Classes e Objetos

Prof. André Backes | @progdescomplicada

Programação procedimental

- Também chamada de programação procedural
  - Contêm um conjunto de passos computacionais a serem executados
  - Problemas são decompostos em sub-problemas
    - Modularização
    - Um programa é construído definindo funções
    - Uma função pode ser chamada a qualquer momento durante a execução do programa
  - A ênfase está nas operações desenvolvidas

Programação orientada a objetos

- Trabalha com o conceito de classe e objeto
  - Dados e operações são agregadas a entidades chamadas objetos
  - Problemas são decompostos em objetos que interagem entre si
  - Cada objeto é uma unidade de software
- A ênfase está na interação
- **Forma geral em Python**

```
class nome:
    ...
    definições de métodos
    ...
```

Terminologia

- Classe
  - Representa um conjunto de objetos que possuem a mesma estrutura de dados (atributos) e comportamento (operações)
  - Exemplo: classe dos Seres Humanos

```
class Ponto:
    x = 0
    y = 0
    def getX(self):
        return self.x
    def getY(self):
        return self.y
    def setX(self,valor):
        self.x = valor
    def setY(self,valor):
        self.y = valor

p = Ponto()
print("valor de x =",p.getX())
p.setX(10)
print("valor de x =",p.x)
```

Terminologia

- Objeto
  - O objeto é uma instância de uma classe
    - Os métodos definem o comportamento dos objetos
    - Seu estado é mantido por meio de atributos
  - Forma de instanciar

    objeto = nome-classe()

```
class Ponto:
    x = 0
    y = 0
    def getX(self):
        return self.x
    def getY(self):
        return self.y
    def setX(self,valor):
        self.x = valor
    def setY(self,valor):
        self.y = valor

p = Ponto()
print("valor de x =",p.getX())
p.setX(10)
print("valor de x =",p.x)
```

Terminologia

- Objeto
  - Cada objeto tem uma identidade própria
  - Ele é distinguível de qualquer outro objeto mesmo que seus atributos sejam idênticos
  - Exemplo de objetos da classe Seres Humanos: Ricardo, João, Ana, ...

```
class Ponto:
    x = 0
    y = 0
    def getX(self):
        return self.x
    def getY(self):
        return self.y
    def setX(self,valor):
        self.x = valor
    def setY(self,valor):
        self.y = valor

p = Ponto()
print("valor de x =",p.getX())
p.setX(10)
print("valor de x =",p.x)
```

Terminologia

- Atributos
  - Basicamente, é a estrutura de dados que vai representar a classe
  - Conjunto de propriedades do objeto
  - Valores internos do objeto
  - Exemplo de atributos da classe Seres Humanos: nome, idade, altura, ...

```
class Ponto:
    x = 0
    y = 0
    def getX(self):
        return self.x
    def getY(self):
        return self.y
    def setX(self,valor):
        self.x = valor
    def setY(self,valor):
        self.y = valor

p = Ponto()
print("valor de x =",p.getX())
p.setX(10)
print("valor de x =",p.x)
```

Terminologia

- Método
  - Conjunto de funcionalidades da classe
  - Definem as habilidades dos objetos
  - Exemplo de métodos da classe Seres Humanos: correr, nadar, ...

```
class Ponto:
    x = 0
    y = 0
    def getX(self):
        return self.x
    def getY(self):
        return self.y
    def setX(self,valor):
        self.x = valor
    def setY(self,valor):
        self.y = valor

p = Ponto()
print("valor de x =",p.getX())
p.setX(10)
print("valor de x =",p.x)
```

Acessando os atributos

- Para acessar os atributos de um objeto utilizamos a seguinte notação
  - **objeto.atributo**

- Desse modo, podemos modificar o seu valor ou usá-lo em expressões
  - Em Python não podemos proibir o acesso aos atributos de objetos
  - Podemos até incluir novos atributos (**p.y**)

```
class Ponto:
    x = 0

p = Ponto()
print("valor de x =",p.x)
p.x = 1
print("valor de x =",p.x)
p.y = 10
print("valor de y =",p.y)


>>>
valor de x = 0
valor de x = 1
valor de y = 10
>>>
```

Acessando os atributos

- O acesso direto aos atributos de objetos não é aconselhável
  - Algumas linguagens permitem restringir o acesso aos atributos de um objeto
  - Neste caso, o atributo é chamado de privado
- Python não possui uma construção sintática equivalente

Acessando os atributos

- Felizmente, o interpretador Python possui suporte parecido a variáveis privadas
  - Métodos e atributos cujo nome é iniciado por dois sublinhados (__y) são considerados privados e não podem ser acessados diretamente

```
class Ponto:
    x = 10    # atributo publico
    __y = 20 # atributo privado

p = Ponto()
print("valor de x =",p.x)
p.x = 1
print("valor de x =",p.x)
print("valor de y =",p.y)

>>>
valor de x = 10
valor de x = 1
Traceback (most recent call last):
  File "G:\Temp\testes_classes.py", line 131, in <module>
    print("valor de y =",p.y)
AttributeError: 'Ponto' object has no attribute 'y'
>>>
```

Encapsulamento

- Serve para controlar o acesso aos atributos de um objeto
  - Trata-se de uma forma eficiente de proteger os dados manipulados da classe
  - Ao invés de modificar os atributos diretamente, eles somente poderão ser acessados pelos métodos da classe
  - O parâmetro **self** é o objeto sobre o qual o método opera

```
class Ponto:
    __x = 10
    __y = 20
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y

p = Ponto()
print("valor de x =",p.getX())
print("valor de y =",p.getY())


>>>
valor de x = 10
valor de y = 20
>>>
```

Encapsulamento

- Cuidado:
  - Os métodos falham se o atributo a ser acessado não existir dentro da classe
  - Uma forma de evitar isso é definir um **construtor**

```
class Ponto:
    __x = 10
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y

p = Ponto()
print("valor de x =",p.getX())
print("valor de y =",p.getY())
>>>
valor de x = 10
Traceback (most recent call last):
  File "G:\Temp\testes_classes.py", line 169, in <module>
    print("valor de y =",p.getY())
  File "G:\Temp\testes_classes.py", line 165, in getY
    return self.__y
AttributeError: 'Ponto' object has no attribute '_Ponto__y'
>>>
```

Construtor

- Construtor é um método especial que é chamado assim que uma nova instância do objeto é criada
  - É responsável pela alocação de recursos necessários ao funcionamento do objeto e da definição inicial dos estados dos atributos
  - Por meio dele podemos garantir que o atributo sempre existe (inicializador de atributo)

Construtor

- Forma geral do construtor em Python

  def __init__(self)

  - Dentro dele podemos definir e inicializar (com um valor pré-definido ou passado por parâmetro) todos os atributos do objeto
  - O parâmetro **self** deve ser sempre o primeiro e é o objeto sobre o qual o método opera

```
class Ponto:
    def __init__(self, valorX):
        self.__x = valorX
        self.__y = 0
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y

p = Ponto(10)
print("valor de x =",p.getX())
print("valor de y =",p.getY())



>>>
valor de x = 10
valor de y = 0
>>>
```

Construtor

- Apesar do nome do construtor iniciar com dois sublinhados, ele não é considerado um método "privado" pelo interpretador Python
  - Outros métodos cujo nome é iniciado por dois sublinhados (**__getY**) são considerados privados e não podem ser acessados diretamente

```
class Ponto:
    def __init__(self, valorX):
        self.__x = valorX
        self.__y = 0
    def getX(self):
        return self.__x
    def __getY(self):
        return self.__y

p = Ponto(10)
print("valor de x =",p.getX())
print("valor de y =",p.__getY())


>>>
valor de x = 10
Traceback (most recent call last):
  File "G:\Temp\testes_classes.py", line 207, in <module>
    print("valor de y =",p.__getY())
AttributeError: 'Ponto' object has no attribute '__getY'
>>>
```

Imprimindo um objeto

- Por definição, a impressão de um objeto não é muito informativa
  - Basicamente, apenas algumas informações técnicas

```
class Ponto:
    def __init__(self, valorX):
        self.__x = valorX
        self.__y = 0
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y

p = Ponto(10)
print(p)

>>>
<__main__.Ponto object at 0x00000000031F8F28>
>>>
```

Imprimindo um objeto

- Podemos definir o método __**str**__ para converter o objeto para texto
  - Assim, podemos definir o que será exibido sempre que o objeto for impresso
  - O método deve retornar uma **string**

```
class Ponto:
    def __init__(self, valorX):
        self.__x = valorX
        self.__y = 0
    def __str__(self):
        return "ponto = (%d, %d)" % (self.__x,self.__y)
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y

p = Ponto(10)
print(p)


>>>
ponto = (10, 0)
>>>
```

Comparando objetos

- Comparar dois objetos não é uma tarefa tão simples quanto possa parecer
  - Por definição, o operador **==** testa se os dois argumentos são o mesmo objeto
  - Nenhuma comparação entre os atributos dos objetos é realizada

Comparando objetos

Exemplo

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

p1 = Ponto(1,2)
p2 = Ponto(1,2)
p3 = p1

if p1 == p2:
    print("p1 e p2: pontos iguais!")
else:
    print("p1 e p2: pontos diferentes!")

if p1 == p3:
    print("p1 e p3: pontos iguais!")
else:
    print("p1 e p3: pontos diferentes!")

print("Id de p1 = ",id(p1))
print("Id de p2 = ",id(p2))
print("Id de p3 = ",id(p3))
```

Saída

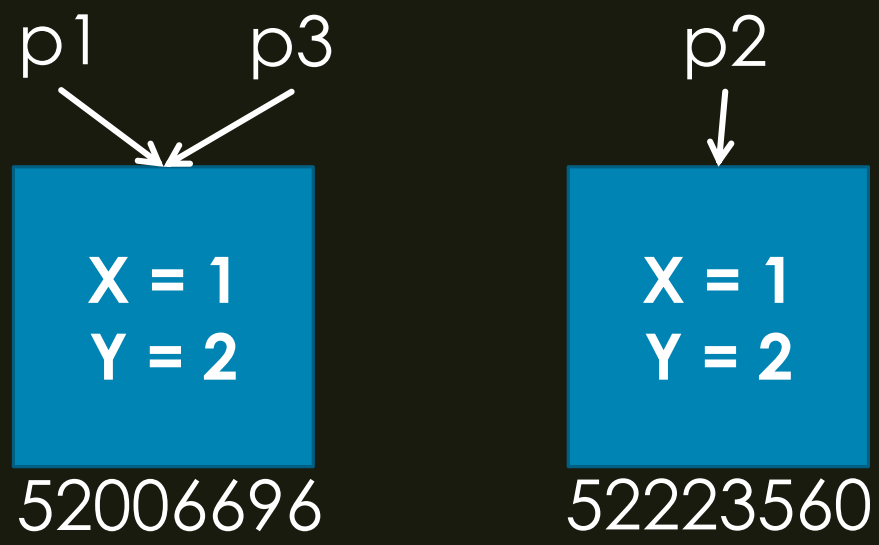
```
>>>
p1 e p2: pontos diferentes!
p1 e p3: pontos iguais!
Id de p1 =  52006696
Id de p2 =  52223560
Id de p3 =  52006696
>>>
```

Comparando objetos

- Nesse exemplo, temos
  - 2 objetos diferentes (instâncias) com o mesmo conteúdo
  - 3 referências diferentes a esses objetos.
  - **A operação de atribuição não cria uma cópia do objeto!**

```
p1 = Ponto(1,2)
p2 = Ponto(1,2)
p3 = p1

Id de p1 =  52006696
Id de p2 =  52223560
Id de p3 =  52006696
```



Comparando objetos

- Para fazer a comparação de objetos, o mais indicado é definir um método na classe para testar se dois objetos possuem os mesmos valores de atributos

```
class Ponto:
    def __init__(self, valorX, valorY):
        self.x = valorX
        self.y = valorY
    def igual(self, po):
        return self.x == po.x and self.y == po.y


p1 = Ponto(1,2)
p2 = Ponto(1,2)

if p1.igual(p2):
    print("Pontos iguais!")
else:
    print("Pontos diferentes!")

print("Id de p1 = ",id(p1))
print("Id de p2 = ",id(p2))
```

```
>>>
Pontos iguais!
Id de p1 =  52268840
Id de p2 =  52485872
>>>
```

Cópia de objetos

- Como vimos, a operação de atribuição não é indica para criar uma cópia de um objeto
  - Precisamos garantir que temos dois objetos diferentes, mas com o mesmo conteúdo
- Uma forma de fazer isso é utilizando a função de cópia que existe no módulo **copy** do Python

Cópia de objetos

Exemplo

```
import copy

class Ponto:
    def __init__(self, valorX, valorY):
        self.x = valorX
        self.y = valorY
    def __str__(self):
        return "ponto = (%d, %d)" % (self.x,self.y)


p1 = Ponto(1,2)
p2 = copy.copy(p1)

print(p1)
print(p2)

print("Id de p1 = ",id(p1))
print("Id de p2 = ",id(p2))
```

Saída

```
>>>
ponto = (1, 2)
ponto = (1, 2)
Id de p1 =  36642160
Id de p2 =  36833328
>>>
```

Cópia de objetos

- A função **copy.copy()** permite duplicar qualquer objeto
  - p1 e p2 não representam mais o mesmo ponto, são objetos diferentes. Mas eles contem os mesmo dados
- Infelizmente, o método copy() faz somente uma ***cópia superficial*** do objeto
  - Esse método não é capaz de copiar objetos embutidos dentro de outros objetos

Cópia de objetos

Exemplo

```
import copy

class Ponto:
    def __init__(self, valorX, valorY):
        self.x = valorX
        self.y = valorY
        self.lista = [10, 20, 30]
    def __str__(self):
        return "ponto = (%d, %d, IdLista = %d)" % (self.x,self.y,id(self.lista))

p1 = Ponto(1,2)
p2 = copy.copy(p1)

print(p1)
print(p2)

print("Id de p1 = ",id(p1))
print("Id de p2 = ",id(p2))
```

Saída

```
>>>
ponto = (1, 2, IdLista = 36631936)
ponto = (1, 2, IdLista = 36631936)
Id de p1 =  36773232
Id de p2 =  39520368
>>>
```

Cópia de objetos

- Tentando copiar um objeto **lista** de dentro do objeto **Ponto**: mesma lista em objetos diferentes!
- Nesse caso, precisamos de uma ***cópia profunda*** dos dados
  - Precisamos copiar todos os níveis de um objeto
- Uma forma de fazer isso é utilizando a função de **deepcopy()**, também do módulo **copy** do Python
  - Esse método copia não somente o objeto, mas também todo e qualquer objeto embutido neste objeto

Cópia de objetos

Exemplo

```
import copy

class Ponto:
    def __init__(self, valorX, valorY):
        self.x = valorX
        self.y = valorY
        self.lista = [10, 20, 30]
    def __str__(self):
        return "ponto = (%d, %d, IdLista = %d)" % (self.x,self.y,id(self.lista))

p1 = Ponto(1,2)
p2 = copy.deepcopy(p1)

print(p1)
print(p2)

print("Id de p1 = ",id(p1))
print("Id de p2 = ",id(p2))
```

Saída

```
>>>
ponto = (1, 2, IdLista = 36566480)
ponto = (1, 2, IdLista = 30727904)
Id de p1 =  36707696
Id de p2 =  36898864
>>>
```

Sobrecarga de operadores

- Nada mais é do que a possibilidade definir o comportamento de alguns operadores básicos da linguagem para novos tipos de dados
  - Exemplo: ==, >, <, +, -, *, etc.
- É apenas uma conveniência. Com a sobrecarga, podemos escrever
  - p1 + p2 ao invés de p1.soma(p2)
  - p1 == p2 ao invés de p1.igual(p2)

Sobrecarga de operadores

- A linguagem Python disponibiliza vários métodos que podem ser implementados e que correspondem a vários operadores

Operador	Método	Exemplo
+	__add__	A + B
-	__sub__	A – B
*	__mul__	A * B
/	__div__	A / B
==	__eq__	A == B
!=	__ne__	A != B
>	__gt__	A > B
<	__lt__	A < B

Sobrecarga de operadores

Exemplo

```
class Ponto:
    def __init__(self, valorX, valorY):
        self.x = valorX
        self.y = valorY
    def __str__(self):
        return "ponto = (%d, %d)" % (self.x,self.y)
    def __add__(self, po): #soma
        pt = Ponto(self.x + po.x, self.y + po.y)
        return pt
    def __eq__(self, po): #igualdade
        return self.x == po.x and self.y == po.y

p1 = Ponto(1,2)
p2 = Ponto(1,2)

if p1 == p2:
    print("Pontos iguais!")
else:
    print("Pontos diferentes!")

p3 = p1 + p2
print(p3)
```

Saída

```
>>>
Pontos iguais!
ponto = (2, 4)
>>>
```

Material Complementar

- Vídeo Aulas
  - Aula 38 - Programação Orientada a Objetos (POO)
  - https://youtu.be/jm3jDYIOAxs
  - Aula 39 - POO: Acesso aos atributos e métodos
  - https://youtu.be/i6tgjRB3mtA
  - Aula 40 - POO: Construtor e Destrutor
  - https://youtu.be/VBQRmafpQBs
  - Aula 41 - POO: Imprimindo um objeto
  - https://youtu.be/PRriYps2Pcw

Material Complementar

- Vídeo Aulas
  - Aula 42 - POO: Comparando objetos
  - https://youtu.be/_5MYPk_6EtQ
  - Aula 43 - POO: Sobrecarga de operadores
  - https://youtu.be/c83D0BUsgiw
  - Aula 44 - POO: Cópia de objetos
  - https://youtu.be/69jsXNCrGjI
  - Aula 45 - POO: Herança
  - https://youtu.be/iEFQ_2_nTi0
  - Aula 46 - POO: Iterator
  - https://youtu.be/5wVWM5ReBrs