@IT
atmarkIT

# Python チートシート

かわさきしんじ, Deep Insider 編集部［著］

# ［Python チートシート］基本要素編

組み込みのデータ型、変数、制御構造、演算子、関数定義など、Python プログラムを構成する基本要素をギュッとまとめて紹介する。

かわさきしんじ，Deep Insider 編集部（2020 年 01 月 09 日）

Python のコードを書くときに「ここってどう書くんだっけ?」となることがある。本章は組み込みのデータ型、変数、制御構文、演算子、関数定義についてまとめる（基本の基本なので忘れることも少ないかもしれないが）。

## 変数とデータ型

Python では変数には型がなく、変数が参照するオブジェクトに型がある。Python に組み込みのデータ型としては以下が用意されている（一部）。

| データ型 | 説明 | 例 |
|---|---|---|
| **int** | 整数 | 0、128、1_234_567（アンダースコアによる桁区切り）、0b1010（2進表記）、-0o11（8進表記）、0x1111（16進表記） |
| **float** | 浮動小数点 | 1.0、-1_000.000_123（アンダースコアによる桁区切り）、1.2e+2（指数表記） |
| **complex** | 「実数部＋虚数部」からなる複素数。虚数部の末尾には「j」を付加 | 1+1j、(2.5-2.5j) |
| **str** | 文字列。変更不可能 | 'string'、"double quote"、'''triple quote'''、"""triple quote"""、r"abc¥n"（raw文字列。「¥n」はエスケープシーケンスではなく、「¥」と「n」が連続したものと見なされる）、f"value: {x}"（f文字列。「{}」で囲まれた部分に変数屋敷の値が埋め込まれる） |
| **bytes** | バイナリデータ。変更不可能 | b'hello world'、b"goodbye world"、b'''triple quoted bin data'''、b"""triple quoted bin data""" |
| **bytearray** | 変更可能なバイナリデータ | bytearray(b'hello world')（bytes型の値をbytearray関数に渡して変更可能なバージョンを取得） |
| **bool** | 真偽値。TrueかFalseのみをその値とする | True、False |
| **list** | リスト。複数要素を格納する | [0, 1, 2]、[[0, 1], [2, 3]]（リストを要素とするリスト）、[x for x in range(10)]（リスト内包表記。0〜9を要素とするリストを生成） |
| **tuple** | タプル。複数要素を格納する。変更不可能 | (0, 1, 2)、((0, 1), (2, 3))（タプルを要素とするタプル）、([0, 1], [2, 3])（リストを要素とするタプル。格納されたリストの要素は変更可能）、()（要素を持たないタプル）、(1,)（要素を1つだけ持つタプル） |
| **dict** | キー／値の組を格納する | {'name': 'deep insider'}、{'zero': 0, 'one': 1}、{key: value for key, value in zip(['zero', 'one'], [0, 1])}（辞書内包表記） |
| **set** | 集合。1つの集合の中で要素の重複はなく、要素に順序もない | {0, 1, 2}、{x for x in range(10)}（集合内包表記） |
| **frozenset** | 変更不可能な集合 | frozenset([0, 1, 2])（リスト[0, 1, 2]をfrozenset関数に与えて、リストと同じ要素を持つ変更不可能な集合を得る） |
| **range** | 整数の範囲。スタート値、ストップ値、ステップ値を持つ。実際の範囲にはストップ値は含まれない（ストップ値より小さな整数の範囲となる） | range(5)（0〜4の範囲）、range(0, 10, 2)（0〜9の範囲だが、ステップ値が2なので偶数のみから成る） |

Python に組み込みのデータ型（一部）

各データ型にはその名前と同じ名前の関数が用意されているので、その関数を使い、その型のオブジェクトを作成できる（他の言語におけるコンストラクタと似た効果を持つ）。以下に例を示す。

```
print(int('100'))  # 文字列'100'から整数値100を作成
print(float(1))  # 整数値1から浮動小数点数値1.0を作成
print(str(100))  # 整数値100から文字列'100'を作成
print(list('hello'))  # 文字列'hello'の各文字を要素とするリストを作成
print(tuple([0, 1, 2]))  # 引数に指定したリストと同じ要素を持つタプルを作成
print(tuple())  # 要素を持たないタプルを作成
print(tuple([1]))  # 要素を1つだけ持つタプルを作成
print(dict(foo='foo', bar='bar'))  # dict関数にキーワード引数を渡して辞書を作成
```

**関数を使用した各データ型のオブジェクトの定義例**

変数は事前に定義や宣言をせずに必要なところで使用できる。ただし、変数の値を読み出す前には初期化（代入）が必要。

```
x = 1  # 変数は宣言なしに使用可能
x += 1  # 変数xの値に1を加算
mylist = [range(0, 10, 2)]  # 変数mylistにリスト[0, 2, 4, 6, 8]を代入
mylist = tuple(mylist)  # 変数には型がないので、任意の型のオブジェクトを代入可能
print(not_defined)  # 初期化をしていない変数not_definedの値を表示（例外が発生）
```

**変数の使用例**

既に述べたように、変数には型がないので、任意の型のオブジェクトを代入できる。上記コード例の 4 行目では「mylist」という名前の変数にタプルを代入しているが、これはエラーとはならない（が、コードの可読性を下げるので推奨はされない）。最後の行では、初期化をしていない変数の値を利用しようとしているので、次のように例外が発生する。

```
In [2]: x = 1  # 変数は宣言なしに使用可能
        x += 1  # 変数xの値に1を加算
        mylist = [range(0, 10, 2)]  # 変数mylistにリスト[0, 2, 4, 6, 8]を代入
        mylist = tuple(mylist)  # 変数には型がないので、任意の型のオブジェクトを代入可能
        print(not_defined)  # 初期化をしていない変数not_definedの値を表示（例外が発生）
        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-2-6a69bf406ca4> in <module>
              3 mylist = [range(0, 10, 2)]  # 変数mylistにリスト[0, 2, 4, 6, 8]を代入
              4 mylist = tuple(mylist)  # 変数には型がないので、任意の型のオブジェクトを代入可能
        ----> 5 print(not_defined)  # 初期化をしていない変数not_definedの値を表示（例外が発生）

        NameError: name 'not_defined' is not defined
```

**実行結果**

## 制御構造

Python にはプログラムの実行の流れ（フロー）を制御する構文として以下がある。

| 構文 | 要素 |
|---|---|
| **if文** | 条件分岐 |
| **for文** | 繰り返し処理 |
| **while文** | 繰り返し処理 |
| **break文** | 繰り返し処理の中断 |
| **continue文** | 次の繰り返し処理の開始 |

Python の制御構造

以下ではこれらについてまとめる。なお、これらの制御構文に限らず、Python の複合文（複数の文で構成される文）は、一般に各節の先頭行はコロン「:」で終了し、その後は空白文字によるインデントを付けて、その節に含まれるブロック（コードブロック）を明記する。

### if 文

if 文は何らかの条件に応じて、実行したい処理を分岐させるために使用する。その構文を以下に示す。

```
if 条件0:
    条件0が真だったときに実行するブロック
elif 条件1:
    条件0が偽で、条件1が真だったときに実行するブロック
elif 条件2:
    条件0と条件1が偽で、条件2が真だったときに実行するブロック
else:
    全ての条件が偽だったときに実行するブロック
```

if 文の構文

elif 節は任意の数だけ繰り返すことが可能。また、elif 節と else 節は省略可能である。

条件には比較演算子を使った値比較を記述する。ただし、Python では、全てのオブジェクトが真または偽として評価できるので、直接オブジェクト（を参照する変数）を条件として記述することも可能だ。偽となるのは以下のものだ。

- None（オブジェクトがないことを示す値）と False
- 整数の 0、浮動小数点数の 0.0 など、ゼロを表す数値型の値
- 空文字列、空のリスト、空のタプルなど、要素を持たない反復可能オブジェクト

これらのオブジェクトを条件に指定すると、それは成り立たないことになる。他のオブジェクトは全て真の値として取り扱われる。

以下に例を示す。

```
condition1 = 1
if condition1 == 1:  # elif節とelse節のないif文
    print('condition is True')

name = ""  # 空文字列を変数nameに代入
if name:  # 空文字列は偽として扱われる
    print(f'name: {name}')
else:  # elif節を省略
    print('no name')

condition2 = 100
if condition1 != 0 and condition2 == 0:  # 比較演算子と論理演算子による条件記述
    print('some result')
elif condition1 == 10:
    print('another result')
else:
    print('final result')
```

if 文の使用例

なお、Python には他の言語に見られる switch 文がないので、条件による処理の分岐には if 文を常に使用する。

## for 文

Python の for 文は、リストなどの反復可能オブジェクトの要素を 1 つずつ取り出して、それを使用して何らかの処理を行う際に使用することが多い。あるいは、range 関数を使用することで特定回数の繰り返し処理を行うことも可能だ。その構文を以下に示す。

```
for ループ変数 in 反復可能オブジェクト:
    反復可能オブジェクトから取り出した値（ループ変数の値）を使った処理
else:
    反復可能オブジェクトの要素がなくなったときに行う処理
```

for 文の構文

if 文と同様に、else 節は省略可能だ。else 節は反復可能オブジェクトの要素がなくなり、繰り返し処理が終了したときに実行される（反復可能オブジェクトが空の場合には繰り返し処理を行わずに else 節のコードが実行される）。ただし、for 文で行う繰り返し処理の中で break 文が実行された場合には、else 節は実行されない（後述）。

以下に例を示す。

```
mylist = list(range(5))
for num in mylist:
    if num % 2 == 0:  # 偶数なら
        print(f'{num}は偶数です')
    else:  # 奇数なら
        print(f'{num}は奇数です')

mystr = ''  # 空文字列
for ch in mystr:  # mystrは空文字列なのでelse節のみが実行される
    print(ch, end='')
else:
    print()
    print('finished')
```

for 文の使用例

## while 文

while 文は、それに指定した条件が真である限り、処理を繰り返すのに使用する。構文を以下に示す。

```
while 条件:
    条件が真の間、繰り返す処理
else:
    条件が偽になったときに行う処理
```

while 文

if 文や for 文と同様に、else 節は省略可能だ。else 節は条件が偽となり、繰り返し処理が終了する際に実行される。ただし、while 文で行う繰り返し処理の中で break 文が実行された場合には else 節は実行されない（後述）。

以下に例を示す。

```
counter = 0
while counter < 5:
    print(counter)  # 変数counterの値が5より小さい間実行する処理
    counter += 1
else:
    print('finished')  # 繰り返し処理が終わったときに実行する処理
```

while 文の使用例

## break 文と continue 文

break 文は繰り返し処理の中で、一定の条件が成立したら繰り返しを終了させるのに使用する。以下に例を示す。

```
for num in range(5):  # ループ変数numの値は0～4に変化する
    if num == 3:  # ループ変数numの値が3ならループを終了
        break
    print(num)  # このprint関数呼び出しはelse節に書いても書かなくても同じこと
else:
    print('finished')
```

**break 文の使用例**

この例では、ループ変数 num の値が 3 のときに break 文が実行される。その時点で繰り返し処理が終了するので、ループ変数の値が 4 になることはない。また、print 関数を呼び出す前に break 文が実行されるので、この値が出力されることもないし、else 節が実行されることもない。実行結果を以下に示す。

```
In [6]: for num in range(5):  # ループ変数numの値は0～4に変化する
            if num == 3:  # ループ変数numの値が3ならループを終了
                break
            print(num)  # このprint関数呼び出しはelse節に書いても書かなくても同じこと
        else:
            print('finished')

        0
        1
        2
```

**実行結果**

一方、continue 文は繰り返し処理の中で、一定の条件が成立したら、次の繰り返しを開始するのに使用する。以下に例を示す。

```
for num in range(5):
    if num % 2 == 0:  # 変数numの値が偶数なら次の繰り返しを開始
        continue
    print(num)
else:
    print('finished')
```

**continue 文の使用例**

この例では、ループ変数 num の値が偶数のときに continue 文が実行される。これにより繰り返し処理の新しい繰り返しが開始されるので、print 関数呼び出しが実行されない。そのため、画面には奇数のみが表示される。また、最後には else 節も実行される。

## 演算子

Python では多くの演算子が提供されている。その幾つかを以下に抜粋してまとめる。

### 算術演算子

算術演算子を以下にまとめる。

| 演算子 | 説明 | 例 |
|---|---|---|
| ** | べき乗（累乗） | 2 ** 10→1024 |
| 単項+ | 被演算子の符号をそのままとする | +2→2 |
| 単項- | 被演算子の符号を反転する | -(-2)→2 |
| * | 乗算、文字列の繰り返し | 2 * 5→10、'foo' * 2→'foofoo' |
| / | 除算 | 1 / 2→0.5 |
| // | 整数除算の商 | 7 // 3→2 |
| % | 整数除算の剰余 | 7 % 3→1 |
| + | 加算、文字列の結合 | 1 + 9→10、'foo' + 'bar'→'foobar' |
| - | 減算 | 10 - 1→9 |

Python の四則演算子

### 代入演算子／累算代入演算子

代入演算子と累算代入演算子を以下にまとめる。例では変数 x の値を 10 とし、累算代入演算子では「→」の後に変数 x に代入された値を示す。

| 演算子 | 説明 | 例 |
|---|---|---|
| = | 代入 | x = 10 |
| += | 左辺の値に右辺の値を加算した値を左辺に代入 | x += 10→20（x = x + 10に相当） |
| -= | 左辺の値から右辺の値を減算した値を左辺に代入 | x -= 10→0（x = x - 10に相当） |
| *= | 左辺の値と右辺の値を乗算した値を左辺に代入 | x *= 2→20（x = x * 2に相当） |
| /= | 左辺の値を右辺の値で除算した値を左辺に代入 | x /= 2→5.0（x = x / 2に相当） |
| //= | 左辺の値を右辺の値で整数除算した商を左辺に代入 | x //= 3→3（x = x // 3に相当） |
| %= | 左辺の値を右辺の値で整数除算した剰余を左辺に代入 | x %= 3→1（x = x % 3に相当） |
| **= | 左辺の値を右辺の値で累乗した値を左辺に代入 | x **= 2→100（x = x ** 2に相当） |
| >>= | 左辺の値を右辺の値だけ右ビットシフトした値を左辺に代入 | x >>= 1→5（x = x >> 1に相当） |
| <<= | 左辺の値を右辺の値だけ左ビットシフトした値を左辺に代入 | x <<= 1→20（x = x << 1に相当） |
| &= | 左辺の値と右辺の値のビット単位論理積を左辺に代入 | x &= 8→8（x = x & 8に相当） |
| \|= | 左辺の値と右辺の値のビット単位論理和を左辺に代入 | x \|= 7→15（x = x \| 7に相当） |
| ^= | 左辺の値と右辺の値のビット単位排他的論理和を左辺に代入 | x ^= 12→6（x = x ^ 12に相当） |

代入演算子／累算代入演算子

## 比較演算子とブール演算子

比較演算子とブール演算子は if 文や while 文の条件を記述するときなどに使用する。

| 演算子 | 説明 | 例 |
|---|---|---|
| == | 2つのオブジェクトの値が等しいかどうか | 1 == 0→False<br>[2, 3] == [2, 3]→True |
| != | 2つのオブジェクトの値が等しくないかどうか | 1 != 0→True<br>'foo' != 'foo'→False |
| > | 左側のオブジェクトの値が右側のオブジェクトの値よりも大きいかどうか | 0 > 1→False<br>[1, 2, 3] > [1, 2]→True |
| >= | 左側のオブジェクトの値が右側のオブジェクトの値以上かどうか | 0 >= 0→True<br>[1, 2] >= [1, 2, 3]→False |
| < | 左側のオブジェクトの値が右側のオブジェクトの値よりも小さいかどうか | 0 < 1→True<br>[1, 2, 3] < [1, 2]→False |
| <= | 左側のオブジェクトの値が右側のオブジェクトの値以下かどうか | 0 <= 0→True<br>[1, 2, 3] <= [1, 2]→False |
| is | 2つのオブジェクトが同一のオブジェクトかどうか | True is True→True |
| is not | 2つのオブジェクトが同一のオブジェクトでないかどうか | True is not True→False |
| in | 左側のオブジェクトが右側のオブジェクトに含まれているかどうか | 1 in [1, 2, 3]→True<br>[1] in [1, 2, 3]→False |
| not in | 左側のオブジェクトが右側のオブジェクトに含まれていないかどうか | 1 not in [1, 2, 3]→False<br>[1] not in [1, 2, 3]→True |
| and | 全ての条件が真かどうか | 1 in [1, 2, 3] and 10 in [1, 2, 3]→False<br>'Py' in 'Python' and 'R' in 'Ruby'→True |
| or | 条件のいずれかが真かどうか | 'Py' in 'Ruby' or 'Python' in ['Python', 'Ruby']→True<br>0 > 1 or 1 > 10→False |
| not | 直後の条件の真偽を反転する | not 'Py' in 'Python'→False<br>not 'P' in 'Ruby'→True |

比較演算子とブール演算子

## ビット演算子

ビット演算子は、整数値を引数として、その 2 進数表現に対して処理を行う。例では各整数値のビット表現が分かりやすくなるように「0b」を前置して 2 進数表記としている。

| 演算子 | 説明 | 例 |
|---|---|---|
| & | 2つの整数値のビット単位論理積 | 0b1010 & 0b1000→8（0b1000） |
| \| | 2つの整数値のビット単位論理和 | 0b1010 \| 0b0111→15（0b1111） |
| ^ | 2つの整数値のビット単位排他的論理和 | 0b1010 ^ 0b0110→12（0b1100） |
| >> | 左側の値を右側の値だけ右シフト | 0b1010 >> 1→5（0b0101） |
| << | 左側の値を右側の値だけ左シフト | 0b1010 << 1→20（0b10100） |

ビット演算子



→目次に戻る

### 三項演算子

三項演算子（条件式）は、条件に応じてその値を変える式である。構文を以下に示す。

```
値1 if 条件 else 値2
```

三項演算子（条件式）

条件が真のときには値 1 が、そうでなければ値 2 がこの演算子（式）の値となる。また、これは else 以降に三項演算子を続けることもできる。以下に例を示す。

```
a = 1
x = 'foo' if a == 1 else 'bar'

y = 'insider.net' if a == 0 else 'deep insider' if x == 'foo' else 'atmarkit'
print(y)
```

三項演算子の使用例

最初の三項演算子の使用箇所では、変数 a の値が 1 なら変数 x に 'foo' が代入され、そうでなければ 'bar' が代入される。その次は、三項演算子をネストさせて、変数 a の値が 0 なら変数 y に 'insider.net' が代入され、そうでなければ変数 x の値を調べて、これが 'foo' なら 'deep insider' が代入され、そうでなければ 'atmarkit' が代入されるようにしている。ここでは、変数 a の値は 1 で、変数 x の値は 'foo' になっているので、変数 y には 'deep insider' が代入される。

## 関数定義の基本

関数は def 文で定義する。その基本構文を以下に示す。

```
def 関数名(パラメーターリスト):
    関数本体のコード
    return 戻り値
```

関数定義の基本

パラメーターリストには、関数呼び出し時に呼び出し側から受け取る値を保存する変数をカンマ区切りで並べる。パラメーターにはデフォルト引数値を指定できる。指定する場合には「パラメーター名 = デフォルト引数値」のように記述する。デフォルト引数値を持つパラメーターについては関数呼び出しに引数指定を省略でき、その場合にはデフォルト引数値が指定されたものとして扱われる。

関数が何かを処理して、呼び出し側に値を返送するときには、return 文にその値を並べる。複数の値を返送するときにはそれらをカンマ区切りで並べる。

以下に例を示す。

```
from math import sqrt

def get_distance(x1, y1, x2=0, y2=0):
    distance = sqrt((x1-x2) ** 2 + (y1-y2) ** 2)
    return distance

print(get_distance(1, 1, 0, 0))
print(get_distance(2, 2))
```

**関数定義の例**

この例では math モジュールから sqrt 関数をインポートして、それを使って、2 点間の距離を求める関数を定義している。パラメーターリストには x1、y1、x2、y2 があり、それぞれ最初の点の x 座標と y 座標、もう一つの点の x 座標と y 座標を受け取る。このとき、2 つのパラメーター x2 と y2 にはデフォルト引数値を指定している。そのため、第 3 引数と第 4 引数は関数呼び出し時に省略できる。sqrt 関数を使って距離を計算したら、最後に return 文でその値を返送する。

最後の 2 行はこの関数の呼び出し例で、最後の行では第 3 引数と第 4 引数の指定を省略している。

本章は Python の基本構成要素や基本となる構文をまとめた。次章は関数定義についてほんの少しだが詳しく、その構文をまとめる。

# ［Python チートシート］関数定義編

Python の関数定義と、位置引数／キーワード引数／可変長法引数を受け取る方法。ラムダ式についてギュッとまとめた。

かわさきしんじ，Deep Insider 編集部（2020 年 01 月 14 日）

前章の最後には関数定義の基本を簡単にまとめた。本章では関数定義について少し詳しくまとめよう。関数についての詳細は「Python 入門」の「Python の関数」など関数について触れている回を参照されたい。

## 関数定義の基本

前章で述べたように、Python で関数を定義する際の基本構文は以下のようになる。

```
def 関数名(パラメーターリスト):
    関数本体のコード
    return 戻り値
```

**関数定義の基本構文**

呼び出し側から受け取る値を保存する変数をパラメーターリストに記述し、関数本体のコードではそれらを利用して、何らかの処理を行い、return 文で呼び出し側に値を返すのが基本だ。ただし、Python では関数に値（引数、実引数）を渡す際には幾つかのやり方がある。以下ではそれらについてまとめていこう。

## 位置引数とキーワード引数

関数を呼び出す際に、それに渡す値のことを引数（実引数）と呼ぶ。引数の渡し方にはおおまかに分けて次の 2 種類がある。

- 位置引数：関数呼び出し時にカンマ区切りで並べた引数がそのままの順序でパラメーターに渡される
- キーワード引数：関数定義時のパラメーターリストに示したパラメーター名を使い、「パラメーター名 = 値」とすることで、パラメーターとその値を明記する引数の渡し方

以下に例を示す。

```
from math import sqrt

def get_distance(x1, y1, x2, y2):
    distance = sqrt((x1-x2) ** 2 + (y1-y2) ** 2)
    return distance

distance = get_distance(1, 1, 0, 0)  # 位置引数
print(distance)
distance = get_distance(x1=2, y1=2, x2=0, y2=0)  # キーワード引数
print(distance)
distance = get_distance(3, 3, y2 = 0, x2 = 0)  # 位置引数とキーワード引数の混在
print(distance)
```

**位置引数とキーワード引数を使った呼び出し**

この例では位置引数のみ、キーワード引数のみ、位置引数とキーワード引数の混在の 3 種類方法で関数に引数を渡している。これらは全て問題なく動作する。

ただし、位置引数とキーワード引数を混在させる際には、「キーワード引数を指定するのは位置引数よりも後」という制約があることには注意が必要だ。例えば、以下のコードは例外を発生する。

```
distance = get_distance(x1 = 1, 1, 2, 2)  # 例外
```

**キーワード引数は位置引数よりも後で指定する必要がある**

```
In [2]: distance = get_distance(x1 = 1, 1, 2, 2)  # 例外
  File "<ipython-input-2-b01e59dea27e>", line 1
    distance = get_distance(x1 = 1, 1, 2, 2)  # 例外
                                    ^
SyntaxError: positional argument follows keyword argument
```

**実行結果**

なお、Python 3 では以前よりパラメーターリストで「キーワード専用パラメーター」を指定可能だった（キーワード引数形式でしか値を渡せないパラメーター）。これに追加して、Python 3.8 以降では「位置専用パラメーター」つまり「位置引数としてのみ値を渡せる」パラメーターを明記できるようになった。パラメーターリストの記述時に、スラッシュ「/」を置くと、それよりも前にあるパラメーターは「位置専用パラメーター」になり、アスタリスク「*」を置くと、それよりも後ろにあるパラメーターには「キーワード専用パラメーター」になる。

以下に例を示す。

```
def myfunc(a, b, /, c, *, d, e):
    print(f'a: {a}, b: {b}, c: {c}, d: {d}, e: {e}')
```

**位置専用パラメーターとキーワード専用パラメーターの指定を含んだパラメーターリスト**

この場合、スラッシュ「/」よりも前にある 2 つのパラメーター a と b は位置専用パラメーターだ。一方、アスタリスク「*」よりも後ろにある 2 つのパラメーター d と e にキーワード専用パラメーターだ。間にあるパラメーター c は位置引数としても、キーワード引数としても値を渡せる。以下に例を示す。

```
myfunc(0, 1, 2, d=3, e=4)  # 問題なし
myfunc(0, 1, d=3, e=4, c=2)  # 問題なし
myfunc(a=0, b=1, c=2, d=3, e=4)  # 例外
myfunc(0, 1, 2, 3, 4)  # 例外
```

**位置専用パラメーターとキーワード専用パラメーターを持つ関数の呼び出し例**

上の例では、最初の 2 行は例外を発生させることなく呼び出せる。これは最初の 2 つの位置専用パラメーターは位置引数を使って値を渡し、他のパラメーターにはキーワード引数形式で値を渡しているからだ。その下の 2 行は位置専用パラメーターにキーワード引数を使って値を渡そうとしたり、キーワード専用パラメーターに位置引数形式で値を渡そうとしたりしているために例外が発生する。

以下は Python 3.8.1 上に Jupyter Notebook 環境を構築して、その上で上記コードを実行した結果だ（3 行目と 4 行目では例外の種類が違うので、環境がある方は試してみよう）。

```
In [1]: def myfunc(a, b, /, c, *, d, e):
            print(f'a: {a}, b: {b}, c: {c}, d: {d}, e: {e}')

In [4]: myfunc(0, 1, 2, d=3, e=4)  # 問題なし
        myfunc(0, 1, d=3, e=4, c=2)  # 問題なし
        myfunc(a=0, b=1, c=2, d=3, e=4)  # 例外
        myfunc(0, 1, 2, 3, 4)  # 例外

        a: 0, b: 1, c: 2, d: 3, e: 4
        a: 0, b: 1, c: 2, d: 3, e: 4

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-4-38e12a446cd6> in <module>
            1 myfunc(0, 1, 2, d=3, e=4)  # 問題なし
            2 myfunc(0, 1, d=3, e=4, c=2)  # 問題なし
        ----> 3 myfunc(a=0, b=1, c=2, d=3, e=4)  # 例外
            4 myfunc(0, 1, 2, 3, 4)  # 例外

        TypeError: myfunc() got some positional-only arguments passed as keyword arguments: 'a, b'
```

**実行結果**

## デフォルト引数値

関数のパラメーターには「デフォルト引数値」を持たせることができる。デフォルト引数値を持つパラメーターは関数呼び出し時に、その値の指定を省略でき、省略した場合にはデフォルト引数値が指定されたものとして扱われる。関数が多くのパラメーターを持ち、それらが一般には既定の値（デフォルト値）を指定すればよいという状況ではデフォルト引数値を持たせると関数を呼び出す側が毎回毎回同じ値を指定する必要がなくなり、便利に使えるようになる。

デフォルト引数値はパラメーターリストの中で「パラメーター名 = デフォルト引数値」のようにして指定する。ただし、デフォルト引数値を持つパラメーターは、それを持たないパラメーターよりもリストの中で後ろに置く必要がある。以下に例を示す。

```
from math import sqrt

def get_distance(x1, y1, x2=0, y2=0):
    distance = sqrt((x1-x2) ** 2 + (y1-y2) ** 2)
    return distance
```

**デフォルト引数値を持つパラメーターを含んだ関数の例**

これは 2 点間の位置を計算する関数だが、ある 1 点と原点との距離を計算することはよくある。そのため、ここでは第 3 パラメーターと第 4 パラメーターでは原点を意味する整数値 0 をデフォルト引数値として指定している。

呼び出し例を以下に示す。最初の例では全ての引数を指定し、次の例ではデフォルト引数値を省略している（実行結果は省略）。

```
print(get_distance(2, 2, 1, 1))  # 全ての引数を指定
print(get_distance(2, 2))  # デフォルト引数値を持つものについては指定を省略
```

**パラメーターがデフォルト引数値を持つ関数の呼び出し例**

## 可変長引数

関数に任意の個数の引数を渡せるようにもできる。これには可変長引数リスト（任意引数リスト）を使用する。既に見た通り、Python では引数渡しの方法に位置引数とキーワード引数の 2 つの方法があるため、可変長引数についてもこれらに対応したものがある。

- 可変長位置引数を受け取るパラメーター：パラメーターリスト内でアスタリスク「*」を前置してパラメーターを置く
- 可変長キーワード引数を受け取るパラメーター：パラメーターリスト内でアスタリスク 2 つ「**」を前置してパラメーターを置く

前者には、関数呼び出し時に関数に渡された位置引数のうち、パラメーターリストで対応するものを持たないものがタプルの要素として保管される。このパラメーターの名前としては「args」を使うのが一般的だ。同様に後者には、関数呼び出し時に関数に渡されたキーワード引数のうち、パラメーターリストで対応するものを持たないものが辞書の要素として保管される。このパラメーターの名前としては「kwargs」を使うのが一般的だ。

以下に例を示す。

```
def myfunc2(a, b, *args, **kwargs):
    print(f'a: {a}, b: {b}, args: {args}, kwargs: {kwargs}')
```

**可変長引数を受け取るパラメーターを持つ関数の例**

この例では、パラメーター a と b は位置引数としてもキーワード引数としても値を渡せる。これら 2 つが myfunc2 関数を呼び出すのに必須のパラメーターとなる。そして、それ以上の引数が渡されたときには、位置引数はパラメーター args に、キーワード引数はパラメーター kwargs に渡される。以下に例を示す。

```
myfunc2(1, 2, 3, x=4)
myfunc2(x=3, y=4, b=2, a=1)
```

**myfunc2 関数の呼び出し例**

最初の呼び出し例では、引数「1」と「2」はパラメーター a と b に位置引数として渡される。引数「3」はパラメーター args（タプル）の要素となる。引数「x=4」はパラメーター kwargs（辞書）の要素「{'x': 4}」となる。2 つ目の例では、パラメーター a と b にキーワード引数として値「1」と「2」が渡され、他の 2 つのキーワード引数はいずれもパラメーター kwargs に渡される。

実行結果を以下に示す。

```
In [5]: def myfunc2(a, b, *args, **kwargs):
            print(f'a: {a}, b: {b}, args: {args}, kwargs: {kwargs}')

In [6]: myfunc2(1, 2, 3, x=4)
        myfunc2(x=3, y=4, b=2, a=1)

a: 1, b: 2, args: (3,), kwargs: {'x': 4}
a: 1, b: 2, args: (), kwargs: {'x': 3, 'y': 4}
```

**実行結果**

なお、これらはパラメーターリストの末尾に置く。ただし、可変長位置引数を受け取るパラメーターは、パラメーターリスト中で位置引数の最後に置くことができる。その場合、それ以降のパラメーターはキーワード専用引数として扱われる。

以下に例を示す。

```
def myfunc3(a, b, *args, c, **kwargs):
    print(f'a: {a}, b: {b}, args: {args}, c: {c}, kwargs: {kwargs}')
```

**可変長引数を受け取るパラメーターを関数定義の例（その 2）**

ここでは位置引数を受け取るパラメーターの最後として可変長位置引数を受け取るパラメーター args を置いている。そのため、パラメーター c にはキーワード引数形式でしか値を渡せないことに注意しよう。パラメーター kwargs はそれ以外のキーワード引数を受け取る。呼び出し例を以下に示す。

```
myfunc3(1, 2, 3, 4, c=5, d=6, e=7)
myfunc3(1, 2, 3, 4, d=5)
```

**myfunc3 関数の呼び出し例**

1 つ目の呼び出し例では、最初の 2 つの引数がパラメーター a と b に渡され、残る 2 つの位置引数はパラメーター args の要素となり、キーワード引数「c=5」はパラメーター c に、残る 2 つのキーワード引数はパラメーター kwargs に渡される。

一方、2 つ目の呼び出し例では、キーワード引数として値を渡すしかないパラメーター c に値が渡されないため、例外が発生する。

```
In [7]: def myfunc3(a, b, *args, c, **kwargs):
            print(f'a: {a}, b: {b}, args: {args}, c: {c}, kwargs: {kwargs}')

In [8]: myfunc3(1, 2, 3, 4, c=5, d=6, e=7)
        myfunc3(1, 2, 3, 4, d=5)

        a: 1, b: 2, args: (3, 4), c: 5, kwargs: {'d': 6, 'e': 7}

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-8-9b5e0c1385e8> in <module>
              1 myfunc3(1, 2, 3, 4, c=5, d=6, e=7)
        ----> 2 myfunc3(1, 2, 3, 4, d=5)

        TypeError: myfunc3() missing 1 required keyword-only argument: 'c'
```

**実行結果**

可変長引数を受け取るパラメーターを持たせる際には、それをパラメーターリスト中のどこに置くかは注意しよう。

## ラムダ式

最後にラムダ式の定義についてもまとめておこう。ラムダ式は、1 行で処理が終わる無名の関数を簡単に定義するのに使える。これは、関数の引数として関数を渡す場面などで便利に使える。つまり、関数本体が 1 行で済むような関数を事前に定義することなく、関数が必要な場所ですぐに関数を定義して利用できる。

ラムダ式の構文を以下に示す。

```
lambda パラメーターリスト: 式
```

**ラムダ式の構文**

「パラメーターリスト」にはそのラムダ式が必要とするパラメーターをカンマ区切りで並べる。「式」には、そのラムダ式で実行する処理を単一の式で記述する。書けるのは単一の式のみなので、おのずとラムダ式（が定義する関数）で行える処理は極めて小規模なものになる。

以下に例を示す。

```
mylist = list(range(5))  # [0, 1, 2, 3, 4]
newlist = list(map(lambda x: x ** 2, mylist))
```

**ラムダ式の利用例**

ここでは map 関数の引数としてラムダ式を利用している。このラムダ式「lambda x: x ** 2」では先ほどの「パラメーターリスト」には「x」のみを指定し、「式」の部分には「x ** 2」とそれを 2 乗する式を書いている。この式の評価結果が、ラムダ式の戻り値となる。つまり、これは以下の関数を定義して、map 関数に渡しているのと同じ意味になる。

```
def square(x):
    return x ** 2

mylist = list(range(5))
newlist = list(map(square, mylist))
```

上と同じ意味のコード

ラムダ式を使った方が、コードが簡潔になることが分かる。なお、上記の map 関数呼び出しと、その結果（イテレータ）を list 関数に渡してリストを生成する処理自体はリスト内包表記を使うと次のように書ける。

```
mylist = list(range(5))
newlist = [x ** 2 for x in mylist]
```

リスト内包表記で同じことを行う

ここではラムダ式の使用例として、map 関数を用いたが、実際にはこちらの書き方がよいだろう。

最後にラムダ式を返す関数を定義する例も示しておこう。関数を返す関数の例も並記するので見比べてほしい。

```
def get_incrementer(x):
    return lambda y: x + y

def get_incrementer2(x):
    def incrementer(y):
        return x + y
    return incrementer

one_incrementor = get_incrementer(1)
two_incrementor = get_incrementer2(2)

print(one_incrementor(1))
print(two_incrementor(1))
```

ラムダ式を返す関数と関数を返す関数

本章では関数定義についてまとめた。次章では、文字列やリスト、タプルなど、反復可能オブジェクトの操作についてまとめる。

# [Python チートシート]
# 文字列／リスト／タプル／辞書／集合の操作編

Python が提供する標準的な反復可能オブジェクトとして、文字列、リスト、タプル、辞書、集合の操作をギュギュッとまとめた。

かわさきしんじ，Deep Insider 編集部（2020 年 01 月 21 日）

本章では Python を使う上では避けては通れない、文字列、リスト、タプル、辞書、集合の定義とその操作（演算子、組み込み関数、メソッド）を見ていく（frozenset、バイト列などは省略する）。

## 文字列／リスト／タプル／辞書／集合の定義

文字列（str）、リスト（list）、タプル（tuple）、辞書（dict）、集合（set）には次のような特性がある。

| 特性 | str | list | tuple | dict | set |
|---|---|---|---|---|---|
| 変更可能 | × | ○ | × | ○ | ○ |
| 変更不可能 | ○ | × | ○ | × | × |
| 要素に順序がある | ○ | ○ | ○ | × | × |
| 要素に順序がない | × | × | × | ○ | ○ |
| 要素（キー）の重複 | ○ | ○ | ○ | × | × |
| 要素の種類 | 文字 | 任意 | 任意 | キー／値の組 | 任意 |

文字列／リスト／タプル／辞書／集合の定義

また、これらは組み込み関数もしくはリテラルの形で定義できる。以下に定義例を示す。

```
# 文字列の定義
s = 'foo'  # 文字列リテラルはシングルクオート／ダブルクオートなどで囲む
print('s:', s)  # 'foo'
s = str([0, 1, 2])  # str関数は他のオブジェクトの文字列表現を作成する
print('s:', s)  # '[0, 1, 2]'

# リストの定義
l = ['foo', 'bar']  # リストリテラルは[]で囲む
print('l:', l)  # ['foo', 'bar']
l = list('foo')  # list関数に文字列（反復可能オブジェクト）を渡す
print('l:', l)  # ['f', 'o', 'o']

# タプルの定義
t = (0, 1, 2)  # タプルリテラルは()で囲む
print('t:', t)  # (0, 1, 2)
t = ()  # 空のタプル
print('t:', t)  # ()
t = (0, )  # 要素を1つだけ持つタプル
print('t:', t)  # (0,)
t = tuple(l)  # tuple関数にリスト（反復可能オブジェクト）を渡す
print('t:', t)  # ('f', 'o', 'o')

# 辞書の定義
d = {'a': 'A', 'b': 'B', 'c': 'C'}  # 辞書リテラルは{}内にキー／値の組を書く
print('d:', d)  # {'a': 'A', 'b': 'B', 'c': 'C'}
d = dict(foo='FOO', bar='BAR', baz='BAZ')  # dict関数にキーワード引数を渡す
print('d:', d)  # {'foo': 'FOO', 'bar': 'BAR', 'baz': 'BAZ'}
d = dict([('a', 'A'), ('b', 'B')])  # キーと値のタプルを要素とするリストを渡す
print('d:', d)  # {'a': 'A', 'b': 'B'}

# 集合の定義
s = {0, 1, 2}  # 集合リテラルは{}で囲む
print('s:', s)  # {0, 1, 2}
s = set('foo')  # set関数に文字列（反復可能オブジェクト）を渡す
print('s:', s)  # {'f', 'o'} : 集合は要素の重複を許さない
```

文字列／リスト／タプル／辞書／集合の定義

リスト、辞書、集合については内包表記を使って定義することも可能だ。

```
# リスト内包表記
l = [x for x in range(5)]  # [0, 1, 2, 3, 4] == list(range(5))
l = [x ** 2 for x in l]  # [0, 1, 4, 9, 16]

# 辞書内包表記
d = {x: x * 2 for x in range(3)}  # {0: 0, 1: 2, 2: 4}

# 集合内包表記
s = {x ** 2 for x in range(3)}  # {0, 1, 4}
```

内包表記の例

なお、これらのオブジェクトの詳細については以下を参照されたい。

- 文字列：「Python 入門」の「文字列の基本」「Python の文字列の操作」「文字列の書式指定」
- リスト：「Python 入門」の「リストの基本」「リストの操作」「リストと繰り返し処理」
- タプル：「Python 入門」の「タプル」
- 辞書：「Python 入門」の「辞書」
- 集合：「Python 入門」の「集合」

以下では、これらのオブジェクトに共通する操作、オブジェクトに固有の操作を順に見ていこう。

## 演算子、組み込み関数による操作

以下に文字列（str）、リスト（list）、タプル（tuple）、辞書（dict）、集合（set）に対して可能な演算と、それらを引数に取る組み込み関数を一覧し、それぞれのデータ型でその操作を行えるかを○（可能）と×（不可能）で示す。以下の表の「操作」列では「iter」を反復可能オブジェクトとし、「x」はその要素と同じ型の値、「n」「i」「j」「k」「func」を乗算やインデクシング、スライシング、その他の操作で使用する値とする。なお、「dict」列で「○」となっているものは、辞書の「キー」を対象とすることには注意しよう（例えば、「'x' in {'A': 'a', 'X': 'x'}」は False となるが、これは辞書に「'x'」というキーがないからだ）。一部の操作では、指定可能な引数を省略しているものもある。

| 操作 | 説明 | str | list | tuple | dict | set |
|---|---|---|---|---|---|---|
| x in iter | xが存在するかどうか | ○ | ○ | ○ | ○ | ○ |
| x not in iter | xが存在しないかどうか | ○ | ○ | ○ | ○ | ○ |
| iter1 + iter2 | iter1とiter2を結合した結果を得る | ○ | ○ | ○ | × | × |
| iter1 += iter2 | iter1の末尾にiter2を結合 | × | ○ | × | × | × |
| iter * n、n * iter | iterのn回の繰り返し | ○ | ○ | ○ | × | × |
| iter *= n | iterをn回繰り返したものをiterに代入 | × | ○ | × | × | × |
| iter[i] | iterのi番目の要素 | ○ | ○ | ○ | ○ | × |
| iter[i:j] | iterのi番目からj番目の要素 | ○ | ○ | ○ | × | × |
| iter[i:j:k] | iterのi番目からj番目の要素（k個おきに） | ○ | ○ | ○ | × | × |
| iter[i] = x | iterのi番目の要素にxを代入 | × | ○ | × | ○ | × |
| iter1[i:j] = iter2 | iter1[i:j]をiter2に置き換える | × | ○ | × | × | × |
| iter1[i:j:k] = iter2 | iter[i:j:k]をiter2に置き換える（両者の要素数が同じである必要がある） | × | ○ | × | × | × |
| del iter[i] | iter[i]の要素を削除 | × | ○ | × | ○ | × |
| del iter[i:j] | iter[i:j]の要素を削除 | × | ○ | × | × | × |
| del iter[i:j:k] | iter[i:j:k]の要素を削除 | × | ○ | × | × | × |
| all(iter) | iterの要素が全て真かどうか | ○ | ○ | ○ | ○ | ○ |
| any(iter) | iterのいずれかの要素が真かどうか | ○ | ○ | ○ | ○ | ○ |
| enumerate(iter, i=0) | 「(index, item)」というタプルを返すイテレータを取得 | ○ | ○ | ○ | ○ | ○ |
| filter(func, iter) | iterの要素に対してfuncを実行し、その結果が真となるものだけを返送するイテレータを取得 | ○ | ○ | ○ | ○ | ○ |
| len(iter) | iterの要素数 | ○ | ○ | ○ | ○ | ○ |
| map(func, iter) | iterの要素に対してfuncを実行し、その結果を返送するイテレータを取得 | ○ | ○ | ○ | ○ | ○ |
| max(iter) | iterで最大の要素 | ○ | ○ | ○ | ○ | ○ |
| min(iter) | iterで最小の要素 | ○ | ○ | ○ | ○ | ○ |
| reversed(iter) | 順序を持つiterの要素を逆順に取り出すイテレータを取得 | ○ | ○ | ○ | × | × |
| sorted(iter, key=None) | keyに指定された関数でiterの要素（辞書の場合はキー）を処理したものをキーとして、iterを並べ替えた新しいオブジェクトを取得 | ○ | ○ | ○ | ○ | ○ |
| sum(iter) | iterの要素の和を求める。要素は数値である必要がある | ○ | ○ | ○ | ○ | ○ |
| zip(*iter) | iterに指定された1つ以上の反復可能オブジェクトから、同じインデックスを持つ要素をまとめたタプルを返送するイテレータを取得 | ○ | ○ | ○ | ○ | ○ |

反復可能オブジェクトの操作（演算子、del 文、組み込み関数）

以下に例を示す。

```
# in演算子、not in演算子
print("'y' in 'Python':", 'y' in 'Python')  # True
print("0 not in [1, 2, 3]:", 0 not in [0, 1, 2, 3])  # False

# +演算子
print('(0, 1, 2) + (3, 4):', (0, 1, 2) + (3, 4))  # (0, 1, 2, 3, 4)

# +=演算子
iter1 = list(range(4))
print('iter1:', iter1)  # [0, 1, 2, 3]
iter1 += [4, 5]
print('iter1 += [4, 5]:', iter1)  # [0, 1, 2, 3, 4, 5]

# *演算子
print("'Py' * 2:", 'Py' * 2)  # 'PyPy'

# *=演算子
iter = ['foo', 'bar']
print('iter:', iter)  # ['foo', 'bar']
iter *= 3
print('iter *= 3:', iter)  # ['foo', 'bar', 'foo', 'bar', 'foo', 'bar']

# []演算子
iter = list(range(5))
print('iter:', iter)  # [0, 1, 2, 3, 4]
print('iter[2]:', iter[2])  # 2
print('iter[1:3]:', iter[1:3])  # [1, 2]
print('iter[3:-1]:', iter[3:-1])  # [3]
print('iter[::2]:', iter[::2])  # [0, 2, 4]

# []演算子（代入）
iter = list(range(5))
print('iter:', iter)  # [0, 1, 2, 3, 4]
iter[2] = 100
print('iter[2] = 100:', iter)  # [0, 1, 100, 3, 4]
iter = {'a': 'A', 'b': 'B', 'c': 'C'}
print('iter:', iter)  # {'a': 'A', 'b': 'B', 'c': 'C'}
iter['c'] = 'c'
print("iter['c'] = 'c':", iter)  # {'a': 'A', 'b': 'B', 'c': 'c'}
```

```
iter1 = list(range(5))
print('iter1:', iter1)  # [0, 1, 2, 3, 4]
iter2 = [10, 20]
print('iter2:', iter2)  # [10, 20]
iter1[1:3] = iter2  # インデックス1と2の要素を[10, 20]で置き換える
print('iter1[1:3] = iter2:', iter1)  # [0, 10, 20, 3, 4]

print('iter1:', iter1)  # [0, 10, 20, 30, 4]
iter2 = [100, 200, 400]
print('iter2:', iter2)  # [100, 200, 400]
iter1[0:5:2] = iter2
print('iter1[0:5:2] = iter2:', iter1)  # [100, 10, 200, 3, 400]

# del文
iter = list(range(5))
print('iter:', iter)  # [0, 1, 2, 3, 4]
del iter[3]
print('del iter[3]:', iter)  # [0, 1, 2, 4]
del iter[0:1]  # del iter[0:1]はiter[0]のみを削除
print('del iter[0:1]:', iter)  # [1, 2, 4]
del iter[::2]  # 0番目の要素と2番目の要素を削除
print('del iter[::2]:', iter)  # [2]

# all関数／any関数
iter = list(range(5))
print('iter:', iter)  # [0, 1, 2, 3, 4]
print('all(iter):', all(iter))  # False : iter[0]==0は偽と評価される
print('any(iter):', any(iter))  # True : iter[0]以外は真と評価される

# enumerate関数
iter = 'iter'
for (idx, ch) in enumerate(iter):
    print(f"iter[{idx}] = '{ch}'")  # "iter[0] = 'i'"、"iter[1] = 't'"、……

# filter関数
iter = [2, 7, 8, 10, 1]
print('iter:', iter)
result = list(filter(lambda x: x % 2 == 0, iter))  # 偶数のみからなるリスト
print('list(filter(lambda x: x % 2 == 0, iter)):', result)  # [2, 8, 10]
```

```
# len関数
iter = {chr(x + ord('a')): chr(x + ord('A')) for x in range(26)}
print('iter:', iter)  # {'a': 'A', 'b': 'B', ……, 'y': 'Y', 'z': 'Z'}
print('len(iter):', len(iter))  # 26

# map関数
iter = list('iter')
print('iter:', iter)  # ['i', 't', 'e', 'r']
result = list(map(lambda x: chr(ord(x) - 32), iter))  # 32＝ord('a') - ord('A')
print('list(map(lambda x: chr(ord(x) - 32), iter)):', result)  # ['I', 'T', 'E', 'R']

# max関数／min関数
iter = set(range(10))
print('iter:', iter)  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print('max(iter):', max(iter))  # 9
iter = 'string'
print('iter:', iter)  # 'string'
print('min(iter):', min(iter))  # 'g'

# reversed関数
iter = list(range(5))
print('iter:', iter)  # [0, 1, 2, 3, 4]
print('list(reversed(iter)):', list(reversed(iter)))  # [4, 3, 2, 1, 0]

# sorted関数
iter = 'Python'
print('iter:', iter)  # 'Python'
result = ''.join(sorted(iter))  # ASCII順に並べ替え
print("''.join(sorted(iter)):", result)  # 'Phnoty'
result = ''.join(sorted(iter, key=str.upper))  # 全てを大文字化した値で並べ替え
print("''.join(sorted(iter, key=str.upper)):", result)  # 'hnoPty'

# sum関数
iter = list(range(5))
print('iter:', iter)
print('sum(iter):', sum(iter))  # 10
iter = '012345'
print('iter:', iter)
print('sum([int(x) for x in iter]):', sum([int(x) for x in iter]))  # 15

# zip関数
for (x, y) in zip([0, 1, 2], 'str'):
    print(x, y)  # '0 s'、'1 t'、'2 r'
```

反復可能オブジェクトの操作例



→目次に戻る

## 反復可能オブジェクトのメソッド

反復可能オブジェクトには変更可能（ミュータブル）なものと、変更不可能（イミュータブル）なものがある。また、要素が順序を持つもの（シーケンス）と持たないものがある。こうした特性の違いに応じて、それらに対して呼び出し可能なメソッドが決まる。例えば、文字列とタプルはともに変更不可能な反復可能オブジェクトであるため、呼び出せるのは index メソッドと count メソッドのみとなる（変更可能かつ要素が順序を持つ反復可能オブジェクトはリストのみであることから、以下の表に示すメソッドの多くはリストのみでサポートされている）。また、辞書と集合は clear メソッドなど幾つかのメソッドを提供している。

以下に文字列、リスト、タプル、辞書、集合で呼び出し可能／呼び出し不可能なメソッドを一覧する。以下の表の「メソッド」列では「iter」を反復可能オブジェクトとし、「x」はその要素と同じ型の値、「i」を要素のインデックスを表す値とする。

| メソッド | 説明 | str | list | tuple | dict | set |
|---|---|---|---|---|---|---|
| **iter.index(x, i, j)** | iter（のi番目からj番目）の要素で最初にxが登場する位置 | ○ | ○ | ○ | × | × |
| **iter.count(x)** | iterにxが何個含まれているか | ○ | ○ | ○ | × | × |
| **iter1.append(x)** | iterの末尾にxを追加 | × | ○ | × | × | × |
| **iter.clear()** | iterの内容を全て削除 | × | ○ | × | ○ | ○ |
| **iter.copy()** | iterの浅いコピーを作成 | × | ○ | × | ○ | ○ |
| **iter1.extend(iter2)** | iter1の末尾にiter2を結合 | × | ○ | × | × | × |
| **iter.insert(i, x)** | iterのi番目の位置にxを挿入 | × | ○ | × | × | × |
| **iter.pop(i)** | iterの先頭（またはi番目）の要素を削除して、その値を戻り値とする<br>辞書の場合はiで示されるキーを削除し、その値を戻り値とする<br>集合の場合は引数を取らず、集合の任意の要素を削除して、それを戻り値とする | × | ○ | × | ○ | ○ |
| **iter.remove(x)** | iterから指定された要素xを削除する。戻り値はなし | × | ○ | × | × | ○ |
| **iter.reverse(x)** | iterをインプレースで逆順に並べ替える | × | ○ | × | △ | × |

**反復可能オブジェクトのメソッド**

△：辞書の reversed メソッドは Python 3.8 で追加された。

以下に例を示す。

```
# indexメソッド
iter = 'foobarbaz'
print('iter:', iter)
print("iter.index('a'):", iter.index('a'))  # 4
print("iter.index('a', 5, 8):", iter.index('a', 5, 8))  # 7

# countメソッド
iter = (0, 1, 2) * 2
print('iter:', iter)  # (0, 1, 2, 0, 1, 2)
print('iter.count(0):', iter.count(0))  # 2

# appendメソッド
iter = list(range(5))
print('iter:', iter)  # [0, 1, 2, 3, 4]
iter.append(5)  # 戻り値はないので別に実行しておく
print('iter.append(5):', iter)  # [1, 2, 3, 4, 5]
iter.append([6])
print('iter.append([6]):', iter)  # [1, 2, 3, 4, 5, [6]]

# clearメソッド
iter = {'a': 'A', 'b': 'B'}
print('iter:', iter)  # {'a': 'A', 'b': 'B'}
iter.clear()  # 戻り値はないので別に実行しておく
print('iter.clear():', iter)  # {}

# copyメソッド
iter1 = {0, 1, 2, 3}
print('iter1:', iter1)  # {0, 1, 2, 3}
iter2 = iter1.copy()
print('iter2 = iter1.copy():', iter2)  # {0, 1, 2, 3}
print('iter1 == iter2:', iter1 == iter2)  # True
print('iter1 is iter2:', iter1 is iter2)  # False

iter1 = [[0, 1], [2, 3]]
print('iter1:', iter1)  # [[0, 1], [2, 3]]
iter2 = iter1.copy()  # copyメソッドは浅いコピーを作成する
print('iter2 = iter1.copy():', iter2)  # [[0, 1], [2, 3]]
print('iter1 is iter2:', iter1 is iter2)  # False
print('iter1[0] is iter2[0]:', iter1[0] is iter2[0])  # True
```



→目次に戻る

```
# extendメソッド
iter1 = list(range(5))
iter2 = [5, 6]
print('iter1:', iter1)  # [0, 1, 2, 3, 4]
print('iter2:', iter2)  # [5, 6]
iter1.extend(iter2)  # 戻り値はないので別に実行しておく
print('iter1.extend(iter2):', iter1)  # [0, 1, 2, 3, 4, 5, 6]

# insertメソッド
iter = [0, 1, 3, 4]
print('iter:', iter)  # [0, 1, 3, 4]
iter.insert(2, 20)  # 戻り値はないので別に実行しておく
print('iter.insert(2, 20):', iter)  # [0, 1, 20, 3, 4]

# popメソッド
iter = list(range(5))
print('iter:', iter)  # [0, 1, 2, 3, 4]
print('iter.pop():', iter.pop())  # 4
print('iter:', iter)  # [0, 1, 2, 3]  # 末尾の要素が削除され、取り出された
print('iter.pop(1):', iter.pop(1))  # 1
print('iter:', iter)  # [0, 2, 3] : 指定したインデックス位置の要素が対象

iter = {'a': 'A', 'b': 'B', 'c': 'C'}
print('iter:', iter)  # {'a': 'A', 'b': 'B', 'c': 'C'}
print("iter.pop('c'):", iter.pop('c'))  # 'C'
print('iter:', iter)  # {'a': 'A', 'b': 'B'}
print("iter.pop('d', 'default'):", iter.pop('d', 'default'))  # 'default'

iter = {1, 2, 4, 7, 9}
print('iter:', iter)  # {1, 2, 4, 7, 9}
print('iter.pop():', iter.pop())  # 1

# removeメソッド
iter = [0, 1, 2] * 2
print('iter:', iter)  # [0, 1, 2, 0, 1, 2]
iter.remove(2)  # 戻り値はないので別に実行しておく
print('iter.remove(2):', iter)  # [0, 1, 0, 1, 2]

iter = set(range(5))
print('iter:', iter)  # {0, 1, 2, 3, 4}
iter.remove(2)  # 戻り値はないので別に実行しておく
print('iter.remove(2):', iter)  # {0, 1, 3, 4}
```



→目次に戻る

```
# reverseメソッド
iter = list(range(5))
print('iter:', iter)
iter.reverse()  # 戻り値はないので別に実行しておく
print('iter.reverse():', iter)  # [4, 3, 2, 1, 0]
```

反復可能オブジェクトのメソッドの使用例

以下では、文字列、リストなど、それぞれの反復可能オブジェクトが持つ固有のメソッドについてまとめる。

## リストに固有のメソッド

リストに固有のメソッドを以下に示す（リストは「lst」と表記）。

| メソッド | 説明 |
|---|---|
| lst.sort(key=None, reverse=False) | lstの要素をインプレースでリストの要素を並べ替える |

リストに固有のメソッド

表に示した通り、リストに固有のメソッドは「破壊的な」（リスト自体の要素を書き換える）sort メソッドのみだ。元のリストの順序を維持したまま、並べ替え後のリストがほしいときには組み込みの sorted 関数を使用する。

引数 key は、上で見た sorted 関数と同様に並べ替える際のキーを取得するために使用する。引数 reverse に True を指定すると降順に、False（デフォルト値）を指定すると昇順に並べ替えが行われる。

以下に例を示す。

```
lst = ['foo', 'bar', 'baz']
print('lst:', lst)  # ['foo', 'bar', 'baz']
lst.sort()  # 戻り値はないので別に実行しておく
print('lst.sort():', lst)  # ['bar', 'baz', 'foo']

lst = list('Python')
print('lst:', lst)  # ['P', 'y', 't', 'h', 'o', 'n']
lst.sort(key=str.upper, reverse=True)  # 大文字小文字の区別なしで並べ替え。逆順
print('lst.sort(key=str.upper, reverse=True):', lst)  # ['y', 't', 'P', 'o', 'n', 'h']
```

sort メソッドの使用例

## 辞書に固有のメソッド

辞書に固有のメソッドを以下に示す。以下の表では、辞書オブジェクトを「dct」と表記する。なお、辞書で実装されているメソッドや操作のうち、上で紹介したものは省略している。

| メソッド | 説明 |
|---|---|
| **dict.fromkeys(iter, value)** | dictクラスのクラスメソッド。iterの要素をキーとした新しい辞書を作成する。valueが指定されれば、キーの値はvalueになる。指定しなければ、Noneになる |
| **dct.get(key, default=None)** | keyの値を取得する。keyが辞書になければdefaultの値が返される。dct[key]のようにして値を取得しようとすると、keyがないときにはKeyError例外が発生するが、getメソッドではKeyError例外が発生しない |
| **dct.items()** | (key, value)となるタプルを返送するビューを取得する |
| **dct.keys()** | 辞書のキーを返送するビューを取得する |
| **dct.popitem()** | (key, value)の組を任意（またはPython 3.7以降では辞書にキー／値の組を追加したのとは逆の順）にdctから削除して、それを戻り値とする |
| **dct.setdefault(key, default=None)** | dctに指定したkeyがあれば、その値を戻り値とする。なければ、defaultに指定した値をkeyの値の値として、辞書にエントリを追加する（戻り値はdefaultの値） |
| **dct.update(other)** | otherに指定した辞書、キー／値を要素とする反復可能オブジェクト、キーワード引数などを使って、dctを更新する。既存のキーが指定された場合は上書きされる |
| **dct.values()** | dctの値を返送するビューを取得する |

辞書に固有のメソッド

以下に幾つか例を示す。詳細な説明と例については「Python 入門」の「辞書」を参照されたい。

```
dct = {chr(x+ord('a')): chr(x+ord('A')) for x in range(3)}
print('dct:', dct)  # {'a': 'A', 'b': 'B', 'c': 'C'}

print(dct.get('f'))  # None（辞書にないキーを指定）

for (key, value) in dct.items():
    print(f'{key}: {value}')  # 'a: A'、'b: B'、'c: C'

for key in dct.keys():
    print(dct[key])  # 'A'、'B'、'C'

dct.setdefault('d', 'D')
print('dct:', dct)  # {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}

print(dct.popitem())  # ('d', 'D')
print('dct:', dct)  # {'a': 'A', 'b': 'B', 'c': 'C'}
```

辞書に固有のメソッドの使用例

## 集合に固有のメソッド／演算子

集合に固有のメソッド／演算子を以下に示す。以下の表では、辞書オブジェクトを「st」と表記する。なお、集合で実装されているメソッドや操作のうち、上で紹介したものは省略している。

| メソッド／演算子 | 説明 |
|---|---|
| st.isdisjoint(st2) | stとst2が互いに素なら（重複する要素がなければ）Trueを、そうでなければFalseを返す |
| st.issubset(st2)<br>st <= st2 | stがst2の部分集合もしくは等しい集合であればTrueを、そうでなければFalseを返す |
| st < st2 | stがst2の真部分集合であればTrueを、そうでなければFalseを返す |
| st.issuperset(st2)<br>st >= st2 | stがst2の上位集合もしくは等しい集合であればTrueを、そうでなければFalseを返す |
| st > st2 | stがst2の真上位集合であればTrueを、そうでなければFalseを返す |
| st.union(st2)<br>st \| st2 | stとst2の和を返す |
| st.intersection(st2)<br>st & st2 | stとst2の積を返す |
| st.difference(st2)<br>st - st2 | stとst2の差を返す |
| st.symmetric_difference(st2)<br>st ^ st2 | stとst2の対称差を返す |
| st.update(st2)<br>st \|= st2 | stにst2の和をstの要素とする |
| st.intersection_update(st2)<br>st &= st2 | stとst2の積をstの要素とする |
| st.difference_update(st2)<br>st -= st2 | stとst2の差をstの要素とする |
| st.symmetric_difference_update(st2)<br>st ^= st2 | stとst2の対称差をstの要素とする |
| st.add(x) | stにxを追加する |
| st.remove(x) | stからxを削除する。stにxが含まれていないとKeyError例外が発生する |
| st.discard(x) | stからxを削除する。stにxが含まれていなくてもKeyError例外は発生しない |

集合に固有のメソッドと演算子

以下に幾つか例を示す。詳細な説明と例については「Python 入門」の「集合」を参照されたい。

```
print('{0, 1, 2}.isdisjoint({3, 4}):', {0, 1, 2}.isdisjoint({3, 4}))  # True
print('{0, 1, 2}.union({2, 3}):', {0, 1, 2}.union({2, 3}))  # {0, 1, 2, 3}
print('{0, 1, 2} ^ {2, 3}:', {0, 1, 2} ^ {2, 3})  # {0, 1, 3}

st = {0, 1, 2}
st.intersection_update({1, 2, 3})
print('st.intersection_update({1, 2, 3}):', st)  # {1, 2}
```

集合に固有のメソッドと演算子の使用例

## 文字列に固有のメソッド

文字列に固有のメソッドを抜粋して以下に示す。以下の表では、文字列オブジェクトを「st」と表記する。なお、文字列で実装されているメソッドのうち、上で紹介したものは省略している。また、文字列を返すメソッドは全て、元の文字列を変更した新しい文字列を返送する。簡単なものは例を同時に示す。

| メソッド | 説明 | 例 |
|---|---|---|
| **st.capitalize()** | stの最初の文字を大文字に、残りの文字を小文字にする | 'python'.capitalize()→'Python' |
| **st.center(width, fillchar=' ')** | stを長さwidthで中央寄せされた文字列にする。前後に空きがあればfillcharで埋められる | 'py'.center(6, '=')→'==py==' |
| **st.encode()** | stをバイト列にエンコードする | 'python'.encode()→b'python' |
| **st.endswith(suffix, start, end)** | stがsuffixで終了していればTrueを、そうでなければFalseを返す。startとendは比較の開始位置と終了位置を示す | 'python'.endswith('on')→True<br>'python'.endswith('on', 5)→False |
| **st.expandtabs(tabsize=8)** | st中のタブ文字を空白文字に展開する。tabsizeでタブ位置を指定可能 | 'py¥tthon'.expandtabs(4)→'py thon' |
| **st.find(substr, start, end)** | stからsubstrを検索し、最初に見つかったもののインデックスを返す。見つからなければ-1が戻り値となる。startとendで検索の開始位置と終了位置を指定可能 | 'deep'.find('e')→1<br>'deep'.find('e', 2)→2 |
| **st.format(*args, **kwargs)** | stを書式指定文字列としてargsとkwargsに受け取った値の書式化を行う | 「文字列の書式指定」を参照 |
| **st.index(substr, start, end)** | stからsubstrを検索し、最初に見つかったもののインデックスを返す。見つからなければValueError例外となる。startとendで検索の開始位置と終了位置を指定可能 | 'deep'.index('e')→1<br>'python'.index('a')→ValueError例外 |

| | | |
|---|---|---|
| **st.isalnum()** | stがアルファベットと数字のみで構成されていればTrueを、そうでなければFalseを返す | '0x123'.isalnum()→True<br>'=1234='.isalnum()→False |
| **st.isalpha()** | stがアルファベットのみで構成されていればTrueを、そうでなければFalseを返す | 'foo'.isalpha()→True<br>'123a'.isalpha()→False |
| **st.isascii()** | stがASCIIの範囲の文字だけで構成されていればTrueを、そうでなければFalseを返す（Python 3.7以降） | 'ascii range'.isascii()→True<br>'アスキー'.isascii()→False |
| **st.isdecimal()** | stが1文字以上を含み、それらが全て数字であればTrueを、そうでなければFalseを返す | '1234'.isdecimal()→True<br>'12.34'.isdecimal()→False |
| **st.isdigit()** | stが1文字以上を含み、それらが全て数字またはUNICODEのNumeric_Type=Decimal属性を持つ文字であればTrueを、そうでなければFalseを返す | '¥u2460'.isdigit()→True（'¥u2460'は丸付き数字の1） |
| **st.isidentifier()** | stが識別子（名前）として正しければTrueを、そうでなければFalseを返す | 'a0'.isidentifier()→True<br>'0a'.isidentifier()→False |
| **st.islower()** | stが小文字でのみ構成されていればTrue、そうでなければFalseを返す | 'CAPITAL'.islower()→False<br>'capital'.islower()→True |
| **st.isnumeric()** | stが1文字以上を含み、それらが全て数を表す文字であればTrueを、そうでなければFalseを返す | '二'.isnumeric()→True<br>'円'.isnumeric()→False |
| **st.isprintable()** | stが1文字以上を含み、それらが全て印字可能文字であればTrueを、そうでなければFalseを返す | '¥x00'.isprintable()→False |
| **st.isspace()** | stが1文字以上を含み、それらが全て空白文字であればTrueを、そうでなければFalseを返す | '¥t'.isspace()→True<br>'_'.isspace()→False |
| **st.istitle()** | stがタイトルケース（「Title Case」「Python Cheat Sheet」のように最初が大文字で残りが小文字で構成される1語以上の文字列）であればTrueを、そうでなければFalseを返す | 'Foo Bar'.istitle()→True<br>'FOO Bar'.istitle()→False |
| **st.isupper()** | stが大文字だけで構成されていればTrueを、そうでなければFalseを返す | 'FOO'.isupper()→True<br>'foo'.isupper()→False |
| **st.join(iter)** | stを区切り文字として、反復可能オブジェクトiterの要素を結合する | ','.join(['a', 'b'])→'a,b' |
| **st.ljust(width, fillchar=' ')** | stを長さwidthで左寄せされた文字列にする。空きがあればfillcharで埋められる | 'py'.ljust(6, '=')→'py====' |
| **st.lower()** | stを小文字化する | 'FOO'.lower()→'foo' |
| **st.lstrip(chars=None)** | stの先頭から空白文字もしくはcharsで指定された文字を削除する | ' abc '.lstrip()→'abc '<br>'=-abc-='.lstrip('-=')→'abc-=' |

| | | |
|---|---|---|
| **st.maketrans(x, y, z)** | st.translateメソッドで使用する変換テーブルを作成する（スタティックメソッド） | 以下を参照 |
| **st.partition(s)** | stをsが最初に見つかったところで区切り、(sepより前の部分, sep, sepより後ろの部分)というタプルを返す | 'abcdef'.partition('cd')→('ab', 'cd', 'ef') |
| **st.replace(old, new, count)** | st内のoldをnewでcountで指定した回数だけ置き換える。countを指定しなければ全てのoldがnewに置き換えられる | 'python'.replace('ython', 'erl')→'perl' |
| **st.rfind(substr, start, end)** | st.findメソッドと同様だが、最後に見つかったもののインデックスを返す。見つからなければ-1が返される | 'deep'.rfind('e')→2 |
| **st.rindex(substr, start, end)** | st.indexメソッドと同様だが、最後に見つかったもののインデックスを返す。見つからなければ、ValueError例外が発生する | deep'.rindex('a')→ValueError例外 |
| **st.rjust(width, fillchar=' ')** | stを長さwidthで右寄せされた文字列にする。空きがあればfillcharで埋められる | 'py'.rjust(6, '=')→'====py' |
| **str.rpartition(s)** | stをsが最後に見つかったところで区切り、(sepより前の部分, sep, sepより後ろの部分)というタプルを返す | 'abcabcabc'.rpartition('ca')→('abcab', 'ca', 'bc') |
| **st.rsplit(sep=None, maxsplit=-1)** | sepを区切り文字としてstをmaxsplit回分割したものを要素とするリストを返す（sepの検索は末尾より行う）。maxsplitを指定しなければ全てが分割される | 'abcabcabc'.rsplit('ca', 1)→['abcab', 'bc'] |
| **st.rstrip(chars=None)** | stの先頭から空白文字もしくはcharsで指定された文字を削除する | ' abc '.rstrip()→' abc'<br>'=-abc-='.rstrip('-=')→'=-abc' |
| **st.split(sep=None, maxsplit=-1)** | sepを区切り文字としてstをmaxsplit回分割したものを要素とするリストを返す（sepの検索は先頭より行う）。maxsplitを指定しなければ全てが分割される | 'abcabcabc'.split('ca', 1)→['ab', 'bcabc'] |
| **st.splitlines(keepends=False)** | stをその内部にある改行文字で分割し、それぞれを要素とするリストを返す。keependsにTrueを指定すると要素の末尾に改行文字が含まれるようになり、Falseを指定すると要素の末尾から改行文字が削除される | 'a¥nb'.splitlines()→['a', 'b'] |
| **st.startswith(prefix, start, end)** | stがprefixで始まるならTrueを、そうでなければFalseを返す。startとendで検索範囲を指定可能 | 'python'.startswith('py')→True<br>'python'.startswith('y', 1)→True |
| **st.strip(chars=None)** | stの先頭および末尾から空白文字もしくはcharsで指定された文字を削除する | ' abc '.strip()→'abc'<br>'=-abc-='.strip('-=')→'abc' |

| | | |
|---|---|---|
| **st.swapcase()** | stの大文字小文字を入れ替える | 'hoge'.swapcase()→'HOGE' |
| **st.title()** | st内の単語の先頭を大文字に、それ以降の文字を小文字にする | 'deep insider'.title()→'Deep Insider' |
| **st.translate(tbl)** | st内の文字を変換テーブルtblに従って変換する。変換テーブルはst.maketransメソッドで作成できる | 以下を参照 |
| **st.upper()** | stを大文字化する | 'foo'.upper()→'FOO' |
| **st.zfill(width)** | stを長さwidthの文字列とする。空きがあれば'0'で埋められる | '1234'.zfill(5)→'01234' |

文字列に固有のメソッド

大方のメソッドは例を表中に示したので、ここでは maketrans メソッドと translate メソッドの使用例を示す。

maketrans メソッドには引数を 1 つ指定する場合、2 つ指定する場合、3 つ指定する場合がある。

- 1 つ指定：変換前の文字／変換後の文字を要素とする辞書を渡す
- 2 つ指定：同じ長さの文字列を渡す。第 1 引数が変換前の文字を表し、それが第 2 引数の同じインデックス位置にある文字に変換されるようなテーブルが作られる
- 3 つ指定：第 1 引数と第 2 引数は 2 つ指定の場合と同様。第 3 引数に指定した文字列に含まれる文字は削除されるようなテーブルが作られる

translate メソッドに、maketrans メソッドで作成したテーブルを渡すと、それに従って変換が行われる。

```
# 小文字を大文字に変換するテーブル
d = {chr(x+ord('a')): chr(x+ord('A')) for x in range(26)}
tbl = str.maketrans(d)  # 引数を1つ指定する場合
print(tbl)  # {97: 'A', 98: 'B', 99: 'C', ……, 121: 'Y', 122: 'Z'}
print("'abc'.translate(tbl):", 'abc'.translate(tbl))  # 'ABC'

tbl = str.maketrans('abcd', 'ABCD')  # 引数を2つ指定する場合
print(tbl)  # {97: 65, 98: 66, 99: 67, 100: 68}
print("'abcdefg'.translate(tbl):", 'abcdefg'.translate(tbl))  # 'ABCDefg'

tbl = str.maketrans('abc', 'ABC', 'def')  # 引数を3つ指定する場合
print(tbl)  # {97: 65, 98: 66, 99: 67, 100: None, 101: None, 102: None}
print("'abcdef'.translate(tbl):", 'abcdef'.translate(tbl))  # 'ABC'
```

maketrans メソッドと translate メソッドの使用例

format メソッドについては「Python 入門」の「文字列の書式指定」を参照されたい。

本章では Python プログラムでデータを扱う上では欠かすことのできない文字列、リスト、タプル、辞書、集合の操作方法を圧縮してまとめた。次章ではクラス定義の基本についてまとめる。

# ［Python チートシート］クラス定義編

クラスの定義の基本からさまざまな属性の定義、クラスの継承や多重継承まで、クラスに関わるさまざまな構文をギュッとまとめた。

かわさきしんじ，Deep Insider 編集部（2020 年 01 月 21 日）

本章ではクラスの定義の基本構文、クラス変数／クラスメソッド／スタティックメソッドの定義、プライベートな属性、プロパティ、クラスの継承、多重継承の構文をまとめる。

なお、クラス定義にまつわる詳細については「Python 入門」の以下の記事などを参照されたい。

- クラスの基礎知識
- クラス変数／クラスメソッド／スタティックメソッド
- クラスの継承
- クラスのスコープとプライベートな属性
- 多重継承
- 多重継承と mixin

## クラス定義の基本

クラス定義の基本を以下に示す。

```
class クラス名:
    def __init__(self, パラメーターリスト):
        インスタンスの初期化処理

    def インスタンスメソッド名(self, パラメーターリスト):
        インスタンスメソッドのコード
```

**クラス定義の基本構文**

このようにして定義したクラスは object クラスの派生クラスとなる。

__init__ メソッドは、そのクラスのインスタンス（オブジェクト）を定義する際に、インスタンスが持つ属性（インスタンス変数）の初期化などを行うために使用する。インスタンスメソッドは、インスタンスを介して呼び出し可能な操作である。

これらのメソッドの定義では第 1 パラメーターには暗黙の「self」を置く（self には初期化を行う対象となるインスタンス、またはメソッド呼び出しに使われたインスタンスが渡される）。メソッド内でインスタンスの属性にアクセスする際には「self. 属性」の形でアクセスする。また、（一般には）メソッドを呼び出す際には self には引数の形で値を与えることはない。

クラスを定義すると、「クラスオブジェクト」が作成される。そのクラスのインスタンスを定義するにはクラス名にかっこ「()」を付加して呼び出しを行う（クラスオブジェクトは「呼び出し可能なオブジェクト」であり、かっこを付加することで呼び出し可能。この結果、__init__ メソッドが呼び出されて、インスタンスの初期化が行われる）。

以下にクラス定義の例を示す。

```
class Foo:
    def __init__(self, name):
        self.name = name  # インスタンス変数nameの初期化

    def print_name(self):
        print('name:', self.name)  # 「self.name」として自身の属性にアクセス

f = Foo('foo')  # クラス名にかっこ「()」を付加して呼び出して、インスタンスを定義
f.print_name()  # インスタンスメソッドの呼び出し
Foo.print_name(f)  # 通常はしないが、このような呼び出し方も可能
print(issubclass(Foo, object))  # True
```

クラス定義の例

## クラス変数／クラスメソッド／スタティックメソッドを持つクラス

Python のクラスでは、インスタンス変数／インスタンスメソッド以外に以下を属性として持たせることができる。

- クラス変数
- クラスメソッド
- スタティックメソッド

インスタンス変数とクラス変数の違いを以下の表に示す。

| 変数の種類 | 説明 | クラス外部からのアクセス方法 | クラス内でのアクセス方法 |
|---|---|---|---|
| **インスタンス変数** | インスタンスが持つデータを保存 | インスタンス.インスタンス変数 | self.インスタンス変数 |
| **クラス変数** | クラスが持つデータや複数のインスタンスで共有するデータを保存 | クラス.クラス変数<br>インスタンス.クラス変数 | self.クラス変数（値の書き換えに注意）<br>クラス.クラス変数<br>self.__class__.クラス変数<br>type(self).クラス変数 |

インスタンス変数とクラス変数

クラス変数はクラスが持つ変数で、その値は全てのインスタンスで共有される。クラス変数はクラス定義内に「クラス変数名 = 初期値」として定義できる。クラス変数には「クラス . クラス変数」「インスタンス . クラス変数」などとしてアクセスできる。ただし、「インスタンス . クラス変数 = ……」として代入を行うと、クラス変数ではなく、インスタンス内にクラス変数と同じ名前のインスタンス変数が定義されるので注意が必要だ。

インスタンスメソッド、クラスメソッド、スタティックメソッドの違いを以下に示す。

| メソッドの種類 | 説明 | 第1パラメーター | 呼び出し方 |
|---|---|---|---|
| **インスタンスメソッド** | インスタンスのデータを使って何らかの処理を行う | 呼び出しに使ったインスタンスがselfにセットされる | インスタンス.インスタンスメソッド(引数)<br>クラス.インスタンスメソッド(インスタンス, 引数) |
| **クラスメソッド** | クラスに関連した処理を行う | 呼び出しに使ったクラスがclsにセットされる | クラス.クラスメソッド(引数)<br>インスタンス.クラスメソッド(引数) |
| **スタティックメソッド** | クラスともインスタンスとも関連のない処理を行う | 特定のクラスやインスタンスはセットされない | クラス.スタティックメソッド(引数)<br>インスタンス.スタティックメソッド(引数) |

インスタンスメソッド／クラスメソッド／スタティックメソッド

クラスメソッドは、クラスに結び付けられたメソッドで、呼び出しには「クラス名 . クラスメソッド名 ()」あるいは「インスタンス変数 . クラスメソッド名 ()」の構文を使う。クラスメソッドの定義では第 1 パラメーターには（暗黙の）「cls」を置く（cls にはクラス自体が渡される）。インスタンス変数の属性にはアクセスできず、クラス自身を介した何らかの操作を行いたいときに定義する。クラスメソッドの定義では、一般にメソッド定義を「@classmethod」デコレーターで修飾する。

スタティックメソッドは、インスタンスともクラスとも結び付いていないメソッド。スタティックメソッドは「クラスを名前空間として、そこで定義された関数（メソッド）」と考えられる。その呼び出しは「クラス名 . スタティックメソッド名 ()」または「インスタンス変数 . スタティックメソッド名 ()」という形式で行う。インスタンスメソッドやクラスメソッドとは異なり、メソッド定義では暗黙の第 1 パラメーターを持たない。スタティックメソッドの定義では、一般にメソッド定義を「@staticmethod」デコレーターで修飾する。

これらをまとめると次のようになる。

```
class クラス名(基底クラス):
    クラス変数名 = 初期値

    def __init__(self, パラメーターリスト):
        インスタンスの初期化処理

    def インスタンスメソッド名(self, パラメーターリスト):
        インスタンスメソッドのコード

    @classmethod  # クラスメソッドの定義
    def クラスメソッド名(cls, パラメーターリスト):
        クラスメソッドのコード

    @staticmethod  # スタティックメソッドの定義
    def スタティックメソッド名(パラメーターリスト):
        スタティックメソッドのコード
```

**クラス変数、クラスメソッド、スタティックメソッドを含んだクラス定義の構文**

以下に例を示す。

```
class Foo:
    clsvar = 0  # 今までに何個インスタンスが作成されたかを記録

    def __init__(self, name=''):
        Foo.clsvar += 1  # クラス変数の値をインクリメント
        self.name = name

    def instance_method(self):
        print('in instance_method')
        print('cls var:', self.clsvar)  # self.clsvarとしてクラス変数を読み出し
        print('instance var:', self.name)

    @classmethod
    def cls_method(cls):
        print('in cls_method')
        print('cls var:', cls.clsvar)

    @staticmethod
    def static_method():
        print('in static_method')
        print('cls var:', Foo.clsvar)

f = Foo('foo')
f.instance_method()  # インスタンスメソッド呼び出しにはインスタンスを使う
f.cls_method()  # クラスメソッド呼び出しにはインスタンスも使える
Foo.cls_method()  # クラスメソッド呼び出しにはクラスも使える
Foo.static_method()  # スタティックメソッド呼び出しにはクラスしか使えない
print()
b = Foo('bar')
b.instance_method()
print('Foo.clsvar:', b.clsvar)
b.clsvar += 1  # インスタンスbにインスタンス変数clsvarを定義してしまう
print('b.clsbar:', b.clsvar)
print('Foo.clsbar:', Foo.clsvar)
```

**クラス変数、クラスメソッド、スタティックメソッドを含んだクラス定義の例**

この例では、instance_method メソッドではクラス変数の値を読み取るのに「self.clsvar」としている。一方、__init__ メソッドではクラス変数の値を変更するのに「Foo.clsvar += 1」としている。読み取りに関しては「self.clsvar」で構わないが、「self.clsvar」に代入をすると、それを行ったインスタンス内にクラス変数と同じ名前のインスタンス変数を作ることになるので、__init__ メソッドでは「Foo.clsvar += 1」としてクラス変数の値を変更している。

また、cls_method メソッドでは「cls.clsvar」としてクラス変数にアクセスしている。これによりクラス変数にアクセスできる。なお、クラスメソッドは特定のインスタンスとは関連付けられておらず、暗黙の第 1 パラメーター cls にはクラス自体が渡される。

static_method メソッドは暗黙の第 1 パラメーターに特定のインスタンスやクラスが渡されることはないので、その内部でクラス変数にアクセスするには「クラス名 . クラス変数名」とするしかない。そのため、上の例では「Foo.clsvar」としてアクセスをしている。

このクラスのインスタンスを定義して、各メソッドを呼び出しているコードの下の方では、「b.clsvar += 1」としているが、これによりインスタンス b に「clsvar」という「インスタンス変数」が作成されてしまう。そのため、「b.clsvar」と「Foo.clsvar」ではその値が異なるようになる（興味のある方はコードを実行して確認してほしい）。

## プライベートな属性

外部に公開したくない属性（インスタンス変数、メソッド）をクラスが持つ場合、それらの名前はアンダースコア「_」で始めるようにする。これはその属性が「プライベート」であることを意味する命名規約であり、それを使わないようにすることはあくまでも「紳士協定」である。

以下に例を示す。

```
class Foo:
    def __init__(self, name):
        self._name = name  # _nameはプライベートな属性

    def get_name(self):  # _nameを読み取るためのメソッド
        return self._name

    def set_name(self, new_name):  # _nameを書き換えるためのメソッド
        self._name = new_name

f = Foo('insider.net')
print(f.get_name())  # name属性をメソッド経由で取得
f.set_name('deep insider')
print(f._name)  # 1つのアンダースコアは「紳士協定」なので、直接アクセスも可能
```

プライベートな属性の例

最後の行で「f._name」としているように、アンダースコア「_」で始まる属性にも直接アクセス可能である。

あるいは、2 つのアンダースコア「__」で始めることで、さらに強い効力を持たせることもできる。2 つのアンダースコアで属性の名前を始めると、その名前で直接その属性にアクセスできなくなる（「_ クラス名 __ 属性名」のような名前で外部からは見えるようになる。これを「名前マングリング」あるいは単に「マングリング」と呼ぶ）。

以下に例を示す。

```
class Foo:
    def __init__(self, name):
        self.__name = name  # __nameはプライベートな属性

    def get_name(self):  # __nameを読み取るためのメソッド
        return self.__name

    def set_name(self, new_name):  # __nameを書き換えるためのメソッド
        self.__name = new_name

f = Foo('insider.net')
print(f.get_name())  # name属性をメソッド経由で取得
f.set_name('deep insider')
print(f._Foo__name)  # 無理やりにアクセスは可能
print(f._name)  # AttributeError例外
```

アンダースコア 2 つで始めるより強制度の高いプライベートな属性の例

この場合も、「f._Foo__name」とすれば、クラス内部で「__name」として定義されているインスタンス変数にアクセスは可能だが、以前のように「__name」で外部からはアクセスできなくなる。

## プロパティ

プライベートな属性と、それらにアクセスするためのメソッド（インタフェース）を用意することはよくある。それらは上のような方法でも実現可能だが、Python では「プロパティ」と呼ばれる機構を用いて、これを実現できる。

プロパティを設定するには組み込みの property 関数を使用するか、@property デコレーター（セッタ、ゲッタなどのデコレーター指定）を使用する。

property 関数の構文を以下に示す。

```
property(fget=None, fset=None, fdel=None, doc=None)
```

property 関数

fget にはプロパティの値を読み出すためのメソッドを、fset にはプロパティの値を設定するためのメソッドを、fdel にはプロパティの値を削除するためのメソッドを、doc にはプロパティのヘルプ文字列（docstring）を指定する。

以下に例を示す。

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

    def get_prop(self):
        return self.__name

    def set_prop(self, new_name):
        if not isinstance(new_name, str):
            raise TypeError('name attrib is str')
        self.__name = new_name

    name = property(get_prop, set_prop)

f = Foo('insider.net')
print(f.name)  # 'insider.net'
f.name = 'deep insider'
print(f.name)  # 'deep insider'
f.name = 0  # TypeError例外
```

property 関数の使用例

この例では、__init__ メソッドでプライベートなインスタンス変数 __name の値を初期化して、その後は get_prop メソッドと set_prop メソッドを介して、その値をやりとりするようなコードになっている。しかし、クラス定義の最後では property 関数を使用して、プロパティの値の読み取りは get_prop メソッドで行い、書き込みは set_prop メソッドで行うように指定し、その戻り値を name 属性に代入している。これにより、「インスタンス変数 name を使ってその値を読み書きする」体裁で、これらのメソッドを使った値の読み書きを行えるようになる。

@property デコレーターを使用する例を以下に示す。

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

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, new_name):
        if not isinstance(new_name, str):
            raise TypeError('name attrib is str')
        self.__name = new_name

f = Foo('insider.net')
print(f.name)  # 'insider.net'
f.name = 'deep insider'
print(f.name)  # 'deep insider'
f.name = 0  # TypeError例外
```

@property デコレーターの使用例

@property デコレーターを使用する場合には、@property デコレーターで修飾したメソッドが値を読み取るために使われる（ゲッタ）。そのメソッドと同じ名前のメソッドを定義して、「@ メソッド名 .setter」デコレーター（この場合は「@name.setter」デコレーターとなる）で修飾することで、それがプロパティの値を設定するために使われる（セッタ）。「@ メソッド名 .deleter」デコレーターで修飾したメソッドがあれば、それはプロパティの値を削除するために使われる。

## クラスの継承

クラスを継承するには、class 文で基底クラスを指定する。その基本構文は次のようになる。

```
class 派生クラス名(基底クラス):
    def __init__(self, パラメーターリスト):
        super().__init__(基底クラスの初期化に必要な引数)
        派生クラスのインスタンスの初期化処理

    def インスタンスメソッド名(self, パラメーターリスト):
        インスタンスメソッドのコード
```

クラス継承の基本構文

このとき、派生クラスでは基底クラスのインスタンスの初期化も忘れずに行う必要がある。一般には組み込みの super 関数を使用して、基底クラスへの参照を受け取り、その __init__ メソッドを明示的に呼び出す。

基底クラスで既に定義されているメソッドと同名のメソッドを派生クラスで定義（オーバーライド）するときには、基底クラスの同名メソッドを呼び出す必要があれば、上と同様に組み込みの super 関数を使う。

以下に例を示す。

```
class Base:  # 基底クラスの定義
    def __init__(self, base_val):
        self.base_val = base_val

    def print_value(self):
        print('base_val:', self.base_val)

    def base_only(self):  # 基底クラスでのみ定義されている
        print('defined in Base class')

class Derived(Base):
    def __init__(self, base_val, derived_val):
        super().__init__(base_val)
        self.derived_val = derived_val

    def print_value(self):  # 基底クラスのメソッドをオーバーライド
        super().print_value()  # super関数経由で基底クラスの同名メソッド呼び出し
        print('derived_val:', self.derived_val)

    def derived_only(self):  # 派生クラス独自のメソッド
        self.base_only()  # self経由で基底クラスのメソッド呼び出し
        print('defined in Derived class')

b = Base(100)
b.print_value()
d = Derived(100, 200)
d.print_value()
d.base_only()
d.derived_only()
```

クラス継承の例

## 多重継承

複数のクラスを基底クラスとして、派生クラスを定義するには class 文で基底クラスを複数指定する。

```
class 派生クラス名(基底クラス1, 基底クラス2, ……)
    # ……
```

多重継承

以下に例を示す。

```
class A:
    def hello(self):
        print('Hello from A')

class B:
    def hello(self):
        print('Hello from B')

class C(A, B):
    pass

class D(B, A):
    pass
```

多重継承の例

この例では、クラス C とクラス D はともにクラス A とクラス B を基底クラスとして多重継承を行っている。ただし、基底クラスの指定の順番が異なっている。このときに、クラス C とクラス D のインスタンスで、hello メソッドを呼び出すと、実際に呼び出されるメソッドが異なるものになる。

```
c = C()
d = D()

c.hello()  # 'Hello from A'
d.hello()  # 'Hello from B'
```

クラス C とクラス D では呼び出される hello メソッドが異なる

多重継承時に、基底クラスのどのメソッドが呼び出されるかは MRO（Method Resolution Order：メソッド解決順序）を調べることで分かる。MRO は各クラスの __mro__ 属性の値を調べるか、mro メソッドを呼び出すことで調べられる。なお、MRO 自体は多重継承ではない場合（単一継承の場合）でも存在するが、そのときには継承階層が一直線になるのでメソッド呼び出しやインスタンス変数のアクセスはシンプルなものになる。

```
print(C.__mro__)  # (<class '__main__.C'>, <class '__main__.A'>, <class
'__main__.B'>, <class 'object'>)
print(D.mro())  # [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>,
<class 'object'>]
```

クラス C とクラス D の MRO を調べる

この例では、クラス C ではメソッドを呼び出す際には「クラス C で定義されているメソッド→クラス A で定義されているメソッド→クラス B で定義されているメソッド→ object クラスで定義されているメソッド」という順番で検索が行われ、呼び出そうとしたメソッドの名前が見つかったところで、そのメソッドが呼び出される。クラス D の場合は「D → B → A → object」の順で検索が行われる。そのため、呼び出されるメソッドが異なることになる。

__init__ メソッドの呼び出し時に組み込みの super 関数を使って基底クラスのインスタンスの初期化を行おうとしたときにも、MRO に沿って順番にメソッドが検索されるので、多重継承しているときにインスタンスの初期化を行う際には、このことに注意して全てのインスタンス変数を初期化できるようにする必要がある。

本章ではクラス定義にまつわる要素をまとめた。次章ではファイル操作についてまとめる。

# ［Python チートシート］ファイル操作編

ファイルのオープンとクローズ、with 文を使った書き方からテキストファイルやバイナリファイルの読み書き、struct モジュールまでをギュッとまとめた。

かわさきしんじ，Deep Insider 編集部（2020 年 02 月 04 日）

ファイルを扱う上での基本となる組み込みの open 関数、with 文との組み合わせ、テキストファイルの読み書き、バイナリファイルの読み書きについて見ていこう。

なお、ファイルの操作については以下の記事も参考にしてほしい。

- ファイル操作の基本
- バイナリファイルの操作

## ファイルのオープンとクローズ

ファイルは組み込みの open 関数を用いてオープンし、それに対して読み込み、書き込みを行った後、クローズするのが通常のファイル操作の手順となる。その流れを以下にまとめる。

```
file = open(ファイル名, モード, その他の引数)  # ファイルのオープン
content = file.read()  # 読み込みの例
file.write('foo')  # 書き込みの例
file.close()  # ファイルのクローズ
```

ファイル操作の流れ

あるいは、with 文を使用することで、ファイルのクローズを自動的に行うことも可能だ。

```
with open(ファイル名, モード, その他の引数) as file:  # ファイルのオープン
    content = file.read()  # 読み込みの例
    file.write('foo')  # 書き込みの例。ブロック終了後にファイルはクローズされる
```

with 文を利用したファイル操作の流れ

with 文でファイルを扱うときには、そのブロックが終了すると、ファイルが自動的にクローズされる。with 文はファイル操作中に例外が発生しても、そのクローズが確実に行えるので、こちらの使い方が推奨されている。

ファイルのオープンに使用する組み込みの open 関数の構文は次のようになっている。

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

open 関数

それぞれのパラメーターの意味は以下の通り。

- file：ファイル名またはファイル記述子を与える
- mode：ファイルをオープンするモード。デフォルト引数値は 'r' で、これはファイルをテキストモードで読み込み用にオープンすることを意味する（'rt' と同じ）。詳細は後述
- buffering：バッファリング方法の指定。指定可能な値は 0（バッファリングを行わない）、1（1 は行単位のバッファリングを行う。テキストファイルでのみ有効）、1 以上の整数（バッファサイズをバイト単位で指定）となっている
- encoding：テキストファイルの読み書きで使用するエンコーディングを指定
- errors：テキストファイルの読み書きでエンコード／デコードに失敗したときの対応を指定。指定可能な値は 'strict'（ValueError 例外を発生）、'ignore'（無視）など
- newline：テキストファイルの読み書きにおけるユニバーサル改行モードの動作を制御する。詳細は newline パラメーターの説明を参照のこと
- closefd：file パラメーターにファイル記述子が与えられた場合に、ファイルクローズ時にそのファイル記述子（が参照するファイル）もクローズするかどうかを指定する。True を指定すると元のファイル記述子もクローズされ、False を指定すると元のファイル記述子はオープンされたままとなる
- opener：ファイルのオープンに利用する呼び出し可能オブジェクトを指定する

mode パラメーターには以下を指定できる。

- 'r'：読み込み用にオープン
- 'w'：書き込み用にオープン
- 'x'：ファイルを新規に作成して、書き込み用にオープン。指定したファイルが既にあればエラーとなる
- 'a'：追加書き込み用にオープン
- 't'：テキストファイルをオープン（テキストモード）
- 'b'：バイナリファイルをオープン（バイナリモード）
- '+'：'r' ／ 'w' ／ 'a' ／ 'x' に付加して、ファイルを更新用にオープン（読み込み／書き込み両用でオープン）

'+' は 'r'、'w'、'a'、'x' と一緒に指定して、そのファイルを読み書き両用にオープンする。

- 'r+'：既存ファイルの先頭をファイル位置とする。指定した名前のファイルがなければエラーとなる
- 'w+'：ファイルを新規に作成するか、既存のファイルをオープンして、その内容を全て削除する。ファイルの先頭がファイル位置になる
- 'a+'：ファイルを新規に作成するか、既存のファイルをオープンして、末尾をファイル位置とする（新規ファイルの場合はファイル末尾＝ファイル先頭となる）
- 'x+'：ファイルを新規作成して、そのファイルを読み書き両用にオープンする

## テキストファイルの読み書き

open 関数で mode パラメーターに 'rt'（または単に 'r'）を指定するとテキストファイルを読み込み用にオープンできる。mode パラメーターに 'wt' などを指定すると、テキストファイルを書き込み用にオープンできる。Python のテキストファイルは「文字列の読み書き」を行う。整数など、文字列以外のものをファイルに書き込むには書式化して、文字列に埋め込む必要があるので注意しよう。

テキストファイルを読み込むメソッドとしては以下がある。

- read メソッド：ファイルの内容を全て読み込む。戻り値は 1 つの文字列
- readline メソッド：ファイルから 1 行を読み込む。戻り値は 1 つの文字列
- readlines メソッド：ファイルの内容を全て読み込む。戻り値は、各行を要素とするリスト

これらのメソッドはファイルの末尾まで読み込みが終わると、その後に呼び出しを行うと空文字列を返す。空文字列は「偽」として扱われるので、これを使ってループの繰り返しを制御できる（ファイル中の空行には改行文字のみが含まれるので、空文字列とはならない）。また、open 関数の戻り値であるファイルオブジェクト自身はその各行を返送する反復可能オブジェクトとしても使用できる。

テキストファイルに書き込みを行うメソッドとしては以下がある。

- write メソッド：引数に渡した「文字列」をファイルへ書き込む。戻り値は書き込んだ文字数
- writelines メソッド：引数に渡した反復可能オブジェクトの内容をファイルへ書き込む。戻り値はなし。各要素の末尾に改行文字が追記されることはないので、改行したいときには末尾に改行を自分で追加する必要がある

これらのメソッドでは、size パラメーター（読み込む文字数）や hint パラメーター（読み込む行数。readlines メソッドの場合）を指定できる。

mode パラメーターに 'wt' を指定した場合を例として、これらのメソッドの使い方を以下に示す。

```
with open('foo.txt', 'wt') as file:  # 'foo.txt'ファイルを書き込み用にオープン
    mylist = ['foo', '', 'bar']
    for num, item in enumerate(mylist):
        file.write(f'{num}: {item}¥n')  # f文字列に行カウントを埋め込み
    # writelinesメソッドでは各要素の末尾に改行文字が追加されないので自分で追加
    file.writelines([item.upper() + '¥n' for item in mylist])

file = open('foo.txt')
print(file.read())  # ファイルの内容を全て読み込む

file.seek(0)  # seekメソッドに0を渡してファイル先頭をファイル位置にする
line = file.readline()  # ファイルの内容を1行ずつ読み込む
while line:  # 変数lineの内容が空文字列となるまで（ファイル末尾となるまで）
    print(line, end='')  # 各要素には改行文字が含まれている
    line = file.readline()  # 次の行を読み込む

file.seek(0)
lines = file.readlines()  # 全ての行を読み込む
for line in lines:  # その内容をfor文でループ
    print(line, end='')

file.seek(0)
for line in file:  # ファイルオブジェクトを使って繰り返し
    print(line, end='')

file.close()  # 使い終わったらファイルを閉じる
```

テキストファイルの読み書きの例

上の例では、ファイルオブジェクトの seek メソッドを用いて、各行の表示が終わった後にもう一度ファイル先頭にファイル位置を移動するようにしている。ファイルオブジェクトが持つその他のメソッドについては Python のドキュメント「io --- ストリームを扱うコアツール」にある「TextIOBase」などを参照のこと。例えば、現在のファイル位置を知る「tell メソッド」などがある。

## バイナリファイルの読み書き

バイナリファイルの読み書きを行うには、open 関数の mode パラメーターに 'b' を付加する。'rb' ならバイナリファイルの読み込みを、'wb' ならバイナリファイルへの書き込みを意味する。バイナリファイルの内容は bytes 型のデータとしてやりとりをする。

バイナリファイルの読み込みを行うメソッドには以下がある。

- read メソッド：バイナリファイルの内容を全て読み込む。size パラメーターに値を指定した場合、最大で指定したバイト数を読み込む
- readall メソッド：バイナリファイルの内容を全て読み込む
- readinto メソッド：引数に渡した bytearray オブジェクトに、読み出した内容を書き込む

バイナリファイルへの書き込みを行うメソッドには以下がある。

- write メソッド：バイナリファイルにバイトオブジェクトを書き込む

読み込んだデータは bytes 型（または bytearray 型）のオブジェクトなので、それらを使用する際には何らかの型のデータへと変換する必要がある。例えば、文字列なら decode メソッドでそれらを文字列にデコードできる。逆に書き込みを行う際には、元のデータを bytes 型のデータに変換する必要もある。例えば、文字列であれば、encode メソッドで bytes 型のデータに変換できる。整数なら to_bytes メソッド、from_bytes メソッドで bytes 型の値との変換が行える。

以下にバイナリファイルの読み書きの例を示す。

```
with open('foo.bin', 'wb') as file:
    mystr = 'Hello Python'
    size = file.write(mystr.encode())  # 文字列をエンコードして書き込み
    mynumbers = [x.to_bytes(2, 'little') for x in [0, 1, 2]]  # bytes型に変換
    for item in mynumbers:
        file.write(item)  # bytes型に変換された整数値を書き込み

file = open('foo.bin', 'rb')
tmp = file.read(size)
print(tmp.decode())
tmp = file.read(2)  # 2バイトを読み込み
while tmp:
    print(int.from_bytes(tmp, 'little'))
    tmp = file.read(2)

file.close()
```

バイナリファイルの読み書きの例

ここではファイルをバイナリ書き込みモードでオープンした後、文字列 'Hello Python' を encode メソッドでエンコードしたものを write メソッドで書き込んでいる。次に、リスト [0, 1, 2] の各要素を int 型の to_bytes メソッドで bytes オブジェクトに変換し、それらをやはり write メソッドで書き込んでいる。

to_bytes メソッドを呼び出す際には第 1 引数に bytes オブジェクトに変換後の bytes オブジェクトのバイトサイズを（ここでは 2 バイトのデータに変換）、第 2 引数にバイトオーダー（bytes オブジェクトに変換された整数の各バイトの並び順）を指定する。ここでは 'little'（リトルエンディアンと呼ばれるバイトオーダー）を指定している。書き込んだデータを読み込んで復元する際にはバイトオーダーが一致している必要がある点には注意が必要だ。

バイナリファイルからの読み込みでは、文字列の書き込み時に得た write メソッドの戻り値（書き込んだバイト数）を使って、バイナリファイルから文字列に相当するバイトデータを読み込み、それを decode メソッドで復元している。その後は、while 文を使って 2 バイトずつ読み込んで、今度は int 型の from_bytes クラスメソッドを使って、読み込んだバイトデータを整数に変換している。

## struct モジュール

バイナリファイルの読み書きでは、このように書き込みを行うデータを bytes 型に変換し、読み込んだバイトデータはもともとのデータ型に戻す必要がある（float 型にはこうした処理をサポートするメソッドはない）。これを簡単に行うために struct モジュールが用意されている。

struct モジュールでは書式文字列と呼ばれる文字列を使って、複数のデータを単一のバイトオブジェクトに変換したり、逆にバイトオブジェクトから各データを復元したりできる。

書式文字列の先頭ではバイトオーダーの指定を行う。このときには、主に以下の 2 つが使える。

- <：リトルエンディアン
- >：ビッグエンディアン

また、その後には以下に示す書式指定文字を用いて、バイトオブジェクトに含まれるデータの型を指定する。

| 書式指定文字 | 説明 | サイズ（バイト数） |
|---|---|---|
| c | 文字 | 1 |
| b | 符号付き整数 | 1 |
| B | 符号なし整数 | 1 |
| h | 符号付き整数 | 2 |
| H | 符号なし整数 | 2 |
| i | 符号付き整数 | 4 |
| I | 符号なし整数 | 4 |
| l | 符号付き整数 | 4 |
| L | 符号なし整数 | 4 |
| f | 浮動小数点数 | 4 |
| d | 浮動小数点数 | 8 |
| s | 文字列 | - |

struct モジュールで指定可能な書式指定文字（一部）

同じ型のデータが続くときには、その前に繰り返しの回数を指定できる。例えば、「4i」は「4 個の符号付き整数」つまり「iiii」と同じものと解釈される。「s」の前に整数値を指定した場合、それはその文字列の長さを意味する。「12s」なら「12 文字の文字列」となる。

例えば、先ほどの文字列 'Hello Python' と [0, 1, 2] の各要素をバイトオブジェクトへ変換するコードを以下に示す。

```
import struct

mystr = 'Hello Python'
mylist = [0, 1, 2]
packstr = f'<{len(mystr)}s3h'  # 12文字の文字列と3つの符号付き整数を意味する
bobj = struct.pack(packstr, mystr.encode(), mylist[0], mylist[1], mylist[2])
print(bobj)  # b'Hello Python¥x00¥x00¥x01¥x00¥x02¥x00'
```

struct モジュールを使用した文字列と整数値のバイトオブジェクトへのパックの例

ここでは文字列 packstr の内容は「<12s3h」となる（「len('Hello Python')」の戻り値は 12）。これは「リトルエンディアンで、12 文字の文字列、3 つの符号付き整数」を表している。

バイト型への変換には、上に示した通り、pack 関数を使用する。pack 関数の呼び出し時には、第 1 引数に書式文字列を（ここでは packstr）、その後に続けてバイトオブジェクトに変換したいオブジェクトを並べていく。このとき、文字列は encode メソッドであらかじめバイトオブジェクトに変換しておく必要がある。

逆に、バイトオブジェクトからデータを復元するには unpack 関数を使用する。unpack 関数の呼び出し時には第 1 引数に書式文字列を、第 2 引数に復元したいバイトオブジェクトを渡す。復元されたデータはタプルの要素として返される。以下に例を示す。

```
restored = struct.unpack(packstr, bobj)
print(restored)  # (b'Hello Python', 0, 1, 2)
```

struct モジュールを使用したバイトオブジェクトの復元の例

注意点としては、unpack 関数で取り出した文字列に対応するデータはまだバイトオブジェクトのままなので、decode メソッドで文字列に復元する必要がある点だ。

このことを利用して、バイナリファイルとのやりとりを簡単に行える。

```
mystr = 'Hello Python'
mylist = [0, 1, 2]
packstr = f'<{len(mystr)}s3h'  # 12文字の文字列と3つの符号付き整数を意味する
bobj = struct.pack(packstr, mystr.encode(), mylist[0], mylist[1], mylist[2])

with open('foo.bin', 'wb') as file:
    file.write(bobj)

with open('foo.bin', 'rb') as file:
    restored = struct.unpack(packstr, file.read())
    print(restored)  # (b'Hello Python', 0, 1, 2)
```

struct モジュールを利用して、バイナリファイルの読み書きを行う例

本章ではファイルの読み書きをまとめた。次章ではこれまでにまとめていない要素のうち、重要なものを幾つか取り上げることにする。

# ［Python チートシート］モジュール／例外編

モジュールやパッケージのインポート、それらの作成方法、例外の捕捉と送出の基本についてギュッとまとめた。

かわさきしんじ，Deep Insider 編集部（2020 年 02 月 10 日）

本章ではこれまでに取り上げてこなかった要素のうち、モジュールとパッケージ、例外についてまとめていく。

なお、モジュールやパッケージの詳細については「Python 入門」の以下の記事を参考にしてほしい。

- 「モジュールの使い方」
- 「モジュールの作り方」
- 「パッケージ」

例外については同じく「Python 入門」の以下を参考にしてほしい。

- 「例外と例外処理の基礎」
- 「例外の送出と例外クラス」

## モジュールとパッケージのインポート

モジュールやパッケージをインポートするには import 文を使用する。その主な構文を以下に示す。なお、以下ではモジュール名やパッケージ名を記述する際に単に「モジュール名」とだけ示すことがある。パッケージ内のサブパッケージなども含める際には「パッケージ名」と書くこともある。

```
# モジュール／パッケージをインポート
import モジュール名
import パッケージ名.サブパッケージ名.モジュール名

# インポートしたモジュールを別名で使用
import モジュール名 as 別名

# モジュール／パッケージから特定のものだけをインポート
from モジュール名 import 識別子（変数、関数、クラス、モジュールなど）

# インポートしたものを別名で使用
from モジュール名 import 識別子 as 別名

# モジュール／パッケージから全てをインポート
from モジュール名 import *
```

モジュールやパッケージのインポート

以下に例を示す。

```
# モジュール／パッケージをインポート
import sys  # sysモジュールのインポート
import os.path  # osモジュールに含まれるpathモジュールをインポート

# インポートしたモジュールを別名で使用
import numpy as np  # numpyモジュールをnpとしてインポート

# モジュール／パッケージから特定のものだけをインポート
from random import randint  # randomモジュールのranint関数をインポート

# インポートしたものを別名で使用
from random import randint as rand # random.randint関数をrandとしてインポート

# モジュール／パッケージから全てをインポート
from fractions import *  # fractionsモジュールから全てをインポート
```

**モジュールやパッケージのインポートの例**

モジュール（やパッケージ）をインポートした場合、そこで定義されている変数や関数、クラスには「モジュール名 . 関数名」「モジュール名 . クラス名」のようにしてアクセスする必要がある。以下に例を示す。

```
import sys  # sysモジュールをインポート
print(sys.version_info)  # sysモジュールのversion_info属性を使用

from sys import version_info  # sysモジュールからversion_info属性をインポート
print(version_info)  # これなら「sys.」を省略して使用できる
```

**sys モジュールの version 属性の値を表示（実行環境の Python のバージョンを調べる）**

「from モジュール名 import *」形式では、そのモジュール内で定義されている全ての属性をインポートされる。ただし、この形式だと、現在の名前空間にプログラマーが関知していない多くのものを追加することになり、意図しないうちにそれらが上書きされて、プログラマーが意図したオブジェクト以外のオブジェクトを参照することになったり、どのモジュールで定義されているものかがコードを読む側にハッキリとしなかったりすることがある。そのため、この形式でのインポートは一般には推奨はされていない。

## モジュールでの名前の公開（エクスポート）

自分でモジュールを作成する際には、基本的にはそこで定義されているもの全てが、他のモジュールに対して公開される。外部に公開したくないものを制御するには以下の 2 つの方法がある。

- 公開したくないものの名前をアンダースコア「_」で始め、それが内部使用目的であることを示す
- __all__ 変数で外部に公開するものの名前を明記する。__all__ 変数には公開するものの名前（文字列）をリストに列挙する

これらはあくまでも「from モジュール名 import *」形式でインポートを行った際に、特定の名前がインポートされるかどうかを制御するものであり、「from モジュール名 import 特定の名前」としてインポートすることを妨げるものではないことには注意。

公開したくないものを制御する例を以下に示す。

```
def foo():  # 公開する
    print('foo')

def _bar():  # 名前をアンダースコア「_」で始める
    print('bar')

def baz():  # __all__変数で公開しないことになっている
    print('baz')

__all__ = ['foo']  # 名前「foo」のみを公開することを明記
```

**アンダースコア「_」を使った命名と __all__ 変数によって公開するものを制御する例**

このコードを含んだモジュールの名前を「mymod.py」として保存し、これを利用する側でインポートする例を幾つか示す。

```
from mymod import *  # 全てをインポート

foo()  # 'foo'
_bar()  # 例外（アンダースコアで始まるものはインポートされないため）
baz()  # 例外（__all__変数で除外されているため）
```

**「from mymod import *」する例**

この場合は、_bar 関数は名前がアンダースコアで始まるために、baz 関数は __all__ 変数の要素に含まれていないために、どちらも呼び出そうとするとエラー（NameError 例外）が発生する。

ただし、上の例で公開されていなかったものでも、「from モジュール名 import 特定の名前」形式でインポートすることは可能だ。

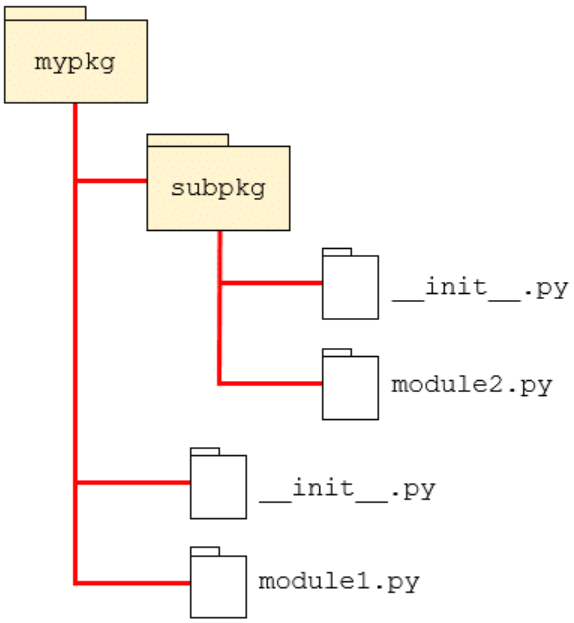
```
from mymod import foo, _bar, baz

foo()
_bar()
baz()
```

公開したくないものでもインポートされる可能性はある

## パッケージでの名前の公開（エクスポート）

パッケージは一般にフォルダを使って、複数のモジュール（.py ファイル）を構造化したものになる。パッケージを作成する際にはフォルダごとに __init__.py ファイルを置く。これは空でもよいが、何らかの初期化処理を行うこともできる。

ここでは以下の構造を持つパッケージを例とする。




サンプルのパッケージ

mypkg がパッケージのトップレベルにあり、その下にサブモジュールを含んだ「module1.py」ファイルと、サブパッケージである「subpkg」フォルダがあり、その中にはサブサブモジュールを含んだ「module2.py」ファイルがある。各フォルダには __init__.py ファイルがある。

module1.py ファイルのコードを以下に示す。

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

module1 モジュールのコード（mypkg/module1.py ファイル）

module2.py ファイルのコードは次の通りだ。

```
def goodbye():
    print('Good-bye')

def test():
    print('test')
```

module2 モジュールのコード（mypkg/subpkg/module2.py ファイル）

2 つの __init__.py ファイルの内容はここでは空とする。

この場合、利用者側では以下のようにしてパッケージ、そのサブモジュールなどをインポートできる。

```
import mypkg  # トップレベルをインポート
import mypkg.module1  # サブモジュールをインポート
from mypkg import module1 # サブモジュールをインポート
from mypkg.module1 import hello # サブモジュールから関数をインポート
```

パッケージのインポートの例

ただし、「import mypkg」としてトップレベルをインポートした場合、実際にはそのサブモジュールは利用できない（このコードは mypkg と同じ階層に例えば「test.py」のようにして置いたものとする）。

```
import mypkg

mypkg.subpkg.module2.goodbye()  # AttributeError例外
```

パッケージのトップレベルをインポートしたが、subpkg はロードされていない

これはサブモジュールやサブパッケージが実行環境にロードされ、mypkg の属性として設定されていないからだ。トップレベル（あるいはその下層レベル）をインポートしたときに、サブモジュールやサブパッケージがロードされて、「パッケージ名 . サブモジュール名 . 関数名」のようにして利用できるようにするには、__init__.py ファイルに初期化処理を記述する。

初期化処理では、サブモジュールやサブパッケージを __init__.py ファイル内部でインポートする。トップレベルの __init__.py ファイルにこれを記述した例を以下に示す。

```
import mypkg.module1  # module1サブモジュールをロード
import mypkg.subpkg  # subpkgサブパッケージをロード
```

トップレベルの __init__.py ファイルの内容（mypkg/__init__.py ファイル）

こうすることで、mypkg パッケージのインポート時に、module1 サブモジュールと subpkg サブパッケージがロードされるようになる。そのため、以下のように module1 サブモジュールの関数を呼び出せる。

```
import mypkg

mypkg.module1.hello()  # 'Hello'
mypkg.subpkg.module2.goodbye()  # AttributeError例外
```

mypkg.module1.hello 関数は呼び出せるが、mypkg.subpkg.module2.goodbye 関数は呼び出せない

ただし、subpkg サブパッケージにある module2 サブモジュールはロードされていないので、そこで定義されている関数は呼び出せる例外が発生する。これには subpkg フォルダにある __init__.py ファイルで以下のようにモジュールをインポートすればよい。

```
import mypkg.subpkg.module2
```

サブパッケージの __init__.py ファイルの内容（mypkg/subpkg/__init__.py ファイル）

__init__.py ファイルはモジュールやパッケージのインポート時に自動的に実行されるので、mypkg/__init__.py ファイルにより、subpkg がインポートされるタイミングで、mypkg/subpkg/__init__.py ファイルも実行される。こうしたロードの連鎖により、うまくパッケージのインポートが行えるようになっている。なお、パッケージ内のサブモジュールやサブパッケージをトップレベルのパッケージ名から指定してインポートすることを「絶対インポート」と呼ぶ。

このようにして、__init__.py ファイルに上の初期化処理を書けば、mypkg パッケージをインポートするだけで、パッケージ内で定義されたものを利用できるようになる（あるいは、トップレベルの __init__.py ファイルで今述べたことに相当するコードを記述してもよい）。

```
import mypkg

mypkg.module1.hello()  # 'Hello'
mypkg.subpkg.module2.goodbye()  # 'Good-bye'
```

mypkg パッケージを利用するコード（test.py ファイル）

__init__.py ファイルでは、既に述べたように、「from パッケージ名 import *」を実行する際に、何をインポートするのかを __all__ 変数で指定することもできる。

ここでは、例としてトップレベルの __init__.py ファイルのコードを以下のようにしてみよう。

```
from .module1 import hello
from .subpkg.module2 import goodbye, test

__all__ = ['hello', 'test']
```

書き換えた後のトップレベルの __init__.py ファイルの内容（mypkg/__init__.py ファイル）

これはトップレベルで、hello 関数（module1 サブモジュール）と goodbye 関数、test 関数（subpkg.module2 サブモジュール）をインポートしている。インポートすることにより、その名前が外部に公開されるので、これらは「from mypkg import hello」のようにしてインポートできる。その一方で、__all__ 変数には 'hello' と 'test' のみを要素とするリストを代入しているので、「from mypkg import *」形式のインポートでは goodbye 関数はインポートされない。

なお、ここで行っている「from .module1 import hello」のようなインポートの仕方を「相対インポート」と呼ぶ。モジュール／パッケージの名前の前にあるドット「.」はパッケージ内で「そのファイル（ここでは __init__.py ファイル）と同じ階層にある」ことを意味する。mypkg/__init__.py ファイルと同じ階層には module1 モジュール（module1.py ファイル）と subpkg サブパッケージ（subpkg フォルダ）があるので、ここではこのように記述している。パッケージ階層が深いときには、「..」「...」のようにして上位の階層を指定していくことも可能だ。

「from mypkg import *」形式でインポートを行う例を示す。

```
from mypkg import *

hello()  # 'Hello'
test()  # 'test'
goodbye()  # NameError例外
```

「from mypkg import *」形式では goodbye 関数がインポートされない

これに対して、次のコードなら goodbye 関数もインポートできる。

```
from mypkg import hello, goodbye, test

hello()  # 'Hello'
test()  # 'test'
goodbye()  # 'Good-bye'
```

名前を明示すれば goodbye 関数もインポート可能

## 例外

プログラムの実行時に発生する例外を処理するには、try 文を記述する。構文を以下に示す。

```
try:
    例外を発生させる可能性があるコード
except 例外クラス1 as 変数:
    例外クラス1で表される例外を捕捉して処理するコード
    例外クラスのオブジェクトは変数に代入される（「as 変数」は省略可能）
except 例外クラス2 as 変数:
    例外クラス2で表される例外を捕捉して処理するコード
    ……
except Exception as 変数:
    上で捕捉できなかった例外を捕捉して処理するコード
else:
    例外が発生しなかったときに実行するコード
finally:
    例外発生の有無に関わらず、最後に実行するコード
```

try 文の構文

try 節には例外を発生する可能性があるコードを記述する。その後には、except 節で個別に例外を捕捉して、それを処理するコード（例外ハンドラ）を記述する。except 節は必要に応じて、複数記述できる（finally 節があるときには省略可）。また、else 節には例外が発生しなかったときに実行するコードを記述する（省略可）。最後の finally 節では例外が発生したかどうかに関係なく、最後に処理をすべきコードを記述する（except 節があるときには省略可）。

except 節に記述する例外クラスは、個々の例外に対応したクラスであり、BaseException クラスを頂点とした階層構造を持つ。ただし、Python プログラムの実行時に発生する例外の多くは Exception クラスから派生する。そのため、通常の例外処理では Exception クラスを捕捉する exception 節を最後に置くことで、ほぼ全ての例外を捕捉して処理できる。

except 節には例外階層の下位にあるクラスから並べていく必要があることには注意すること。例えば、ZeroDivisionError 例外クラスは、ArithmeticError クラスの派生クラスだが、それらを捕捉したいときに次のように書くと、ZeroDivisionError 例外が ArithmeticError クラスの例外ハンドラ（except 節）で捕捉されてしまう。

```
try:
    何らかの数値計算
except ArithmeticError as e:
    例外を処理
except ZeroDivisionError as e:
    ZeroDivisionErrorはArithmeitcErrorでもあるので上で捕捉されてしまう
```

except 節には例外クラスのクラス階層の下位にあるものから記述しないといけない

例外を発生させるには raise 文を使用する。

```
raise 例外クラス(例外についての情報)
raise 例外クラス(例外についての情報) from 例外オブジェクト
```

raise 文の構文

1 つ目の構文は例外を発生させる典型的な構文で、例外クラスを関数のように呼び出して、そこに発生した例外に関する情報を与える。2 つ目の構文は、多くの場合、例外処理の際に新たに例外を発生させる場合に使用する。「from」の後には現在処理している例外を表すオブジェクト（except 節で受け取ったもの）を書き、それとは別の例外クラスを新規に生成することになる。

例外処理の例を以下に示す。

```
prompt = """
0) TypeError
1) ZeroDivisionError
2) ArithmeticError
3) no exception
4) Raise Exception in except clause
select error: """

err = int(input(prompt))
errlist = [TypeError, ZeroDivisionError, ArithmeticError]

try:
    if err < len(errlist):
        raise errlist[err]
    elif err == 3:
        print('you select other')
    else:
        raise NameError("select 'Raise Exception in except clause'")
except TypeError as e:
    print('you select ', e.__class__.__name__)
except ZeroDivisionError as e:
    print('you select ', e.__class__.__name__)
except ArithmeticError as e:
    print('you select ', e.__class__.__name__)
except NameError as e:
    raise Exception('raised in excep clause') from e:
else:
    print('in else clause')
finally:
    print('in finally clause')
```

**例外処理の例**

この例では、ユーザーの選択に応じて、何種類かの例外を発生させて、それを処理するか、例外を発生させないか、例外を処理する中で再度例外を生成させるかのいずれかを行うようになっている。なお、例外オブジェクトを「e.__class__.__name__」としているのは例外オブジェクトのクラス名を調べて、その名前を取り出す処理だ。

興味のある方は except 節にある ZeroDivisionError 例外の捕捉と ArithmeticError 例外の捕捉の順番を逆にするなどしてみてほしい。

最後に、既に述べたが例外クラスは一般に Exception クラスの派生クラスとなる。そのため、例外クラスを自作する必要があるときには、以下のように基底クラスに Exception クラスを指定する。

```
class SomeError(Exception):
    pass
```

例外クラスの定義の例

このとき、例外クラスの名前は「Error」で終わらせるのが推奨されている。

本章ではモジュールとパッケージ、例外についてまとめた。次章では Python の特殊メソッドについてまとめる。

# ［Python チートシート］特殊メソッド編

インスタンスの初期化など、オブジェクトの振る舞いをプログラマーが細かく調整するために使用できる特殊メソッドについてギュッとまとめた。

かわさきしんじ，Deep Insider 編集部（2020 年 02 月 18 日）

Python では、クラスの定義時に、そのクラスのインスタンスの振る舞いを細かく調整するためにさまざまな「特殊メソッド」が提供されている。今回はこれらについて見ていこう。

## 特殊メソッドとは

Python のクラスでは「特殊メソッド」と呼ばれるメソッドを定義（オーバーライド）できる。特殊メソッドとは、各種の演算子や組み込み関数などの操作の対象として、独自のクラスを利用できるようにするための仕組みだと考えられる。つまり、クラスを自分で定義しているときに、適切な名前の特殊メソッドを適切にオーバーライドすることで、例えば、次のような処理が可能になる。

- インスタンスの生成と初期化
- インスタンス同士の比較
- 他の型への変換
- 数値として演算
- 反復可能オブジェクト（コンテナ）的な動作
- 関数的な動作

特殊メソッドの名前は、特定の処理を示す名前を、2 つのアンダースコア「__」で囲んだものになる。例えば、「__init__」は「インスタンスの初期化」を意味する「init」を「__」で囲んだものになっている。

特殊メソッドは非常に多いので、以下ではその幾つかを抜粋して紹介していく。

## インスタンスの生成と破壊

上でも軽く触れたように、クラスのインスタンス（オブジェクト）を初期化するための __init__ メソッドは、特殊メソッドの代表的な例である。Python では、インスタンスの生成や破壊に関する特殊メソッドとして次のようなものがある。

| 特殊メソッド | 対応する操作／演算 | 説明 |
|---|---|---|
| __new__(cls, ……) | クラスのインスタンス生成時 | クラスのインスタンスを生成するために呼び出される |
| __init__(self, ……) | クラスのインスタンスの初期化時 | クラスのインスタンスの生成後に、それを初期化するために呼び出される |
| __del__(self) | クラスのインスタンスの破壊時 | クラスのインスタンスが破壊されるときに呼び出される |

インスタンスの生成と破壊に関連する特殊メソッド

__new__ メソッドは、クラスのインスタンス生成をカスタマイズする際に定義する。暗黙の第 1 パラメーターには「self」ではなく「cls」を指定する。インスタンスの生成自体は、親クラスの __new__ メソッドを呼び出して、cls（とその他の引数）を指定する、つまり「super().__new__(cls, ……)」とするのが典型的なやり方だ。加えて、__new__ メソッドでしか行えないインスタンスの初期化も行える（後述）。

__init__ メソッドは、既にご存じの通り、インスタンスの初期化を行うために使用する。__new__ メソッドを定義した場合、そこで作成されたインスタンスは __init__ メソッドへと引き渡される。

__del__ メソッドは、インスタンスが破壊されるタイミングで自動的に呼び出される。特殊なリソースを破壊する必要があるようなときには、これを定義する必要があるだろう。注意点は、このメソッドが呼び出されるタイミングだ。つまり、「del インスタンス」を実行したタイミングで呼び出されるわけではない。インスタンスが破壊される（このメソッドが呼び出される）のは、それに結び付けられている名前がなくなった時点なので注意しよう。また、基底クラスで __del__ メソッドが定義されているのであれば、それらを呼び出して、オブジェクトの破壊が確実に行われるようにする必要がある。

以下に例を示す。

```
class Foo:
    def __new__(cls):
        print('__new__')
        self = super().__new__(cls)  # インスタンス生成を行う典型的なコード
        self.attr = 'set in __new__'  # ここでしかできない初期化処理を書いてもよい
        return self  # 生成したインスタンスを返す

    def __init__(self, name='foo'):
        print('__init__')
        self.name = name  # インスタンスの初期化処理

    def __del__(self):
        #super().__del__()  # 基底クラスに__del__メソッドがあれば必ず呼び出す
        print('__del__')  # インスタンスが破壊されるときに行う処理

foo = Foo()  # '__new__'と'__init__'が表示される
print('foo.attr:', foo.attr)  # 'foo.attr: set in __new__'
bar = foo
print('bar.name:', bar.name)  # 'bar.name: foo'
print('del foo')  # この時点ではまだ生成したインスタンスには別名がある
del foo
print('del bar')
del bar  # '__del__' : この時点でインスタンスを束縛する名前がなくなる
```

__new__ ／ __init__ ／ __del__ メソッドの使用例

この例では、__new__ メソッドで「super().__new__(cls)」によりインスタンスを作成した後、「attr」という名前の属性（インスタンス変数）を定義して、それを戻り値として返送している。__init__ メソッドでは、これを受け取り、それに対して「name」という名前の属性を設定している。これにより、このインスタンスは 2 つの属性を持つことになる。

クラス定義後のコードでは、このクラスのインスタンスを生成して、それを 2 つの変数に代入している。そのため、「del foo」「del bar」の 2 つを実行してインスタンスに関連付けられた名前がなくなるまでは、__del__ メソッドが呼び出されることはない。

__new__ メソッドはインスタンス生成に、__init__ メソッドはインスタンスの初期化に使うが、これらをまとめずに 2 つのメソッドとしている理由の一つとして、__init__ メソッドでは「変更不可能なオブジェクトを初期化できない」ことが挙げられる。変更不可能なクラスのインスタンス生成処理においては、それが生成された時点で変更不可能なので、その初期化は __init__ メソッドが呼び出されるよりも前に行う必要がある。つまり、そうしたインスタンスの生成と初期化は __new__ メソッドで行う。

以下に例を示す。これは tuple クラスを継承して、反復可能オブジェクトを受け取り、「( インデックス , 要素 )」で構成されるタプルを要素とするタプルを生成するクラスだ。

```
class NumberedTuple(tuple):
    def __new__(cls, iterable):
        tmp = [(idx, value) for idx, value in enumerate(iterable)]
        self = super().__new__(cls, tmp)
        return self

nt = NumberedTuple(['foo','bar', 'baz'])
print(nt)  # ((0, 'foo'), (1, 'bar'), (2, 'baz'))
```

NumberedTuple クラス

__new__ メソッドでは、enumerate 関数を使って「( インデックス , 要素 )」というタプルを要素とするリストを作成して、それを親クラス（tuple クラス）の __new__ メソッドに渡している。これにより、番号付きのタプルを生成できるようになる。このような変更不可能な型を基に派生クラスを定義するような際には、__new__ メソッドでインスタンス生成と初期化の処理を独自に行う必要があるだろう。

インスタンスを別の型のオブジェクトに変換するのに使える特殊メソッドもある。

| 特殊メソッド | 対応する操作／演算 | 説明 |
|---|---|---|
| __bool__(self) | bool関数<br>真偽テスト | オブジェクトがTrue／Falseのどちらであるかを返す |
| __bytes__(self) | bytes関数 | オブジェクトをバイト文字列で表現したものを生成する |
| __complex__(self) | complex関数 | オブジェクトをcomplex型（複素数型）の値に変換する |
| __float__(self) | float関数 | オブジェクトをfloat型の値に変換する |
| __format__(self, format_spec) | format関数<br>f文字列化時<br>str.formatメソッド | format_specに従ってオブジェクトを文字列化する |
| __int__(self) | int関数 | オブジェクトをint型の値に変換する |
| __repr__(self) | repr関数 | オブジェクトを表す公式な文字列を生成する |
| __str__(self) | str関数 | オブジェクトを表す非公式な文字列を生成する |

インスタンスを別の型のオブジェクトに変換するための特殊メソッド

__repr__ メソッドと __str__ メソッドはどちらも「オブジェクトを文字列化」する際に、repr 関数や str 関数から呼び出されるが、その違いは前者は「オブジェクトの公式な文字列表現」を得るために、後者は「オブジェクトの非公式な文字列表現」を得るために使う点だ。「オブジェクトの公式な文字列表現」とは、eval 関数などにその表現を渡すことで、元の値と同じものを得られるような表現（もしくは、詳細なオブジェクト情報）のこと。「オブジェクトの非公式な文字列表現」とはよりシンプルで読みやすい形でオブジェクトを表したものになる。

object.__str__ メソッドは、object.__repr__ メソッドを呼び出すので、__repr__ メソッドのみをオーバーライドして、__str__ メソッドをオーバーライドしなかったときには、str 関数や print 関数でそのオブジェクトを文字列化しようとすると、__repr__ メソッドの結果が得られる。

以下に例を示す。このクラスは文字列として、数値を格納し、必要に応じてそれらを各データ型に変換できるようにしている（先頭にあるのは、float 型の値に変換可能かを簡易的に調べるスタティックメソッド。float 型に変換可能なら、int 型にも変換できるので、ここでは受け取った文字列のチェックに使っている）。

```
class MyDigit:
    @staticmethod
    def _isfloat(val):
        if val.startswith('-'):
            val = val[1:]
        if val.count('.') <= 1 and val.replace('.', '').isdigit():
            return True
        return False

    def __init__(self, val="0.0"):
        if MyDigit._isfloat(val):
            self.val = val
        else:
            self.val = "0.0"

    def __bool__(self):
        return bool(float(self.val))

    def __int__(self):
        return int(float(self.val))

    def __float__(self):
        return float(self.val)

    def __repr__(self):  # eval関数でオブジェクトを復元可能な表現
        return f"MyDigit('{self.val}')"

    #def __str__(self):  # コメントアウトを外すとself.valが返されるようになる
    #    return self.val

mydigit = MyDigit('foo')
print("MyDigit('foo'):", mydigit)  # MyDigit('foo'): MyDigit('0.0')
print(f"bool(MyDigit('foo')):", bool(mydigit))  # bool(MyDigit('foo')): False

mydigit = MyDigit('1.5')
print(f'bool({mydigit}):', bool(mydigit))  # bool(MyDigit('1.5')): True
print(f'int({mydigit}):', int(mydigit))  # int(MyDigit('1.5')): 1
print(f'repr({mydigit}):', repr(mydigit))  # repr(MyDigit('1.5')): MyDigit('1.5')
print(f'str({mydigit}):', str(mydigit))  # str(MyDigit('1.5')): MyDigit('1.5')
print(f"eval('repr(mydigit)'):", eval('repr(mydigit)'))  # eval('repr(mydigit)'):
MyDigit('1.5')
```

**他のデータ型への変換の例**

ここでは bool ／ float ／ int ／ repr 関数にこのクラスのインスタンスを渡したときに、呼び出されるメソッドをオーバーライドした。実際のコードは簡単なので説明は省略する。基本的には、対応する演算を行って、その結果を文字列化したものからこのクラスのインスタンスを生成しているだけだ。注意したいのは、__str__ メソッドはオーバーライドしていないので、str 関数にこのクラスのインスタンスを渡すと、最終的にこのクラスの __repr__ メソッドが呼び出されて、「MyDigit('1.5')」のような表現が得られるところだ。興味のある方は、__str__ メソッドのコメントアウトを外して動作を確認してほしい。

## オブジェクトの比較

2 つのオブジェクトがあったときに、それらの値が等しいかどうか、同一のオブジェクトかどうかなどを比較できる。こうした目的でオーバーライドできる特殊メソッドとしては以下がある。

| 特殊メソッド | 対応する演算 | 説明 |
|---|---|---|
| __eq__(self, other) | ==演算子 | selfとotherの値が等しいかを調べる |
| __ne__(self, other) | !=演算子 | selfとotherの値が等しくないかを調べる |
| __hash__(self) | hash関数 | オブジェクトのハッシュ値を取得する |
| __lt__(self, other) | <演算子 | selfがotherより小さいかどうかを調べる |
| __le__(self, other) | <=演算子 | selfがother以下かどうかを調べる |
| __gt__(self, other) | >演算子 | selfがotherよりも大きいかどうかを調べる |
| __ge__(self, other) | >=演算子 | selfがother以上かどうかを調べる |

オブジェクトの比較を行うための特殊メソッド

以下に例を示す。

```
class AAA:
    def __init__(self, val=0):
        self.val = val

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.val == other.val
        else:
            kls = other.__class__.__name__
            raise NotImplementedError(
                f'comparison between AAA and {kls} is not supported')

    def __ne__(self, other):
        return not self.__eq__(other)

    def __lt__(self, other):
        if isinstance(other, self.__class__):
            return self.val < other.val
        else:
            kls = other.__class__.__name__
            raise NotImplementedError(
                f'comparison between AAA and {kls} is not supported')

    def __le__(self, other):
        return self.__lt__(other) or self.__eq__(other)

    def __gt__(self, other):
        return not self.__le__(other)  # other.__lt__(self)

    def __ge__(self, other):
        return not self.__lt__(other)  # other.__le__(self)

aaa = AAA(10)
bbb = AAA(20)

print(aaa == bbb)  # False
print(aaa != bbb)  # True
print(aaa < bbb)  # True
print(aaa <= bbb)  # True
print(aaa > bbb)  # False
print(aaa >= bbb)  # False

aaa > 100  # NotImplmentedError例外
```

オブジェクトを比較するメソッドを特殊オーバーライドしたクラスの例（その 1）



→目次に戻る

この例では、__lt__ メソッドと __eq__ メソッドで詳細な定義を行い、他のメソッドは 2 つのメソッドを基にそれぞれの比較結果を返すようにしている。__lt__ メソッドと __eq__ メソッドでは、比較対象（other）が自分のクラス（AAA）もしくはその派生クラスであれば比較できるように isinstance 関数を使って型チェックを行い、クラス階層に含まれないインスタンスであれば、NotImplementedError 例外を発生するようにしてある。

__le__ メソッドが定義されていて __ge__ メソッドが定義されていないとき（またはその逆）、あるいは __lt__ メソッドが定義されていて __gt__ メソッドが定義されていないとき（またはその逆）には、self と other を入れ替えてもう一方のメソッドが呼び出される。これは例えば「a > b」という比較を行いたかったが __gt__ メソッドがないので、「b < a」という比較を行い、結果、__lt__ メソッドが呼び出されるということだ。

なお、functools モジュールの total_ordering デコレータを使用することで、上記のクラス定義は以下のようにも書ける（total_ordering デコレータでクラス定義を修飾した上で、__eq__ メソッドと、__lt__ ／ __le__ ／ __gt__ ／ __ge__ の 4 つのメソッドのいずれか 1 つ、合わせて 2 つのメソッドを定義すればよい）。

```
from functools import total_ordering

@total_ordering
class AAA:
    def __init__(self, val=0):
        self.val = val

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.val == other.val
        else:
            kls = other.__class__.__name__
            raise NotImplementedError(
              f'comparison between AAA and {kls} is not supported')

    def __lt__(self, other):
        if isinstance(other, self.__class__):
            return self.val < other.val
        else:
            kls = other.__class__.__name__
            raise NotImplementedError(
              f'comparison between AAA and {kls} is not supported')
```

オブジェクトを比較する特殊メソッドをオーバーライドしたクラスの例（その 2）

## 呼び出し可能オブジェクト

クラスを定義したり、関数を定義したりすると、それらはかっこ「()」を付加し、そこ中に引数を指定することで、「呼び出す」ことができるようになる。こうしたオブジェクトのことを「呼び出し可能オブジェクト」と呼ぶ。

Python では通常、クラスは呼び出し可能オブジェクトだが、そのインスタンスは呼び出し可能ではない。しかし、インスタンスを呼び出し可能オブジェクトのようにもできる。これには __call__ 特殊メソッドを定義すればよい。

| 特殊メソッド | 対応する操作 | 説明 |
|---|---|---|
| __call__(self, args) | 「self(args)」形式の関数呼び出し | オブジェクトを関数として呼び出す |

インスタンスを呼び出し可能にする特殊メソッド

以下に例を示す。

```
class Hello:
    def __call__(self, x):
        print(f'Hello {x}')

hello = Hello()  # クラスは呼び出し可能オブジェクト
hello('world')  # Helloクラスのインスタンスは呼び出し可能オブジェクト
```

インスタンスを呼び出し可能とする例

## 数値演算

加減乗除と整数除算などをカスタマイズするための特殊メソッドもある。

| 特殊メソッド | 対応する操作 | 説明 |
|---|---|---|
| __neg__(self) | -演算子（単項） | 符号反転に対応する操作を定義 |
| __sub__(self, other) | -演算子（二項） | 減算に対応する操作を定義 |
| __pos__(self) | +演算子（単項） | 数値の符号を変更しない操作に対応する操作を定義 |
| __add__(self, other) | *演算子（二項） | 加算に対応する操作を定義 |
| __mul__(self, other) | *演算子 | 乗算に対応する操作を定義 |
| __truediv__(self, other) | /演算子 | 除算に対応する操作を定義 |
| __floordiv__(self, other) | //演算子 | 整数除算（商）に対応する操作を定義 |
| __mod__(self, other) | %演算子 | 整数除算（剰余）に対応する操作を定義 |
| __divmod__(self, other) | divmod関数 | divmod関数から呼び出される |
| __pow__(self, other[, modulo]) | pow関数／**演算子 | 累乗に対応する操作を定義 |
| __lshift__(self, other) | <<演算子 | 左シフト演算に対応する操作を定義 |
| __rshift__(self, other) | >>演算子 | 右シフト演算に対応する操作を定義 |
| __and__(self, other) | &演算子 | ビット単位のAND演算に対応する操作を定義 |
| __xor__(self, other) | ^演算子 | ビット単位のXOR演算に対応する操作を定義 |
| __or__(self, other) | \|演算子 | ビット単位のOR演算に対応する操作を定義 |
| __abs__(self) | abs関数 | 絶対値の取得に対応する操作を定義 |

数値演算をカスタマイズする特殊メソッド（一部）

以下に例を示す。

```
class MyDigit:
    @staticmethod
    def _isint(val):
        if val.startswith('-'):
            val = val[1:]
        if val.count('.') == 0 and val.isdigit():
            return True
        return False

    def __init__(self, val="0.0"):
        if MyDigit._isint(val):
            self.val = val
        else:
            self.val = "0"
```

```
def __add__(self, other):
    if isinstance(other, self.__class__):
        tmp = str(int(self.val) + int(other.val))
        return MyDigit(tmp)
    else:
        raise NotImplementedError()

def __sub__(self, other):
    if isinstance(other, self.__class__):
        tmp = str(int(self.val) - int(other.val))
        return MyDigit(tmp)
    else:
        raise NotImplementedError()

def __mul__(self, other):
    if isinstance(other, self.__class__):
        tmp = str(int(self.val) * int(other.val))
        return MyDigit(tmp)
    else:
        raise NotImplementedError()

def __floordiv__(self, other):
    if isinstance(other, self.__class__):
        tmp = int(self.val) // int(other.val)
        return MyDigit(str(tmp))
    else:
        raise NotImplementedError()

def __mod__(self, other):
    if isinstance(other, self.__class__):
        tmp = int(self.val) % int(other.val)
        return MyDigit(str(tmp))
    else:
        raise NotImplementedError()

def __truediv__(self, other):
    return self.__floordiv__(other)

def __divmod__(self, other):
    q = self.__floordiv__(other)
    r = self.__mod__(other)
    return (q, r)
```

```
    def __neg__(self):
        if self.val.startswith('-'):
            tmp = self.val[1:]
        else:
            tmp = "-" + self.val
        return MyDigit(tmp)

    def __str__(self):
        return self.val

d1 = MyDigit('17')
d2 = MyDigit('4')

print(d1 + d2)  # 21
print(d1 - d2)  # 13
print(d2 - d1)  # -13
print(d1 * d2)  # 68
print(d1 / d2)  # 4
print(d1 % d2)  # 1
(q, r) = (divmod(d1, d2))
print(q, r)  # 4 1
print(-d1)  # -17
```

**数値演算をカスタマイズする特殊メソッドをオーバーライドする例（その 1）**

この例では、数値を文字列の形で保存し、その四則演算を行うように特殊メソッドをオーバーライドした（加えて、値を簡単に表示できるように __str__ メソッドもオーバーライドしている）。多くのメソッドでは、other がこのクラス（かその派生クラス）のインスタンスであることを確認してから演算を行ってその結果を返送し、そうでない場合には先ほどと同様に NotImplementedError 例外を発生させるようにしてある。それ以外は、対応する演算を行って、そこからこのクラスのインスタンスを新たに生成し、それを戻り値として渡すようにしているだけだ。

今紹介した特殊メソッドは、それが二項演算子の場合、self は演算子の左側に置かれた被演算子となる。これに対して、演算子の右側（right hand）に置かれた被演算子を self とする特殊メソッドもある。これらの特殊メソッドの名前は、最初のアンダースコア「_」と演算や操作を表す名前の間に「r」を挟んだものだ。

例えば、「a / b」という操作を行う際に、除数である「b」を self とする特殊メソッドは b が属するクラスの __rtruediv__(self, other) メソッドになる。この場合は、「other / self」という除算を行うことに注意しよう（「__truediv__(self, other)」では「self / other」という除算を行う）。

以下に簡単な例を示す。

```
class AAA:
    def __init__(self, val=0):
        self.val = val

    def __truediv__(self, other):
        if isinstance(other, self.__class__):
            return self.val / other.val
        return NotImplemented

class BBB:
    def __init__(self, val=0):
        self.val = val

    def __rtruediv__(self, other):
        if isinstance(other, AAA):
            return other.val / self.val
        return NotImplemented

aaa = AAA(10)
bbb = BBB(5)
print(aaa / bbb)  # 2.0
```

**数値演算をカスタマイズする特殊メソッドをオーバーライドする例（その 2）**

この例では、AAA と BBB という 2 つのクラスがある。AAA では「/」演算子による除算をオーバーライドしているが、これを使えるのは AAA クラスとその派生クラスに限られる。それ以外の場合は、NotImplemented オブジェクトを戻り値としている（NotImplementedError 例外を発生させているわけではない点に注意）。これが戻り値となった場合、Python の処理系は「右側の被演算子が属するクラスに、対応する __r……__ メソッドがあれば」それを呼び出してくれる。

一方、BBB クラスは AAA クラスとは継承関係にないが、まさに今述べたような、右側の被演算子を self とする __rtruediv__ メソッドをオーバーライドしている。その内部では「other / self」演算の other が AAA クラスのインスタンスであることを前提としてコードを書いている。こうすることで、「AAA クラスのインスタンス / BBB クラスのインスタンス」といった除算が行えるようになる。

最後に累算代入演算子も紹介しておこう。

| 特殊メソッド | 対応する演算子 | 説明 |
|---|---|---|
| __iadd__(self, other) | += | selfとotherを加算した結果をselfに代入する |
| __isub__(self, other) | -= | selfとotherを減算した結果をselfに代入する |
| __imul__(self, other) | *= | selfとotherを乗算した結果をselfに代入する |
| __itruediv__(self, other) | /= | selfをotherで除算した結果をselfに代入する |
| __ifloordiv__(self, other) | //= | selfをotherで整数除算した商をselfに代入する |
| __imod__(self, other) | %= | selfをotherで整数除算した剰余をselfに代入する |
| __ipow__(self, other[, modulo]) | **= | selfの「other」乗をselfに代入する |
| __ilshift__(self, other) | <<= | selfをotherだけ左シフトした結果をselfに代入する |
| __irshift__(self, other) | >>= | selfをotherだけ右シフトした結果をselfに代入する |
| __iand__(self, other) | &= | selfとotherのビット単位ANDの結果をselfに代入する |
| __ixor__(self, other) | ^= | selfとotherのビット単位XORの結果をselfに代入する |
| __ior__(self, other) | \|= | selfとotherのビット単位ORの結果をselfに代入する |

累算代入演算子の振る舞いをカスタマイズする特殊メソッド

__iadd__ メソッドをオーバーライドする例を以下に示す。

```
class MyDigit:
    @staticmethod
    def _isint(val):
        if val.startswith('-'):
            val = val[1:]
        if val.count('.') == 0 and val.isdigit():
            return True
        return False

    def __init__(self, val="0"):
        if MyDigit._isint(val):
            self.val = val
        else:
            self.val = "0"

    def __iadd__(self, other):
        if isinstance(other, self.__class__):
            tmp = str(int(self.val) + int(other.val))
            self.val = tmp
            return self
        elif isinstance(other, int):
            tmp = str(int(self.val) + other)
            self.val = tmp
            return self
        else:
            raise NotImplementedError()

    def __str__(self):
        return self.val

mydigit1 = MyDigit('10')
mydigit2 = MyDigit('20')
mydigit1 += mydigit2
print(mydigit1)  # 30
mydigit1 += -10
print(mydigit1)  # 20
```

__iadd__ メソッドをオーバーライドする例

## コンテナ型

リストやタプル、辞書などの反復可能オブジェクトは複数の要素を格納するという意味で「コンテナ」と呼ばれることもある。これらコンテナオブジェクトの操作をエミュレートするための特殊メソッドもある。

| 特殊メソッド | 対応する操作 | 説明 |
|---|---|---|
| __len__(len) | len関数 | 格納している要素の数を取得する |
| __getitem__(self, key) | self[key] | 指定されたkeyに対応する値を得る。keyはインデックス／キー（辞書）／スライスの場合がある |
| __setitem__(self, key, value) | self[key] = value | 指定されたkeyに対応する値をvalueとする。変更可能なコンテナでのみ定義可能 |
| __delitem__(self, key) | del self[key] | 指定されたkeyの値を削除する。変更可能なコンテナでのみ定義可能 |
| __missing__(self, key) | self[key] | selfが辞書（の派生クラス）の場合に、keyが辞書に存在しなかったときに呼び出される。__missing__メソッドの戻り値がself[key]の戻り値となる。あるいは、__missing__メソッドから例外を発生させてもよい |
| __iter__(self) | iter関数 | コンテナに格納されている要素を反復するイテレータを取得する |
| __contains__(self, item) | x in self<br>x not in self | selfにxが含まれているかどうかを調べる |
| __reversed__(self) | reversed関数 | 要素を逆順に反復するイテレータを取得する |

コンテナ型の振る舞いをエミュレートするための特殊メソッド

以下に例を示す。

```
class Stack:
    def __init__(self):
        self.stack = []

    def __len__(self):
        print('__len__')
        return len(self.stack)

    def __iter__(self):
        print('__iter__')
        return iter(self.stack)

    def __getitem__(self, key):
        print('__getitem__')
        return self.stack[key]

    def __setitem__(self, key, item):
        print('__setitem__')
        self.stack[key] = item
```

```
    def __delitem__(self, key):
        print('__delitem__')
        del self.stack[key]

    def __contains__(self, item):
        print('__contains__')
        return item in self.stack

    def __reversed__(self):
        print('__reversed__')
        return reversed(self.stack)

    def __repr__(self):
        return repr(self.stack)

    def __str__(self):
        return str(self.stack)

    def copy(self):
        return self.stack.copy()

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        return self.stack.pop()

stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack[0])  # __getitem__→1
stack[1] = 20  # __setitem__
del stack[2]  # __delitem__
for item in stack:  # __iter__
    print(item)  # 1→20
print(1 in stack)  # __contains__→True
for num in reversed(stack):  # __reversed__
    print(num)  # 20→1
```

コンテナ型の振る舞いをエミュレートするため例

この例では、リストを内部的に使用したスタック（要素のプッシュとポップを主な操作とするコンテナ）を定義している。その中で、上述した特殊メソッドをオーバーライドしている。オーバーライドした特殊メソッドの多くでは、呼び出されたメソッドの名前を出力するようにしてある。実際の振る舞いは、内部で使用するリストに丸投げの形になっている（多くの場合、継承か包含かはさておき、基盤となるクラスが持つ機能をこのように流用することになるだろう）。

本章では、自分でクラスを定義する際に、そのクラスの振る舞いを細やかに調整するために使える特殊メソッドを紹介した。ここでは取り上げられなかったものもあるので、興味のある方は Python ドキュメント「特殊メソッド名」を参照されたい。

編集：@IT 編集部
発行：アイティメディア株式会社