Lightweight Language Saturday

Objects in Python

Ransui Iso (ransui@alpa.it.aoyama.ac.jp)

Python Japan Users Group / Aoyama Gakuin University

Pythonと名前空間

Python上のオブジェクト理解への近道

Pythonの名前空間

Global	プロセス全体を示す大域空間
Module	importによって作成される名前空間
Class	クラスの名前空間
Instance	各インスタンスの持つ名前空間
Local	関数やブロック内部の名前空間

Pythonでは、これらの名前空間は全てオブジェクトとして表現され、処理される

この図はよくある階層構造を表現したもので、このパターンに限定されるわけではない。Moduleの下位にModuleがくる場合もあるし、Localの下位にModuleがくる場合もある。

モジュールの名前空間

ModuleTest.py

```
import sys

ModuleVar = “Hello World”

def greeting():
  sys.stdout.write(ModuleVar + “¥n”)
```

ファイルのトップレベルに書かれた定義はModuleの名前空間に属する

Interactive Execution

```
>>> import ModuleTest
>>> ModuleTest.greeting()
Hello World
>>> type(ModuleTest)
<type 'module'>
>>> dir(ModuleTest)
['ModuleVar', '__builtins__', '__doc__', '__file__',
'__name__', 'greeting', 'sys']
>>> dir(moduleTest.sys)
[‘__displayhook__’, ‘__doc__’, ‘__excepthook__’,
‘__name__’, ‘__stderr__’, ‘__stdin__’, … 途中省略 … ,
'__stdout__','''stdout', 'version', 'version_info',
'warnoptions']
```

import文によってモジュールが読み込まれ名前空間を形成する。

組み込み関数 dir() で空間内の名前の一覧を得ることが出来る。左の例ではModuleTestの下位にモジュール sys の名前空間が形成されていることが分かる。

from 〜 import と名前空間

ModuleTest.py

```
from sys import stdout

ModuleVar = “Hello World”

def greeting():
  stdout.write(ModuleVar + “¥n”)
```

モジュール sys から stdoutのみをこのモジュールの名前空間に取り込む

Interactive Execution

```
>>> from moduleTest import greeting
>>> greeting()
Hello World
>>> import sys
>>> sys.modules['moduleTest']
<module 'moduleTest' from 'moduleTest.pyc'>
>>> dir(sys.modules['moduleTest'])
['ModuleVar', '__builtins__', '__doc__', '__file__',
'__name__', 'greeting', 'stdout']
```

from 〜 import文によって greeting() 関数のみを取り込んでいるように見える

実際はモジュール全体が読み込まれた上で、greeting 関数のみが現在の名前空間に格納されていることが分かる

クラスの定義とインスタンスの作成

Person.py

```
class Person:
  def __init__(self, name=None, age=None):
    self.name = name
    self.age = age

  def showMe(self):
    print “Name : %s” % self.name
    print “Age : %d” % self.age
```

`__init__` コンストラクタ
`self` インスタンスを指す

クラスインスタンスを作成する際に new のような演算子は必要ない

メンバは基本的に public として扱われる

Interactive Execution

```
>>> import Person
>>> foo = Person.Person(name="foo", age=30)
>>> foo.showMe()
Name : foo
Age : 30
>>> foo.name
'foo'
>>> foo.age
30
```

クラスの定義とインスタンスの作成

Interactive Execution

```
>>> import Person
>>> foo = Person.Person(name="foo", age=30)
>>> bar = Person.Person(name="bar", age=20)
>>> foo.showMe()
Name : foo
Age : 30
>>> bar.showMe()
Name : bar
Age : 20
>>> bar.age = 18
>>> foo.age
30
>>> bar.age
18
```

クラスインスタンスは独立した名前空間を持っているので、当然の事ながらあるインスタンスに加えた変更は、他のインスタンスには影響しない

Classのメンバ

Person.py

```
class Person:
  classVar = 1
  def __init__(self, name=None, age=None):
    self.name = name
    self.age = age

  def showMe(self):
    print “Name : %s” % self.name
    print “Age : %d” % self.age

  def showClassVar(self):
    print Person.classVar
```

classVar は Person クラスの名前空間に属するものとして定義される

変数を参照する場合は、どの名前空間に属しているかを明示する

Interactive Execution

```
>>> import Person
>>> foo = Person.Person(name="foo", age=30)
>>> foo.showClassVar()
1
>>> Person.Person.classVar = 100
>>> foo.showClassVar()
100
```

クラスの継承

Employee.py

```
from Person import *

class Employee(Person):
    def __init__(self, name=None, age=None, salary=None):
        Person.__init__(self, name, age)
        self.salary = salary

    def showMe(self):
        Person.showMe(self)
        print "Salary : %d" % self.salary
```

Personを継承したEmployeeの定義は左のようになる

① コンストラクタを定義する場合は、Super Classのコンストラクタを明示的に呼び出す
② Super Classのメソッドを呼ぶ場合は、クラス名.メソッド名 として呼び出す
③ メソッド呼び出し時には自分自身を与える

Instanceは独自の名前空間を持つ

Interactive Execution

```
>>> from Employee import Employee
>>> foo = Employee(name="foo", age=30, salary=420000)
>>> foo.showMe()
Name : foo
Age : 30
Salary : 420000
>>> foo.salary = 500000
>>> foo.showMe()
Name : foo
Age : 30
Salary : 500000
>>> foo.title = "Sinior Developer"
>>> dir(foo)
['__doc__', '__init__', '__module__', 'age', 'name',
'salary', 'showMe', 'title']
>>> foo.__class__
<class Employee.Employee at 0x1c6ab0>
>>> dir(foo.__class__)
['__doc__', '__init__', '__module__', 'showMe']
>>> foo
<Employee.Employee instance at 0x1cd788>
```

Instanceは、独自の名前空間を持っており、メンバ名はこの名前空間から解決される

代入によって作成された新たな束縛は、インスタンスの名前空間に記録され、クラスの名前空間に変化は発生しない

定義時の名前空間と実行時の名前空間

Interactive Execution / Python1.5

```
>>> def make_adder(base):
...   def adder(x):
...     return base + x
...   return adder
...
>>> add5 = make_adder(5)
>>> add5(6)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in adder
NameError: base
```

adder は make_adder の名前空間で定義されているローカル変数 base を参照できるように見えるが、実行時の名前空間では base が存在しないためエラーとなる。

Python2.2から標準となったNested Scopeでは、定義時の名前空間の情報を保持するようになったので、正しく実行できる

Interactive Execution / Python2.2

```
>>> def make_adder(base):
...   def adder(x):
...     return base + x
...   return adder
...
>>> add5 = make_adder(5)
>>> add5(6)
11
```

名前空間のまとめ

- Module, Class, Instanceはそれぞれ独自の名前空間を持つ
- 名前空間はそれぞれ独自に操作可能
  - 実行時にメンバの追加や、メソッドの入れ替えなどが可能
- 実行環境は名前空間に密接に関連している
  - 動的に名前空間を操作でき、便利な反面、解析困難なバグを生み出すことも
- import文は、名前空間の取り込みを行う
  - from 〜 import で名前を部分的に選択可能
  - ただし、その場合もモジュール全体が読み込まれる

Dynamic Object Environment

全てがオブジェクト・全てが操作可能

Pythonでは全てがObject

- Python2.2より組み込み型はClassと統合された

Interactive Execution

```
>>> type(1)
<type 'int'>
>>> import types
>>> type(1) is types.IntType
True
>>> types.IntType.__base__
<type 'object'>
>>> types.IntType.__base__.__name__
'object'
>>> import Person
>>> type(Person)
<type 'module'>
>>> type(Person).__base__
<type 'object'>
```

__base__
Super Classへの参照

__name__
クラス名への参照

全てのオブジェクトは、object のサブクラスのように振舞うようになっている

→ 組み込み型を継承したクラスを作成できるようになった

MetaDataと内部構造の参照

名前	参照先
`__class__`	オブジェクトのクラス
`__base__`	オブジェクトのスーパークラス
`__dict__`	オブジェクトのローカル名前空間
`__module__`	オブジェクトの属するモジュール

Module, Class, Instance などのMetaDataを参照可能

名前	参照先
`func_code`	コードオブジェクト
`func_globals`	関数・メソッドが参照する名前空間のリスト

関数・メソッドは Function Objectとして扱われ、実際のコード、定義時の名前空間などの情報にアクセス可能

名前	参照先
`co_stacksize`	実行時スタックのサイズ
`co_code`	PythonVM の ByteCode

Code Objectからは、実行されるByteCodeにもアクセスできる

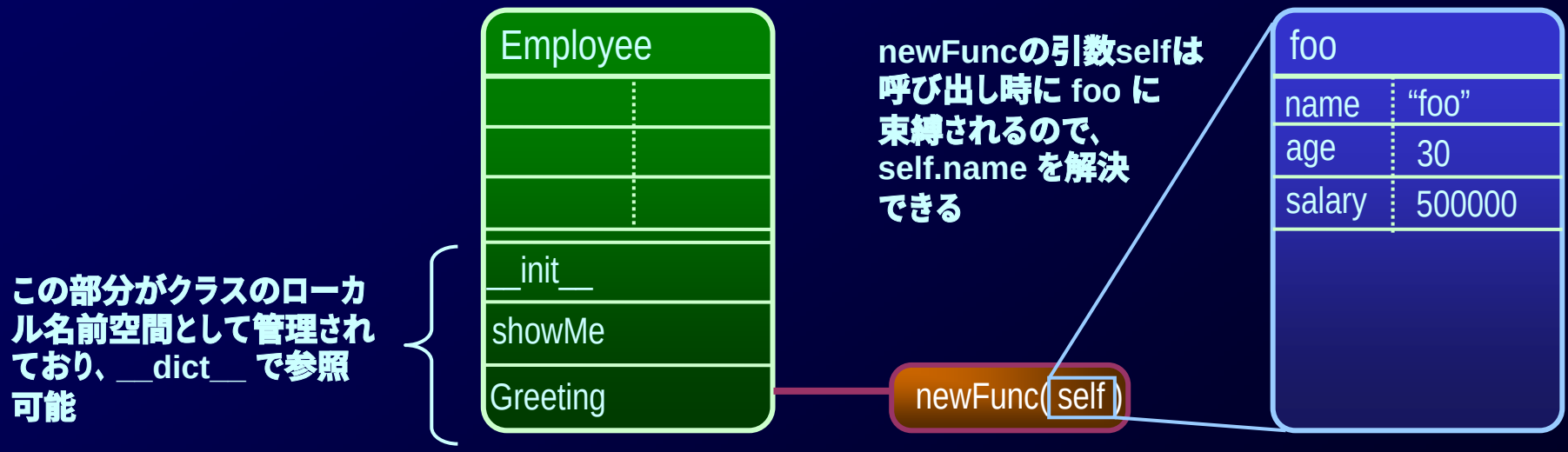
上に挙げた以外にも、多くのMetaDataや内部情報にアクセスできる

MetaDataを使った動的なメソッド追加の例

Interactive Execution

```
>>> from Employee import *
>>> foo = Employee(name="foo", age=30, salary=500000)
>>> def newFunc(self):
...     print "Hello " + self.name
...
>>> Employee.__dict__["Greeting"] = newFunc
>>> foo.Greeting()
Hello foo
```

__dict__
ローカル名前空間への参照



Function Objectの動的な入れ替えと実行時関数定義

Interactive Execution

```
>>> def func():
...   pass
...
>>> def func2(name):
...   "Hello " + name
...
>>> func()
>>> func.func_code = func2.func_code
>>> func("World")
Hello World
>>> exec("""
... def dynamicDefinition(msg):
...   print msg
... """)
>>> dynamicDefinition
<function dynamicDefinition at 0x1c9c30>
>>> dynamicDefinition("Hello World")
Hello World
```

関数はFunction Objectとして、通常のオブジェクトとなんら変わりなく操作できる

func_code
関数に束縛されている関数本体への参照

exec()
現在の環境下で文字列を評価する

これらをうまく組み合わせると、実行時にコードを生成し、自分自身を書き換えるようなプログラムが簡単に作成できる

Dynamic Object Environmentとしての Python

- LISP並みの動的環境
  - しかも構文は親しみやすい形式！
  - S式の柔軟性にはかなわないが、少し工夫すれば同等な処理は簡単に実現可能
- 動的なオブジェクト環境をうまく使いこなそう
  - Zopeなどはその最たる例 (全てが実行時に解釈される)
  - 永続化と組み合わせれば動的に作成したオブジェクトも保存できる
- こんなにManiacなことをしなくても十分に便利
  - 素直に書けば素直に動く
  - Moduleと関数だけを使えば、Objectを意識する必要は最小限に抑えられる

Python の Object システム探検ガイド

- **名前空間を調べる**
  - `type()` 関数　　オブジェクトの型なのかが分かる
  - `dir()` 関数　　オブジェクトのローカル名前空間のエントリが分かる
  - `__dict__`　　ローカル名前空間そのもの
  - `__base__`　　スーパークラスが分かる
  - `sys.modules`　　現在読み込まれている全てのモジュールの一覧が参照できる

- **組み込みドキュメントを利用する**
  - `help()` 関数　　オブジェクト内のドキュメントを表示する
    Python2.1以前の場合は `print` ***`object`***`.__doc__`

Happy Hacking !!

Thank you for listening.