オブジェクト指向プログラミング

第12回

箕原辰夫

自作のクラスを定義する

- class文を用いて自作のクラスを定義することができる
- Pythonのオブジェクトモデルは、クラス定義型(Class definition object model)と進化型(Evolutional object model)の折衷型になっている。
- クラス定義型は、Java, C++, C#, ActionScript（Falsh）などが採用している
- 進化型は、JavaScript, Luaなどが採用している

構造体としてのクラス

- クラスの定義
  - **class** クラス名: **pass**
  - 例： **class** Employee: **pass**
- passは何もしない文、クラス名は通常大文字始まりのことが多い
- クラスに属するオブジェクトの生成
  - クラス名( )
  - 例：Employee( )
- 生成したオブジェクトを参照する変数に代入
  - 変数名 = クラス名( )
  - 例：john = Employee( )

構造体としてのクラス

- あるクラスに属するオブジェクトのことをインスタンス(Instance)と呼ぶ。オブジェクトは、独自に変数をもつことができる。これをインスタンス変数と呼ぶ

- インスタンス変数を定義する
  - オブジェクト変数.変数名 = 値の代入
  - 例： john.name = "John Smith"

- 代入することによって、そのオブジェクトに属するインスタンス変数ができる

一般的なクラスの定義

- 属性（インスタンス変数）とメソッド（関数）を持つクラス

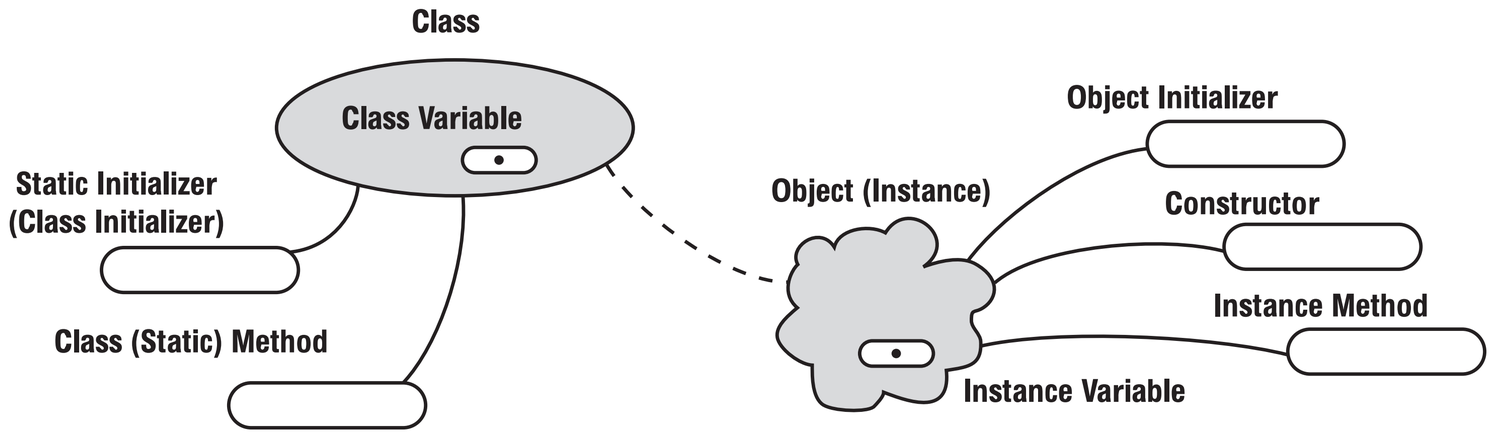
```
class クラス名:
    メソッドの定義
```

- 例：

```
class Binary:
    #self.valueを用いる

    def setValue( self, n ): self.value = ( n != 0 )
    def getValue( self ): return 1 if self.value else 0
```

クラスとオブジェクトの関係

- クラス–オブジェクト関係図



総合的なクラスの作成

- コンストラクタ（**self**引数あり）
- クラスメソッド（**self**引数なし）
- インスタンスメソッド（**self**引数あり）

- クラス変数（**self**なし）
- インスタンス変数（**self**つき）

コンストラクタ

- オブジェクト生成時に呼び出される特別なメソッド
  - __init__（ **self**, パラメータ名)

- 通常はクラス名( )のコンストラクタは仮定されている。もし、クラス名（　パラメータ名　）のコンストラクタを定義した場合は、上記の仮定が外れる。そのため、クラス名（）のコンストラクタを定義する必要がある。

- コンストラクタはインスタンスメソッドなので、**self**が使える。また、スーパークラスのメソッドを呼び出す**super()**も用いることができる

コンストラクタの例

```
class Fraction:
        def __init__( self,  num=0, denom=1 ) :
          self.numerator = num
          self.denominator = denom



number = Fraction( 7, 12 )
another = Fraction( )
```

その他の規定メソッド

- デストラクタ
  - オブジェクトが削除されるときに呼ばれる
  - **def** __del__(self): で定義
- 文字列化メソッド
  - print関数などで、オブジェクトを文字列に変換されるときに呼ばれる
  - **def** __str__(self): で定義
- 演算のためのメソッド
  - 比較演算子 True/Falseを返す

    __eq__, __lt__, __le__, __gt__, __ge__, __ne__
  - 四則演算子 結果のオブジェクトを返す

    __add__, __sub__, __mul__, __truediv__, __floordiv__, __pow__, __mod__, __divmod__, __lshift__, __rshift__, __and__, __or__, __xor__ 等

クラス変数の定義

- クラスの中に属性（フィールド）を持つための変数を定義できる。クラス変数と呼ばれている。

```
class クラス名:
    クラス変数の定義
```

- クラス変数は、オブジェクトを生成しなくても、クラス名を使って直接アクセスすることができる。
- 例：

```
class Tester:
    sample = "Hello" #クラス変数の定義
print( Tester.sample )
```

クラス変数

- そのクラスに（そのクラスに属するオブジェクトに）共通で 1 つの値を保持することができる変数を定義できる
- クラス変数には、**self**修飾子を用いない
- クラス変数にアクセスするときは、
  - クラス名.クラス変数名
- でアクセスする。

クラスメソッド

- クラス変数やクラスメソッドは、クラスだけが持つことのできる変数やメソッドである。

- クラスメソッドでは、特定のインスタンスに依拠しているわけではないので、**self**は使えない

- クラスメソッドを呼び出すときは、

  クラス名.クラスメソッド名(パラメータ)

  という形で呼び出す。

クラスメソッド、クラス変数の例

```
class Color4:

    RED  = "red message"
    def  createBlackColor( ):
        return  "#000000"



    def setColor4( self, r, b, g, a ):
         self.red = r; self.blue = b; self.green = g; self.alpha = a
```

クラスメソッドの種類

- スタティックメソッド
  - @staticmethodデコレータで始まる（つけなくてもOK）
  - 例：

```
class Sample:
@staticmethod
def zeroValue(): return 0
```

- クラスメソッド（狭義のPythonのクラスメソッド）
  - @classmethodデコレータで始まる
  - 仮引数の最初が、クラス自身を表すオブジェクトclsになっている
  - 例:

```
class Sample:
a : int   # class variable
@classmethod
def initialize( cls, param):
  cls.a = 10
```

プライベートメンバの指定

- クラス内の属性やメソッドをクラス外から見えなくすることができる。情報の隠蔽化・カプセル化（encapsulation)に用いられる
- 属性の変数、あるいはメソッド名をアンダーバーで始めたものを定義すると、外部から見えなくなる
  - _（アンダーバー１つで始まる）…内部で使うことを意味するが、外部からも参照可能
    - 例：**def** _innerMethod(self, param): **pass**
  - __（アンダーバー２つで始まる）…そのままでは外部からは参照できない。言語の処理系では、_<クラス名>__<名前＞に置き換えられているので、それを用いれば参照可能
    - 例：**def** __hiddenMethod(self, param): **pass**

property関数

- オブジェクトで使う変数を隠蔽しておいて、その値を参照する関数（getter）、設定する関数（setter）、削除する関数（deleter）を指定することができる
- 例：

```
class C:
    def __init__(self):  self._x = None
    def getx(self):   return self._x
    def setx(self, value):  self._x = value
    def delx(self):   del self._x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

- c が C のインスタンスならば、c.x は getter を呼び出し、c.x = value は setter を、del c.x は deleter を呼び出す。
- 最後のパラメータは、doc属性のためのパラメータ

propertyデコレータ

- property組込み関数をデコレータとして使えば、読み出し専用のプロパティを作れる
- 例：

```
class Parrot:
    def __init__(self):  self._voltage = 100000


    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

- @property デコレータは voltage() を同じ名前のまま 読み出し専用属性の "getter" にし、voltage のドキュメント文字列を "Get the current voltage." に設定する。

propertyデコレータ続き

- property オブジェクトは getter, setter, deleter メソッドを持っている。これらのメソッドをデコレータとして使うと、対応するアクセサ関数がデコレートされた関数に設定された、 property のコピーを作成できる。
- 例:

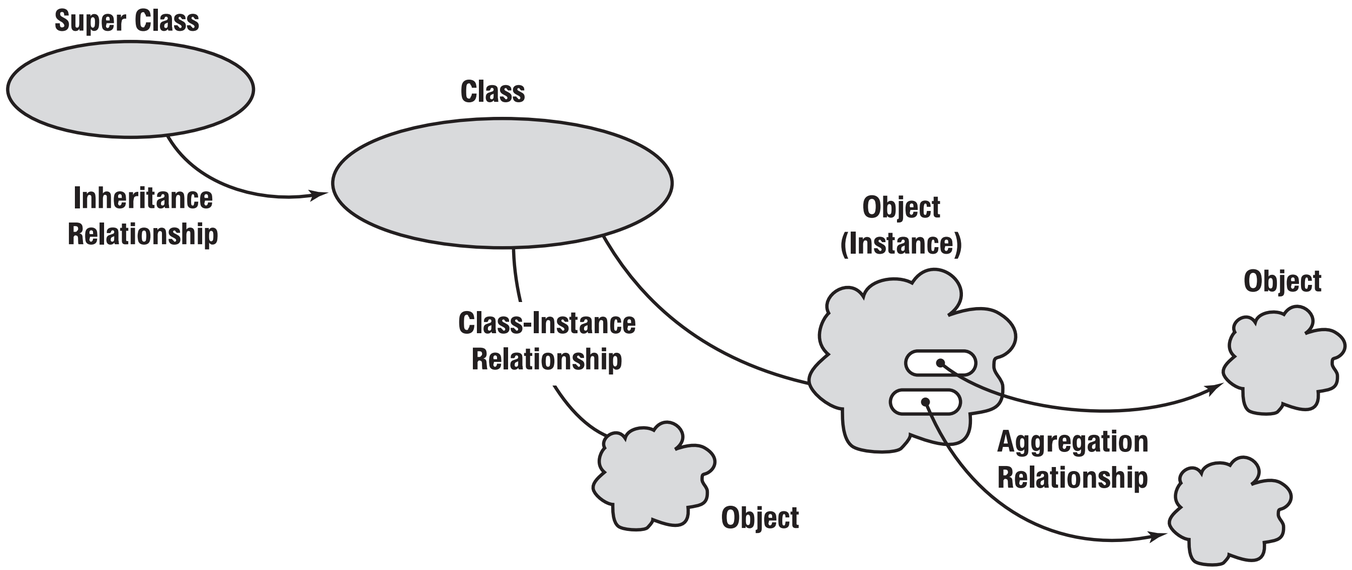
```
class C:
    def __init__(self):  self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):  self._x = value

    @x.deleter
    def x(self):  del self._x
```

継承とクラス

- スーパークラスの変数（クラス変数、インスタンス変数）は、サブクラスから利用ができる
- スーパークラスのメソッド（クラスメソッド、インスタンスメソッド）も、サブクラスから利用ができる
- メソッドについては、同じ名前のメソッドを定義することにより、サブクラスで上書き（overwrite）することが可能になる。

継承関係とクラス–インスタンス関係

- 継承関係図



継承のあるクラスの定義

- 書式：
  - **class** クラス名（スーパークラスの名前）:
  - 　　クラスの定義
- 例：
  - **class** Rabbit(Animal):
  - 　　**pass**

super() 組込み関数

- super()とすると、スーパークラスの変数（属性）やメソッドを利用することができる
- 例：
  - super().method(arg)
    - # スーパークラスのmethodを引数argで呼び出す
    - インスタンスメソッドからの呼出しの場合は、selfが自動でスーパークラスのメソッドに与えられる
- super(クラス名, オブジェクトまたはクラス名)で指定しても呼び出すことができる
- 例：
  - super(Rabbit, self).method(arg)
    - # Rabbitのスーパークラスのインスタンスメソッドmethodを引数argで呼び出す
  - super(Rabbit, Rabbit).clsmethod(arg)
    - # Rabbitのスーパークラスのクラスメソッドclsmethodを引数argで呼び出す

継承を利用した例

- Rabbitクラスは、Animalクラスを継承している

```
class Animal:
    name = "Animal "
    def jump( self ):  print( "Jump: " + Animal.name )
    def shout( self ):  print( "WOW:  " + Animal.name )

class Rabbit( Animal ):
    name = "Rabbit"
    def jump( self ):  print( "Jump: " + Rabbit.name )  # overwrite
    def superjump( self ):  super().jump( )
```

- 呼出し結果は以下のようになる

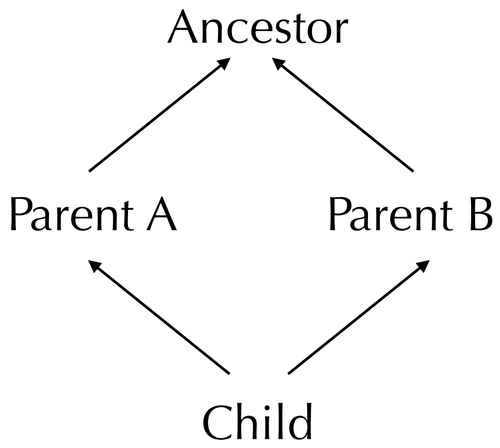
```
Animal().jump()     # Jump: Animalと表示される
r = Rabbit()        # rは、Rabbitクラスのオブジェクト
r.jump()            # Jump: Rabbitと表示される
r.shout()           # WOW: Animalと表示される
r.superjump()       # Jump: Animalと表示される
```

継承と多重継承

- クラス定義の際に、スーパークラスを指定する
- 書式： **class** クラス名(スーパークラス名):
  - 例： **class** Elephant(Animal): **pass**
- スーパークラスは、複数指定することができる（多重継承：multiple inheritance と呼ばれる）
  - 例：**class** SA(Student, Assistant): **pass**
- 多重継承の場合、指定されたスーパクラス名の順番に従って、スーパクラスが優先される

多重継承時の初期化・優先順位

- 右図のような形で多重継承されていた場合、初期化の呼出しは
  Self→A→B→Anchester→B→A→Selfという形で起こる



- プログラム例：

```
class Ancestor:
    def __init__(self):  print( "Ancestor initialized.")

class ParentA( Ancestor ):
    def __init__(self):
        print( "Parent A called.")
        super( ParentA, self ).__init__()
        print( "Parent A initialized.")

class ParentB( Ancestor ):
    def __init__(self):
        print( "Parent B called.")
        super( ParentB, self ).__init__()
        print( "Parent B initialized.")

class Child( ParentA, ParentB ):
    def __init__(self):
        print( "Child called.")
        super( Child, self ).__init__()
        print( "Child initialized.")

me = Child()
```

列挙型

- **from** enum **import** Enum

- **class** 列挙型のクラス名(Enum):
- 　　変数 = 値

- 例：

```
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

抽象クラス

- 具体的な記述を持たないスーパークラス
- サブクラスは、メソッドを上書きする必要がある
- Pythonでは、abcモジュールで定義されているABCクラスを継承する、あるいはABCMetaクラスをメタクラスとして指定すると、抽象クラスになる。
- 例：

```
from abc import ABC
class MyABC(ABC):
    pass

from abc import ABCMeta
class MyABC( metaclass=ABCMeta):
    pass
```

抽象クラス使用のメソッド

- @abstractmethodデコレータを使って、上書きされる必要のあるメソッドを外形だけを定義する
- 抽象クラスを継承した（実体のある）クラスでは、外形だけ定義されたメソッドの実体を定義する必要がある
- 例：

```
from abc import ABCMeta
class Animal( metaclass=ABCMeta):
    @abstractmethod
    def eat( ):
        pass

class Rabbit( Animal ):
    def eat(self):
        print( "Have a lettuce" )
```

モジュールとしての定義

- 1つの.pyファイルに書かれたプログラムは、モジュールと呼ばれる
- 他のPythonのプログラムから、該当のファイルの関数などを呼び出すときは、**import**文を利用して参照することができる。「ファイル名.py」のファイル名の部分がモジュール名として利用される
- 例： matrix.py → **import** matrix などを利用して、中の関数を利用可能
- __name__という内部変数（特殊属性）が、"__main__"という文字列になっていれば、そのモジュールから実行が開始されていることがわかる
- 例： そのモジュールから実行するときだけ、値の入力を要求して、main関数を実行させる

```
if __name__ == "__main__": # このモジュールから実行されたかどうか
    value = input( "値を入力して下さい" )
    main( value ) # そのモジュールから実行されたときだけ呼ぶ
```

  - **import**などで他のモジュールから呼ばれた場合は、この部分は実行されない

モジュールの検索

- **import**文で指定されたモジュール（ファイル）がどこにあるかを検索する順番が以下に規定されている

  1.カレントフォルダ

  2.環境変数PYTHONPATHに設定されているフォルダ

  3.標準ライブラリモジュールのフォルダ

  - 例えば、Mac OS XでPython 3.9の場合は、
    /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/
  - WindowsでPython 3.9のの場合は、
    C:\Users\ユーザ名\AppData\Local\Programs\Python\Python39\Lib\

  4.サードパーティのライブラリのフォルダ

  - 例えば、Mac OS XでPython 3.9の場合は、
    /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages/
  - WindowsでPython 3.9の場合は
    C:\Users\ユーザ名\AppData\Local\Programs\Python\Python39\Lib\site-packages\

- 標準ライブラリのモジュールと同じ名前でファイルをカレントフォルダに作った場合、そちらの方が優先されるので、注意する必要がある。例：math.pyなど

パッケージの利用

- 複数のモジュール（.pyあるいは.pyc）ファイルを1つのフォルダにまとめておき、そのフォルダをパッケージとして利用することができる
- **import**文では、「パッケージ名.モジュール名」などで参照することができる。
  - 例：
    - **from** packageA **import** moduleA
    - **import** packageA.moduleA
    - **from** packageA.moduleA **import** *
- そのフォルダに、「__init__.py」という名前のモジュールがあれば、そのパッケージをimportした際に、最初にこのモジュールが実行されるので、このモジュールに初期化用のプログラムを記述しておく
- パッケージについてもフォルダの検索順序は、モジュールと同様になる