Computação II (MAB 225)

Fabio Mascarenhas - 2015.1

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

Classes abstratas

- Até agora, usamos *interfaces* toda vez que queríamos representar algum conceito abstrato em nosso programa, não importa a forma como ele era implementado

- Programar com interfaces é flexível, mas a restrição de só podermos ter assinaturas de métodos em uma interface às vezes é inconveniente, e pode levar a duplicação de código

- Para contornar isso, temos um segundo mecanismo para representar objetos abstratos: as *classes abstratas*

Classes abstratas - sintaxe

- Uma classe abstrata é como qualquer outra classe, exceto que o corpo de alguns de seus métodos dispara um *erro de método não implementado*:

```
class FiguraPt:
    def mover(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def desenhar(self, canvas):
        raise NotImplementedError()

    def dentro(self, x, y):
        raise NotImplementedError()
```

métodos abstratos

Classes abstratas – métodos abstratos

- Uma classe abstrata pode ter campos e métodos, como qualquer outra classe, mas não devemos instanciar uma classe abstrata, mesmo se ela tiver um construtor `__init__`

- Os métodos que declaramos como `raise NotImplementedError()` são *métodos abstratos*

- A ideia é que os métodos abstratos são métodos que vão ser definidos depois, nas classes concretas que *derivam* da classe abstrata

- Se a classe abstrata não tiver um construtor, a classe concreta também deve fornecer um, e inicializar os campos pedidos pela classe abstrata, assim como seus campos em particular

Usando classes abstratas - herança

- Se não devemos instanciar uma classe abstrata diretamente, para ter instâncias dela criamos uma classe concreta que *deriva* ou *herda* da classe abstrata

herança

```
class Retangulo(FiguraPt):
    def __init__(self, x, y, largura, altura, cor):
        self.x = x
        self.y = y
        self.largura = largura
        self.altura = altura
        self.cor = cor

    def desenhar(self, canvas):
        canvas.retangulo(self.x, self.y, self.largura, self.altura, self.cor)

    def dentro(self, x, y):
        return (x >= self.x and x <= self.x + self.largura and
                y >= self.y and y <= self.y + self.altura)
```

Herança

- A relação de herança, como a implementação de uma interface, também é uma relação “é-um”

- Uma instância de `Retangulo` *é uma* instância de `FiguraPt`

- As relações “é um” são transitivas: uma instância de `Retangulo` é uma instância de `FiguraPt` (via herança), e uma instância de `FiguraPt` é uma instância da interface `Figura` (por implementação de seus métodos), então uma instância de `Retangulo` é uma instância de `Figura`

- A diferença da herança é que nela a classe herda a *forma* da outra classe, incluindo todos os seus campos, seus métodos, e seu construtor

Redefinição de métodos

- A subclasse não está restrita a só fornecer construtores e implementações para métodos abstratos
- Ela também pode *redefinir* métodos, dando uma outra implementação para eles
- Uma redefinição tem a mesma assinatura do método que está sendo redefinido
- Instâncias da subclasse usam sempre a nova implementação do método
- Dentro da redefinição de um método, podemos chamar sua definição original com `<nome da superclasse>.<nome do método>(self, <outros argumentos>)`

FiguraPt e Retangulo, com redefinição

```
class FiguraPt:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cor = cor

    def mover(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def desenhar(self, canvas):
        raise NotImplementedError()

    def dentro(self, x, y):
        raise NotImplementedError()
```

```
class Retangulo(FiguraPt):
    def __init__(self, x, y, largura, altura, cor):
        FiguraPt.__init__(self, x, y, cor)
        self.largura = largura
        self.altura = altura

    def desenhar(self, canvas):
        canvas.retangulo(self.x, self.y,
                         self.largura,
                         self.altura, self.cor)

    def dentro(self, x, y):
        return (x >= self.x and
                x <= self.x + self.largura and
                y >= self.y and
                y <= self.y + self.altura)
```

Herança de classes concretas e `object`

- A superclasse que passamos para a cláusula `extends` não precisa ser uma classe abstrata

- Pode ser uma classe qualquer, assim podemos criar novas versões de uma determinada classe, mas que reaproveitam parte do seu código

- Quando não queremos dar uma superclasse, podemos fazer a nossa classe herdar de object, uma classe que nos dá vários métodos úteis para redefinir

```
class FiguraPt(object):
```

`__str__` e `__repr__`

- `object` tem alguns métodos que são úteis quando redefinidos

- O método `__str__` é chamado toda vez que precisamos converter um objeto para uma string (com `print` ou `str`), enquanto `__repr__` é chamado quando o IDLE precisa mostrar o objeto no console

```
class A(object):
    def __repr__(self):
        return "A"

    def __str__(self):
        return "B"
```

```
>>> x = A()
>>> x
A
>>> print x
B
>>> str(x)
'B'
>>> repr(x)
'A'
>>>
```

__eq__, __ne__, __hash__

- Os métodos __eq__ e __ne__ são usados pelas operações == e !=, e devemos redefini-los quando queremos comparar objetos por estrutura e não por referência

- O método __hash__ é usado pelos dicionários como um "resumo" da estrutura de um objeto (um número inteiro); dois objetos "iguais" devem ter o mesmo __hash__, ou quebramos dicionários com esses objetos como chaves

```
class Ponto(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, o):
        return self.x == o.x and self.y == o.y

    def __neq__(self, o):
        return self.x != o.x or self.y != o.y

    def __hash__(self):
        return hash(self.x) + hash(self.y)
```

```
>>> p1 = Ponto(2, 3)
>>> p2 = Ponto(2, 3)
>>> p3 = Ponto(2, 4)
>>> p1 == p2
True
>>> p1 is p2
False
>>> p1 == p3
False
>>> p1 != p3
True
>>> dict = {}
>>> dict[p1] = 5
>>> dict[p2]
5
```