06 Python基礎

break / continue

- C / Java で使用されるものと同一
- ただし、for - else , while - else のようなelse節が存在する場合は要注意

- それぞれの例の動作をよく読んでください。
  【for_else_break.py】

```
print("start")
j=0
for i in range(10):
    print(f"{i=}, {j=}")
    if i > 5:
        print("break-1")
        break
    else:
        print("else-1")

    for j in range(10):
        print(f"{i=}, {j=}")
        if j > 5:
            print("break-2")
            break
        else:
            print("else-2")
    else:
        print("else-3")
else:
    print("else-4")
print("done")
```

【for_else_continue.py】

```
print("start")
j=0
for i in range(10):
    print(f"{i=}, {j=}")
    if i > 5:
        print("continue-1")
        continue
    else:
        print("else-1")

    for j in range(10):
```

```
            print(f"{i=}, {j=}")
            if j > 5:
                print("continue-2")
                continue
            else:
                print("else-2")
        else:
            print("else-3")
    else:
        print("else-4")
print("done")
```

練習1

- 以下のプログラムを作成してください。

1. whileとcontinueを使用して、1から10までの数字を出力するが、3と7はスキップする。(ren1_1.py)
2. forとbreakを使用して、入力された文字列が回文（前から読んでも後ろから読んでも同じ）かどうかを判定する。(ren1_2.py)
3. whileループとcontinue文を使用して、1から100までの数字のうち、3で割り切れる数字だけを出力する(ren1_3.py)
    - continueを使用しない方法でも作成し、比較してください。
4. forループとbreak文を使用して、与えられたリストの要素が特定の条件を満たす場合にその要素とインデックスを出力し、最初に条件を満たす要素が見つかったらループを終了する。
    - 条件：リストの先頭と同じ値が存在している場合
    - 条件に合わない場合は、その旨を出力する。

    【実行例】

```
$ python ren1_4.py
元のデータ： [3, 7, 1, 5, 7, 3, 9, 1]
最初の要素(3)と同じ値が見つかりました
値: 3, 位置: 5番目
$ python ren1_4.py
元のデータ： [3, 7, 1, 5, 7, 9, 1]
最初の要素と同じ値は見つかりませんでした
```

bool型とbool演算

- True / False　・・・　必ず大文字で始めること
- and / or / not を理解する
- 無限ループは以下のようになる

```
while True:
    if 条件:
        break
    ループ内処理
```

bool型とbool演算の注意点

1. **True/Falseの大文字使用の重要性：**

```
# 正しい使用例
flag = True
if flag:
    print("条件が真です")

# 誤った使用例
true = 1  # これはTrueではなく、単なる変数名
if true:  # これは1を評価しているので、意図した動作にならない
    print("これは意図した動作ではありません")
```

2. **bool値の比較：**

```
# 正しい比較
if flag == True:  # これは動作するが...
if flag:          # この方がPythonらしい書き方

# 誤った比較
if flag == true:  # これはエラーになる
```

3. **bool値の代入：**

```
# 正しい代入
result = True
success = False

# 誤った代入
result = true    # NameError: name 'true' is not defined
success = false  # NameError: name 'false' is not defined
```

4. **bool値の型チェック：**

```
# 正しい型チェック
print(type(True))    # <class 'bool'>
print(type(False))   # <class 'bool'>

# 誤った型チェック
print(type(true))    # NameError
```

5. **bool値の演算：**

```
# 正しいbool演算
a = True
b = False
print(a and b)   # False
print(a or b)    # True
print(not a)     # False

# 誤ったbool演算
print(true and false)  # NameError
```

- Pythonでは `True` と `False` は予約語であり、小文字で書くとエラーになる
- bool値は数値の1や0とは異なる型である
- bool値の比較や演算では、必ず大文字で始める必要がある
- 変数名として `true` や `false` を使用すると、予期せぬ動作やエラーの原因になる
- Pythonのbool型は他の言語（C言語など）と異なり、厳密に `True` と `False` の2値のみを持つ
- bool値は整数型のサブクラスであるが、明示的に `True` / `False` を使用することが推奨される
- 条件式では、 `if flag == True:` よりも `if flag:` の方がPythonらしい書き方である

python特有の演算子

演算子	説明	例
is	2つのオブジェクトが同一のオブジェクトかどうかを調べる	True is True→True
is not	2つのオブジェクトが同一のオブジェクトでないかどうかを調べる	True is not False→True
in	左側の被演算子が右側の被演算子に含まれているかどうかを調べる	'yt' in 'Python'→True 'hoge' in ['foo', 'bar', 'baz']→False
not in	左側の被演算子が右側の被演算子に含まれていないかどうかを調べる	'py' not in 'Python'→True 'foo' not in ['foo', 'bar', 'baz']→False

> https://www.atmarkit.co.jp/ait/articles/1907/23/news010.html

演算子の優先順位

> https://docs.python.org/ja/3.9/reference/expressions.html#operator-precedence

The following table summarizes the operator precedence in Python, from highest precedence (most binding) to lowest precedence (least binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left).

比較 節で述べられているように、比較、所属、同一性のテストは全てが同じ優先順位を持っていて、左から右に連鎖するという特徴を持っていることに注意してください。

演算子	説明
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	結合式または括弧式、リスト表示、辞書表示、集合表示
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	添字指定、スライス操作、呼び出し、属性参照
`await x`	Await 式
`**`	べき乗 5
`+x`, `-x`, `~x`	正数、負数、ビット単位 NOT
`*`, `@`, `/`, `//`, `%`	乗算、行列乗算、除算、切り捨て除算、剰余 6
`+`, `-`	加算および減算
`<<`, `>>`	シフト演算
`&`	ビット単位 AND
`^`	ビット単位 XOR
`|`	ビット単位 OR
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	所属や同一性のテストを含む比較
`not x`	ブール演算 NOT
`and`	ブール演算 AND
`or`	ブール演算 OR
`if` -- `else`	条件式
`lambda`	ラムダ式
`:=`	代入式

比較

> https://docs.python.org/ja/3.9/reference/expressions.html#comparisons

C 言語と違って、Python における比較演算子は同じ優先順位をもっており、全ての算術演算子、シフト演算子、ビット単位演算子よりも低くなっています。また `a < b < c` が数学で伝統的に用いられているのと同じ解釈になる点も C 言語と違います:

```
comparison    ::=  or_expr (comp_operator or_expr)*
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "!="
                   | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: `True` or `False`. Custom *rich comparison methods* may return non-boolean values. In this case Python will call `bool()` on such value in boolean contexts.

比較はいくらでも連鎖することができます。例えば `x < y <= z` は `x < y and y <= z` と等価になります。ただしこの場合、前者では `y` はただ一度だけ評価される点が異なります (どちらの場合でも、 `x < y` が偽になると `z` の値はまったく評価されません)。

形式的には、 *a*, *b*, *c*, ..., *y*, *z* が式で *op1*, *op2*, ..., *opN* が比較演算子である場合、 `a op1 b op2 c ... y opN z` は `a op1 b and b op2 c and ... y opN z` と等価になります。ただし、前者では各式は多くても一度しか評価されません。

`a op1 b op2 c` と書いた場合、 *a* から *c* までの範囲にあるかどうかのテストを指すのではないことに注意してください。例えば `x < y > z` は (きれいな書き方ではありませんが) 完全に正しい文法です。

演算子の優先順位の例と注意点

1. **基本的な優先順位の例**

```python
# 乗算は加算より優先順位が高い
result = 2 + 3 * 4  # 3*4が先に計算され、2+12=14になる
print(result)  # 14

# 括弧で優先順位を変更
result = (2 + 3) * 4  # 2+3が先に計算され、5*4=20になる
print(result)  # 20
```

2. **比較演算子の連鎖**

```python
# 正しい使用例
x = 5
if 1 < x < 10:  # 1 < x and x < 10 と同じ意味
    print("xは1より大きく10より小さい")

# 誤った使用例
if 1 < x > 10:  # 意図が不明確
    print("これは意図した動作ではない可能性がある")
```

3. **論理演算子の優先順位**

```
# andはorより優先順位が高い
result = True or False and False  # False and Falseが先に評価され、True or False=True
print(result)  # True

# 括弧で優先順位を明確化
result = (True or False) and False  # True and False=False
print(result)  # False
```

4. **よくある間違い**

```
# 誤った例：代入演算子の優先順位
x = 5
y = 10
# if x = y:  # これはエラーになる
if x == y:  # 正しい比較演算子
    print("xとyは等しい")

# 誤った例：ビット演算子の優先順位
result = 1 << 2 + 3  # 2+3が先に計算され、1<<5=32
print(result)  # 32
# 意図した動作が1<<2 + 3=4+3=7の場合
result = (1 << 2) + 3  # 括弧で優先順位を明確化
print(result)  # 7
```

5. **複合代入演算子の注意点**

```
# 複合代入演算子の優先順位
x = 5
x *= 2 + 3  # x = x * (2 + 3) と同じ
print(x)  # 25

# 意図した動作が (x * 2) + 3 の場合
x = 5
x = x * 2 + 3
print(x)  # 13
```

6. **三項演算子の優先順位：**

```
# 三項演算子の優先順位
x = 5
y = 10
result = x if x > y else y if y > 0 else 0  # 右から評価される
print(result)  # 10

# 括弧で優先順位を明確化
result = (x if x > y else y) if y > 0 else 0
print(result)  # 10
```

- 演算子の優先順位によって、式の評価順序が決まる
- 括弧を使用することで、優先順位を明示的に指定できる

- 複雑な式では、優先順位を考慮して適切に括弧を使用する必要がある
- よくある間違いを避けるために、優先順位を理解することが重要
- 優先順位が不明確な場合は、括弧を使用して明示的に指定する
- 複雑な式は、複数の行に分けて記述することで可読性を向上させる
- 演算子の優先順位は、言語によって異なる場合がある
- デバッグ時には、式の評価順序を確認することが重要

> **※ 教科書では、次の項目は関数定義（P.26）だが、データ構造を終えた後、戻って関数を解説する。**

リスト/タプルについて

- list型
- tuple型

共通の特徴

- 格納される各要素の型は、バラバラでも良い
- インデックスとスライス操作
- イテラブルなオブジェクト

特徴	リスト (List)	タプル (Tuple)
可変性	要素の追加、削除、変更が可能（ミュータブル）	要素の追加、削除、変更が不可能（イミュータブル）
宣言	`[]` または `list()` を使用	`()` または `tuple()` を使用
代入	`my_list = [1, 2, 3]`	`my_tuple = (1, 2, 3)`
要素へのアクセス	インデックスやスライスを使用	インデックスやスライスを使用
メモリ使用量	より多くのメモリを使用	少ないメモリを使用
パフォーマンス	要素数が多い場合に効率が低下する可能性がある	要素数が多い場合でも高速
使いどころ	要素の追加や変更が必要な場合	一度設定した要素を変更する必要がない場合

タプルのイミュータブル性の利点

1. **データの保護：**

```
# 座標を表すタプル
point = (10, 20)

# 誤って変更しようとしてもエラーになる
point[0] = 5  # TypeError: 'tuple' object does not support item assignment

# 新しい座標が必要な場合は、新しいタプルを作成
new_point = (5, point[1])
```

2. **辞書のキーとして使用：**

```
# リストは変更可能なので辞書のキーにできない
invalid_key = [1, 2, 3]
# d = {invalid_key: "value"}  # TypeError: unhashable type: 'list'

# タプルは変更不可なので辞書のキーにできる
valid_key = (1, 2, 3)
d = {valid_key: "value"}  # 正常に動作
```

3. **関数の戻り値として使用：**

```
def get_coordinates():
    x = 10
    y = 20
    return (x, y)  # 戻り値が変更されないことが保証される

coordinates = get_coordinates()
# coordinates[0] = 5  # エラーになるので、戻り値が保護される
```

4. **定数の定義：**

```
# 曜日を表す定数
WEEKDAYS = ('月', '火', '水', '木', '金', '土', '日')

# 誤って変更しようとしてもエラーになる
# WEEKDAYS[0] = 'Mon'  # TypeError

# リストの場合、誤って変更される可能性がある
weekdays_list = ['月', '火', '水', '木', '金', '土', '日']
weekdays_list[0] = 'Mon'  # 変更できてしまう
```

5. **パフォーマンスの向上：**

```
# タプルの方がメモリ使用量が少ない
import sys

list_size = sys.getsizeof([1, 2, 3, 4, 5])
tuple_size = sys.getsizeof((1, 2, 3, 4, 5))

print(f"リストのサイズ: {list_size} bytes")
print(f"タプルのサイズ: {tuple_size} bytes")
```

6. **マルチスレッド環境での安全性**：

```
# 複数のスレッドで共有するデータ
shared_data = (1, 2, 3)  # タプルは変更不可なので安全

# リストの場合、他のスレッドで変更される可能性がある
unsafe_data = [1, 2, 3]  # 変更可能なので注意が必要
```

- タプルのイミュータブル性により、データの整合性が保証される
- 辞書のキーとして使用できる
- 関数の戻り値として安全に使用できる
- 定数の定義に適している
- メモリ効率が良い
- マルチスレッド環境での安全性が高い
- タプルは変更不可であるため、デバッグが容易になる
- タプルはハッシュ可能（hashable）であるため、集合（set）の要素としても使用できる
- タプルはリストよりも高速に処理できる場合がある
- タプルはデータの構造を明確に表現できる

以下のプログラムを実行した結果を予想してください。

【list_1.py】

```
a1 = ['a', 'b', 'c']
a2 = ['1', '2', '3']
a3 = [1, 3, 5]

a4 = a1 + a2    # a4→?
a5 = a4 + a3    # a5→?

print('-'*30)
print(a4)   # 何が出力される？
a4.sort()
print(a4)   # 何が出力される？

print('-'*40)
```

```
print(a5)    # 何が出力される？
a5.sort()
print(a5)    # 何が出力される？
```

リスト内のデータが、整数型・文字列型が混在している場合は要注意。

- どうすれば、このようなデータを並び替えることができるだろうか？

作業

- list型のオブジェクトを操作する以下のメソッドの使い方を確認し、1〜2行程度のサンプルを記述してください。
- 注意すべき事項があれば、記載してください。

メソッド名	使い方・サンプル
append( )	
extend( )	
insert( )	
remove( )	
pop( )	
clear( )	
index( )	
count( )	
sort( )	
reverse( )	
copy( )	

内包表記

- 教科書 P.46-48までをよく読んで理解してください。
    - 以下のサイトの内包表記についてよく読んで理解してください。
      （集合・辞書に関しては別途説明後に実施。今回は省略）

      > https://note.nkmk.me/python-list-comprehension/

練習2

- 以下のプログラムを作成してください。

    1. 与えられたリストの各要素を2倍にして新しいリストを作成し出力する。(ren2_1.py)
    2. 与えられたリストから偶数のみを取り出して新しいリストを作成し出力する。(ren2_2.py)
    3. 1から10までの数値のリストがある。このリストの各要素に対して、その要素が偶数なら"Even"、奇数なら"Odd"という文字列に変換し出力する。(ren2_3.py)
    4. 与えられた文字列の各文字を大文字に変換して新しい文字列を作成し出力する。(ren2_4.py)

練習3

- 以下のプログラムを作成してください。

    1. 与えられた2次元リストの各要素を2倍にした2次元リストを作成する。(ren3_1.py)

```
original_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

doubled_matrix =
print(doubled_matrix)
```

    2. 与えられた2次元リストの各要素を大文字に変換した2次元リストを作成する。(ren3_2.py)

```
original_matrix = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]

uppercase_matrix =
print(uppercase_matrix)
```

    3. 与えられた2次元リストの各要素を絶対値に変換した2次元リストを作成する。(ren3_3.py)

```
original_matrix = [[-1, 2, -3], [4, -5, 6], [-7, 8, -9]]

absolute_matrix =
print(absolute_matrix)
```

    4. 与えられた2次元リストの各要素の平均値を計算したリストを作成する。(ren3_4.py)

```
original_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

average_list =
print(average_list)
```

練習4

- 以下のプログラムを作成してください。

1. リストの中のデータを調べる。
   【実行結果】

```
果物をカタカナで入力してください：リンゴ
リンゴは、知っています！
果物をカタカナで入力してください：イチゴ
イチゴは、知りませんでした。覚えておきます。
果物をカタカナで入力してください：イチゴ
イチゴは、知っています！
果物をカタカナで入力してください：パイナップル
パイナップルは、知りませんでした。覚えておきます。
果物をカタカナで入力してください：
知っている果物
['バナナ', 'リンゴ', 'オレンジ', 'イチゴ', 'パイナップル']
```

   【ren4_1.py】

```
fruits = ['バナナ','リンゴ','オレンジ']

while True:




print('知っている果物')
print(fruits)
```

2. リスト data の中に含まれる、不要なデータを全て削除する。
   【実行結果】

```
$ python ren4_2.py
[10, 1, 30, 1, 20, 5, 60]
```

   【ren4_2.py】

```
data = [10, 1, 0, 30, 1, 20, 0, 5, 0, 60]

# この中から、目的の要素を全て削除する。
# [10, 1, 30, 1, 20, 5, 60] ← ほしい結果
ng_data = 0 # 不要なデータ




print(data)
```

練習5

- (タプルの練習) 以下のプログラムを作成してください。

1. トランプのカードを52枚を作成する。
   【実行結果】

```
$ python ren5_1.py
----------
[('S', 1), ('S', 2), ('S', 3), ('S', 4), ('S', 5), ('S', 6), ('S', 7), ('S', 8),
('S', 9), ('S', 10), ('S', 11), ('S', 12), ('S', 13), ('H', 1), ('H', 2), ('H',
3), ('H', 4), ('H', 5), ('H', 6), ('H', 7), ('H', 8), ('H', 9), ('H', 10), ('H',
11), ('H', 12), ('H', 13), ('C', 1), ('C', 2), ('C', 3), ('C', 4), ('C', 5), ('C',
6), ('C', 7), ('C', 8), ('C', 9), ('C', 10), ('C', 11), ('C', 12), ('C', 13),
('D', 1), ('D', 2), ('D', 3), ('D', 4), ('D', 5), ('D', 6), ('D', 7), ('D', 8),
('D', 9), ('D', 10), ('D', 11), ('D', 12), ('D', 13)]
----------
```

   - この中から、乱数を用いて、1枚選択する。

     【実行結果】

```
$ python ren5_1.py
----------
[('S', 1), ('S', 2), ('S', 3), ('S', 4), ('S', 5), ('S', 6), ('S', 7), ('S',
8), ('S', 9), ('S', 10), ('S', 11), ('S', 12), ('S', 13), ('H', 1), ('H', 2),
('H', 3), ('H', 4), ('H', 5), ('H', 6), ('H', 7), ('H', 8), ('H', 9), ('H',
10), ('H', 11), ('H', 12), ('H', 13), ('C', 1), ('C', 2), ('C', 3), ('C', 4),
('C', 5), ('C', 6), ('C', 7), ('C', 8), ('C', 9), ('C', 10), ('C', 11), ('C',
12), ('C', 13), ('D', 1), ('D', 2), ('D', 3), ('D', 4), ('D', 5), ('D', 6),
('D', 7), ('D', 8), ('D', 9), ('D', 10), ('D', 11), ('D', 12), ('D', 13)]
----------
選んだカードは('S', 5)です
```

     【ren5_1.py】

```
# 初期化
marks = ('S','H','C','D') # 4種類のマーク
cards = []                    # デッキ用リスト

for
  for




print('-'*10)
print(cards)
print('-'*10)

# 1枚選択
import random         # 乱数用モジュール
r = random.randrange(52)   # 0〜51の乱数生成
print(f'選んだカードは{cards[r]}です')
```

- モジュールrandomについて調べてください。
    - randam.randrange( ) も調べてください。

- 52枚のカードを混ぜて(Shuffle)ください。（ren5_2.py）
  【実行結果】

```
----------
[('S', 1), ('S', 2), ('S', 3), ('S', 4), ('S', 5), ('S', 6), ('S', 7), ('S',
8), ('S', 9), ('S', 10), ('S', 11), ('S', 12), ('S', 13), ('H', 1), ('H', 2),
('H', 3), ('H', 4), ('H', 5), ('H', 6), ('H', 7), ('H', 8), ('H', 9), ('H',
10), ('H', 11), ('H', 12), ('H', 13), ('C', 1), ('C', 2), ('C', 3), ('C', 4),
('C', 5), ('C', 6), ('C', 7), ('C', 8), ('C', 9), ('C', 10), ('C', 11), ('C',
12), ('C', 13), ('D', 1), ('D', 2), ('D', 3), ('D', 4), ('D', 5), ('D', 6),
('D', 7), ('D', 8), ('D', 9), ('D', 10), ('D', 11), ('D', 12), ('D', 13)]
----------
選んだカードは('C', 7)です
----------
[('C', 1), ('D', 8), ('C', 2), ('S', 8), ('D', 5), ('D', 2), ('S', 3), ('H',
10), ('C', 8), ('C', 4), ('H', 11), ('D', 3), ('S', 4), ('S', 7), ('S', 1),
('C', 5), ('H', 13), ('S', 2), ('S', 13), ('D', 1), ('C', 3), ('C', 13), ('H',
7), ('C', 10), ('D', 10), ('S', 5), ('D', 13), ('H', 5), ('H', 12), ('S', 10),
('D', 12), ('D', 6), ('C', 6), ('H', 3), ('H', 9), ('D', 9), ('D', 11), ('S',
12), ('S', 9), ('H', 2), ('C', 11), ('H', 8), ('S', 11), ('H', 4), ('C', 12),
('S', 6), ('C', 9), ('H', 1), ('C', 7), ('H', 6), ('D', 7), ('D', 4)]
----------
```