Computação II (MAB 225)

Fabio Mascarenhas - 2015.1

http://www.dcc.ufrj.br/~fabiom/pythonoo

Sobrecarga de operadores

- Quase todos os operadores de Python são como `==` e `!=`: podemos definir métodos para *sobrecarregá-los*
- Operações aritméticas: `__add__` (+), `__sub__` (-), `__mul__` (*), `__div__` (/), ...
- Sequências: `__len__` (`len`), `__getitem__` e `__setitem__` (indexação), `__delitem__`(`del`), `__contains__` (`in`), ...
- Relações: `__lt__` (<), `__le__`, (<=), `__gt__` (>), `__ge__` (>=)
- Conversão para booleano: `__nonzero__`
- Chamada: `__call__`

Iteradores e `for-in`

- O comando `for <vars> in <seq>` de Python usa o método `__iter__(self)` para obter um objeto *iterador* para a sequência

- Um iterador implementa uma interface bem simples: o método `next(self)` retorna o próximo item da sequência, ou um erro se já acabaram os itens

```
class Intervalo(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __iter__(self):
        return IteradorIntervalo(self)
```

```
>>> i = Intervalo(2, 5)
>>> for x in i: print x

2
3
4
5
```

```
class IteradorIntervalo(object):
    def __init__(self, i):
        self.a = i.a
        self.b = i.b

    def next(self):
        if self.a > self.b:
            raise StopIteration()
        else:
            atual = self.a
            self.a = self.a + 1
            return atual
```

Herança vs. Composição

- Herança implica uma relação “é um” entre a subclasse e a superclasse, então ela deve ser evitada se essa relação seria espúria

- Nesse caso, podemos ter o mesmo reaproveitamento de código que temos na herança, com um pouco mais de burocracia, usando composição e delegação

- Cuidado com a literatura introdutória de OO, ela está cheia de exemplos espúrios do uso de herança, como “Círculo extende Elipse”

- Os princípios de projeto SOLID nem sempre podem ser seguidos, mas devem ser o ideal

Template Method

- Um padrão de programação comum em classes abstratas, onde métodos da classe abstrata delegam parte do seu comportamento para as subclasses através de métodos abstratos

- Os métodos de uma classe abstrata podem chamar métodos abstratos dessa classe sem problemas; como uma subclasse concreta precisa fornecer implementações para os métodos abstratos, eles “estarão lá” quando necessário

- O uso desse padrão é comum em *frameworks*: ao invés da aplicação implementar uma interface, ela estende uma classe abstrata e fornece os métodos que faltam

Tkinter

- Tkinter é uma biblioteca para criação de interfaces gráficas em Python

- As aplicações Tkinter usam uma estrutura de *componentes* parecida com a do Editor Gráfico: uma aplicação tem uma *janela*, onde adicionamos *controles* como botões, texto, caixas de desenho, sliders, caixas de entrada de texto...

- Os controles respondem a *eventos*: manipulação do mouse, cliques, teclas apertadas no teclado

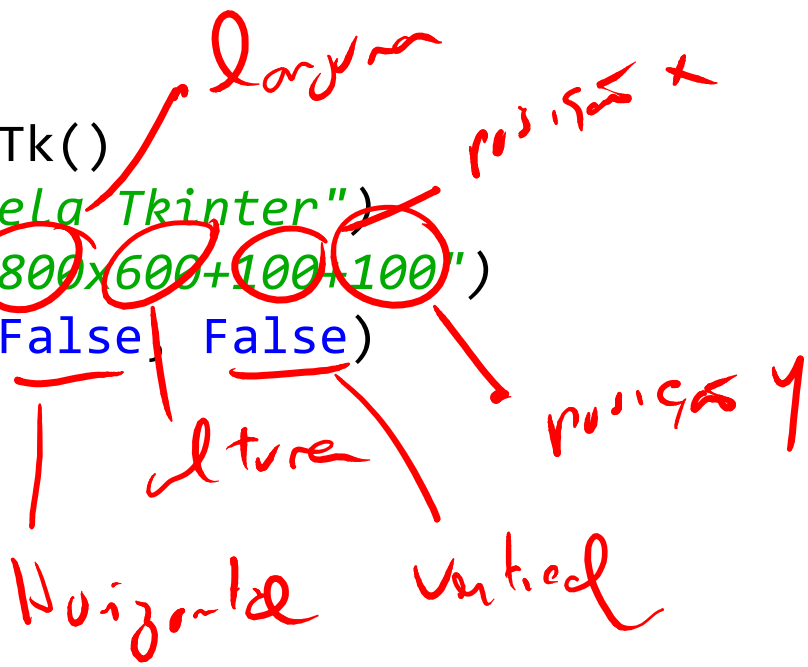
- Para usar o Tkinter precisamos primeiro importar ele: `import Tkinter`

Criando a janela do programa

- A janela do programa é uma instância da classe Tk

- Essa classe tem vários métodos, mas vamos usar quatro em particular: `title` diz o texto da barra de título da janela, `geometry` dá as dimensões e posição dela, `resizable` diz se a janela pode ser redimensionada pelo usuário ou não, e `mainloop` mostra a janela e dispara a aplicação

```
import Tkinter
raiz = Tkinter.Tk()
raiz.title("Janela Tkinter")
raiz.geometry("800x600+100+100")
raiz.resizable(False, False)
raiz.mainloop()
```



Encapsulando a janela em uma classe

- Vamos trabalhar sempre com janelas que não podem ser redimensionadas, então podemos simplificar a criação delas com uma classe:

```
class Janela(Tkinter.Tk):
    def __init__(self, titulo, larg, alt):
        Tkinter.Tk.__init__(self)
        self.title(titulo)
        x = (self.winfo_screenwidth()-larg)/2
        y = (self.winfo_screenheight()-alt)/2
        self.geometry("%dx%d+%d+%d" % (larg, alt, x, y))
        self.resizable(False, False)

raiz = Janela("Janela Tk", 800, 600)
raiz.mainloop()
```

Botões

- Botões são instâncias da classe Button; quando criamos um botão, dizemos a qual janela ele pertence (ao invés de criar um botão e adicionar ele na janela)

- Precisamos dizer qual o texto do botão, para isso atribuímos ao atributo “`text`” como se o botão fosse um dicionário

- Mas nada disso faz o botão aparecer: para isso precisamos do método place, passando um dicionário com os atributos x, y, width e height

```
bot = Tkinter.Button(raiz)
bot["text"] = "Ok"
pos = { "x": 50, "y": 100, "width": 150, "height": 80 }
bot.place(pos)
```

Uma classe para botões

- Novamente, podemos simplificar nosso programa com uma classe para botões com uma interface mais parecida com o que temos usado:

```
class Botao(Tkinter.Button):
    def __init__(self, janela, x, y, larg, alt, texto):
        Tkinter.Button.__init__(self, janela)
        self["text"] = texto
        self.place({ "x": x, "y": y,
                     "width": larg, "height": alt })
```

- E como fazer o botão responder aos cliques do mouse?

Comandos

- Assim como damos o texto do botão com o atributo “`text`”, dizemos qual comando executar quando o botão é clicado com o atributo “`command`”

- Esse atributo pode ser uma função ou método sem parâmetros

- Podemos adicionar ações à nossa classe Botao do mesmo jeito que fizemos com o Editor Gráfico:

```
class Botao(Tkinter.Button):
    def __init__(self, janela, x, y, larg, alt, texto, acao):
        Tkinter.Button.__init__(self, janela)
        self["text"] = texto
        self.place({ "x": x, "y": y,
                     "width": larg, "height": alt })
        self["command"] = self.executar
        self.acao = acao

    def executar(self):
        self.acao.executar()
```

Uma classe abstrata para botões

- Também podemos fazer a classe Botao ser abstrata, com a ação executada sendo um método abstrato:

```
class Botao(Tkinter.Button):
    def __init__(self, janela, x, y, larg, alt, texto):
        Tkinter.Button.__init__(self, janela)
        self["text"] = texto
        self.place({ "x": x, "y": y,
                     "width": larg, "height": alt })
        self["command"] = self.executar

    def executar(self):
        raise NotImplementedError()
```

Labels

- Instâncias da classe Tkinter.Label são controles que servem para mostrar texto
- Ele tem um atributo “`text`”, como um botão, e também é posicionado com `place`
- Se mudamos o valor de “`text`” enquanto o label está visível o texto que ele está mostrando muda
- Podemos usar o atributo “`anchor`” para controlar onde o texto fica em relação ao label: “`n`”, “`ne`”, “`e`”, “`se`”, “`s`”, “`sw`”, “`w`”, “`nw`”, ou “`center`”

Conectando labels ao modelo

- As aplicações que fizemos com a biblioteca `gui` redesenham sua interface 60 vezes por segundo

- Isso permite que elas sempre estejam retirando do modelo a informação mais recente

- Uma biblioteca de interface tradicional como `Tkinter` só redesenha a interface quando necessário, então o modelo precisa avisar a camada de visão de mudanças nele

- Podemos criar uma classe `Rotulo` para encapsular um label, e fazer essa classe ser um *observador*

A classe `Rotulo`

- A classe `Rotulo` recebe eventos do modelo através do seu método `mudou`:

```
class Rotulo(Tkinter.Label):
    def __init__(self, janela, x, y, larg, alt, texto, ancora):
        Tkinter.Label.__init__(self, janela)
        self["text"] = texto
        self["anchor"] = ancora
        self.place({ "x": x, "y": y, "width": larg, "height": alt })

    def mudou(self, texto):
        self["text"] = texto
```

- Como exemplo vamos fazer uma aplicação com um modelo bem simples, um botão que modifica o estado desse modelo, e um label que mostra esse estado