

変数の集合体：コンテナ(Container)

int, float, str, boolなどの変数を集合させた型があります。これらをコンテナ(Container)と呼んでいます。

- リスト(list): 四角カッコの中に並べたもの
- タプル(tuple) : コンマで区切って並べたもの
- 辞書ディクト(dict) : 波カッコの中で、キーワード：値のペアを並べたもの
- 集合(set) : 波カッコの中に並べたもの。いわゆる集合

class / type	空集合の作り方	集合の作り方の例	要素の呼び出し方の例
list	<code>L = list()</code> または, <code>L = []</code>	<code>L = [13, 'hello', 0b001]</code>	<code>L[1:] => ['hello', 1]</code>
tuple	<code>T = tuple()</code> または, <code>T = ()</code>	<code>T = 13, 'hello', 0b001,</code>	<code>T[1:] => ('hello', 1)</code>
set	<code>S = set()</code>	<code>S = {13, 'hello', 0b001}</code>	-
dict	<code>D = dict()</code> または, <code>D = {}</code>	<code>D = {'first':13, 'second':'hello', 'third':0b001}</code>	<code>D['second'] => hello</code>

list と set

listとsetの違いは、

- listは順序付き集合
- setは順序なし集合

という違いです。この特性の違いのせいで、

listは インデックス を用いて要素を参照できるのに対し、setは インデックス（順位）がないため、要素を参照できません。

```
>> L = [13, 'hello', 0b001]
>> S = {13, 'hello', 0b001}
>> print(L)
[13, 'hello', 1]
>> print(S)
{1, 'hello', 13}
>> print(L[1])
hello
>> print(S[1])
TypeError: 'set' object is not subscriptable
```

dictはsetをkeyで参照できるようにしたもの

dictもsetと同様に波カッコで囲まれる集合で、順番はありません。
ですが要素を参照できるようにkeyを指定しています。

```
>> S = {13, 'hello', 0b001}
>> D = {'first':13, 'second':'gello', 'third':0b001}
>> print(S)
{1, 'hello', 13}
>> print(D)
{'first': 13, 'second': 'hello', 'third': 1}
>> print(S[1])
TypeError: 'set' object is not subscriptable
>> print(D['second'])
'hello'
```

list（リスト）とtuple（タプル）

listとtupleの違いは、

- tuple は immutable（イ・ミュータブル）
- list は mutable(ミュータブル)

immutable な tuple の集合はオブジェクトID(格納されたアドレス)に、要素や集合の形が固定されます。つまり初期化後に要素の変更ができません。

対して、mutable な listの集合は要素が固定されません。

```
>> T = (13, 'hello', 0b001)
>> L = [13, 'hello', 0b001]
>> print(T)
(13, 'hello', 0b001)
>> print(L)
[13, 'hello', 0b001]
>> id(T)
2089650759296
>> id(L)
2089650910976

>> # copy objectID
>> T2 = T
>> print(T2)
(13, 'hello', 0b001)
>> id(T2)
2089650759296 # Tと同じObjectID
>> T += ('add',) # objectIDは変わる
>> id(T)
2089650847888
>> print(T)
(13, 'hello', 1, 'add')
>> print(T2)
```

```
[13, 'hello', 1) # T と T2は異なるオブジェクト

>> L2 = L
>> print(L2)
[13, 'hello', 0b001]
>> id(L2)
2089650910976 # Lと同じObjectID
>> L += ('add') # objectIDは変わらない
>> id(L)
2089650910976
>> print(L)
[13, 'hello', 1, 'add']
>> print(L2)
[13, 'hello', 1, 'add'] # L と L2は同じオブジェクトなので、Lを変えるとL2も変わる
```

list, tuple, dict, set

name	height	weight	gender
Suzuki	175	65	male
Saito	173	64	male
Nakamura	168	50	female
Takahashi	181	90	male

list

data-frame: dict in set

- df = {
 - {'name': 'Suzuki', 'height': 175, 'weight': 65, 'gender': 'male' },
 - {'name': 'Saito', 'height': 173, 'weight': 64, 'gender': 'male' },
 - {'name': 'Nakamura', 'height': 168, 'weight': 50, 'gender': 'female'},
 - {'name': 'Takahashi', 'height': 181, 'weight': 90, 'gender': 'female' }

list:

- Name = ['Suzuki', 'Saito', 'Nakamura', 'Takahashi']
- Height = ['175', '173', '168', '181']

tuple:

- T1 = ('Suzuki', 175, 65, 'male',)
- T2 = ('Saito', 173, 64, 'male',)

dict:

- D1 = {'name': 'Suzuki', 'height': 175, 'weight': 65, 'gender': 'male' }

range関数

start値からstop値までstep値きざみでタプルを作る関数`range(start=0, stop, step=1)`が用意されています。(厳密には、展開したタプルを作るわけではなく、展開方法の記述のみで range型です)

例えば,

```
L = list(10, 12, 14, 16, 18, 20)
```

は、10から**21まで** 2刻みでコンマ区切りの数列を作り、それをリストにしていますが、同じことは`range(10, 21, 2)`でおこなえます。

```
L = list(range(10, 21, 2))
```

unpack

コンテナの要素を抽出することができます。

```
>> L = [13, 'hello', 0b001]
>> l1 = L[0]
>> l2 = L[1]
>> l3 = L[2] # これでもいいですが, , ,
>> l1, l2, l3 = L # アンパック. 数が合っていないといけません.
>> l1, *l2 = L # *のついた変数がコンテナになって残りをアンパック
>> print(l2)
['hello', 1]
>> *l1, l2 = L # *のついた変数はどこでもいい.
>> print(l1)
[13, 'hello']
```

for 文

仮の引数をxとして,

```
pre-process

for x in コンテナ
    x に対するprocess

post-process
```

コンテナ要素のそれぞれひとつずつ取り出してxに代入し, 処理します. すべての要素を取り出し終わったらブロックを抜けます.

- リスト

```
>> L = [13, 'hello', 0b001]
>> for i in L:
...     print(i)
...
13
hello
1
```

- タプル

```
>> T = (13, 'hello', 0b001,)
>> for i in T:
...     print(i)
...
13
hello
1
```

- set

```
>> S = {13, 'hello', 0b001}
>> for i in S:
...     print(i)
...
1
hello
13
```

- 辞書. keyが参照されます.

```
>> D = {'first':13, 'second':'hello', 'third':0b001}
>> for i in D:
...     print(i)
...
first
second
third
>> for i in D:
...     print(D[i])
...
13
hello
1
```

- range関数

```
>> for i in range(3, 10, 2):
...     print(i)
...
3
5
7
9
```

内包表記

コンテナは、他のコンテナを使ったfor文で作ることもできます。内包表記といいます。

```
[x for x in コンテナ xに対するprocess]
```

```
>> L = [13, 'hello', 0b001]
>> L2 = [x for x in L]
>> print(L2)
[13, 'hello', 1]

>> L = [x for x in range(10) if x%2==0 ]
>> print(L)
[0, 2, 4, 6, 8]
```

for-ifのbreak-continue-pass

- breakは、その後のループブロックを処理せず、ループを抜ける
- continueは、その後のループブロックを処理せず、次のループに進む
- passは、何もしない

```
for i in range(0, 10):
    if i == 2:
        pass # 何もしない。空行と同じ。（条件分岐による特別処理はしないということ）
    elif i == 4:
        continue # この下の`elif i == 6`以降をおこなわず、次のiに進む
    elif i == 6:
        break # この下の`print(i)`以降をおこなわず、ループをやめる
    print(i)
print(99)
```

を実行すると、

```
0
1
2
3
5
99
```

メソッド(Method)と関数(Function)

関数

数学で「関数」というのがありますね.

$f(x,y) = x^2 + y - 3$

この f のことを関数, x,y のことを変数と呼んでいて, この関数を使うときは,

$3 \cdot f(4,-1) + 5$

のように具体的な数値を入れるのでした. 関数の部分の答えは関数の定義の x,y の部分に具体的な数値を入れたときの計算結果になりますね.

プログラミング言語でも, 「関数」があります.

関数を定義するときは,

```
def func1(x,y):                                # def 関数名 ( 引数1・引数2, ) コロン
    '''                                         # このコメントアウトをDocStringと言います.
    func1(x,y) : x^2 + y -3 を返します.      # 関数の説明などを書きます.
    '''
    ans = x ** 2
    ans = ans + y - 3
    return ans                                # 呼び出し側に返すものを returnで指定します.
```

のように書き, 使うときは,

```
x = 4
y = -1
z = 3 * func1(x,y)
```

のように使います. 呼び出し側に何も返さない場合もあります.

```
def func2(x,y):
    '''
    func2(x,y) : x^2 + y -3 の計算結果を表示します.
    '''
    ans = x ** 2
    ans = ans + y - 3
    print(ans)

# 関数定義のブロックに続けるときは1行あける.
x = 4
y = -1
z = func2(x,y)
print(z)
```

これを実行すると

```
12
None
```

となります。最初の「12」は関数の中の`print(ans)`の実行結果です。最後の「None」は呼び出し側の`print(z)`の実行結果。

つまり、`z`には何も値が入って来なかったということです。

また

```
def func3(x,y):
    x = x ** 2
    y = x + y - 3
    return y

x = 4
y = -1
z = 3 * func3(x,y)
```

これを実行したあと、`x`や`y`の値はどうなるでしょうか。

実は、変わりません。関数定義の中で使った`x,y`は、呼び出し側の`x,y`と無関係であり、単に1番目の引数、2番目の引数の値を受け取る新しい変数でしかないということです。

よって、

```
def func3(a,b):
    a = a ** 2
    b = a + b - 3
    return b

x = 4
y = -1
z = 3 * func3(x,y)
```

でも構いません。

もともと用意されている関数として、`print()`、`id()`、`type()`、`bin()`、`str()`などなどあります。

method

関数の一種ですが、メソッドと呼ぶ特殊関数があります。これは、型毎に定義されている関数で、「変数.関数名(引数1, 引数2, ...)」という呼び出し方をします。関数定義のしかたは、のちのち時間があれば紹介しますが、今は呼び出し方をマスターしましょう。

`container`において、要素の追加に用いた `コンテナ.append(追加する要素)`、要素の削除に用いた `.pop()` がメソッドです。

メソッドと関数の大きな違い, というか使われ方の違いは,

- 呼び出し方の違い
 - メソッド : `object1.method(object2,...)`
 - 関数 : `function(object1, object2,...)`
- オブジェクトへの関与 : 「引数は参照されるだけ」
 - メソッド : `object1` が書き換えられる. `object1`はミュータブル
 - 関数 : `object1` が書き換えられない.
- 戻り値
 - メソッド : 戻り値が無い場合もある. `object1`自体が答えになる場合がある.
 - 関数 : 原則的に戻り値がある.