@IT
atmarkIT

解決！ Python
タプル／辞書／集合編

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。

タプルを作成するには

Python でタプルを作成するには幾つかの方法がある。それらの方法と注意点を紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 03 月 07 日）

```
# タプルの要素をカンマ「,」で並べる
t0 = 0, 1, 2
print(type(t0))  # <class 'tuple'>
print(t0)  # (0, 1, 2)

# カンマで区切った要素をかっこ「()」で囲む
t0 = (0, 1, 2)  # タプルであることが明示的になる
t0 = ()  # 要素が0個のタプル
print(t0)  # ()
print(len(t0))  # 0

t0 = (1)  # タプルではない
print(type(t0))  # <class 'int'>
t0 = (1,)  # 1要素のタプルは要素とカンマを記述する
print(t0)  # (1, )

# tuple関数を使う
t0 = tuple()  # 要素が0個のタプル
print(t0)
t1 = tuple([0, 1, 2])  # tuple関数に反復可能オブジェクトを渡す
print(t1)  # (0, 1, 2)

# 注意点
t0 = (0, 1, 2)
t0[0] = -1  # TypeError

l = [0, 1, 2]  # リストはその要素を変更可能
t = (l,)  # 変更可能なオブジェクトをタプルの要素とする
t[0][0] = -1  # OK
print(t)  # ([-1, 1, 2],)
```

タプルとは

Python におけるタプルは以下のような特徴を持ったオブジェクトだ。

- その要素を変更できない
- 要素の反復（イテレート）が可能
- インデックスを使って要素にアクセスできる
- 要素をアンパックして、複数の変数に要素を個別に代入できる

本稿ではタプルの作成（初期化）方法を紹介する。

→目次に戻る

タプルを作成する

タプルを作成するには以下の方法がある。

- タプルの要素となる値をカンマ「,」で区切って並べる
- tuple 関数を呼び出す

1 つ目の方法ではタプルを作成することを明確にするかっこ「()」の有無や要素が 1 つだけのタプルの作成などで注意する点もある。

以下はシンプルにカンマで区切ってタプルを作成する例だ。

```
t0 = 0, 1, 2
print(type(t0))  # <class 'tuple'>
print(t0)  # (0, 1, 2)
```

1 行目では 3 つの数値「0」「1」「2」をカンマで区切って並べている。これによりタプルが作成され、変数 t0 に代入されている。type 関数にこれを渡してその型を調べると「<class 'tuple'>」となっている。また、print 関数にこのタプルオブジェクトを渡すと、タプルの要素がかっこ「()」で囲まれて表示される。このかっこはタプルがタプルであることを明確にするものだ。

上で見たように、タプルを作成するときにはカンマで区切ってその要素を並べるだけでよいが、かっこを使ってそれがタプルであることを明確にできる。以下に例を示す。

```
t0 = (0, 1, 2)  # タプルであることが明示的になる
```

要素が 0 個のタプル（空のタプル）を作成するには以下のようにかっこのみを記述する。

```
t0 = ()  # 要素が0個のタプル
print(t0)  # ()
print(len(t0))  # 0
```

最初の行では「()」として空のタプルを作成している。print 関数に渡すと「()」とだけ表示され、これが要素を持っていないことが分かる。タプルやリストなどの要素数を調べる len 関数に渡すと要素がないので戻り値は「0」となる。

要素を 1 個だけ持つタプルを作成するときには注意が必要だ。「(要素)」とするのではなく、以下のように最後にカンマを付加する必要がある。

```
t0 = (1)  # タプルではない
print(type(t0))  # <class 'int'>
t0 = (1,)  # 1要素のタプルは要素とカンマを記述する
print(t0)  # (1, )
```

最初の行では「1」をかっこで囲んでいるが、これは式の評価順序を優先するためのかっことして処理され、結果は整数の 1 となる。3 行目では「1」に続けてカンマを記述することでそれがタプルであると Python の処理系に伝えている。これによりタプルが作成される。

tuple 関数を使ってもタプルを作成できる。この関数は引数を指定しなければ空のタプルを作成し、反復可能オブジェクトを渡せば、それを基にタプルを作成する。

以下に例を示す。

```
t0 = tuple()  # 要素が0個のタプル
print(t0)
t1 = tuple([0, 1, 2])  # tuple関数に反復可能オブジェクトを渡す
print(t1)  # (0, 1, 2)
```

1 行目では引数なしで tuple 関数を呼び出しているので空のタプルが作成されている。3 行目では反復可能オブジェクトの一つであるリストを tuple 関数に渡しているので、その要素からタプルが作成されている。

注意点

タプルはその要素を変更できない。例えば、以下のコードは (0, 1, 2) というタプルを作成して、その先頭要素の値を変更しようとしているが（t0[0] = -1）、これは TypeError 例外を発生させる。

```
t0 = (0, 1, 2)
t0[0] = -1  # TypeError
```

しかし、リストのような変更可能なオブジェクトをタプルの要素とすることは可能だ。

```
l = [0, 1, 2]  # リストはその要素を変更可能
t = (l,)  # 変更可能なオブジェクトをタプルの要素とする
t[0][0] = -1  # OK
print(t)  # ([-1, 1, 2],)
```

この例では「t[0][0] = -1」という操作をしているが、これは「t[0]」としてタプルの要素となっているリストを取り出して、その次の「[0]」でリストの先頭要素を対象に -1 を代入している。これは可能な操作である。

また、タプルを作成するときにかっこを省略できるのは上で紹介した通りだが、省略できない場合もある。

例えば、関数の引数としてタプルをリテラル値として指定するときにはかっこを省略できない。以下に例を示す。

```
def func(x):
    print(x)
```

func 関数は引数を 1 つ受け取り、それを表示するだけのものだ。この関数に (0, 1, 2) というタプルを渡してみよう。

```
func(0, 1, 2)  # TypeError
```

func 関数が受け取る引数は 1 つだけなのに、上のような書き方では 3 つの引数を渡していると解釈され、TypeError 例外が発生する。この場合、かっこを省略せずに次のように記述する必要がある。

```
func((0, 1, 2))  # OK
```

変数にタプルを代入しているのなら、その変数を func 関数に渡すのはもちろん問題ない。

```
t = (0, 1, 2)
func(t)  # OK
```

関数が可変長引数を受け取る場合にも、タプルを囲むかっこを省略すると想定とは異なる振る舞いをする。以下に例を示す。

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

(0, 1, 2) というタプルを渡しているつもりかもしれないが、これは 3 つの整数 0、1、2 を print 関数に渡しているだけだ。そのため、print 関数の出力は 3 つの整数を半角空白文字で区切ったものになる。タプルを渡したければ、かっこを忘れずに記述する。

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

こうすれば、print 関数は 3 つの要素を持つタプルオブジェクトを 1 個受け取り、その内容を表示してくれる。

この他にもタプルを囲むかっこを省略できない場面はある。タプルを渡したつもりが想定外の振る舞いをするときには、かっこを省略していないかを調べてみるのも解決手段の一つとなるだろう。

タプルの要素を取り出すには

Python でタプルの要素を取り出すにはインデックスやスライスを使う方法と要素のアンパックを使う方法がある。これらを紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 03 月 14 日）

```
# アンパック
mytuple = (0, 1, 2)
x, y, z = mytuple
print(x, y, z)  # 0 1 2

# インデックスによるアクセス
mytuple = tuple(range(10))
print(mytuple)  # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

n = mytuple[2]  # 正数はタプル先頭からのインデックスを表す
print(n)  # 2
n = mytuple[-3]  # 負数はタプル末尾からのインデックスを表す
print(n)  # 7

# スライスによるアクセス
s = mytuple[0:5]  # タプルの0～4番目の要素
print(s)  # (0, 1, 2, 3, 4)
s = mytuple[2:]  # 2番目以降の全要素
print(s)  # (2, 3, 4, 5, 6, 7, 8, 9)
s = mytuple[:5]  # タプルの0～4番目の要素
print(s)  # (0, 1, 2, 3, 4)
s = mytuple[:]  # タプルの全要素
print(s)  # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
s = mytuple[::2]  # 0、0+1×2=2、0+2×2=4、……の要素（偶数インデックス）
print(s)  # (0, 2, 4, 6, 8)
s = mytuple[0:5:2]  # 0、0+1×2=2、0+2×2=4がインデックスの要素
print(s)  # (0, 2, 4)

#  スライスに負数を指定した場合
s = mytuple[-1:]  # 末尾要素のみで構成されるタプル
print(s)  # (9,)
s = mytuple[-5:-2]  # s = mytuple[5:8]と同じ
print(s)  # (5, 6, 7)
s = mytuple[::-1]  # 全要素を逆順に並べたタプル
print(s)  # (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

タプルの要素のアンパック

タプルはリストとは異なり、その要素を変更できない。そのため、タプルに対して行えるのは、主にその要素の取り出しとなる。タプルから要素を取り出すには、インデックスを使って個別の要素を指定するか、スライスを使って特定範囲の要素群を指定するか、要素をアンパックするかとなる。

一番簡単なのは要素をアンパックすることだ。ここでいうアンパック（アンパック代入）とはタプル（やリスト、集合、辞書など）に含まれる個々の要素を別の変数へ代入する操作のことだ。これには代入先の変数の数とタプルの要素数が一致する必要がある。

以下に例を示す。

```
mytuple = (0, 1, 2)
x, y, z = mytuple
print(x, y, z)  # 0 1 2
```

この例では mytuple には 0、1、2 の 3 つの要素を持つタプルが代入されている。2 行目でタプルのアンパック代入を行っている。タプルは 3 つの要素で構成されているので、代入先の変数もそれに合わせて 3 つ用意してある。変数の数と要素の数が一致しない場合には「too many values to unpack」「not enough values to unpack」などのメッセージとともに ValueError 例外が発生する。

インデックスを使ったタプルの要素の取り出し

インデックスを使ってタプルの要素を取り出すには、角かっこ「[]」に取り出したい要素のインデックスを指定する。

以下に例を示す。

```
mytuple = tuple(range(10))
print(mytuple)  # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

n = mytuple[2]  # 正数はタプル先頭からのインデックスを表す
print(n)  # 2
n = mytuple[-3]  # 負数はタプル末尾からのインデックスを表す
print(n)  # 7
```

正の整数をインデックスに指定した場合は、タプルの先頭からのインデックスを表す。タプルの先頭要素のインデックスは 0 となることには注意すること。よって、0 が先頭の要素を表し、1 がその次の要素を表し、……となる。上の例では、タプルには 0 ～ 9 の 10 個の要素が含まれているのでインデックス 0 はその先頭要素である「0」を、インデックス 2 は 3 つ目の要素である「2」を表す。

負の整数を指定した場合は、タプル末尾からのインデックスとなる。－ 1 が末尾の要素を表し、－ 2 は末尾から 2 番目の要素を表し、……となる。上の例では、タプルには 0 ～ 9 の 10 個の要素が含まれているのでインデックス－ 1 はその末尾要素である「9」を、インデックス－ 3 は末尾から 3 番目の要素である「7」を表す。

スライスを使ったタプルの要素の取り出し

タプルやリストなど、要素をインデックスで指定可能なオブジェクトに対して、その要素の範囲を角かっこに続けて「タプル [lower_bound:upper_bound:stride]」のようにして、その範囲を指定することで、その範囲の要素を取り出せる。取り出したものや、このような範囲指定のことをスライスと呼ぶ。ここで lower_bound はスライスの開始位置を、upper_bound はスライスの終了位置を、stride は lower_bound と upper_bound に含まれる範囲のインデックスを計算する際の増分（または減分）を表す。

lower_bound、upper_bound、stride は全て省略可能であり、省略時には lower_bound には 0 が指定されたものとして、upper_bound にはタプルの長さ（len(タプル)）が指定されたものとして扱われる（stride に正数を指定した場合。stride に負数を指定すると、lower_bound がタプル末尾を、upper_bound がタプル先頭を指すようになる）。stride が省略されたときには「1」が指定されたものとして扱われる。

lower_bound と upper_bound は「タプル [lower_bound]」「タプル [upper_bound]」のようにインデックス指定したタプルの要素と同じものを表す。「タプル [lower_bound:upper_bound]」と開始位置と終了位置を指定したとき、上でも述べたように lower_bound はスライスの開始位置の要素を、upper_bound は終了位置の要素を表すが、実際には upper_bound で指定される要素が取り出されることはない。つまり、「タプル [0:5]」はタプルの 0、1、2、3、4 番目の要素を取り出すことになる（上述したように stride を省略したときには「1」が指定されたものとして扱われるので、これは「タプル [0:5:1]」と指定したとの同じ）。

以下に例を示す。

```
mytuple = tuple(range(10))
print(mytuple)  # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

s = mytuple[0:5]  # タプルの0～4番目の要素
print(s)  # (0, 1, 2, 3, 4)
```

upper_bound（終了位置）を省略した場合には、タプルの末尾要素までスライスに含まれる。以下に例を示す。

```
s = mytuple[2:]  # 2番目以降の全要素
print(s)  # (2, 3, 4, 5, 6, 7, 8, 9)
```

lower_bound（開始位置）を省略した場合には、タプルの先頭要素を示すインデックス 0 が指定されたものとして扱われる。以下に例を示す。

```
s = mytuple[:5]  # タプルの0～4番目の要素
print(s)  # (0, 1, 2, 3, 4)
```

lower_bound と upper_bound（と stride）を省略したときには、タプルの全要素がスライスに含まれる。以下に例を示す。

```
s = mytuple[:]  # タプルの全要素
print(s)  # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

stride（増分）を指定した場合には、lower_bound ≦ idx ＜ upper_bound の範囲に含まれる idx が「idx ＝ lower_bound ＋ stride×n」という式で計算されて（n は 0 以上の整数）、その idx に対応する要素が、取り出されるスライスの要素となる。以下に例を示す。

```
s = mytuple[::2]  # 0、0+1×2=2、0+2×2=4、……の要素（偶数インデックス）
print(s)  # (0, 2, 4, 6, 8)
```

この例では、lower_bound と upper_bound が省略されているが、stride が「2」と指定されている。そのため、lower_bound と upper_bound の間で「idx ＝ 0 ＋ n×2」（n は 0 以上の整数）で計算されるインデックスに対応する要素が取り出される（idx が lower_bound と upper_bound の範囲外になったところで計算は終了する）。この範囲に含まれる idx は 0、2、4、6、8 となるのでこれらのインデックスに対応する要素である 0、2、4、6、8 が取り出される。

以下は lower_bound と upper_bound を指定したものだ。lower_bound には 0 が、upper_bound には 5 が指定されているので、インデックス 0、2、4 に対応する要素が取り出されている。

```
s = mytuple[0:5:2]  # 0、0+1×2=2、0+2×2=4がインデックスの要素
print(s)  # (0, 2, 4)
```

スライスに負数を指定した場合

スライスに負の整数を指定した場合は、インデックスに負数を指定した場合と同様、タプル末尾からのインデックスが指定されたものとして扱われる。注意が必要なのは、lower_bound、upper_bound、stride を省略した場合、それらはそれぞれ 0、タプルの長さ、1 を指定されたものとして扱われる点だ。

つまり、末尾から先頭の要素を取り出すつもりで以下のようなコードを書いても、それは想定とは意味が異なり、末尾の要素から末尾の要素＋ 1 の範囲に含まれる要素を取り出すことになる（つまり、末尾要素のみを要素とするタプルとなる）。

```
s = mytuple[-1:]  # 末尾要素のみで構成されるタプル
print(s)  # (9,)
```

以下の例も同様だ。ここでは lower_bound に「-5」を、upper_bound に「-2」を指定しているが、これは lower_bound に「5」を、upper_bound に「8」を指定したのと同じ結果になる。「-5」は末尾から 5 つ目の要素を意味するもので、末尾から 5 つ目とはインデックスで表すと 9 → 8 → 7 → 6 → 5、すなわちインデックス 5 を表しているからだ（「-2」も同様）。

```
s = mytuple[-5:-2]  # s = mytuple[5:8]と同じ
print(s)  # (5, 6, 7)
```

そのため、lower_bound と upper_bound の値を逆にすると、要素が存在しなくなり、空のタプルが得られる。

```
s = mytuple[-2:-5]  # s = mytuple[8:5]と同じ
print(s)  # ()
```

これは「lower_bound ≦ idx < upper_bound」すなわち「8 ≦ idx < 5」の範囲に含まれる idx が存在しないからだ。

stride（増分）に負値を与えると、範囲の計算が切り替わり「5 < idx ≦ 8」の範囲に含まれるインデックスに対応する要素が得られる。

```
s = mytuple[-2:-5:-1]  # s = mytuple[8:5:-1]と同じ
print(s)  # (8, 7, 6)
```

ただし、stride（増分）が -1 なので、要素が取り出されるのは 8、7、6 と逆順になっている点に注意すること。

また、lower_bound または upper_bound の指定を省略して、stride に負値を指定した場合には、省略した lower_bound はタプルの末尾を、upper_bound はタプルの先頭を表すように調整される。

以下に例を示す。

```
s = mytuple[::-1]  # 全要素を逆順に並べたタプル
print(s)  # (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

この例では、lower_bound と upper_bound の両方が省略し、stride に負値を指定しているので、タプルの末尾から先頭へと向かって、要素が順に取り出されている。

辞書を作成するには

Python で辞書を作成するには幾つかの方法がある。それらの方法と辞書を作成する際に注意する点を紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 03 月 22 日）

```
# 波かっこを使う
d = {}  # 空の辞書
print(d)  # {}

d = {'key0': 0, 'key1': 1}
print(d)  # {'key0': 1, 'key1': 1}

# キーにはハッシュ可能なオブジェクトとしなければならない
key = [0, 1, 2]  # リストはハッシュ不可能
value = 102
d = {key: value}  # TypeError

key = (0, 1, 2)  # このタプルはハッシュ可能
d = {key: value}  # OK
print(d)  # {(0, 1, 2): 102}

key = ([0, 1, 2], 3, 4)  # このタプルはハッシュ不可能
d = {key: value}  # TypeError

# dict関数を使う
d = dict()  # 空の辞書
print(d)  # {}

# dict関数にキーワード引数を指定する
d = dict(key0=0, key1=1)
print(d)  # {'key0': 0, 'key1': 1}

# Pythonのマッピング型（辞書的なオブジェクト）を指定する
d2 = dict({'value2': 2, 'value3': 3})
print(d2)  # {'value2': 2, 'value3': 3}

# dict関数に2要素の反復可能オブジェクトを要素とする反復可能オブジェクトを渡す
l0 = ['key0', 0]
t0 = ('key1', 1)
l1 = [l0, t0]
print(l1)  # [['key0', 0], ('key1', 1)]
d = dict(l1)
print(d)  # {'key0': 0, 'key1': 1}

# 内包表記を使う方法
d = {f'key{x}': x for x in range(2)}
print(d)  # {'key0': 0, 'key1': 1}
```

辞書

Python における辞書とは「キー」と「値」の組を 0 個以上含むオブジェクトのことで、その要素（キーと値の組）は「{ キー : 値 }」のようにキーとそのキーに対応する値をコロンで区切ったものを波かっこ「{}」で囲んで表現される。

辞書を作成するには幾つかの方法がある。

- 波かっこで囲んで、キーと値の組を 0 個以上記述する
- dict 関数を使う
- 内包表記を使う

以下ではこれらの方法を紹介する。

波かっこを使った辞書の作成

波かっこ「{}」を使って辞書を作成するには、キーと値の組をコロンで区切ったものを 0 個以上波かっこの中に記述する。キーと値の組が複数あるときには、それらをカンマで区切る。

以下に例を示す。

```
d = {}  # 空の辞書
print(d)  # {}

d = {'key0': 0, 'key1': 1}
print(d)  # {'key0': 1, 'key1': 1}
```

最初の例は波かっこのみで、キーと値の組がない。この場合は要素（キーと値の組）を持たない空の辞書が作成される。

次の例では、'key0' という文字列キーとその値「0」、'key1' という文字列キーとその値「1」を格納する辞書を作成している。

辞書のキーはハッシュ可能でなければならない

注意が必要なのは、辞書のキーはハッシュ可能でなければならないことだ。文字列や数値はハッシュ可能だが、リストや集合はハッシュ可能ではない（要素が変更されることのない frozenset はハッシュ可能）。タプルはその要素を変更できないが、タプルの要素の要素が変更可能な場合にはハッシュ可能ではない（リストを格納するタプルはハッシュ不可能）。

以下に例を示す。

```
key = [0, 1, 2]  # リストはハッシュ不可能
value = 102
d = {key: value}  # TypeError
```

この例では、整数を要素とするリストをキーとして辞書を作成しようとしている。しかし、リストはハッシュ可能ではないので、これをキーとすることはできない。そのため、辞書を作成しようとしたところで TypeError 例外が発生する。

一方、以下の例では整数を要素とするタプルをキーにしようとしている。

```
key = (0, 1, 2)  # このタプルはハッシュ可能
d = {key: value}  # OK
print(d)  # {(0, 1, 2): 102}
```

整数はハッシュ可能であり、これを要素とするタプルもハッシュ可能なので、これはキーとして使用できる。そのため、問題なく辞書を作成できる。

ハッシュ可能でないリストを要素とするタプルをキーにしようとするとどうなるだろう。

```
key = ([0, 1, 2], 3, 4)  # このタプルはハッシュ不可能
d = {key: value}  # TypeError
```

この場合は、やはり、辞書を作成しようとしたところで TypeError 例外が発生する。

dict 関数を使った辞書の作成

dict 関数を使う場合には、引数の渡し方として、以下のように幾つかの方法がある。

- 引数を指定しない
- キーと値の組を「キー＝値」というキーワード引数の形で渡す
- Python のマッピングオブジェクト（辞書と同様に使えるオブジェクト）を渡す
- 「キー」「値」を要素とする反復可能オブジェクトを要素とする反復可能オブジェクトを渡す

最後の方法は文にすると分かりにくいがコードを見ればその意味が分かるはずだ。以下ではこれらを順番に見ていこう。

```
d = dict()  # 空の辞書
print(d)  # {}
```

引数を指定しないで dict 関数を呼び出したときには空の辞書が作成される。これは「d = {}」のように波かっこのみを使って辞書を作成した場合と同じだ。

キーワード引数を使って「キー＝値」のように引数を指定した場合の例を以下に示す。

```
d = dict(key0=0, key1=1)
print(d)  # {'key0': 0, 'key1': 1}
```

この場合、キーワード引数「key0=0」の「key0」の部分が文字列キーとなり、「0」のようにキーワード引数の値として指定したものがその値となる。

マッピングオブジェクトを指定する例を以下に示す。

```
d2 = dict({'value2': 2, 'value3': 3})
print(d2)  # {'value2': 2, 'value3': 3}
```

この例では、マッピングオブジェクトとして辞書を指定している。この場合は、指定したマッピングオブジェクトと同じ要素（キーと値の組）を持つ辞書が作成される。

要素を 2 つ持つ反復可能オブジェクトを要素とする反復可能オブジェクトを渡すと書くと、分かりにくいが以下のような反復可能オブジェクトを渡すと考えればよい。

```
[[キー0, 値0], (キー1, 値1), {キー2, 値2}, ……]
```

外側の反復可能オブジェクトの要素がリスト、タプル、集合になっている点に注目しよう。反復可能オブジェクトであれば、何でもよい（外側の反復可能オブジェクトもリストである必要はない）。

このような反復可能オブジェクトから dict 関数は次のような辞書を作成してくれる。

```
{キー0: 値0, キー1: 値1, キー2: 値2, ……}
```

以下に例を示す。

```
l0 = ['key0', 0]
t0 = ('key1', 1)
l1 = [l0, t0]
print(l1)  # [['key0', 0], ('key1', 1)]
d = dict(l1)
print(d)  # {'key0': 0, 'key1': 1}
```

この場合は、l0 と t0 はどちらも 2 つの要素を持つ反復可能オブジェクト（リストとタプル）である。l1 はこれらを要素とする反復可能オブジェクト（リスト）になっている。そして、これを dict 関数に渡すと、l1（外側の反復可能オブジェクト）の要素を反復して、l0 と t0 が順に取り出され、1 つ目の要素（l0 なら 'key0'、t0 なら 'key1'）がキーとして、2 つ目の要素（l0 なら 0、t0 なら 1）がそのキーに対応する値として辞書が作成される。

内側の反復可能オブジェクトに辞書を含める場合には注意が必要だ。

```
d = dict([{'key0': 0, 'key1': 1}])
print(d)  # {'key0': 'key1'}
```

キーと値の組で辞書の 1 つの要素となり、単純に上のように書くとキーの部分だけが取り出されてしまう。

マッピングオブジェクトとキーワード引数を混在させることや（dict({'key0': 0}, key1=1)）、反復可能オブジェクトとキーワード引数を混在させること（dict([('key0', 0)], key1=1)）も可能だが、マッピングオブジェクトと反復可能オブジェクトの両方（とキーワード引数）を一度に渡すこと（dict([('key0', 0)], {'key1': 1}, key2=2)）はできない。また、混在させるときにはキーワード引数をマッピングオブジェクトまたは反復可能オブジェクトの後に置く必要がある。

内包表記を使用した辞書の作成

辞書の内包表記は次のような形式で記述する。

```
{キーを算出する式: 値を算出する式 for 変数 in 反復可能オブジェクト}
```

「キーを算出する式」と「値を算出する式」では、「反復可能オブジェクト」から取り出した値を代入する「変数」の値を使用することになるだろう。以下に例を示す。

```
d = {f'key{x}': x for x in range(2)}
print(d)  # {'key0': 0, 'key1': 1}
```

この例では「キーを算出する式」がf文字列「f'key{x}'」に、「値を算出する式」は「x」になっている。「変数」である「x」には「反復可能オブジェクト」である「range(2)」から順に値（0、1）が代入され、その値を使って「'key0': 0」「'key1': 1」という要素を持つ辞書が作成されている。なお、内包表記を使用した辞書の作成については次の「内包表記で辞書を作成するには」も参照のこと。

内包表記で辞書を作成するには

辞書内包表記を使って、辞書を作成する方法と、zip 関数と辞書内包表記を組み合わせる方法、注意点などを紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 07 月 13 日）

```
# 辞書内包表記の基本型
d = {k: k ** 2 for k in range(5)}
print(d)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# 辞書内包表記とzip関数との組み合わせ
articles = [
    'リストの内包表記と「if」を組み合わせるには',
    '内包表記でリストを作成するには',
    'バイナリファイルの読み書きまとめ',
    'テキストファイルの読み書きまとめ'
]

urls = [
    'https://www.atmarkit.co.jp/ait/articles/2107/06/news020.html',
    'https://www.atmarkit.co.jp/ait/articles/2106/29/news021.html',
    'https://www.atmarkit.co.jp/ait/articles/2106/22/news022.html',
    'https://www.atmarkit.co.jp/ait/articles/2106/15/news016.html'
]

d = {title: url for title, url in zip(articles, urls)}
for title, url in d.items():
    print(f'{title}: {url[-20:]}')
# 出力結果:
#リストの内包表記と「if」を組み合わせるには: 2107/06/news020.html
#内包表記でリストを作成するには: 2106/29/news021.html
#バイナリファイルの読み書きまとめ: 2106/22/news022.html
#テキストファイルの読み書きまとめ: 2106/15/news016.html

# 同じことはdict関数でもできる
d = dict(zip(articles, urls))
for title, url in d.items():
    print(f'{title}: {url[-20:]}')  # 出力結果は上に同じ

# 反復可能オブジェクトから得た値を加工するには内包表記を使う
d = {t: u[-20:] for t, u in zip(articles, urls) if '内包表記' in t}
for title, url in d.items():
    print(f'{title}: {url}')
# 出力結果:
#リストの内包表記と「if」を組み合わせるには: 2107/06/news020.html
#内包表記でリストを作成するには: 2106/29/news021.html
```

辞書の内包表記の基本型

内包表記は辞書の作成にも使える。以下にその基本構文を示す。

```
d = {キー: 値 for キー／値の計算で使用する変数 in 反復可能オブジェクト}
```

辞書内包表記ではリスト内包表記とは異なり、外側を波かっこ「{}」で囲む。また、その中に新しく作成する辞書の要素を「キー：値」として書く。辞書の要素（キーと値）を計算する際には、上の構文に示した「キー／値の計算で使用する変数」に「反復可能オブジェクト」から要素が 1 つずつ渡されるので、その値を使って「キー」または「値」を計算する。

以下にシンプルな例を示す。

```
d = {k: k ** 2 for k in range(5)}
print(d)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

この例では、range オブジェクトから変数 k に整数値が順次渡され、その値を使って、辞書のキー（k の値と同値）とその値（k の値を二乗したもの）を計算している。

zip 関数との組み合わせ

内包表記と zip 関数を組み合わせることで、2 つのリストの一方をキーに、もう一方をその値とする辞書を作成することも可能だ。以下に例を示す。

```
articles = [
    'リストの内包表記と「if」を組み合わせるには',
    '内包表記でリストを作成するには',
    'バイナリファイルの読み書きまとめ',
    'テキストファイルの読み書きまとめ'
]

urls = [
    'https://www.atmarkit.co.jp/ait/articles/2107/06/news020.html',
    'https://www.atmarkit.co.jp/ait/articles/2106/29/news021.html',
    'https://www.atmarkit.co.jp/ait/articles/2106/22/news022.html',
    'https://www.atmarkit.co.jp/ait/articles/2106/15/news016.html'
]

d = {title: url for title, url in zip(articles, urls)}
for title, url in d.items():
    print(f'{title}: {url[-20:]}')
# 出力結果:
#リストの内包表記と「if」を組み合わせるには: 2107/06/news020.html
#内包表記でリストを作成するには: 2106/29/news021.html
#バイナリファイルの読み書きまとめ: 2106/22/news022.html
#テキストファイルの読み書きまとめ: 2106/15/news016.html
```

articles は本連載の記事タイトルを要素とするリストで、urls は各記事の URL を要素とするリストとなっている。上に示した内包表記「{title: url for title, url in zip(articles, urls)}」は、zip 関数でこれらのリストをまとめて、そこから「記事タイトル : URL」という組を要素とする辞書を作成している。

ただし、これだけであれば、dict 関数を使うとよりシンプルに書ける。

```
d = dict(zip(articles, urls))
for title, url in d.items():
    print(f'{title}: {url[-20:]}')
```

dict 関数は「2 つの要素からなる反復可能オブジェクトを要素とする反復可能オブジェクト」を受け取れる。例えば、「[['k1', 'v1'], ['k2', 'v2']]」のようなリストがそうだ。この場合、dict 関数は外側のリストの要素（['k1', 'v1']）の先頭要素をキーとして、次の要素を値として、辞書を作成する。つまり、今の例であれば、{'k1': 'v1', 'k2': 'v2'} という辞書を作成する。

```
sample_list = [['k1', 'v1'], ['k2', 'v2']]
d2 = dict(sample_list)
print(d2)  # {'k1': 'v1', 'k2': 'v2'}
```

zip 関数の戻り値は、引数に受け取った反復可能オブジェクトと同一インデックス位置にある要素を組としたタプルを返すイテレータであり、これを dict 関数に渡すと、辞書内包表記で zip 関数を使った場合と同じことができるということだ。

では、常に dict 関数を使えばよいかというと、そうでもない。元の反復可能オブジェクトの値を加工したり、フィルタリングしたりするには内包表記を使うことになるだろう。以下に例を示す（ループ変数の名前を 1 文字としている点に注意）。

```
d = {t: u[-20:] for t, u in zip(articles, urls) if '内包表記' in t}
for title, url in d.items():
    print(f'{title}: {url}')
# 出力結果:
#リストの内包表記と「if」を組み合わせるには: 2107/06/news020.html
#内包表記でリストを作成するには: 2106/29/news021.html
```

この例では、内包表記の if 句を利用して、記事リスト（articles）の要素の中で「内包表記」という語が含まれているものだけをキーとするようにフィルタリングを行い、その値の URL については「url[-20:]」のようにして 4 桁の整数値以降だけを抽出するように元のリストの要素を加工している。なお、if 句については「リストの内包表記と「if」を組み合わせるには」も参照されたい。

注意点

最後に注意点だが、辞書のキーにできるのはハッシュ可能なオブジェクトのみであることは覚えておこう。リストや辞書のようにその内容が変更可能なオブジェクトは、ハッシュ可能なオブジェクトではなく、キーとすることはできない。

```
klist = [[1, 2], [3, 4]]
vlist = [['foo', 'bar'], ['bar', 'baz']]

d = {k: v for k, v in zip(klist, vlist)}  # TypeError: unhashable type: 'list'
```

また、タプルのようなイミュータブル（変更不可能）なオブジェクトであっても、その要素がミュータブル（変更可能）な場合がある。それらのオブジェクトはハッシュ可能ではないので、やはりキーにはできない。

```
ktuple1 = ((1, 'foo'), (3, 'bar'))  # 変更可能な要素は含まれていない
ktuple2 = ((1, ['foo', 'bar']), (2, ['bar', 'baz']))  # 変更可能な要素を含む

vlist = [['foo', 'bar'], ['bar', 'baz']]  # 値は変更可能でも構わない

d = {k: v for k, v in zip(ktuple1, vlist)}
print(d)  # {(1, 'foo'): ['foo', 'bar'], (3, 'bar'): ['bar', 'baz']}

d = {k: v for k, v in zip(ktuple2, vlist)}  # TypeError: unhashable type: 'list'
```

辞書の要素を取得するには

辞書からキーに対応した値を取得するには幾つかの方法がある。それらの方法と注意点を紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 03 月 28 日）

```
d = {'key0': 0, 'key1': 1}

# 辞書から値を取得
# キーを指定
val = d['key0']
print(val)  # 0

# 存在しないキーを指定すると例外
val = d['key100']  # KeyError

# 例外を発生させずに値を取得するにはgetメソッドを使う
val = d.get('key0', 'not found')
print(val)  # 0

val = d.get('key100', 'not found')  # OK
print(val)  # not found

# popメソッドは引数に指定したキーの値を取得して、そのキー／値を辞書から削除
val = d.pop('key1')
print(val)  # 1
print(d)  # {'key0': 0}

# popメソッドで存在しないキーを指定した場合
val = d.pop('key100')  # KeyError

val = d.pop('key100', 'not found')
print(val)  # not found

# popitemメソッドは辞書の末尾からキー／値の組を戻し、それを辞書から削除
d = {'foo': 0, 'bar': 1, 'baz': 2}
print(d)  # {'foo': 0, 'bar': 1, 'baz': 2}

k, v = d.popitem()
print(f'key: {k}, val: {v}')  # key: baz, val: 2
print(d)  # {'foo': 0, 'bar': 1}
```

辞書から値を取得する方法

辞書から値を取得するには以下のような方法がある（ここでいう「値」とは、辞書に含まれるキーに対応する値のこと）。

- 辞書に対して角かっこ「[]」でキーを囲む
- get メソッドの引数にキーを指定する
- pop メソッドの引数にキーを指定する（対応するキー／値は辞書から削除される）
- popitem メソッドで辞書の末尾要素のキーと値を取得する（末尾要素は辞書から削除される）

以下では、これらについて簡単に見ていく。

角かっこにキーを指定

辞書に対して角かっこにキーを指定することで、その値を取得できる。これはリストやタプルで角かっこの中にインデックスを指定して、そのインデックス位置にある値を取得するのに似ている。

以下に例を示す。

```
d = {'key0': 0, 'key1': 1}

val = d['key0']
print(val)  # 0
```

この例では辞書 d には 'key0' と整数 0、'key1' と整数 1 という 2 つのキー／値の組が含まれている。そして、辞書 d に対して「d['key0']」のように角かっこで囲んでキー 'key0' を指定すると、その値である整数 0 が得られている。

これは辞書からキーに関連付けられた値を取得する最も一般的な方法といえる。ただし、注意点がある。それは存在しないキーを指定すると例外が発生することだ。以下に例を示す。

```
val = d['key100']  # KeyError
```

これに対して、次に紹介する get メソッドでは存在しないキーを指定した場合にも例外を発生させない。

get メソッドの引数にキーを指定

get メソッドの構文は次のようになっている。

```
get(key[, default])
```

key には取得したい要素のキーを指定する。default は指定したキーが存在しなかった場合に get メソッドが返す値を指定する。省略時には None が指定されたものとされる。これにより、get メソッドでは指定したキーが辞書になくても、KeyError 例外を発生させることなく default の値が返送される。

以下に例を示す（辞書 d は先ほどと同様）。

```
val = d.get('key0', 'not found')
print(val)  # 0

val = d.get('key100', 'not found')  # OK
print(val)  # not found
```

最初の例では存在するキーである 'key0' を指定している。そのため、その値である整数 0 が返されている。

次の例では、存在しないキー 'key100' を指定している。また、引数 default には 'not found' を指定している。よって、ここでは get メソッドの戻り値は 'not found' になる。

多くの場合は、ここまでに説明した 2 つの方法で辞書の値を取り出すが、pop メソッドと popitem メソッドでも辞書から値を取り出せる。

pop メソッドと popitem メソッド

pop メソッドと popitem メソッドの特徴を以下に簡単にまとめる。

- pop メソッド：引数に指定したキーの値を戻り値とし、そのキー／値の組を辞書から削除する
- popitem メソッド：辞書の末尾要素のキー／値の組を戻り値として、その組を辞書から削除する

どちらのメソッドもキー／値の組を辞書から削除することには注意すること。

また、pop メソッドは次のような構文になっている。

```
pop(key[, default])
```

get メソッドとは異なり、default を省略したときには None が指定されたものとはならない点には注意が必要だ。つまり、default を指定しなかったときには、指定したキーが辞書に含まれていなければ、KeyError 例外が発生する。指定したキーが存在すれば、その値が戻り値となる。

以下に例を示す。

```
val = d.pop('key1')
print(val)  # 1
print(d)  # {'key0': 0}
```

この例では存在するキーを指定している。よって、指定したキー 'key1' に対応する値が返されていることと、辞書 d からは対応する要素（キー／値の組）が削除されていることに注目されたい。

存在しないキーを指定した場合の例を以下に示す。

```
val = d.pop('key100')  # KeyError

val = d.pop('key100', 'not found')
print(val)  # not found
```

default を指定するかしないかで pop メソッドの振る舞いが異なるので、pop メソッドを使う場合には注意しよう。

popitem メソッドは辞書の末尾からキー／値の組を戻し、それを辞書から削除する。引数はない。Python 3.7 以降では、popitem メソッドは辞書に追加した順序とは逆順（LIFO：Last in First Out）でキー／値の組を取り出す。

以下に例を示す。ここでは辞書には 3 つの要素が格納されている。

```
d = {'foo': 0, 'bar': 1, 'baz': 2}
print(d)  # {'foo': 0, 'bar': 1, 'baz': 2}

k, v = d.popitem()
print(f'key: {k}, val: {v}')  # key: baz, val: 2
print(d)  # {'foo': 0, 'bar': 1}
```

辞書 d には 3 つの要素があり、最後の要素はキー 'baz' と整数 2 の組となっている。この状態で popitem メソッドを呼び出すとそれらが取り出されると同時に辞書から削除されていることが分かる。

なお、popitem メソッドは辞書が格納する全ての要素について何らかの処理をしながら、最後に辞書を空にしたいといった場合に役立つ。典型的には次のようなコードになるだろう。

```
d = {'foo': 0, 'bar': 1, 'baz': 2}

while d:
    k, v = d.popitem()
    # k, vを使って何かする
    print(f'key: {k}, value: {v}')
```

辞書の要素を削除したり追加したり変更したりするには

辞書から要素を削除するには del 文や clear メソッドなどを使える。また、辞書の要素を追加／上書きするには代入文や update メソッド、「|=」演算子を使える。これらの方法を一覧する。

かわさきしんじ，Deep Insider 編集部（2023 年 04 月 04 日）

```
# 辞書の要素の削除
d = {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3}
print(d)  # {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3}

# del文で指定してキー／値の組を削除
del d['key1']
print(d)  # {'key0': 0, 'key2': 2, 'key3': 3}

# clearメソッドで辞書の全要素を削除
d.clear()
print(d)  # {}

# popメソッドは指定されたキーの値を返し、そのキー／値の組を辞書から削除
d = {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3}
item = d.pop('key0')
print(item)  # 0
print(d)  # {'key1': 1, 'key2': 2, 'key3': 3}

# popitemメソッドは末尾からキー／値を取り出し、その組を辞書から削除
item = d.popitem()
print(item)  # ('key3', 3)
print(d)  # {'key1': 1, 'key2': 2}

# 要素（キー／値の組）の追加と上書き
d = {'key0': 0, 'key1': 1}

# 辞書に要素を追加
# 存在しないキーを指定してその値を代入
d['key2'] = 2
print(d)  # {'key0': 0, 'key1': 1, 'key2': 2}

# 存在するキーを指定するとその値が上書きされる
d['key1'] = 10
print(d)  # {'key0': 0, 'key1': 10, 'key2': 2}

# updateメソッドで辞書を更新
# 存在するキーの値は上書きされ、存在しないキーとその値は追加される
d = {'key0': 0, 'key1': 1, 'key2': 2}

d.update({'key1': 11, 'key3': 3})
print(d)  # {'key0': 0, 'key1': 11, 'key2': 2, 'key3': 3}

d.update([('key1', 1), ('key3', 33)])  # 反復可能オブジェクト
```


→目次に戻る

```
print(d)  # {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 33}

d.update(key2=22, key4=4)  # キーワード引数を使う場合
print(d)  # {'key0': 0, 'key1': 1, 'key2': 22, 'key3': 33, 'key4': 4}

d2 = {'key4': 44, 'key5': 5}
d |= d2
print(d)  # {'key0': 0, 'key1': 1, 'key2': 22, 'key3': 33, 'key4': 44, 'key5': 5}
```

辞書の要素を削除したり追加したり変更したりする方法

辞書からその要素を削除するには次の方法がある（本稿ではキーと値の組のことを「要素」と表記する）。

- del 文を使う（指定したキーとその値の組を削除）
- clear メソッドを使う（辞書の全要素を削除）
- pop メソッド／ popitem メソッドを使う

また、辞書に要素を追加したり、既存のキーの値を変更したりするには次の方法がある。

- 存在しないキーを指定して辞書に値を代入する（要素の追加）
- 既存のキーを指定して辞書に値を代入する（要素の上書き）
- update メソッドに辞書などを指定する（要素の追加または上書き）
- 累算代入演算子「|=」を使用する

以下ではこれらについて見ていくことにする。

辞書の要素の削除

辞書の特定の要素を指定するには、del 文で辞書に対してその要素のキーを指定する。

以下に例を示す。

```
d = {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3}
print(d)  # {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3}

del d['key1']
print(d)  # {'key0': 0, 'key2': 2, 'key3': 3}
```

この例では {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3} という 4 つの要素を持つ辞書を作成し、その後、del 文でキー「'key1'」を指定して辞書の要素を削除している。最後の行では辞書から指定したキーの要素が削除されているかを確認している。

なお、存在しないキーを削除しようとすると、KeyError 例外が発生する。

```
del d['key100']  # KeyError
```

辞書の全ての要素をまとめて削除するときには、clear メソッドを使用する。以下に例を示す。

```
d.clear()
print(d)  # {}
```

先ほどの del 文で辞書 d には {'key0': 0, 'key2': 2, 'key3': 3} という要素が格納されていた。そこで clear メソッドを呼び出したので、その内容がまとめて削除され、辞書が空になった。

辞書の要素を削除するといった場合、通常はここまでに見てきた 2 つを使うことになるだろう。しかし、辞書には要素を削除する結果をもたらす pop メソッドと popitem メソッドもある。これらについても簡単に触れておこう（これらについては「辞書の要素を取得するには」でも触れているので、そちらも参照されたい）。

- pop メソッド：引数に指定したキーの要素があれば、その値を返し、その要素を辞書から削除する
- popitem メソッド：辞書の末尾にある要素のキーと値をタプルとして返し、その要素を辞書から削除する

以下に例を示す。

```
d = {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3}
item = d.pop('key0')
print(item)  # 0
print(d)  # {'key1': 1, 'key2': 2, 'key3': 3}
```

ここでは辞書 d は {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3} となっている。上のコードでは、キーとして 'key0' を指定して pop メソッドを呼び出している。そのため、戻り値は 'key0' の値である「0」となり、キー／値の組は辞書から削除され {'key1': 1, 'key2': 2, 'key3': 3} になる。なお、pop メソッドの第 2 引数には、指定したキーが辞書になかったときの戻り値も指定できる。第 2 引数を指定した場合は、キーが辞書に存在しなければ、第 2 引数の値が戻り値となる。第 2 引数を指定しなかった場合、キーが辞書に存在しなければ KeyError 例外が発生する。

popitem メソッドの使用例は以下の通り。

```
item = d.popitem()
print(item)  # ('key3', 3)
print(d)  # {'key1': 1, 'key2': 2}
```

先の例で pop メソッドを呼び出したことで、辞書は {'key1': 1, 'key2': 2, 'key3': 3} となっている。この状態で popitem メソッドを呼び出すと、末尾の要素のキーである 'key3' とその値である「3」が戻り値となり、その要素が辞書から削除されていることが分かるはずだ。

辞書の要素の追加／上書き

辞書に要素を追加したり、既存のキーの値を上書き（変更）したりするには以下の方法がある。

- 要素の追加：辞書に存在しないキーを指定して、値を代入
- 要素の上書き：辞書に存在するキーを指定して、値を代入

キーが辞書に存在していれば上書き、存在していなければ追加となる。上では「代入」と書いたが、これは代入文でなくとも update メソッドや累算代入演算子「|=」を使う場合でも同様だ。

以下に代入文で辞書に要素を追加する例と、既存のキーの値を上書きする例を示す。まずは要素を追加する例から。

```
d = {'key0': 0, 'key1': 1}

d['key2'] = 2
print(d)  # {'key0': 0, 'key1': 1, 'key2': 2}
```

この例では辞書 d の内容は {'key0': 0, 'key1': 1} になっている。ここで、辞書に存在しないキー 'key2' を指定して代入を行うことで、辞書にキーを 'key2'、その値を 2 とする要素が追加される。

既存のキーの値を上書きする例を以下に示す。

```
d['key1'] = 10
print(d)  # {'key0': 0, 'key1': 10, 'key2': 2}
```

ここでは既存のキー 'key1' を指定して代入文を実行しているので、その値が上書きされている。

単一の要素を追加したり、値を上書きするのであれば代入文で十分だが、複数の要素を一括して追加したり、複数の要素の値を上書きしたりしたいのであれば update メソッドを使うとよい。

update メソッドには幾つかの使い方がある。ここでは update メソッドを呼び出す辞書を d として表記する。

```
d.update({キーx: 値x, キーy: 値y, ……})
d.update([(キーx, 値x), (キーy, 値y), ……])
d.update(キーx=値x, キーy=値y, ……)
```

1 つ目の使い方は引数に辞書を渡すもの。このとき、辞書 d に存在しないキーが引数に指定した辞書にあれば、そのキーと値が追加されるし、辞書 d と引数に指定した辞書とで同じキーがあれば、辞書 d の値は上書きされる。これは他の 2 つの呼び出し方でも同様だ。

2 つ目の使い方は、キーとその値の組からなる反復可能オブジェクト（リストやタプルなど）を要素とする反復可能オブジェクトを update メソッドに渡すものだ。3 つ目の使い方はキーワード引数を指定するものだ。

以下に例を示す。言葉で説明すると分かりにくいがコードで見ればすぐに分かるはずだ。

```
d = {'key0': 0, 'key1': 1, 'key2': 2}

d.update({'key1': 11, 'key3': 3})
print(d)  # {'key0': 0, 'key1': 11, 'key2': 2, 'key3': 3}

d.update([('key1', 1), ('key3', 33)])  # 反復可能オブジェクト
print(d)  # {'key0': 0, 'key1': 1, 'key2': 2, 'key3': 33}

d.update(key2=22, key4=4)  # キーワード引数を使う場合
print(d)  # {'key0': 0, 'key1': 1, 'key2': 22, 'key3': 33, 'key4': 4}
```

Python 3.9 以降では累算代入演算子「|=」も使える。「|=」の右辺には辞書または [(キー x, 値 x), (キー y, 値 y), ……] のような（キーと値を要素とする反復可能オブジェクトを要素とする）反復可能オブジェクトを指定できる。

以下に例を示す。この場合も追加と上書きのどちらになるかは元の辞書と右辺の値による。

```
d2 = {'key4': 44, 'key5': 5}
d |= d2
print(d)  # {'key0': 0, 'key1': 1, 'key2': 22, 'key3': 33, 'key4': 44, 'key5': 5}
```

辞書のキーや値を反復したり
特定のキーや値が含まれているかを確認したりするには

辞書の keys ／ values ／ items メソッドを使ってビューオブジェクトを取得して、キーや値、それらの組を反復したり、特定のキーや値が辞書に含まれているかを確認したりする方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 04 月 11 日）

```
d = {'key0': 0, 'key1': 1, 'key2': 2}

# キーを反復
keys = d.keys()
print(keys)  # dict_keys(['key0', 'key1', 'key2'])

for key in keys:
    print(key)  # 'key0'、'key1'、'key2'が反復される

# 値を反復
values = d.values()
print(values)  # dict_values([0, 1, 2])

for value in values:
    print(value)  # 0、1、2が反復される

# キーと値を反復
items = d.items()
print(items)  # dict_items([('key0', 0), ('key1', 1), ('key2', 2)])

for key, value in items:
    print(key, ':', value)  # 「key0 : 0」「key1 : 1」「key2 : 2」

# キーの存在確認
existence = 'key4' in d.keys()
print(existence)  # False

existence = 'key1' in d
print(existence)  # True

# 値の存在確認
existence = 0 in d.values()
print(existence)  # True

existence = 0 in d
print(existence)  # False

# キー／値の組の存在確認
existence = ('key0', 0) in d.items()
print(existence)  # True
```


→目次に戻る

```
existence = ('key0', 1) in d.items()
print(existence)  # False

# ビューオブジェクトは動的
keys = d.keys()
print(keys)  # dict_keys(['key0', 'key1', 'key2'])

d['key3'] = 3
print(keys)  # dict_keys(['key0', 'key1', 'key2', 'key3'])

# キーは集合的な演算も可能
d2 = {'key0': 0, 'key1': 1, 'key2': 2}
print(d2)  # {'key0': 0, 'key1': 1, 'key2': 2}
keys2 = d2.keys()
print(keys2)  # dict_keys(['key0', 'key1', 'key2'])
print(keys)  # dict_keys(['key0', 'key1', 'key2', 'key3'])

result = keys > keys2  # keysはkey2を包含している
print(result)  # True

result = keys ^ keys2  # keysとkeys2の対称差を計算
print(result)  # {'keys3'}

result = d > d2  # TypeError
result = d ^ d2  # TypeError
```

辞書のキーや値を反復するには

辞書からは「辞書ビューオブジェクト」と呼ばれるオブジェクトを取得できる。これらを使って、キー／値／キーと値を要素とするタプルを反復できる。辞書ビューオブジェクトを得るには以下のメソッドを呼び出す。

- dict.keys メソッド：キーのビューを取得する
- dict.values メソッド：値のビューを取得する
- dict.items メソッド：キーと値のビューを取得する（ビューの要素は (キー , 値) というタプルになる）

取得したビューを使うと、上記メソッドで得られる要素（キー、値、それらを要素とするタプル）を反復したり、存在確認をしたりできる。

キーや値の反復

キーや値、それらを要素とするタプルを反復するには、上で述べた通り、keys ／ values ／ items メソッドを使用する。以下に例を示す。

```
d = {'key0': 0, 'key1': 1, 'key2': 2}

# キーを反復
keys = d.keys()
print(keys)  # dict_keys(['key0', 'key1', 'key2'])

for key in keys:
    print(key)  # 'key0'、'key1'、'key2'が反復される
```

この例では keys メソッドを用いて、キーのビューオブジェクトを取得している（dict_keys オブジェクト）。反復処理をサポートしているので、後は for 文でその値（キー）を取り出すだけだ。なお、インデックスやスライスによる特定要素の取り出しはサポートされない。

値やキー／値のタプルの反復についても同様だ。以下に例を示す。

```
# 値を反復
values = d.values()
print(values)  # dict_values([0, 1, 2])

for value in values:
    print(value)  # 0、1、2が反復される

# キーと値を反復
items = d.items()
print(items)  # dict_items([('key0', 0), ('key1', 1), ('key2', 2)])

for key, value in items:
    print(key, ':', value)  # 「key0 : 0」「key1 : 1」「key2 : 2」
```

items メソッドでは「(キー , 値)」というタプルの形で辞書の要素が格納されることは覚えておこう。

キーや値の存在確認

辞書ビューオブジェクトは in 演算子／ not in 演算子を使って、そのオブジェクトに特定の値が含まれているかを確認できる。以下に例を示す。

```
existence = 'key4' in d.keys()
print(existence)  # False
```

ここで例としている辞書 d は「{'key0': 0, 'key1': 1, 'key2': 2}」となっているので、'key4' というキーは含まれていない。そのため、上の「'key4' in d.keys()」は False となる。

なお、キーに関していえば、keys メソッドでキーのビューオブジェクトを取得しなくても、以下のように辞書に対して in 演算子や not in 演算子を適用しても特定のキーの存在確認を行える。

```
existence = 'key1' in d
print(existence)  # True
```

特定の値が辞書に含まれているかや、特定のキー／値の組が辞書に含まれているかを確認したいときにはビューオブジェクトを取得する必要があるので注意されたい。

以下は値の存在確認をする例だ。

```
existence = 0 in d.values()
print(existence)  # True
```

辞書の値には「0」があるので、上の「0 in d.values()」は True となる。しかし、何かの値が辞書に含まれているかを知るために、in 演算子を辞書に直接適用するのは上で述べたように間違いだ。

```
existence = 0 in d
print(existence)  # False
```

辞書の値として「0」があるかを調べようとして、上の例では「0 in d」としているが、これは辞書のキーに対する存在確認となるので、その結果は False となる。

これはキー／値の組（ここではタプル）でも同様だ。以下に例を示す。

```
existence = ('key0', 0) in d.items()
print(existence)  # True

existence = ('key0', 1) in d.items()
print(existence)  # False
```

ビューオブジェクトは動的

ビューオブジェクトは、それを取得した時点での辞書の状態を表す静的なものではなく、対象となる辞書の現在の状態を表す動的なものであることには注意しよう。

```
keys = d.keys()
print(keys)  # dict_keys(['key0', 'key1', 'key2'])

d['key3'] = 3
print(keys)  # dict_keys(['key0', 'key1', 'key2', 'key3'])
```

この例では、辞書 d に含まれているキーを表すビューオブジェクトを取得して、その状態を表示した後に、辞書に新しいキー「'key3'」とその値「3」を追加している。そして、その前に取得したキーのビューオブジェクトの内容を表示しているが、新しい辞書の状態がビューオブジェクトに反映されていることが分かる。

これは値のビューオブジェクト、キー／値のビューオブジェクトでも同様だ。

集合的な操作

辞書のキーには「辞書内でユニークであり、かつハッシュ可能」という特徴がある。このため、辞書のキーを集めたキーのビューオブジェクトは集合（Python の set オブジェクト）と同様であり、集合の抽象基底クラスである collections.abc.Set で定義されている操作を適用できる。

例えば、これまでに見てきた辞書 d に加えて、次のような辞書 d2 があったとする。

```
d2 = {'key0': 0, 'key1': 1, 'key2': 2}
print(d2)  # {'key0': 0, 'key1': 1, 'key2': 2}
keys2 = d2.keys()
print(keys2)  # dict_keys(['key0', 'key1', 'key2'])
print(keys)  # dict_keys(['key0', 'key1', 'key2', 'key3'])
```

辞書 d に対するキーのビューオブジェクト keys は「dict_keys(['key0', 'key1', 'key2', 'key3'])」で、辞書 d2 のキーのビューオブジェクト keys2 は「dict_keys(['key0', 'key1', 'key2'])」である。keys と keys2 とでは、keys に 'key3' があること以外は同じである。そのため、keys は keys2 を包含しているといえる。

このことは、集合演算でサポートされている > 演算子で調べられる。

```
result = keys > keys2  # keysはkey2を包含している
print(result)  # True
```

同様に、keys と keys2 の対称差は ^ 演算子で求められる。

```
result = keys ^ keys2  # keysとkeys2の対称差を計算
print(result)  # {'keys3'}
```

なお、「keys |= {'key100'}」のような操作を行うと、keys は上書きされてビューオブジェクトではなくなってしまうことには注意されたい（単なる set オブジェクトとなる）。

辞書自体を対象としてこうした演算を行うことはサポートされない。

```
result = d > d2  # TypeError
result = d ^ d2  # TypeError
```

キーの存在確認は辞書自体を対象としても行えるが、キーを対象として集合的な演算を行うにはキーのビューオブジェクトが必要になることは覚えておこう（そういう状況はそうそうないかもしれないが）。

集合を作成するには

Python で集合を作成するには波かっこや set 関数や内包表記を使う。それらの方法と注意点を簡単に紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 04 月 18 日）

```
# 波かっこを使って集合を作成
s = {0, 1, 2}  # 波かっこ内に集合の要素を記述
print(s)  # {0, 1, 2}

# 集合の要素は重複しない
s = {0, 0, 1, 2}
print(s)  # {0, 1, 2}

# 集合の要素には順序がない
s = {'foo', 'bar', 'baz'}
print(s)  # {'foo', 'baz', 'bar'}など

# 集合の要素はハッシュ可能である必要がある
s = {[0, 1], [2, 3]}  # TypeError

# set関数を使って集合を作成
s = set()  # 空の集合
print(s)  # set()

d = {}  # {}は空の集合ではなく、空の辞書を作成する
print(s == d)  # False
print(type(d))  # <class 'dict'>

s = set(['foo', 'bar', 'baz'])  # 集合の要素を含む反復可能オブジェクトを与える
print(s)  # {'foo', 'baz', 'bar'}

s = set('foo')  # 文字列は反復可能オブジェクト
print(s)  # {'f', 'o'}

# 集合内包表記
s = {x * 2 for x in range(4)}
print(s)  # {0, 2, 4, 6}
```

集合を作成するには

集合を作成するには幾つかの方法がある。

- 集合の要素を波かっこ「{}」内にカンマで区切って並べる
- set 関数を使用する
- 集合内包表記を使用する

以下ではこれらの方法を見ていこう。

波かっこを使って集合を作成する

集合を作成する最も簡単な方法は波かっこの内部に、その要素をカンマ区切りで並べていくことだ。以下に例を示す。

```
s = {0, 1, 2}  # 波かっこ内に集合の要素を記述
print(s)  # {0, 1, 2}
```

この例では集合の要素となる 3 つの整数値をカンマ区切りで並べている。注意したいのは、集合では同一の要素が重複することがない点だ。以下に例を示す。

```
s = {0, 0, 1, 2}
print(s)  # {0, 1, 2}
```

ここでは、整数値「0」を 2 つ波かっこの中に記述しているが、同一の要素が重複することがないので、整数値「0」は集合 s には 1 つだけ存在するようになる。

また、集合の要素には順序もない。集合の要素を取り出そうというときにどんな順番でそれらが出現するかは分からない。例えば、以下の集合 s は「{'foo', 'bar', 'baz'}」という順番で要素を並べているが、print 関数でその内容を表示したときに作成時とは異なる順番で要素が表示されるかもしれない（し、作成時と同じ順番で表示されるかもしれない）。

```
s = {'foo', 'bar', 'baz'}
print(s)  # {'foo', 'baz', 'bar'}など
```

最後に集合の要素はハッシュ可能である必要がある。文字列や数値はハッシュ可能だが、リストは変更可能でありハッシュ可能ではない。そのため、リストを集合の要素とすることはできない。

以下に例を示す。

```
s = {[0, 1], [2, 3]}  # TypeError
```

リストを集合の要素としようとすると、TypeError 例外が発生する。

set 関数を使って集合を作成する

set 関数を使っても集合は作成できる。特に空の集合は引数を与えずに set 関数を呼び出すことで作成する。「{}」は「空の辞書」を作成するのであって、空の集合を作成するわけではないことには注意しよう。

```
s = set()  # 空の集合
print(s)  # set()

d = {}  # {}は空の集合ではなく、空の辞書を作成する
print(s == d)  # False
print(type(d))  # <class 'dict'>
```

set 関数には、集合の要素となる値を格納している反復可能オブジェクトを渡す。以下に例を示す。

```
s = set(['foo', 'bar', 'baz'])  # 集合の要素を含む反復可能オブジェクトを与える
print(s)  # {'foo', 'baz', 'bar'}
```

この例では、'foo' ／ 'bar' ／ 'baz' という 3 つの文字列を含むリスト（反復可能オブジェクト）を set 関数に渡している。そのため、これら 3 つの要素を含んだ集合が作成される。

なお、文字列はそれ自体が個々の文字を要素とする反復可能オブジェクトだ。そのため、以下のように単体の文字列を set 関数に渡すと、文字列を構成する個々の文字が集合の要素となる。

```
s = set('foo')  # 文字列は反復可能オブジェクト
print(s)  # {'f', 'o'}
```

集合では同一要素が重複することはないので、集合に含まれる 'o' が 1 つだけであることにも注意しよう。

集合内包表記を使って集合を作成する

集合内包表記は次のような形式で記述する。

```
{要素を算出する式 for 変数 in 反復可能オブジェクト}
```

このときには反復可能オブジェクトから変数に取り出した値を使って、要素を計算することになるだろう。以下に例を示す。

```
s = {x * 2 for x in range(4)}
print(s)  # {0, 2, 4, 6}
```

この例では要素を算出する式は「x * 2」となっている。この変数 x には range(4) が返送する値が順に代入されるので、式の値は「0 * 2」「1 * 2」「2 * 2」「3 * 2」となる。そのため、この集合には「0」「2」「4」「6」が含まれるようになる。

内包表記で集合を作成するには

集合内包表記を使って集合を作成する方法と、set 関数との差、if 句との組み合わせ、注意点などを紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 07 月 20 日）

```
# 集合内包表記の基本型
squared = {n ** 2 for n in range(10)}
print(squared)  # {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}など

# 集合なので同じ値の要素が重複することはない
mystr = 'abccba'
diff = ord('a') - ord('A')  # 32
myset = {chr(ord(c) - diff) for c in mystr}
print(myset)  # {'A', 'C', 'B'}など

# 単に重複する要素を削除したいのであればset関数を使う
mystr = 'abccba'
myset = {c for c in mystr}
print(myset)  # {'a', 'b', 'c'}など

myset = set(mystr)
print(myset)  # {'a', 'b', 'c'}など

# if句と組み合わせる
mylist = [7, 4, 4, 7, 5, 9, 8, 5, 1, 10]
myset = {n for n in mylist if n % 2 == 0}  # mylistから偶数のみの集合を作成
print(myset)  # {8, 10, 4}など
```

集合内包表記の基本型

内包表記は集合の作成にも使える。以下にその基本構文を示す。

```
s = {要素の値を計算する式 for 値の計算で使用する変数 in 反復可能オブジェクト}
```

集合内包表記では、リストとは異なり（また、辞書とは同様に）、外側を波かっこ「{}」で囲む。このとき、辞書内包表記ではキーと値を計算する式を記述するが、集合内包表記では集合に含まれる要素の値を計算する式だけを記述すればよい。

以下に簡単な例を示す。

```
squared = {n ** 2 for n in range(10)}
print(squared)  # {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}など
```

この例では、0 ～ 9 の整数値を二乗した値を要素とする集合を作成している。なお、要素の表示順は上の出力結果とは異なるかもしれない。

集合内包表記が作成するのは集合なので、ある集合の中に重複する要素は含まれない。以下はその例だ。

```
mystr = 'abccba'
diff = ord('a') - ord('A')  # 32
myset = {chr(ord(c) - diff) for c in mystr}
print(myset)  # {'A', 'C', 'B'}など

mylist = [chr(ord(c) - diff) for c in mystr]
print(mylist)  # ['A', 'B', 'C', 'C', 'B', 'A']:リストでは要素の重複が許される
```

この例では、'abccba' という文字列（反復可能オブジェクト）を基に、各文字を大文字化したものを要素とする集合（とリスト）を作成している。同じ要素を重複して集合に含めることはできないので、集合では 'A'、'B'、'C' が 1 つずつ格納されている。同一要素の重複が許されるリストには、それらが 2 つずつ格納されている点に注目しよう。

なお、何らかの反復可能オブジェクトから重複する要素を削除したいだけであれば、内包表記を使わずに以下のように set 関数に渡すだけで、そうした集合が得られることは覚えておこう。

```
mystr = 'abccba'
myset = set(mystr)
print(myset)  # {'a', 'b', 'c'}など
```

集合内包表記は、反復可能オブジェクトの値を何らかの形で加工やフィルタリングした結果を要素とする集合を得るのに使用するということだ。

リストや辞書の内包表記と同様に if 句や条件式（三項演算子）も使用できる。以下に簡単な例を示す。

```
mylist = [7, 4, 4, 7, 5, 9, 8, 5, 1, 10]
myset = {n for n in mylist if n % 2 == 0}  # mylistから偶数のみの集合を作成
print(myset)  # {8, 10, 4}
```

ここでは集合の要素が偶数だけとなるように if 句でフィルタリングをしている。

注意点

辞書のキーと同様に、集合の要素にできるのはハッシュ可能なオブジェクトのみである点には注意。リストや辞書など、その内容が変更可能なオブジェクト（ハッシュ不可能なオブジェクト）を集合の要素にはできない。

```
klist = [[1, 2], [3, 4]]
vlist = ['foo', 'bar']

s = {(k, v) for k, v in zip(klist, vlist)}  # TypeError: unhashable type: 'list'
```

この例では、リストを要素とするリストと文字列リストを zip 関数で組み合わせて、「リストと文字列を要素とするタプル」を含んだ集合を作ろうとしているが、リストはハッシュ不可能であるため、TypeError 例外が発生する。

タプルのようにイミュータブル（変更不可能）なオブジェクトであっても、その要素がリストや辞書のように変更可能なオブジェクトであれば、それらを集合の要素とすることはやはりできない。

集合の要素を追加したり削除したりするには

集合に要素を追加したり、集合から要素を削除したりするのに使えるメソッドと演算子を紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 04 月 25 日）

```
s = {0, 1, 2}
print(s)  # {0, 1, 2}

# 集合に要素を追加
s.add(3)
print(s)  # {0, 1, 2, 3}

# addメソッドで追加できる要素は一度に1つで、ハッシュ可能なオブジェクトは渡せない
s.add(4, 5)  # TypeError
s.add([4, 5])  # TypeError

# 複数の要素を集合に加えるにはupdateメソッドか|演算子か|=演算子を使う
s.update({4, 5})
print(s)  # {0, 1, 2, 3, 4, 5}

s.update([6, 7])  # 集合以外の反復可能オブジェクトでもよい
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}

s2 = s | {8, 9}  # |演算子は新しい集合オブジェクトを作成する
print(s2)  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}

s2 = s | [8, 9]  # TypeError

s |= {8, 9}  # |=演算子は集合オブジェクト自身を変更する
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

# 集合から要素を削除
s.remove(9)
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7, 8}

s.remove(10)  # KeyError

s.discard(8)
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}

s.discard(10)  # 例外は発生しない
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}

# 任意の要素を取り出して戻り値とし、その要素を集合から削除
x = s.pop()
print(f'popped {x}, s: {s}')  # popped 0, s: {1, 2, 3, 4, 5, 6, 7}
```

```
# 複数の要素を集合から削除するには
s = {0, 1, 2, 3, 4, 5, 6}

s.difference_update({5, 6})
print(s)  # {0, 1, 2, 3, 4}

s.difference_update((3, 4))
print(s)  # {0, 1, 2}

# -演算子は新しい集合オブジェクトを作成する
s2 = s - {1, 2}
print(s2)  # {0}
print(s)  # {0, 1, 2}

# -=演算子は集合オブジェクト自身を変更する
s -= {1, 2}
print(s)  # {0}

s -= [0, 1]  # TypeError

# 集合から全ての要素を削除
s.clear()
print(s)  # set()
```

集合の要素を追加したり削除したりする方法

集合に要素を追加するには以下のような方法がある。

メソッド／演算子	説明
addメソッド	要素を1つ追加
updateメソッド	要素を1つ以上追加
|演算子	要素を1つ以上追加。新たな集合を作成
|=演算子	要素を1つ以上追加。集合自身を更新

集合に要素を追加するのに使えるメソッド／演算子

集合から要素を削除するには以下のような方法がある。

メソッド／演算子	説明
removeメソッド	要素を1つ削除。指定した要素がないと例外が発生
discardメソッド	要素を1つ削除。指定した要素がなくても例外が発生しない
popメソッド	集合から任意の要素を削除し、それを戻り値とする
difference_updateメソッド	要素を1つ以上削除
-演算子	要素を1つ以上削除。新たな集合を作成
-=演算子	要素を1つ以上削除。集合自身を更新
clearメソッド	全ての要素を削除する

集合から要素を削除するのに使えるメソッド／演算子

以下ではこれらを順に説明する。

要素の追加

集合に要素を 1 つだけ追加するには add メソッドを使用する。以下に例を示す。

```
s = {0, 1, 2}
print(s)  # {0, 1, 2}

# 集合に要素を追加
s.add(3)
print(s)  # {0, 1, 2, 3}
```

この例では集合 s は {0, 1, 2} という 3 つの要素を持つように初期化されている。その集合に対して、「s.add(3)」を実行することで集合に新たな要素が追加されている。

add メソッドは引数を 1 つだけ受け取り、それを集合の要素として追加する。引数の値がすでに集合の要素として格納されていれば、その集合の要素に変化はない。また、集合の要素として格納できるものでなければ TypeError 例外が発生する。

```
s.add(4, 5)  # TypeError
s.add([4, 5])  # TypeError
```

上の 1 つ目の例では複数の引数を渡していることで TypeError 例外が発生している。2 つ目の例では、リストを集合の要素として追加しようとしているが、リストは集合の要素にはできないので TypeError 例外が発生する。

複数の要素をまとめて集合に追加したいときには update メソッドか | 演算子か |= 演算子を使用する。

以下は update メソッドの呼び出し例だ。

```
s.update({4, 5})
print(s)  # {0, 1, 2, 3, 4, 5}
```

この例では先ほどの例でも使用した集合 s に対し、引数に {4, 5} という集合を指定して update メソッドを呼び出している。その結果、集合 s には 2 つの要素が追加されている。引数に指定した値の中で、既に集合の要素となっているものについては重複することはない。

なお、update メソッドには集合以外の反復可能オブジェクトを渡してもよい。以下はその例だ。

```
s.update([6, 7])  # 集合以外の反復可能オブジェクトでもよい
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}
```

この例では、[6, 7] というリストを渡しているが、2 つの整数値が集合に加えられているのが分かるはずだ。

集合と集合を足し合わせて、それらの和集合を作成するには | 演算子を使える。| 演算子は 2 つの集合の和集合となる集合オブジェクトを新しく作成する。

```
result = {0, 1, 2} | {2, 3, 4}
print(result)  # {0, 1, 2, 3, 4}
```

これを使って、集合に新たな要素を追加できる。以下に例を示す。

```
s2 = s | {8, 9}  # |演算子は新しい集合オブジェクトを作成する
print(s2)  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}
```

このコード例では、「s | {8, 9}」と | 演算子を使って、集合 s と {8, 9} という集合の和集合を作成し、その結果を変数 s2 に代入している。そのため、集合 s2 の内容は上の式で得られる和集合となり、集合 s の内容は以前とは変わらない。

update メソッドとは異なり、リストなどの反復可能オブジェクトをオペランドとして置くことはできないことには注意（次に説明する |= 演算子でもリストなど、集合以外の反復可能オブジェクトをオペランドとすることはできない）。

```
s2 = s | [8, 9]  # TypeError
```

update メソッドと |= 演算子では代入先の集合オブジェクト自身の内容が変化する（更新される）ことが異なる。以下に例を示す。

```
s |= {8, 9}  # |=演算子は集合オブジェクト自身を変更する
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

要素の削除

集合から要素を 1 つだけ削除するには、remove メソッドか discard メソッドを使用する。これらは引数に指定した値を集合から削除するが、指定した値が集合に含まれていなかったときの挙動に違いがある。remove メソッドは例外を発生し、discard メソッドは例外を発生しない。

以下に remove メソッドの使用例を示す。先ほどまでの例で集合 s の内容は {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} のようになっている。

```
s.remove(9)
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7, 8}

s.remove(10)  # KeyError
```

最初に値「9」を指定して remove メソッドを呼び出している。そのため、集合 s からはこれが削除される。次に値「10」を指定して remove メソッドを呼び出しているが、集合 s にこの値は含まれていないので KeyError 例外が発生する。

これに対して、discard メソッドでは存在しない要素を指定しても例外が発生しない点が異なる。

```
s.discard(8)
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}

s.discard(10)  # 例外は発生しない
print(s)  # {0, 1, 2, 3, 4, 5, 6, 7}
```

remove メソッドと discard メソッドはどちらも要素を 1 つだけ削除するものだ。これ以外にも任意の要素を戻り値とするとともに、集合からそれを削除する pop メソッドもある。以下に例を示す。

```
x = s.pop()
print(f'popped {x}, s: {s}')  # popped 0, s: {1, 2, 3, 4, 5, 6, 7}
```

なお、集合に要素がない場合、pop メソッドは KeyError 例外を発生する。

複数の要素を集合から削除するには difference_update メソッドと - 演算子、-= 演算子を使う。

difference_update メソッドは引数に受け取った集合やその他の反復可能オブジェクトに含まれる要素を集合から削除する。以下に difference_update メソッドの使用例を示す。

```
s = {0, 1, 2, 3, 4, 5, 6}

s.difference_update({5, 6})
print(s)  # {0, 1, 2, 3, 4}
```

update メソッドと同様、difference_update メソッドは集合以外の反復可能オブジェクトも受け取る。以下に例を示す。

```
s.difference_update((3, 4))
print(s)  # {0, 1, 2}
```

この例ではタプルを渡しているが、問題なく集合からタプルに含めた要素が削除されていることを確認しよう。

- 演算子は、第 1 オペランドに指定した集合から、第 2 オペランドに指定した集合に含まれている要素を削除した集合（差集合）を新しく作成する。

```
result = {0, 1, 2} - {1, 2, 3}
print(result)  # {0}
```

この例では第 1 オペランドに指定した集合は {0, 1, 2} であり、第 2 オペランドに指定した集合は {1, 2, 3} となっている。この中で重複する要素である {1, 2} が第 1 オペランドの集合から削除される。

このことを使用して、集合から 1 つ以上の特定の要素を削除できる。

```
s2 = s - {1, 2}
print(s2)  # {0}
print(s)  # {0, 1, 2}
```

- 演算子は | 演算子と同様に、新しく集合を作成する。そのため、上のコード例では集合 s2 の内容は差集合となり、集合 s は以前のままとなっている。

-= 演算子は、|= 演算子と同様に代入先の集合オブジェクト自身を変更（更新）する。以下に例を示す。

```
s -= {1, 2}
print(s)  # {0}
```

集合以外の反復可能オブジェクトをオペランドに置くことができないのも |= 演算子と同様だ。

```
s -= [0, 1]  # TypeError
```

最後に集合の全要素を削除するには clear メソッドを使う。以下に例を示す。

```
s.clear()
print(s)  # set()
```

集合の要素を操作するには

集合の要素数を調べたり、集合と集合の包含関係を調べたり、集合と集合から和／差／交差／対称差からなる新たな集合を得たりする方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2023 年 05 月 09 日）

```
# sの要素数を調べる
s = {0, 1, 2, 3}
l = len(s)
print(l)  # 4

# sに要素xが含まれているかどうかを調べる
print(1 in s)  # True
print(5 in s)  # False

# sに要素xが含まれていないかかどうかを調べる x not in s
print(1 not in s)  # False
print(5 not in s)  # True

# s1とs2が同じ(要素のみを含んでいる)かどうかを調べる
s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 == s2)  # True

# s1とs2が異なるかどうかを調べる
s1, s2 = {0, 1, 2}, {0, 1, 3}
print(s1 != s2)  # True

# s1とs2が互いに素かどうかを調べる
s1, s2 = {0, 1, 2}, {3, 4, 5}
print(s1.isdisjoint(s2))  # True

s1, s2 = {0, 1, 2}, {2, 3, 4}
print(s1.isdisjoint(s2))  # False

# どちらかの集合がもう一方の集合に包含されるかどうかを調べる
s1, s2 = {0, 1, 2, 3}, {0, 2}
print(s1.issubset(s2))  # False
print(s2.issubset(s1))  # True
print(s2 <= s1)  # True

s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 <= s2)  # True

# s1がs2の真部分集合かどうかを調べる
s1, s2 = {0, 1, 2}, {0, 1, 2, 3}
print(s1 < s2)  # True

s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 < s2)  # False
```


→目次に戻る

```
# どちらかの集合がもう一方の集合を包含するかどうかを調べる
s1, s2 = {0, 1, 2, 3}, {0, 1, 2}
print(s1.issuperset(s2))  # True
print(s2.issuperset(s1))  # False
print(s1 >= s2)  # True

s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 >= s2)  # True

# s1がs2の真上位集合かどうかを調べる
s1, s2 = {0, 1, 2, 3}, {0, 1, 2}
print(s1 > s2)  # True

s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 > s2)  # False

# s1とs2の和からなる集合を作成する
s1, s2 = {0, 1, 2}, {1, 2, 3}
result = s1.union(s2)
print(result)  # {0, 1, 2, 3}
print(s1)  # {0, 1, 2}

result = s1 | s2
print(result)  # {0, 1, 2, 3}
print(s1)  # {0, 1, 2}

s3 = {3, 4, 5}
result = s1 | s2 | s3
print(result)  # {0, 1, 2, 3, 4, 5}

# s1とs2の差からなる集合を作成する
s1, s2 = {0, 1, 2, 3, 4, 5}, {1, 3, 5}
diff = s1.difference(s2)
print(diff)  # {0, 2, 4}

s1, s2 = {0, 1, 2, 3, 4, 5}, {0, 2, 4}
diff = s1 - s2
print(diff)  # {1, 3, 5}

# s1とs2の交差からなる集合を作成する
s1, s2 = {0, 1, 2, 3}, {2, 3, 4, 5}
intersection = s1.intersection(s2)
print(intersection)  # {2, 3}
intersection = s1 & s2
print(intersection)  # {2, 3}

# s1とs2の対称差からなる集合を作成する
s1, s2 = {0, 1, 2, 3, 4, 5}, {3, 4, 5, 6, 7, 8}
symdiff = s1.symmetric_difference(s2)
print(symdiff)  # {0, 1, 2, 6, 7, 8}
```

```
s1, s2 = {0, 1, 2, 3, 4, 5}, {4, 5, 6, 7, 8, 9}
symdiff = s1 ^ s2
print(symdiff)  # {0, 1, 2, 3, 6, 7, 8, 9}
```

集合に対する操作

以下に集合で行える各種操作を一覧する。表中に出てくる s ／ s1 ／ s2 は全て何らかの要素を含んだ集合オブジェクトとする。

操作	メソッド／演算子
sの要素数を調べる	len(s)
sに要素xが含まれているかどうかを調べる	x in s
sに要素xが含まれていないかかどうかを調べる	x not in s
s1とs2が同じ（要素のみを含んでいる）かどうかを調べる	s1 == s2
s1とs2が異なるかどうかを調べる	s1 != s2
s1とs2が互いに素かどうかを調べる	s1.isdisjoint(s2)
s1がs2に包含されるかどうかを調べる	s1.issubset(s2) s1 <= s2
s1がs2の真部分集合かどうかを調べる	s1 < s2
s1がs2を包含するかどうかを調べる	s1.issuperset(s2) s1 >= s2
s1がs2の真上位集合かどうかを調べる	s1 > s2
s1とs2の和集合を作成する	s1.union(s2) s1 | s2
s1とs2の交差を作成する	s1.intersection(s2) s1 & s2
s1からs2の要素を削除した差集合を作成する	s1.difference(s2) s1 - s2
s1とs2の対称差集合を作成する	s1.symmetric_difference(s2) s1 ^ s2

集合に対する操作

以下では、これらの操作を幾つかの分野に分けて見ていく。

基本

リストなどの反復可能オブジェクトと同様に、集合の要素数は len 関数で調べられる。また、特定の要素が集合に含まれているか／含まれていないかは in 演算子／ not in 演算子で調べられる。2 つの集合が同じか（同じ要素だけを含んでいるか）どうかは == 演算子や != 演算子で調べられる。

以下に例を示す。

```
# sの要素数を調べる
s = {0, 1, 2, 3}
l = len(s)
print(l)  # 4

# sに要素xが含まれているかどうかを調べる
print(1 in s)  # True
print(5 in s)  # False

# sに要素xが含まれていないかかどうかを調べる x not in s
print(1 not in s)  # False
print(5 not in s)  # True

# s1とs2が同じ(要素のみを含んでいる)かどうかを調べる
s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 == s2)  # True

# s1とs2が異なるかどうかを調べる
s1, s2 = {0, 1, 2}, {0, 1, 3}
print(s1 != s2)  # True
```

包含関係のチェック

ある集合 s1 と s2 があったとすると、それらには包含関係が成り立つことがある。集合 s1 の要素が集合 s2 の要素と全く同じであれば、それらの集合は等しい。

また、集合 s1 が集合 s2 の全ての要素を含んでいれば、集合 s1 は集合 s2 を「包含」している。このとき、集合 s1 は集合 s2 の「上位集合」（superset）であり、集合 s2 は集合 s1 の「部分集合」（subset）である。さらに集合 s1 には集合 s2 に含まれていない要素があるのであれば、集合 s1 は集合 s2 の「真上位集合」であり、集合 s2 は集合 s1 の「真部分集合」となる。

一方、集合 s1 と集合 s2 の間で共通する要素が 1 つもないような場合、それらの集合は「互いに素」という。このような集合同士の関係を調べるために Python では以下のようなメソッドおよび演算子が用意されている（表中の s1 と s2 は共に何らかの集合を表す）。

操作	メソッド／演算子
s1とs2が互いに素かどうかを調べる	s1.isdisjoint(s2)
s1がs2に包含されるかどうか（s1がs2の部分集合かどうか）を調べる	s1.issubset(s2) s1 <= s2
s1がs2の真部分集合かどうかを調べる	s1 < s2
s1がs2を包含するかどうか（s1がs2の上位集合かどうか）を調べる	s1.issuperset(s2) s1 >= s2
s1がs2の真上位集合かどうかを調べる	s1 > s2

集合と集合の関係を調べるメソッド／演算子

以下は isdisjoint メソッドを使って、2 つの集合が互いに素であるかどうかを調べる例だ。

```
s1, s2 = {0, 1, 2}, {3, 4, 5}
print(s1.isdisjoint(s2))  # True

s1, s2 = {0, 1, 2}, {2, 3, 4}
print(s1.isdisjoint(s2))  # False
```

最初の例では {0, 1, 2} と {3, 4, 5} という共通する要素がない（つまり、互いに素な）2 つの集合を比べているので結果は True となる。2 つ目の例では共通する要素があるので結果は False となる。

次は {0, 1, 2, 3} という集合と {0, 2} という集合があったときに、どちらかがどちらかの部分集合であるかどうかを issubset メソッドと <= 演算子で調べる例だ。

```
s1, s2 = {0, 1, 2, 3}, {0, 2}
print(s1.issubset(s2))  # False
print(s2.issubset(s1))  # True
print(s2 <= s1)  # True
```

issubset メソッドの呼び出しに使用した集合が、引数に指定した集合の部分集合なら結果は True となる。<= 演算子の場合は、演算子の左側の集合が右側の集合の部分集合であるときに結果が True となる。なお、issubset メソッドも <= 演算子も 2 つの集合が同一である場合にも True となる（以下は <= 演算子での例）。

```
s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 <= s2)  # True
```

そうではなく、どちらかがどちらかの真部分集合であるかどうかを調べるのなら、< 演算子を使用する。

```
s1, s2 = {0, 1, 2}, {0, 1, 2, 3}
print(s1 < s2)  # True

s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 < s2)  # False
```

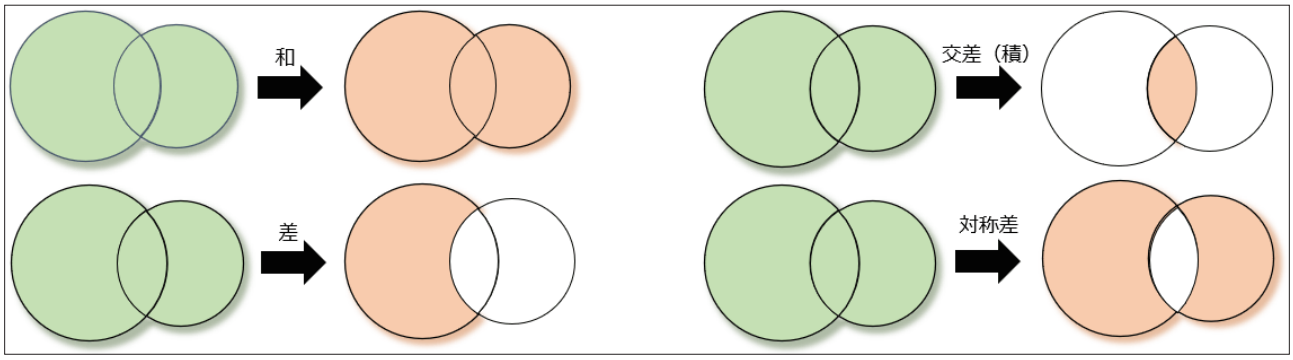
最初の例では、{0, 1, 2} という集合が {0, 1, 2, 3} の真部分集合であるかどうかを調べているので、結果は True となる。次の例では、{0, 1, 2} という同じ要素からなる 2 つの集合を比べているので <= 演算子のときとは異なり、結果は False となる。

同様に、一方の集合がもう一方の集合を包含しているかどうかは issuperset メソッド、>= 演算子、> 演算子を使って調べられる。以下に例を示す。説明は不要だろう。

```
# どちらかの集合がもう一方の集合を包含するかどうかを調べる
s1, s2 = {0, 1, 2, 3}, {0, 1, 2}
print(s1.issuperset(s2))  # True
print(s2.issuperset(s1))  # False
print(s1 >= s2)  # True

s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 >= s2)  # True

# s1がs2の真上位集合かどうかを調べる
s1, s2 = {0, 1, 2, 3}, {0, 1, 2}
print(s1 > s2)  # True

s1, s2 = {0, 1, 2}, {0, 1, 2}
print(s1 > s2)  # False
```

2 つ以上の集合から集合を新たに作成する

2 つ以上の集合を基に和、差、交差（積）、対称差という新たな集合を作成できる。



和／差／交差／対称差

これらを作成するメソッドと演算子を以下に示す。

操作	メソッド／演算子
s1とs2の和集合を作成する	s1.union(s2) s1 | s2
s1とs2の交差を作成する	s1.intersection(s2) s1 & s2
s1からs2の要素を削除した差集合を作成する	s1.difference(s2) s1 - s2
s1とs2の対称差集合を作成する	s1.symmetric_difference(s2) s1 ^ s2

和／差／交差／対称差を作成するメソッド／演算子

以下は {0, 1, 2} という集合と {0, 1, 2, 3} という集合の和からなる集合を新たに作成する例だ。

```
s1, s2 = {0, 1, 2}, {1, 2, 3}
result = s1.union(s2)
print(result)  # {0, 1, 2, 3}
print(s1)  # {0, 1, 2}

result = s1 | s2
print(result)  # {0, 1, 2, 3}
print(s1)  # {0, 1, 2}
```

最初の例では union メソッドを使用している。次の例では | 演算子を使用している。これらのメソッド／演算子は新たに集合を作成する点は覚えておこう。もともとの集合は変化しない。また、| 演算子を連続することで、3 つ以上の集合の和を計算することもできる（こうした特性は他の演算子でも同様）。

```
s3 = {3, 4, 5}
result = s1 | s2 | s3
print(result)  # {0, 1, 2, 3, 4, 5}
```

以下は difference メソッドと - 演算子を使って、2 つの集合の差からなる集合を作成する例だ。

```
s1, s2 = {0, 1, 2, 3, 4, 5}, {1, 3, 5}
diff = s1.difference(s2)
print(diff)  # {0, 2, 4}

s1, s2 = {0, 1, 2, 3, 4, 5}, {0, 2, 4}
diff = s1 - s2
print(diff)  # {1, 3, 5}
```

最初の例では集合 s1（{0, 1, 2, 3, 4, 5}）から集合 s2（{1, 3, 5}）に含まれている要素を除いたものが差となる。2 つ目の例では集合 s2 が {0, 2, 4} となっているので、その要素を集合 s1 から取り除いたものが差となる。また、集合 s1 から集合 s2 に含まれる要素を削除して、それを集合 s1 の値として更新する difference_update メソッドもある（例は省略）。

以下は intersection メソッドと & 演算子を使って、交差からなる集合を新たに作成する例だ。交差は 2 つの集合に共通する要素からなるので、この場合は {2, 3} という集合が新たに作成される。また、集合 s1 と集合 s2 の交差を得て、それを集合 s1 の値として更新する intersection_update メソッドもある（例は省略）。

```
s1, s2 = {0, 1, 2, 3}, {2, 3, 4, 5}
intersection = s1.intersection(s2)
print(intersection)  # {2, 3}
intersection = s1 & s2
print(intersection)  # {2, 3}
```

最後に、symmetric_difference メソッドと ^ 演算子を使って、対称差を求める例だ。対称差とは 2 つの集合でいずれかの集合にしか存在しない要素のこと（2 つの集合で共通する要素を取り除いたもの）。

```
s1, s2 = {0, 1, 2, 3, 4, 5}, {3, 4, 5, 6, 7, 8}
symdiff = s1.symmetric_difference(s2)
print(symdiff)  # {0, 1, 2, 6, 7, 8}
```

この例では symmetric_difference メソッドを使用している。2 つの集合は {0, 1, 2, 3, 4, 5} と {3, 4, 5, 6, 7, 8} なので、対称差は共通する要素を取り除いた {0, 1, 2, 6, 7, 8} となる。また、集合 s1 と集合 s2 の対称差を得て、それを集合 s1 の値として更新する symmetric_difference_update メソッドもある（例は省略）。

以下は ^ 演算子を使用する例だ。

```
s1, s2 = {0, 1, 2, 3, 4, 5}, {4, 5, 6, 7, 8, 9}
symdiff = s1 ^ s2
print(symdiff)  # {0, 1, 2, 3, 6, 7, 8, 9}
```

この例では 2 つの集合は {0, 1, 2, 3, 4, 5} と {4, 5, 6, 7, 8, 9} なので対称差は {0, 1, 2, 3, 6, 7, 8, 9} となる。

編集：@IT 編集部
発行：アイティメディア株式会社