02

Encapsulamento

Segue o link (https://s3.amazonaws.com/caelum-online-public/python/09-python.zip) com os arquivos utilizados nesta aula.

Encapsulamento

Nas redes sociais concorrentes, podemos curtir atividades desses perfis ou até mesmo o próprio perfil. Que tal implementarmos esta funcionalidade em nossa aplicação? Mas temos uma regra a seguir: as curtidas precisam ser incrementais, isto é, um perfil que tem 11 curtidas, só pode pular para 12 e depois para 13 e por aí vai. Vamos adicionar o atributo curtidas em nossa classe. Todos os novos perfis começarão com valor zero:

```
# -*- coding: UTF-8 -*-
class Perfil(object):
    'Classe padrão para perfis de usuários'

    def __init__(self, nome, telefone, empresa):
        self.nome = nome
        self.telefone = telefone
        self.empresa = empresa
        self.curtidas = 0

    def imprimir(self):
        print "Nome : %s, Telefone: %s, Empresa %s" % (self.nome, self.telefone, self.empresa)
```

Não temos ainda um menu, muito menos uma interface gráfica para interagir com nosso perfil. Porém, nossa aplicação em algum momento precisará incrementar o total de curtidas do perfil. Vamos simular esta chamada:

```
>>> perfil = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> perfil.curtidas
0
>>> perfil.curtidas+=1
>>> perfil.curtidas
1
```

Excelente! Porém, é certo fazermos isso?

```
>>> perfil.curtidas = 1000
>>> perfil.curtidas
1000
```

Programaticamente é correto, mas fere ferozmente nossa regra de negócio. E agora? Como garantiremos que as curtidas sejam feitas incrementalmente? Que tal criarmos um método em nossa classe `Perfil` chamado `curtir` que atualizará nosso atributo incrementalmente? Vamos tentar:

```
# -*- coding: UTF-8 -*-
class Perfil(object):
    'Classe padrão para perfis de usuários'
```

```
    def __init__(self, nome, telefone, empresa):
        self.nome = nome
        self.telefone = telefone
        self.empresa = empresa
        self.curtidas = 0

    def imprimir(self):
        print "Nome : %s, Telefone: %s, Empresa %s" % (self.nome, self.telefone, self.empresa)

   def curtir(self):
        self.curtidas+=1
```

Agora, vamos testá-lo:

```
>>> perfil = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> perfil.curtir()
>>> perfil.curtir()
>>> perfil.curtidas
2
```

Excelente, criamos um método em nossa classe responsável pelo incremento de nossas curtidas. Mas o problema é que nada em nosso código nos obriga a chamarmos nosso método, ainda podemos fazer algo do tipo:

```
>>> perfil.curtidas = 1000
```

Continuamos com o mesmo problema, porque qualquer um pode *botar as mãos* no atributo `curtidas`. Por mais que o desenvolvedor seja bem intencionado, ele pode esquecer de chamar nosso método e acessar diretamente o atributo.

Será que podemos tirar essa preocupação do desenvolvedor que não quer correr o risco de acessar diretamente `curtidas`? E se nós conseguíssemos esconder este atributo do desenvolvedor, ele seria capaz de alterá-lo? Não, mas como ele então conseguirá incrementar o total de curtidas? Essa é fácil, pelo método `curtir` que criamos. Ai eu te pergunto: se a única maneira de incrementarmos `curtidas` é através de nosso método, corremos o risco de alguém incrementá-lo indevidamente? Neste caso, é o próprio objeto que sabe o que é melhor para si, isto é, é o próprio objeto que opera sobre os seus dados. Em orientação a objetos estamos querendo encapsular o atributo `curtidas`, tornando-o uma variável de instância privada. O problema é que o Python não possui este recurso na linguagem. Segundo sua documentação (https://docs.python.org/2/tutorial/classes.html#private-variables-and-class-local-references):

```
"Private" instance variables that cannot be accessed except from inside an object don't exist in Py
```

Porém, há uma convenção no Python para indicarmos que determinado atributo (também pode ser feito com método) da classe não deve ser publico que é através do prefixo `__` em sua declaração. Vamos fazer isso em nossa classe:

```
# -*- coding: UTF-8 -*-
class Perfil(object):
    'Classe padrão para perfis de usuários'

    def __init__(self, nome, telefone, empresa):
        self.nome = nome
        self.telefone = telefone
```

```
        self.telefone = telefone
        self.empresa = empresa
        self.__curtidas = 0

    def imprimir(self):
            print "Nome : %s, Telefone: %s, Empresa %s" % (self.nome, self.telefone, self.empres

     def curtir(self):
        self.__curtidas+=1
```

Sinalizamos que `__curtidas` não deve ser acessado diretamente pela sintaxe `perfil.curtidas`, mas se quisermos saber o total de curtidas? Vamos tentar da maneira que já fizemos?

```
>>> perfil = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> perfil.curtidas
Perfil instance has no attribute 'curtidas'
>>> perfil.__curtidas
Perfil instance has no attribute '__curtidas'
```

Espere um segundo? Não estamos conseguindo acessar o atributo `curtidas` do perfil, parece que ele está privado, não? Não exatamente, o que o Python faz quando encontra o `__` é alterar para um único sublinhado, o nome da classe envolvente e o restante do nome original (no caso, `perfil.__curtidas` vira `perfil._Perfil__curtidas`). O atributo continua lá, acessível, porém um pouco menos óbvio de se referenciar. Não deixa de ser uma forma de escondermos o atributo do mundo externo. Além disso, essa espécie de encapsulamento trata-se de uma maneira de tornar local um nome na classe que o criou, levando esse recurso a ser usado também para evitar conflitos de espaço de nome em instâncias.

Mas como vamos acessar agora o total de curtidas com facilidade? Basta criarmos um método em nossa classe que retorna como valor `self.__curtidas`:

```
# -*- coding: UTF-8 -*-
class Perfil(object):
    'Classe padrão para perfis de usuários'

    def __init__(self, nome, telefone, empresa):
        self.nome = nome
        self.telefone = telefone
        self.empresa = empresa
        self.__curtidas = 0

        def imprimir(self):
                print "Nome : %s, Telefone: %s, Empresa %s" % (self.nome, self.telefone, self.empres

       def curtir(self):
              self.__curtidas+=1

      def obter_curtidas(self):
            return self.__curtidas
```

Agora, testando nossa modificação:

```
>>> perfil = Perfil('Flávio Almeida', 'não informado', 'Caelum')
>>> perfil.curtir()
>>> perfil.curtir()
>>> perfil.obter_curtidas()
2
```

Atributos dinâmicos

Gastamos um bom tempo criando nossa classe `Perfil` que define os atributos e métodos que todo perfil precisa ter. Mas se o programador fizer isso, será que dará erro?

```
>>> perfil.idade = 12
>>> perfil.idade
12
```

Que erro nada, funciona! Conseguimos burlar a definição da nossa classe adicionando mais um atributo que não estava presente na especificação. Isso é possível devido a natureza dinâmica da linguagem: Python permite adicionarmos dinamicamente novos atributos nas instâncias de uma classe.

Agora que você já sabe um pouco dessa característica dinâmica da linguagem, o que acontecerá com o código abaixo:

```
>>> perfil.curtidas = 1000
>>> perfil.curtidas
1000
```

Funciona, mas será que ele alterou nosso total de curtidas, quebrando nosso encapsulamento?

```
>>> perfil.obter_curtidas()
2
```

Não, nosso total continua íntegro. Você lembra que o `__` faz com que o Python altera o nome da variável internamente por um nome aleatório difícil de chutar? Nós desenvolvedores não precisamos nos preocupar, porque interagimos com essa variável através de métodos que também são modificados para apontar para o novo nome da variável. Isso significa que quando nosso código está rodando, não existe mais o atributo com o nome 'curtidas' e o que acabamos de fazer foi adicioná-lo dinamicamente em nossa instância.

Será que isso é uma desvantagem que pode confundir o desenvolvedor? Existem frameworks web como o Django que faz uso pesado dessa característica da linguagem, adicionando dinamicamente em nossas classes atributos e métodos que nos auxiliam bastante nas tarefas do dia-a-dia. É aquele velho ditado "grandes poderes trazem grandes responsabilidades"

Excelente, aprendemos o conceito de encapsulamento da orientação a objetos e como o Python implementa este recurso. Agora é hora dos exercícios.