# リスト

- リストの立ち位置（全体での分類を考えた場合）
  - イテラブル（for文で順番に取り出せる）
  - シーケンス型（順番が重要：タプル・文字列 もそうである）
  - 大きな4つのコレクションの仲間の一つ（他には、タプル・集合・辞書 あり）
- スライス
  - 使い方
  - 負のインデックス
  - 位置をつかむコツ
- メソッド

```
リスト.append(x)                        # x：要素
リスト.extend(iterable) # iterable：イテラブル
リスト.insert(index, x)
リスト.remove(x)
リスト.pop([index])             # 【重要】 []はリストとは関係なく、省略できると言う意味
リスト.clear()
リスト.index(x[, start[, end]]) # 【重要】：該当箇所で記法を説明します
リスト.count(x)
リスト.sort(key=None, reverse=False)
リスト.reverse()
リスト.copy()
```

- スタック・キュー
  - 実装方法
- リスト内包表記
  - 書き方

## リストの概要

- サイズは自由で、必要に応じて、大きくも小さくも出来る

- 宣言(定義)は、`[]`内に、`,`で区切ってください

```
[1, 3, 5, 7]
```

- 要素(リストのデータ)は、いろいろな型を混ぜることが出来る

```
[1, "one", ["dog", "one one"]]  #（数値、文字列、リスト）
```

- len()組込み関数
  - 要素数を返す

    ```
    print(len([2, 4, 6]))  # 3
    ```

  - 注意：ネストされたリスト内の要素はカウントされない

    ```
    print(len([1, "one", ["dog", "oneone"]]))  # 3
    ```

- indices(indexの複数形)
  - インデックスの使い方：`0スタート`としてその位置の値を返すことが出来る

    ```
    lst = ["one", "two", "three", "four"]
    print(f"lst[0] = {lst[0]}")
    print(f"lst[1] = {lst[1]}")
    print(f"lst[2] = {lst[2]}")
    print(f"lst[3] = {lst[3]}")

    """実行結果：indexが0の時、1番目の値を返す

    lst[0] = one
    lst[1] = two
    lst[2] = three
    lst[3] = four
    """
    ```

  - `-`(マイナス)の指定
    - `-1`を指定すると、一番最後の要素を取得できる (-2なら最後から2つ目、-3なら…)

      ```
      lst = ["one", "two", "three", "four"]

      if lst[3] == lst[-1]:
          print(f"lst[3] == lst[-1] → True：インデックスの-1は最後の値")

      """ lst[3] == lst[-1] → True：インデックスの-1は最後の値 """
      ```

- リストに`+`を使う

  ```
  lst_1 = [1, 2]
  lst_2 = [3, 4]
  ```

```
lst_1_2 = lst_1 + lst_2
print(lst_1_2)  # [1, 2, 3, 4]
```

- ネストされたリストの要素にアクセスする

```
# ネストされた要素にアクセスする

lst = [0, ["a", 1, True], [9, ["Can you catch me?", "Yes, we can!"], 10]]

# Yes, we can! にアクセスする
wecan = lst[2][1][1]
print(wecan)  # Yes, we can!
```

## スライス

- [a:b]
  - インデックス`a`〜インデックス`bの一つ前`
  - `スライス操作`では、`新たなリスト`を返す
    - リストはミュータブル(変更可能)なので、勘違いしやすい所である
    - ただし、copy()メソッドと同じで、「浅いコピー」である

- `スライス`を考える時の`インデックスの考え方`：
  - `要素の間（または前）にあると考えると分かりやすい`
  - スライスとは、部分リスト・サブリスト のように考えることが出来る
  - インデックス(上段)、負のインデックス(中段)、`要素(下段)`

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| -5 | -4 | -3 | -2 | -1 |
| "a" | "b" | "c" | "d" | "e" |

```
lst = ["a", "b", "c", "d", "e"]

print(lst[1:4])  # ['b', 'c', 'd']
print(lst[0:-1])  # ['a', 'b', 'c', 'd']
```

- `スライス`に関する多くの人が誤解する落とし穴
  - 2番目のインデックスが、1番めのインデックスより、前の値になる時、`逆順で返すわけではない`！
    - 上記の例で考えると、`[3:1]`は、`["c", "b"]を返すわけではない`！
      - この場合、空のリストを返します

```
lst = ["a", "b", "c", "d", "e"]
print(lst[1:3])  # ['b', 'c']
print(lst[3:1])  # []
```

- スライス`[start:stop:step]`の1番目、2番目、あるいは両方を省く場合 (stepは増分)
  - 1番目を省く時：一番最初の要素からスタート
  - 2番目を省く時：最後の要素までを含める
  - 両方を省く時は、元のリストと同じになる

```
lst = ["a", "b", "c", "d", "e"]
print(lst[:3])  # ['a', 'b', 'c']
print(lst[3:])  # ['d', 'e']
print(lst[:])  # ['a', 'b', 'c', 'd', 'e']
```

  - ここで、`[:]`は、コピーをしていることは重要ポイントです

```
lst = ["a", "b", "c", "d", "e"]
print(lst[:3])  # ['a', 'b', 'c']
print(lst[3:])  # ['d', 'e']
print(lst[:])  # ['a', 'b', 'c', 'd', 'e']

# copied_lstは、lstのコピーである
copied_lst = lst[:]
copied_lst.append("f")  # 末尾に"f"を追加する（append：後述）
print(copied_lst)  # ['a', 'b', 'c', 'd', 'e', 'f']
# 元の`lst`は、変更されていない！
print(lst)  # ['a', 'b', 'c', 'd', 'e']
```

  - トライ!：リストを2つに分割しよう（奇数個の時は、前半の要素が一つ少なくなる）

```
# リストを、2つに分割する
#   要素が奇数個の時は、前半が1つ少なくなる
lst = [0, 1, 2, 3, 4, 5, 6]
half = int(len(lst) / 2)  # スライスには、float型は使えない
print(lst[:half], lst[half:])  # [0, 1, 2] [3, 4, 5, 6]
```

  - インデックスにスライスを適用して、リストを柔軟に扱っている例

```
# リストをスライスを使って、さらに自由に変更する
learning_lst1 = ["0:環境構築", "1:変数"]
learning_lst2 = ["2:型", "3:条件分岐"]

learning_lst1[2:] = learning_lst2
```

```
print(learning_lst1)
"""
['0:環境構築', '1:変数', '2:型', '3:条件分岐']
"""

learning_lst1[:0] = ["Python"]
print(learning_lst1)
"""
['Python', '0:環境構築', '1:変数', '2:型', '3:条件分岐']
"""

learning_lst1[1:-1] = []
print(learning_lst1)
"""
['Python', '3:条件分岐']
"""

learning_lst1 = []
print(learning_lst1)  # []
```

## リストのメソッド

- 認定テキストにある11個のメソッド

### リスト.append(値)

1. 値が、要素の時：1個値を増やす
2. 値が、リストの時：リスト`自体`を増やす
    - extend()メソッドとの違いに注意

- リストは、`ミュータブル`(値を変更出来る)なので、コードでは、`リストa自体が変更`していきます

```
# append()

a = [1, 2, 3]
b = 4
c = [5, 6, 7]

# 要素の追加
a.append(b)
print(a)  # [1, 2, 3, 4]

# リスト自体を追加する
a.append(c)
print(a)  # [1, 2, 3, 4, [5, 6, 7]]
```

**リスト.extend(イテラブルオブジェクト)**

- イテラブルオブジェクト：for文で一つ一つ取り出せるオブジェクト：リスト・タプルなど
- append()と違って、全要素を末尾に追加する

```
# extend()

a = [1, 2, 3]
b = ("four", "five", "six")

a.extend(b)
print(a)  # [1, 2, 3, 'four', 'five', 'six']

# スライスを使って、リストの末尾に追加する方法
c = [7, 8, 9]
# len(a) は、最後のインデックスの値プラス1 になる
#   なぜなら、要素数は、1から数えるからである
# つまり、len(a)は、現在のリストa の、まだ存在しない次のインデックスとなる
#   現在リストaは、[1, 2, 3, 'four', 'five', 'six']
#   であり、6個の要素があり、len(a) は、6 であり、
#   インデックス は、0～5である
# a[6:] ということは、リストa のまだ存在していない部分である
# 次のコードは、まだ存在していない部分に、リストc を、挿入することである
a[len(a):] = c
print(a)  # [1, 2, 3, 'four', 'five', 'six', 7, 8, 9]
```

**リスト.insert(インデックス, 値)**

- `インデックスにある元の値の、前の位置` に、2つ目の引数の値 を、挿入する

```
# insert()

a = [0, 2]

# 値1 を、値2 の前に入れたい
a.insert(1, 1)
print(a)  # [0, 1, 2]

# 値3 を、末尾に入れたい
#   インデックス3 は無いが、あるとしたら、その前に 値3 を入れたい
a.insert(3, 3)
print(a)  # [0, 1, 2, 3]

# 値4 を、末尾に入れる方法2（この方が汎用的な方法である）
#   len(a)は、現在のリストa の次のインデックス値 である
a.insert(len(a), 4)
print(a)  # [0, 1, 2, 3, 4]
```

### リスト.remove(値)

- 値 と等しい最初の要素を削除する
  - 無い時は、ValueErrorとなる

```
# remove()

lst = ["apple", "orange", "grape"]

# 以下(fruit = None)は不要だが、書いた方がロジックが分かりやすいと思う
fruit = None

try:
    fruit = "orange"
    lst.remove(fruit)  # 成功
    print(f"{fruit}、 これです！")
    print(lst)  # ['apple', 'grape']

    fruit = "banana"
    lst.remove(fruit)  # ここでValueError発生
    print(f"{fruit}、これです")  # この行は実行されない
except ValueError:
    print(f"{fruit}、品切れでした")  # ここが実行される
else:
    print("今日の在庫はOKでした")  # 例外が発生したので実行されない

"""
orange、 これです！
['apple', 'grape']
banana、品切れでした
"""
```

### リスト.pop( [ インデックス ] )

- `[]` ： ドキュメンテーションなどに書かれている記法で、省略可能と言う意味である
  - リストの学習中に、[ ] が出てくると、紛らわしいが、リストとは関係ない話です
- インデックスが指定されていない場合
  - 末尾のアイテムを削除
  - pop()メソッドは削除するだけでなく、削除した値を返す
- インデックスが指定されている場合
  - 指定されている値を削除する
  - そして返す

```
# pop()

lst = ["りんご", "みかん", "なし", "ぶどう"]

print("店長：在庫を調べてください")
print(f"店員：リストにしました：{lst}\n")

print("お客さん：リンゴをください")
print(f"店員：はい、{lst.pop(0)}です")  # インデックス指定
print(f"在庫状況：{lst}\n")

print("お客さん：一番奥にあるのは、ぶどうですか？")
print(f"店員：はい、{lst.pop()}です")  # インデックスを指定しない
print(f"在庫状況：{lst}\n")

"""
店長：在庫を調べてください
店員：リストにしました：['りんご', 'みかん', 'なし', 'ぶどう']

お客さん：リンゴをください
店員：はい、りんごです
在庫状況：['みかん', 'なし', 'ぶどう']

お客さん：一番奥にあるのは、ぶどうですか？
店員：はい、ぶどうです
在庫状況：['みかん', 'なし']
"""
```

**リスト.clear()**

- リストからすべてのアイテムを削除する
- `del リスト[:]` と同じである

```
# clear()

lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
lst.clear()
print(lst)  # []
```

**del文：（メソッドではありません）**

- del リスト[2]
  - インデックスが2 の値を削除する
- del リスト[:]

    - 空リストにする
        - [:] とは、リスト内すべての要素である

- del リスト
    - リスト自体削除する

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

del lst[1]
print(lst)  # [1, 3, 4, 5]

del lst[:]
print(lst)  # []

del lst
# NameError：変数lst は、存在しない
# print(lst)
```

**リスト.index(値[, start[, end]])**

- 引数x[, 引数y[, 引数z]]
    - ドキュメンテーションでよく使われる記法
        - [ ]内は省略できる
        - `ただし 引数y を省略して 引数z だけ書くことは出来ない`
            - つまり、引数x と 引数y だけが使われることはあり得る

- `値 の インデックス を返します`
- ここで、start・end は、[start:end] のようにスライスとして範囲を限定している

**Cluade氏にこの書き方で分かりやすいか尋ねたところ以下のように提案されました**
**その説明も含めておこうと思います**

**引数の省略ルール**

- `[ ]` 内は省略可能（ドキュメンテーション記法）
- 引数は `左から順に指定` する必要がある
    - `index(値)` ✓
    - `index(値, start)` ✓
    - `index(値, start, end)` ✓
    - `index(値, , end)` ✗ （ `startを省略してendだけ指定は不可` ）

```
# index(値[, start[, end]])

lst = ["zero", "one", "two", "three", "four", "five"]

print(lst.index("three"))  # 3

print(lst.index("four", 2, 5))  # 4（インデックス2-4の範囲で検索）

# start, end は、スライスに当てはめて考える
# よって以下はValueErrorとなる
# （インデックス0-4の範囲で検索）
# print(lst.index("five", 0, 5))
```

**リスト.count(値)**

- 引数の値 の数を返す

```
lst = ["apple", "orange", "banana", "kiwi", "kiwi", "strawberry"]

print(lst.count("kiwi"))  # 2
print(lst.count("strawberry"))  # 1
print(lst.count("pear"))  # 0
```

**リスト.sort(key=None, reverse=False)**

- ソート(整頓して並べる)する
- デフォルトで、key=None(特に基準無し)なので、小さい順・アルファベット順が、採用される
  - keyを指定すると、それを基準に並べ替える
- デフォルトで、reverse=False なので、昇順(降順でない)となる
  - reverse=True であると、降順に並べる
- インプレースである
  - つまりコピーを作らず、元のリストを変更する(リストはミュータブル)
- `注意`
  - ビルトイン関数の`sorted()関数`は、元のリストは変更せず、変更したリストを返す

```
# sort(key=None, reverse=False)

lst = [1, 5, 3, 8]
```

```
lst.sort()
print(lst)  # [1, 3, 5, 8]

lst.sort(reverse=True)
print(lst)  # [8, 5, 3, 1]

# 次に使うkeyの内容を調べる
lst2 = []
for i in lst:
    lst2.append(i % 6)
print(lst2)  # [2, 5, 3, 1]

# key は ラムダ式の結果を基準にする
#       上のコードは、以下のコードが複雑なので、目安を付けるために書いたわけである
# `[8, 5, 3, 1]` は、lst2で示された値を昇順の基準とするわけである
# ラムダ式は、仮引数x に対して 戻り値・x%6 を返すからである
# すると[8(2), 5(5), 3(3), 1(1)] の、()内の昇順になるわけである
# [8,5,3,1] を [2番目(2), 4番目(5), 3番目(3), 1番目(1)]とするわけである
# これはかなりややこしいので、AIに聞いた方が良いかもしれない(私も聞いた)
# 試験で大事なことは、keyが指定されていると、それを基準に並べる：ということである
lst.sort(key=lambda x: x % 6)
print(lst)  # [1, 8, 3, 5]
```

**リスト.reverse()**

- リスト内の要素を、逆に並び替える
- リスト自体を変更する
    - つまり、変更されたリストを返すわけではない

```
# reverse

lst = [4, 8, 1, 3]
lst.reverse()
print(lst)  # [3, 1, 8, 4]
```

**リスト.copy()**

- 浅いコピーを返す
    ミュータブル と イミュータブル では大違いで、ここは【ミュータブル】である ← ミュータブル：変更可能なオブジェクトである
    - ~~イ~~ミュータブルな部分で問題が起きそうなときは、copy.deepcopy()を検討しよう
    - 問題が起きないと分かっていれば(中・上級者)、copy()の方がパフォーマンスがよい
- スライスを使った、リスト[:] と、同じことをする

```python
# copy()

lst = [1, 2, 3]

lst_copy1 = lst.copy()
print(lst_copy1)  # [1, 2, 3]

# copy()メソッドと、リスト[:] は、同じことをする
lst_copy2 = lst[:]
print(lst_copy2)  # [1, 2, 3]
```

## list_test.py（リストのメソッドを覚えるためのステップ1）

- Claude氏に依頼しました
  - AIにお願いする時のコツ：
    - なるべく具体的に説明する
    - 目的なども含めた方が理想に近くなる
    - プログラミング言語の知識があればあるほど、詳細なことをお願いできる
    - 必ず自分でもチェックしてPyCharmなどの弱い警告でも相談しましょう
  - コピペして貼り付ければ使えます
    - コンソールアプリなので、コマンドプロンプトなどで実行しましょう
    - PyCharmを使えれば、実行ボタンを押すだけです
  - 大ざっぱな説明ですが、これをきちんと覚えておくと、詳細を覚えるのも楽になるはずです

```python
"""
Python3基礎認定試験対策 - メソッド問題集
メソッド名から正しい説明の番号を選択するクイズアプリ
"""

import random


class MethodQuiz:
    def __init__(self):
        self.method_data = [
            ["append", "リストの末尾にアイテムを一つ追加"],
            ["extend", "末尾にイテラブルの全アイテムを追加"],
            ["insert", "指定された位置にアイテムを挿入"],
            ["remove", "引き数に等しい最初のアイテムを削除"],
            ["pop", "デフォルトで最後のアイテムを削除・リターンする"],
            ["clear", "空リストにする"],
            ["index", "引き数に等しい最初のインデックスを返す"],
            ["count", "リスト内の引数に等しい値の数を返す"],
            ["sort", "デフォルトで昇順に並べる"],
            ["reverse", "要素の順序を逆順にする"],
            ["copy", "浅いコピーで、使い方で注意もある！"]
        ]
```

```
        self.score = 0
        self.total_questions = 0

    def create_question(self):
        """問題を作成する（ランダム選択）"""
        # ランダムに問題を選択
        correct_method = random.choice(self.method_data)
        return self.create_question_for_method(correct_method)

    @staticmethod
    def display_question(method_name, choices):
        """問題を表示する"""
        print("\n" + "=" * 50)
        print(f"【問題】 メソッド: {method_name}")
        print("=" * 50)
        print("以下の説明の中から、正しいものの番号を入力してください：")
        print()

        for i, choice in enumerate(choices, 1):
            print(f"{i}. {choice}")
        print()

    @staticmethod
    def get_user_answer(max_choice):
        """ユーザーの回答を取得する"""
        while True:
            try:
                answer = input(
                    f"答えを入力してください (1-{max_choice}): ").strip()
                if answer.lower() in ['q', 'quit', '終了']:
                    return None

                answer_num = int(answer)
                if 1 <= answer_num <= max_choice:
                    return answer_num
                else:
                    print(f"1から{max_choice}の間で入力してください。")
            except ValueError:
                print(
                    "数字を入力してください。\
                    終了する場合は 'q' を入力してください。")

    def display_result(self, user_answer, correct_answer, method_name,
                       choices):
        """結果を表示する"""
        if user_answer == correct_answer:
            print("✅ 正解です！")
            self.score += 1
        else:
            print("❌ 不正解です。")
            print(f"正解は {correct_answer}. "
                  f"{choices[correct_answer - 1]} でした。")

        print(f"メソッド '{method_name}' の正しい説明: "
              f"{choices[correct_answer - 1]}")
```

```
    def display_final_score(self):
        """最終スコアを表示する"""
        print("\n" + "=" * 50)
        print("【テスト結果】")
        print("=" * 50)
        print(f"正解数: {self.score}/{self.total_questions}")

        if self.total_questions > 0:
            percentage = (self.score / self.total_questions) * 100
            print(f"正答率: {percentage:.1f}%")

            if percentage >= 90:
                print("🎉 素晴らしい！完璧に近い理解です！")
            elif percentage >= 70:
                print("👏 良くできました！もう少しで完璧です！")
            elif percentage >= 50:
                print("📚 もう少し復習が必要ですね。頑張りましょう！")
            else:
                print("💪 基礎からしっかり復習しましょう！")

    def run_all_questions(self):
        """全ての問題を実行する（11問モード）"""
        print("\n📚 全ての問題にチャレンジします！(11問)")
        print("=" * 50)

        # 全てのメソッドをランダムな順番で出題
        all_methods = self.method_data.copy()
        random.shuffle(all_methods)

        try:
            for i, method_info in enumerate(all_methods, 1):
                print(f"\n【問題 {i}/11】")
                question_data = self.create_question_for_method(
                    method_info)
                method_name, choices, correct_answer = question_data

                # 問題表示
                self.display_question(method_name, choices)

                # 回答取得
                user_answer = self.get_user_answer(len(choices))

                if user_answer is None:  # 終了
                    print("テストを中断しました。")
                    break

                self.total_questions += 1

                # 結果表示
                self.display_result(user_answer, correct_answer,
                                    method_name, choices)

                # 最後の問題でなければ次へ進む確認
                if i < len(all_methods):
                    input("\n次の問題に進みます... (Enterキー)")
```

```
        except KeyboardInterrupt:
            print("\n\nテストを中断しました。")

    def create_question_for_method(self, method_info):
        """指定されたメソッドの問題を作成する"""
        correct_answer = method_info

        # 選択肢を作成（正解 + ランダムな3つの不正解）
        choices = [correct_answer[1]]  # 正解の説明
        other_descriptions = [item[1] for item in self.method_data
                              if item != correct_answer]

        # ランダムに3つの不正解を選択
        wrong_choices = random.sample(other_descriptions,
                                      min(3, len(other_descriptions)))
        choices.extend(wrong_choices)

        # 選択肢をシャッフル
        random.shuffle(choices)

        # 正解の位置を記録
        correct_index = choices.index(correct_answer[1]) + 1

        return correct_answer[0], choices, correct_index

    def run_random_quiz(self):
        """ランダム問題モード"""
        print("\n🎲 ランダム問題モード")
        print("=" * 50)
        print("終了する場合は 'q' を入力してください。")

        try:
            while True:
                # 問題作成
                question_data = self.create_question()
                method_name, choices, correct_answer = question_data

                # 問題表示
                self.display_question(method_name, choices)

                # 回答取得
                user_answer = self.get_user_answer(len(choices))

                if user_answer is None:  # 終了
                    break

                self.total_questions += 1

                # 結果表示
                self.display_result(user_answer, correct_answer,
                                    method_name, choices)

                # 続行確認
                print("\n続けますか？ (Enter: 続行, q: 終了)")
                continue_input = input().strip().lower()
                if continue_input in ['q', 'quit', '終了']:
```

```
                    break

            except KeyboardInterrupt:
                print("\n\nテストを中断しました。")

    def run_quiz(self):
        """クイズを実行する"""
        print("🐍 Python3基礎認定試験対策 - リストメソッド問題集")
        print("=" * 60)
        print("各メソッドの正しい説明を選択してください。")

        # モード選択
        print("\n📋 学習モードを選択してください：")
        print("y: 全ての問題をやる（11問すべて）")
        print("n: ランダム問題モード（好きなだけ）")

        while True:
            choice = input("\n全ての問題をやりますか（11問）？ (y/n): ")
            mode_choice = choice.strip().lower()
            if mode_choice in ['y', 'yes', 'はい']:
                self.run_all_questions()
                break
            elif mode_choice in ['n', 'no', 'いいえ']:
                self.run_random_quiz()
                break
            else:
                print("'y' または 'n' を入力してください。")

        # 最終結果表示
        if self.total_questions > 0:
            self.display_final_score()

        print("\nお疲れ様でした！Python3基礎認定試験頑張ってください！🎯")


def main():
    """メイン関数"""
    quiz = MethodQuiz()
    quiz.run_quiz()


if __name__ == "__main__":
    main()
```

## リストをスタックとして使う

- スタックとは？
  - 「最後に入れたものを最初に出す」という原則でデータの追加と削除が行われること
  - Geminiがこんな例を挙げてくれました。
    - ウェブブラウザの「戻る」ボタンの履歴
      - その前に見たページに一つずつ戻る機能
    - テキストエディタの「元に戻す（Undo）」機能

        - テキストを編集する時、少し前まで戻りたい時に使える

```
import time

# スタック

stack_lst = [4, 9, 1, 3]
print(f"初期状態： \n{stack_lst}\n")

# 末尾に追加
# timeモジュールをインポートすると、1秒ずつスリープできる
stack_lst.append(15)
time.sleep(1)
print(stack_lst)

stack_lst.append(25)
time.sleep(1)
print(stack_lst)

# 末尾から取り出し、空になった時起きるIndexErrorを利用して、breakさせる
try:
    while True:
        stack_lst.pop()
        # timeモジュールをインポートすると、1秒ずつスリープできる
        time.sleep(1)
        print(stack_lst)
except IndexError:
    print("空になりました！")

"""
初期状態：
[4, 9, 1, 3]

[4, 9, 1, 3, 15]
[4, 9, 1, 3, 15, 25]
[4, 9, 1, 3, 15]
[4, 9, 1, 3]
[4, 9, 1]
[4, 9]
[4]
[]
空になりました！
"""
```

## リストをキューとして使う（正確にはリストに似ているのものを使う）

- お米を買うために並んだ順で購入できるイメージ
- from collections import deque
    - dequeを使えるようにする
        - リストの `insert` 、 `pop` は、低速だそうでです
- import time

- 一つ前のコードで説明しました

```
from collections import deque
import time

# q：キューとして使うオブジェクトの参照
q = deque([])

print("お客さん：ラーメンお願い")
time.sleep(0.5)
print("お客さん：ギョーザください")
time.sleep(0.5)
print("お客さん：カレーください")
q.append("ラーメン")  # 店員さんの頭の中
q.append("ギョーザ")  # 店員さんの頭の中
q.append("カレー")  # 店員さんの頭の中
print(f"注文：{q}")

time.sleep(1.5)
print(f"注文を伝える：{q.popleft()}、{q.popleft()}、{q.popleft()}です")
print(f"注文：{q}")

print()

time.sleep(2.5)
print("お客さん：チャーハンです")
q.append("チャーハン")  # 店員さんの頭の中
print(f"注文：{q}")

time.sleep(1.5)
print(f"注文を伝える：{q.popleft()}です")
print(f"注文：{q}")

"""
お客さん：ラーメンお願い
お客さん：ギョーザください
お客さん：カレーください
注文：deque(['ラーメン', 'ギョーザ', 'カレー'])
注文を伝える：ラーメン、ギョーザ、カレーです
注文：deque([])

お客さん：チャーハンです
注文：deque(['チャーハン'])
注文を伝える：チャーハンです
注文：deque([])
"""
```

## 内包表記

- 慣れると簡単に書けるだけではなく、以下のような利点があります
  - 名前空間の汚染を防ぐ
    - 不要な変数が残ることで、意図しない変数の再利用やバグの原因になる可能性があります

- 意図の明確化
  - 「ここでは単純にリストを生成したいだけ」という意図が明確になります
- スコープの制御
  - ループ変数のスコープがリスト内包表記内に限定されるため、より安全です

```
lst = []
for i in range(10):
    lst.append(i + 10)
print(lst)  # [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
# PyCharmは言っています：変数i はどこでも使われていません！
print(i)  # 9

lst2 = [j + 10 for j in range(10)]
print(lst2)  # [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
# 変数jは残っていない（そもそも必要ない場合が多いと思う）
# print(j) # NameError：`j`という変数はありません
```

## 色々な内包表記

- 超基本内包表記イメージ
  - [変数+1　⇐　for 変数 in [0, 1, 2] ]
    - ↑：[1, 2, 3]

```
# 内包表記の基本形(リスト)

lst = [x + 1 for x in [0, 1, 2]]

print(lst)  # [1, 2, 3]
```

- if文と組み合わせるイメージ
  - [変数　⇐②　for 変数 in [-2, -1, 0, 1, 2]　①⇒　if 変数 > 0] ]
    - ↑：[1, 2]

```
# 内包表記にif文を使う

lst = [y for y in [-2, -1, 0, 1, 2] if y > 0]
print(lst)  # [1, 2]
```

- for文がネストされている内包表記
  - やや難しいが、タプルが要素の内包表記にしました

- イメージ：[(a, b)　⇐②　for a in [🍎, 🍌]　①⇒　for b in [🍇, 🥝] ]
  - タプルの場合（ ）を使うのは必須である！
  - ↑：[('🍎', '🍇'), ('🍎', '🥝'), ('🍌', '🍇'), ('🍌', '🥝')]

```
# タプル＋for文ネスト の内包表記

lst = [(a, b) for a in ["🍎", "🍌"] for b in ["🍇", "🥝"]]
print(lst)
"""[('🍎', '🍇'), ('🍎', '🥝'), ('🍌', '🍇'), ('🍌', '🥝')]"""


# 普通のfor文で書いた場合
lst2 = []
for x in ["Hello", "Hi"]:
    for y in ["Bob", "Lucy"]:
        lst2.append((x, y))  # タプルを append しているわけである

print(lst2)
"""
[('Hello', 'Bob'), ('Hello', 'Lucy'), ('Hi', 'Bob'), ('Hi', 'Lucy')]
"""
```

**これ以上難しい内包表記**

- 公式認定テキストには、これ以上に難しい内包表記が載っていますが、
  そんなに難しいのが出たら私は10分考えても必ず正解できる自信はありません。
- つまり時間的に、そこまで難しいのは問題にできないのではないか？？

END