MC102 — Classes e Objetos

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

Atualizado em: 2023-05-15 15:57

Relembrando: Tipo

O tipo de um dado define:

- as operações que podemos fazer com ele
- e qual é o resultado

Ex:

- O que acontece ao somar um `int` com um `int`?
- O que acontece ao somar um `int` com um `float`?
- O que acontece ao somar uma `string` com um `string`?
- O que acontece ao somar uma `string` com um `int`?

Orientação a Objetos — Conceito

Um paradigma de programação (entre muitos outros) onde:

- Objetos armazenam dados como seus atributos
  - "Variáveis" que pertecem ao objeto
- Os objetos podem ser manipulados através de seus métodos
  - Funções que acessam ou modificam os atributos
- Objetos de uma Classe têm os mesmos atributos e métodos
  - Mas os valores dos atributos podem ser diferentes
  - A classe faz o papel do tipo
  - Ex: `[1, 2, 3]` e `[]` são objetos da classe `list`
    - Ambos respondem ao método `append`
- A computação é feita pela interação entre os vários objetos

Criando uma classe bem simples

```
1  class Estudante:  # Define a classe Estudante
2
3      # Define um método chamado __init__ que nos diz como
4      # inicializar o objeto. self é o próprio objeto.
5      def __init__(self, nome, RA, curso, nota):
6          self.nome = nome
7          self.RA = RA
8          self.curso = curso
9          self.nota = nota
10
11
12 # poderia ter usado parâmetros posicionais também
13 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
14 print(ana.nome, ana.RA, ana.curso, ana.nota)
```

Criando uma classe bem simples

```
1  class Estudante:  # Define a classe Estudante
2
3      # Define um método chamado __init__ que nos diz como
4      # inicializar o objeto. self é o próprio objeto.
5      def __init__(self, nome, RA, curso, nota):
6          self.nome = nome
7          self.RA = RA
8          self.curso = curso
9          self.nota = nota
10
11
12 # poderia ter usado parâmetros posicionais também
13 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
14 print(ana.nome, ana.RA, ana.curso, ana.nota)
```

Criamos um novo estudante escrevendo `Estudante(...)`

- Recebe um parâmetro a menos (o `self`)

Criando uma classe bem simples

```
1  class Estudante:  # Define a classe Estudante
2
3      # Define um método chamado __init__ que nos diz como
4      # inicializar o objeto. self é o próprio objeto.
5      def __init__(self, nome, RA, curso, nota):
6          self.nome = nome
7          self.RA = RA
8          self.curso = curso
9          self.nota = nota
10
11
12 # poderia ter usado parâmetros posicionais também
13 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
14 print(ana.nome, ana.RA, ana.curso, ana.nota)
```

Dizemos que:

- `ana` é um objeto da classe `Estudante`
- `ana` é uma instância de `Estudante`

Criando uma classe bem simples

```
1  class Estudante:  # Define a classe Estudante
2
3      # Define um método chamado __init__ que nos diz como
4      # inicializar o objeto. self é o próprio objeto.
5      def __init__(self, nome, RA, curso, nota):
6          self.nome = nome
7          self.RA = RA
8          self.curso = curso
9          self.nota = nota
10
11
12 # poderia ter usado parâmetros posicionais também
13 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
14 print(ana.nome, ana.RA, ana.curso, ana.nota)
```

Podemos acessar os atributos da instância usando o `.`

- i.e., `objeto.atributo`
- Para a leitura ou escrita

Um método para a classe Estudante

```
1  class Estudante:
2      def __init__(self, nome, RA, curso, nota):
3          self.nome = nome
4          self.RA = RA
5          self.curso = curso
6          self.nota = nota
7
8      def aprovado(self):              # define o método aprovado
9          return self.nota >= 5.0
10
11
12 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
13
14 if ana.aprovado():                  # chama o método aprovado
15     print(ana.nome, "está aprovado")
16 else:
17     print(ana.nome, "está reprovado")
```

Note que não passamos parâmetro para `ana.aprovado()`

- O Python já sabe que `self` é `ana`

Outro Método — Imprimindo o estudante

```
1  class Estudante:
2      def __init__(self, nome, RA, curso, nota):
3          self.nome = nome
4          self.RA = RA
5          self.curso = curso
6          self.nota = nota
7
8      def aprovado(self):
9          return self.nota >= 5.0
10
11     def imprime(self):
12         print("RA:", self.RA,
13               "Nome:", self.nome,
14               "Curso:", self.curso,
15               "Nota:", self.nota)
16
17
18 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
19 ana.imprime()
20 # RA: 123456 Nome: Ana Curso: 42 Nota: 10.0
```

O Python nos deixa fazer algo ainda mais legal do que isso...

Método __str__

O __str__ é chamado quando precisa converter para str!

```
1  class Estudante:
2      def __init__(self, nome, RA, curso, nota):
3          self.nome = nome
4          self.RA = RA
5          self.curso = curso
6          self.nota = nota
7
8      def aprovado(self):
9          return self.nota >= 5.0
10
11     def __str__(self):
12         return (f"RA: {self.RA} Nome: {self.nome}" +
13                 f"Curso: {self.curso} Nota: {self.nota}")
14
15
16 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
17 print(ana)
```

Ele é o que chamamos de método mágico

- E existem vários outros que podemos definir

Exercício

Crie uma classe `Turma` que:

- Armazena estudantes
- Permite adicionar estudantes
- Permite imprimir os estudantes
- Permite imprimir os estudantes aprovados
- Permite imprimir os estudantes reprovados

Encapsulamento

De maneira geral, é ruim escrevermos algo do tipo:

```
1 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
2 ...
3 ana.nota = 9.3
```

Isso porque estamos acessando o atributo diretamente

- `nota` pode ser apenas entre 0 e 10...
- Queremos alterar `nota` apenas através de um método!
- Chamamos isso de encapsulamento
  - Deveríamos acessar o objeto apenas pelos seus métodos
  - Já que os atributos são de sua responsabilidade

Primeira versão

```
1  class Estudante:
2      def __init__(self, nome, RA, curso, nota):
3          self.nome = nome
4          self.RA = RA
5          self.curso = curso
6          self.set_nota(nota)
7
8      def get_nota(self):
9          return self.nota
10
11     def set_nota(self, nota):
12         self.nota = nota
13
14
15 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
16 ana.set_nota(9.3)
```

Regras:

- Leitura deve ser feita pelo método `get_nota`
- Escrita deve ser feita pelo método `set_nota`

Mas ainda podemos escrever `ana.nota = 9.3`...

Segunda versão — Um passo atrás...

```
1  class Estudante:
2      def __init__(self, nome, RA, curso, nota):
3          self.nome = nome
4          self.RA = RA
5          self.curso = curso
6          self.nota = nota
7
8      @property  # isso é chamado de decorator em Python
9      def nota(self):
10         return self._nota
11
12     @nota.setter
13     def nota(self, nota):
14         self._nota = nota
15
16
17 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
18 ana.nota = 9.3
```

Voltamos a poder escrever `ana.nota = 9.3`

- Porém, a função da linha 13 é sempre chamada!
- E, se formos ler, a função da linha 9 é sempre chamada!

Terceira versão — Nota inválida

```
1  class Estudante:
2      def __init__(self, nome, RA, curso, nota):
3          self.nome = nome
4          self.RA = RA
5          self.curso = curso
6          self.nota = nota
7
8      @property
9      def nota(self):
10         return self._nota
11
12     @nota.setter
13     def nota(self, nota):
14         if nota < 0 or nota > 10:
15             raise ValueError("Nota inválida!")
16         self._nota = nota
17
18
19 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
20 ana.nota = 10.3
```

Observações

- Você poderia ter apenas a função de leitura (`@property`)
- Em geral, a boa pratica seria que todos os atributos fossem privados e acessados apenas por funções
- Mas você não precisa se preocupar com isso nessa disciplina...
- O _ pode ser usado para métodos também
  - Indica que o método não deve ser chamado de fora
  - Pode ser um cálculo parcial, por exemplo
  - Isto é, um método auxiliar

Exercícios

1. Refaça a classe Turma usando encapsulamento
2. Refaça a classe Estudante usando encapsulamento de forma que a nota do estudante é a média aritmética de três notas

Dataclasses

É comum termos uma classe apenas para guardar dados. Ex:

```
1  class Estudante:
2      def __init__(self, nome, RA, curso, nota):
3          self.nome = nome
4          self.RA = RA
5          self.curso = curso
6          self.nota = nota
7
8      def aprovado(self):
9          return self.nota >= 5.0
10
11     def __str__(self):
12         return (f"RA: {self.RA} Nome: {self.nome}" +
13                 f"Curso: {self.curso} Nota: {self.nota}")
```

Para simplificar a escrita, o Python adicionou as `dataclasses`

Dataclasses

Código usando dataclass

```
1  from dataclasses import dataclass
2
3
4  @dataclass  # decorador para a classe estudante
5  class Estudante:
6      nome: str
7      RA: int
8      curso: int
9      nota: float
10
11     def aprovado(self):
12         return self.nota >= 5.0
13
14
15 ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
16 print(ana)
```

É impresso `Estudante(nome='Ana', RA=123456, curso=42, nota=10.0)`

Dataclasses

Basicamente usando dataclasses você ganha várias coisas:

- Um método `__init__`
- Um método `__repr__`
- Um método `__eq__`
- Outros métodos e comportamentos dependendo da opções passada para o decorador
  - Ex: `@dataclass(order=True)`
- Há também como ter valores padrão para atributos
  - Porém precisa ter alguns cuidados

Sugestão de leitura:
`https://realpython.com/python-data-classes`

Exercícios

1. Refaça a classe Turma usando dataclass

   Dica para não ter problemas:
   - `from dataclasses import dataclass, field`
   - defina a lista de estudantes na turma da seguinte forma:
     `estudantes: list = field(default_factory=list)`

Herança

Outro conceito comum em orientação a objetos é Herança

Muitas vezes temos o conceito de "é um":

- Estudante de Graduação é um Estudante
- Estudante de Pós-Graduação é um Estudante
- Quadrado é um Retângulo que é um Paralelogramo
- Inteiro é um Racional que é um Real

E isso nos permite reutilizar códigos:

- A classe filha herda métodos e atributos da classe mãe

A classe Retângulo

```
1  class Retangulo:
2      def __init__(self, largura, altura):
3          self._largura = largura
4          self._altura = altura
5
6      def area(self):
7          return self._largura * self._altura
8
9      def perimetro(self):
10         return 2 * self._largura + 2 * self._altura
11
12     def __str__(self):
13         return ("Retângulo " + str(self._largura)
14                 + "x" + str(self._altura))
15
16
17 r = Retangulo(10, 3)
18 print(r)
19 print(r.area(), r.perimetro())
```

Será impresso:

```
1 Retângulo 10x3
2 30 26
```

A classe Quadrado

```
1  class Quadrado(Retangulo):  # Quadrado herda de Retangulo
2      def __init__(self, lado):
3          super().__init__(lado, lado)  # super é o objeto mãe
4
5      @property
6      def lado(self):
7          return self._largura  # lemos da classe mãe
8
9      @lado.setter
10     def lado(self, lado):
11         self._largura = lado  # alteramos na classe mãe
12         self._altura = lado
13
14     def __str__(self):  # Estamos sobrescrevendo __str__
15         return "Quadrado de lado " + str(self.lado)
16
17
18 q = Quadrado(4)
19 print(q)
20 print(q.area(), q.perimetro())
```

Será impresso:

```
1 Quadrado de lado 4
2 16 16
```

Exercício

Faça uma classe `EstudanteDePos`, considerando que esses estudantes recebem conceito **A**, **B**, **C**, ou **D**, dependendo de sua nota final

Comentando classes

A ideia é a mesma de comentar funções

```
1  class Estudante:
2      """Representa um estudante com suas informações e nota.
3
4      Capaz de armazenar o nome, RA, curso e a nota do estudante.
5      """
6
7      def __init__(self, nome, RA, curso, nota):
8          self.nome = nome
9          self.RA = RA
10         self.curso = curso
11         self.nota = nota
12
13     @property
14     def nota(self):
15         """Nota do estudante.
16
17         A nota deve estar entre 0 e 10.
18         """
19         return self._nota
```

`id` de Objetos

Cada objeto no Python tem um único identificador

- Pode ser acessado pela função `id`

Ex:

```
1 r1 = Retangulo(10, 3)
2 r2 = Retangulo(3, 3)
3 print(r1, r2)
4 print("Mesma id?", id(r1) == id(r2))
```

Será impresso:

```
1 Retângulo 10x3 Retângulo 3x3
2 Mesma id? False
```

Os retângulos são objetos diferentes... e se executarmos:

```
1 r1.largura = 7
2 print(r1, r2)
```

Será impresso:

```
1 Retângulo 7x3 Retângulo 3x3
```

`id` de Objetos

Outro exemplo:

```
1 r1 = Retangulo(10, 3)
2 r2 = Retangulo(10, 3)
3 print(r1, r2)
4 print("Mesma id?", id(r1) == id(r2))
5 r1.largura = 7
6 print(r1, r2)
```

Será impresso:

```
1 Retângulo 10x3 Retângulo 10x3
2 Mesma id? False
3 Retângulo 7x3 Retângulo 10x3
```

Isto é, os retângulos continuam sendo objetos diferentes...

- Cada chamada de `Retangulo(10, 3)` criou um novo objeto

`id` de Objetos

Mais um exemplo:

```
1 r1 = Retangulo(10, 3)
2 r2 = r1
3 print(r1, r2)
4 print("Mesma id?", id(r1) == id(r2))
5 r1.largura = 7
6 print(r1, r2)
```

Será impresso:

```
1 Retângulo 10x3 Retângulo 10x3
2 Mesma id? True
3 Retângulo 7x3 Retângulo 7x3
```

Isto é, `r1` e `r2` são o mesmo objeto!

- Dizemos que eles referenciam o mesmo objeto

`id` de Objetos

Um último exemplo:

```
1 r1 = 10
2 r2 = 10
3 print(r1, r2)
4 print("Mesma id?", id(r1) == id(r2))
5 r1 = 15
6 print(r1, r2)
```

Será impresso:

```
1 10 10
2 Mesma id? True
3 15 10
```

A constante `10` existe apenas uma vez

- Por isso `id(r1) == id(r2)`
- Mas `r1` passou a referenciar outro objeto
  - Note que atribuímos para `r1`
  - Não usamos um método do objeto
- O `10` que o `r2` referencia continua o mesmo

Mas, e daí?

Daí que isso explica porque funções alteram listas!

- E objetos em geral!

Exemplo:

```
1 def altera(lista):  # lista referencia [1, 2, 3]
2     lista.clear()  # método que remove os elementos da lista
3     print(id(lista))
4
5
6 lista = [1, 2, 3]
7 print(id(lista))
8 altera(lista)
9 print(lista)
```

Será impresso:

```
1 4547883488
2 4547883488
3 []
```

Mas, e daí?

Daí que isso explica porque funções alteram listas!

- E objetos em geral!

Exemplo:

```
1 def nao_altera(lista):  # lista referencia [1, 2, 3]
2     lista = []  # lista passa a referenciar []
3     print(id(lista))
4
5
6 lista = [1, 2, 3]
7 print(id(lista))
8 nao_altera(lista)
9 print(lista)
```

Será impresso:

```
1 4550516976
2 4547883488
3 [1, 2, 3]
```

Mas, e daí?

Daí que isso explica porque funções alteram listas!

- E objetos em geral!

Exemplo:

```
1 def nao_altera2(x):
2     x = 0
3     print(id(x))
4
5
6 x = 10
7 print(id(x))
8 nao_altera2(x)
9 print(x)
```

Será impresso:

```
1 4546739392
2 4546739072
3 10
```

Objetos Mutáveis e Imutáveis

Objetos mutáveis são aqueles que podem ser alterados

- Através de mudanças em seus atributos
- Através de chamadas de método
- Ex: `list`, `dict`, `set`

Estes objetos podem ser alterados chamadas de funções

Objetos imutáveis não podem ser alterados

- Não há como acessar os atributos
- Não há métodos que alteram o objeto
- Ex: `int`, `float`, `bool`, `None`, `tuple`

Métodos de Classe e Métodos Estáticos

Uma classe pode ter métodos que podem ser acessados sem termos um objeto

Esses métodos são de dois tipos

- Métodos de Classe: sabem quem é a classe
  - Muitas vezes são formas de construir um objeto
  - Ex: `datetime.date.today()`
  - Ex: `datetime.date.fromisoformat('2022-12-25')`
- Métodos Estáticos: não sabem quem é a classe
  - Em geral são métodos utilitários
  - Não são tão usados

Métodos de Classe e Métodos Estáticos

```
1  @dataclass
2  class Retangulo:
3      largura: int
4      altura: int
5
6      @classmethod  # decorador de método de classe
7      def da_lista(cls, lista):  # cls é a classe
8          return cls(lista[0], lista[1])
9
10     @staticmethod  # decorador de método estático
11     def formula_da_area():
12         return "base x altura"
13
14     def area(self):
15         return self._largura * self._altura
16
17
18 lista = [2, 3]
19 r = Retangulo.da_lista(lista)
20 s = Retangulo.formula_da_area()
```

- `r` é um `Retangulo` 2×3
- `s` é a string `"base x altura"`