Paradigmas de Programação (INE5616)
Atividade 2: Orientação a Objetos em Python

Ramna Sidharta de Andrade Palma

February 2018

1 Declaração de classes, métodos, atributos e construtor

Como sabido da Orientação a Objetos, uma classe associa dados (atributos) e operações (métodos). E um objeto é uma variável cujo tipo é uma classe, ou seja, um objeto é uma instância de uma classe.

A definição de uma classe na sua sintaxe mais elementar segue a seguinte sintaxe:

```
class ClassName(object):
    pass
```

Como é sabido, na linguagem **Python** usa-se o espaçamento para determinar o escopo, ao invés de chaves ({, }), então todos os atributos e métodos da classe são declarados no nível de indentação do `pass` acima. Esta é a palavra reservada usada quando algo é requerido sintaticamente, mas não tem-se ou não quer-se ter nenhum comando ou código a ser executado. Por isso ele foi usado acima, a classe é vazia.

Definimos funções dentro de uma classe (métodos), usando primeiro a palavra reservada `def`, precedendo seu nome. Como abaixo (os argumentos, opcionais, são separados por vírgula):

```
def method_name(argument1, argument2):
    pass
```

Se um valor deve ser retornado, usa-se simplesmente `return value`. Como o **Python** não tem tipagem explícita, nada precisa ser alterado na assinatura do método.

Os argumentos da classe são definidos ao mesmo nível dos métodos. Abaixo temos a construção básica de uma classe com argumento e método:

```
class ClassName(object):
    attribute = None
```

```
    def method_name(self, argument1, argument2):
        return argument1
```

Vemos no exemplo acima que o método está recebendo um argumento de nome `self`. Na verdade toda função definida dentro de uma classe deve definir como primeiro parâmetro o `self`, que representa a própria instância (o objeto) que está executando aquele código.

O construtor define um objeto "customizado", especificando um estado inicial. Ele é sempre nomeado `__init__`, e pode receber parâmetros. Vemos a seguir um exemplo:

```
class ClassName(object):

    # o construtor também deve sempre receber o parâmetro 'self'
    def __init__(self):
        self.data = []
```

2 Instanciação de Classes

A instância de um objeto é feita usando o nome da classe seguido por parêntesis. Se o construtor da classe exigir argumentos, eles devem ser passados entre os parêntesis, separados por vírgula, assim como na chamada de um método.

```
class ClassName(object):

    def __init__(self, arg1):
        self.attribute = arg1
        self.data = []

    def add(self, x):
        self.data.append(x)


obj = ClassName("an argument")
obj.add(1)
print(obj.data)
```

O código acima resulta em "`[1]`" na saída padrão do programa.

3 Herança Simples, Herança Múltipla e Polimorfismo

Começaremos vendo um exemplo, para então entendermos o que foi escrito.

```
class Parent(object):
```

```
    def __init__(self, age):
        self.age = age

    def add(self, x):
        self.data.append(x)

    def id(self):
        print(self.age)


class Son(Parent):

    def __init__(self, age, hair):
        super().__init__(age)
        self.hair = hair

    def id(self):
        print(self.hair)
```

Para definir de quem uma classe filha herda, basta definir o nome da classe mãe entre os parêntesis após o nome da classe sendo construída. O **Python** também suporta **Herança Múltipla**, isto é, uma classe pode herdar de mais de uma classe. Para isso basta separar o nome das classes mãe entre parêntesis.

O **construtor** de uma classe filha deve sempre chamar o construtor de sua classe mãe, com o comando `super().__init__()` (este comando não precisa ser o primeiro a ser executado no construtor).

No exemplo acima temos que a classe `Son` recebe um terceiro argumento (`hair`), mostrando que é uma classe que especializa `Parent`, portanto pode ter mais argumentos. Além disso, para sobre-escrevermos um método da classe mãe, basta usarmos a mesma assinatura, como é feito com o método `id()`.

Tendo entendido como funciona a herança em **Python**, abaixo define-se um exemplo sobre polimorfismo.

```
class GraphicObject(object):
    def __init__(self, centro):
        super(GraphicObject, self).__init__()
        self.center = center

    @abstractmethod
    def draw(self):
        pass

    def erase(self):
        self.setPenColor(self.BACKGROUND_COLOR)
        self.draw()
        self.setPenColor(self.FOREGROUND_COLOR)
```

```
    def moveTo(self, p):
        self.erase()
        self.center = p
        self.draw()


class Circle(GraphicObject):
    def __init__(self, center, radius):
        super().__init__(center)
        self.radius = radius

    def draw(self):
        # desenha círculo


class Rectangle(GraphicObject):
    def __init__(self, center, height, width):
        super().__init__(center)
        self.height = height
        self.width = width

    def draw(self):
        # desenha retângulo

class Square(Rectangle):
    def __init__(self, center, width):
        super().__init__(center, width, width)
```

Imaginando o seguinte cenário de **polimormismo**: vários objetos gráficos (círculos, retângulos, quadrados...), todos devem suportar o mesmo tipo de operação, contudo eles se comportam de maneira diferente às vezes. A classe `GraphicaObject` representa a entidade que define as operações comums para cada especialização de objeto gráfico. Qualquer objeto gráfico pode ser apagado ou movido, então esses métodos são implementador naquela classe, já o método de desenhar não pode ser definido ainda, pois cara objeto gráfico possui uma forma diferente, sendo assim ele é declarado como um método abstrato.

Exemplo de uso:

```
c = Circle(Point(0, 0), 5)

c.draw()
c.moveTo(Point(10, 10))
c.erase()
```

Ao instanciar `Circle` o construtor de classe `GraphicObject` já foi chamado, `c` é um `GraphicObject` também, portanto pode ser movido e apagado.

Considere agora o código abaixo:

```
g1 = Circle(Point(0,0), 5)
g2 = Square(Point(0,0), 5)
g1.draw()
g2.draw()
```

A linha `g1.draw()` executa `Circle.draw()`, enquanto que a linha `g2.draw()` executa `Rectangue.draw()`, pois esta sabe o método real a ser invacado, este é o Polimorfismo. Horas um objeto se comporta como sendo de um Tipo, horas de outro, sendo este comportamento transparente ao programador.

4 Métodos Abstratos e Estáticos

Na seção anterior foi mostrado o uso de método abstrato: usa-se simplesmente `@abstractmethod` acima do método, sem definir nenhuma implementação (está é feita na subclasse, que simplesmente define a assinatura do método abstrato, sem a adição do "`@abstractmethod`").

Para o caso de **métodos estáticos**, basta alterar a tag citada acima por `staticmethod`. É claro que neste caso o método com essa tag deve ter implementação. A chamada de um método estático é feita trivialmente, como na grande maioria das linguagens que suportam o paradigma orientado a objetos: `NomeDaClasse.metodoEstatico(args)`.