

Luís Henrique Carvalho da Cruz
Fundamentos do Desenvolvimento de Software
2022.1-B

Orientação a Objetos com Python

Recife
2022

A Programação Orientada a Objetos é um paradigma centrado em entidades de dados, tendo como principais elementos os objetos e as classes, e ao redor dessas entidades é construída a lógica para manipulá-las. Entre as vantagens desse paradigma estão:

- Reusabilidade de código
- Encapsulamento
- Modularidade
- Eficiência
- Flexibilidade

Esse documento irá discutir os principais elementos que compõem a orientação a objetos, com códigos explicativos na linguagem Python.

Classes e Objetos

Uma classe é como uma planta baixa para a criação de objetos, um estabelecimento de um tipo específico de dado e indicando quais devem ser seus atributos e métodos (atributos e métodos serão abordados mais adiante).

Um objeto é uma unidade de dados que deriva de uma classe e segue essa “planta baixa”.

Observe esse conceito com uma analogia ao mundo real: imagine a espécie humana.

Generalizando um pouco, todos os humanos têm nome e idade. No exemplo, a espécie humana é nossa classe e cada pessoa no mundo é um objeto derivado da classe Humano, pois as pessoas seguem essa “planta baixa” definida, onde as características de nome e idade são atributos. Talvez um exemplo em código seja mais prático:

```
1  class Human:
2
3      # função construtora
4      def __init__(self, name, age):
5
6          # self diz "eu mesmo"
7          self.name = name
8          self.age = age
```

Explicando linha por linha:

- Linha 1: declaração da classe em Python. É comum em todas as linguagens que o nome da classe comece em maiúsculo.
- Linha 4: função construtora, que é chamada automaticamente ao instanciar uma classe em um objeto. A implementação pode diferir mas nesse caso ela apenas atribui os valores passados de parâmetro aos atributos da classe.
- Linhas 7 e 8: atribuição de valores aos atributos do objeto. Self diz à classe para referir a si mesma, leia algo do tipo “eu declaro que meu nome agora é X(valor passado na chamada)”

Agora explicando a função construtora: ao instanciar uma classe, a chamada da

construtora será feita automaticamente. Ela se difere das outras pelo nome “__init__” e ela pode fazer o que o programador quiser. No caso do exemplo anterior, ela recebe dois parâmetros (name, age) na chamada, sendo *self* um parâmetro passado implicitamente pelo Python para o objeto referir a si mesmo. Veja a seguir:

```
ricardo = Human('Ricardo Costa', 30)
luis = Human('Luís Cruz', 18)
```

Perceba que a cada instanciação são passados parâmetros diferentes, mas seguindo a mesma planta baixa. Agora, **ricardo** e **luis** são objetos instanciados a partir da classe Human, cada um com seus atributos.

```
ricardo = Human('Ricardo Costa', 30)
luis = Human('Luís Cruz', 18)

print(ricardo.name)
print(luis.name)
```

Saída:

```
PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\01exclassesobjs.py
Ricardo Costa
Luís Cruz
```

Atributos de Classes

De uma forma simples, atributos de classes são basicamente variáveis que pertencem a classes. Em Python, os atributos são definidos na declaração da classe ou na instanciação de um objeto a partir da função construtora. Em outras linguagens é possível declarar variáveis e não atribuir valores na hora mas no Python é obrigatório.

```
class Example:

    # atributo definido no top-level da classe
    name = 'class example'

class Example2:

    def __init__(self, name):
        # atributo definido pela construtora ao instanciar
        self.name = name
```

Agora observe ao printar:

```

ex1 = Example()
ex2 = Example2('another class example')

print(ex1.name)
print(ex2.name)

PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\02exatributos.py
class example
another class example

```

Observe que o atributo definido pela função construtora é encapsulada a aquela instância específica, enquanto o predefinido é inerente a todas as instâncias:

```

newObj = Example()
anotherObj = Example()
lastObj = Example()

print(newObj.name)
print(anotherObj.name)
print(lastObj.name)

walter = Example2('heisenberg')
jesse = Example2('pinkman')

print(walter.name)
print(jesse.name)

```

Output:

```

class example
class example
class example
heisenberg
pinkman

```

Métodos de classes

Um método é como uma função pertencente a uma classe. Mas não é apenas isso, um método é um procedimento atrelado a aquela classe geralmente mas não obrigatoriamente relacionado com os dados que ela lida, além de estar presente em todos os objetos instanciados, desde que seja um método público. Eis um exemplo de método que manipula objetos do tipo daquela classe específica:

```
# qualquer string é um objeto do tipo str()
str = 'this is a string'

print(str)

# .upper() é um método de strings, um método proveniente de str()
# .upper() retorna uma cópia da sua string convertida em maiúsculas
str = str.upper()
print(str)

PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\03exmetodos.py
this is a string
THIS IS A STRING
```

Agora um exemplo de função pertencente a uma classe que não manipula objetos daquele tipo:

```
class Utils:
    # classe utils não tem atributos, não tem um tipo de dado específico
    # método não manipula objetos do tipo Utils

    def sum(num1, num2):
        return num1 + num2
```

Tipos de Objetos

Como conhecemos dos fundamentos da programação, os dados manipulados pelo seu programa tem tipos, como string, int, float, char, double, etc. Ao criar uma classe, está sendo definido também um tipo. Em Python curiosamente todos os tipos são classes, por isso dizemos “métodos de strings” ou “métodos de int” pois quando usamos os métodos de manipulação de strings por exemplo, estamos chamando métodos da classe **str()** num objeto de string, ou seja, um objeto do tipo string.

```
class Human:
    pass

# definida e instanciada uma classe Human, vamos verificar
# o tipo do objeto de Human
luis = Human()
print(type(luis))

# paralelamente, vamos verificar o tipo de uma string
```

```
str = 'oi mate, care for a cup of tea?'
print(type(str))

# por fim, o tipo de um inteiro
num = 1
print(type(num))
```

```
PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\04objtypes.py
<class '__main__.Human'>
<class 'str'>
<class 'int'>
```

Observamos na saída que nosso objeto instanciado recebe como tipo a classe do qual ele deriva, e qualquer classe que criarmos também será o tipo dos objetos que dela derivam.

Variáveis privadas

Variáveis privadas são variáveis inacessíveis por um objeto de dada classe, apenas internamente a classe pode manipular aquela variável. Em algumas linguagens como Java ou Typescript existe a palavra chave “private” para definir se tal variável é pública ou privada, já em Python definimos isso ao colocar dois *underlines* antes do nome da variável:

```
class Example:

    # definição de variável privada, inacessível por objeto
    __privateVariable = 10

    # variável pública para comparação
    regularVariable = 20
```

```
obj1 = Example()

# printando var acessível pelo objeto
print(obj1.regularVariable)

# printando var inacessível pelo objeto
print(obj1.__privateVariable)
```

Output:

```
PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\05exvarprivadas.py
20
Traceback (most recent call last):
  File "C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab\05exvarprivadas.py", line 15, in <module>
    print(obj1.__privateVariable)
AttributeError: 'Example' object has no attribute '__privateVariable'
```

É printado apenas a variável regular enquanto que a tentativa de acessar a privada resultou em exceção.

Embora inacessível pelo objeto instanciado, os métodos da classe ainda podem acessar essas variáveis privadas. Observe esse exemplo de uma classe muito séria de criptografia com atributos super secretos:

```
class Cryptography:

    # definindo var privada
    __secretHash = 10

    # método que utiliza var privada
    def hashPassword(self, passw):
        return passw * self.__secretHash

# instancição
passwGenerator = Cryptography()

# chamada de método salvando retorno em myPassword
myPassword = passwGenerator.hashPassword(123)

# printando retorno salvo
print(myPassword)
```

Output:

```
PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\05exvarprivadas.py
1230
```

Como está posto no output do programa, o método recebeu o parâmetro da sua chamada e fez seu procedimento de multiplicar acessando a variável privada, logo, métodos internos conseguem acessar variáveis privadas, enquanto objetos não conseguem diretamente.

Herança

Imagine que classes, assim como pessoas, podem ter relações de parentesco. Sendo mais específico, imagine que uma classe pode ter uma classe filha (ou child) e a classe child pode herdar todas as características (atributos e métodos) da classe pai. Fazendo uma analogia com a genética humana, um bebê que nasce herdando todas as características de um pai. Observe o exemplo a seguir:

```
class Parent:

    # definindo um atributo da classe parent
    whatami = 'a class'

class Child(Parent):

    # a classe child apenas estende a classe parent
    # não foi definido nenhum atributo, a child apenas herdou os atributos
    # usamos a palavra chave "pass" para o python seguir em frente
    pass
```

Agora se tentarmos printar os atributos relativos a cada classe:

```
print(Parent.whatami)
print(Child.whatami)
```

Observe que o atributo **whatami** é acessível pela classe filha e o output é o mesmo, pois o que era da classe parent agora também é da classe child:

```
PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\06exheranca.py
a class
a class
```

Também é possível herdar todos os atributos da classe parent e ainda ter os próprios atributos, afinal, nenhum filho nasce uma cópia idêntica do pai. Observe:

```
class Child(Parent):

    iAm = 'the bluest of blues'

print(Child.iAm)

# atributo dado a classe filha além das heranças
# não existe na classe pai
```



```
print(Parent.iAm)
```

```
the bluest of blues
Traceback (most recent call last):
  File "C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab\06exheranca.py", line 17, in <module>
    print(Parent.iAm)
AttributeError: type object 'Parent' has no attribute 'iAm'
```

No output acima, está comprovado que o atributo iAm, atribuído exclusivamente à classe Child é acessível apenas por ela e não pela classe Parent, enquanto tudo que era da Parent é acessível pela Child.

Métodos Herdados

Da mesma forma que classes filhas herdam atributos das classes pais, elas também podem herdar métodos. A lógica e a analogia é a mesma da herança de atributos, mas haverão algumas diferenças. Observe em código:

```
class Parent:
    # definindo a função sum na classe pai
    def sum(num1, num2):
        return num1 + num2

class Child(Parent):
    # definindo a classe child herdando tudo da classe pai
    pass

# acessando sum pela classe pai
print(Parent.sum(2, 3))

# acessando sum pela classe child
print(Child.sum(2, 3))

# as duas chamadas são acessíveis e retornam com sucesso
```

Observe a saída, comprovando os comentários:

```
PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\07exherancamets.py
5
5
```

Como citado anteriormente, a herança de métodos pode ter algumas diferenças, mas esses conceitos serão explicados em seus próprios tópicos posteriormente.

Herança múltipla

Até agora, as analogias usadas têm tomado como exemplo um filho auto-gerado de uma gravidez espontânea, porém o tópico atual requer uma mudança de analogias. Imagine agora que uma classe tenha um pai e uma mãe, ou vários pais ou várias mães. O conceito de herança múltipla se dá pela capacidade de uma classe filha herdar atributos e métodos de duas ou mais classes pais. Veja na prática:

```
class Vader:

    # classe pai e seu método
    def fatherMessage():
        print('luke...')

class Padme:

    # outra classe pai e seu método
    def motherMessage():
        print('lol your dad went full sith mode then i died')

class Luke(Vader, Padme):

    # classe child herdando duas classes pais
    pass
```

Evidenciando que a classe filha tem acesso aos métodos das duas classes pais, observe esse exemplo:

```
print(Luke.fatherMessage())
print(Luke.motherMessage())
```

Ao tentar acessar as mensagens, a saída é a seguinte:

```
PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\08exmultiheranca.py
luke...
lol your dad went full sith mode then i died
```

Um detalhe, digamos que suas classes pais tenham métodos de mesmo nome:

```
class Skater:

    # Skater tem um método greeting
    def greeting():
```

```

    print('i skate')

class Surfer:

    # Surfer também tem um método greeting
    def greeting():
        print('i surf')

    # classe AverageCalifornian herdando tanto Skater quanto Surfer
class AverageCalifornian(Skater, Surfer):

    pass

# qual será o output dessa chamada?
AverageCalifornian.greeting()

```

Observe o output para entender o comportamento dessa situação:

```

PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\08exmultiheranca.py
i skate

```

Quando herdando várias classes com métodos de mesmo nome, a prioridade é dada de forma “*left to right*” quando passando os parâmetros de herança na definição da classe filha:

```

class AverageCalifornian(Skater, Surfer):

```

No caso da classe AverageCalifornian, o Python deu prioridade ao método greeting da classe pai Skater, pois essa é passada primeiro.

Sobrescrita de Métodos

Até agora, vimos que ao herdar métodos, as classes filhas realizam a mesma implementação da classe pai. Essa implementação, contudo, pode ser sobrescrita para a classe filha ter a própria implementação:

```

class Animal:

    # o método original
    def walk():

```

```

    print('using legs...')

class Dog(Animal):

    # o método sobrescrito
    def walk():
        print('on four legs...')

```

Se você lembrar do tópico anterior, surge a questão: quando o método walk da classe Dog for chamado, o que será printado? O walk herdado ou o walk sobrescrito?

```
Dog.walk()
```

```

PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\09exoverriding.py
on four legs...

```

Como visto, o walk de Dog usa o método sobrescrito. Agora compare com a saída de outra classe que não sobrescreve Animal:

```

class Person(Animal):

    # nada é sobrescrito nem definido
    pass

Person.walk()

```

```

PS C:\Users\luis\Documents\faculdade\2022.1\fds\oop-trab> python3.10 .\09exoverriding.py
using legs...

```

Em suma: quando sobrescrito, a prioridade é da sobrescrição, quando não sobrescrito, é chamada a implementação original.

Sobrecarga de Métodos

A sobrecarga de métodos é um conceito que permite a um mesmo método se comportar de forma diferente de acordo com os parâmetros a ele passados. Veja a seguir:

```

class Greeter:

    # método definido com parâmetro opcional
    def sayHi(self, name = None):

        # comportamentos diferentes de acordo com os parâmetros
        if name is not None:

```

```
print(f'Oh hi, {name}')
```

```
else:
    print('hello, individual')
```

Agora testando diferentes chamadas:

```
greetObj = Greeter()

greetObj.sayHi()
greetObj.sayHi('Mark')
```

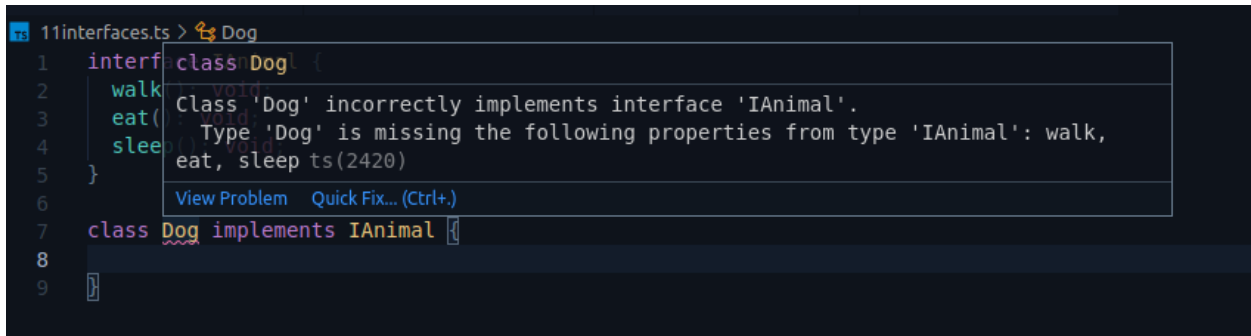
```
hello, individual
Oh hi, Mark
```

Infelizmente, Python não é a melhor das linguagens para exemplificar esse conceito. Em outras linguagens como Java, o programador poderia declarar múltiplas funções de mesmo nome mas com parâmetros diferentes e ele executaria apenas a que fosse chamada com os parâmetros correspondentes à definição. No Python, contudo, estamos limitados a usar uma condicional.

Interfaces

Se classes são uma planta baixa para objetos, interfaces são como uma planta baixa para classes. Imagine como uma série de exigências que dadas classes devem cumprir, diz-se “tal classe deve implementar tal interface”. Esse conceito permite definir uma maior organização e facilita o debugging, pois fica mais óbvio para o programador e ao programa o que dada classe deve ou não ter. A implementação de interfaces em Python não é algo nativo, diferente de outras linguagens como Typescript, Java, C#, etc.

Exemplificando em outra linguagem primeiro, ao aplicarmos uma interface de Animal, ordenamos uma classe Dog a seguir essa interface:



```
11 interfaces.ts > Dog
1  interface IAnimal {
2      walk(): void;
3      eat(): void;
4      sleep(): void;
5  }
6
7  class Dog implements IAnimal {
8      walk(): void;
9      eat(): void;
10     // sleep(): void;
11 }
```

Class 'Dog' incorrectly implements interface 'IAnimal'.
Type 'Dog' is missing the following properties from type 'IAnimal': walk, eat, sleep ts(2420)

View Problem Quick Fix... (Ctrl+.)

No exemplo acima, a classe Dog não está implementando a interface IAnimal corretamente, então o Intellisense do Typescript reclama imediatamente. Agora vejamos a implementação correta:

```

11 interfaces.ts > ...
1  interface IAnimal {
2      walk(): void;
3      eat(): void;
4      sleep(): void;
5  }
6
7  class Dog implements IAnimal {
8
9      walk(): void {
10         console.log('on four legs...')
11     }
12
13     eat(): void {
14         console.log('meat...')
15     }
16
17     sleep(): void {
18         console.log('all day long...')
19     }
20 }
21

```

Agora a classe dog está implementando a interface corretamente colocando sua própria implementação para cada método exigido pela interface, então o programa não irá acusar problemas. Como implementar uma interface em Python? Um *workaround* seria definir uma classe base e cada classe derivada ser uma child da base. Na prática:

```

class IAnimal:

    # é definida uma classe com seus métodos sem implementação
    # para que cada classe filha faça a sua própria implementação
    # e se não fizer, o programa não quebra, apenas não faz nada (nesse caso)
    def walk():
        pass

    def eat():
        pass

    def sleep():
        pass

```

Então implementando a classe Dog:

```
class Dog(IAAnimal):  
  
    def walk():  
        print('on four legs...')  
  
    def eat():  
        print('meat...')  
  
    def sleep():  
        print('all day long...')
```

A classe Dog está basicamente usando a sobrescrita de métodos para fazer suas próprias implementações da “interface” definida anteriormente. Esse método, no entanto, não vai ter o mesmo efeito que outras linguagens em forçar essas implementações pois Python não é fortemente tipado, então se essas sobrescritas de métodos não acontecessem, o código não ia dar erro e nesse exemplo também não iria quebrar, embora os métodos fossem ficar vazios, mas o programador não iria identificar essa falha no código até testar pois nada estaria ilegal.

Conclusão

Na comunidade de desenvolvedores e engenheiros de software, a discussão entre paradigmas de programação é muito acirrada há décadas. No fim das contas, é questão de opinião e necessidade, afinal, não existe bala de prata. Esse trabalho apresentou os conceitos básicos para iniciantes começarem a implementar orientação a objetos com Python, e também serviu para o autor relembrar e aprender conceitos necessários para prosseguir na atividade contínua do semestre na cadeira de Fundamentos do Desenvolvimento de Software.

Link do repositório contendo os exemplos em código:

<https://github.com/lhckb/trabalho-oop-fds>