

引き継ぎ資料 Vol.4

オブジェクトの話

2016/08/??

コンセプト

“オブジェクトって何?” を解決

目次

1. 導入
2. クラス
3. カプセル化
4. 繙承
5. ダック・タイピング
6. まとめ

目次

1. 導入
2. クラス
3. カプセル化
4. 繙承
5. ダック・タイピング
6. まとめ

導入

Q.

オブジェクトって何?

導入

Q.

オブジェクトって何?

A.

データや構造としての“何か”

例

```
>>> int_variable = 10 # int object
>>> float_variable = 3.7 # float object
>>> str_object = "Hello" # str object
>>> none_object = None # NoneType object
>>> list_object = [] # list object
>>> dict_object = {} # dict object

>>> # numpy.ndarray object
>>> array_object = numpy.array([1, 2, 3])
```

例

```
>>> int_variable = 10 # int object
>>> float_variable = 3.7 # float object
>>> str_object = "Hello" # str object
>>> none_object = None # NoneType object
>>> list_object = [] # list object
>>> dict_object = {} # dict object

>>> # numpy.ndarray object
>>> array_object = numpy.array([1, 2, 3])
```

なんでもオブジェクト

オブジェクトの中身

- ・ 関数 (正確にはメソッド)
- ・ 他のオブジェクト

例

```
>>> list_object = [1, 2, 3, 4] # list object
>>> # list_objectの中にあるappendメソッド
>>> list_object.append(5)
>>> # list_objectがappendという操作を受け付けて
>>> # リストの中に5を追加
>>> print(list_object)
[1, 2, 3, 4, 5]
>>> # numpy.ndarray object
>>> array_object = numpy.array([
...     [1, 2, 3],[4, 5, 6]
... ])
>>> print(a.shape) # aの中のタプルオブジェクト
(2, 3)
```

型

オブジェクトは様々な中身を持つことが可能
中身のテンプレート → 型

オブジェクトの作成 = 型から物を作成



型って自分で作れないの？

型って自分で作れないの？

自作できる型 → クラス

目次

1. 導入
2. クラス
3. カプセル化
4. 繙承
5. ダック・タイピング
6. まとめ

クラスの話をする前に…

C の “構造体” 覚えていますか？

クラス

クラス…C の構造体 α

クラス

クラス…C の構造体 $+\alpha$

$+\alpha$ の部分がとても巨大

復習 - 構造体 -

“車”を表す構造体

```
typedef struct Car{  
    double x;  
    double y;  
} Car;
```

- ・メンバは x と y
- ・それぞれには. でアクセス

復習 - 構造体 -

```
int main(int argc, char** argv){  
    Car car1, car2;  
    car1.x = 1.0;  
    car1.y = 1.0;  
    car2.x = 5.0;  
    car2.y = 5.0;  
    return 0;  
}
```

car1 と car2 は別物

復習 - 構造体 -

```
int main(int argc, char** argv){  
    Car car1, car2;  
    car1.x = 1.0;  
    car1.y = 1.0;  
    car2.x = 5.0;  
    car2.y = 5.0;  
    return 0;  
}
```

car1 と car2 は別物

car1 と car2 は Car 型のオブジェクト

Python で構造体的なこと

```
class Car:  
    def __init__(self):  
        self.x = 0.0  
        self.y = 0.0  
  
def main():  
    car1 = Car()  
    car2 = Car()  
    car1.x, car1.y = 1.0, 1.0  
    car2.x, car2.y = 5.0, 5.0
```

Car はクラス

__init__

```
class Car:  
    def __init__(self):  
        self.x = 0.0  
        self.y = 0.0
```

- ・ __init__ は関数
- ・ クラスの中に宣言されている → メンバ関数
- ・ オブジェクトの作成時に呼び出し
- ・ self は自分自身を指す

2つの run 関数は同じ処理

```
class Car:  
    def __init__(self):  
        self.x, self.y = 0.0, 0.0  
    def run(self):  
        self.x += 1.0 # selfのxを増加  
  
def run(car):  
    car.x += 1.0 # 与えられたcarのxを増加  
  
def main():  
    car1 = Car()  
    car1.run()  
    run(car1) # car1.x == 2.0
```

メンバ関数

どちらの関数でも同じ処理が可能

メンバ関数

どちらの関数でも同じ処理が可能

メンバ関数が不要?

メンバ関数

どちらの関数でも同じ処理が可能

メンバ関数が不要?

そんなことはない

2つの run 関数は同じ？

```
class Car:  
    # 中略  
    def run(self): # ここにはCarしかこない  
        self.x += 1.0  
  
def run(car): # ここにはCarに関係ないものが来れる  
    car.x += 1.0
```

- ・ 関係あるものは関係する場所に配置
- ・ 事故防止は大事

2つの run 関数は同じ？

```
class Car:  
    # 中略  
    def run(self): # ここにはCarしかこない  
        self.x += 1.0  
  
def run(car): # ここにはCarに関係ないものが来れる  
    car.x += 1.0
```

- ・ 関係あるものは関係する場所に配置
- ・ 事故防止は大事

メンバ関数超大事

オブジェクト指向

プログラム全体をオブジェクトの集合で構成する手法

概念

- ・ カプセル化
- ・ 繙承
- ・ ポリモーフィズム

大流行した考え方

説明する内容

- ・ カプセル化
- ・ 繙承
- ・ ダック・タイピング

ポリモーフィズムよりダック・タイピングの方が
Pythonic なのでそっちを説明

目次

1. 導入
2. クラス
3. カプセル化
4. 繙承
5. ダック・タイピング
6. まとめ

カプセル化 = クラスの中身を隠蔽



例

Car クラスに燃料の概念を追加

```
class Car:  
    def __init__(self):  
        self.gas = 0.0 # 燃料  
  
def main():  
    car = Car()  
    car.gas = 10.0 # 燃料を外部から直接操作
```

例

Car クラスに燃料の概念を追加

```
class Car:  
    def __init__(self):  
        self.gas = 0.0 # 燃料  
  
def main():  
    car = Car()  
    car.gas = 10.0 # 燃料を外部から直接操作
```

問題点

車内部のガソリン (gas) を外部から操作可能

問題点なのか？

この状態なら燃料を操作されて困ることは無い

問題点なのか？

この状態なら燃料を操作されて困ることは無いが

問題点なのか？

この状態なら燃料を操作されて困ることは無いが
例えばこれが

- ・重要なフラグ
- ・変更された時に何らかの処理をしたい変数

だったなら？

例

```
class Car:  
    def __init__(self):  
        self.gas = 0.0 # 燃料  
  
    def lamp(self):  
        # 燃料が少なくなったら呼びたい！  
        print("燃料が少ない！")
```

どうやって lamp 関数を呼び出すか？

解

```
def main():
    car = Car()
    car.gas = 10.0 # 燃料を外部から直接操作
    # 何らかの処理
    if car.gas <= 1.0: # main関数側でチェック
        car.lamp()
```

解

```
def main():
    car = Car()
    car.gas = 10.0 # 燃料を外部から直接操作
    # 何らかの処理
    if car.gas <= 1.0: # main関数側でチェック
        car.lamp()
```

問題点

Car を使う側が lamp を知っている必要あり

自動でランプは起動しない!

カプセル化を使用

```
class Car:  
    def __init__(self, gas):  
        self.__gas = gas # 燃料  
    def set_gas(self, value):  
        self.__gas = value  
        if value <= 0.1:  
            self.lamp()  
    def lamp(self):  
        print("燃料が少ない！")  
def main():  
    car = Car(10.0) # ガスを渡して初期化  
    # 何らかの処理  
    car.set_gas(0.05)
```

カプセル化を使用

- Car を使う側は lamp を知らなくていい
- gas に応じた処理を内部に追加可能
 - 追加されてもユーザは知らなくていい

カプセル化

- 外部から見えていいもの・いけないものを区別
- アクセス権限を制御

Table 1: アクセス権限

名称	範囲	命名規則 ¹
private	自分のクラス内	_ から名前を開始
protected	自分と継承先内	_ から名前を開始
public	どこからでも	_ をつけない

¹Google Python Style Guide

注意点

- Python は private でも外から呼ぶことが可能
- private なものを外から呼んではいけない
private = 外から呼ぶなという意思表示
- _ から始まるものは外から呼ばない!

まとめると…

ゲームの中身を知らなくても遊べるでしょうって話
改造してはいけない



目次

1. 導入
2. クラス
3. カプセル化
4. 繙承
5. ダック・タイピング
6. まとめ

継承 = クラスを拡張



例

```
class Car:  
    def __init__(self):  
        self.gas = 0.0 # prublic  
class SunRoof(Car):  
    def __init__(self):  
        self.__roof.opend = False # private  
    def open_roof(self): # public  
        self.__roof.opend = True  
def main():  
    car = SunRoof() # carはgasとopen_roofを持つ  
    car.gas = 10.0 # ガスを注入  
    car.open_roof() # 屋根を開く！
```

よくある使い方

ライブラリで容易されたクラスを継承
自分に必要なクラス化

目次

1. 導入
2. クラス
3. カプセル化
4. 繙承
5. ダック・タイピング
6. まとめ

名言



“If it walks like a duck and quacks like a duck, it must be a duck.”

ダック・タイピング

そのオブジェクトが決まった関数を持っているなら,
それが何であろうと無関係

例

```
def add(x, y):
    return a + b

def main():
    add(1, 3) # 数字の足し算
    add([1, 2, 3], [4, 5]) # リストの足し算
```

add 関数は対象が足し算可能ならなんでもよい!

例

```
def add(x, y):
    return a + b

def main():
    add(1, 3) # 数字の足し算
    add([1, 2, 3], [4, 5]) # リストの足し算
```

add 関数は対象が足し算可能ならなんでもよい!

このようにオブジェクトに一定の関数があればよい

Python におけるダックタイピング

特殊メソッド

特殊メソッド

- ・特定の構文を使う際に呼び出される関数
- ・`__init__`, `__call__`, `__str__` など様々

例

```
class QuadraticFunction:  
    def __init__(self, a, b, c):  
        self.a, self.b, self.c = a, b, c  
    def __call__(self, x):  
        return self.a*x**2 + self.b*x + self.c  
    def __str__(self):  
        return "{}x^2+{}x+{}".format(  
            self.a, self.b, self.c)  
  
def main():  
    fx = QuadraticFunction(1, 2, 1) #call __init__  
    print(str(fx)) #call __str__  
    for x in range(10):  
        print(fx(x)) #call __call__
```

特殊メソッド

- ・特定の構文を使う際に呼び出される関数
- ・`__init__`, `__call__`, `__str__` など様々
- ・関数のようなオブジェクトも作成可能

目次

1. 導入
2. クラス
3. カプセル化
4. 繙承
5. ダック・タイピング
6. まとめ

“オブジェクト”の概念は使いこなせば便利
データ解析の分野では作る必要は無い可能性が高い
使えないのは危険

Questions?