Algoritmos Computacionais

Conteúdo: Sobrecarga de métodos - polimorfismo

Prof. Dsc. Giomar Sequeiros
giomar@eng.uerj.br

Função super

Função super()

- A função super em Python nos fornece a facilidade de referir-se **explicitamente** à classe **pai**.
  - É útil quando temos que **chamar funções** de **superclasse**.
- A função super() é usada na classe filha com herança múltipla para acessar a função da próxima classe pai ou superclasse.
- Se a classe for uma classe de herança única, a função super() é útil para usar os métodos das classes pai sem usar seus nomes explicitamente.

Uso correto da função super()

- Considere o seguinte código:

Herança simples

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
```

- Testando:

```
>>> base = Base()
Base.__init__
>>> a = A()
Base.__init__
A.__init__
```

Uso correto da função super()

- Considere o seguinte código:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

Herança simples

Herança múltipla

Testando:

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
```

O método Base.__init__() é chamado duas vezes

Uso correto da função super()

- A forma correta de escrever seria usando a função super():

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__() # Apenas uma chamada de super()
        print('C.__init__')
```

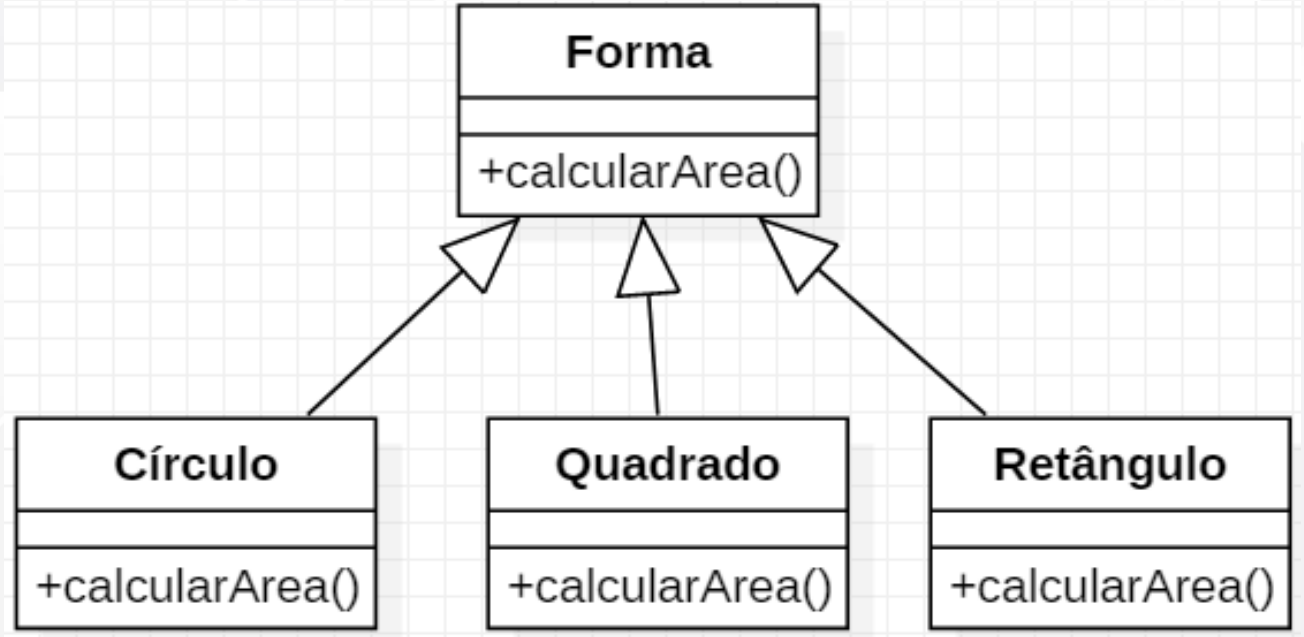
Testando:

```
>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
```

Polimorfismo

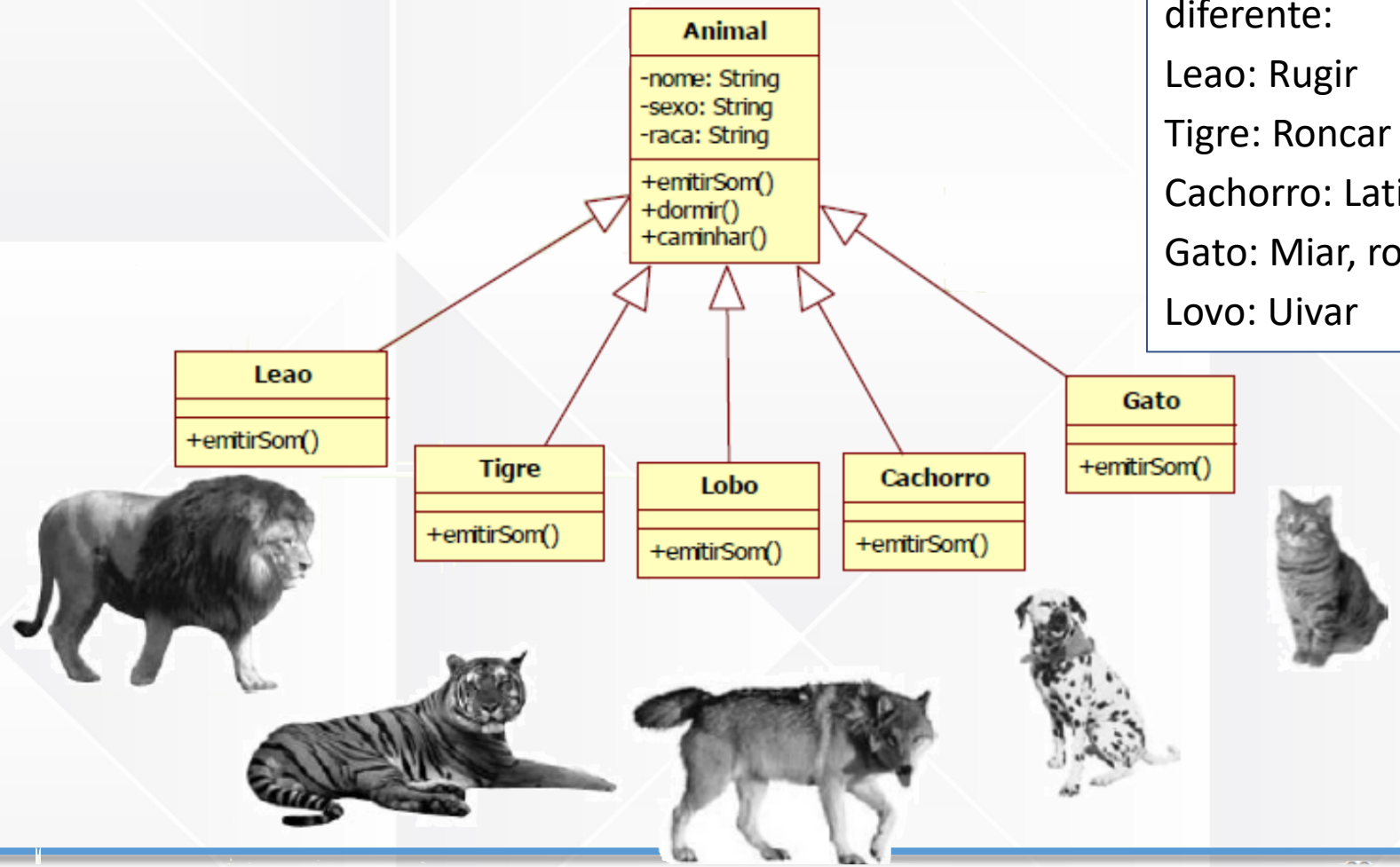
Polimorfismo

"O polimorfismo ocorre quando um método que já foi definido no ancestral é redefinido no descendente com um comportamento diferente."



Poli→várias;
Morfos→formas

Polimorfismo: exemplo 1



Cada animal emite sons diferente:

Leao: Rugir

Tigre: Roncar

Cachorro: Latir

Gato: Miar, rosnar

Lovo: Uivar

Polimorfismo: exemplo 2

- Considere a classe Funcionario e a classe Gerente subclasse de Funcionario

```
class Funcionario:
    def __init__(self, nome, cpf, salario, depto):
        self.nome = nome
        self.cpf = cpf
        self.salario = salario
        self.depto = depto
```

```
class Gerente(Funcionario):
    def __init__(self, nome, cpf, salario, depto, senha, numFunc):
        super().__init__(nome, cpf, salario, depto)
        self.senha = senha
        self.numFunc = numFunc
```

Polimorfismo: exemplo 2

- Todo fim de ano, os funcionários da empresa recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.
- Vamos ver como fica a classe Funcionario:

```
class Funcionario:
    def __init__(self, nome, cpf, salario, depto):
        self.nome = nome
        self.cpf = cpf
        self.salario = salario
        self.depto = depto

    def get_bonificacao(self):
        return self.salario * 0.10
```

Teste da classe funcionario

```
gerente = Gerente('José', '222222222-22', 5000.0, '1234', 0)
print(gerente.get_bonificacao())
```

A saida será 500, o qual está errado

Polimorfismo: exemplo 2

- Todo fim de ano, os funcionários da empresa recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.
- Vamos ver como fica a classe Gerente:

```
class Gerente(Funcionario):
    def __init__(self, nome, cpf, salario, depto, senha, numFunc):
        super().__init__(nome, cpf, salario, depto)
        self.senha = senha
        self.numFunc = numFunc

    def get_bonificacao(self):
        return self.salario * 0.15
```

Teste

```
gerente = Gerente('José', '222222222-22', 5000.0, '1234', 0)
print(gerente.get_bonificacao())
```

A saida será 750

Polimorfismo: exemplo 2

- Imagine que para calcular a bonificação de um Gerente, devemos fazer igual ao cálculo de um Funcionario, adicionando 1000.0 reais. Poderíamos fazer assim:

```
class Gerente(Funcionario):
    def __init__(self, nome, cpf, salario, depto, senha, numFunc):
        super().__init__(nome, cpf, salario, depto)
        self.senha = senha
        self.numFunc = numFunc

    def get_bonificacao():
        return self.salario * 0.10 + 1000.0
```

- Aqui teríamos um problema: o dia que o get_bonificacao() do Funcionario mudar, precisaremos mudar o método do Gerente para acompanhar a nova bonificação.

Polimorfismo: exemplo 2

- Para evitar isso, o get_bonificacao() do Gerente pode chamar o do Funcionario utilizando o método super().

```
class Gerente(Funcionario):
    def __init__(self, nome, cpf, salario, depto, senha, numFunc):
        super().__init__(nome, cpf, salario, depto)
        self.senha = senha
        self.numFunc = numFunc

    def get_bonificacao():
        return super().get_bonificacao() + 1000
```

- Essa invocação vai procurar o método com o nome get_bonificacao() de uma superclasse de Gerente. No caso, ele logo vai encontrar esse método em Funcionario.

Sobrecarga de Operadores

Sobrecarga de Operadores

- Em Python, o comportamento dos operadores é definido por métodos especiais, que por convenção, têm nomes que começam e terminam com “__” (duplo underline).
- Exemplos:

Operador	Método	Operação
+	__add__	Adição
-	__sub__	Subtraçao
*	__mul__	Multiplicação
**	__pow__	Potência
/	__div__	Divisão

Sobrecarga de Operadores

- Operadores como +, – e * podem ser aplicados a objetos como inteiros e strings gerando diferentes resultados dependendo do tipo do objeto.
- O sinal **+** aplicado a dois **inteiros** resulta na sua **soma**, enquanto sobre **strings** resulta na sua **concatenação**.
- Podemos implementar o **comportamento** deste **operador** sobre duas **instâncias** definindo-o dentro de sua classe.

Exemplo 1

Considere a classe inteiro com atributo privado __valor.

```
class Inteiro():
    def __init__(self, entrada):
        self.__valor = entrada

    @property
    def valor(self):
        return self.__valor

    def __add__(self, outro):
        return self.valor + outro.valor
```

A saída será:

```
>>> a = Inteiro(5)
>>> b = Inteiro(7)
>>> print(a + b)
12
```

Exemplo 2

Crie uma classe para operar com frações. Uma fração são dois inteiros na forma de a/b, sendo b um inteiro diferente de zero. Vamos criar uma classe para representar as frações.

```
class Fracao:
    def __init__(self, numerador, denominador):
        self.__numerador = numerador
        self.__denominador = denominador

    def __str__(self):
        return str(self.__numerador) + '/' + str(self.__denominador)
```

Sobrecarga do comando de impressão:

Testamos:

```
>>> a = Fracao(1,2)
>>> print(a)
1/2
```

Exemplo 2

Adicione as propriedades (getter) para os atributos numerador e denominador

```
    @property
    def numerador(self):
        return self.__numerador

    @property
    def denominador(self):
        return self.__denominador
```

Exemplo 2

Acrescente o método multiplicar

```
    def __mul__(self, outro):
        numerador = self.numerador * outro.numerador
        denominador = self.denominador * outro.denominador
        return Fracao(numerador, denominador)
```

Sobrecarga do comando de multiplicar

Testamos:

```
>>> a = Fracao(1,2)
>>> b = Fracao(3,4)
>>> c = a*b
>>> print(c)
3/8
```

Sobrecarga de Operadores

- Alguns outros operadores que podem ser sobrecarregados:

Operador	Método	Operação
//	__floordiv__	Divisão truncada
%	__mod__	Módulo
+	__pos__	Positivo
-	__neg__	Negativo
<	__lt__	Menor que
>	__gt__	Maior que
<=	__le__	Menor ou igual a
>=	__ge__	Maior ou igual a
==	__eq__	Igual a
!=	__ne__	Diferente de
print	__str__	Impressão

Exercício 1

Complete o exemplo da Fração. Defina métodos para **adição**, **subtração**, **divisão** e **impressão** de frações. No construtor crie uma validação para verificar tentativas de criar fração com denominador com valor zero. Nesse caso, não permita e imprima a mensagem “Denominador inválido!”.

Exercício 2

Desenvolva uma classe para trabalhar com **números complexos**, na qual estejam definidas as quatro **operações básicas** com este conjunto numérico. Sobrescreva também o __str__ para imprimir o número na forma de complexo.

Adição $(a+bi)+(c+di)=(a+c)+(b+d)i$

Subtração $(a+bi)-(c+di)=(a-c)+(b-d)i$

Multiplicação $(a+bi)\times(c+di)=(ac-bd)+(ad+bc)i$

Divisão

$$\frac{a+bi}{c+di}=\frac{ac+bd}{c^2+d^2}+\frac{bc-ad}{c^2+d^2}i$$