Python: Orientação a Objetos

Prof. Dr. Dilermando Piva Jr.

Definindo classes

... Noções de Orientação a Objetos

… E tudo são objetos...

- Tudo em Python é no fim um objeto!
- “hello”.upper()
- list3.append(‘a’)
- dict2.keys()
- Eles parecem como chamadas de método em Java ou C++ !
- Novos objetos (classes) podem ser facilmente adicionados aos tipos de dados já existentes em Python!
- De fato, programar em Python é normalmente feito de forma orientada a objetos!

... Definindo uma classe...

- **Conceitos necessários**
  - Classe: Molde
  - Objeto: Agente ativo na programação
  - Método: Capacidades de ação do agente ativo
  - Atributo: Características do agente ativo
- Exemplo:
  - Classe: Humorista
  - Objeto: Tiririca
  - Método: Contar piadas, Imitar Pessoas
  - Atributo: Baixo, 44 anos

- Python não usa o conceito de definição de interfaces como em outras linguagens. Basta você definir a clase e utilizá-la!

... Métodos em classes...

- Definir um método em uma classe , basta incluir a definição da função seguindo o escopo de bloco da classe.

- Em todos métodos associados à instância definido dentro de uma classe devem ter o argumento *self* definido como primeiro argumento.

- Há geralmente um método especial *__init__* definido na maioria das classes.

Definição de uma classe

Automovel
+ placa : str
__init__(str) : None
get_placa() : str
dirigir(int) : None

```
class Automovel:

    def __init__(self, placa='XX-123'):
      self.placa = placa

    def get_placa(self):
      return self.placa

    def dirigir(self, velocidade):
      print 'Estou dirigindo a %d' \
            ' km/h' % velocidade
```

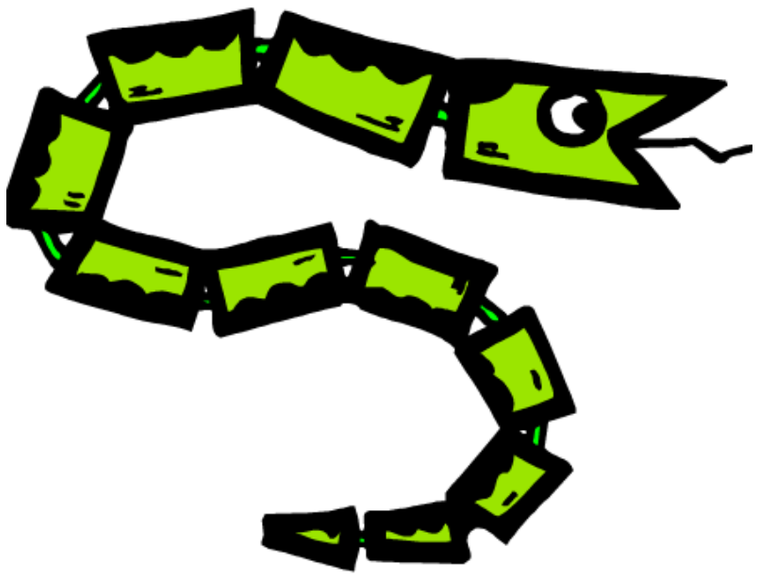
construtor

self

métodos

Orientação a objetos

Criando e Deletando instâncias



Instanciando Objetos

- Não há “new” como feito em Java!
  - a = student(“Sheldon”, 34) (** sem o operador new!)
- “__init__” serve como construtor de uma classe. Geralmente faz o trabalho de inicialização.
  - Não há limite para o número de argumentos passados para o método __init__ . Como em qualquer outra função, os argumentos podem ser definidos com valores default, tornando-os assim opcionais ao chamador

Instanciando Objetos

- ***self*** : O primeiro argumento de qualquer método é a referência para a própria instância da classe

- Em "__init__" *self* referencia o objeto criado recentemente, e em outros métodos, referencia a instância de qual o método foi invocado.

  - Similar ao *this* usado em Java ou C++

  - Porém Python usa mais *self* do que Java com *this*

__init__

```
>>> class Carro:
...     def __init__(self):
...         self._nrodas = 4
...     def set_nrodas(self, n):
...         self._nrodas = n
>>> gol = Carro()
>>> gol._nrodas
4
>>> gol.set_nrodas(10)
>>> gol._nrodas
10
```

self

- Não é necessário incluí-lo no método que faz a chamada do mesmo, apenas na definição!

- Python passa ele automaticamente.

```
a = Automovel()

print (a.get_placa())
```

Recapitulando….

- O statement *class* indica uma uma definição de uma nova classe
- A função *__init__(self, args...)* define o construtor da classe, sendo o primeiro argumento obrigatoriamente a própria instância da classe (*self*)
- Classes podem ter variáveis de classe que são compartilhadas entre todas as instâncias dessa mesma classe e variáveis de instância que são exclusivas a cada instância
- Instâncias podem receber atributos dinamicamente

oo_example.py

```
1   class Parent:
2       # Class variable
3       count = 0
4
5       # constructor
6       def __init__(self, name, age):
7           # instance variables
8           self.name = name
9           self.age = age
10          Parent.count += 1
11
12      # overriding str
13      def __str__(self):
14          return 'Name: {}, Age: {}'.format(self.name, self.age)
15
16
17  p1 = Parent('Joao', 10)
18  p2 = Parent('Maria', 11)
19  print(p1)
20  print(p2)
21  print(Parent.count)
22
```

Deletando instâncias

- Quando estiver finalizado com o objeto, você não precisa deletá-lo ou liberá-lo explicitamente.

- Python possui *garbage collection* de forma automática.

- Python irá automaticamente detectar quando todas as referências para um trecho de memória estiver não sendo mais referenciado. Automaticamente, a memória é liberada.

- Poucos leaks de mémoria, e não há métodos “destrutores” em Python!

Desvendando a classe...

Acesso de atributos e métodos

Acessibilidade

- Acesso de métodos e atributos
- Diretamente objeto.atributo ou por algum método objeto.getAtributo()

```
a = Automovel()

print (a.n_rodas)
```

Acessibilidade

- Atributos (class e ou instâncias)
  - **Privados**
    - *Atributos e métodos só podem ser acessados dentro da classe, usa-se “__” no ínicio do nome.*
  - **Protected**
    - *Apenas convenção e usa-se apenas um “_” no nome de métodos ou atributos*

```
>>> class Carro: #Classe
...     _nrodas = 4
...     __nparafusos = 3000
...
>>> gol = Carro() #Instância
>>> gol.__nparafusos #Error
```

Especial: property

```
class Fone(object):

    (...)

    def pegar_volume(self):
       return self.volume

    volume = property(pegar_volume,alterar_volume)
```

Especial: property

```
>>>fone = Fone(200)
>>>fone.pegarVolume()
100
>>>fone.volume
100
>>>fone.volume = - 50
0
>>>fone.volume = 200
>>>fone.volume
100
```

Atributos

Como declarar os membros de uma classe ?!

Atributos

- Exceto métodos, todos os demais dados dentro de uma classe são armazenados como atributos.

- **Atributos de instância**
  - Variáveis que pertencem a uma instância particular da classe
  - Cada instância tem o seu próprio valor para o atributo
  - Os mais freqüentes em classes
- **Atributos de classe**
  - Variáveis que pertencem à classe como um todo.
  - Todas as instâncias da classe compartilham o mesmo atributo (valor).
  - Conhecidos como “estáticos”em outras linguagems

Atributos

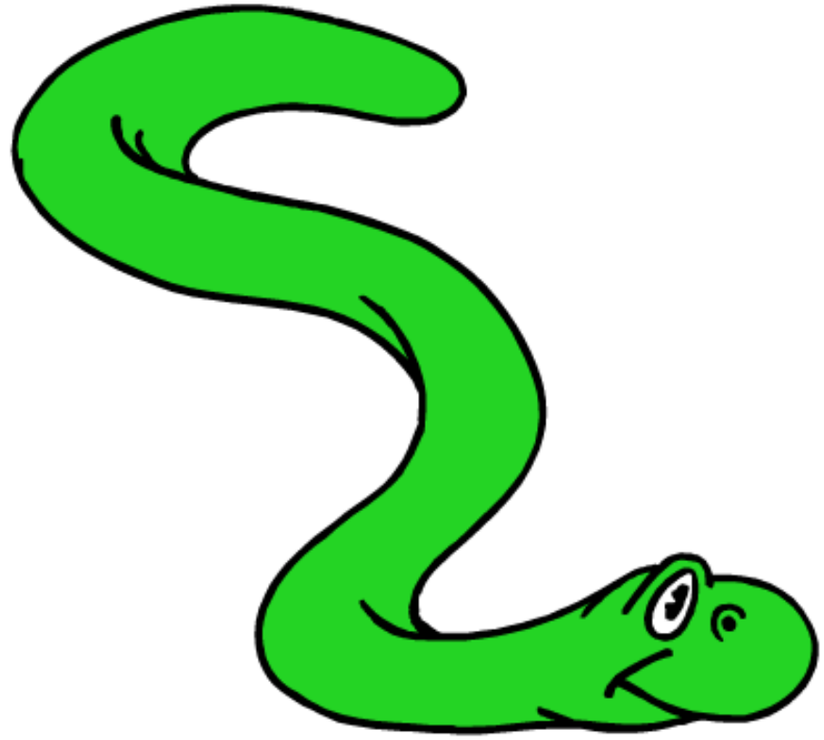
- Atributos de instância são criados e inicializados pelo método *__init__()*
  - Simplesmente atribuindo um valor a um rótulo
  - Dentro da classe, referir-se ao atributo usando **self**
    - Exemplo: ***self.full_name***

Atributos

- Atributos de classe são compartilhados (apenas uma cópia) por todas instâncias da classe.
  - Qualquer instância alterá-lo, o valor é alterado para todas instâncias.
  - Atributos de classe são definidas:
    - Dentro da definição de uma classe
    - Fora de quaisquer métodos da classe
  - Já que estes atributos são compartilhados por todas instâncias de uma classe, eles são acessados através de uma notação diferente:
    - self.__class__.name

Herança

Python **é** uma linguagem de programação...



Herança

- Uma classe pode **herdar** a definição de outra classe
  - Permite o uso ou a extensão de métodos e atributos previamente definidos por outra classe.
  - Nova classe: subclasse. Original: classe pai, ancestral ou superclasse
  - Para definir uma subclasse, coloque o nome da superclasse entre parênteses depois do nome da subclasse na primeira linha da definição.
    - Python não tem a palavra ‘extends’ como em Java
    - Múltipla herança é suportada

Herança

```
>>> class Veiculo:
...     def andar(self): print "andei"
...
>>> class Carro(Veiculo):
...     _nrodas = 4
...
>>> gol = Carro()
>>> gol.andar()
andei
```

Herança

- Herança é definida quando na definição da classe é indicada a classe da qual ela extenderá

oo_example.py

```
16      class Child(Parent):
17
18          def __init__(self, name, age, school_grade):
19              super().__init__(name, age)
20              self.school_grade = school_grade
21
22          def __str__(self):
23              return super().__str__() + ', School Grade: {}'.format(self.school_grade)
24
25      p = Parent('Joao', 50)
26      c = Child('Joaozinho', 10, 5)
27
```

Herança Múltipla

- Herança multipla funciona da mesma forma, inserindo-se outras classes que deverão ser extendidas

oo_example.py

```
16  class Walker:
17
18      def walk(self):
19          print('Walking...')
20
21      def run(self):
22          print('Running...')
23
24  class Child(Parent, Walker):
25
26      def __init__(self, name, age, school_grade):
27          super().__init__(name, age)
28          self.school_grade = school_grade
29
30      def __str__(self):
31          return super().__str__() + ', School Grade: {}'.format(self.school_grade)
32
```

Redefinindo métodos

- Você pode redefinir métodos declarados na superclasse

- O mesmo vale para o método __init__ .

- Geralmente você algo assim no método __init__ das subclasses:

  - parentClass.__init__(self,x,y)

onde parentClass é o nome da classe pai.

Redefinindo métodos

```
>>> class Veiculo:
...     def andar(self): print "andei"
...
>>> class Carro(Veiculo):
...     _nrodas = 4
...     def andar(self): print "andei de carro"
...
>>> gol = Carro()
>>> gol.andar()
andei de carro
```

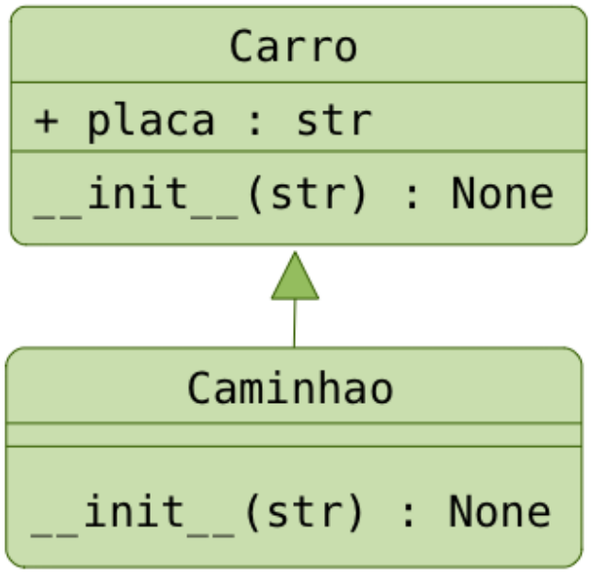
Redefinindo métodos

```
>>> class Veiculo:
...     def andar(self): print "andei"
...
>>> class Carro(Veiculo):
...     _nrodas = 4
...     def andar(self):
...         Veiculo.andar(self)
...
>>> gol = Carro()
>>> gol.andar()
andei
```

Herança

- Java

```
public class Caminhao extends Carro {
    public Caminhao(String placa) {
        super(placa);
    }
}
```



- Python

```
class Caminhao (Carro):
    def __init__(self, placa):
        Carro.__init__(self, placa)
```

Alguns métodos e atributos especiais nativos

I am special!

Membros nativos

- As classes contêm métodos e atributos especiais que são incluídos por Python mesmo se você não os defina explicitamente.

- A maioria destes métodos são invocados automaticamentos a partir de alguma ação ou evento por meio de operadores ou uso da classe.

- Algums atributos nativos definem informações que devem ser armazenadas para todas as classes.

- Todos os membros nativos tem 2 underscores ao redor dos nomes: __init__ , __doc__

Membros nativos

- Alguns métodos como por exemplo __repr__ existem para todas as classes e você pode sempre redefiní-las.

- A definição deste método especifica como tornar a instância de uma classe em uma string.

  - **print f** algumas vezes chama **f.__repr__()** para chamar a representação em string do objeto f

  - Se você digitar **f** e pressionar ENTER, então você também está chamando **__repr__** para informar ao display o que deve ser exibido ao usuário

Métodos nativos

- Você pode redefinir estes métodos também:
  - **__init__ :** O construtor da classe
  - **__cmp__:** Define como == funciona para a classe
  - **__len__ :** Define como len(obj) funciona
  - **__copy__ :** Define como copiar uma classe
  - Outros métodos nativos permitem você dar a classe o poder de usar notação [] como um array ou () como uma chamada de função.

Métodos nativos

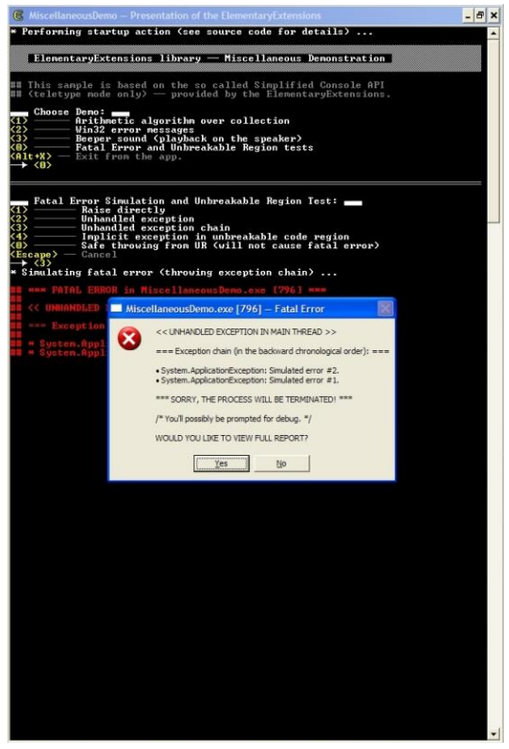
```
>>> class Carro:
...     def __init__(self, nr):
...         self.__nrodas = nr
...     def __add__(self, car):
...         return self.__nrodas + car.get_nr()
...     def __repr__(self):
...         return "Eu sou um carro de %d rodas!" % self._nrodas
...     def __cmp__(self, car):
...         return cmp(self.__nrodas, car.get_nr())
...     def get_nr(self):
...         return self.__nrodas
>>> a = Carro(4); b = Carro(6)
>>> a + b
10
>>> a > b
False
>>> print a
Eu sou um carro de 4 rodas!
```

Atributos especiais

- Estes atributos existem para todas as classes.
  - **__doc__ :** Armazena a documentação (string) para a classe.
  - **__class__:** Retorna a referência à classe de qualquer instância dela.
  - **__module__ :** Retorna a referência ao módulo que aquela classe em particular foi definida.
  - Outro método bem útil **dir(x)** retorna a lista de todos os métodos e atributos definidos pelo objeto x.

Tratamento de exceções

Fatal Error! E agora?!



Try/Except

- Python tenta executar o código de programa dentro do bloco inserido dentro do try.

- Caso algum erro aconteça, uma exceção é levantada!
  - Seu programa é interrompido por alguma falha em tempo de execução.

- except permite tratar as exceções levantadas. O programador define como lidar com estes erros inesperados!

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Try/Except

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

- Você pode capturar vários tipos de exceção!

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

raise

- Se você não desejar naquele bloco de código tratar a exceção, você pode optar por levantar a exceção para a chamada cujo o bloco foi chamado.
  - No final de contas, alguem tem que tratar exceção! Você está apenas adiando o inevitável! Importante quando você quer que outro desenvolvedor monte tratamento de exceções customizadas.

```
>>> raise Exception("Classe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Classe
>>> erro = Exception("Instancia")
>>> raise erro
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Instancia
>>> raise "String"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
String
```

Finally

- Tenta remediar a situação, última ação antes do estouro da exceção ou fim do try.
  - Sempre executado antes do fim de um comando try.

```
>>> try:
...     [1,2,3,4,5][10]
... except IndexError, e:
...     print e
... finally:
...     print "o remendo vem aqui!"
...
list index out of range
o remendo vem aqui!
>>>
```

Exemplo

```
import sys

try:
    f = open('meuarquivo.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[ 0]
    raise
```

Você pode criar suas exceções!

- Sua classe deve herdar da classe diretamente ou indiretamente da classe Exception
  - Métodos redefinidos: __init__ e __str__

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Documentação

- A documentação de módulos, classes, métodos, funções, tudo, tem que estar dentro do que está se documentando. Evite acentuação!
- Para gerar a documentação de um código basta usar o comando pydoc -w <nomeDoModulo>