Capítulo 11

Objetos em Python

Historicamente o paradigma de objetos nasceu como uma das reações da comunidade de software ao problema cada vez mais importante da complexidade crescente do software. É só você olhar para os lados (bancos, companhias aéreas, governos, grandes corporações, hospitais, um carro moderno, etc etc) para ver que a humanidade literalmente lança seu futuro nas mãos do software e seja o que Deus quiser.

A idéia do objeto é encapsular (esconder, impedir que alguém indeterminado faça alterações, coisas desse estilo) dados e procedimentos de maneira a garantir a integridade de uns e de outros. Dito de outra maneira, o paradigma de objetos interpõe um intermediário (o objeto) entre os ativos tecnológicos (dados e procedimentos) e seus usuários (programas e programadores que os manipulam).

Isto é necessário já que é uma evidência empírica que boa parte dos bugs (erros) de programa derivam de efeitos colaterais de manipulação inadvertida de áreas comuns de dados dentro dos programas. Sobretudo, se não são os maiores erros em número, certamente são os mais difíceis de corrigir por (aparentemente) muito pouco terem a ver com as ações que os causaram. Em outras palavras quando a relação causa $\Leftrightarrow$ efeito de um erro é tênue ou escondida, sua correção demora e custa muito mais.

A explicação parece complexa e o é. Razão pela qual – na minha opinião – a metodologia de objetos se aplica maravilhosamente a sistemas grandes e/ou complexos, mas para aprender a programar, ou mesmo para aplicações simples é um exagero seu uso. É como matar um mosquito usando uma bala de canhão.

Mas aqui, de novo, a superioridade de Python: ele pode ser aprendido e usado o tempo todo como uma linguagem imperativa: como se viu este é o melhor jeito de aprender a programar. Quando (e se) lá na frente, você sentir falta das qualidades inerentes ao paradigma orientado ao objeto, *voilà*, não é que Python é 100% orientado ao objeto?

Dito isto, um objeto em Python é uma entidade que tem dados e procedimentos associados, e devidamente encobertos (encapsulados). Um objeto é definido em python pelo uso da cláusula *class*. Veja como:

```
class carro:
```

Note que a definição termina por `:` o que caracteriza que ela continua. A próxima coisa é definir um objeto python de nome `__init__`. Veja

```
  def __init__(self):
```

Este objeto é o *construtor* e ele é automaticamente chamado pelo python toda vez que um objeto é instanciado (definido e usado pela primeira vez). A variável *self* é o objeto propriamente dito. Note que o construtor é uma função, pelo que também termina por `:`.

Nesta função construtora definem-se os valores iniciais do objeto que passarão a fazer parte dele assim que o objeto for criado (instanciado). Acompanhe no exemplo completo

```
class carro:
  def __init__(self):
    self.kilo = 0
    self.marca = 'peugeot'
    self.portas = 5
```

Aqui definiu-se um objeto *carro* e disse-se que a kilometragem do carro é 0, a marca é peugeot e a quantidade de portas é 5.

Depois dessa definição, pode-se criar quantos objetos se quiserem, como em

```
carro1 = carro()
carro2 = carro()
carro2.portas = 3
```

Neste caso, há 2 carros: carro1 e carro2. Em particular, o carro2 tem um número diferente de portas, a saber, 3. Acompanhe

```
print(carro1.portas)
print(carro2.portas)
```

O python responderá 5 para a primeira impressão e 3 para a segunda.

11.1 Um exemplo de objeto: um formigueiro

A seguir uma prova de como é poderoso o conceito de objeto. Eis uma possível simulação de um formigueiro, que pode ser descrito como

- Uma coletividade de inúmeras (50, 100, 200 ? quantas ?) formigas
- As formigas são similares, isto é, todas têm o mesmo comportamento,
- Mas cada uma é um indivíduo separado das demais, e nesta implementação, uma não interfere na outra.

Se não se usasse a orientação a objeto, haveria que programar a coisa na unha, controlando para cada formiga seus dados, e lutando bravamente para que os dados de cada formiga não fossem (inadvertidamente) alterados por outra formiga. O software seria grande, complexo, sujeito a largos processos de depuração e correção.

Ao usar objetos, veja-se as vantagens adquiridas.

- Programa-se apenas 1 formiga. Não importa quantas formigas existam. Este enfoque permite que o comportamento dessa formiga seja tão complexo e rebuscado quanto se queira. Afinal é só uma formiga.
- Cada formiga enxerga seus dados e pode manipulá-los sem medo. Não há como uma formiga alterar dados de outra: lembre-se que no programa só há uma formiga.
- A criação de várias formigas é tarefa simplória: Definido o objeto, muitas formigas são criadas em um bloco for, que gera instâncias diferentes da mesma formiga. Agora, passar de 10 a 1000 formigas é apenas mudar o valor final de um laço for.
- O programa fica menor. Seus erros de implementação são sensivelmente menores e portanto sua implementação mais rápida.
- Os bugs misteriosos não aparecem.

```
# exemplo de objetos (formigas em um gramado)
import turtle, random
class formiga:
  def __init__(self,i): #aqui define-se o comportamento inicial da formiga
                        # um parâmetro passado é um número i
    self.t=turtle.Pen() #cria-se uma turtle de nome t no objeto e ela tem o método pen
    tcor=["violet","red","yellow","green","blue","brown","orange","pink","black",
          "purple","olive","fuchsia","blueviolet","plum",
          "coral","cornsilk","chartreuse","cyan","aqua","crimson"] #as diversas cores
#    ind=random.randint(0,7)
    self.t.color=tcor[i%len(tcor)]    #a cor é atribuida de maneira circular
    self.t.ondex=200-random.randint(0,399) #a formiga começa em x= -200..200 randomicamente
    self.t.ondey=200-random.randint(0,399) # idem para y
    self.t.direcao=random.randint(0,359)   # ganha uma direção randômica 0..360
    self.t.potencia=random.randint(1,11)   # ganha uma potência randômica 1..11
    self.t.penup()                         # levanta a caneta
    self.t.goto(self.t.ondex,self.t.ondey) # vai para onde ela está
    self.t.pendown()                       # abaixa a caneta

  def anda(self):  # como a formiga anda
    self.t.penup()                         # levanta a caneta
    self.t.goto(self.t.ondex,self.t.ondey) # vai para onde ela está
    self.t.pensize(self.t.potencia)        # associa a grossura do traço à potência
    self.t.pendown()                       # abaixa a caneta
    self.t.pencolor(self.t.color)          # associa a cor da caneta
    dist=random.randint(25,70)             # dist= randomica 25..70
    ndir=random.randint(0,359)             # ndir= direção randômica 0..359
    self.t.right(ndir)                     # manda girar à direita ndir
    self.t.fd(dist)                        # manda andar dist
    x=self.t.pos()                         # descobre onde parou (x,y)
```

```
    self.t.ondex=x[0]                    # guarda onde parou (x)
    self.t.ondey=x[1]                    # guarda onde parou (y)


x=[0]*60 # criam-se 60 formigas
for i in range(0,len(x)):
  x[i]=formiga(i)   # cada formiga ganha a cor i
  x[i].anda()       # anda
for j in range(len(x)):   realizam-se 14400 (60*60*4) movimentos
  for i in range(len(x)):   # cada formiga (são 60) realiza 4 movimentos a cada ciclo
    x[i].anda()
  for i in range(len(x)):
    x[i].anda()
  for i in range(len(x)):
    x[i].anda()
  for i in range(len(x)):
    x[i].anda()
```

11.1.1 Herança

Em uma relação de herança, existem no mínimo 2 classes envolvidas: A mais genérica chamada superclasse ou classe pai. A outra é a mais específica, chamada de subclasse, classe filha ou ainda derivada. É comum em uma classe derivada sobrecarregar os métodos da classe pai.