@IT
atmarkIT

解決！ Python：リスト（配列）編

かわさきしんじ, Deep Insider編集部［著］

リスト（配列）の使い方まとめ

Python のリスト（配列）の初期化、要素へのアクセス、要素の追加、要素の削除、要素の検索、要素の並べ替えを行う方法をまとめて紹介する。

（2020 年 12 月 25 日）

Python では他の言語における配列と同等なデータ構造はリストとして実装されている。ここでは、リスト（配列）の使い方をコード例を主体にまとめる。コード例の後には、それらを説明する章へのリンクもあるので、詳細はそちらを参照されたい。

リストを初期化する

```
# リスト（配列）の初期化：角かっこ「[]」で囲み、要素をカンマ「,」区切りで並べる
empty_list = []  # 空のリスト
print(empty_list)  # []

int_list = [0, 1, 2]  # 整数リスト（整数配列）
print(int_list)  # [0, 1, 2]

mylist = [0, 'abc', 1, 'def']  # リスト（配列）には任意の型の要素を格納できる
print(mylist)  # [0, 'abc', 1, 'def']

# リスト（配列）の初期化：list 関数を呼び出す
int_list = list()  # 空のリスト（配列）
print(int_list)  # []

int_list = list((4, 5, 6))  # タプルの要素を基にリスト（配列）が作成される
print(int_list)  # [4, 5, 6]

int_list = list(range(5))  # range オブジェクトから整数リスト（整数配列）を作成
print(int_list)  # [0, 1, 2, 3, 4]

str_list = list('python')  # 文字列の各文字を要素とするリストを作成
print(str_list)  # ['p', 'y', 't', 'h', 'o', 'n']
```

```
# リスト内包表記
int_list = [x for x in range(0, 10, 2)]  # range オブジェクトからリストを作成
print(int_list)  # [0, 2, 4, 6, 8]

int_list = [x for x in range(10) if x % 2 == 1]  # if 節を用いる例
print(int_list)  # [1, 3, 5, 7, 9]

# if else 式を使うときには for 節に続けずに、内包表記の先頭に記述する
str_list = [c.upper() if c.islower() else c.lower() for c in 'AbCdE']
print(str_list)  # ['a', 'B', 'c', 'D', 'e']

# リストのリスト（配列の配列）
mylist = [[0, 1, 2], [3, 4, 5]]  # 2 次元のリスト（配列）の作成
print(mylist)  # [[0, 1, 2], [3, 4, 5]]

mul_tbl = [[x * y for x in range(1, 4)] for y in range(1, 5)]
print(mul_tbl)  # [[1, 2, 3], [2, 4, 6], [3, 6, 9], [4, 8, 12]]
```

リストの初期化については「リスト（配列）を初期化するには（[]、list 関数、リスト内包表記）」を参照のこと。

リストの要素にアクセスする（インデックス）

```
mylist = list(range(10))
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# インデックスによるアクセス
n = mylist[1]  # 正数は先頭からのインデックス。先頭の要素のインデックスが 0
print(n)  # 1
n = mylist[-2]  # 負数は末尾からのインデックス。末尾の要素のインデックスが -1
print(n)  # 8

# インデックスを使った要素の削除
del mylist[4]
print(mylist)  # [0, 1, 2, 3, 5, 6, 7, 8, 9]
```

```
# インデックスを使って要素を変更する
mylist[-1] = 90
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 90]
```

リストの要素へのアクセスについては「リスト（配列）の要素にインデックスやスライスを使ってアクセスするには」を参照のこと。

リストの要素にアクセスする（スライス）

```
# スライスによるアクセス
mylist = list(range(10))

s = mylist[0:5]  # 0 ～ 4 番目の 5 要素を取り出す
print(s)  # [0, 1, 2, 3, 4]

s = mylist[:]  # 全要素を取り出す
print(s)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

s = mylist[5:]  # 5 番目以降の全要素を取り出す
print(s)  # [5, 6, 7, 8, 9]

s = mylist[:5]  # 0 ～ 4 番目の 5 要素を取り出す
print(s)  # [0, 1, 2, 3, 4]

s = mylist[0:10:2]  # 0、2、4、6、8 番目の要素を取り出す
print(s)  # [0, 2, 4, 6, 8]

s = mylist[::2]  # 同上
print(s)  # [0, 2, 4, 6, 8]

s = mylist[-4:]  # -4 番目から末尾までの全要素を取り出す
print(s)  # [6, 7, 8, 9]

s = mylist[-1:-9:-1]  # リスト末尾から逆順に要素を取り出す
print(s)  # [9, 8, 7, 6, 5, 4, 3, 2]
```

```
# スライスによる要素の削除
mylist = list(range(10))

del mylist[3:5]  # 3 番目と 4 番目の要素を削除
print(mylist)  # [0, 1, 2, 5, 6, 7, 8, 9]

del mylist[0::2]  # 上の削除結果から、0、2、4、6 番目の要素を削除
print(mylist)  # [1, 5, 7, 9]

mylist[0:2] = []  # 上の削除結果から先頭の 2 要素を削除
print(mylist)  # [7, 9]

# スライスを使って要素を変更する
mylist = list(range(10))

mylist[1:3] = [10, 20]  # スライスの先頭と末尾を指定して要素を変更
print(mylist)  # [0, 10, 20, 3, 4, 5, 6, 7, 8, 9]

mylist[1:3] = [1, 1.5, 2] # この場合はスライスと新しい値の要素数が異なっても OK
print(mylist)  # [0, 1, 1.5, 2, 3, 4, 5, 6, 7, 8, 9]

mylist = list(range(10))
mylist[1::2] = [10, 30, 50, 70, 90]  # 先頭／末尾／増分を指定して要素を変更
print(mylist)  # [0, 10, 2, 30, 4, 50, 6, 70, 8, 90]

mylist[0::2] = [0.0, 2.0, 4.0]  # ValueError：このときには同じ要素数でないとダメ
```

リストの要素へのアクセスについては「リスト（配列）の要素にインデックスやスライスを使ってアクセスするには」を参照のこと。

リストへ要素を追加する（+ 演算子／ += 演算子）

```
mylist = list(range(5))
print(mylist)  # [0, 1, 2, 3, 4]

# + 演算子によるリストの結合（リストの末尾へのリストの追加）
tmp = mylist + [5, 6]  # + 演算子では新しいリストが作成される
print(tmp)  # [0, 1, 2, 3, 4, 5, 6]
print(mylist)  # [0, 1, 2, 3, 4]

mylist = mylist + 5  # TypeError：リストはリスト以外と結合できない
mylist = mylist + (5, 6)  # TypeError：リストはリスト以外と結合できない

# += 演算子によるリスト末尾への要素の追加
mylist += [5]  # += 演算（累算代入）では元のリストが変更される
print(mylist)  # [0, 1, 2, 3, 4, 5]

mylist += 6  # TypeError：リストへ累算代入できるのは反復可能オブジェクトのみ

mylist += (6, 7)  # リスト末尾にタプルの個々の要素を追加
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7]
```

リストへの要素の追加については「リスト（配列）に要素を追加するには（+ 演算子／ += 演算子／ append ／ extend ／ insert メソッド）」を参照のこと。

リストへ要素を追加する（append ／ extend ／ insert メソッド）

```
mylist = list(range(5))

mylist.append(5)  # リストの末尾に要素を追加
print(mylist)  # [0, 1, 2, 3, 4, 5]

mylist.append(6, 7)  # TypeError：append メソッドの引数は 1 つだけ

mylist.append([6, 7])  # リストの末尾に単一の要素として「リスト」を追加
```

```
print(mylist)  # [0, 1, 2, 3, 4, 5, [6, 7]]

mylist.extend([8, 9])  # リストの末尾に反復可能オブジェクトの個々の要素を追加
print(mylist)  # [0, 1, 2, 3, 4, 5, [6, 7], 8, 9]

mylist.extend(10)  # TypeError：extend メソッドには反復可能オブジェクトのみ渡せる

mylist.extend({10: 'foo', 11: 'bar'})  # 反復可能オブジェクトなので OK
print(mylist)  # [0, 1, 2, 3, 4, 5, [6, 7], 8, 9, 10, 11]

mylist.insert(1, 0.5)  # 指定したインデックスに要素を挿入
print(mylist)  # [0, 0.5, 1, 2, 3, 4, 5, [6, 7], 8, 9, 10, 11]
```

リストへの要素の追加については「リスト（配列）に要素を追加するには（+ 演算子／ += 演算子／ append ／ extend ／ insert メソッド）」を参照のこと。

リストから要素を削除する（del 文、インデックス／スライス）

```
# インデックスを指定して del 文で要素を削除
mylist = list(range(5))  # [0, 1, 2, 3, 4]
del mylist[0]
print(mylist)  # [1, 2, 3, 4]

# スライスを指定して del 文で要素を削除
mylist = list(range(5))  # [0, 1, 2, 3, 4]
del mylist[1:4]  # 1 ～ 3 番目の要素を削除
print(mylist)  # [0, 4]

mylist = list(range(5))  # [0, 1, 2, 3, 4]
del mylist[::2]  # 0、2、4 番目の要素を削除
print(mylist)  # [1, 3]

mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del mylist[1:7:3]  # 1 番目と 4 番目の要素を削除
print(mylist)  # [0, 2, 3, 5, 6, 7, 8, 9]
```

```
# スライスで指定した範囲に空のリストを代入することで削除
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
mylist[4:8] = []  # 4 ～ 7 番目の要素を削除
print(mylist)  # [0, 1, 2, 3, 8, 9]

print('mylist[:]:', mylist[:])  # mylist[:]: [0, 1, 2, 3, 8, 9]
mylist[:] = []  # リストの全要素を削除
print(mylist)  # []
mylist = list(range(5))
del mylist[:]  # このようにも書ける
```

リストからの要素の削除については「リスト（配列）から要素を削除するには（del 文、remove ／ clear ／ pop メソッド、リスト内包表記）」を参照のこと。

リストから要素を削除する（clear ／ pop ／ remove メソッド、リスト内包表記）

```
# clear メソッドでリストの全要素を削除
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
mylist.clear()
print(mylist)  # []

# pop メソッドで指定したインデックスにある要素をリストから削除して、その値を取得
mylist = list(range(10))
value = mylist.pop()  # インデックスを指定しない場合は末尾の要素が削除される
print('popped value:', value)  # popped value: 9
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4, 5, 6, 7, 8]

value = mylist.pop(5)  # 5 番目の要素を削除
print('popped value:', value)  # popped value: 5
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4, 6, 7, 8]

# remove メソッドで削除したい値を指定して、リストからその値を削除
mylist = [2, 0, 6, 0, 5, 7, 2, 1, 5, 5]
```

```
mylist.remove(0)  # リスト内の要素 0 のうち先頭にあるものを削除
print(mylist)  # [2, 6, 0, 5, 7, 2, 1, 5, 5]
mylist.remove(0)  # リスト内の要素 0 のうち先頭にあるものを削除
print(mylist)  # [2, 6, 5, 7, 2, 1, 5, 5]
mylist.remove(0)  # ValueError：指定した値がリスト中にないと例外となる

# リストから特定の条件に合致するものをまとめて削除
mylist = [2, 0, 6, 0, 5, 7, 2, 1, 5, 5]
target = 0  # 要素 0 を mylist から取り除きたい

mylist = [item for item in mylist if item != target]  # 取り除く＝それ以外を残す
print(mylist)  # [2, 6, 5, 7, 2, 1, 5, 5]
```

リストからの要素の削除については「リスト（配列）から要素を削除するには（del 文、remove ／ clear ／ pop メソッド、リスト内包表記）」を参照のこと。

リストから要素を検索する（in ／ not in 演算子、count ／ index メソッド、min ／ max 関数）

```
mylist = [2, 5, 11, 15, 1, 3, 18, 3, 19, 8]  # 10 個の整数値を要素とするリスト

# 指定した値がリストに含まれているか含まれていないかを調べる
result = 0 in mylist  # 整数値 0 が mylist に含まれているかを調べる
print(result)  # False

result = 0 not in mylist  # 整数値 0 が mylist に含まれていないかを調べる
print(result)  # True

# 指定した値がリストに何個含まれているかを調べる
cnt = mylist.count(3)  # 整数値 3 が mylist に何個含まれているかを調べる
print(cnt)  # 2

# 指定した値がどのインデックスにあるかを検索
idx = mylist.index(3)  # mylist に含まれる整数値 3 の最小のインデックスを求める
print(idx)  # 5
```

```
idx = mylist.index(3, 6, len(mylist))  # 検索範囲を指定
print(idx)  # 7

idx = mylist.index(100)  # ValueError：存在しない値を指定

# リストの要素で最大／最小のものを求める
max_value = max(mylist)  # 最大値を求める
print(max_value)  # 19

min_value = min(mylist)  # 最小値を求める
print(min_value)  # 1

max_value = max([])  # ValueError：max 関数／ min 関数に空のリストを渡すと例外
max_value = max([], default='none')  # defalut 引数は空リストを渡したときの戻り値
print(max_value)  # none

max_value = max([0, 1], [4, 5], [2, 3])  # 複数のリストの中で最大のリストを取得
print(max_value)  # [4, 5]

mylist = ['pYthon', 'pyThon', 'Python', 'PYTHON']
max_value = max(mylist, key=str.lower)  # key 引数を指定する例
print(max_value)  # pYthon
```

リストからの要素の検索については「リスト（配列）から要素を検索するには（in ／ not in 演算子、count ／ index メソッド、min ／ max 関数）」を参照のこと。

リストの要素をソート／並べ替える（sort ／ reverse メソッド、sorted ／ reversed 関数）

```
# リストの要素をソートする（インプレース）
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
mylist.sort()
print(mylist)  # [0, 1, 2, 5, 6, 7, 9, 15, 16, 18]

# リストの要素をソートした新しいリストを作成する
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
newlist = sorted(mylist)
print('mylist:', mylist)  # mylist: [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
print('newlist:', newlist)  # newlist: [0, 1, 2, 5, 6, 7, 9, 15, 16, 18]

# リストの要素をソートする（インプレース／逆順）
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
mylist.sort(reverse=True)
print(mylist)  # [18, 16, 15, 9, 7, 6, 5, 2, 1, 0]

# ソートに使用するキーを決定する関数を指定
mylist = [-1.7, 0.1, 4.5, -1.5, -4.6]
mylist.sort(key=abs)  # 絶対値順でソート
print(mylist)  # [0.1, -1.5, -1.7, 4.5, -4.6]

# ラムダ式を使って、リストのリストをソートする
mylist = [[0, 1, 2], [1, 2, 0], [2, 0, 1]]
mylist.sort(key=lambda x: x[1])  # 1 番目の要素をキーとしてリストをソート
print(mylist)  # [[2, 0, 1], [0, 1, 2], [1, 2, 0]]

# key キーワード引数と reverse キーワード引数を使用
mylist = [-1.7, 0.1, 4.5, -1.5, -4.6]
mylist.sort(key=abs, reverse=True)  # 絶対値を基に降順でソート
print(mylist)  # [-4.6, 4.5, -1.7, -1.5, 0.1]
```


→目次に戻る

```
# 要素を逆順に並べ替える（インプレース）
mylist = list(range(5))  # [0, 1, 2, 3, 4]
mylist.reverse()
print(mylist)  # [4, 3, 2, 1, 0]

# 要素を逆順に並べ替えた新しいイテレータを作成する
mylist = list(range(5))  # [0, 1, 2, 3, 4]
iterator = reversed(mylist)  # reversed 関数の戻り値はイテレータ
print(iterator)  # <list_reverseiterator object at ...>
newlist = list(iterator)  # イテレータからリストを作成
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4]
print('newlist:', newlist)  # newlist: [4, 3, 2, 1, 0]
```

リストの要素の並べ替えについては「リスト（配列）をソートしたり、逆順にしたりするには（sort ／ reverse メソッド、sorted ／ reversed 関数）」を参照のこと。

リスト（配列）を初期化するには（[]、list関数、リスト内包表記）

Python では配列は「リスト」というデータ構造として実装されている。list 関数やリスト内包表記などを使って、これを初期化する方法をまとめて紹介する。

（2020 年 12 月 04 日）

```
# リスト（配列）の初期化：角かっこ「[]」で囲み、要素をカンマ「,」区切りで並べる
empty_list = []  # 空のリスト
print(empty_list)  # []

int_list = [0, 1, 2]  # 整数リスト（整数配列）
print(int_list)  # [0, 1, 2]

mylist = [0, 'abc', 1, 'def']  # リスト（配列）には任意の型の要素を格納できる
print(mylist)  # [0, 'abc', 1, 'def']

# リスト（配列）の初期化：list 関数を呼び出す
int_list = list()  # 空のリスト（配列）
print(int_list)  # []

int_list = list((4, 5, 6))  # タプルの要素を基にリスト（配列）が作成される
print(int_list)  # [4, 5, 6]

int_list = list(range(5))  # range オブジェクトから整数リスト（整数配列）を作成
print(int_list)  # [0, 1, 2, 3, 4]

str_list = list('python')  # 文字列の各文字を要素とするリストを作成
print(str_list)  # ['p', 'y', 't', 'h', 'o', 'n']

# リスト内包表記
int_list = [x for x in range(0, 10, 2)]  # range オブジェクトからリストを作成
print(int_list)  # [0, 2, 4, 6, 8]

int_list = [x for x in range(10) if x % 2 == 1]  # if 節を用いる例
print(int_list)  # [1, 3, 5, 7, 9]
```

```
# if else 式を使うときには for 節に続けずに、内包表記の先頭に記述する
str_list = [c.upper() if c.islower() else c.lower() for c in 'AbCdE']
print(str_list)  # ['a', 'B', 'c', 'D', 'e']

# リストのリスト（配列の配列）
mylist = [[0, 1, 2], [3, 4, 5]]  # 2 次元のリスト（配列）の作成
print(mylist)  # [[0, 1, 2], [3, 4, 5]]

mul_tbl = [[x * y for x in range(1, 4)] for y in range(1, 5)]
print(mul_tbl)  # [[1, 2, 3], [2, 4, 6], [3, 6, 9], [4, 8, 12]]
```

リスト（配列）の初期化

多くのプログラミング言語における「配列」は、Python ではリストとして実装されている。ここでは、リスト（配列）を初期化する方法をまとめる。

リスト（配列）を初期化するには、幾つかの方法がある。

- 角かっこ「[]」内に、その要素をカンマ「,」で区切って並べていく
- list 関数にその要素となるものを格納している反復可能オブジェクトを与える
- リスト内包表記を使用する

下の 2 つは後で見るとして、まず角かっこを使用する方法を見ていこう。要素を持たない空のリスト（空の配列）を作成するには、角かっこに何も含めないか、後で見る list 関数に引数を指定しないで呼び出せばよい。

```
# 空のリスト（配列）の作成
empty_list = []
print(empty_list)  # []

empty_list = list()
print(empty_list)  # []
```

リストに要素を含めるには、それらをカンマで区切って並べていく。

```
int_list = [1, 2, 3]
print(int_list)  # [1, 2, 3]

str_list = ['foo', 'bar', 'baz']
print(str_list)  # ['foo', 'bar', 'baz']
```

その要素は、インデックスを用いてアクセスしたり、for 文で列挙したりすることで使用できる。

```
print(int_list[0])  # 先頭要素にアクセス：1

print(str_list[-1])  # 末尾要素にアクセス：baz

for item in str_list:
    print(item)  # foo、bar、baz が順番に表示される
```

リストには異なる型の要素を混在させて、格納できる。他の言語では、例えば int 型の配列には整数しか格納できない、ということがよくあるが、Python のリストではそういうことはない。

```
mylist = [1, 'foo', 2, 'bar']  # 整数と文字列を要素とするリスト
```

とはいえ、リストは繰り返し処理でその要素を扱ったり、要素を一括して処理したりすることを念頭に置いたデータ構造なので、例えば整数のみを含むリスト、あるいは複数の項目で構成されるタプルを要素とするリスト、さらに構造化を進めて、何らかのクラスのインスタンスを要素とするリストのように、同種の項目の集まりを格納するために使用するのが好ましい。

list 関数

リスト（配列）は、list 関数を使っても作成できる。このときには、既に見たように引数を指定しなければ空のリストが作成される。そうでないときには、リストの要素となるものを格納している反復可能オブジェクトを引数に指定する。以下に例を示す。

```
empty_list = list()  # 空のリスト（配列）の作成

int_tuple = (0, 1, 2)
int_list = list(int_tuple)  # 反復可能オブジェクト（タプル）からリストを作成
print(int_list)  # [0, 1, 2]
```

連続する整数配列を作成するときには、range 関数と list 関数を組み合わせるのが一般的といえる。

```
int_list = list(range(5))
print(int_list)  # [0, 1, 2, 3, 4]
```

ただし、range 関数が返す range オブジェクトは、リスト（配列）よりもメモリ消費の点で効率的である。整数列をリストとして持っている必要がなければ、多くの場所では range オブジェクトを使った方がよいかもしれない。

なお、文字列もまた反復可能オブジェクトである。よって、文字列を構成する各文字を要素とするリストは次のようにすることで簡単に作成できる。

```
my_str = 'python'
str_list = list(my_str)
print(str_list)  # ['p', 'y', 't', 'h', 'o', 'n']
```

リスト内包表記

Python ではリスト内包表記という方法でもリスト（配列）を初期化できる。その基本構文は次のようになっている。

```
[ 変数を使って要素の値を計算する式 for 変数 in 反復可能オブジェクト ]
```

実際にはこれは、以下の for 文と同じ意味となる。

```
結果を保存するリスト = []
for 変数 in 反復可能オブジェクト :
    結果を保存するリスト .append( 変数を使って要素の値を計算する式 )
```

以下に例を示す。

```
int_list = [x * 2 for x in range(5)]
print(int_list)  # [0, 2, 4, 6, 8]
```

上の for 文での置き換えに従うと、これは次のコードと同等ということだ。

```
int_list = []
for x in range(5):
    int_list.append(x * 2)

print(int_list)  # [0, 2, 4, 6, 8]
```

リスト内包表記の for 節には続けて if 節を記述できる。これは反復可能オブジェクトから取り出した値が特定の条件に合致するときにだけ、「変数を使って要素の値を計算する式」の評価を行うために使用する。以下に例を示す。

```
int_list = [x for x in range(10) if x % 2]
print(int_list)  # [1, 3, 5, 7, 9]
```

上記コード例の if 節では「if x % 2」と条件を指定している。これは反復可能オブジェクトである range(10) オブジェクトから変数 x に取り出した値が奇数のときにだけ、「変数を使って要素の値を計算する式」である「x」を評価することを意味する（奇数であれば「if x % 2」の値は 1 となり、Python では 1 は真と見なされるので、ここでは「== 1」という記述は省略している）。「x」という式は取り出した値そのものなので、ここでは「[1, 3, 5, 7, 9]」という結果が得られる。

その一方で、変数 x に取り出した値が条件に合致しないときにも、何らかの値を算出したいという場合がある。そうしたときには、for 節に続けて if 節を記述するのではなく、要素の値を計算する式の中で三項演算子（if 式）を記述する。

```
str_list = [c.upper() if c.islower() else c.lower() for c in 'AbCdE']
print(str_list)  # ['a', 'B', 'c', 'D', 'e']
```

ここでは、反復可能オブジェクトとして文字列 'AbCdE' を用いている。新しく作成するリストの要素の値を計算する式は「c.upper() if c.islower() else c.lower()」だ。これは、文字列の個々の文字が小文字かどうかを調べて、小文字であればそれを upper メソッドで大文字化して、そうでなければ lower メソッドで小文字化する。よって、元の文字列とは大文字小文字が反転したものを要素とするリストが得られるということだ。

リストのリスト（配列の配列）

最後にリストのリスト（配列の配列）、つまりリストを要素とするリストの初期化についても簡単に見ておく。といっても、難しいことはなく、角かっこの中にさらに角かっこを使って記述していくだけだ。あるいは、角かっこの中に既存のリストを置いてもよい。

```
int_list_list = [[0, 1, 2], [3, 4, 5]]
print(int_list_list)  # [[0, 1, 2], [3, 4, 5]]

l1 = [0, 1, 2]
l2 = [3, 4, 5]
int_list_list = [l1, l2]
print(int_list_list)  # [[0, 1, 2], [3, 4, 5]]
```

ただし、後者のように既存のリストを、新しいリストの要素とするときには注意が必要だ。以下のコードを見てほしい。

```
l1 = [0, 1, 2]
l2 = [3, 4, 5]
int_list_list = [l1, l2]

l1[0] = -1
print(int_list_list)  # [[-1, 1, 2], [3, 4, 5]]

int_list_list[1][0] = 100
print(l2)  # [100, 4, 5]
```

ここでは変数 l1 と l2 に保存されているリストを要素として、リストのリストを作成しているが、l1 の要素を書き換えた結果が int_list_list に影響し、逆に int_list_list の要素を書き換えた結果が l2 に影響している。これは、リストの要素は他のオブジェクトへの参照となっているからだ。この点には注意が必要だ。

リスト内包表記でリストのリストを作成するには、リスト内包表記の内部でもう一度リスト内包表記を使用する。以下に例を示す。

```
mul_tbl = [[x * y for x in range(1, 5)] for y in range(1, 4)]
print(mul_tbl)  # [[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12]]
```

上のコードと同等なコードを for 文で書き下すと次のようになる。

```
mul_tbl = []
for y in range(1, 4):
    tmp = []
    for x in range(1, 5):
        tmp.append(x * y)
    mul_tbl.append(tmp)

print(mul_tbl)  # [[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12]]
```

このコードを見ると、結果のリストのリストが 3 行 4 列となっている理由が分かるはずだ。つまり、最初の for 文では反復可能オブジェクトが range(1, 4) となっているので、1 ～ 3 の各値が変数 y に代入される。外側の for 文は 3 回実行されるということだ。内側の for 文では反復可能オブジェクトは range(1, 5) なので、ループは 4 回実行される。

よって、外側のループの 1 回目では、リスト tmp には 1×1、1×2、1×3、1×4 が追加されて、ループの終了時にこの 4 要素のリストがリスト mul_tbl に追加される。次のループでは 2×1、2×2、2×3、2×4 を要素とするリストが mul_tbl に追加される。といった具合に処理が行われ、結果として 3 行 4 列のリストが作成される。

リスト（配列）の要素にインデックスやスライスを使ってアクセスするには

インデックスやスライスを使って、リスト（配列）の要素を取得、削除、変更する方法を紹介する。

（2020 年 12 月 08 日）

インデックスとスライスの基本

```
mylist = list(range(10))
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# インデックスによるアクセス
n = mylist[1]  # 正数は先頭からのインデックス。先頭の要素のインデックスが 0
print(n)  # 1
n = mylist[-2]  # 負数は末尾からのインデックス。末尾の要素のインデックスが -1
print(n)  # 8

# スライスによるアクセス
mylist = list(range(10))

s = mylist[0:5]  # 0 ～ 4 番目の 5 要素を取り出す
print(s)  # [0, 1, 2, 3, 4]
```

Python では、他の言語における配列はリストとして実装されている。その要素には、インデックスやスライスといった機構を通じてアクセスできる。その基本は次のようになっている。

- インデックス：角かっこ「[]」内にアクセスしたい要素のインデックスを指定
- スライス：角かっこ「[]」内にアクセスしたい要素の範囲を「lower_bound:upper_bound:stride」の形で記述する。lower_bound はリスト中でスライスの始まる位置を、upper_bound はスライスの終わる位置を、stride は増分を表す（いずれも省略可能）

スライスの場合、upper_bound に指定するインデックス位置にある要素はスライスの要素には含まれず、その 1 つ手前の要素までがスライスに含まれることには注意が必要だ。詳細については「インデックスによるアクセス」および「スライスによるアクセス」を参照されたい。

インデックスを使ったリストの操作

```
mylist = list(range(10))

# インデックスを使った要素の取り出し
n = mylist[7]
print(n)  # 7

# インデックスを使った要素の削除
del mylist[4]
print(mylist)  # [0, 1, 2, 3, 5, 6, 7, 8, 9]

# インデックスを使った要素の変更
mylist[-1] = 90
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 90]
```

インデックスを使って、要素を取り出すときにはその値が必要な場所で、「リスト [取り出したい要素のインデックス]」と記述する。特定の要素の値を削除するには del 文を使用して「del リスト [削除したい要素のインデックス]」のように書く。値を変更するには、代入文で「リスト [値を変更したい要素のインデックス] = 新しい値」のようにする。詳細については「インデックスによるアクセス」を参照のこと。

スライスを使ったリストの操作

```
# スライスによるアクセス
mylist = list(range(10))
s = mylist[:]  # 全要素を取り出す
print(s)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

s = mylist[5:]  # 5 番目以降の全要素を取り出す
print(s)  # [5, 6, 7, 8, 9]

s = mylist[:5]  # 0 ～ 4 番目の 5 要素を取り出す
print(s)  # [0, 1, 2, 3, 4]

s = mylist[0:10:2]  # 0、2、4、6、8 番目の要素を取り出す
print(s)  # [0, 2, 4, 6, 8]
```

```
s = mylist[::2]  # 同上
print(s)  # [0, 2, 4, 6, 8]

s = mylist[-4:]  # -4 番目から末尾までの全要素を取り出す
print(s)  # [6, 7, 8, 9]

s = mylist[-1:-9:-1]  # リスト末尾から逆順に要素を取り出す
print(s)  # [9, 8, 7, 6, 5, 4, 3, 2]

# スライスを使った要素の削除
mylist = list(range(10))

del mylist[3:5]  # 3 番目と 4 番目の要素を削除
print(mylist)  # [0, 1, 2, 5, 6, 7, 8, 9]

del mylist[0::2]  # 上の削除結果から、0、2、4、6 番目の要素を削除
print(mylist)  # [1, 5, 7, 9]

mylist[0:2] = []  # 上の削除結果から先頭の 2 要素を削除
print(mylist)  # [7, 9]

# スライスを使った要素の変更
mylist = list(range(10))

mylist[1:3] = [10, 20]  # スライスの先頭と末尾を指定して要素を変更
print(mylist)  # [0, 10, 20, 3, 4, 5, 6, 7, 8, 9]

mylist[1:3] = [1, 1.5, 2] # この場合はスライスと新しい値の要素数が異なっても OK
print(mylist)  # [0, 1, 1.5, 2, 3, 4, 5, 6, 7, 8, 9]

mylist = list(range(10))
mylist[1::2] = [10, 30, 50, 70, 90]  # 先頭／末尾／増分を指定して要素を変更
print(mylist)  # [0, 10, 2, 30, 4, 50, 6, 70, 8, 90]

mylist[0::2] = [0.0, 2.0, 4.0]  # ValueError：このときには同じ要素数でないとダメ
```

スライスを使って、リストから特定の範囲の要素を取り出すには、それらが必要となる部分で「リスト [開始位置 : 終了位置 : 増分]」のように記述する。特定の範囲に含まれる要素を削除するには del 文を使って「del リスト [開始位置 : 終了位置 : 増分]」と書くか、代入文で「リスト [開始位置 : 終了位置 : 増分] = []」のようにスライスに空のリストを代入する。スライスに含まれる要素を変更するには、代入文で「リスト [開始位置 : 終了位置 : 増分] = [新しい値を含んだリスト]」のようにする。ただし、注意点がある。詳細については「スライスによるアクセス」を参照してほしい。

インデックスによるアクセス

Python のリスト（配列）に格納される要素にアクセスしたり利用したりするには幾つかの方法がある。その中でも基本的なのが、以下で紹介するインデックスやスライスを用いて、要素を取り出したり、変更したりすることだ。以下では、まずインデックスによる要素の取得、削除、変更について見ていこう。

インデックスによる要素の取得

インデックスによるアクセスでは、角かっこ「[]」内にアクセスしたい要素のインデックスを指定するだけだ（0 始まり）。以下に例を示す。なお、以下では数値のみを要素とするリスト（整数リスト、整数配列）を例とする。

```
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# インデックスによる要素の読み出し
n = mylist[7]  # 7 番目の要素の取得（0 始まり）
print(n)  # 7

n = mylist[10]  # IndexError：要素がないインデックスにアクセスすると例外
```

範囲外へのアクセスは上のコード例に示した通り、IndexError 例外となる。

全ての要素に順次アクセスして、その値を使うために、次のようなコードを書きたくなるかもしれない。

```
for idx in range(len(mylist)):
    print(mylist[idx])
```

このコードでは、リスト mylist の要素数を len 関数で取得して、range 関数を用いてその数だけループを行っている。だが、Python ではリスト（配列）の値を使用するだけであれば、上のようなコードではなく、以下のようなコードとするのが一般的だ。

```
for num in mylist:
    print(num)
```

こちらのコードでは、リスト mylist に格納されている要素をループ変数 num に列挙して、それをループ内で利用している。この方がコードはシンプルで見やすくなるだろう。

インデックスには負値も指定できる。このとき、-1 は末尾の要素を表し、「-len(リスト)」が先頭の要素を表す。

```
n = mylist[-1]
print(n)  # 9

n = mylist[-len(mylist)]
print(n)  # 0
```

インデックスを使った要素の削除

特定のインデックスの要素を削除するには、del 文で「del リスト [インデックス]」と記述する。

```
idx = mylist.index(3)  # mylist 中で要素 3 のインデックスを検索
del mylist[idx]  # 要素 3 を削除
print(mylist)  # [0, 1, 2, 4, 5, 6, 7, 8, 9]
```

リストの pop メソッドにインデックスを渡しても同様な処理は可能だ。

インデックスを使った要素の値の変更

インデックスを用いて、要素の値を変更するには、代入文で「リスト [インデックス] = 新しい値」と記述する。

```
mylist = list(range(10))
mylist[9] = 90
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 90]
```

なお、for 文を用いて、リストの要素を変更したいときには、以下のようなコードだとうまくいかない。

```
mylist = list(range(10))
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for num in mylist:
    num *= 2  # 各要素の値を 2 倍したつもりだが……

print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  # 元のリストに変わりはない
```

この例ではループのたびに、ループ変数 num が、対応するインデックスにある mylist の要素を参照するようになるが、その後の代入（num *= 2）により、num は mylist の要素とは別の値を参照するようになるからだ。for ループを使って、リストの要素を変更したいのであれば、先ほどはお勧めしなかったインデックスをループ変数に代入していく方法をとる。

```
for idx in range(len(mylist)):
    mylist[idx] *= 2

print(mylist)  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# 参考：mylist = list(map(lambda x: x * 2, mylist))
```

上記コードの最終行のコメントも参考にされたい（map 関数を用いて、リストの各要素の値を 2 倍した新しいリストを作成している）。

スライスによるアクセス

Python のリスト（配列）では「ここからここまでの要素」を指定できる。これをスライスと呼ぶ。スライスはインデックスと同様に角かっこ「[]」で囲んで「[lower_bound:upper_bound:stride]」のように指定する。lower_bound はスライスがリストのどの要素から始まるかを、upper_bound はスライスがリストのどの要素（の手前の要素）で終わるかを、stride は増分を指定する。

スライスの指定では、lower_bound ／ upper_bound ／ stride はいずれも省略可能だが、lower_bound に続くコロン「:」だけは必須である。lower_bound ／ stride を省略した場合、lower_bound の値は「0」と、stride の値は「1」と見なされると考えられる。また、upper_bound を省略したときには、リストの末尾までが範囲に含められるような指定を行ったものと考えられる。

これらから、スライスを表す最低限の表記は「[:]」となり、その場合は「[0: 末尾まで含む :1]」を指定したものと考えられる（先頭から末尾までの全要素。「末尾まで含む」までは便宜的な表現）。

スライスに含まれる要素のインデックスを idx とすると、「idx ＝ lower_bound ＋ stride×i」（i は 0 以上の整数）として計算でき、その範囲は「lower_bound ≦ idx ＜ upper_bound」（stride が正値の場合）もしくは「lower_bound ≧ idx ＞ upper_bound」（stride が負値の場合）となる。いずれにしても、upper_bound に指定するインデックスの値は、スライスには含まれないことに注意。

スライスによる要素の取得

以下にスライスを用いてリストの要素を取得する例を幾つか示す（コメントでは lower_bound を「lower」と、upper_bound を「upper」と表記する）。

```
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

s = mylist[:]  # lower ＝ 0、upper ＝末尾まで、stride ＝ 1：全要素
print(s)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

s = mylist[0:5]  # lower ＝ 0、upper ＝ 5、stride ＝ 1：0 ～ 4 番目までの 5 要素
print(s)  # [0, 1, 2, 3, 4]

s = mylist[5:]  # lower ＝ 5、upper ＝末尾まで、stride ＝ 1：5 番目から末尾までの要素
print(s)  # [5, 6, 7, 8, 9]
```

```
s = mylist[:5]  # lower=0、upper = 5、stride = 1：0 ～ 4 番目までの 5 要素
print(s)  # [0, 1, 2, 3, 4]

s = mylist[0:10:2]  # lower = 0、upper = 10、stride = 2：0、2、4、6、8 番目の要素
print(s)  # [0, 2, 4, 6, 8]

s = mylist[::2]  # lower = 0、upper =末尾まで、stride = 2：0、2、4、6、8 番目の要素
print(s)  # [0, 2, 4, 6, 8]
```

最後の例のように、スライスの指定では lower_bound も upper_bound も（もちろん、stride も）省略できることは覚えておこう。この例では、stride = 2 としているので、その上のコード例と同じスライスが得られている。

スライスの範囲の指定方法によっては、要素が存在しない範囲となることもある。

```
s = mylist[5:0]  # lower = 5、upper = 0、strider = 1
print(s)  # []
```

この例では、lower_bound の値が 5、upper_bound の値が 0、stride の値が 1 となる。先ほどの計算式からスライスに含まれる要素のインデックスは「idx = 5 + 1×i」で、その範囲は「5 ≦ idx < 0」となる。これに該当する idx は存在しないので、スライスに含まれる要素も存在しない。このときには、スライスは []（空のリスト）となる。

スライスに負値を指定するときには、少し注意が必要だ。-1 は末尾の要素を指すインデックスだが、これを upper_bound に指定した場合、末尾の要素はスライスには含まれなくなる。以下に例を示す。

```
s = mylist[-4:-1]  # lower = -4、upper = -1、stride = 1：-4 ～ -2 番目までの 3 要素
print(s)  # [6, 7, 8]：ここでは -4 番目の要素は 6 番目の、-2 番目の要素は 8 番目の要素

s = mylist[-4:]  # lower = -4、upper =末尾まで、stride = 1：-4 番目から末尾までの要素
print(s)  # [6, 7, 8, 9]
```

この 2 つのスライス指定では、upper_bound に -1 を指定しているかどうかが異なる点だが、1 つ目のコードでは末尾の要素が含まれず、2 つ目のコードでは末尾の要素が含まれている。「末尾まで」と思って -1 を upper_bound に指定するとビックリする結果となるかもしれない。

stride に負値を指定すると、リストの末尾から順に要素が取り出されることも覚えておこう。以下に例を示す。

```
s = mylist[-1:-9:-1]  # lower=-1、upper = -9、stride = -1：リスト末尾から逆順
print(s)  # [9, 8, 7, 6, 5, 4, 3, 2]
```

このときにも upper_bound に指定したインデックスの要素は取り出されるリストには含まれないので注意すること。この例では -9 番目の要素とは、1 番目の要素「1」であり、上の実行結果を見ると、これがスライスには含まれていないことが分かる。

スライスを使って逆順に全ての値を取り出すには次のように「-1×（要素数＋ 1）」を指定する必要がある（実際には、reversed 関数や reverse メソッドを使うのが一般的だと思われる）。

```
s = mylist[-1:-(len(mylist)+1):-1]
print(s)  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

スライスを使った要素の削除

スライスを用いることで、リスト（配列）に格納されている複数の要素をまとめて削除できる。これを行うには、幾つかの方法がある。

1 つはスライスした範囲を del 文で削除する方法だ。以下に例を示す。

```
mylist = list(range(10))

del mylist[3:5]  # 3 番目と 4 番目の要素を削除
print(mylist)  # [0, 1, 2, 5, 6, 7, 8, 9]

del mylist[0::2]  # 上の削除結果から、0、2、4、6 番目の要素を削除
print(mylist)  # [1, 5, 7, 9]
```

上の 1 つ目の例では、連続する範囲を del 文で削除している。2 つ目の例では、1 つ飛びの範囲を削除している。

連続する範囲のスライスについては、次のように空リストを代入することでも要素を削除できる。

```
mylist[0:2] = []  # 上の削除結果から、最初の 2 要素を削除
print(mylist)  # [7, 9]
```

ただし、元のリストの要素を飛び飛びに選択するように、stride を指定した場合、この方法では例外が発生する（以下の「スライスを使った要素の値の変更」を参照のこと）。

```
mylist = list(range(10))
mylist[::2] = []  # ValueError
```

スライスを使った要素の値の変更

スライスを使って、リストの要素の値を変更するには、代入文で「リスト [スライスの指定] = 新しい値」と記述する。

```
mylist = list(range(10))

mylist[1:3] = [10, 20]  # 1 番目と 2 番目の要素を変更
print(mylist)  # [0, 10, 20, 3, 4, 5, 6, 7, 8, 9]
```

このとき、スライスが連続する範囲となっていれば、代入する値の要素数が、スライスの要素数と異なっていても構わない。

```
mylist[1:3] = [1, 1.5, 2] # スライスの要素数＝ 2、代入する要素数＝ 3：OK
print(mylist)  # [0, 1, 1.5, 2, 3, 4, 5, 6, 7, 8, 9]
```

両者の要素数が異なっていても大丈夫なので、上でも見たように空のリスト（[]）を代入することもできる。この場合には、対応する要素は削除される。

一方、元のリストの要素を飛び飛びに選択するように stride を指定したときには、スライスの要素数と、代入する値の要素数が同じでなければならない点には注意が必要だ。

```
mylist = list(range(10))
mylist[1::2] = [10, 30, 50, 70, 90]  # 両者の要素数が同じ：OK
print(mylist)  # [0, 10, 2, 30, 4, 50, 6, 70, 8, 90]

mylist[0::2] = [0.0, 2.0, 4.0]  # 両者の要素数が異なる：ValueError
```

リスト（配列）に要素を追加するには（+演算子／+=演算子／append／extend／insertメソッド）

各種の演算子／メソッドを使って、リストの末尾に要素を追加したり、リスト内の指定した位置に要素を挿入したりする方法を紹介する。

（2020 年 12 月 11 日）

+ 演算子／ += 演算子によるリスト（配列）末尾への要素の追加（結合）

```
mylist = list(range(5))
print(mylist)  # [0, 1, 2, 3, 4]

# + 演算子によるリストの結合（リストの末尾へのリストの追加）
tmp = mylist + [5, 6]  # + 演算子では新しいリストが作成される
print(tmp)  # [0, 1, 2, 3, 4, 5, 6]
print(mylist)  # [0, 1, 2, 3, 4]

mylist = mylist + 5  # TypeError：リストはリスト以外と結合できない
mylist = mylist + (5, 6)  # TypeError：リストはリスト以外と結合できない

# += 演算子によるリスト末尾への要素の追加
mylist += [5]  # += 演算（累算代入）では元のリストが変更される
print(mylist)  # [0, 1, 2, 3, 4, 5]

mylist += 6  # TypeError：リストへ累算代入できるのは反復可能オブジェクトのみ

mylist += (6, 7)  # リスト末尾にタプルの個々の要素を追加
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7]
```

Python でリスト（配列）に要素を追加するには幾つかの方法がある。+ 演算子を使って、「リスト 1 + リスト 2」とすると、リスト 1 の末尾にリスト 2 の内容が追加されたリストが新たに作成される。+= 演算子を使って、「リスト 1 += リスト 2」とすると、リスト 1 の末尾にリスト 2 の内容が追加される。詳しくは、「+ 演算子または += 演算子を使う方法」を参照。

リストのメソッドを使用した要素の追加

```
mylist = list(range(5))

mylist.append(5)  # リストの末尾に要素を追加
print(mylist)  # [0, 1, 2, 3, 4, 5]

mylist.append(6, 7)  # TypeError：append メソッドの引数は 1 つだけ

mylist.append([6, 7])  # リストの末尾に単一の要素として「リスト」を追加
print(mylist)  # [0, 1, 2, 3, 4, 5, [6, 7]]

mylist.extend([8, 9])  # リストの末尾に反復可能オブジェクトの個々の要素を追加
print(mylist)  # [0, 1, 2, 3, 4, 5, [6, 7], 8, 9]

mylist.extend(10)  # TypeError：extend メソッドには反復可能オブジェクトのみ渡せる

mylist.extend({10: 'foo', 11: 'bar'})  # 反復可能オブジェクトなので OK
print(mylist)  # [0, 1, 2, 3, 4, 5, [6, 7], 8, 9, 10, 11]

mylist.insert(1, 0.5)  # 指定したインデックスに要素を挿入
print(mylist)  # [0, 0.5, 1, 2, 3, 4, 5, [6, 7], 8, 9, 10, 11]
```

リストの末尾に要素を追加するメソッドとしては、append メソッドと extend メソッドが使える。append メソッドは引数を 1 つ受け取り、それをリストの末尾に単一の要素として追加する。extend メソッドは引数に反復可能オブジェクトを 1 つ受け取り、その個々の要素をリストの末尾に順次追加する（リストの拡張）。任意の位置に要素を挿入するには、insert メソッドが使える。詳しくは「append ／ extend ／ insert メソッドを使う方法」を参照。

+ 演算子または += 演算子を使う方法

Python でリスト（配列）の末尾に要素を追加したり、リスト同士を結合したりするには以下のような方法がある（この他にもスライスを使う方法もあるが、ここでは割愛する。スライスを用いたリストの要素へのアクセスについては「リスト（配列）の要素にインデックスやスライスを使ってアクセスするには」を参照されたい）。

- + 演算子または += 演算子を使う
- リストが持つ append ／ extend ／ insert メソッドを使う

以下ではまず、+ 演算子と += 演算子を使う方法から見ていこう。

+ 演算子を使って「リスト 1 + リスト 2」とすると、2 つのリストの内容を結合した新しいリストが作成される。

```
mylist = [0, 1] + [2, 3]  # 1 つ目のリストの末尾に 2 つ目のリストの内容が追加される
print(mylist)  # [0, 1, 2, 3]

tmp = mylist + [4, 5]  # mylist と [4, 5] の要素から成る新しいリストが tmp に代入される
print(tmp)  # [0, 1, 2, 3, 4, 5]
print(mylist)  # mylist は以前のまま
```

注意点としては、リストに結合できるのはリストだけである点だ。例えば、「リスト + 整数値」のようにすると、リストに整数値が要素として追加されるのではなく、TypeError 例外が発生する。

```
mylist + 6  # TypeError
```

+= 演算子を使って「リスト 1 += リスト 2」とすると、リスト 1 の末尾にリスト 2 の内容が追加される（これを「リストの拡張」などと呼ぶこともある）。+ 演算子とは異なり、新しいリストが作成されることはない。

```
mylist = [0, 1]
mylist += [2, 3]  # mylist が拡張されて [0, 1, 2, 3] というリストになる
print(mylist)  # [0, 1, 2, 3]
```

+= 演算子を使って、末尾に整数値などを追加できないのは、+ 演算子と同様だ。その一方で、+ 演算子とは異なり、+= 演算子を使うと、リスト以外の反復可能オブジェクトをリストの末尾に追加できる。

```
# 反復可能オブジェクト以外は += 演算子の右側には置けない
mylist += 4  # TypeError

# タプルの要素をリスト末尾に追加
mylist += (4, 5)
print(mylist)  # [0, 1, 2, 3, 4, 5]

# 辞書のキーをリスト末尾に追加
mylist += {6: 'foo', 7: 'bar'}  # この場合はキーの値が mylist の末尾に追加される
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7]

# 文字列の要素をリスト末尾に追加
mylist += '89'  # 文字列も反復可能オブジェクト
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, '8', '9']
```

辞書も反復可能オブジェクトなので、上記のコードはエラーとはならない。上のコードでは、辞書のキーがリスト末尾に追加される。注意したいのは、文字列は反復可能オブジェクトなので、1 つの文字列を単一の要素としてリストに追加しようとしても、+= 演算子では文字列を構成する個々の文字が 1 つ 1 つの要素としてリストに追加されてしまうところだ。文字列を単一の要素としてリストに追加するのであれば、後述の append メソッドを使用する。

append ／ extend ／ insert メソッドを使う方法

Python ではリストが持つ以下の 3 つのメソッドを使っても、リスト（配列）に要素を追加できる。

- append メソッド：引数に受け取った値を、単一の要素としてリスト末尾に追加
- extend メソッド：引数に受け取った反復可能オブジェクトの要素を、個別の要素としてリスト末尾に追加（リストの拡張）
- insert メソッド：第 1 引数で指定したインデックスに、第 2 引数に渡した値を単一の要素として挿入する

append メソッドと extend メソッドはどちらも、受け取った値をリスト末尾に追加するがその方法が異なる。まず append メソッドから見てみよう。

```
mylist = [0, 1, 2]
mylist.append(3)  # 要素 3 を末尾に追加
print(mylist)  # [0, 1, 2, 3]

mylist.append('45')  # 文字列 '45' を末尾に追加
print(mylist)  # [0, 1, 2, 3, '45']

mylist.append([6, 7])  # 要素 [6, 7] を末尾に追加
print(mylist)  # [0, 1, 2, 3, '45', [6, 7]]
```

1 つ目の例では、append メソッドに整数 3 を指定している。このときには、これが単一の要素としてリスト（mylist）に追加されているのが分かる。次の例では、文字列をリスト末尾に追加している。+= 演算子の例とは異なり、文字列全体が単一の要素としてリスト末尾に追加されているのが分かるはずだ。最後の例では、引数にリスト [4, 5] を指定している。このときには、引数に与えたリストがそのまま単一の要素として追加されている。

そうではなく、[0, 1, 2, 3] というリストに [4, 5] というリストを与えて、[0, 1, 2, 3, 4, 5] というリストにしたいのであれば、extend メソッドを使用する。以下に例を示す。

```
mylist = [0, 1, 2, 3]
mylist.extend([4, 5])  # リストの要素を末尾に追加
print(mylist)  # [0, 1, 2, 3, 4, 5]
```

こちらでは、引数に指定したリストに含まれる 1 つ 1 つの要素が、mylist の末尾に順番に追加されている。このように、append と extend の使い分けの大まかな方針はどのような形でリスト末尾に要素を追加したいかによる。

なお、extend メソッドに渡せるのは反復可能オブジェクトだけである（+= 演算子と同様）。1 個の整数を extend メソッドでリスト末尾に追加するのであれば、次のようにリスト（あるいは他の反復可能オブジェクト）の要素にして渡す必要がある。

```
mylist.extend(6)  # TypeError
mylist.extend([6])
print(mylist)  # [0, 1, 2, 3, 4, 5, 6]
```

その一方で、extend メソッドではリスト以外の反復可能オブジェクトもリスト末尾に追加できる。

```
mylist = [0, 1, 2]

mylist.extend((3, 4))  # タプルの要素がリスト末尾に追加される
print(mylist)  # [0, 1, 2, 3, 4]

mylist.extend({5: 'foo', 6: 'bar'})  # 辞書のキーがリスト末尾に追加される
print(mylist)  # [0, 1, 2, 3, 4, 5, 6]

mylist.extend('78')  # 文字列も反復可能オブジェクト
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, '7', '8']
```

3 つ目の例に示したように、文字列は反復可能オブジェクトなので、その文字列を構成する各文字がリストに 1 文字ごとに追加される点には注意しよう（これも += 演算子と同様）。文字列全体を単一要素としてリスト末尾に追加したいのであれば、既に見たように、append メソッドを使用する（か、文字列全体をリストの要素として、append メソッドに与える）。

append メソッドと extend メソッドはどちらもリストの末尾に要素を追加するものだが、insert メソッドはその名の通り、リスト中の任意の位置に要素を追加するものだ。insert メソッドは第 1 引数に挿入したい位置（インデックス）を、第 2 引数に挿入したい値を指定して呼び出す。

```
mylist = list(range(5))

mylist.insert(3, 2.5)  # インデックス 3 に要素 2.5 を追加
print(mylist)  # [0, 1, 2, 2.5, 3, 4]

mylist.insert(5, [3.25, 3.5])  # 第 2 引数に指定した値は単一の要素として挿入される
print(mylist)  # [0, 1, 2, 2.5, 3, [3.25, 3.5], 4]

mylist.insert(-1, 3.75)  # 負数のインデックスも指定可能
print(mylist)  # [0, 1, 2, 2.5, 3, [3.25, 3.5], 3.75, 4]
```

2 つ目の例に示したように、第 2 引数に指定した値は単一の要素として挿入されることは覚えておこう（append メソッドと同様）。なお、要素が存在しない範囲をインデックスに指定した場合、リストの先頭または末尾に要素が挿入される。

```
mylist = [0, 1, 2]
mylist.insert(10, 10)
print(mylist)  # [0, 1, 2, 10]

mylist.insert(-5, -5)
print(mylist)  # [-5, 0, 1, 2, 10]
```

リスト（配列）から要素を削除するには（del文、remove／clear／popメソッド、リスト内包表記）

インデックスやスライス、各種のメソッドを使用してリストから要素を削除する方法を紹介する。

（2020 年 12 月 15 日）

インデックスやスライスを使って要素を削除する

```
# インデックスを指定して del 文で要素を削除
mylist = list(range(5))  # [0, 1, 2, 3, 4]
del mylist[0]
print(mylist)  # [1, 2, 3, 4]

# スライスを指定して del 文で要素を削除
mylist = list(range(5))  # [0, 1, 2, 3, 4]
del mylist[1:4]  # 1 ～ 3 番目の要素を削除
print(mylist)  # [0, 4]

mylist = list(range(5))  # [0, 1, 2, 3, 4]
del mylist[::2]  # 0、2、4 番目の要素を削除
print(mylist)  # [1, 3]

mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del mylist[1:7:3]  # 1 番目と 4 番目の要素を削除
print(mylist)  # [0, 2, 3, 5, 6, 7, 8, 9]

# スライスで指定した範囲に空のリストを代入することで削除
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
mylist[4:8] = []  # 4 ～ 7 番目の要素を削除
print(mylist)  # [0, 1, 2, 3, 8, 9]

print('mylist[:]:', mylist[:])  # mylist[:]: [0, 1, 2, 3, 8, 9]
mylist[:] = []  # リストの全要素を削除
print(mylist)  # []
mylist = list(range(5))
del mylist[:]  # このようにも書ける
```


→目次に戻る

del 文でリスト（配列）の要素を削除するには、「del リスト [インデックスまたはスライスの指定]」とする。インデックスを指定したときには、対応する要素が 1 つ削除される。スライスを指定したときには、その範囲に含まれる要素が全て一括で削除される。あるいは、リストに対してスライスを指定して、そこに空のリストを代入してもよい。詳しくは「インデックスやスライスを使う方法」を参照のこと。

clear ／ pop ／ remove メソッドを使って要素を削除する

```
# clear メソッドでリストの全要素を削除
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
mylist.clear()
print(mylist)  # []

# pop メソッドで指定したインデックスにある要素をリストから削除して、その値を取得
mylist = list(range(10))
value = mylist.pop()  # インデックスを指定しない場合は末尾の要素が削除される
print('popped value:', value)  # popped value: 9
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4, 5, 6, 7, 8]

value = mylist.pop(5)  # 5 番目の要素を削除
print('popped value:', value)  # popped value: 5
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4, 6, 7, 8]

# remove メソッドで削除したい値を指定して、リストからその値を削除
mylist = [2, 0, 6, 0, 5, 7, 2, 1, 5, 5]

mylist.remove(0)  # リスト内の要素 0 のうち先頭にあるものを削除
print(mylist)  # [2, 6, 0, 5, 7, 2, 1, 5, 5]
mylist.remove(0)  # リスト内の要素 0 のうち先頭にあるものを削除
print(mylist)  # [2, 6, 5, 7, 2, 1, 5, 5]
mylist.remove(0)  # ValueError：指定した値がリスト中にないと例外となる

# リストから特定の条件に合致するものをまとめて削除
mylist = [2, 0, 6, 0, 5, 7, 2, 1, 5, 5]
target = 0  # 要素 0 を mylist から取り除きたい
```

```
mylist = [item for item in mylist if item != target]  # 取り除く＝それ以外を残す
print(mylist)  # [2, 6, 5, 7, 2, 1, 5, 5]

#target = 0  # 上のリスト内包表記は以下と同値
#result = []
#for item in mylist:
#    if item != target:
#        result.append(item)
#
#mylist = result
```

Python のリストには、全要素を削除する clear メソッド、指定したインデックスにある要素を削除してその値を返送する pop メソッド、指定した値を削除する remove メソッドがある。詳細については「clear ／ pop ／ remove メソッドを使う方法」を参照のこと。

インデックスやスライスを使う方法

del 文でリスト（配列）の要素を削除するには「del リスト [インデックスまたはスライスの指定]」と記述する。インデックスを指定したときには、そのインデックスにある要素が削除される。スライスを指定したときには、その範囲に含まれる要素が全て一括で削除される。「リスト [削除範囲の先頭 : 削除範囲の最後] = []」とすることでも要素を削除できる。

まず、インデックスを指定した場合の例を以下に示す。

```
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

del mylist[0]  # 0 番目の要素を削除
print(mylist)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

del mylist[-1]  # -1 番目（末尾）の要素を削除
print(mylist)  # [1, 2, 3, 4, 5, 6, 7, 8]
```

インデックスには、正数と負数のいずれでも指定できる（-1 は末尾の要素を、-2 はその 1 つ手前の要素を指す）。要素が存在しない範囲を指定すると IndexError 例外が発生する。

次に、スライスを指定する場合の例を以下に示す。スライスを使ったリストの要素へのアクセスについては「リスト（配列）の要素にインデックスやスライスを使ってアクセスするには」も参照されたい。

```
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del mylist[1:5]  # 1 ～ 4 番目の要素を削除
print(mylist)  # [0, 5, 6, 7, 8, 9]

mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del mylist[:]  # [:] はリストの全要素を表すスライスなので、全要素が削除される
print(mylist)  # []

mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del mylist[5:]  # 5 番目以降の要素が全て削除される
print(mylist)  # [0, 1, 2, 3, 4]

mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del mylist[:7]  # 0 ～ 6 番目の要素が削除される
print(mylist)  # [7, 8, 9]

mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del mylist[0:5:2]  # 0、2、4 番目の要素が削除される
print(mylist)  # [1, 3, 5, 6, 7, 8, 9]
```

スライスの指定を「[削除範囲の先頭 : 削除範囲の最後]」としたときには、その範囲にある要素が全て削除される（ただし、「削除範囲の最後」に指定したインデックスの要素はスライスには含まれない）。また、「[削除範囲の先頭 : 削除範囲の最後 : 増分]」としたときには、「削除範囲の先頭＋増分 ×i」（i は 0 以上の整数）という式で算出されるインデックスにある要素が一括で削除される（この式で算出されるインデックスは「削除範囲の先頭」と「削除範囲の最後」に含まれるものとする）。

スライスに削除範囲の先頭と最後だけを指定した（増分を指定しなかった）場合には、del 文ではなく、代入文で空のリストをスライスに代入することでも対応する要素を削除できる。以下に例を示す。

```
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

mylist[2:8] = []  # 2 ～ 7 番目の要素を削除
print(mylist)  # [0, 1, 8, 9]

print(mylist[0:3:2])  # [0, 8]
mylist[0:3:2] = []  # ValueError
```

2 つ目の例のように、先頭と最後、増分を指定したスライスでは空のリストを代入しての削除はできない。これは、とびとびのスライスに代入するには、代入先と代入元の要素数が同じでなければならないからだ。

clear ／ pop ／ remove メソッドを使う方法

インデックスとスライス以外にも、以下の 3 つのメソッドを使ってもリストから要素を削除できる。

- clear メソッド：全ての要素を削除
- pop メソッド：指定したインデックスにある要素を削除して、その値を返送する。範囲外のインデックスを指定すると例外となる
- remove メソッド：指定した値がリストにあれば、そのうち先頭（インデックスが最小）のものを削除する。なければ例外となる

これらについて簡単に見ていこう。

まずは clear メソッドからだ。このメソッドはリストに含まれる全要素を一括して削除する。引数はない。

```
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(mylist)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
mylist.clear()  # 全要素を削除
print(mylist)  # []

# 参考：del mylist[:]、mylist[:] = []
```

既に述べたが、「リスト [:]」というスライスはそのリストの全要素を表すので、要素の全削除は「del リスト [:]」や「リスト [:] = []」のようにしても行える。

pop メソッドは、指定したインデックスにある要素を削除して、その値を戻り値として返送する。インデックスを指定しなかった場合には、末尾のリストが取り除かれ、その値が返される。以下に例を示す。

```
mylist = list(range(10))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

value = mylist.pop()  # 末尾の要素を取り出して、削除
print('popped value:', value)  # popped value: 9
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4, 5, 6, 7, 8]

value = mylist.pop(5)  # 5 番目の要素を取り出して、削除
print('popped value:', value)  # popped value: 5
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4, 6, 7, 8]

value = mylist.pop(-2)  # 最後から 2 つ目の要素を取り出して、削除
print('popped value:', value)  # popped value: 7
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4, 6, 8]

value = mylist.pop(10)  # IndexError：範囲外のインデックスは指定できない
```

上に示した通り、インデックスには負数も指定可能だ。また、範囲外のインデックスを指定すると IndexError 例外が発生する。

最後の remove メソッドでは、インデックスではなく、削除したい値を指定する。指定した値がリストに複数格納されているときには、その中で先頭（つまり、インデックスの値が最小）の要素が削除される。

以下に例を示す。

```
mylist = [2, 0, 6, 0, 5, 7, 2, 1, 5, 5]

mylist.remove(0)  # リスト内の要素 0 のうち先頭にあるものを削除
print(mylist)  # [2, 6, 0, 5, 7, 2, 1, 5, 5]
mylist.remove(0)  # リスト内の要素 0 のうち先頭にあるものを削除
print(mylist)  # [2, 6, 5, 7, 2, 1, 5, 5]
mylist.remove(0)  # ValueError：指定した値がリスト中にないと例外となる
```

最初の remove メソッド呼び出しでは、インデックス 1 にある要素 0 が削除され、次の remove メソッド呼び出しで（要素が 1 つ削除された後の新しい）インデックス 2 にある要素 0 が削除されていることに注目してほしい。2 番目の remove メソッド呼び出しにより、mylist には要素 0 が存在しなくなったので、最後の remove メソッド呼び出しでは ValueError 例外が発生している。

最後に、リストから特定の条件に合致する値をまとめて削除することを考えてみよう。例えば、上の例では全ての要素 0 をリストから削除するのに remove メソッドを 2 回呼び出している。この処理は、mylist に要素 0 が 2 つあることと、それを削除するには 2 回 remove メソッドを呼び出せばよいことを人の目で確認していたので、このようになったと考えられる。しかし、こうした「リストから特定の条件に合致する値を削除」という処理を一般化すると、次のようなコードになるだろう（この他にも while 文を使うなど、さまざまな方法が考えられる）。

```
mylist = [2, 0, 6, 0, 5, 7, 2, 1, 5, 5]

target = 0
result = []
for item in mylist:
    if item != target:
        result.append(item)

mylist = result
print(mylist)  # [2, 6, 5, 7, 2, 1, 5, 5]
```

このコードでは、for ループで mylist から要素を取り出して、その要素の値が目的の値（target）と等しくなければ、一時的なリストにその要素を追加して、ループ終了後にそれを元のリストに代入している。注意する点としては、「特定の要素を削除する」ことを「特定の要素以外で構成されるリストを作成する」ことに置き換えている点だ。

そして、上記のコードは実はリスト内包表記にまとめることもできる。

```
mylist = [2, 0, 6, 0, 5, 7, 2, 1, 5, 5]
target = 0

mylist = [item for item in mylist if item != target]
print(mylist)  # [2, 6, 5, 7, 2, 1, 5, 5]
```

特定の条件に合致する値をまとめて削除するのであれば、このようなリスト内包表記の方が、remove メソッドを使ったり、del 文を使ったりして何度も要素を削除するよりも効率的で読みやすいコードになるだろう。

```
[ 残したい要素 for 残したい要素 in 元のリスト if 削除したい条件の否定（反転）]
```

リスト（配列）から要素を検索するには（in／not in演算子、count／indexメソッド、min／max関数）

リスト（配列）に特定の値が含まれているか含まれていないか、特定の値が何個含まれているか、特定の値がどこにあるか、最大値と最小値は何かを検索するための方法を紹介する。

（2020 年 12 月 18 日）

```
mylist = [2, 5, 11, 15, 1, 3, 18, 3, 19, 8]  # 10 個の整数値を要素とするリスト

# 指定した値がリストに含まれているか含まれていないかを調べる
result = 0 in mylist  # 整数値 0 が mylist に含まれているかを調べる
print(result)  # False

result = 0 not in mylist  # 整数値 0 が mylist に含まれていないかを調べる
print(result)  # True

# 指定した値がリストに何個含まれているかを調べる
cnt = mylist.count(3)  # 整数値 3 が mylist に何個含まれているかを調べる
print(cnt)  # 2

# 指定した値がどのインデックスにあるかを検索
idx = mylist.index(3)  # mylist に含まれる整数値 3 の最小のインデックスを求める
print(idx)  # 5

idx = mylist.index(3, 6, len(mylist))  # 検索範囲を指定
print(idx)  # 7

idx = mylist.index(100)  # ValueError：存在しない値を指定

# リストの要素で最大／最小のものを求める
max_value = max(mylist)  # 最大値を求める
print(max_value)  # 19

min_value = min(mylist)  # 最小値を求める
print(min_value)  # 1
```

```
max_value = max([])  # ValueError：max 関数／ min 関数に空のリストを渡すと例外
max_value = max([], default='none')  # defalut 引数は空リストを渡したときの戻り値
print(max_value)  # none

max_value = max([0, 1], [4, 5], [2, 3])  # 複数のリストの中で最大のリストを取得
print(max_value)  # [4, 5]

mylist = ['pYthon', 'pyThon', 'Python', 'PYTHON']
max_value = max(mylist, key=str.lower)  # key 引数を指定する例
print(max_value)  # pYthon
```

指定した値がリストに含まれているか含まれていないかを調べる

リスト（配列）に特定の値が含まれているか含まれていないかを調べるだけであれば、in 演算子／ not in 演算子を使える。in 演算子はその左側に置いた被演算子が右側に置いた被演算子であるリストに含まれていれば True を、そうでなければ False を返す。not in 演算子はその反対の動作をする。

```
mylist = [15, 8, 1, 0, 20, 19, 1, 2, 13, 7]

# 整数値 0 が mylist に含まれているかどうかを調べる
result = 0 in mylist
print(result)  # True

# 整数値 5 が mylist に含まれているかどうかを調べる
result = 5 in mylist
print(result)  # False

# 整数値 0 が mylist に含まれていないかどうかを調べる
result = 0 not in mylist
print(result)  # False

# 整数値 5 が mylist に含まれていないかどうかを調べる
result = 5 not in mylist
print(result)  # True
```

最初の 2 つの例は整数値 0 または 5 がリスト mylist に含まれているかどうかを調べている。次の 2 つの例は同じ 2 つの値が含まれていないかどうかを調べている。整数値 0 について調べたとき、mylist にこれは含まれているので、in 演算子の結果は True で、not in 演算子の結果は False となる。整数値 5 はリストには含まれていないので、in 演算子の結果は False で、not in 演算子の結果は True となっている。

in 演算子と not in 演算子はあくまでも、リストに特定の要素が含まれているかどうか、つまり存在確認を行うだけで、その値が何個含まれているかや、含まれているとしてどのインデックスに存在しているかなどまでを知るには、以下で紹介する count メソッドや index メソッドを使用する。

指定した値がリストに何個含まれているかを調べる

リスト（配列）に、ある要素が何個含まれているかを調べるには count メソッドを使用する。引数には、何個含まれているかを調べたい値を 1 つだけ指定する。

```
mylist = [15, 8, 1, 0, 20, 19, 1, 2, 13, 7]

cnt = mylist.count(1)  # 整数値 1 が mylist に何個含まれているかを調べる
print(cnt)  # 2

cnt = mylist.count(100)  # 指定した値が含まれていないと 0 が返される
print(cnt)  # 0
```

指定した値がどのインデックスにあるかを検索

index メソッドは、引数に指定した値がリストのどの位置（インデックス）にあるかを返す。その値がリストに存在しない場合は ValueError 例外が発生する。また、第 2 引数と第 3 引数には、検索を開始する位置と終了する位置を指定できる。

以下に例を示す。

```
mylist = [2, 5, 11, 15, 1, 3, 18, 3, 19, 8]

idx = mylist.index(3)  # mylist に含まれている整数値 3 の最小のインデックスを調べる
print(idx)  # 5

idx = mylist.index(3, 6, len(mylist))  # 検索範囲を指定
print(idx)  # 7

idx = mylist.index(100)  # ValueError：存在しない値を指定
```

最初の例では、リスト mylist から整数値 3 のインデックスを検索している。mylist には整数値 3 が 2 つ含まれていて、そのインデックスは 5 と 7 だ。そして、index メソッドの戻り値は 5 となっている。index メソッドは、リスト中で指定した値が最初に登場するインデックスの値を返す仕様になっている。

次の例は、同じく整数値 3 があるインデックスを mylist から検索しているが、今度は第 2 引数と第 3 引数に検索開始位置と検索終了位置を指定している。ここでは末尾までを検索するように第 3 引数に「リストの長さ」を指定している（末尾までを検索するのであれば、第 3 引数の指定は不要だが、ここでは検索終了位置を指定する例として明示した）。

最後の例は、リストには含まれないものを指定した場合だ。このときには、既に述べた通り、ValueError 例外が発生する。

上で見た count メソッドと index メソッドを使うと、リスト内にある特定の値のインデックスを列挙できる。

```
mylist = [2, 5, 11, 15, 1, 3, 18, 3, 19, 8]

tgt = 3
idx = -1
result = []
for cnt in range(mylist.count(tgt)):
    idx = mylist.index(tgt, idx+1)
    result.append(idx)

print(result)  # [5, 7]
```

リスト内包表記を使うと次のようになる。ただし、代入式を使っているので、このコードを実行できるのはPython 3.8 以降となる。

```
mylist = [2, 5, 11, 15, 1, 3, 18, 3, 19, 8]
idx = -1
tgt = 3

result = [idx := mylist.index(tgt, idx+1) for cnt in range(mylist.count(tgt))]
print(result)  # [5, 7]
```

リストの要素で最大／最小のものを求める

リスト（配列）に格納されている要素から最大値を検索するには Python に組み込みの max 関数を、最小値を検索するには min 関数を使用する。

max 関数と min 関数の基本的な使い方は、リストを 1 つだけ渡すものだ。このとき、空のリストを渡すと ValueError 例外が発生することには注意しよう。以下の例では、空のリストを明示的に渡しているので、「こんなことをするはずがない」と思うだろうが、何かの関数を呼び出して得たリストを max 関数／ min 関数に渡すというときには、それが空である可能性はある。

```
mylist = [2, 5, 11, 15, 1, 3, 18, 3, 19, 8]  # 10 個の整数値を要素とするリスト

vmax = max(mylist)  # 最大値を求める
print(vmax)  # 19

min_value = min(mylist)  # 最小値を求める
print(min_value)  # 1

max_value = max([])  # ValueError：空のリストを渡すと例外となる
min_value = min([])  # ValueError：空のリストを渡すと例外となる
```

空のリストを渡してしまったときに、例外ではなく何らかの値を受け取りたいのであれば、default キーワード引数を指定する。

```
max_value = max([], default='none')  # defalut 引数は空リストを渡したときの戻り値
print(max_value)  # none

min_value = min([], default='none')  # defalut 引数は空リストを渡したときの戻り値
print(min_value)  # none
```

default キーワード引数の値をどうするかは難しい。というのは、Python ではインデックスを表す整数には正値でも 0 でも負値でも使えるからだ。範囲外のインデックスとなるような値を使うのは一案かもしれないが、Python 3.8 以降では default キーワード引数の値として None 値を指定できるようになっているので、上の例では文字列の 'none' としたが、None 値を使うのがよいかもしれない。

max 関数と min 関数には複数のリストを渡すことも可能だ。この場合には、リスト同士で大小比較を行い最大のリスト／最小のリストが決定される。

以下に例を示す。

```
max_value = max([0, 1], [4, 5], [2, 3])  # 複数のリストの中で最大のリストを取得
print(max_value)  # [4, 5]

min_value = min([0, 1], [4, 5], [2, 3])  # 複数のリストの中で最小のリストを取得
print(min_value)  # [0, 1]
```

最後に、max 関数／ min 関数には key キーワード引数を指定して、要素の大小比較に何らかのロジックを介入させられる。key キーワード引数には引数を 1 つ持つ関数（やラムダ式）を指定する。これは例えば、英単語を要素とするリストで、大文字／小文字を無視して大小比較を行うといったときに役立つ。

以下に例を示す。

```
mylist = ['pYthon', 'pyThon', 'Python', 'PYTHON']

max_value = max(mylist)  # 辞書式順序で最大の要素を検索
print(max_value)  # pyThon

max_value = max(mylist, key=str.lower)  # 全てを小文字化すると
print(max_value)  # pYthon

min_value = min(mylist)  # 辞書式順序で最小の要素を検索
print(min_value)  # PYTHON

min_value = min(mylist, key=str.lower)  # 全てを小文字化すると
print(min_value)  # pYthon
```

この例では、「python」という単語内で大文字／小文字を混在させたものを要素とするリストを対象に最大の要素、最小の要素を調べている。Python では文字列を比較する際に、大文字が小文字よりも小さく、アルファベット順で前にあるものが後ろにあるものよりも小さくなる。

そのため、key キーワード引数を指定していない最初の max 関数呼び出しでは、大文字の「P」で始まるものよりも小文字の「p」で始まるものが大きくなり（「pYthon」か「pyThon」が最大）、さらに 2 文字目の「Y」と「y」の大小関係は小文字の方が大きいので、結果、「pyThon」が最大値となっている。

これに対して、key キーワード引数に str.lower メソッドを指定して、全てを小文字化して大小比較を行っている 2 つ目の max 関数では全ての要素が「python」として比較される。このときには、全ての要素が等しくなるが、そのときには等しい要素の中で先頭にあるものが戻り値となる（仕様）。そのため、戻り値は「pYthon」となっている。

min 関数でも同様なので、こちらについては説明を省略する。

リスト（配列）をソートしたり、逆順にしたりするには（sort／reverseメソッド、sorted／reversed関数）

sort メソッドや sorted 関数でリスト（配列）の要素をソートしたり、reverse メソッドや reversed 関数で要素を逆順に並べたりする方法を紹介する。

（2020 年 12 月 22 日）

```
# リストの要素をソートする（インプレース）
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
mylist.sort()
print(mylist)  # [0, 1, 2, 5, 6, 7, 9, 15, 16, 18]

# リストの要素をソートした新しいリストを作成する
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
newlist = sorted(mylist)
print('mylist:', mylist)  # mylist: [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
print('newlist:', newlist)  # newlist: [0, 1, 2, 5, 6, 7, 9, 15, 16, 18]

# リストの要素をソートする（インプレース／逆順）
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
mylist.sort(reverse=True)
print(mylist)  # [18, 16, 15, 9, 7, 6, 5, 2, 1, 0]

# ソートに使用するキーを決定する関数を指定
mylist = [-1.7, 0.1, 4.5, -1.5, -4.6]
mylist.sort(key=abs)  # 絶対値順でソート
print(mylist)  # [0.1, -1.5, -1.7, 4.5, -4.6]

# ラムダ式を使って、リストのリストをソートする
mylist = [[0, 1, 2], [1, 2, 0], [2, 0, 1]]
mylist.sort(key=lambda x: x[1])  # 1 番目の要素をキーとしてリストをソート
print(mylist)  # [[2, 0, 1], [0, 1, 2], [1, 2, 0]]

# key キーワード引数と reverse キーワード引数を使用
mylist = [-1.7, 0.1, 4.5, -1.5, -4.6]
mylist.sort(key=abs, reverse=True)  # 絶対値を基に降順でソート
```

```
print(mylist)  # [-4.6, 4.5, -1.7, -1.5, 0.1]

# 要素を逆順に並べ替える（インプレース）
mylist = list(range(5))  # [0, 1, 2, 3, 4]
mylist.reverse()
print(mylist)  # [4, 3, 2, 1, 0]

# 要素を逆順に並べ替えた新しいイテレータを作成する
mylist = list(range(5))  # [0, 1, 2, 3, 4]
iterator = reversed(mylist)  # reversed 関数の戻り値はイテレータ
print(iterator)  # <list_reverseiterator object at ...>
newlist = list(iterator)  # イテレータからリストを作成
print('mylist:', mylist)  # mylist: [0, 1, 2, 3, 4]
print('newlist:', newlist)  # newlist: [4, 3, 2, 1, 0]
```

sort メソッドと sorted 関数

リスト（配列）の要素をソートするには、リストが持つ sort メソッドか Python に組み込みの sorted 関数を使う。前者はリストをインプレースに変更する。つまり、元のリストの要素が直接変更される。後者は、リストの要素をソートした結果の新しいリストを作成する。

以下に例を示す。

```
# リストの要素をソートする（インプレース）
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
mylist.sort()
print(mylist)  # [0, 1, 2, 5, 6, 7, 9, 15, 16, 18]

# リストの要素をソートした新しいリストを作成する
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
newlist = sorted(mylist)
print('mylist:', mylist)  # mylist: [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
print('newlist:', newlist)  # newlist: [0, 1, 2, 5, 6, 7, 9, 15, 16, 18]
```

1 つ目の例ではリスト mylist の sort メソッドを呼び出している。このため、リストの要素がインプレースで並べ替えられている。対して、2 つ目の例では sorted 関数にリスト mylist を渡しているので、元のリストでは要素は以前のままで、sorted メソッドの戻り値である newlist では要素がソートされている。

sort メソッドや sorted 関数を使ってソートを行う際には、内部的には < 演算子を使ってそれぞれの要素の大小関係が比較されている。このため、リストに含まれている全ての要素同士をこの演算子で比較できる必要がある。例えば、整数と浮動小数点数の 2 種類の値だけがリストの要素であれば、問題なくソートできる。

```
mylist = [3.0, 0, 1.1, 2]
mylist.sort()  # 整数と浮動小数点数の比較はサポートされている
print(mylist)  # [0, 1.1, 2, 3.0]
```

これに対して、リストに文字列と整数値が含まれているというときには例外となる。

```
mylist = [3, 0, '1', 2]
mylist.sort()  # TypeError：整数と文字列の比較はサポートされていない
```

このようなときには、何らかの形で要素の型を互換性のあるものに統一できるのであれば、以下で紹介する key キーワード引数に型変換するための関数を指定すれば、ソートすることは可能だ（そうした行為に意味があるかどうかは時と場合によるだろう）。

```
mylist = [3, 0, '1', 2]
mylist.sort(key=int)  # 文字列を整数値に。「key=str」も同様
print(mylist)  # [0, '1', 2, 3]
```

sort メソッドと sorted 関数には reverse と key という 2 つのキーワード引数がある。

reverse キーワード引数はソートを昇順／降順に行うかどうかを指定するものだ。これを True に指定すると、ソートの結果は通常とは逆の順序（降順）になる。

以下に例を示す。

```
mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
mylist.sort(reverse=True)
print(mylist)  # [18, 16, 15, 9, 7, 6, 5, 2, 1, 0]

mylist = [18, 0, 16, 6, 15, 7, 9, 1, 2, 5]
newlist = sorted(mylist, reverse=True)
print(newlist)  # [18, 16, 15, 9, 7, 6, 5, 2, 1, 0]
```

先ほども出てきたもう 1 つの key キーワード引数は、要素を比較する際のキーとなる値を決定するのに使う関数を指定する。この関数は 1 つの引数を受け取り、何らかの値を返すものである必要がある。

以下に例を示す。

```
mylist = [-1.7, 0.1, 4.5, -1.5, -4.6]
mylist.sort(key=abs)  # 絶対値順でソート
print(mylist)  # [0.1, -1.5, -1.7, 4.5, -4.6]

mylist = [-1.7, 0.1, 4.5, -1.5, -4.6]
newlist = sorted(mylist, key=abs)
print(newlist)  # [0.1, -1.5, -1.7, 4.5, -4.6]
```

この例では、正負の浮動小数点数値がリストの要素となっている。そして、key キーワード引数に Python に組み込みの abs 関数を指定して、sort メソッドを呼び出している。そのためここでは、リストの各要素を abs 関数に渡した結果、つまり各要素の絶対値を基に要素の並べ替えが行われる。

このように既存の関数（や自作の関数）を指定することもできるが、ラムダ式を使うことも可能だ。例として、リストを要素とするリストで、要素となっているリストをソートすることを考える。key キーワード引数を指定しなければ、リストとリストが < 演算子で比較されるが、簡単なラムダ式を書くだけで、要素となっているリストの何番目かの要素を基にソートを行える。

```
mylist = [['isshiki', 175, 68], ['kawasaki', 172, 80], ['endo', 180, 75]]
mylist.sort()  # キー関数を指定せずにソート
print(mylist)  # [['endo', 180, 75], ['isshiki', 175, 68], ['kawasaki', 172, 80]]

mylist.sort(key=lambda x: x[1])  # 1 番目の要素をキーとしてリストをソート
print(mylist)  # [['kawasaki', 172, 80], ['isshiki', 175, 68], ['endo', 180, 75]]

newlist = sorted(mylist, key=lambda x: x[2])  # 2 番目の要素をキーとしてソート
print(newlist)  # [['isshiki', 175, 68], ['endo', 180, 75], ['kawasaki', 172, 80]]
```

ここでは [名前 , 身長 , 体重] という要素で構成されるリストを要素とするリストがある。キー関数を指定せずにソートを行ったのが一番上の例だ。次の例では、key キーワード引数に「lambda x: x[1]」を指定してソートをしているので、身長の低い順にリストがソートされている。最後の例では key キーワード引数に「lambda x: x[2]」を指定しているので、体重が少ない順にソートが行われている。

最後に、2 つのキーワード引数を使った例も示しておく。説明は不要だろう。

```
mylist = [-1.7, 0.1, 4.5, -1.5, -4.6]
mylist.sort(key=abs, reverse=True)  # 絶対値を基に降順でソート
print(mylist)  # [-4.6, 4.5, -1.7, -1.5, 0.1]
```

なお、sorted 関数にはリストだけではなく、任意の反復可能オブジェクトを与えられる（Python が標準で提供している反復可能オブジェクトで、sort メソッドが定義されているのは list 型のみ）。

```
t = ('foo', 'bar', 'baz')
mylist = sorted(t)  # タプルの要素をソート（戻り値はリストになる）
print(mylist)  # ['bar', 'baz', 'foo']

s = 'python'
mylist = sorted(s)  # 文字列も反復可能オブジェクト
print(mylist)  # ['h', 'n', 'o', 'p', 't', 'y']

t.sort()  # AttributeError：タプルに sort メソッドはない（タプルは変更できない）
s.sort()  # AttributeError：文字列に sort メソッドはない（文字列は変更できない）
```

reverse メソッドと reversed 関数

リストの reverse メソッドは、その要素をインプレースで逆順に並べ替えるものだ。

```
mylist = list(range(5))  # [0, 1, 2, 3, 4]
print(mylist)  # [0, 1, 2, 3, 4]
mylist.reverse()
print(mylist)  # [4, 3, 2, 1, 0]
```

sort メソッドと同様に、reverse メソッドは元のリストの要素を並べ替える。そうではなく、要素を逆順にした新しいリストが必要であれば、Python に組み込みの reversed 関数を使用する。ただし、reversed 関数はリストではなくイテレータを戻り値とする点には注意すること。

```
mylist = list(range(5))  # [0, 1, 2, 3, 4]
list_iter = reversed(mylist)  # reversed 関数の戻り値はイテレータ
print(list_iter)  # <list_reverseiterator object at 0x...>
newlist = list(list_iter)  # リストにするには list 関数を使う
print(newlist)  # [4, 3, 2, 1, 0]
```

この例では、reversed 関数に元のリストを渡して、要素を逆順に列挙するイテレータを取得した後、それを list 関数に渡して要素が逆順に並んだリストを手に入れている。

編集：@IT 編集部
発行：アイティメディア株式会社