@IT
atmarkIT

解決！ Python
ファイルパス操作編

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。

os.path.abspath 関数で絶対パスを取得するには

os.path モジュールが提供する abspath 関数で特定のパスの絶対パスを取得する方法や、その際に注意する点、pathlib.Path.absolute メソッドとの振る舞いの違いなどを紹介する。

かわさきしんじ，Deep Insider 編集部（2024 年 10 月 08 日）

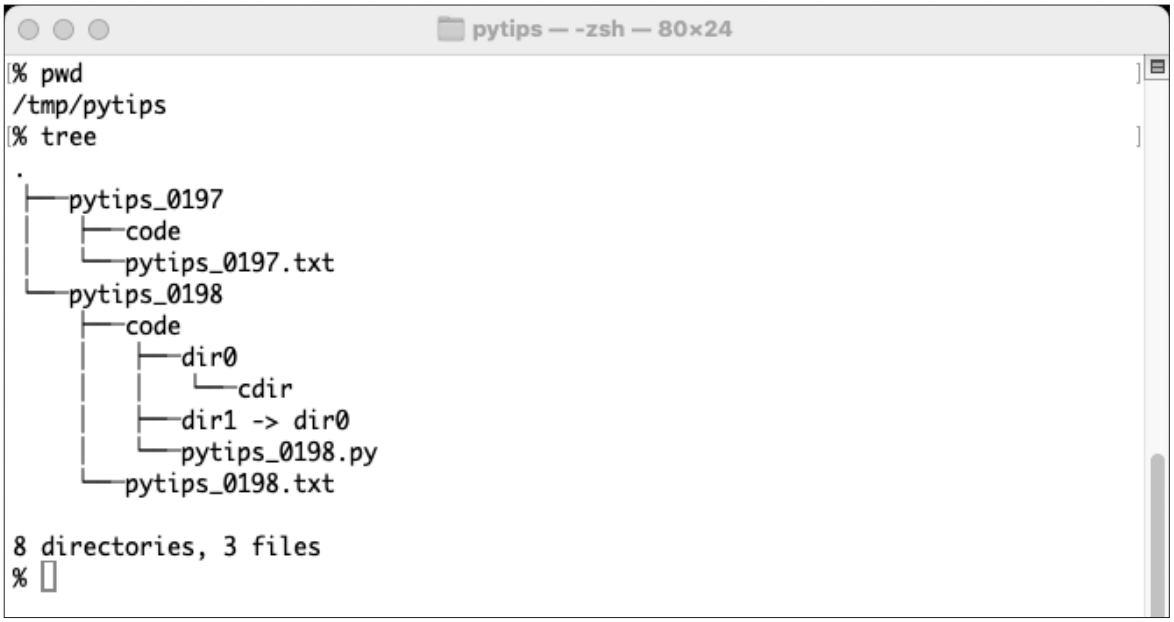
```
from os import getcwd, chdir
from os.path import abspath, join, normpath, realpath, islink

# 現在の作業ディレクトリを基点とした絶対パスを取得
cwd = getcwd()
print(cwd)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code
# Windows:C:¥tmp¥pytips¥pytips_0198¥code

p = abspath('foo.txt')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥foo.txt

tmp = normpath(join(getcwd(), 'foo.txt'))
print(tmp)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥foo.txt
print(p == tmp)  # True

p = abspath('../../pytips_0197/pytips_0197.txt')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0197/pytips_0197.txt
# Windows:C:¥tmp¥pytips¥pytips_0197¥pytips_0197.txt

# シンボリックリンクを含むパスの絶対パスを取得
print(islink('dir1'))  # True
chdir('dir1')
print(getcwd())
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir0
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir1

p = abspath('../dir1/cdir')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir1/cdir
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir1¥cdir
```

```
# pathlibモジュールのPath.absoluteメソッドとは挙動が異なることがある
from pathlib import Path

path = Path('../dir1/cdir')
p = path.absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir0/../dir1/cdir
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir1¥..¥dir1¥cdir

# シンボリックリンクを含むパスから正規の絶対パスを取得
chdir(cwd)
p = realpath('dir1/cdir')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir0/cdir
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir0¥cdir
```

os.path.abspath 関数による絶対パスの取得

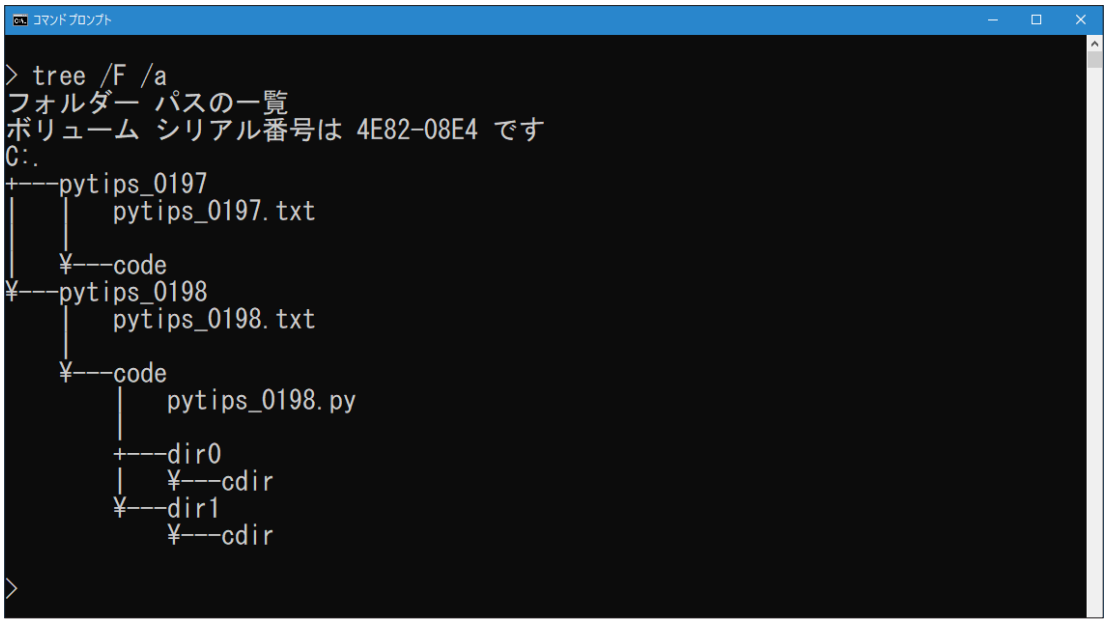
os.path モジュールが提供する abspath 関数を使うと、その関数に指定したパスの絶対パスを取得できる。ここでは macOS と Windows を例として、その使い方を見ていく。

macOS では次のようなディレクトリ階層を作成した。



macOS でサンプルとするディレクトリ階層

このうち、dir1 ディレクトリは dir0 ディレクトリに対するシンボリックリンクになっている。Windows でも同様なディレクトリ階層を作成している。



Windows でサンプルとするディレクトリ階層

こちらでも dir1 ディレクトリは dir0 ディレクトリに対するシンボリックリンクになっている（管理者権限でコマンドプロンプトを実行し、「mklink /D」コマンドで作成）。

また、os モジュールおよび os.path モジュールから以下の関数をインポートした。

```
from os import getcwd, chdir
from os.path import abspath, join, normpath, realpath, islink
```

Python のスクリプトを実行する作業ディレクトリは pytips_0198/code ディレクトリとする（後でディレクトリを移動する）。

```
cwd = getcwd()
print(cwd)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code
# Windows:C:¥tmp¥pytips¥pytips_0198¥code
```

macOS の出力結果は「/private/tmp/pytips/pytips_0198/code」となっているが、これは「/tmp/……」と読み替えてほしい（/tmp ディレクトリが /private/tmp ディレクトリへのシンボリックリンクになっているため）。

既に述べたが、abspath 関数は渡したパスの絶対パスを返す。簡単な例を以下に示す。

```
p = abspath('foo.txt')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥foo.txt
```

ここでは pytips_0198/code ディレクトリに存在しないファイルのパス「foo.txt」を渡しているが問題はない。この場合は、現在の作業ディレクトリ（os.getcwd 関数で得られるパス）と、abspath 関数に渡したパスを連結したものが絶対パスとして戻される。

os.path.abspath 関数のドキュメントにもあるが、これは normpath(join(getcwd(), 'foo.txt')) を呼び出したのと同じ結果になる。

```
tmp = normpath(join(getcwd(), 'foo.txt'))
print(tmp)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥foo.txt
print(p == tmp)  # True
```

ここで使っている normpath 関数と join 関数についても簡単に見ておこう。join 関数は 1 つ以上のパスを連結した結果を返す関数だ。ただし、渡した引数の中でルートとなる要素があった場合、その要素よりも前の要素については捨てられる点には注意しよう。以下に例を示す。

```
p = join('foo', 'bar')
print(p)
# 出力結果:
# macOS:foo/bar
# Windows:foo¥bar

p = join('foo', '/bar', 'baz')
print(p)
# 出力結果:
# macOS:/bar/baz
# Windows:/bar¥baz
```

Windows 環境では異なるドライブ文字を含んだ場合も同様にそれ以前の要素は無視される（以下の例では先頭の 'foo' は C ドライブのカレントディレクトリにある 'foo' というパスを意味するが、次の要素が 'd:bar' なので無視される。なお、この d:bar は D ドライブのカレントディレクトリにある bar というパスを表すだけで、D ドライブのルートディレクトリにある bar というパスを表しているわけではないことにも注意）。

```
p = join('foo', 'd:bar', 'baz')
print(p)  # d:bar¥baz
```

normpath 関数はパスを正規化する関数だ。ここでいう「正規化」とはカレントディレクトリを表すドット「.」や親ディレクトリを表す 2 つのドット「..」などを解決したり、連続するスラッシュを単一のスラッシュとしたりする処理だと考えればよい。

```
p = normpath('foo//bar/../baz.txt')
print(p)
# 出力結果:
# macOS:foo/baz.txt
# Windows:foo¥baz.txt
```

この例では foo と bar の間のダブルスラッシュが単一のスラッシュに変換されるとともに、親ディレクトリを意味する「..」が解決された結果、foo ディレクトリの下にある baz.txt ファイルを意味するパスが返された。

normpath 関数による正規化が行われるので、abspath 関数に「..」を含むパスを与えるとそれが正規化されたものが返される。以下に例を示す。

```
p = abspath('../../pytips_0197/pytips_0197.txt')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0197/pytips_0197.txt
# Windows:C:¥tmp¥pytips¥pytips_0197¥pytips_0197.txt
```

現在の作業ディレクトリは「/tmp/pytips/pytips_0198/code」ディレクトリなので、abspath 関数に渡した「../../pytips_0197/pytips_0197.txt」というパスは code ディレクトリの上の上のディレクトリ（pytips ディレクトリ）にある pytips_0197 ディレクトリにある pytips_0197.txt ファイルを意味し、実際にそのパスが得られている。

パスの一部（ディレクトリ）にシンボリックリンクが含まれている場合には注意が必要になる。冒頭で述べたように dir1 ディレクトリは dir0 ディレクトリに対するシンボリックリンクとなっている。これを例に見てみよう。

```
print(islink('dir1'))  # True
chdir('dir1')
print(getcwd())
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir0
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir1
```

この例では os.path.islink 関数で dir1 ディレクトリがシンボリックリンクであることを確認し、そこに作業ディレクトリを移動している。macOS ではその結果、リンク先である dir0 ディレクトリに移動したことが分かる（Windows では dir1 ディレクトリが得られている。OS による挙動の異なりであり、シンボリックリンクを扱う場合には注意が必要になるかもしれない）。

ここで親ディレクトリにある dir1 ディレクトリ（とは現在の作業ディレクトリのことだ）にある cdir というパスの絶対パスを得てみよう。

```
p = abspath('../dir1/cdir')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir1/cdir
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir1¥cdir
```

「..」を含むパスが正規化され、ここでは dir0 という要素が出てくることなく、dir1 を含むパスが得られた。これでも問題はないだろうが、シンボリックリンクを解決したパスが必要なときには以下で紹介する os.path.realpath 関数を使う必要がある。また、正規化を含む振る舞いは、pathlib モジュールの Path クラスで同様な処理を行う Path.absolute メソッドとは異なる点にも注意すること。以下は同じことを Path.absolute メソッドで試した結果だ。

```
from pathlib import Path

path = Path('../dir1/cdir')
p = path.absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir0/../dir1/cdir
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir1¥..¥dir1¥cdir
```

absolute メソッドでは「..」が解決されずに、それを含んだままの絶対パスが戻されている。こちらの方が望ましい情報を含んでいるという場合には、abspath 関数ではなく、Path.absolute メソッドを使うのがよいだろう。

最後にシンボリックリンクを含むパスから、それを解決した正規の絶対パスを realpath 関数で取得する例も示しておこう。

```
chdir(cwd)
p = realpath('dir1/cdir')
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0198/code/dir0/cdir
# Windows:C:¥tmp¥pytips¥pytips_0198¥code¥dir0¥cdir
```

pathlib.Path.absolute ／ resolve メソッドで絶対パスを取得するには

pathlib モジュールが提供する Path クラスの absolute メソッドと resolve メソッドはどちらも絶対パスを得るためのものである。その違いや使い分けについて紹介する。

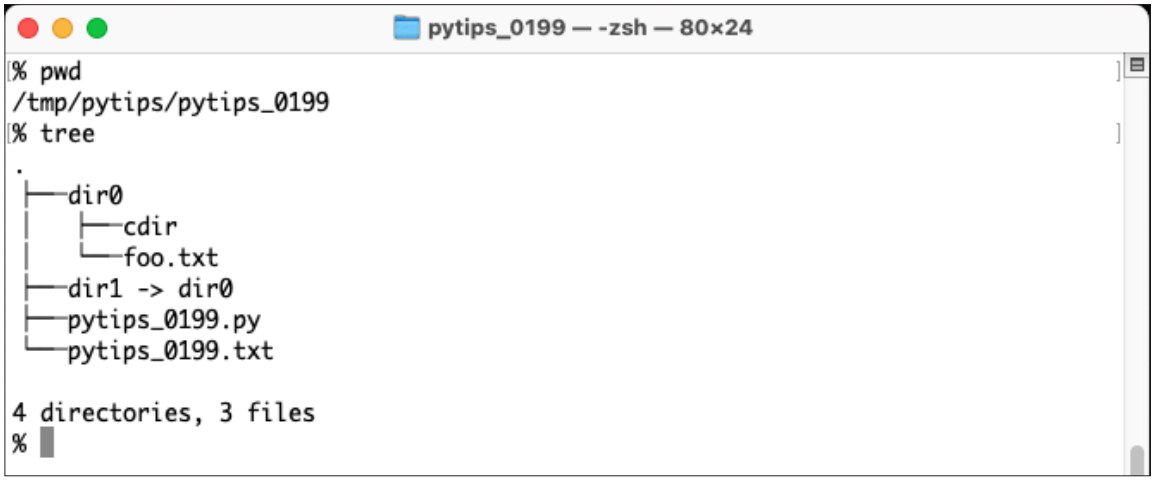
かわさきしんじ，Deep Insider 編集部（2024 年 10 月 15 日）

```
import os
from pathlib import Path

d = Path.cwd()  # d = os.getcwd()
print(d)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199
# Windows:C:¥tmp¥pytips¥pytips_0199

# カレントディレクトリを基点とした相対パスを取得
p = Path('foo.txt').absolute()
print(p)  # /private/tmp/pytips/pytips_0199/foo.txt
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥foo.txt
print(type(p))
# 出力結果:
# macOS:<class 'pathlib.PosixPath'>／<class 'pathlib._local.PosixPath'>
# Windows:<class 'pathlib.WindowsPath'>／<class 'pathlib._local.WindowsPath'>

# Path.absoluteメソッドは「..」やシンボリックリンクなどを解決しないまま返す
p = Path('dir0/../pytips_0199.txt').absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir0/../pytips_0199.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir0¥..¥pytips_0199.txt

# Path.resolveメソッドは「..」やシンボリックリンクを解決する
p = p.resolve()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/pytips_0199.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥pytips_0199.txt

# シンボリックリンクの解決
print(Path('dir1').is_symlink())  # True

p = Path('dir1/foo.txt').absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir1/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir1¥foo.txt
```


→目次に戻る

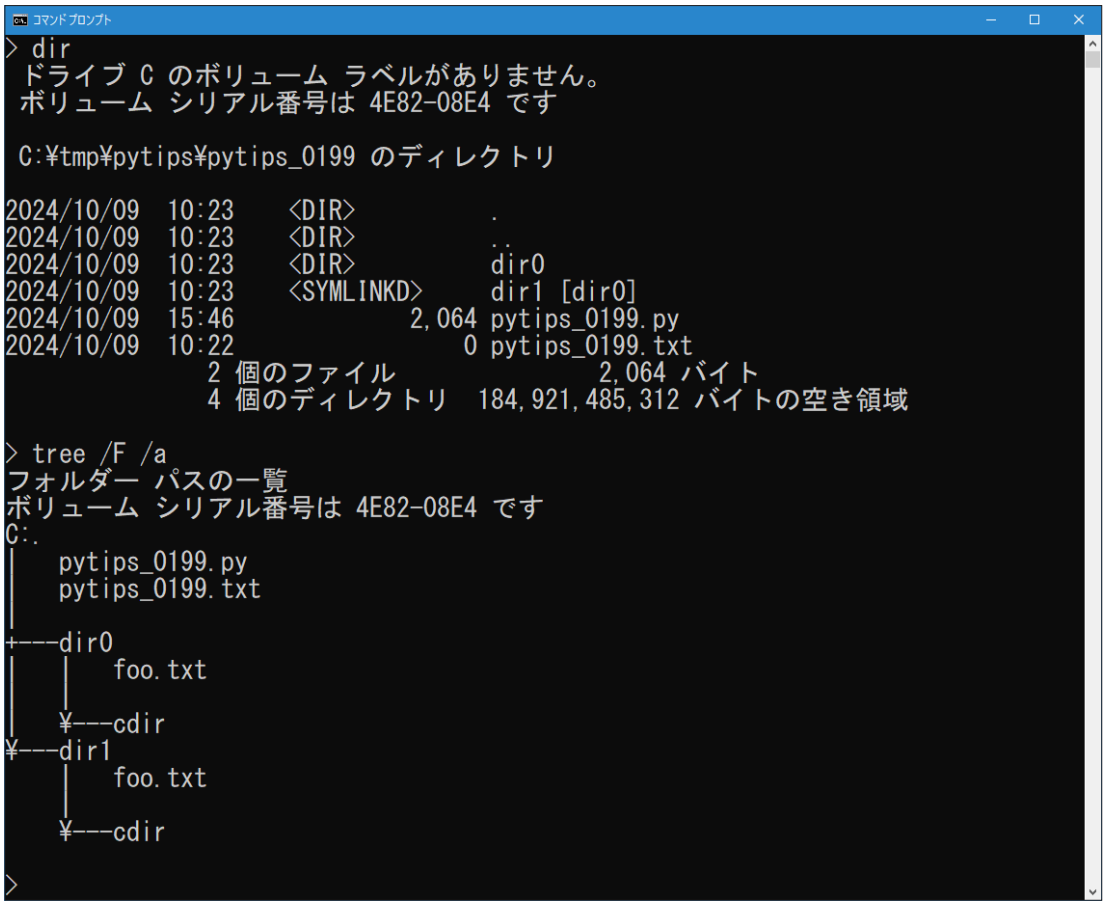
```
os.chdir('dir1')
p = Path('foo.txt').absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir0/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir1¥foo.txt

os.chdir('..')

p = Path('dir1/foo.txt').resolve()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir0/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir0¥foo.txt
```

以下では次のようなディレクトリ階層で pytips_0199 ディレクトリが現在の作業ディレクトリとなっているものとする。macOS では /tmp/pytips ディレクトリ以下に次のようなディレクトリ階層を作成した。dir1 ディレクトリは dir0 ディレクトリへのシンボリックリンクとなっている。



macOS でのサンプルのディレクトリ階層

Windows では C:¥tmp¥pytips ディレクトリ以下に次のようなディレクトリ階層を作成した。



Windows でのサンプルのディレクトリ階層

macOS のサンプルディレクトリ階層と同様、dir1 ディレクトリは dir0 ディレクトリへのシンボリックリンクとなっている（管理者権限でコマンドプロンプトを開き「mklink /D dir1 dir0」コマンドで作成）。

また、事前に os モジュールをインポートし、pathlib モジュールから Path クラスもインポートしておく。

```
import os
from pathlib import Path

d = Path.cwd()  # d = os.getcwd()
print(d)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199
# Windows:C:¥tmp¥pytips¥pytips_0199
```

macOS での Path.cwd クラスメソッドの出力結果を見ると、「/tmp/……」ではなく「/private/tmp/……」となっているのは /tmp が /private/tmp へのシンボリックリンクとなっているからなので、以降ではこれを /tmp と読み替えてほしい。

pathlib.Path.absolute メソッドによる絶対パスの取得

pathlib モジュールにはファイルパスを表す Path クラス（とその具象クラス）があり、そのメソッドとしてパス関連の機能が用意されている。現在の作業ディレクトリを基点として、特定のパスの絶対パスを取得するにはそれらのうちの absolute メソッドを使用する。

以下にシンプルな例を示す。

```
p = Path('foo.txt').absolute()
print(p)  # /private/tmp/pytips/pytips_0199/foo.txt
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥foo.txt
```

ここでは foo.txt というパスを表すオブジェクトを作成し、absolute メソッドを呼び出している（現在の作業ディレクトリにはこのファイルは存在しないが、抽象的なパス操作をしている分には問題はない）。これにより、現在の作業ディレクトリを基点とした foo.txt の絶対パスが返される。

返されるのは Path クラス（の具象クラス）のオブジェクトである（os.path モジュールの関数は文字列を返す）。

```
print(type(p))
# 出力結果:
# macOS:<class 'pathlib.PosixPath'>／<class 'pathlib._local.PosixPath'>
# Windows:<class 'pathlib.WindowsPath'>／<class 'pathlib._local.WindowsPath'>
```

なお、2025 年 6 月 30 日の時点では、Python 3.13.5 環境でこれを実行すると、pathlib._local.PosixPath や pathlib._local.WindowsPath のように「_local」を間に含むようになる（Python 3.13 では pathlib モジュールに手が加えられた結果と思われる）。

Path.absolute メソッドは自身のパスに現在のディレクトリを表すドット「.」や親ディレクトリを表す「..」、シンボリックリンクなどが含まれていても、それらを解決しない点には注意しよう。以下に例を示す。

```
p = Path('dir0/../pytips_0199.txt').absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir0/../pytips_0199.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir0¥..¥pytips_0199.txt
```

この例ではカレントディレクトリの下にある dir0 ディレクトリ、dir0 の親ディレクトリを示す「..」、最後に pytips_0199.txt という 3 つの要素で構成されるパスを表す Path オブジェクトを作成して、absolute メソッドを呼び出している。このとき、「..」が解決されていれば、得られる絶対パスは「/private/tmp/pytips/pytips_0199/pytips_0199.txt」「C:¥tmp¥pytips¥pytips_0199¥pytips_0199.txt」のようになるが、実際に得られるのは「..」入りのパスとなる。

このようなパスから「..」などを削除して、シンボリックリンクを解決した絶対パスを得たいのであれば、Path.resolve メソッドを使用する。以下は「..」を含むパスからそれを削除する例だ。

```
p = p.resolve()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/pytips_0199.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥pytips_0199.txt
```

次にシンボリックリンクを解決する例も見ておこう。既に述べた通り、dir1 ディレクトリは dir0 ディレクトリへのシンボリックリンクである。そのため、そのパスに対して、Path.is_symlink メソッドを呼び出すと True が返される。

```
print(Path('dir1').is_symlink())  # True
```

このシンボリックリンクを含む Path('dir1/foo.txt') というパスに対して、absolute メソッドを呼び出すと次のようになる。

```
p = Path('dir1/foo.txt').absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir1/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir1¥foo.txt
```

macOS でも Windows でもシンボリックリンクは解決されずに、そのまま dir1 として絶対パスが返されている。

この状態で、dir1 ディレクトリに移動して、そこにある foo.txt ファイルを指すパスである Path('foo.txt') で absolute メソッドを呼び出したのが以下だ。

```
os.chdir('dir1')
p = Path('foo.txt').absolute()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir0/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir1¥foo.txt
```

macOS では dir1 ではなく、dir0 が絶対パスに含まれるようになった。Windows ではシンボリックリンクである dir1 が絶対パスに含まれたままとなる（プラットフォームごとの挙動の異なり）。

ここで、ディレクトリを元に戻して、Path('dir1/foo.txt') というパスに対して resolve メソッドを呼び出すと、今度はシンボリックリンクが解決され、dir0 を含む絶対パスが（macOS でも Windows でも）取得できた。

```
os.chdir('..')

p = Path('dir1/foo.txt').resolve()
print(p)
# 出力結果:
# macOS:/private/tmp/pytips/pytips_0199/dir0/foo.txt
# Windows:C:¥tmp¥pytips¥pytips_0199¥dir0¥foo.txt
```

普段は absolute メソッドで絶対パスを取得すればよいだろうが、パスの正規化（「..」の削除など）が必要になったり、シンボリックリンクを解決したりする必要があるときには resolve メソッドを呼び出す必要がある。

os.path モジュールの exists 関数を使ってパスが存在するかどうかを確認するには

exists 関数に渡したパスが実際に存在するかどうかを確認したり、isdir ／ isfile 関数と組み合わせて処理を振り分けたりする方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2022 年 10 月 25 日）

```
import os
from os.path import exists, splitext, isdir, isfile

mydir = 'mydir'
myfile = 'myfile.txt'
nofile_or_dir = 'nofile_or_dir'

# mydirディレクトリとmyfile.txtファイルを作成
os.mkdir(mydir)

with open(myfile, 'w'):
    pass

print(exists(mydir))  # True
print(exists(myfile))  # True
print(exists(nofile_or_dir))  # False

# ディレクトリが存在しなければ作成する
if not exists(mydir):
    os.mkdir(mydir)
else:
    print(f'{mydir} already exists')

# ファイルが存在していればバックアップを取り、書き込みを行う
if exists(myfile):
    newfile = splitext(myfile)[0] + '.bak'
    os.rename(myfile, newfile)

with open(myfile, 'w') as f:
    f.write('some text')

# 指定されたファイル／ディレクトリの存在確認の後、処理を振り分ける
def do_some_work(path):
    if not exists(path):
        res = input(f'{path} not exists. create it? (y/n)')
        if res == 'y':
            with open(path, 'w'):
                pass

    if isdir(path):
        print(f'{path} is a directory')
```


→目次に戻る

```
    elif isfile(path):
        print(f'{path} is a file')
    else:
        print(f'{path} is not a directory nor file')

paths = [mydir, myfile, nofile_or_dir]
for path in paths:
    do_some_work(path)
```

os.path モジュールの exists 関数

os.path モジュールの exists 関数は引数に指定したパス（文字列または pathlib モジュールで定義される Path クラスのインスタンスなど）が実際に存在するディレクトリやファイルを指しているかどうかを調べるのに使える。パスが存在していれば True が、そうでなければ False が返される。

以下に例を示す。

```
import os
from os.path import exists, splitext, isdir, isfile

mydir = 'mydir'
myfile = 'myfile.txt'
nofile_or_dir = 'nofile_or_dir'

# mydirディレクトリとmyfile.txtファイルを作成
os.mkdir(mydir)

with open(myfile, 'w'):
    pass

print(exists(mydir))  # True
print(exists(myfile))  # True
print(exists(nofile_or_dir))  # False
```

このコードではまず mydir ディレクトリを os モジュールの mkdir 関数を使って作成し、その後、myfile.txt ファイルを作成している。変数 nofile_or_dir の値である「nofile_or_dir」という名前のファイルやディレクトリは存在していない。なお、os.pth モジュールを使用したディレクトリの作成については「ディレクトリを作成／削除するには：os モジュール編」を、open 関数を使ったファイルの作成については「ファイルを作成／削除するには」を参照されたい。

最後の 3 行では exists 関数を使って 3 つのパス（mydir、mydir.txt、nofile_or_dir）についてディレクトリやファイルが存在しているかを試している。最初の 2 つについてはその上で作成しているので True が、最後の nofile_or_dir についてはそのような名前のディレクトリもファイルもないので False が返されている。

os モジュールの mkdir 関数は指定した名前のディレクトリ（またはファイル）が既に存在しているときには例外を発生させる。これを回避するには、exists 関数で指定したパスが存在するかどうかを確認するとよい。以下に例を示す。

```
if not exists(mydir):
    os.mkdir(mydir)
else:
    print(f'{mydir} already exists')
```

また、ファイルを新規に作成して、そこに何かを書き込みたいが、同名のファイルがあればそのファイルの拡張子を「.bak」に変更する必要があるといった場合には次のようなコードが書ける。

```
if exists(myfile):
    newfile = splitext(myfile)[0] + '.bak'
    os.rename(myfile, newfile)

with open(myfile, 'w') as f:
    f.write('some text')
```

最初の if 文では exists 関数で変数 myfile が指すパスが存在しているかどうかを確認している。その名前のディレクトリまたはファイルが存在していれば、os.path モジュールの splitext 関数を使って変数 myfile の値である「myfile.txt」を拡張子とそれ以外の部分に分離し（splitext 関数の戻り値は「(拡張子以外 , 拡張子)」というタプル）、拡張子以外の部分と '.bak' を結合してバックアップファイルの名前を作成して、os.rename 関数で名前を変更している。その後、元のファイル（myfile.txt）を書き込みモードでオープンしてテキストを書き込んでいる（os.rename 関数でファイル名を変更しているので、ファイルをオープンするまでは myfile.txt ファイルは存在していないことには注意）。

このように「パスの存在確認」に続けて何かの処理を行うことはよくあるだろう。そのときには「os.path モジュールの isdir ／ isfile 関数を使ってパスがディレクトリやファイルであるかどうかを調べるには」で紹介した isdir ／ isfile 関数と組み合わせることが考えられる。

以下に例を示す。

```
def do_some_work(path):
    if not exists(path):
        res = input(f'{path} not exists. create it? (y/n)')
        if res == 'y':
            with open(path, 'w'):
                pass

    if isdir(path):
        print(f'{path} is a directory')
    elif isfile(path):
        print(f'{path} is a file')
    else:
        print(f'{path} is not a directory nor file')

paths = [mydir, myfile, nofile_or_dir]
for path in paths:
    do_some_work(path)
```

関数の先頭では、パスが存在するかどうかを確認し、存在していない場合はファイルを作成するかどうかを問い合わせて、その返答に応じてファイルを作成するかどうかを決めている。その後、渡されたパスの種類に応じて、if 文で処理を切り分けるようにしている。

Path.exists メソッドを使って パスが存在するかどうかを確認するには

pathlib モジュールの Path クラスが持つ exists メソッドを使って、パスが実際に存在するディレクトリやファイルを参照しているかを確認する方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2022 年 10 月 18 日）

```
from pathlib import Path

mydir = Path('mydir')
myfile = Path('myfile.txt')
nofile_or_dir = Path('nofile_or_dir')

# mydirディレクトリとmyfile.txtファイルを作成
mydir.mkdir(exist_ok=True)
myfile.touch()

# existsメソッドはパスがディレクトリやファイルを指していればTrueを返す
print(mydir.exists())  # True
print(myfile.exists())  # True
print(nofile_or_dir.exists())  # False

# ディレクトリが存在しなければ作成する
if not mydir.exists():  # d.mkdir(exist_ok=True)
    mydir.mkdir()

# ファイルが存在していればバックアップを取ってから書き込みを行う
if myfile.exists():
    myfile.rename(myfile.stem + '.bak')
myfile.write_text('some text')

# 指定されたファイル／ディレクトリの存在確認の後、処理を振り分ける
def do_some_work(path):
    if not path.exists():
        res = input(f'{path} not exists. create it? (y/n)')
        if res == 'y':
            path.touch()

    if path.is_dir():
        print(f'{path} is a directory')
    elif path.is_file():
        print(f'{path} is a file')
    else:
        print(f'{path} is not a directory nor file')

paths = [mydir, myfile, nofile_or_dir]
for path in paths:
    do_some_work(path)
```


→目次に戻る

Path クラスの exists インスタンスメソッド

pathlib モジュールの Path クラスには exists インスタンスメソッドが備わっている。exists メソッドは対象となる Path クラスのインスタンスがディレクトリかファイルを指していれば True を、そうでなければ False を返す（引数はない）。

以下に例を示す。

```
from pathlib import Path

mydir = Path('mydir')
myfile = Path('myfile.txt')
nofile_or_dir = Path('nofile_or_dir')

# mydirディレクトリとmyfile.txtファイルを作成
mydir.mkdir(exist_ok=True)
myfile.touch()

# existsメソッドはパスがディレクトリやファイルを指していればTrueを返す
print(mydir.exists())  # True
print(myfile.exists())  # True
print(nofile_or_dir.exists())  # False
```

この例では、最初に Path クラスをインポートした後で、mydir ディレクトリと myfile.txt ファイルを作成している（ディレクトリの作成については「ディレクトリを作成／削除するには：pathlib モジュール編」を、ファイルの作成については「ファイルを作成／削除するには」を参照のこと）。

最後の 3 行では存在しないパス「nofile_or_dir」への参照を含む、3 つの Path クラスのインスタンスについて exists メソッドを呼び出している。mydir ディレクトリと myfile.txt ファイルは存在しているので True が、nofile_or_dir についてはそのようなディレクトリもファイルも存在していないので False が返される。

このメソッドは例えば、ユーザーが指定したり、設定ファイルから読み出したりしたパスが実際に存在するかどうかを確認してから、何らかの処理を行う場合に使える。

例えば、指定されたディレクトリがなければ作成し、既に存在していれば何もしないという場合には以下のようなコードが書けるだろう（上でも見たが、これは冗長であり、Path.mkdir インスタンスメソッドなら exist_ok 引数に True を指定した方がスマートである）。

```
if not mydir.exists():  # d.mkdir(exist_ok=True)
    mydir.mkdir()
```

また、ファイルを新規に作成して、そこに何かを書き込みたいが、同名のファイルがあればそのファイルの拡張子を「.bak」に変更する必要があるといった場合には次のようなコードが書ける。

```
if myfile.exists():
    myfile.rename(myfile.stem + '.bak')
myfile.write_text('some text')
```

この例では exists メソッドでパスが存在しているかどうかを確認し、myfile が指すファイルの名前を rename メソッドで「元ファイルのベース部分 .bak」に変更してから（この場合は「myfile.bak」になる）、myfile.txt ファイルに「some text」を書き込んでいる（rename メソッドにより myfile が指すファイルの名前は変わり、書き込みを行うまでは、myfile 自体は存在しないファイル「myfile.txt」を指していることに注意）。

このように「パスの存在確認」に続けて何かの処理を行うことはよくあるだろう。そのときには「パスがディレクトリやファイルであるかどうかを調べるには：pathlib モジュール編」で紹介する is_dir ／ is_file メソッドと組み合わせることが考えられる。

以下に例を示す。do_some_work 関数は Path クラスのインスタンスを受け取り、そのパスが存在するかどうかを確認した後に、is_dir メソッドと is_file メソッドを使って処理を切り分けている。

```
def do_some_work(path):
    if path.exists():
        if path.is_dir():
            print(f'{path} is a directory')
        elif path.is_file():
            print(f'{path} is a file')
        else:
            print(f'{path} is not a directory nor file')
    else:
        print(f'{path} not exists')

paths = [mydir, myfile, nofile_or_dir]
for path in paths:
    do_some_work(path)
```

このようなネストした if 文を書いてしまいそうになるが（実際、筆者もこれでよいだろうとサンプルコードを書いていた）、is_dir ／ is_file メソッドは対象が存在しなければ False を返すので上のようにネストさせる必要はない。以下のコードで十分な場合もあるだろう。

```
def do_some_work(path):
    if path.is_dir():
        print(f'{path} is a directory')
    elif path.is_file():
        print(f'{path} is a file')
    else:
        print(f'{path} is not a directory nor file')

paths = [mydir, myfile, nofile_or_dir]
for path in paths:
    do_some_work(path)
```

exists メソッドを使った条件分岐は is_dir ／ is_file メソッドによる分岐からは独立させて「パスが存在しない場合」にディレクトリやファイルを作成してから、その後の処理を続けるような使い方をするのがよいかもしれない。

```
def do_some_work(path):
    if not path.exists():  # パスが存在しなければファイルを作成するかを尋ねる
        res = input(f'{path} not exists. create it? (y/n)')
        if res == 'y':
            path.touch()

    if path.is_dir():
        print(f'{path} is a directory')
    elif path.is_file():
        print(f'{path} is a file')
    else:
        print(f'{path} is not a directory nor file')

paths = [mydir, myfile, nofile_or_dir]
for path in paths:
    do_some_work(path)
```

os.path モジュールの isdir ／ isfile 関数を使って パスがディレクトリやファイルであるかどうかを調べるには

あるパスがディレクトリかどうかや、ファイルかどうかを調べるには os.path モジュールの isdir ／ isfile 関数が使える。その使い方や、os.scandir 関数と組み合わせて使う方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2022 年 10 月 11 日）

```
import os
from os.path import isdir, isfile, exists

# テスト用にディレクトリとファイルを作成
d = 'mydir'
if not exists(d):
    os.mkdir(d)

f = 'myfile.txt'
with open(f, 'wt'):
    pass

if isdir(d):
    print(f'{d} is a directory')
else:
    print(f'{d} is not a directory')
# 出力結果:
#mydir is a directory

if isfile(f):
    print(f'{f} is a file')
else:
    print(f'{f} is not a file')
# 出力結果:
#myfile.txt is a file

# os.scandir関数でカレントディレクトリにあるエントリを反復
for entry in os.scandir():
    if isdir(entry):
        print(f'{entry.name} is a directory')
    elif isfile(entry):
        print(f'{entry.name} is a file')
    else:
        print(f'{entry.name} is not a directory nor file')
# 出力結果:
#test.py is a file
#myfile.txt is a file
#mydir is a directory

# isdir／isfile関数を使わずにDirEntryクラスのメソッドを使える
for entry in os.scandir():
```

```
    if entry.is_dir():
        print(f'{entry.name} is a directory')
    elif entry.is_file():
        print(f'{entry.name} is a file')
    else:
        print(f'{entry.name} is not a directory nor file')
# 出力結果は上と同じ

# os.listdir関数の場合はisdir／isfile関数を使う
for path in os.listdir():
    if isdir(path):
        print(f'{path} is a directory')
    elif isfile(path):
        print(f'{path} is a file')
    else:
        print(f'{path} is not a directory nor file')
# 出力結果は上と同じ
```

os.path モジュールの isdir ／ isfile 関数

os.path モジュールには isdir 関数と isfile 関数があり、これらを使うことで、引数に指定したパスがディレクトリかどうかや、ファイルかどうかを確認できる。

ここではまず os モジュール、os.path モジュールを使ってテスト用のディレクトリとファイルを作成しておこう。

```
import os
from os.path import exists

# テスト用にディレクトリとファイルを作成
d = 'mydir'
if not exists(d):
    os.mkdir(d)

f = 'myfile.txt'
with open(f, 'wt'):
    pass
```

ここでは mydir ディレクトリと myfile.txt ファイルを作成している。

os.mkdir 関数は指定されたディレクトリが既に存在している場合、FileExistsError 例外を発生するので、ここでは os.path.exists 関数でファイルの存在確認を行うようにしている。pathlib モジュールの Path.mkdir インスタンスメソッドでは exist_ok 引数に True を指定することで例外の発生を抑止できるが、os.mkdir 関数ではそうはいかないことは覚えておこう。

isdir 関数は引数に指定されたパス（文字列、os.DirEntry クラスのインスタンス、Path クラスのインスタンス）がディレクトリを指していれば True を、そうでなければ False を返す。

パス「mydir」がディレクトリかどうかを示す例を以下に示す。

```
from os.path import isdir, isfile

# パスがディレクトリかどうかを調べる
if isdir(d):
    print(f'{d} is a directory')
else:
    print(f'{d} is not a directory')
# 出力結果:
#mydir is a directory
```

ここではディレクトリ作成時に使用した文字列を isdir 関数に渡しているので、その結果は True となり「mydir is a directory」と表示される。

isfile 関数も同様に、引数に指定されたパス（文字列、os.DirEntry クラスのインスタンス、Path クラスのインスタンス）がファイルを指していれば True を、そうでなければ False を返す。

以下に例を示す。ここでもファイルの作成に使った文字列を isfile 関数に渡しているので、その結果は True となる。

```
if isfile(f):
    print(f'{f} is a file')
else:
    print(f'{f} is not a file')
# 出力結果:
#myfile.txt is a file
```

os モジュールには scandir 関数があり、これを使うと指定したディレクトリ（デフォルトはカレントディレクトリ）にあるファイルやディレクトリを反復処理できる。このときには、isdir ／ isfile 関数などを用いて、反復されたディレクトリエントリオブジェクト（os.DirEntry クラスのインスタンス）が何を指しているかで処理を切り分けられる（なお、以下のコードの出力結果に含まれている test.py ファイルはサンプルコードを記述した Python ファイルである。以下、同様）。

以下に例を示す。

```
for entry in os.scandir():
    if isdir(entry):
        print(f'{entry.name} is a directory')
    elif isfile(entry):
        print(f'{entry.name} is a file')
    else:
        print(f'{entry.name} is not a directory nor file')
# 出力結果:
#test.py is a file
#myfile.txt is a file
#mydir is a directory
```

ただし、反復される os.DirEntry クラスのインスタンスには is_dir メソッドと is_file メソッドがあるので、実際には isdir ／ isfile 関数を使うのではなく、上記メソッドを使って以下のようにも書ける。

```
for entry in os.scandir():
    if entry.is_dir():
        print(f'{entry.name} is a directory')
    elif entry.is_file():
        print(f'{entry.name} is a file')
    else:
        print(f'{entry.name} is not a directory nor file')
```

コードを読むという観点では、こちらの方がスッと読めるはずだ（「if is file entry」よりも「if entry is file」の方が流れがよいというくらいだが）。

os モジュールにはやはり指定したディレクトリに含まれるエントリを反復する os.listdir 関数もあるが、こちらが反復するのはエントリの名前（文字列）である。そのため、こちらを使ってディレクトリの内容を反復するのであれば、os モジュールの isdir ／ isfile 関数を使う必要がある。

```
for path in os.listdir():
    if isdir(path):
        print(f'{path} is a directory')
    elif isfile(path):
        print(f'{path} is a file')
    else:
        print(f'{path} is not a directory nor file')
```

パスがディレクトリやファイルであるかどうかを調べるには：pathlib モジュール編

Pathlib モジュールの Path クラスが提供する is_dir ／ is_file メソッドを使って、特定のパスがディレクトリ／ファイルを指しているかどうかを確認する方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2022 年 10 月 04 日）

```
from pathlib import Path

# テスト用にディレクトリとファイルを作成
d = Path('mydir')
d.mkdir(exist_ok=True)

f = Path('myfile.txt')
f.touch()

if d.is_dir():
    print(f'{d} is a directory')
else:
    print(f'{d} is not a directory')
# 出力結果:
#mydir is a directory

if f.is_file():
    print(f'{f} is a file')
else:
    print(f'{f} is not a file')
# 出力結果:
#myfile.txt is a file

for entry in Path('.').iterdir():
    if entry.is_dir():
        print(f'{entry} is a directory')
    elif entry.is_file():
        print(f'{entry} is a file')
    else:
        print(f'{entry} is not a directory nor file')
# 出力結果:
#test.py is a file
#myfile.txt is a file
#mydir is a directory
```

pathlib モジュールの Path クラス

pathlib モジュールの Path クラスには is_dir インスタンスメソッドと is_file インスタンスメソッドがある。これらを使うことで Path クラスのインスタンスがディレクトリを参照しているのかどうかや、ファイルを参照しているのかどうかを調べられる。is_dir ／ is_file インスタンスメソッドはどちらも引数を持たない。

以下に簡単な例を示す。

```
from pathlib import Path

# テスト用にディレクトリとファイルを作成
d = Path('mydir')
d.mkdir(exist_ok=True)

f = Path('myfile.txt')
f.touch()

# パスがディレクトリかどうかを調べる
if d.is_dir():
    print(f'{d} is a directory')
else:
    print(f'{d} is not a directory')
# 出力結果:
# mydir is a directory
```

最初に行っているのは、Path クラスの mkdir メソッドと touch メソッドを使ったディレクトリ／ファイルの作成だ。これらについては「ディレクトリを作成／削除するには：pathlib モジュール編」と「ファイルを作成／削除するには」を参照のこと。

is_dir メソッドはパスがディレクトリを参照しているときには True を、そうでなければ False を返す。上のコードでは、ディレクトリの作成に使用した Path オブジェクトに対して is_dir メソッドを呼び出しているので True が返される。

is_file メソッドも同様だ。パスがファイルを参照しているときには True を、そうでなければ False を返す。以下のコード例でもファイル作成に使用した Path オブジェクトに対して is_file メソッドを呼び出しているので、上と同様に True が返される。

```
if f.is_file():
    print(f'{f} is a file')
else:
    print(f'{f} is not a file')
# 出力結果:
#myfile.txt is a file
```

Path クラスには指定したディレクトリに含まれるディレクトリやファイルを反復するための iterdir インスタンスメソッドもある。これを使って、特定のディレクトリに含まれているディレクトリやファイルをその種類に応じて処理するにも is_dir ／ is_file メソッドを使える。

以下に例を示す。

```
for entry in Path('.').iterdir():
    if entry.is_dir():
        print(f'{entry} is a directory')
    elif entry.is_file():
        print(f'{entry} is a file')
    else:
        print(f'{entry} is not a directory nor file')
# 出力結果:
#test.py is a file
#myfile.txt is a file
#mydir is a directory
```

この例では、カレントディレクトリの内容を反復するために「Path('.').iterdir()」として for ループでカレントディレクトリに含まれているディレクトリ／ファイルをループ変数の entry に受け取っている（test.py ファイルはサンプルコードが記述されている Python スクリプト）。

for ループのボディーでは、is_dir ／ is_file メソッドを使って、反復されたパスがディレクトリかファイルかで処理を切り分けている。

os.path.join 関数を使ってパスを結合するには

os.path.join 関数を使って相対パスや絶対パスを結合する方法や、最後にパス区切り文字を付加する方法、Windows のドライブ文字の扱いなどについて紹介する。

かわさきしんじ，Deep Insider 編集部（2022 年 11 月 22 日）

```
from os.path import join

# UNIX系統のOSの場合
# 相対パスの結合
result = join('foo', 'bar', 'baz.txt')
print(result)  # foo/bar/baz.txt

# 絶対パスの結合
result = join('/foo', 'bar', 'baz.txt')
print(result)  # /foo/bar/baz.txt

# 引数の途中に絶対パスが含まれている場合
result = join('foo', '/bar', 'baz.txt')
print(result)  # /bar/baz.txt

# 引数の最後を空文字列にすると区切り文字でパスが終わる
result = join('foo', 'bar', 'baz', '')
print(result)  # foo/bar/baz/

data = 'data'
start_year = 2020
end_year = 2022
for year in range(start_year, end_year+1):
    for month in range(1, 13):
        path = join(data, f'{year}', f'{month:02}', '')
        # data/2020/01/、data/2020/02/などのパスを使って何らかの処理を行う
        print(path)

# Windowsの場合
# 相対パスの結合
result = join('foo', 'bar', 'baz.txt')
print(result)  # foo¥bar¥baz.txt

# 絶対パスの結合
result = join('¥¥foo', 'bar', 'baz.txt')
print(result)  # ¥foo¥bar¥baz.txt

# 引数の途中に絶対パスが含まれている場合
result = join('foo', '¥¥bar', 'baz.txt')
print(result)  # ¥bar¥baz.txt

# 絶対パスの結合（ドライブ文字を含む）
result = join('c:', '¥¥foo', 'bar', 'baz.txt')
print(result)  # c:\foo\bar\baz.txt
```

```
# ドライブ文字とパスだけを含む場合
result = join('d:', 'foo')
print(result)  # d:foo

# 引数の最後を空文字列にすると区切り文字でパスが終わる
result = join('foo', 'bar', 'baz', '')
print(result)  # foo¥bar¥baz¥

data = 'data'
start_year = 2020
end_year = 2022
for year in range(start_year, end_year+1):
    for month in range(1, 13):
        path = join(data, f'{year}', f'{month:02}', '')
        # data¥2020¥01¥、data¥2020¥02¥などのパスを使って何らかの処理を行う
        print(path)
```

パスの結合

os.path モジュールの join 関数を使うと、パスを構成する各要素を結合できる（このとき、使用している OS で決められているパス区切り文字がパスとパスの間に挟み込まれる）。join 関数には 1 つ以上の文字列（または Path オブジェクトなど）を指定する。

以下に macOS で実行した例を示す。

```
result = join('foo', 'bar', 'baz.txt')
print(result)  # foo/bar/baz.txt
```

この例では、join 関数に 'foo'、'bar'、'baz.txt' という 3 つの文字列を渡している。その結果は、これらをパス区切り文字であるスラッシュ「/」を使って結合した「foo/bar/baz.txt」となる。

Windows では次のようになる。

```
result = join('foo', 'bar', 'baz.txt')
print(result)  # foo¥bar¥baz.txt
```

Windows ではパス区切り文字がバックスラッシュ「¥」になっているが、それ以外は同様だ。

上の例は相対パスを結合したものだが、絶対パスであっても同様に結合できる。以下はその例だ（macOS）。

```
result = join('/foo', 'bar', 'baz.txt')
print(result)  # /foo/bar/baz.txt
```

パスを結合して、絶対パスにするには最初の引数をパス区切り文字で始めるのが簡単だ。だが、引数の中に絶対パスを含めることも可能だ。この場合、その引数よりも前に置いたパス構成要素は無視され、引数に並べた絶対パス以降の要素が結合される。

```
result = join('foo', '/bar', 'baz.txt')
print(result)  # /bar/baz.txt
```

この例では第 2 引数がスラッシュ「/」で始まっている（つまり、これは絶対パス）。そのため、第 1 引数は無視されて、第 2 引数と第 3 引数を結合した結果が返されている。

このことは Windows でも同様だ。以下に例を示す。

```
result = join('¥¥foo', 'bar', 'baz.txt')
print(result)  # ¥foo¥bar¥baz.txt
```

注意してほしいのは、Windows にはドライブ文字があるので、上の例の結果「¥foo¥bar¥baz.txt」はカレントドライブの絶対パスになるということだ。

絶対パスを引数の途中に置いた場合の例は次の通り。

```
result = join('foo', '¥¥bar', 'baz.txt')
print(result)  # ¥bar¥baz.txt
```

既に述べたように Windows のパスにはドライブ文字という要素がある。ドライブ文字を含んだ絶対パスを結合する例を以下に示す。

```
result = join('c:', '¥¥foo', 'bar', 'baz.txt')
print(result)  # c:¥foo¥bar¥baz.txt
```

この例ではドライブ文字を第 1 引数に、ルートディレクトリを第 2 引数に指定している。ちなみに、引数の途中でドライブ文字を指定した場合は、絶対パスを途中で指定したときと同様に、それよりも前の引数は無視される。

```
result = join('¥¥foo', 'c:', 'bar')
print(result)  # c:bar
```

この例では第 1 引数にルートディレクトリを意味する「'¥¥foo'」を指定しているが、その直後にドライブ文字を指定している。そのため、「'¥¥foo'」は無視されて「c:bar」が返されている。この「c:bar」が意味するのは「C ドライブのカレントディレクトリの下にある bar というパス」である。「C ドライブのルートディレクトリの直下にある bar というパス」ではないことに注意しよう。

よって、以下は「D ドライブのカレントディレクトリにある foo」というパスを示す。

```
# ドライブ文字とパスだけを含む場合
result = join('d:', 'foo')
print(result)  # d:foo
```

パスの最後に区切り文字を含めたいときには引数の最後に空文字列を置けばよい。

```
# UNIX系統のOSの例
result = join('foo', 'bar', 'baz', '')
print(result)  # foo/bar/baz/

# Windowsの例
result = join('foo', 'bar', 'baz', '')
print(result)  # foo¥bar¥baz¥
```

UNIX 系統の OS と Windows とで区切り文字が異なるが、最後にパス区切り文字が付加されていることに注目されたい。

join 関数を使うと、例えば data ディレクトリの下に年ごと、月ごとにディレクトリが作成されていて、そこに何らかのファイルがあり、それらを定型処理するといった場合に以下のようなコードが書けるだろう。

```
data = 'data'
start_year = 2020
end_year = 2022
for year in range(start_year, end_year+1):
    for month in range(1, 13):
        path = join(data, f'{year}', f'{month:02}', '')
        # data/2020/01/、data/2020/02/などのパスを使って何らかの処理を行う
        print(path)
```

これを実行すると以下のように機械的にパスを自動生成できる（Windows ではパス区切り文字が異なるが同様な結果になる）。

```
>>> data = 'data'
>>> start_year = 2020
>>> end_year = 2022
>>> for year in range(start_year, end_year+1):
...     for month in range(1, 13):
...         path = join(data, f'{year}', f'{month:02}', '')
...         # data/2020/01/、data/2020/02/などのパスを使って何らかの処理を行う
...         print(path)
...
data/2020/01/
data/2020/02/
data/2020/03/
    # ……中略……
data/2022/10/
data/2022/11/
data/2022/12/
```

pathlib.Path.joinpath メソッドを使って パスを結合するには

pathlib モジュールの Path クラスが提供する joinpath メソッドを使って相対パスや絶対パスを結合する方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2022 年 11 月 29 日）

```
from pathlib import Path

# UNIX系統のOSの場合
# 相対パスの結合
path = Path('foo')
result = path.joinpath('bar', Path('baz'))
print(result)  # foo/bar/baz

# 絶対パスの結合
path = Path('/foo')
result = path.joinpath('bar', 'baz.txt')
print(result)  # /foo/bar/baz.txt

# 引数の途中に絶対パスが含まれている場合
path = Path('foo')
result = path.joinpath('bar', '/baz', 'qux')
print(result)  # /baz/qux

# os.path.join関数とは異なり空文字列を最後においてもパス区切り文字は付加されない
from os.path import join
path = Path('foo')
result = join(path, '')
print(result)  # foo/

result = path.joinpath(Path(''))
print(result)  # foo

data = Path('data')
start_year = 2020
end_year = 2022
for year in range(start_year, end_year+1):
    for month in range(1, 13):
        path = data.joinpath(f'{year}', f'{month:02}')
        # data/2020/01、data/2020/02などのパスを使って何らかの処理を行う
        somefile = path.joinpath('somefile.txt')
        print(somefile)

# Windowsの場合
# 相対パスの結合
path = Path('foo')
result = path.joinpath('bar', 'baz.txt')
print(result)  # foo¥bar¥baz.txt
```


→目次に戻る

```
# 絶対パスの結合
path = Path('¥¥foo')
result = path.joinpath('bar', 'baz.txt')
print(result)  # ¥foo¥bar¥baz.txt

# 引数の途中に絶対パスが含まれている場合
path = Path('foo')
result = path.joinpath('bar', '¥¥baz', 'qux')
print(result)  # ¥baz¥qux

# 絶対パスの結合(ドライブ文字を含む)
path = Path('c:')
result = path.joinpath('¥¥foo', 'bar', 'baz.txt')
print(result)  # c:¥foo¥bar¥baz.txt

path = Path('c:')
result = path.joinpath('foo', '¥¥bar', 'baz.txt')
print(result)  # c:¥bar¥baz.txt

# ドライブ文字とパスだけを含む場合
path = Path('d:')
result = path.joinpath('foo')
print(result)  # d:foo

# os.path.join関数とは異なり空文字列を最後においてもパス区切り文字は付加されない
from os.path import join
path = Path('foo')
result = join(path, '')
print(result)  # foo¥

result = path.joinpath(Path(''))
print(result)  # foo

data = Path('data')
start_year = 2020
end_year = 2022
for year in range(start_year, end_year+1):
    for month in range(1, 13):
        path = data.joinpath(f'{year}', f'{month:02}')
        # data¥2020¥01、data¥2020¥02などのパスを使って何らかの処理を行う
        somefile = path.joinpath('somefile.txt')
        print(somefile)
```


→目次に戻る

pathlib.Path クラスの joinpath メソッドを使ったパスの結合

pathlib モジュールの Path クラスのインスタンスには joinpath メソッドがある（実際には Path クラスの基底クラスである PurePath クラスがこのメソッドを提供している）。このメソッドを使うと、Path クラスのインスタンスと引数に指定したパス構成要素を基に、新たなパスを表す Path オブジェクトを生成できる。

このメソッドの動作は次のようになっている。

- joinpath メソッドの引数に絶対パスが含まれていない場合：Path オブジェクトの値をパスのトップレベルとして、引数に指定したパス構成要素をつないだものを返送する
- joinpath メソッドの引数に絶対パスが含まれている場合：最後に指定された絶対パスとそれ以降に指定されているパス構成要素をつないだものを返送する

簡単な例を以下に示す（macOS で実行した例）。

```
from pathlib import Path

path = Path('foo')
result = path.joinpath('bar', Path('baz'))
print(result)  # foo/bar/baz
```

この例では joinpath メソッドの呼び出しに使用している Path オブジェクトは「foo」という相対パスを表している。また、joinpath メソッドの引数には 'bar' と Path('baz') の 2 つの相対パスを表す値が渡されている（このように joinpath メソッドには文字列や Path クラスのインスタンスを渡せることにも注目）。そのため、joinpath はこれらを結合した結果である「foo/bar/baz」を表す Path オブジェクトを返送する。

これは Windows でも同様だ。Windows 上で実行した例を以下に示す。パス区切り文字が異なることを除けば同様な結果が得られている。

```
from pathlib import Path

path = Path('foo')
result = path.joinpath('bar', 'baz.txt')
print(result)  # foo¥bar¥baz.txt
```

絶対パスを結合するには次の 2 つの方法がある。

- 絶対パスを参照する Path オブジェクトで joinpath メソッドを呼び出して、結合したいパスを指定する
- joinpath メソッドを呼び出して、その引数に絶対パスを含むパス構成要素を指定する

前者の例を以下に示す（macOS）。

```
path = Path('/foo')
result = path.joinpath('bar', 'baz.txt')
print(result)  # /foo/bar/baz.txt
```

この例では変数 path は「/foo」という絶対パスを参照している。そして、このオブジェクトで joinpath メソッドを呼び出しているので、引数に指定した 2 つのパス構成要素「bar」と「baz.txt」をパス区切り文字でつないだものが戻り値となっている。

以下は joinpath メソッドの引数に絶対パスを含めた例だ（macOS）。

```
path = Path('foo')
result = path.joinpath('bar', '/baz', 'qux')
print(result)  # /baz/qux
```

この例では、joinpath メソッドの呼び出しに使用している Path オブジェクトは相対パスを参照しているが、引数には「/baz」という絶対パスが含まれている。このような場合、呼び出しに使用したオブジェクトの値や絶対パスよりも前に指定されているパス構成要素は無視されて、（最後に指定された）絶対パスとそれ以降の構成要素をつないだものが戻り値となる。そのため、戻り値の値は絶対パスである「/baz」とその後にある「qux」をつないだ「/baz/qux」となっている。

Windows でもパス区切りがバックスラッシュ「¥」になることとそのエスケープが必要になることを除けば、基本的な動作は同様だ（バックスラッシュではなく円マークを使って UNIX 系統の OS と同様な書き方をしてもよい）。

```
path = Path('¥¥foo')
result = path.joinpath('bar', 'baz.txt')
print(result)  # ¥foo¥bar¥baz.txt
```

この例は、バックスラッシュとそのエスケープを除けば、先ほどの macOS の例と同様なコードであり、結果も同様だ。以下は joinpath メソッドの引数に絶対パスを含んだ例だが、これも同様だ。

```
path = Path('foo')
result = path.joinpath('bar', '¥¥baz', 'qux')
print(result)  # ¥baz¥qux
```

ただし、Windows にはドライブ文字という概念があるので、そこには注意が必要だ。以下はドライブ文字、ルートディレクトリ、サブディレクトリ、ファイルで構成されるパスを結合する例だ。

```
path = Path('c:')
result = path.joinpath('¥¥foo', 'bar', 'baz.txt')
print(result)  # c:¥foo¥bar¥baz.txt
```

ドライブ文字と絶対パスと他の構成要素を結合する例を以下に示す。

```
path = Path('c:')
result = path.joinpath('foo', '¥¥bar', 'baz.txt')
print(result)  # c:¥bar¥baz.txt
```

この場合、ドライブ文字である「c:」は有効だが、絶対パスである「¥¥bar」よりも前にある「foo」は無視されて「c:¥bar¥baz.txt」というパスが得られている。

以下はドライブ文字と相対パスを結合する例だ。

```
path = Path('d:')
result = path.joinpath('foo')
print(result)  # d:foo
```

この場合は、パスはあくまでも D ドライブのカレントディレクトリにある foo というパスを表すことになる。

joinpath メソッドを使うと、各種の要素で構成されるパスを機械的に扱って、処理対象のパスを自動的に生成できる。例えば、data ディレクトリの下に年、月ごとに作成されたディレクトリがあり、さらにその下にあるファイルを対象に何かの処理を行うとすると、以下のようなコードが書けるだろう。

```
data = Path('data')
start_year = 2020
end_year = 2022
for year in range(start_year, end_year+1):
    for month in range(1, 13):
        path = data.joinpath(f'{year}', f'{month:02}')
        # data/2020/01、data/2020/02などのパスを使って何らかの処理を行う
        somefile = path.joinpath('somefile.txt')
        print(somefile)
```

これを実行すると、以下のようなパスを自動的に生成できる（Windows ではパス区切り文字が異なる以外は同様な結果となる）。

```
>>> data = Path('data')
>>> start_year = 2020
>>> end_year = 2022
>>> for year in range(start_year, end_year+1):
...     for month in range(1, 13):
...         path = data.joinpath(f'{year}', f'{month:02}')
...         # data/2020/01、data/2020/02などのパスを使って何らかの処理を行う
...         somefile = path.joinpath('somefile.txt')
...         print(somefile)
...
data/2020/01/somefile.txt
data/2020/02/somefile.txt
    # ……省略……
data/2022/11/somefile.txt
data/2022/12/somefile.txt
```


→目次に戻る

split 関数でファイルパスを分割するには

os.path モジュールの split 関数はファイルパスを末尾の要素とそれ以外に分割する。その使い方と注意点を紹介する。

かわさきしんじ，Deep Insider 編集部（2024 年 03 月 19 日）

```
from os.path import split

p = '/tmp/foo/bar.txt'
result = split(p)
print(result)  # ('/tmp/foo', 'bar.txt')

p = 'C:¥¥tmp¥¥foo¥¥bar.txt'
result = split(p)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥bar.txt')
# Windows:('C:¥¥tmp¥¥foo', 'bar.txt')

# Windows上で動作するPythonと同様なパス分割を行う
import ntpath

p = 'C:¥¥tmp¥¥foo¥¥bar.txt'
result = ntpath.split(p)
print(result)  # ('C:¥¥tmp¥¥foo', 'bar.txt')

# パスがセパレーターで終わっている場合
p = '/tmp/foo/'
result = split(p)
print(result)  # ('/tmp/foo', '')

p = 'C:¥¥tmp¥¥foo¥¥'
result = split(p)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥')
# Windows:('C:¥¥tmp¥¥foo', '')

result = ntpath.split(p)
print(result)  # ('C:¥¥tmp¥¥foo', '')

p = '/tmp/foo'  # fooはディレクトリかファイルか分からない
result = split(p)
print(result)  # ('/tmp', 'foo')

# os.path.split関数とos.path.isdir関数、os.path.isfile関数を組み合わせる
```

```
from pathlib import Path

Path('sample').mkdir(exist_ok=True)
Path('sample/foo').mkdir(exist_ok=True)
Path('sample/bar').mkdir(exist_ok=True)
Path('sample/foo.txt').touch(exist_ok=True)
Path('sample/bar.txt').touch(exist_ok=True)

import os

d = 'sample'
for item in os.listdir(d):
    p = d + '/' + item
    head, tail = split(p)
    if os.path.isdir(p):
        print(f'{tail} is a directory')
    elif os.path.isfile(p):
        print(f'{tail} is a file')

# パスにセパレーターが含まれていない場合
p = 'foo.txt'
result = split(p)
print(result)  # ('', 'foo.txt')

# パスがファイルシステムのルートを参照する場合
p = '/'
result = split(p)
print(result)

# Pathクラスのインスタンスをos.path.split関数に渡す
from pathlib import Path

p = Path('/tmp/foo/bar.txt')
result = split(p)
print(result)  # ('/tmp/foo', 'bar.txt')

p = Path('C:¥¥tmp¥¥foo¥¥bar.txt')
result = split(p)
print(result)  # UNIXとWindowsで結果が異なる

result = ntpath.split(p)
print(result)  # ('C:¥¥tmp¥¥foo', 'bar.txt')
```

os.path.split 関数

os.path モジュールにはパスを特定の条件で分割する関数が幾つかある。

- **os.path.split 関数**：パスを「(パスの末尾より前 , パスの末尾)」という 2 要素のタプルに分割する
- os.path.splitdrive 関数：パスを「(ドライブ , それ以外)」という 2 要素のタプルに分割する
- os.path.splitext 関数：パスを「(拡張子以外の部分 , 拡張子)」という 2 要素のタプルに分割する
- os.path.splitroot 関数:パスを「(ドライブ , パスの末尾より前 , パスの末尾)」という 3 要素のタプルに分割する（Python 3.12 以降）

このうち、以下では os.path.split 関数を使って、パスを末尾とそれ以外の部分に分割する方法を紹介する。

以下に os.path.split 関数の構文を示す。

```
os.path.splitext(path)
```

os.path.split 関数はパラメーターを 1 つ持ち、これにパスを渡すと「(パスの末尾以外の要素 , 末尾要素)」という 2 つの要素からなるタプルが返される。ここでいう末尾以外の要素と末尾要素とは次のようなものだ。

- 末尾以外の要素：ディレクトリを区切る最後のセパレーターより前の要素
- 末尾要素：ディレクトリを区切る最後のセパレーターより後ろの要素

セパレーターは UNIX 系統の OS であればスラッシュ「/」であり、Windows では一般的にはバックスラッシュ「¥」となる。

基本的な例を以下に示す。

```
from os.path import split

p = '/tmp/foo/bar.txt'
result = split(p)
print(result)  # ('/tmp/foo', 'bar.txt')
```

ここでは os.path.split 関数に '/tmp/foo/bar.txt' という文字列を渡している。末尾以外の要素は最後のセパレーターよりも前なので「/tmp/foo」となり、末尾要素は最後のセパレーターより後ろなので「bar.txt」となっているのが分かる。ただし、Windows 形式のパスについては、実行している OS によって結果が異なるので注意が必要だ。以下に例を示す。

```
p = 'C:¥¥tmp¥¥foo¥¥bar.txt'
result = split(p)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥bar.txt')
# Windows:('C:¥¥tmp¥¥foo', 'bar.txt')
```

この例では 'C:¥¥tmp¥¥foo¥¥bar.txt' という文字列を os.path.split 関数に渡しているが、筆者が macOS と Windows で試したところでは、その戻り値は macOS では「('', 'C:¥¥tmp¥¥foo¥¥bar.txt')」に、Windows では「('C:¥¥tmp¥¥foo', 'bar.txt')」となった。

これは UNIX 系統の OS では os.path モジュールの実体は posixpath モジュールであり、Windows では os.path モジュールの実体が ntpath モジュールとなっていて、それぞれの OS（のファイルシステム）で都合が良くなるように処理が行われているからだ（os.path モジュールは Python プログラムを実行している OS によって、ローカルのファイルパスを処理しやすいように切り替えられているということだ）。

UNIX 系統の OS で Windows 上で動作している Python と同様にパスを分割したいのであれば、ntpath モジュールをインポートして、その split 関数を呼び出す。以下はその例だ。

```
import ntpath

p = 'C:¥¥tmp¥¥foo¥¥bar.txt'
result = ntpath.split(p)
print(result)  # ('C:¥¥tmp¥¥foo', 'bar.txt')
```

逆に、Windows 上で UNIX 系統の OS で動作している Python と同様にパスを分割したいのであれば、posixpath モジュールをインポートして、その split 関数を呼び出せばよい（例は省略）。

特殊なケース

パスがディレクトリを区切るセパレーターで終わっている場合、戻り値であるタプルの第 1 要素（2 つ目の要素）は空文字列になる。UNIX 形式のパス、Windows 形式のパス、ntpath モジュールの split 関数のそれぞれで試した例を以下に示す。

```
p = '/tmp/foo/'
result = split(p)
print(result)  # ('/tmp/foo', '')

p = 'C:¥¥tmp¥¥foo¥¥'
result = split(p)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥')
# Windows:('C:¥¥tmp¥¥foo', '')

result = ntpath.split(p)
print(result)  # ('C:¥¥tmp¥¥foo', '')
```

パスがセパレーターで終わっていなければ、パスの末尾がディレクトリかファイルかに関係なく、末尾要素とそれ以外に分割される。

```
p = '/tmp/foo'  # fooはディレクトリかファイルか分からない
result = split(p)
print(result)  # ('/tmp', 'foo')
```

パスが存在するかどうかは os.path.exists 関数で、ディレクトリかどうかは os.path.isdir 関数で、ファイルかどうかは os.path.isfile 関数で調べられるので、処理を切り分けるのであれば、そうした関数の使用を考えよう（あるいは os.scandir 関数で反復される DirEntry クラスの is_dir ／ is_file メソッドや、pathlib.Path クラスの is_dir ／ is_file メソッドでもよいだろう）。

以下に例を示す（わざとらしい例だがご容赦願いたい）。

```
from pathlib import Path

Path('sample').mkdir(exist_ok=True)
Path('sample/foo').mkdir(exist_ok=True)
Path('sample/bar').mkdir(exist_ok=True)
Path('sample/foo.txt').touch(exist_ok=True)
Path('sample/bar.txt').touch(exist_ok=True)

import os

d = 'sample'
for item in os.listdir(d):
    p = d + '/' + item
    head, tail = split(p)
    if os.path.isdir(p):
        print(f'{tail} is a directory')
    elif os.path.isfile(p):
        print(f'{tail} is a file')
```

この例ではカレントディレクトリに sample ディレクトリを作成して、その下にさらにディレクトリとファイルを幾つか作成している。その後、sample ディレクトリを起点にファイル／ディレクトリを反復し、それがディレクトリかファイルかで処理を切り分けている。上のコードで os.listdir 関数は sample ディレクトリからの相対パスを反復するので、os.path.isdir 関数や os.path.isfile 関数でパスがディレクトリかファイルかを判定できるようにパスを加工している。加工後のパスは、os.path.split 関数で分割して、sample ディレクトリにあるファイルやディレクトリの名前を取り出している（ここがわざとらしいところで、実際には「f'{item} is ……'」でよい）。

また、パスにセパレーターが含まれていない場合（つまり、カレントディレクトリにあるファイルやディレクトリをパスが参照している場合）には、戻り値であるタプルの第 0 要素は常に空文字列となる。

```
# パスにセパレーターが含まれていない場合
p = 'foo.txt'
result = split(p)
print(result)  # ('', 'foo.txt')
```

os.path.split 関数が返すタプルの第 0 要素は最後にセパレーターが付かないが、パスとしてルートを渡したときだけはセパレーターで終わる。

```
p = '/'
result = split(p)
print(result)
```

os.path.split 関数と path-like object

os.path.split 関数は文字列以外にも path-like object を受け取れる。そうしたオブジェクトの典型例が pathlib モジュールの Path クラスだ。以下に os.path.split 関数に Path クラスのインスタンスを渡す例を示す。

```
from pathlib import Path

p = Path('/tmp/foo/bar.txt')
result = split(p)
print(result)  # ('/tmp/foo', 'bar.txt')
```

ここでは UNIX 形式のファイルパスを渡しているので、その結果は UNIX 形式のファイルパスを文字列として渡したときと変わらない。なお、戻り値であるタプルの要素は Path クラスのインスタンスではなく、文字列であることには注意しよう。

Windows 形式のファイルパスでもこれは同様だ（つまり、Windows と UNIX 系統の OS で分割結果が異なる）。

```
p = Path('C:¥¥tmp¥¥foo¥¥bar.txt')
result = split(p)
print(result)  # UNIXとWindowsで結果が異なる

result = ntpath.split(p)
print(result)  # ('C:¥¥tmp¥¥foo', 'bar.txt')
```

なお、Windows 環境以外で Windows 形式のパスから Path クラスのインスタンスを生成しようすると、実際には pathlib.PosixPath クラスのインスタンスが生成されることには注意しよう（これに対して、Windows 環境で Path クラスのインスタンスを生成しようとすると、実際には pathlib.WindowsPath クラスのインスタンスが生成される）。pathlib モジュールには Windows 形式のファイルパスを表すクラスとして PureWindowsPath クラスがある。これは Windows のファイルシステムとは関係なく、Windows 形式のパスの操作を確認するために便利に使える。

```
from pathlib import PureWindowsPath

p = PureWindowsPath('C:¥¥tmp¥¥foo¥¥bar.txt')
result = split(p)
print(result)  # ('', 'C:¥¥tmp¥¥foo¥¥bar.txt')

result = ntpath.split(p)
print(result)  # ('C:¥¥tmp¥¥foo', 'bar.txt')
```

上の例を見ると分かるが、PureWindowsPath クラスは Windows 形式のパスを表すものの、Windows 環境でのパス分割と同じ結果を得たいのであれば、加えて ntpath モジュールを使用する必要がある。

splitext 関数でファイルパスから拡張子を取得するには

os.path.splitext 関数は渡されたパスを拡張子とそれ以外の部分に分割する。その基本的な使い方と注意点、拡張子ごとに処理を切り分けるサンプルコードを紹介する。

かわさきしんじ，Deep Insider 編集部（2024 年 03 月 12 日）

```
from os.path import splitext

# os.path.splitext関数はパスを拡張子とそれ以外に分割する
w_file = 'C:¥¥tmp¥¥pytips¥¥foo.txt'
u_file = '/tmp/pytips/foo.txt'

result = splitext(w_file)
print(result)  # ('C:¥¥tmp¥¥pytips¥¥foo', '.txt')
result = splitext(u_file)
print(result)  # ('/tmp/pytips/foo', '.txt')

w_dir = 'C:¥¥tmp¥¥pytips¥¥'
u_dir = '/tmp/pytips/'

result = splitext(w_dir)
print(result)  # ('C:¥¥tmp¥¥pytips¥¥', '')
result = splitext(u_dir)
print(result)  # ('/tmp/pytips/', '')

# Pathクラスのインスタンスでも0K
from pathlib import Path

u_file = Path(u_file)
print(u_file)  # /tmp/pytips/foo.txt
result = splitext(u_file)
print(result)  # ('/tmp/pytips/foo', '.txt')

# os.path.splitext関数は最後の拡張子以降を拡張子として扱う
f = '/tmp/foo/bar.tar.gz'
result = splitext(f)
print(result)  # ('/tmp/foo/bar.tar', '.gz')

f = '/tmp/foo/readme'
result = splitext(f)  # 拡張子がなければタプルの拡張子部分は空文字列
print(result)  # ('/tmp/foo/readme', '')

# 拡張子に応じて何らかの処理を行う
# 準備
from pathlib import Path

d = Path('pytips/test')
d.mkdir(parents=True, exist_ok=True)
```

```
f0 = Path(d, 'foo.txt')  # 'pytips/test/foo.txt'
f1 = d / Path('bar.csv')  # 'pytips/test/bar.csv'
f2 = d / 'baz.tsv'  # pytips/test/baz.tsv'

print(d)  # pytips/test
print(f0)  # pytips/test/foo.txt
print(f1)  # pytips/test/bar.csv
print(f2)  # pytips/test/baz.tsv

f0.touch()
f1.touch()
f2.touch()

# Path.iterdirメソッドでディレクトリを走査する
for item in d.iterdir():
    root, ext = splitext(item)
    match(ext):
        case '.txt':
            print('processing a text file:', item)
        case '.csv' | '.tsv':
            print('processing a csv/tsv file:', item)
        case _:
            print('do nothing:', item)

# os.scandir関数でディレクトリを走査する
import os
for item in os.scandir(d):
    root, ext = splitext(item)  # DirEntryクラスのインスタンスでもOK
    if ext == '.txt':
        print('processing a text file:', item.name)
    elif ext == '.csv' or ext == '.tsv':
        print('processing a csv/tsv file:', item.name)
    else:
        print('do nothing:', item.name)
```

os.path.splitext 関数

os.path モジュールにはパスを特定の条件で分割する関数が幾つかある。

- os.path.split 関数：パスを「(パスの末尾より前 , パスの末尾)」という 2 要素のタプルに分割する
- os.path.splitdrive 関数：パスを「(ドライブ , それ以外)」という 2 要素のタプルに分割する
- **os.path.splitext 関数**：パスを「(拡張子以外の部分 , 拡張子)」という 2 要素のタプルに分割する
- os.path.splitroot 関数：パスを「(ドライブ , パスの末尾より前 , パスの末尾)」という 3 要素のタプルに分割する（Python 3.12 以降）

このうち、以下では os.path.splitext 関数を使って、パスを拡張子とそれ以外の部分に分割する方法を紹介する。

以下に os.path.splitext 関数の構文を示す。

```
os.path.splitext(path)
```

os.path.splitext 関数はパラメーターを 1 つ持ち、これにパスを渡すと「(拡張子以外の部分 , 拡張子)」という 2 つの要素からなるタプルが返される。

基本的な例を以下に示す。

```
from os.path import splitext

# os.path.splitext関数はパスを拡張子とそれ以外に分割する
w_file = 'C:¥¥tmp¥¥pytips¥¥foo.txt'
u_file = '/tmp/pytips/foo.txt'

result = splitext(w_file)
print(result)  # ('C:¥¥tmp¥¥pytips¥¥foo', '.txt')
result = splitext(u_file)
print(result)  # ('/tmp/pytips/foo', '.txt')
```

変数 w_file には Windows 形式のパス（テキストファイルを表している）が、変数 u_file には Unix 形式のパス（こちらもテキストファイルを表している）が代入されている。これらを os.path.splitext 関数に渡すと結果は「('C:¥¥tmp¥¥pytips¥¥foo', '.txt')」「('/tmp/pytips/foo', '.txt')」と拡張子以外の部分と拡張子を含んだタプルが得られる。

注意したいのは、拡張子の部分がピリオドで始まっている点だ。

次のように拡張子を含まないパス（多くの場合はディレクトリやフォルダーを意味しているだろう）を渡すと、タプルの拡張子部分は空文字列となる。

```
w_dir = 'C:¥¥tmp¥¥pytips¥¥'
u_dir = '/tmp/pytips/'

result = splitext(w_dir)
print(result)  # ('C:¥¥tmp¥¥pytips¥¥', '')
result = splitext(u_dir)
print(result)  # ('/tmp/pytips/', '')
```

ここまでパスとして文字列を渡していたが、os.path.splitext 関数にはいわゆる「path-like object」を渡してもよい。そうしたオブジェクトとして典型的なのが pathlib モジュールの Path クラスのインスタンスだ。

```
from pathlib import Path

u_file = Path(u_file)
print(u_file)  # /tmp/pytips/foo.txt
result = splitext(u_file)
print(result)  # ('/tmp/pytips/foo', '.txt')
```

この例では、文字列として表現されていたパスを Path クラスのインスタンスに変換し、それを os.path.splitext 関数に渡している。

パスの最後の構成要素が複数のピリオドを含んでいる場合、os.path.splitext 関数は最後のピリオド以降を拡張子として扱う。以下に例を示す。

```
f = '/tmp/foo/bar.tar.gz'
result = splitext(f)
print(result)  # ('/tmp/foo/bar.tar', '.gz')
```

ここではパスは /tmp/foo ディレクトリにある bar.tar.gz ファイルを指しているが、これを os.path.splitext 関数に渡すと最後のピリオド以降の「.gz」が拡張子として扱われる。

逆に拡張子がない場合は（先ほども見たが）、戻り値となるタプルの拡張子部分は空文字列になる。以下の例では readme ファイルを指しているパスを os.path.splitext 関数に渡しているが、戻り値であるタプルの拡張子部分は空文字列になっている。

```
f = '/tmp/foo/readme'
result = splitext(f)  # 拡張子がなければタプルの拡張子部分は空文字列
print(result)  # ('/tmp/foo/readme', '')
```

拡張子に応じて何らかの処理を行う

最後に、os.path.splitext 関数で拡張子を取り出して、その種類に応じた処理を行う例を示す。その準備として、ここではカレントディレクトリに pytips ディレクトリを作成し、その下にさらに test ディレクトリを作成して、そこに 3 つのファイルを配置している。

```
from pathlib import Path

d = Path('pytips/test')
d.mkdir(parents=True, exist_ok=True)

f0 = Path(d, 'foo.txt')  # 'pytips/test/foo.txt'
f1 = d / Path('bar.csv')  # 'pytips/test/bar.csv'
f2 = d / 'baz.tsv'  # pytips/test/baz.tsv'

print(d)  # pytips/test
print(f0)  # pytips/test/foo.txt
print(f1)  # pytips/test/bar.csv
print(f2)  # pytips/test/baz.tsv

f0.touch()
f1.touch()
f2.touch()
```

最初に Path.iterdir メソッドで pytips/test ディレクトリを走査して、そこにある 3 つのファイルの拡張子を得て、match 文で処理を分岐させる例だ。

```
for item in d.iterdir():
    root, ext = splitext(item)
    match(ext):
        case '.txt':
            print('processing a text file:', item)
        case '.csv' | '.tsv':
            print('processing a csv/tsv file:', item)
        case _:
            print('do nothing:', item)
```

2 つ目の case ブロックでは「case '.csv' | '.tsv':」として拡張子が '.csv' と '.tsv' の場合の処理をまとめている（実際には CSV ファイルと TSV ファイルでは内部で使用する列の区切り文字が異なるため、このように処理をまとめるのは難しいかもしれない）。

同じことを os.scandir 関数と if 文の組み合わせで行う例は以下の通りだ。

```
import os
for item in os.scandir(d):
    root, ext = splitext(item)  # DirEntryクラスのインスタンスでもOK
    if ext == '.txt':
        print('processing a text file:', item.name)
    elif ext == '.csv' or ext == '.tsv':
        print('processing a csv/tsv file:', item.name)
    else:
        print('do nothing:', item.name)
```

os.scandir 関数で列挙されるのは DirEntry クラスのインスタンスだが、これも path-like object なので、os.path.splitext 関数には問題なく渡せる点にも注意しよう。

splitroot 関数でファイルパスを ドライブ、ルート、それ以外に分割するには

ファイルパスをドライブ、ルート、それ以降に分割するには os.path モジュールの splitroot 関数を使える。その使い方、Windows と UNIX での動作の違い、Windows と同様な分割結果を得るための方法などを紹介する。

かわさきしんじ，Deep Insider 編集部（2024 年 04 月 02 日）

```
from os.path import splitroot

up = '/tmp/foo/bar/baz.txt'
result = splitroot(up)  # UNIX環境ではドライブ要素は常に空文字列
print(result)  # ('', '/', 'tmp/foo/bar/baz.txt')

wp = 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitroot(wp)  # Windows形式のパスは環境で結果が異なる
print(result)
# 出力結果:
# macOS:('', '', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')

import ntpath  # Windowsと同じ結果を得るにはntpathモジュールを使う
result = ntpath.splitroot(wp)
print(result)  # ('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')

# 相対パス
up = 'tmp/foo/bar/baz.txt'
result = splitroot(up)
print(result)  # ('', '', 'tmp/foo/bar/baz.txt')

wp = 'tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitroot(wp)
print(result)  # ('', '', 'tmp¥¥foo¥¥bar¥¥baz.txt')

# ドライブ文字付の相対パス
wp = 'D:tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitroot(wp)
print(result)
# 出力結果:
# macOS:('', '', 'D:tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('D:', '', 'tmp¥¥foo¥¥bar¥¥baz.txt')

result = ntpath.splitroot(wp)
print(result)  # ('D:', '', 'tmp¥¥foo¥¥bar¥¥baz.txt')

# UNCパス
uncp = '¥¥¥¥server¥¥SharedFolders¥¥pytips'
result = splitroot(uncp)
print(result)
```

```
# 出力結果:
# macOS:('', '', '¥¥¥¥server¥¥SharedFolders¥¥pytips')
# Windows:('¥¥¥¥server¥¥SharedFolders', '¥¥', 'pytips')

result = ntpath.splitroot(uncp)
print(result)  # ('¥¥¥¥server¥¥SharedFolders', '¥¥', 'pytips')

# path-like object
from pathlib import Path

up = Path('/tmp/foo/bar/baz.txt')
result = splitroot(up)
print(result)
# 出力結果:
# macOS:('', '/', 'tmp/foo/bar/baz.txt')
# Windows:('', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')

wp = Path('C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
result = splitroot(wp)
print(result)
# 出力結果:
# macOS:('', '', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')

result = ntpath.splitroot(wp)
print(result)  # ('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')
```

os.path.splitroot 関数

os.path モジュールにはパスを特定の条件で分割する関数が幾つかある。

- os.path.split 関数：パスを「(パスの末尾より前 , パスの末尾)」という 2 要素のタプルに分割する
- os.path.splitdrive 関数：パスを「(ドライブ , それ以外)」という 2 要素のタプルに分割する
- os.path.splitext 関数：パスを「(拡張子以外の部分 , 拡張子)」という 2 要素のタプルに分割する
- **os.path.splitroot 関数**：パスを「(ドライブ , パスの末尾より前 , パスの末尾)」という 3 要素のタプルに分割する（Python 3.12 以降）

このうち、以下では os.path.splitroot 関数を使って、パスをドライブ（ドライブ文字）要素とルート要素、それ以外の部分に分割する方法を紹介する。

以下に os.path.splitroot 関数の構文を示す。

```
os.path.splitroot(path)
```

os.path.split 関数はパラメーターを 1 つ持ち、これにパスを渡すと「(ドライブ要素 , ルート要素 , その他の要素)」という 3 つの要素からなるタプルが返される。パスにドライブ文字が含まれるのは Windows なので、UNIX（POSIX）環境では戻り値の第 0 要素は常に空文字列となる。

UNIX 環境では、os.path.splitroot 関数は次のように振る舞う。

- ドライブ要素は常に空文字列
- ルート要素はシングルスラッシュ「/」もしくはダブルスラッシュ「//」、もしくは空文字列（相対パスの場合）
- ルート要素以降が最後の要素に含まれる

Windows 環境では次のように振る舞う。

- ドライブ要素はドライブ文字か、サーバ名と共有名（ファイル共有の UNC パスの場合）、もしくは空文字列（ドライブ指定がない場合）
- ルート要素はパスを区切るセパレーターか（絶対パスの場合）、空文字列（相対パスの場合）。セパレーターは多くの場合バックスラッシュ「¥」が使われる
- ルート要素以降が最後の要素に含まれる

基本的な例を以下に示す。

```
from os.path import splitroot

up = '/tmp/foo/bar/baz.txt'
result = splitroot(up)  # UNIX環境ではドライブ要素は常に空文字列
print(result)  # ('', '/', 'tmp/foo/bar/baz.txt')
```

この例では UNIX 形式の絶対パス（/tmp/foo/bar/baz.txt）を splitroot 関数に渡している。既に述べたように、UNIX ではドライブ要素は常に空となる。この分割結果は Windows 環境でも同じだ。

Windows 形式の絶対パスを分割する例を以下に示す。

```
wp = 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitroot(wp)  # Windows形式のパスは環境で結果が異なる
print(result)
# 出力結果:
# macOS:('', '', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')
```

こちらはコードを実行する環境によって結果が異なる。UNIX 環境ではドライブ要素とルート要素が空文字列で、パス全体が残りの要素に含まれる。Windows 環境ではドライブ文字、ルートとなるバックスラッシュ、それ以外の要素に分割される。UNIX 環境で Windows と同じ結果を得るには、ntpath モジュールの splitroot 関数を使用する。

```
import ntpath  # Windowsと同じ結果を得るにはntpathモジュールを使う
result = ntpath.splitroot(wp)
print(result)  # ('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')
```

相対パスを渡した場合は、ルート要素が空文字列になる。従って、UNIX 環境ではこの場合、ドライブ要素とルート要素が空文字列になる。

```
up = 'tmp/foo/bar/baz.txt'
result = splitroot(up)
print(result)  # ('', '', 'tmp/foo/bar/baz.txt')
```

このことは Windows でも基本的には同じだ。

```
wp = 'tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitroot(wp)
print(result)  # ('', '', 'tmp¥¥foo¥¥bar¥¥baz.txt')
```

ただし、Windows では相対パスにドライブ指定が付加されることがある。その場合は、ドライブ要素は空文字列ではないが、ルート要素は空文字列になる。UNIX 環境でそうした相対パスを処理すると、全てが残りの要素に含まれる（Windows 形式のパスは全てそうなる）。既に見たように、ntpath.splitroot 関数を使えば同じ結果が得られる。

```
wp = 'D:tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitroot(wp)
print(result)
# 出力結果:
# macOS:('', '', 'D:tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('D:', '', 'tmp¥¥foo¥¥bar¥¥baz.txt')

result = ntpath.splitroot(wp)
print(result)  # ('D:', '', 'tmp¥¥foo¥¥bar¥¥baz.txt')
```

ファイル共有で使われる UNC パスを Windows 環境で os.path.splitroot 関数に渡すと、サーバ名と共有名（とディレクトリのセパレーター）がドライブ要素に含まれる。ルート要素はディレクトリのセパレーターとなり、それ以降が残りの要素に含まれる。以下に例を示す。

```
uncp = '¥¥¥¥server¥¥SharedFolders¥¥pytips'
result = splitroot(uncp)
print(result)
# 出力結果:
# macOS:('', '', '¥¥¥¥server¥¥SharedFolders¥¥pytips')
# Windows:('¥¥¥¥server¥¥SharedFolders', '¥¥', 'pytips')

result = ntpath.splitroot(uncp)
print(result)  # ('¥¥¥¥server¥¥SharedFolders', '¥¥', 'pytips')
```

最後に os.path.splitroot 関数には path-like object も渡せる。以下に例を示す（説明は不要だろう）。

```
from pathlib import Path

up = Path('/tmp/foo/bar/baz.txt')
result = splitroot(up)
print(result)
# 出力結果:
# macOS:('', '/', 'tmp/foo/bar/baz.txt')
# Windows:('', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')

wp = Path('C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
result = splitroot(wp)
print(result)
# 出力結果:
# macOS:('', '', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')

result = ntpath.splitroot(wp)
print(result)  # ('C:', '¥¥', 'tmp¥¥foo¥¥bar¥¥baz.txt')
```

splitdrive 関数でファイルパスを ドライブ文字とその他の部分に分割するには

Windows ではファイルパスにドライブ文字が含まれる場合がある。os.path モジュールの splitdrive 関数を使って、ドライブ文字とその他に分割する方法や、UNIX でこれと同様な処理を行う方法などを紹介する。

かわさきしんじ，Deep Insider 編集部（2024 年 03 月 26 日 ）

```
from os.path import splitdrive

wp = 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitdrive(wp)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')

# UNIX形式のパスを渡すとドライブ要素は常に空文字列
up = '/tmp/foo/bar/baz.txt'
result = splitdrive(up)
print(result)  # ('', '/tmp/foo/bar/baz.txt')

# Windows上で動作するPythonと同様にパスを分割する
import ntpath

result = ntpath.splitdrive(wp)
print(result)  # ('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')

uncp = '¥¥¥¥server¥¥SharedFolders¥¥pytips'
result = splitdrive(uncp)
print(result)
# macOS:('', '¥¥¥¥server¥¥SharedFolders¥¥pytips')
# Windows:('¥¥¥¥server¥¥SharedFolders', '¥¥pytips')

uncp = '¥¥¥¥.¥¥z:¥¥pytips'
result = splitdrive(uncp)
print(result)
# 出力結果:
# macOS:('', '¥¥¥¥.¥¥z:¥¥pytips')
# Windows:('¥¥¥¥.¥¥z:', '¥¥pytips')

# 相対パス
up = 'foo/bar/baz.txt'
result = splitdrive(up)
print(result)  # ('', 'foo/bar/baz.txt')

wp = 'D:foo¥¥bar¥¥baz.txt'  # fooはDドライブのカレントディレクトリにある
result = splitdrive(wp)
print(result)
```


→目次に戻る

```
# 出力結果:
# macOS:('', 'D:foo¥¥bar¥¥baz.txt')
# Windows:('D:', 'foo¥¥bar¥¥baz.txt')

# path-like object
from pathlib import Path

wp = Path('C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
result = splitdrive(wp)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')

result = ntpath.splitdrive(wp)
print(result)  # ('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
```

os.path.splitdrive 関数

os.path モジュールにはパスを特定の条件で分割する関数が幾つかある。

- os.path.split 関数：パスを「(パスの末尾より前 , パスの末尾)」という 2 要素のタプルに分割する
- **os.path.splitdrive 関数**：パスを「(ドライブ , それ以外)」という 2 要素のタプルに分割する
- os.path.splitext 関数：パスを「(拡張子以外の部分 , 拡張子)」という 2 要素のタプルに分割する
- os.path.splitroot 関数:パスを「(ドライブ , パスの末尾より前 , パスの末尾)」という 3 要素のタプルに分割する（Python 3.12 以降）

このうち、以下では os.path.splitdrive 関数を使って、パスをドライブ（ドライブ文字）とそれ以外の部分に分割する方法を紹介する。

以下に os.path.splitdrive 関数の構文を示す。

```
os.path.splitextdrive(path)
```

os.path.split 関数はパラメーターを 1 つ持ち、これにパスを渡すと「(パスのドライブ文字 , その他の要素)」という 2 つの要素からなるタプルが返される。パスにドライブ文字が含まれるのは Windows なので、UNIX（POSIX）環境では戻り値の第 0 要素は常に空文字列となる。

基本的な例を以下に示す。

```
from os.path import splitdrive

wp = 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt'
result = splitdrive(wp)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
```

この例では、変数 wp には Windows 形式の絶対パスを代入している。これを os.path.splitdrive 関数に渡すと、戻り値は UNIX ならドライブ部分は空文字列になり、残りの部分に関数へ渡したパスがそのまま含まれる。一方、Windows ではドライブ文字とその他の部分が分割される。

UNIX 形式のパスでは UNIX でも、Windows でも戻り値のドライブ要素は空文字列となる。

```
up = '/tmp/foo/bar/baz.txt'
result = splitdrive(up)
print(result)  # ('', '/tmp/foo/bar/baz.txt')
```

UNIX 環境で Windows 形式のパスを操作する場合に、Windows 環境と同様なパス分割をしたければ ntpath モジュールをインポートして、その splitdrive 関数を使用する。

```
import ntpath

result = ntpath.splitdrive(wp)
print(result)  # ('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
```

Windows のファイル共有パス（UNC パス）を与えた場合、UNIX 環境ではドライブ文字要素は空文字列に、Windows 環境ではドライブ文字要素にはサーバ名と共有名がドライブ文字要素になり、その他の部分と分割される。以下に例を示す。

```
uncp = '¥¥¥¥server¥¥SharedFolders¥¥pytips'
result = splitdrive(uncp)
print(result)
# 出力結果:
# macOS:('', '¥¥¥¥server¥¥SharedFolders¥¥pytips')
# Windows:('¥¥¥¥server¥¥SharedFolders', '¥¥pytips')

uncp = '¥¥¥¥.¥¥z:¥¥pytips'
result = splitdrive(uncp)
print(result)
# 出力結果:
# macOS:('', '¥¥¥¥.¥¥z:¥¥pytips')
# Windows:('¥¥¥¥.¥¥z:', '¥¥pytips')
```

相対パスを指定した場合も、ドライブ文字が含まれていれば、分割される。

```
# 相対パス
up = 'foo/bar/baz.txt'
result = splitdrive(up)
print(result)  # ('', 'foo/bar/baz.txt')

wp = 'D:foo¥¥bar¥¥baz.txt'  # fooはDドライブのカレントディレクトリにある
result = splitdrive(wp)
print(result)
# 出力結果:
# macOS:('', 'D:foo¥¥bar¥¥baz.txt')
# Windows:('D:', 'foo¥¥bar¥¥baz.txt')
```

最初の例は UNIX 形式の相対パスなので、ドライブ文字要素は空文字列になる。次の例は Windows 形式の相対パスだが、パスの先頭に「D:」とドライブ文字がある点に注意。foo ディレクトリは D: ドライブのカレントディレクトリにあるということだ。これを os.path.splitdrive 関数に渡すと、UNIX ではドライブ文字要素は空文字列に、Windows ではドライブ文字と他の部分が分割される。

path-like object を渡すこともできる。簡単な例を以下に示す。

```
from pathlib import Path

wp = Path('C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
result = splitdrive(wp)
print(result)
# 出力結果:
# macOS:('', 'C:¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
# Windows:('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')

result = ntpath.splitdrive(wp)
print(result)  # ('C:', '¥¥tmp¥¥foo¥¥bar¥¥baz.txt')
```

なお、本稿冒頭では UNIX（POSIX）環境ではドライブ文字要素は常に空文字列になると書いたが、これは posixlib.splitdrive 関数の実装が次のようになっているからだ。

```
def splitdrive(p):
    """Split a pathname into drive and path. On Posix, drive is always
    empty."""
    p = os.fspath(p)
    return p[:0], p
```

os.fspath 関数はパスのファイルシステム表現となる文字列またはバイト列を返す。そして、その先頭から 0 番目の要素の手前まで（つまり、空文字列か空のバイト列）と、パスのファイルシステム表現をタプルにまとめたものを戻り値とするようになっている。また、コメントにも「POSIX では、drive 要素は常に空」とある。そのため、本稿でもそのように表現している。

編集：@IT 編集部
発行：アイティメディア株式会社