がある。

chardet パッケージはまさにこれを行ってくれる。ただし、Python に標準で添付されるパッケージではないので、「pip install chardet」などとしてあらかじめインストールしておく必要がある。

chardet パッケージの最も簡単な使い方は、エンコーディングが不明のテキストファイルを「バイナリファイルとして」オープンして、それをこのパッケージが提供する detect 関数に渡すことだ。以下に例を示す。

```
# 「pip install chardet」などを実行して、chardetパッケージをインストールしておく
from chardet import detect  # chardetパッケージからdetect関数をインポート

with open('sjis.txt', 'rb') as f:  # バイナリファイルとして読み込みオープン
    b = f.read()  # ファイルから全データを読み込み
    enc = detect(b)  # 読み込んだデータdetect関数に渡して判定する

print(enc)
# 出力結果:
#{'encoding': 'shift_jis', 'confidence': 0.99, 'language': 'Japanese'}
```

detect 関数の戻り値は辞書であり、この関数によって推測されたエンコーディングは 'encoding' キーの値となっている。上の例を見ると分かる通り、確度（'confidence' キー）と言語（'language' キー）も戻される。この場合は、エンコーディングはシフト JIS で、その確度は 99%ということだ。

エンコーディングが分かったら、それを open 関数に指定して、テキストファイルを再度オープンすると、そのエンコーディングを使ってファイルの読み書きが行われるようになる。

```
with open('sjis.txt', encoding=enc['encoding']) as f:
    s = f.read()

print(repr(s))  # 'このファイルはシフトJISでエンコーディングされています¥n'
```

あるいは、全てのデータを既に読み込み済みなので、バイト列の decode メソッドにエンコーディングを指定する方法もある。

```
s = b.decode(enc['encoding'])
print(repr(s))  # 'このファイルはシフトJISでエンコーディングされています¥r¥n'
```

2つの方法の出力結果を見ると分かるが、バイト列として読み込んだテキストデータを decode メソッドでデコードしたときには、改行文字がプラットフォーム固有の値のままである点には注意が必要だ（Windows では「¥r¥n」、macOS や Linux では「¥n」）。そのため、後者の方法でデコードしたときには、文字列の replace メソッドなどで改行文字を変換する必要があるかもしれない（エンコーディングを指定してオープンし直した場合は、デフォルトで改行コードは自動的に変換される）。

ファイルサイズが大きな場合

ファイルサイズが大きな場合に、エンコーディングを調べるためだけに全ての内容を読み込むのは無駄が大きいと考えるかもしれない。そのようなときには、chardet パッケージが提供する UniversalDetector クラスを使用する。

UniversalDetector クラスには、feed というインスタンスメソッドと done という属性がある。feed メソッドはバイト列を引数に受け取り、上で見た detect 関数と同様にエンコーディングの推定を行う。そして、ある程度の確度で推定ができると、done 属性を True にする。これらを次のような手順で使用して、エンコーディングを判定できる。

1. テキストファイルをバイナリファイルとしてオープンする
2. UniversalDetector オブジェクトを生成する
3. ファイルから行末またはファイル末尾（EOF）までをバイト列として読み込む
4. 読み込んだバイト列を UniversalDetector オブジェクトの feed メソッドに渡す
5. UniversalDetector オブジェクトの done 属性をチェックする
6. done 属性が True になる（判定できたと判断される）まで 3 ～ 5 を繰り返す
7. UniversalDetector オブジェクトの close メソッドを呼び出す
8. ファイルをクローズする
9. 判定結果は UniversalDetector オブジェクトの result 属性に格納される

これをコードにすると次のようになる。

```
from chardet.universaldetector import UniversalDetector

with open('sjis.txt', 'rb') as f:  # ファイルをバイナリファイルとしてオープン
    detector = UniversalDetector()  # UniversalDetectorオブジェクトを生成
    for line in f:  # 行末またはEOFまでを読み込みながら、以下を繰り返す
        detector.feed(line)  # 読み込んだデータをfeedメソッドに渡す
        if detector.done:  # 判定できたらdone属性がTrueになるのでループを終了
            break
    detector.close()  # ループ終了時にUniversalDetectorオブジェクトをクローズ

print(detector.result)  # 結果を出力
```

この例ではテキストファイルをバイナリファイルとしてオープンし、それを for 文に反復可能オブジェクトとして渡している。これにより行末またはファイル末尾（EOF）までを読み込みながら、for 文のブロックが実行される（バイナリファイルでは、「¥n」が改行文字として扱われる）。そのブロックの中で読み込んだバイト列を feed メソッドに渡して推測を行い、done 属性の値をチェックする。これが True であれば、エンコーディングの推測が終わったものとしてループを終了する。推測結果は UniversalDetector オブジェクトの result 属性に保存されている。そうではなくファイル末尾まで読み込んでも推測が終わらなかった場合も、ループ終了時点での推測結果が result 属性に保存されている。

なお、このコードは contextlib.closing クラスを使って、次のようにも記述できる。

```
from chardet.universaldetector import UniversalDetector
from contextlib import closing

with open('sjis.txt', 'rb') as f, closing(UniversalDetector()) as detector:
    for line in f:
        detector.feed(line)
        if detector.done:
            break

print(detector.result)
```

こうすることで、with 文のブロックが実行完了した際に、あるいはブロック内で例外が発生したときに必ず closing メソッドが呼び出されるようになる。

1 行分のデータではなく、決まったサイズのデータを feed メソッドに渡すのであれば、次のように書くことも可能だ。

```
from chardet.universaldetector import UniversalDetector
from contextlib import closing

with open('sjis.txt', 'rb') as f, closing(UniversalDetector()) as detector:
    size = 100  # 一度に読み込むデータのサイズ(バイト数)
    b = f.read(size)  # 指定したサイズだけファイルから読み込み
    while b:  # データがある間、以下を実行
        detector.feed(b)
        if detector.done:
            break
        b = f.read(size)

print(detector.result)
```


→目次に戻る

バイナリファイルを読み書きするには：文字列と整数編

バイナリファイルを読み書きする基本と、文字列および整数をバイナリファイルに書き込む方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 05 月 18 日）

```
# 文字列のバイナリファイルへの書き込み
with open('test.bin', 'wb') as f:
    s = 'ディープインサイダー'
    b = s.encode()  # 文字列もバイト列にエンコードする必要がある
    f.write(b)  # バイナリファイルにはバイト列しか渡せない

# 文字列のバイナリファイルからの読み込み
with open('test.bin', 'rb') as f:
    b = f.read()
    s = b.decode()

print(s)  # ディープインサイダー

# 整数のバイナリファイルへの書き込みと読み込み
from sys import byteorder

print(byteorder)  # littleもしくはbig
length = 4  # 4バイト長整数に変換する

with open('test.bin', 'wb') as f:
    n = 123456
    b = n.to_bytes(length, byteorder)  # nを4バイト長の符号なし整数に変換
    f.write(b)

# 整数のバイナリファイルからの読み込み
with open('test.bin', 'rb') as f:
    b = f.read(length)  # lengthだけバイナリファイルから読み込み
    n = int.from_bytes(b, byteorder)  # バイト列を整数に変換する

print(f'read data: {n}')  # read data: 123456

# 整数リストのバイナリファイルへの書き込みと読み込み
nlist = [1, 2, 3, 4, 5, 6]

with open('test.bin', 'wb') as f:
    # 整数リストをバイト列リストに変換してファイルに書き込み
    blist = [n.to_bytes(length, byteorder) for n in nlist]
    f.writelines(blist)
    # 整数リストをバイト列に変換してファイルに書き込み
    #b = b''.join([n.to_bytes(length, byteorder) for n in nlist])
    #f.write(b)
```

```
# バイナリファイルからの整数の連続読み込み
with open('test.bin', 'rb') as f:
    result = []
    b = f.read(length)  # lengthだけバイナリファイルから読み込み
    while b:  # データがあるだけ以下を繰り返す
        n = int.from_bytes(b, byteorder)  # 読み込んだデータを整数に変換
        result.append(n)  # リストに追加
        b = f.read(length)  # lengthだけバイナリファイルから読み込み

print(result)  # [1, 2, 3, 4, 5, 6]

# バイト列を読み込んだ後にmemoryviewオブジェクトを得て、型変換を行う
with open('test.bin', 'rb') as f:
    b = f.read()

mv = memoryview(b)
result = mv.cast('i').tolist()
print(result)  # [1, 2, 3, 4, 5, 6]
```

バイナリファイルを読み書きする基本

バイナリファイルを読み書きする際には、open 関数の mode 引数に 'b' を付加する（読み込みなら 'rb'、書き込みなら 'wb'）。

```
# バイナリファイルを読み込みモードでオープンする
with open('filename', 'rb') as f:
    pass  # ファイルを利用する

# バイナリファイルを書き込みモードでオープンする
with open('filename', 'wb') as f:
    pass  # ファイルを利用する
```

オープンしたファイルに対しては、テキストファイルと同様に、read メソッドや write メソッドなどを呼び出せる。

バイナリファイルを読み込み用にオープンしたときには、以下のように反復可能オブジェクトとして for 文にファイルを渡すことも可能だ。

```
with open('filename', 'rb') as f:
    for data in f:  # 行末(¥n)またはEOFまでを読み込んで以下を繰り返す
        pass  # 何かの処理を行う
```

上のような形でバイナリファイルから読み込むときには改行文字は常に「¥n」となる点は覚えておこう。

特定のバイト数だけを読み込むのであれば、read メソッドに読み込みたいバイト数を指定する。

```
with open('filename', 'rb') as f:
    size = 32
    data = f.read(size)
    while data:
        # 何かの処理を行う
        data = f.read(size)
```

バイナリファイルの読み込みを行うと、得られるのは「バイト列」（bytes 型のオブジェクト）になっている点には注意しよう。それらは何らかの形で特定の型のオブジェクトに変換する必要がある。

バイナリファイルを書き込み用にオープンしたときには、元のデータを「バイト列」に変換したものを write メソッドなどで書き込んでいく。

```
with open('filename', 'wb') as f:
    s = 'hello'
    b = s.encode()  # 文字列をバイト列に変換
    f.write(b)  # 変換後のバイト列をバイナリファイルに書き込み
```

以下では、バイナリファイルへの書き込みの例として、文字列と整数を取り上げる。これら以外のオブジェクトは次回に説明をする struct モジュールを使用することでバイナリファイルへの書き込みが比較的簡単に行えるようになる。

バイナリファイルに対して文字列を読み書きする

文字列にはバイト列への変換を行う encode メソッドがあり、バイト列には文字列への変換を行う decode メソッドがある。

バイナリファイルに文字列を書き込む例を以下に示す。

```
with open('test.bin', 'wb') as f:
    s = 'ディープインサイダー'
    b = s.encode()  # 文字列もバイト列にエンコードする必要がある
    f.write(b)
```

この例では、バイナリファイルを書き込みモードでオープンした後、文字列 s の encode メソッドを呼び出してバイト列にエンコードして、それを write メソッドで書き出している。なお、encode メソッドの encoding 引数にエンコーディングを指定することも可能だ。

次にバイナリファイルの読み込みの例を示す。

```
with open('test.bin', 'rb') as f:
    b = f.read()
    s = b.decode()

print(s)  # ディープインサイダー
```

このコードでは、バイナリファイルを読み込み用にオープンして、read メソッドで全ての内容を読み出して、取得したバイト列を decode メソッドで文字列にデコードしている。文字列の encode メソッドと同様に、バイト列の decode メソッドでもエンコーディングを指定できる。

バイナリファイルに対して整数を読み書きする

文字列とバイト列の変換には文字列の encode メソッド／バイト列の decode メソッドを使用したが、整数とバイト列の変換には整数型（int 型）の to_bytes インスタンスメソッドと from_bytes クラスメソッドを使用する。

to_bytes メソッドの構文は以下の通り。

```
int.to_bytes(length, byteorder, signed=False)
```

length 引数と byteorder 引数は指定が必須で、signed 引数は省略可能なキーワード専用引数である。length 引数は整数を何バイトのバイト列に変換するかを指定する。byteorder 引数はバイトオーダーの指定で、signed 引数は 2 の補数を使用するかどうかの指定となる（負数をバイト列に変換する場合には「signed=True」のようにキーワードを明示して指定）。

整数をバイト列に変換する例を以下に示す。

```
from sys import byteorder  # プログラムを実行するPCのバイトオーダー

print(byteorder)  # littleもしくはbig
length = 4  # 4バイト長整数に変換する
n = 123456
b = n.to_bytes(length, byteorder)
print(b)  # b'@¥xe2¥x01¥x00'もしくはb'¥x00¥x01¥xe2@'
```

バイト列を整数に変換するのに使用する int.from_bytes メソッドの構文は次の通り。

```
int.from_bytes(bytes, byteorder, signed=False)
```

bytes 引数には整数へ変換したいバイト列を渡す。byteorder ／ signed 引数は to_bytes メソッドと同じである。

上で変換したバイト列を整数に変換する例を以下に示す。

```
v = int.from_bytes(b, byteorder)
print(v)  # 123456
```

以上を踏まえて、整数のバイト列の変換とバイナリファイルへの書き込みの例を以下に示す。

```
from sys import byteorder

print(byteorder)
length = 4

with open('test.bin', 'wb') as f:
    n = 123456
    b = n.to_bytes(length, byteorder)
    f.write(b)
```

ここでは、バイナリファイルを書き込みモードでオープンし、上で見た手順で整数をバイト列に変換し、それを write メソッドで書き込んでいる。

バイト列に変換された値が書き込まれているバイナリファイルから、値を読み込む例を以下に示す。

```
with open('test.bin', 'rb') as f:
    b = f.read(length)  # lengthだけバイナリファイルから読み込み
    n = int.from_bytes(b, byteorder)  # バイト列を整数に変換する

print(f'read data: {n}')  # read data: 123456
```

重要なのは、書き込みを行った上のコードでは、整数値を 4 バイト長の符号なし整数に変換していたところだ。そのため、このコードではバイナリファイルから 4 バイトだけデータを読み込むように read メソッドに読み込むバイト数をしている。読み込んだ後は、先ほど見たように int.from_bytes メソッドを使って整数値に変換するだけだ。

バイナリファイルに対して、複数の整数値を書き込む

バイナリファイルに整数を 1 つだけ書き込むのではなく、複数の整数値を書き込むには整数値を要素とするリスト（またはタプルなど）をバイト列のリストに変換して、それを writelines メソッドで書き込むのが簡単だろう。

以下に例を示す。

```
nlist = [1, 2, 3, 4, 5, 6]

with open('test.bin', 'wb') as f:
    blist = [n.to_bytes(length, byteorder) for n in nlist]
    f.writelines(blist)
    #b = b''.join([n.to_bytes(length, byteorder) for n in nlist])
    #f.write(b)
```

ここでは、整数値を要素とするリストを基に、上で見た to_bytes メソッドを使用して各要素をバイト列に変換したバイト列リストを用意している。そして、それを writelines メソッドで一気にファイルに書き込んでいる。ただし、コメントアウトされている行のように、バイト列の join メソッドを使って、バイト列のリストを 1 つのバイト列に連結し、それを write メソッドで書き込む方法もある。

バイナリファイルから複数の整数値を読み込む例を以下に示す。

```
with open('test.bin', 'rb') as f:
    result = []
    b = f.read(length)
    while b:
        n = int.from_bytes(b, byteorder)
        result.append(n)
        b = f.read(length)

print(result)  # [1, 2, 3, 4, 5, 6]
```

この例では、上で見たように read メソッド呼び出し時に一度に読み込むサイズを指定して、バイナリファイルからバイト列を整数 1 個分ずつ読み込んで、それを int.from_bytes メソッドで整数に変換したものをリストに追加している。

データを 1 つずつ読み込むのではなく、一度に全てのデータを読み込んでから、それを memoryview オブジェクトの cast メソッドと tolist メソッドを使って以下のようにしてもよいだろう（詳細な解説は省略する）。

```
with open('test.bin', 'rb') as f:
    b = f.read()

mv = memoryview(b)
result = mv.cast('i').tolist()
print(result)
```

文字列だけ、あるいは整数値だけをバイナリファイルに書き込むには今見た方法でも十分だが、文字列や整数以外の型のデータをバイナリファイルに対して読み書きしたり、一定の構造を持ったデータをバイナリファイルに対して読み書きしたりするには次に紹介する struct モジュールを使用するのがよい。

バイナリファイルを読み書きするには：struct モジュール編

struct モジュールを使って、一定の構造を持ったデータをバイナリファイルに対して読み書きする方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 05 月 25 日）

```
# structモジュールを使ってバイナリファイルに書き込み
from struct import pack, unpack, calcsize, iter_unpack

person = ('かわさき', 120, 99.9)
fmt = '20sid'  # 長さ20のバイト列(20s)、整数(i)、倍精度浮動小数点(d)
b = pack(fmt, person[0].encode(), person[1], person[2])  # fmtに従ってバイト列化

with open('data.bin', 'wb') as f:
    f.write(b)

# データサイズの計算
data_size = calcsize(fmt)
print(data_size)  # 32(このデータは32バイト長)

# バイナリファイルから読み込んで、structモジュールを使って復元
with open('data.bin', 'rb') as f:
    b = f.read(data_size)

data = unpack(fmt, b)
data = (data[0].strip(b'¥x00').decode(), data[1], data[2])
print(data)  # ('かわさき', 120, 99.9)

# pathlibモジュールを使う
from pathlib import Path
p = Path('data2.bin')
b = pack(fmt, person[0].encode(), person[1], person[2])
p.write_bytes(b)

b = p.read_bytes()
data = unpack(fmt, b)
data = (data[0].strip(b'¥x00').decode(), data[1], data[2])
print(data)  # ('かわさき', 120, 99.9)

# 複数のデータをバイナリファイルに書き込み
p_list = [('かわさき', 120, 99.9),
          ('えんどう', 60, 68.3),
          ('いっしき', 25, 65.2)]

fmt = '20sid'
with open('data.bin', 'wb') as f:
    b = [pack(fmt, p[0].encode(), p[1], p[2]) for p in p_list]
    f.writelines(b)
```

```
# 複数のデータをバイナリファイルから読み込み
data_size = calcsize(fmt)

with open('data.bin', 'rb') as f:
    b = f.read()

result = [(d[0].strip(b'¥x00').decode(), d[1], d[2]) for d in iter_unpack(fmt, b)]

print(result)
```

struct モジュールを使ったバイナリファイルの読み書き

「バイナリファイルを読み書きするには：文字列と整数編」では、バイナリファイルに対して文字列もしくは整数を読み書きする方法を紹介した。しかし、これらが混合したデータをバイナリファイルに対して読み書きしたり、浮動小数点数をバイナリファイルに対して読み書きしたり、一定の構造を持った（C の構造体のような）データをバイナリファイルに対して読み書きしたりするには、Python が標準で提供する struct モジュールを使うのが簡単だ。

struct モジュールには、バイナリファイルに対して読み書きするデータの構造（フォーマット）に従って、データをバイト列に変換する pack 関数、バイト列からデータを復元する unpack 関数、指定したフォーマットが何バイトのデータかを計算する calcsize 関数、指定したフォーマットに従って復元したバイト列を反復する iter_unpack 関数などが含まれている。

フォーマットの指定に使える書式指定文字を幾つか以下に抜粋する。詳細については Python のドキュメント「書式指定文字」を参照のこと。

書式指定文字	Pythonのデータ型	Cのデータ型	サイズ
x	パディング	–	–
c	長さ1のバイト列	char	1
b	整数	signed char	1
B	整数	unsigned char	1
i	整数	int	4
I	整数	unsigned int	4
f	浮動小数点数	float	4
d	浮動小数点数	double	8
s、p	文字列をバイト列に変換したもの	char[]	–

struct モジュールで使われる書式指定文字（抜粋）

これらの書式指定文字の前に、バイトオーダー（リトルエンディアンやビッグエンディアンなど）やアラインメントを指示する記号を付加できるが、これについては「バイトオーダ、サイズ、アラインメント」を参照のこと（省略時は「@」つまりそのコードを実行するマシンにネイティブなバイトオーダーやアラインメントが指定されたものと見なされる）。

Python では整数や浮動小数点数にはサイズという概念がないが、これらをバイナリファイルに対して読み書きするときには、それが何バイトのデータとなるかを指定する必要がある。例えば、'i' という書式指定文字を指定すると、対応する値は 4 バイトの符号付き整数として取り扱われるということだ。なお、書式指定文字の前にはその型のデータが連続する数を指定できる。例えば、「4i」は 4 バイトの符号付き整数が 4 つ連続することを意味する（「iiii」と同じ）。

バイナリファイルに対して文字列を読み書きするときには、文字列の encode メソッドやバイト列の decode メソッドで文字列とバイト列の変換を行う必要がある。バイト列化した文字列を表現する書式指定文字は「s」となる。「s」は 1 バイトの char 型配列を、「3s」は 3 バイトの char 型配列を表すが、これでは Python の文字列を特定のエンコーディングで変換したものを格納しきれない場合がある。

例えば、UTF-8 エンコードの日本語では多くの場合、1 文字が 3 バイトで表現される（一部、4 バイトで表現されるものもある）。

```
s = 'あ'
b = s.encode()
print(f'sizeof b: {len(b)}, b: {b}')  # sizeof b: 3, b: b'¥xe3¥x81¥x82'
```

そのため、' あ ' という 1 文字を UTF-8 エンコードでバイト列に変換したものをバイナリファイルに書き込むには、3 バイトの領域が必要になる（終端文字の NULL を含めるのであれば、4 バイトが必要になるだろう）。

バイナリファイルへの書き込み

ここでは例として、次のようなデータ構造（名前、年齢、体重を要素とするタプル）を考える。

```
person = ('かわさき', 120, 99.9)
```

これらをここでは「20 バイトのバイト列」「4 バイトの符号付き整数」「倍精度の浮動小数点数」としてバイト列に変換したいとしよう。すると、上に示した書式指定文字を使って、このデータ構造は次のように書ける。

```
fmt = '20sid'  # 長さ20のバイト列(20s)、整数(i)、倍精度浮動小数点(d)
```

書式指定文字「20s」は 20 バイトのバイト列を表す（ここには 6 文字程度の日本語を格納できるだろう。余った部分には b'¥x00' が埋め込まれる）。次の「i」は 120 という整数値を 4 バイトの符号付き整数として、最後の「d」は 99.9 という浮動小数点数値を倍精度の浮動小数点数値として取り扱うことを意味する。

これを使って、タプルに格納されたデータをバイト列に変換するには struct モジュールの pack 関数を使用する。

```
pack(fmt, v1, v2, ……)
```

第 1 引数には上で見た書式指定文字を渡す。それに続けて、バイト列化したいもの（v1、v2、……）を列挙すればよい。ここではタプル person に格納されているデータを渡せばよいが、文字列については事前に encode メソッドでバイト列に変換しておく必要がある点には注意しよう。

上のデータをバイト列に変換するコードを以下に示す（バイトオーダーは実行環境にネイティブ）。

```
from struct import pack, unpack, calcsize, iter_unpack

person = ('かわさき', 120, 99.9)
fmt = '20sid'
b = pack(fmt, person[0].encode(), person[1], person[2])
# b = pack(fmt, person[0].encode(), *person[1:])
print(b)  # b'¥xe3¥x81¥x8b¥xe3¥x82¥x8f……¥x00¥x9a¥x99¥x99¥x99¥x99¥xf9X@'
```

このコードではタプルの先頭要素が文字列なので encode メソッドを呼び出しているが、タプルなどに文字列を含まないデータだけが格納されているのであれば、「b = pack(fmt, *data)」のように書ける。

あとはこれをバイナリファイルに書き込むだけだ。これらをまとめると次のようになる。

```
from struct import pack, unpack, calcsize, iter_unpack

person = ('かわさき', 120, 99.9)
fmt = '20sid'
b = pack(fmt, person[0].encode(), person[1], person[2])

with open('data.bin', 'wb') as f:
    f.write(b)
```

このデータがどのくらいのサイズになるかは、calcsize 関数に書式指定文字を渡すことで計算できる。

```
data_size = calcsize(fmt)
print(data_size)  # 32
```

バイナリファイルからの読み込み

バイナリファイルに書き込んだデータを読み込むコードは次のようになる。

```
with open('data.bin', 'rb') as f:
    b = f.read(data_size)
```

先ほどの書き込みコードではデータを 1 つだけ書き込んでいるので、2 行目では「b = f.read()」としても構わないが、ここでは calcsize 関数で計算したデータサイズだけのデータを読み込むようにしている。

バイナリファイルから読み込んだデータ（バイト列）を復元するには、struct モジュールの unpack 関数を使用する。

```
unpack(fmt, buffer)
```

第 1 引数にはバイト列がどのような構造を持つかを示す書式指定文字を指定する。第 2 引数には読み込んだバイト列を指定する。ここでは、上で書き込んだデータを復元するので、第 1 引数にはこれまでと同じ '20sid' を、第 2 引数にはバイナリファイルから読み込んだデータを指定すればよい。外部で作成されたデータを読み込むときには、バイナリデータがどのような構造になっているかを調べて、それに応じた書式指定文字を指定する必要がある。

unpack 関数は書式指定文字に従って、バイト列を復元し、復元した値を含んだタプルを返す。実際に復元したデータに文字列が含まれていれば、それはやはり decode メソッドで文字列化する必要がある。以下に実際のコードを示す。

```
data = unpack(fmt, b)
data = (data[0].strip(b'¥x00').decode(), data[1], data[2])
print(data)  # ('かわさき', 120, 99.9)
```

ここではバイト列をデコードする前に「strip(b'¥x00')」メソッドを呼び出しているが、これはバイト列から不要な部分を削除するためだ。

pathlib モジュールの Path クラスを使用する

open 関数を使わずに pathlib モジュールが提供する Path クラスを使う方法もある。Path クラスにはバイト列を書き込む write_bytes メソッドと、バイト列を読み込む read_bytes メソッドがあるので、上で見た要領で pack 関数を使いバイト列化したデータを write_bytes メソッドで書き込んで、read_bytes メソッドで読み込んだバイト列を unpack 関数を使い復元するだけだ。

以下に例を示す。

```
from pathlib import Path
p = Path('data2.bin')
b = pack(fmt, person[0].encode(), person[1], person[2])
p.write_bytes(b)

b = p.read_bytes()
data = unpack(fmt, b)
data = (data[0].strip(b'¥x00').decode(), data[1], data[2])
print(data)  # ('かわさき', 120, 99.9)
```

複数のデータを読み書きする

最後に複数のデータをバイナリファイルに対して読み書きする方法を見る。ここでは例として以下のようなタプルを要素とするリストをバイナリファイルに保存したいとする。

```
p_list = [('かわさき', 120, 99.9),
          ('えんどう', 60, 68.3),
          ('いっしき', 25, 65.2)]
```

個々のデータの構造は先ほどと同じなので、やることは文字列をエンコードして、他のデータとまとめてバイト列に変換して、それをファイルに書き込むだけだ。実際のコードは以下の通り。

```
fmt = '20sid'
with open('data.bin', 'wb') as f:
    b = [pack(fmt, p[0].encode(), p[1], p[2]) for p in p_list]
    f.writelines(b)
```

ここではリスト内包表記を使って、文字列をエンコードしたものと他のデータを pack 関数でバイト列に変換したものを要素とするリストを作って、それを writelines メソッドでファイルに書き込むようにした。

これらのデータを読み込むコードは例えば次のように書けるだろう。

```
data_size = calcsize(fmt)

with open('data.bin', 'rb') as f:
    result = []
    b = f.read(data_size)
    while b:
        tmp = unpack(fmt, b)
        result.append((tmp[0].strip(b'¥x00').decode(), tmp[1], tmp[2]))
        b = f.read(data_size)

print(result)
#出力結果:
# [('かわさき', 120, 99.9), ('えんどう', 60, 68.3), ('いっしき', 25, 65.2)]
```

このコードは、calcsize 関数で計算したサイズだけ、データを読み込みながら、それを元のデータに復元していくものだ。しかし、struct モジュールの iter_unpack 関数を使うと、バイナリファイルから全てのデータを読み込んで、それを書式指定文字に従って復元したものを列挙できる。これを使ったコードを以下に示す。

```
data_size = calcsize(fmt)

with open('data.bin', 'rb') as f:
    b = f.read()

result = [(d[0].strip(b'¥x00').decode(), d[1], d[2]) for d in iter_unpack(fmt, b)]

print(result)  # 上と同じ結果
```

iter_unpack 関数とリスト内包表記を使うと、上と同じコードをより簡潔に記述できる。

バイナリファイルを読み書きするには：pickle 編

pickle モジュールを使用して、Python のオブジェクトを直列化／復元（pickle 化／非 pickle 化、シリアライズ／デシリアライズ）する方法と、その際の注意点を紹介する。

かわさきしんじ，Deep Insider 編集部（2024 年 03 月 01 日）

* 本稿は 2021 年 6 月 1 日に公開された記事を Python 3.12.2 で動作確認したものです（確認日：2024 年 3 月 1 日）。

```
import pickle

favs = ['beer', 'sake']
mydata = {'name': 'かわさき', 'age': 999, 'weight': 123.4, 'favs': favs}

# pickle化してファイルに書き込み
with open('pickled.pkl', 'wb') as f:
    pickle.dump(mydata, f)

# 非pickle化
with open('pickled.pkl', 'rb') as f:
    mydata2 = pickle.load(f)
    favs2 = mydata['favs']

print(mydata2)
# 出力結果
# {'name': 'かわさき', 'age': 999, 'weight': 123.4, 'favs': ['beer', 'sake']}

print(f'mydata2 == mydata: {mydata2 == mydata}')  # mydata2 == mydata: True
print(f'mydata2 is mydata: {mydata2 is mydata}')  # mydata2 is mydata: False

# クラスのインスタンスのpickle化
class Foo:
    def __init__(self, name, age):
        self.name = name
        self.age = age

foo = Foo('かわさき', 999)

with open('pickled.pkl', 'wb') as f:
    pickle.dump(foo, f)

# クラスのインスタンスの非pickle化
del foo  # インスタンスを削除
with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # 復元

print(f'name: {foo.name}, age: {foo.age}') # name: かわさき, age: 999

# 関数オブジェクトとクラスオブジェクトのpickle化
```

```
def hello():
    print('hello')

with open('pickled.pkl', 'wb') as f:
    pickle.dump(Foo, f)  # 一つのファイルに複数のオブジェクトをpickle化できる
    pickle.dump(hello, f)

with open('pickled.pkl', 'rb') as f:
    Bar = pickle.load(f)  # Fooクラスを Barクラスに復元
    greet = pickle.load(f)  # hello関数をgreet関数に復元

bar = Bar('bar', 101)
print(f'name: {bar.name}, age: {bar.age}')  # name: bar, age: 101
greet()  # hello

# Fooインスタンスの復元にはFooクラスが定義されている必要がある
foo = Foo('かわさき', 999)
with open('pickled.pkl', 'wb') as f:
    pickle.dump(foo, f)

del Foo, foo  # Fooクラスとそのインスタンスであるfooを削除
with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # Fooクラスがないので AttributeError例外

class Foo:  # 上とは別のFooクラスを定義してみる
    def __init__(self, a, b):
        self.a = a
        self.b = b

with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # 復元できてしまう

print(foo.a)  # AttributeError例外(復元したfooにはa属性はない)
```

pickle モジュールとは

Python が標準で提供している「pickle モジュール」は、オブジェクトの直列化（シリアライズ）とその復元（デシリアライズ）を行うために使用できる。ここでいう直列化とは Python のオブジェクトをバイト列に変換する処理のことで、復元とはバイト列を Python のオブジェクトに変換する処理のことである。直列化によりバイト列に変換されたデータはバイナリファイルに保存したり、バイト列として他のプログラムにネットワークを介して送信したりできる。なお、pickle モジュールでオブジェクトを直列化することを pickle 化、復元することを非 pickle 化と呼ぶ。

pickle モジュールを使って、pickle 化を行うにはそのモジュールが提供する dump 関数もしくは dumps 関数を呼び出す。前者は pickle 化されたオブジェクトがバイナリファイルへ書き込まれ、後者は pickle 化された結果（バイト列）が戻り値となる。非 pickle 化には load 関数もしくは loads 関数を呼び出す。前者はバイナリファイルから pickle 化されたデータを読み込んで非 pickle 化するもので、後者はバイト列を受け取ってそれを非 pickle 化するものだ。

以下に基本的な構文を示す。

```
# pickle化
dump(obj, file, protocol=None)
dumps(obj, protocol=None)

# 非pickle化
load(file)
loads(data)
```

dump ／ dumps 関数の第 1 引数には pickle 化するオブジェクトを指定する。dump 関数では第 2 引数に pickle 化した結果のバイト列を書き込むバイナリファイル（を表すファイルオブジェクト）を指定する。dump 関数の第 3 引数と dumps 関数の第 2 引数には、pickle 化の際に使用するプロトコルのバージョンを指定する。省略時には、pickle モジュールの DEFAULT_PROTOCOL 値が指定されたものと見なされる。

2025 年 4 月現在、pickle 化／非 pickle 化に使われるプロトコルにはバージョン 0 ～ 5 の 6 種類があり、Python 3.0 ～ 3.7 では DEFAULT_PROTOCOL の値は 3、Python 3.8 以降では 4 となっている。プロトコルバージョン 5 は大きなサイズのデータを、余計なメモリコピーを行うことなく高速に pickle 化／非 pickle 化を実行するために使われるものだ（本稿では扱わない）。

load 関数の第 1 引数には非 pickle 化するデータを格納しているバイナリファイル（を表すファイルオブジェクト）を、loads 関数の第 1 引数には非 pickle 化するデータ（バイト列）を渡す。pickle 化の時点で、使用しているプロトコルのバージョンが、pickle 化されるデータストリームの先頭に書き込まれるため、load ／ loads 関数ではこれを指定する必要はない。

Python のオブジェクトを pickle 化してバイナリファイルに書き込むコードの例を以下に示す。

```
import pickle

favs = ['beer', 'sake']
mydata = {'name': 'かわさき', 'age': 999, 'weight': 123.4, 'favs': favs}

# pickle化してファイルに書き込み
with open('pickled.pkl', 'wb') as f:
    pickle.dump(mydata, f)
```

ここでは文字列、整数値、浮動小数点数値、リスト、辞書を pickle 化している（これらのオブジェクトが pickle 化可能であることを意味している）。dumps 関数を使って、バイト列に変換するなら次のようになる。

```
b = pickle.dumps(mydata)
print(b)  # b'¥x80¥x04¥x95O¥x00……¥x04beer¥x94¥x8c¥x04sake¥x94eu.'
```

上のコードを実行してバイナリファイルに書き込まれたデータから Python のオブジェクトを復元するコードの例は次のようになる。

```
with open('pickled.pkl', 'rb') as f:
    mydata2 = pickle.load(f)
    favs2 = mydata['favs']

print(mydata2)
# 出力結果
# {'name': 'かわさき', 'age': 999, 'weight': 123.4, 'favs': ['beer', 'sake']}
```

バイト列へ pickle 化したものを復元するには次のようになる。

```
mydata3 = pickle.loads(b)
print(mydata3)  # 上と同じ出力結果
```

以下を実行すると、復元されたオブジェクトは元のオブジェクトと同じ値を持つが、異なるオブジェクトであることが分かる。

```
print(f'mydata2 == mydata: {mydata2 == mydata}')  # mydata2 == mydata: True
print(f'mydata2 is mydata: {mydata2 is mydata}')  # mydata2 is mydata: False
```

pickle 化できるもの

Python のドキュメント「pickle 化、非 pickle 化できるもの」には pickle 化／非 pickle 化できるものとして以下が挙げられている。

- None 値、ブーリアン値（True ／ False）
- 整数値、浮動小数点数値、複素数値
- 文字列、バイト列（bytes オブジェクト）、バイト配列（bytearray オブジェクト）
- pickle 化可能なオブジェクトだけを要素とするリスト、タプル、辞書、集合
- モジュールトップレベルで定義された組み込み関数、関数（ラムダ式を除く）、クラス
- __dict__ 属性の値が pickle 化可能なクラスのインスタンス。または __getstate__ メソッドの戻り値が pickle 化可能なクラスのインスタンス

これら以外のオブジェクト（例えば、ファイルオブジェクトなど）は pickle 化できない。

def 文で定義した関数（関数オブジェクト）や class 文で定義したクラス自身（クラスオブジェクト）も pickle 化可能だ。ただし、関数やクラスのコードそのものが pickle 化されるのではなく、完全修飾された名前参照（それが定義されているモジュール名と関数名またはクラス名だけ）が pickle 化される点には注意すること。簡単にいうと、関数やクラスを pickle 化した場合、それを非 pickle 化する環境にはその関数やクラスを定義しているモジュール（から対応する関数またはクラス）がインポートされている必要があるということだ。

例として、クラスを定義して、そのインスタンスを pickle 化してみよう（関数やクラスの pickle 化はその後で見る）。

```
class Foo:
    def __init__(self, name, age):
        self.name = name
        self.age = age

foo = Foo('かわさき', 999)

with open('pickled.pkl', 'wb') as f:
    pickle.dump(foo, f)

del foo
with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # 復元

print(f'name: {foo.name}, age: {foo.age}') # name: かわさき, age: 999
```

このコードでは、Foo クラスを定義して、そのインスタンス foo を生成した後に pickle 化している。その後、もともとのインスタンスを削除してから、バイナリファイルからデータを読み込んで非 pickle 化している（ここでは Foo クラスが定義されているので、問題なく非 pickle 化できている）。

次に、上で定義した Foo クラスに加えて、hello 関数を定義して、今度はクラスと関数を pickle 化してみよう。実際のコードは次の通り。

```
def hello():
    print('hello')

with open('pickled.pkl', 'wb') as f:
    pickle.dump(Foo, f)  # 一つのファイルに複数のオブジェクトをpickle化できる
    pickle.dump(hello, f)

with open('pickled.pkl', 'rb') as f:
    Bar = pickle.load(f)  # FooクラスをBarクラスに復元
    greet = pickle.load(f)  # hello関数をgreet関数に復元

bar = Bar('bar', 101)
print(f'name: {bar.name}, age: {bar.age}')  # name: bar, age: 101
greet()  # hello
```

ここでは with 文のブロック内で dump 関数を二度呼び出して、1 つのファイルに pickle 化された複数のデータを書き込んでいる。このとき、Foo クラスは Bar クラスに、hello 関数は greet 関数に復元した。Bar クラスのインスタンス生成、greet 関数が成功して、Foo クラスと hello 関数と同じように振る舞っている点に注目されたい。

pickle 化では完全修飾の名前参照が用いられるので、__main__.Foo という完全修飾の名前参照を用いて復元された Bar クラスは Foo クラスと同一のオブジェクトとなる。

```
print(f'Bar is Foo: {Bar is Foo}')  # True
```

注意点

ここで、Foo クラスを削除してから、インスタンス foo を非 pickle 化してみよう（他の環境で pickled.pkl ファイルを非 pickle 化することのシミュレートともいえる）。

```
foo = Foo('かわさき', 999)
with open('pickled.pkl', 'wb') as f:
    pickle.dump(foo, f)

del Foo, foo  # Fooクラスとそのインスタンスであるfooを削除
with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # Fooクラスがないのでで AttributeError例外
```

この場合、現在のモジュール（ここでは __main__ モジュール）のトップレベルでは Foo クラス（__main__.Foo クラス）が定義されていないので、AttributeError 例外となる。

ここで上とは異なる Foo クラスを定義して、非 pickle 化したらどうなるかを実験してみよう。

```
class Foo:  # 上とは別のFooクラスを定義してみる
    def __init__(self, a, b):
        self.a = a
        self.b = b

with open('pickled.pkl', 'rb') as f:
    foo = pickle.load(f)  # 復元できてしまう

print(foo.a)  # AttributeError例外(復元したfooにはa属性はない)
```

驚いたことに復元できてしまう（__main__.Foo というクラスが存在していれば、復元が可能だからだ）。つまり、pickle 化されたデータを扱う場合、全体的な整合性を取るのはプログラマーに任されるということだ。これ以外にも、pickle 化されたデータを改ざんして、任意のコードを実行させるようにすることも可能だ。

こうしたことから、pickle モジュールは安全ではないことには注意すること。自分が知らないところで pickle 化されたファイルを安易に非 pickle 化しないようにして、非 pickle 化するときには pickle 化したときと同じ環境を整えるようにしよう。

最後にラムダ式は pickle 化できないことを確認するコードを示しておく。

```
fnc = lambda x: print(x)
fnc('hello')  # hello

pickle.dumps(fnc)  # PicklingError
```


→目次に戻る

バイナリファイルを読み書きするには：shelve 編

shelve モジュールを使って、辞書と同じ使い勝手で外部ファイルにオブジェクトを永続化したり、そこからオブジェクトを復元したりする方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 06 月 08 日）

```
import shelve

data_file = 'mydata'
key = 'person_data'

person_data = [('kawasaki', 120), ('isshiki', 38)]

with shelve.open(data_file) as d:
    d[key] = person_data

with shelve.open(data_file) as d:
    data = d[key]

print(data)  # [('kawasaki', 120), ('isshiki', 38)]

# デフォルトでは読み書き両用でオープンされる
num_data = [1, 2, 3, 4, 5]

with shelve.open(data_file) as d:
    data = d[key]  # 読み込み
    print(data)  # [('kawasaki', 120), ('isshiki', 38)]
    d['num_data'] = num_data  # 書き込み
    nums = d['num_data']  # 読み込み
    print(nums)  # [1, 2, 3, 4, 5]

# writeback=False
more_data = ('endo', 45)

with shelve.open(data_file) as d:
    data = d[key]
    data.append(more_data)  # 読み出したデータに追加
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38)]
    d[key] = data  # 反映するには元のキーの値を置き換える必要がある
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38), ('endo', 45)]

# writeback=True
one_more_data = ('shimada', 50)

with shelve.open(data_file, writeback=True) as d:
    data = d[key]
    data.append(one_more_data)  # Shelfオブジェクトへの操作はキャッシュされる
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38), ('endo', 45), ('shimada', 50)]
    # closeメソッドかsyncメソッドの呼び出しで、キャッシュの内容が書き込まれる。
    # キャッシュサイズが大きくなると書き戻しに時間がかかる点には注意
```

shelve モジュールとは

Python に標準で添付されている shelve モジュールを利用すると、Python の辞書と似た形式でオブジェクトを永続化できる。つまり「shelve オブジェクト [キー] = 値」のようにして「値」に指定したオブジェクトを外部ファイルに永続化したり、逆に「値 = shelve オブジェクト [キー]」として外部ファイルからオブジェクトを取り出したりできる（「shelve」は「棚に何かを置く」といった意味）。

このとき、「キー」には文字列を指定する。「値」には pickle 化可能なオブジェクトなら何でも指定できる（内部では pickle モジュールが使われているので、pickle モジュールと同様な安全でない操作が可能な点には注意）。

shelve モジュールを使って、永続化を行う基本的な方法はその open 関数を呼び出して、Shelf クラス（またはそのサブクラス）のインスタンスを取得して、それを用いて辞書的なアクセスを行うことだ。以下に shelve.open 関数の構文を示す。

```
shelve.open(filename, flag='c', protocol='None', writeback=False)
```

filename には内部で使用するデータベースファイルを指定する（多くの場合は、filename に指定した文字列に何らかの拡張子が付いたものが実際のファイル名になるだろう）。flag にはファイルをオープンするモードを指定する。指定できるのは以下の値。

- 'r'：読み込み専用
- 'w'：書き込み専用
- 'c'：読み書き両用
- 'n'：新規作成

指定を省略した場合には 'c' が指定されたものとして扱われ、ファイルは読み書き両用でオープンされる。protocol は内部で使用する pickle のプロトコルバージョンである。writeback を True にすると、open 関数で取得した Shelf オブジェクトに格納されているエントリ（キーと値の組）に対する操作がキャッシュされるようになる。キャッシュの内容は、close メソッドか sync メソッドを呼び出したときに外部ファイルへ書き込まれる。デフォルト値は False であり、キャッシュは行われない。それぞれの動作の違いについては後述）。

open 関数により取得した Shelf オブジェクトには辞書的にアクセスして、オブジェクトの永続化や復元を行い、Shelf オブジェクトの使用が終わったら close メソッドを呼び出す。

```
# shelveモジュールの使い方
import shelve

d = shelve.open('somefile')  # shelve.open関数でShelfオブジェクトを取得
d[somekey] = somevalue  # 書き込み(永続化)
somevalue = d[somekey]  # 復元
d.close()  # 使い終わったらcloseメソッドを呼び出す
```

shelve.open 関数は with 文と組み合わせて次のようにも書ける。

```
# with文と組み合わせる
import shelve

with shelve.open('somefile') as d:
    d[somekey] = somevalue
    somevalue = d[somekey]
```

これにより、close メソッドを明示的に呼び出す必要がなくなるので、以下ではこちらの書き方でサンプルを示す。

shelve モジュールを使ったオブジェクトの永続化と復元

以下に shelve モジュールを使ってオブジェクトの永続化と復元を行う例を示す。ここでは永続化先のファイル名は「mydata」（mydata.db）として、永続化／復元を行うデータはタプルを要素とするリストにしてある。これを 'person_data' というキーを使って永続化／復元する。

```
import shelve

data_file = 'mydata'
key = 'person_data'

person_data = [('kawasaki', 120), ('isshiki', 38)]

with shelve.open(data_file) as d:
    d[key] = person_data  # 永続化

with shelve.open(data_file) as d:
    data = d[key]  # 復元

print(data)  # [('kawasaki', 120), ('isshiki', 38)]
```

この例では、永続化（書き込み）と復元（読み込み）を別々の with 文で行っているが、先ほども述べた通り、デフォルトでは shelve.open 関数は読み書き両用でファイルをオープンする。そのため、実際には次のようなコードも記述できる。

```
num_data = [1, 2, 3, 4, 5]

with shelve.open(data_file) as d:
    data = d[key]  # 読み込み
    print(data)  # [('kawasaki', 120), ('isshiki', 38)]
    d['num_data'] = num_data  # 書き込み
    nums = d['num_data']  # 読み込み
    print(nums)  # [1, 2, 3, 4, 5]
```

writeback パラメーターの値による動作の違い

shelve.open 関数の writeback パラメーターの値を True にすると、Shelf オブジェクトに格納されているエントリの操作がキャッシュされる。一方、False にすると、キャッシュはされず、Shelf オブジェクトを介した読み込み／書き込みは外部ファイルにすぐに反映される（デフォルト）。

writeback パラメーターの値が True か False で、永続化と復元に関連する振る舞いが異なることがある。例えば、以下はこのパラメーターの値が False のときの振る舞いの例だ。

```
more_data = ('endo', 45)

with shelve.open(data_file) as d:
    data = d[key]
    data.append(more_data)  # 読み出したデータに追加
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38)]
    d[key] = data  # 反映するには元のキーの値を置き換える必要がある
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38), ('endo', 45)]
```

ここでは、先ほど書き込みを行った外部ファイルをオープンして、データを取得し（「data = d[key]」行）、それに append メソッドでデータを追加している。次の「print(d[key])」行では、直前に読み込みを行ったのと同じキーの値を出力しているが、この結果は「[('kawasaki', 120), ('isshiki', 38)]」となる。つまり、データを追加したことが外部ファイルには反映されていない。反映するには「d[key] = data」を実行する必要がある。

一方、以下は writeback パラメーターの値を True とした場合の振る舞いの例だ。

```
one_more_data = ('shimada', 50)

with shelve.open(data_file, writeback=True) as d:
    data = d[key]
    data.append(one_more_data)  # Shelfオブジェクトへの操作はキャッシュされる
    print(d[key])  # [('kawasaki', 120), ('isshiki', 38), ('endo', 45), ('shimada', 50)]
```

with 文のブロックの先頭 3 行では、同じ処理をしていることに注意されたい（追加しているデータ自体は異なる）。だが、「print(d[key])」行の出力結果は「[('kawasaki', 120), ('isshiki', 38), ('endo', 45), ('shimada', 50)]」となる。つまり、「d[key] = data」行を実行しないでも、データが反映されている（ように見える）ということだ。これが、writeback=True にした場合に、操作がキャッシュされるということだ。実際には、外部ファイルには反映はされておらず、キャッシュから適切な結果が得られるように shelve モジュールが取り計らってくれている。

writeback=True にした場合、Shelf オブジェクトを介して操作するデータを実際に扱うのではなく、そのキャッシュに対して操作を行うことになるので、最後に close メソッドを呼び出すか（または with 文のブロックが終了するか）、どこか適切なタイミングで sync メソッドを呼び出すまでは外部ファイルへのアクセスを抑制できる。上で見たように、writeback=False としたときには、永続化したデータを取得してそれを変更したら、それをどこかの時点で（「d[key] = data」行で行っているように）明示的に書き戻さないといけないが、writeback=True ではそうした処理を書かずにより直観的なコードを書けるようになる（と感じる人もいるだろう）。

ただし、writeback=True としたときには、上で見たような操作が全てキャッシュされるので、あまりに多くのエントリを読み書きするとキャッシュサイズが大きくなり、外部ファイルへそれらを反映するための時間がかかるかもしれない。キャッシュの有無で振る舞いが変わる（コードも変わる）ことと、外部ファイルへの反映にかかる時間などを考慮して、キャッシュを使用するかどうかは決めるようにしよう。

実際にどのような振る舞いになっているかは、次のようなコードからある程度は推測できる（説明は省略。コメントを参照のこと）。

```
import shelve

mydict = {}
myshelf = shelve.Shelf(mydict, writeback=True)  # 外部ファイルではなく辞書を使用

mydata = [0, 1]
key = 'key'

# 内部の辞書にデータが保存され、キャッシュされた
myshelf[key] = mydata
print(myshelf.dict)  # {b'key': b'¥x80¥x03]q¥x00(K¥x00K¥x01e.'}
print(myshelf.cache)  # {'key': [0, 1]}

# 操作の結果はキャッシュに反映されるが、辞書には反映されない
myshelf[key].append(2)
print(myshelf.dict)  # {b'key': b'¥x80¥x03]q¥x00(K¥x00K¥x01e.'}
print(myshelf.cache)  # {'key': [0, 1, 2]}

# キーを指定して取得した値はキャッシュと同一のオブジェクト
mylist = myshelf[key]
print(mylist is myshelf.cache[key])  # True

# syncメソッドにより辞書に内容が反映され、キャッシュがクリアされる
myshelf.sync()
print(myshelf.dict)  # {b'key': b'¥x80¥x03]q¥x00(K¥x00K¥x01K¥x02e.'}
print(myshelf.cache)  # {}

myshelf.close()  # Shelfオブジェクトをクローズ
```

編集：@IT 編集部
発行：アイティメディア株式会社