Sistemas de Objetos de Python

Paulo J. S. Silva

baseado no Capítulo 5 de Python in a Nutshell

de Alex Martelli

Introdução

- Python é orietado a objetos
- Mas permite programação procedural
- Por outro lado: **tudo em Python são objetos**
- Em Python temos acesso a todo o mecanismo que implemnta OO.
- Há dois modelos de objetos: o clássico e o novo (que “melhora” o clássico)

Modelo Clássico

- Uma classe pode ser chamada para criar um outro objeto "à sua imagem", a instância. Instâncias sabem a classe pertecem.
- Classes possuem atributos de nomes arbitrários que podem ser lidos e modificados
- Os valores desses atributos pode ser dados ou funções (métodos de classe)
- Há **métodos especiais** que são chamados quando várias operações ocorrem sobre os diversos objetos

- Classes podem herdar de outras classes **delegando a procura por atributos**
- Instâncias por sua vez, são objetos com atributos arbitrários que podem ser acessados e modificados.
- Instâncias podems delegar a busca de atributos para a classes a que pertecem.
- Classes também são objetos e podem ser usadas em qualquer lugar.

Class Statement

```
class nome_da_classe[(classes_base)]:
    statement(s)
```

`nome_da_classe` é um identificador que vai referenciar o objeto class que será criado.

Use `issubclass(C1,C2)` para verificar a relação de herança.

Objetos classe possuem atributos que serão partilhados por todas as instâncias.

Os atributos da classe são definitos (usualmente) no seu corpo.

Atributos pré-definidos

- `__name__`: nome da classe
- `__bases__`: lista de classes base
- `__dict__`: dicionário para guardar todos os outros atributos.

`C.x = 23` é equivalente a `C.__dict__['x'] = 23`.

A referência a atributos tem definição mais rica e será vista depois.

Métodos e Instâncias

A funções definidas no corpo da classe são chamadas de métodos.

Métodos possuem um primeiro parâmetro obrigatório que representará a instância.
Normalmente esse parâmetro é chamado de `self`.

Ao chamar um objeto classe ele cria uma instância e em seguida, se existir, chama implicitamente o método `__init__` se ele existir. Os parâmetros têm que casar.

Obs: `__init__` não retorna o objeto, ele o recebe.

`__init__` tem o papel de construtor. Normalmente aí são inicializados os atributos de dados da instância.
Há funções especiais que controlam o acesso e a modificação de atributos.
Há dois atributos pré-definidos: `__class__` e __dict__.
Esses atributos pré-definidos podem ser modificados.

Referências a atributos

O operador ponto é usado pace ter acesso aos atributos.

Considere o exemplo ex1.py

Há os atributos especiais que sempre existem (mas podem ser modificados):

`__name__, __bases__, __class__, __dict__`

Todos os outros atributos estão em `__dict__`.

Caso o atributo não seja especial a busca por um atributo obedece as seguintes regras:

buscando `x.nome`

Se `x` é uma instância:

1. Se `name` é uma chave do dicionário pega o item repectivo;
2. Senão, delega o trabalho a classe (tenta pegar `x__class__.name`).

Se `x` é uma classe:

1. Se `name` é uma chave do dicionário pega o item repectivo;
2. Senão, delega o trabalho às classes base (usando `x__bases__.name`).

Se o atributo não for encontrado:

1. Se o `x` define o método especial `__getattr__` existir ele é chamado
2. Senão, lança-se uma exceção do tipo `AttributeError`.

O mesmo vale para métodos.

Objetos em python são apenas dicionários com esteróides!

exemplo:

```
print x.e, x.d, x.c, x.b, x.a
```

Métodos vinculados e não-vinculados

Há um passo extra quando o atributo encontrado é um método:

O método não é retornado mas é empacotado em um objeto de método vinculado ou não-vinculado.

O que raios é isso?

Métodos vinculados estão associados a instâncias e não vinculados não.

Isso explica porque podemos omitir o `self` em `x.h()`.

Exemplos:

```
print x.h, x.g, x.f, C.h, C.g, C.f
```

Caso se queira chamar um método não vinculado, é preciso deixar a instância explícita: `C.h(x)`.

Métodos não vinculados possuem três atributos especiais: `im_class`, `im_func` e `im_self == None`.

Eles são somente para leitura.

O primeiro parâmetro deve sempre ser uma instância de `im_class`.

Métodos vinculados

Quando o método é obtido a partir da instância ele vem vinculado através de `im_self`.

Nesse caso o primeiro parâmetro é passado implicitamente.

```
x.name(y) ==
x.__class__.__dict__['name'](x, y).
```

Os métodos vinculados só são gerados se forem encotrados na classe de `x`, nada especial ocorre se eles estiverem em `x.__dict__`.

Durante a sua execução não há nenhuma ligação implícita entre o método, a instância e/ou a classe!

Métodos vinculados são objetos de primeira ordem e podem servir como uma alternativa para um fechamento.

Herança

Não há nada especial na herança: está tudo implícito na busca pelos atributos.

Sobre-escrita de métodos ou atributos de classe ocorrem naturalmente.

Não há sintaxe especial para chamar o método da classe pai (algo como super), chama-se explicitamente o método não vinculado (isso é **feio**)!

O construtor da classe pai deve ser chamado explicitamente (isso também é feio!).

Modelo Novo

Aparece para adicionar um monte de pequenas alterações:

- Classe base global (até para tipos primitivos): `object`
- Métodos estáticos e métodos de classe
- Propriedades (get/set, valores computados em tempo de execução)
- Controle da alocação de um objeto

Modelo Novo (cont.)

- Funções especiais de controle do acesso a atributos
- Nova ordem para lidar com herança múltipla
- Função especial `super` para delegação (menos feia, mas feia)
- **Metaclasses**: vamos ter uma aula só sobre isso.

`object`

Para criar uma classe no modelo novo ela deve herdar (mesmo que indiretamente) de `object`.

Seu uso deve ser preferido: orgamniza certos aspectos do modelo e nos dá novos brinquedos.

Os tipos pré-definidos herdam de object e podem ser estendidos.

É `object` que define vários dos métodos especiais novos:

```
__new__, __init__,
__delattr__, __getattribute__, __setattr__,
__hash__, __repr__, __str__
```

Métodos estáticos e de classe

Métodos estáticos não possuem a semântica especial dos métodos: não há self, métodos vinculados, etc.

Na verdade isso é desnecessário em Python, pode-se usar funções comuns (de módulos).

Sintaxe engraçada (usa padrão decorador):

```
class UmaClasse(object):
  def metEstatico():
    print 'Meu método estático'
  metEstatico = staticmethod(metEstatico)
instancia = UmaClasse()
UmaClasse.metEstatico()
instacia.metEstatico()
```

Métodos de classe

Métodos ligados à classe e não a uma instância.

Primeiro parâmetro normalmente chamado de `cls`.

```
class Base(object):
  def metClasse(cls):
    print 'Método de classse', cls.__name__
  metClasse = classmethod(metClasse)
class Derivada(Base): pass
instanciaB = Base()
instanciaD = Derivada()
Base.metClasse()
instanciaB.metClasse()
instanciaD.metClasse()
```

`__init__` *e* `__new__`

- `__init__`: Caso não seja sobre-escrito, passa a ignorar parâmetros extras. Isso é perigoso: é bom sempre sorescrever para evitar problemas.
- `__new__`: Quando se chama a classe `C( *args, **kwds)` para criar uma instância esse método é chamado através de `C.__new__(C, *args, **kwds)`. O seu retorno é considerado o objeto alocado. `__init__` normalmente o segue.
- Pode ser usado como uma “função fábrica”.
- `object.__new__` cria uma instância de objeto e ignora parâmetros extra.

Propriedades

Propriedade parecem com atributos comuns mas são acessadas por métodos definidos pelo usuário na sua criação.

Você pode controlar os vários aspectos do acesso: leitura, escrita e “destruição”.

O que não estiver defindo é proibido.

Bom uso: introduzir métodos get e set **quando necessários** sem mudar o código cliente.

__getattribute__

Essa é a função que implementa a busca de um atributo de uma instância.

É implementada em `object`.

Pode ser sobre-escrita:

```
class ListaSemAppend(list):
  def __getattribute__(self, name):
    if name == 'append':
      raise AttributeError, name
    return list.__getattribute__(self,
                                  name)
```

A alteração de atributos e sua “destruição” chamam `__setattr__` e `__delattr__` respectivamente.

Herança múltipla

Herança múltipla não combina com tipos básicos.

No modelo clássico a busca é feita em profundidade (e da esquerda para direita). Isso pode esconder métodos sobre-escritos.

No modelo novo a busca é feita em largura para evitar o problema acima.
Atributo especial `__mro__` possui a ordem de resolução.

`super`

Com herança múltipla a solução dada para delegar para a classe base parte de uma tarefa pode ficar confusa.

O mesmo método pode ser chamado mais de uma vez!

Se você quer usar isso use `super`.

A cada chamada, super elimina um elemento de `__mro__`;

Outros métodos especiais

Há vários métodos especiais que podem ser usados para controlar o sistema de objetos, o comportamento de operadores ou funções especiais, etc.

Dois métodos úteis para reflexão:

`__call__` Usado quanto um objeto é “chamado”, seja classe, método ou função.

`__del__` Chamado antes que o coletor de lixo destrua o objeto. Especialmente interessante em extensões.

Exemplos - Singleton

O padrão singleton é um ótimo exemplo, pois permite múltiplas soluções.

1. Usando `__getattr__` e “classe interna”;
2. Criando um único objeto de “verdade”;
3. Pondo para funcionar com herança;
4. Transformando em Singleton em tempo de execucao, usando decoradores.