

引き継ぎ資料 Vol.3

CにないPythonの世界

2016/07/??

コンセプト

Cには無いPythonの世界を堪能しよう!

目次

1. タプル・リスト・ディクショナリ
2. 関数
3. オブジェクト
4. 名前空間
5. numpy・matplotlib

目次

1. タプル・リスト・ディクショナリ

2. 関数

3. オブジェクト

4. 名前空間

5. numpy・matplotlib

Cの言語仕様にはないデータ構造

- ・ タプル
- ・ リスト
- ・ ディクショナリ

タプルとは

数が並んだもの

Cで近い機能は配列

タプル

簡単な例

```
>>> t = (1, 2, 3)
>>> print(t)
(1, 2, 3)
>>> print(t[0])
1
```

ここまではCの配列と同じ

タプル

あえてCで書くなら...

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int t[] = { 1, 2, 3 };
    printf("%d\n", t[0]);
    return 0;
}
```

簡単!

難しい例

```
>>> t = 1, 2, 3
>>> x, y, z = t
>>> def hoge():
...     return 4, 5, 6
...
>>> a1, a2, a3 = hoge()
>>> a = hoge()
>>> u, v, w = z, y, x
```

それぞれの変数の中身は？

```
>>> t = 1, 2, 3
>>> x, y, z = t
>>> def hoge():
...     return 4, 5, 6
...
>>> a1, a2, a3 = hoge()
>>> a = hoge()
>>> u, v, w = z, y, x
```

```
>>> print(t)
(1, 2, 3)
>>> print(x, y, z)
1 2 3
>>> print(a1, a2, a3)
4 5 6
>>> print(a)
(4, 5, 6)
>>> print(u, v, w)
3 2 1
```

タプルの要点

- ・ タプルに必要なのは“,”(カンマ)
- ・ 複数の値を返す関数はタプルを一つ返す関数
- ・ タプルは自動的に展開され複数の変数に代入
- ・ **タプルでは要素の変更は不可**

タプルでは要素の変更は認められない

```
>>> t = (1, 2, 3)
```

```
>>> t[1] = 4
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

リストとは

オブジェクト(整数, 小数, その他)が並んだもの
後から変更が可能

```
>>> t = []
>>> t.append(1)
>>> t.append(2)
>>> print(t)
[1, 2]
>>> t[1] = 3
>>> print(t)
[1, 3]
>>> lst = [1, 3, 5, 7, 9]
>>> print(lst)
[1, 3, 5, 7, 9]
>>> [a, b] = [1, 3]
```

便利な例

```
>>> t = [1, 3, 5, 7, 9]
>>> print(t[1:4])
[3, 5, 7]
>>> print(len(t))
5
>>> s = [2, 4, 6, 8]
>>> print(t + s)
[1, 3, 5, 7, 9, 2, 4, 6, 8]
>>> print(t * 2)
[1, 3, 5, 7, 9, 1, 3, 5, 7, 9]
>>> print(2 in t, 3 in t)
False True
```

リスト

難しい例

```
>>> t = [1, 2, 3, 4, 5]
>>> def hoge(x):
...     return x**2
...
>>> xs = [hoge(x) for x in t]
```


ディクショナリとは

添字に数字以外が使えるリスト

言語によっては連想配列・Map・HashMapなど

簡単な例

```
>>> d = {"a": 1, "b": 2, "c": 3}
>>> print(d["a"])
1
>>> d["b"] = 5
>>> print(d)
{'b': 5, 'c': 3, 'a': 1}
```

注意点

- ・ 存在しない要素を取得しようとするとエラー
- ・ 存在しない要素に代入すると要素を追加

まとめ

タプル・リスト・ディクショナリはCには無い

Pythonにはデフォルトで存在

お互いにネスト(入れ子)が可能

Cよりも柔軟に複雑なデータ構造を表現可能

目次

1. タプル・リスト・ディクショナリ

2. 関数

3. オブジェクト

4. 名前空間

5. numpy・matplotlib

Cの関数どこまでマスターしてますか？

Cの関数

```
int add(int a, int b) {  
    return a + b;  
}  
  
void swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
void map(int* array, int N, int (*func)(int)) {  
    for (int i = 0; i < N; ++i)  
        array[i] = func(array[i]);  
}
```

Cで出てきた概念

- ・ 引数
- ・ 戻り値
- ・ 値渡し・参照渡し
- ・ 関数ポインタ

Cで出てきた概念

- ・ 引数
- ・ 戻り値
- ・ 値渡し・参照渡し
- ・ 関数ポインタ

Pythonはもう少し難しい概念を持つ

Python関数の基本

```
>>> def add(a, b):  
...     return a + b  
...  
>>> print(add(1, 3))  
4
```

Python関数の基本

```
>>> def add(a, b):  
...     return a + b  
...  
>>> print(add(1, 3))  
4
```

add 関数名

a, b 引数

return a + b a + bを戻り値とする

add(1, 3) 関数呼び出し

Pythonの関数を持つ概念

- ・ 引数
 - ・ 参照の値渡し
 - ・ キーワード引数
- ・ 戻り値
- ・ 関数オブジェクト
 - ・ lambda式

引数

関数処理の材料

Cの場合二種類:

値渡し 変数をそのまま渡す

参照渡し 変数のポインタを渡す

Pythonは全て**参照の値渡し**

引数

関数処理の材料

Cの場合二種類:

値渡し 変数をそのまま渡す

参照渡し 変数のポインタを渡す

Pythonは全て**参照の値渡し**

これがとってもややこしい

参照の値渡し

```
>>> def swap(a, b):  
...     tmp = a  
...     a = b  
...     b = tmp  
...  
>>> def swap_e(lst, i, j):  
...     tmp = lst[i]  
...     lst[i] = lst[j]  
...     lst[j] = tmp  
...
```

```
>>> a, b = 1, 2  
>>> swap(a, b)  
>>> print(a, b)  
1 2  
>>> lst = [1, 3, 5, 7]  
>>> swap_e(lst, 2, 3)  
>>> print(lst)  
[1, 3, 7, 5]
```

参照の値渡し

Pythonの引数はローカル変数に**代入**される

不変性のオブジェクト(整数・小数・タプル…)

- ・ Cの値渡しと似た動作
- ・ 関数内での書き換えは呼び出し元に影響しない

可変性のオブジェクト(リスト・ディクショナリ…)

- ・ Cの参照渡しと似た動作
- ・ 関数内での書き換えが呼び出し元に影響する

参照の値渡し

Pythonの引数はローカル変数に**代入**される

不変性のオブジェクト(整数・小数・タプル…)

- ・ Cの値渡しと似た動作
- ・ 関数内での書き換えは呼び出し元に影響しない

可変性のオブジェクト(リスト・ディクショナリ…)

- ・ Cの参照渡しと似た動作
- ・ 関数内での書き換えが呼び出し元に影響する

思わぬバグの可能性あり

高度な引数の指定

ライブラリのリファレンスでまれに見る形式

```
def func(*args, **kwargs)
```

こいつの*args, **kwargsって何?って話

引数の形式

どちらも可変長引数を実現

***args**

- ・ 位置で指定された引数が格納
- ・ 引数がタプル形式で格納

****kwargs**

- ・ キーワードと共に指定された引数が格納
- ・ 引数がディクショナリ形式で格納

```
>>> def hoge(*args, **kwargs):  
...     print(args)  
...     print(kwargs)  
...  
>>> hoge(1, "a", cat="cute", dog="mustdie")  
(1, 'a')  
{ 'dog': 'mustdie', 'cat': 'cute' }
```

```
>>> def hoge(*args, **kwargs):  
...     print(args)  
...     print(kwargs)  
...  
>>> hoge(1, "a", cat="cute", dog="mustdie")  
(1, 'a')  
{ 'dog': 'mustdie', 'cat': 'cute' }
```

リファレンスに明確に書かれていない引数も渡せる

基本的にはCと同じ

ただし...

- ・ 複数の値を返す事が可能
- ・ return文を書かなくても関数はNoneを返す

関数オブジェクト

Pythonでは関数もオブジェクトの一つ
いろんなことが可能

- ・ 変数に代入
- ・ 引数として渡す

Cの関数ポインタに近い

例-変数に格納-

```
>>> def square(x):  
...     return x ** 2  
...  
>>> square_func = square  
>>> print(square_func(2))  
4
```


例-引数-

```
>>> def mapping(seq, func):  
...     mapped = []  
...     for e in seq:  
...         mapped.append(func(e))  
...     return mapped  
...  
>>> def square(x):  
...     return x ** 2  
...  
>>> t = [1, 3, 5]  
>>> t_squared = mapping(t, square)  
>>> print(t_squared)  
[1, 9, 25]
```

何に使うのか？

例えば...

- ・ 条件に合うものを抽出したい
 - ・ 条件とリストをそれぞれ格納
 - ・ 両方をfilter関数の引数に
- ・ 積分をしたい
 - ・ 被積分関数を作る
 - ・ `scipy.integrate.quad`関数に渡す
- ・ 極値を探したい
 - ・ 微分した関数を作る
 - ・ `scipy.optimize.root`関数に渡す

数学的なことを考える時に相性が良い(気がする)

lambda式

いちいち関数を外部に定義するの面倒
そんなあなたにlambda式

ちょっとした関数なら簡単にかける

```
>>> square_func = lambda x: x ** 2
```

ちょっとした関数なら簡単にかける

```
>>> square_func = lambda x: x ** 2
```

複雑な関数は書くと読みづらいので注意

目次

1. タプル・リスト・ディクショナリ

2. 関数

3. オブジェクト

4. 名前空間

5. numpy・matplotlib

目次

1. タプル・リスト・ディクショナリ

2. 関数

3. オブジェクト

4. 名前空間

5. numpy・matplotlib

目次

1. タプル・リスト・ディクショナリ

2. 関数

3. オブジェクト

4. 名前空間

5. numpy・matplotlib