第 3 回 Python オブジェクト指向ゼミ

ゼミ担当者　：　藤田 宗佑，西本 要
開催日　：　2009 年 5 月 12 日

ゼミ内容: Python にとってオブジェクト指向はオプションであると言われている．しかし，クラスは Python の構成要素の中でも特に再利用性が高いものであり，プログラムの規模が大きくなった場合に以前作ったクラスを再利用できれば，大幅な労力と時間の節約につながる．本ゼミでは，Python でオブジェクト指向を実装する場合，どのように記述し，実行されるかについて学ぶ．

1　クラス

クラスはオブジェクト指向を語る上で重要な要素であり，プログラムで用いる部品の設計図とも言える．オブジェクト指向プログラミングではこのクラス（設計図）を基に，インスタンス（部品）を生成することでプログラムを構成する．

1.1　クラスを定義する

Python においてクラスを定義するには class 文に続けてクラス名を記述する．以下にクラスの定義とインスタンスの生成を示す．

クラスの定義

```
class クラス名:
    #クラス定義の内容を記述する
#インスタンスの生成 変数 = クラス名()
```

1.2　メソッドの定義と呼び出し

クラス内部で定義する関数のことを，メソッドと呼ぶ．以下にメソッドの定義と呼び出しについて示す．

メソッドの定義

```
class クラス名:
    def メソッド名(self,arg1,arg2,…):
        #メソッドの定義内容を記述する
```

メソッドは関数同様に def 文によって定義されるが，第 1 番目の引数として必ず self という引数を添える．この引数は，そのクラスから生成されたインスタンスそのものを示す．

クラスのメソッドの呼び出しは，インスタンス名とメソッド名をドットで区切ることで行う．

クラスメソッドの呼び出し

```
#インスタンスの生成
a = クラス名()

#メソッドの呼び出し 1
a.メソッド名()

#メソッドの呼び出し 2
クラス名.メソッド名(a) #第一引数に注意！
```

1.3　アトリビュート（属性）の定義と呼び出し

クラスやインスタンス独自のデータを保存するには，アトリビュート（属性）を用いる．

アトリビュートの定義

```
#メソッド内外を問わずに定義可能
class クラス名:
    アトリビュート名 = value #代入が必須
    def メソッド名(self,引数):
        self.アトリビュート名 = 引数
        print self.アトリビュート名

c = クラス名()
c.メソッド名(10)
>>> 10
```

アトリビュートの作成は，”self. アトリビュート名”を指定し，値を代入することによって行われる．

また，”クラス名. アトリビュート名”によってクラス定数を生成することが可能である．このクラス定数は，インスタンス毎に独自に生成されるものではなく，一つのクラスから生成された全てのインスタンスにおいて共通の値を保持する．

sample code 1(monster.py)

```
class Monster:
   name = "aaa"
   def setName(self, name):
      self.name = name
   def getName(self):
      return self.name

m = Monster()
print m.getName()
# >>> aaa

m.setName("bbb")
print m.getName()
# >>> bbb
```

1.4 インスタンスの初期化

Python では，インスタンスの生成の際に呼び出す初期化メソッドとして__init__() を用いる．

インスタンスの初期化と引数の指定

```
class [クラス名]:
   def __init__(self, [引数]):
      self.[アトリビュート名 1] = 10
      self.[アトリビュート名 2] = [引数]

a = [クラス名]([引数]):
```

1.5 オーバーロード

オーバーロードとは，同じ名の関数であるが，戻り値や引数の数，データ型などが異なるメソッドを複数定義することである．しかし，Python では引数が異なるメソッドでも，同じ名前を持つ場合，先に定義したメソッドが後に定義したメソッドによって上書きされる．

オーバーロードの失敗

```
class C:
   def a(self, x, y):
      print x * y
   def a(self, x):
      print x

c = C()
c.a(10, 10)
>>> エラー出力
c.a(10)
>>> 10
```

1.6 クラスのドキュメンテーション文字列

クラス定義の直後に置かれた文字列のことをドキュメンテーション文字列と呼ぶ．クラスの機能を説明するために用いられ，オブジェクトの__doc__アトリビュートの値としてアクセスできる．

クラスのドキュメンテーション文字列

```
class C:
   """C.documentation"""

c = C()
print c.__doc__
>>> C.documentation
```

sample code 2(monster.py)

```
class Monster:
   """This is a class of monster."""
   def __init__(self, name, skill1,
skill2):
      self.name = name
      self.skill1 = skill1
      self.skill2 = skill2
   def setName(self, name):
      self.name = name
   def getName(self):
      return self.name

m = Monster("hoge", "fire" , "ice")
print m.__doc__
# >>> This is a class of monster.
print m.getName()
# >>> hoge
```

1.7 特殊なメソッド

演算子をオーバーロードする特殊なメソッドがある．これは，クラスのインスタンスに対して”+”，”|”のような四則演算や論理演算を用いる，またはインスタンスのインデクシングに用いることができる．特殊なメソッドは，前後に二つの下線”__”を付けて記述する．以下に特殊なメソッドの一部を示す．

インスタンスもフックメソッドの 1 つである．

1.8 特殊なアトリビュート

メソッドだけでなく，前後に二つの下線”__”を付けた特殊なアトリビュートが存在する．以下にその一例を示す．

ドキュメンテーション文字列も特殊なアトリビュートの 1 つである．

Table 1　特殊なメソッド例

メソッド	対応する処理	コード例
__init__	インスタンスの作成時の処理	Class()
__del__	インスタンスの消滅時の処理	
__add__	+演算子	X + Y, X += Y
__and__	|演算子 (ビット論理和)	X | Y, X |= Y
__repr__,__str__	出力，型変換	print X, 'X', str(X)
__call__	関数の呼び出し	X()
__getattr__	. を使用した属性へのアクセス	X.undefined
__setattr__	属性の値の代入	X.any = value
__getitem__	インデクシング	X[key], for ループ, X in Y
__setitem__	シーケンスの特定の要素への値の代入	X[key] = value
__len__	シーケンスの長さの確認	len(X), true/false の確認
__cmp__	比較一般	X == Y, X < Y
__lt__	小なり演算	X < Y (__cmp__でも対応可能)
__eq__	同等かの比較	X == Y (__cmp__でも対応可能)
__radd__	インスタンスが+演算子の右側に使われた場合	Noninstance + X
__iadd__	累算加算	X += Y (__add__でも対応可能)
__iter__	ループ，反復	for ループ, X in Y

Table 2　特殊なアトリビュート

アトリビュート	値
__dict__	インスタンス変数を保持する辞書
__class__	インスタンスの雛形となるクラスオブジェクト
__bases__	元のクラスが継承したスーパークラス
__doc__	ドキュメンテーション文字列
__slots__	インスタンスが利用可能な変数名のリスト

1.9　外部モジュール・クラスの読み込み

import 文や from 文を使うことで，モジュールに定義されているクラスをインポートできる．.py 拡張子を持つ外部ファイルからは，以下のようにしてクラスを読み込む．

クラスのインポート

```
# import 文によるインポート
import モジュール
a = モジュール.クラス名()

# from 文によるインポート
from モジュール import クラス名
a = クラス名()
```

2　クラスの継承

クラスの継承とは，あるクラスを雛形にして別のクラスを作ることである．雛形となるクラスをスーパークラス，それを元に作られたクラスをサブクラスと呼ぶ．スーパークラスに定義されたメソッド，アトリビュートは，基本的にはそのまま流用できる．

クラスの継承を行うには，class 文でスーパークラスを指定する．以下に示すようにクラス名の後に丸括弧を付け，その中に継承したいクラス（スーパークラス）の名前を記述する．なお，Python では複数のクラスを継承する多重継承が可能であり，これを行う際には括弧内部に継承したいクラス名を列記する．

クラスの継承

```
#継承の例
class サブクラス名(スーパークラス名):
   #クラス定義の内容を記述する

#多重継承の例
class サブクラス名(スーパークラス名１,
スーパークラス名２):
   #クラス定義の内容を記述する
```

2.1　メソッドのオーバーライド

スーパークラスにあるメソッドを，サブクラスのメソッドで上書きすることをオーバーライドという．

継承を行う際，初期化を行う__init__() メソッドも自動的に上書きされてしまう．よって，スーパークラスの__init__() における定義内容が必要な場合，スーパークラスの__init__() を明示的に呼び出す必要がある．

__init__() のオーバーライド

```
class サブクラス名(スーパークラス名):
   def __init__(self,引数):
      スーパークラス名.__init__(self,
引数)
```

sample code 3(suraimu.py)

```
from monster import Monster

class Suraimu(Monster):
   """This is a class of suraimu."""
   def __init__(self, name, skill1,
skill2, type):
      Monster.__init__(self, name, skill1,
skill2)
      self.type = type

s = Suraimu("suraimu", "fire", "ice",
"metal")
print s.getName()
# >>> suraimu
```

3 集約

集約とは，あるクラスが，他のクラスによって構成されていることを言う．例えば，車がタイヤやハンドル等の様々な部品から構成されているように，車というオブジェクトがタイヤオブジェクトやハンドルオブジェクト等の様々なオブジェクトで構成されている関係を示す．以下に集約の例を示す．

集約の例

```
class Car:
   def __init__(self):
      self.tire = Tire()
   def getTire(self):
      return self.tire

class Tire:
   def __init__(self):
      self.name = "BRIDGESTONE"
   def getName(self):
      return self.name

car = Car()
print car.getTire().getName()
>>> BRIDGESTONE
```

4 データの隠蔽

データの隠蔽はオブジェクト指向におけるカプセル化と混同されやすいが，本来別の物である．

カプセル化とは，関係の深いアトリビュート及びその振る舞いを記述したメソッドを一つのクラスにまとめることである．クラス内部をどのように変更しても，インタフェース部分が一貫していれば，クラスを使う側のコードは影響を受けない．一方で，データの隠蔽は，外部に公開する必要のないアトリビュートやメソッドを隠して保護することを意味する．

Python においてデータの隠蔽は実質不可能である．Python は，Java の private 修飾子のような隠蔽のための構文を持たない．そのため，実装による隠蔽が必要とされている．

なお，データの隠蔽のために用いられる機能として，ネームマングリングがある．これは，本来はアトリビュート名の重複を防ぐために利用される．以下に，ネームマングリングを実装したコードを示す．

メソッドの定義

```
class C:
    def meth(self):
       self.__x = 20
       print self.__x

I = C()     #インスタンス生成
I.meth()
>>> 20
I.__x     #__x に直接アクセス
>>> エラー表示
I._C__x     #_[クラス名]__[アトリビュート名]
>>> 20
```

ネームマングリングでは，アトリビュートを宣言する際，名前の先頭に二つの下線”__”を付加する．これにより，アトリビュートは”インスタンス名._クラス名.__アトリビュート名”としてアクセスされることになる．

本来は多重継承を行う場合の，アトリビュート名の衝突を防ぐために用いられるが，アクセスのための記述を増やす事で，偶然のアクセスを防ぐことが期待されている．

5 演習

5.1 スライムに技を使わせてみる

sample code 2(monster.py) において，アトリビュート「skill1」「skill2」についてメソッド「set***」「get***」を定義しなさい．

以下の Main.py 通りに実行できるように，sample code 2(monster.py) を完成させなさい．

メソッドの定義

```
from Suraimu import Suraimu

s = Suraimu("suraimu", "fire", "ice",
"metal")
s.useSkill1()
# >>> 'metalsuraimu' use 'fire'!!
s.useSkill2()
# >>> 'metalsuraimu' use 'ice'!!
```

5.2 社長命令「起立！」

以下の文章からクラスを作成しなさい．

社長クラスは 3 人の社員クラスを抱えている．社員クラスには担当クラス，主任クラス，部長クラスが存在する．社長クラスが「起立！」と命令すると社員クラスが起立する．その際，担当クラスは普通に起立し，主任クラスはすばやく起立し，部長クラスは偉そうに起立する．実行結果を以下に示す．

実行結果例 1

```
担当は普通に起立した.
主任はすばやく起立した.
部長は偉そうに起立した.
```

余力がある人は，以下の文章からクラスを拡張しなさい．

- 社長クラスから新たな命令が追加される．社長クラスが基本給を社員クラスに教えると，社員クラスは自分の給料を答える．担当クラスの給料は基本給に等しく，主任クラスの給料は基本給の 2 倍，部長クラスの給料は基本給の 3 倍である．

- 社長クラスから新たな命令が追加される．社長クラスが基本給を社員クラスに教えると，社員クラスは自分のボーナスを答える．どの社員のボーナスも基本給の 3 倍である．

実行結果例を以下に示す．ただし，実行結果例の基本給は 10000 円である．

実行結果例 2

```
担当は普通に起立した.
給料は 10000 円です.
ボーナスは 30000 円です.
主任はすばやく起立した.
給料は 20000 円です.
ボーナスは 30000 円です.
部長は偉そうに起立した.
給料は 30000 円です.
ボーナスは 30000 円です.
```