Paradigma Orientado a Objetos

Polimorfismo e Sobrecarga

ERICK GALANI MAZIERO
erick.maziero@ufla.br

Departamento de Ciências da Computação
Universidade Federal da Lavras

Polimorfismo

- Habilidade de apresentar a mesma interface para formas diferentes, tipos diferentes de dados
  - Por exemplo, métodos polimórficos podem aceitar tipos de dados diferentes e, dependendo dos mesmos, ter comportamentos diferentes

```
def f(x, y):
    print("values: ", x, y)

f(42, 43)
f(42, 43.7)
f(42.3, 43)
f(42.0, 43.9)
f([1,2,3], [6,5,4,3,4,7])
f({"A":65, "B":66}, {"C":67, "D":68})
```

O método f recebe dois parâmetros e simplesmente os exibe na tela. Por não ser uma linguagem tipada (Python). Esse método pode ser chamado com diversos tipos de argumentos: inteiros, ponto-flutuante, listas e dicionários

Polimorfismo

Habilidade de apresentar a mesma interface para formas diferentes, tipos diferentes de dados

- Por exemplo, métodos polimórficos podem aceitar tipos de dados diferentes e, dependendo dos mesmos, ter comportamentos diferentes

Em C++ para o polimorfismo, é necessária a definição de diversos métodos, um para cada configuração de tipos de argumentos

```
#include <iostream>
using namespace std;

void f(int x, int y ) {
    cout << "values: " << x << ", " << x << endl;
}

void f(int x, double y ) {
    cout << "values: " << x << ", " << x << endl;
}

void f(double x, int y ) {
    cout << "values: " << x << ", " << x << endl;
}

void f(double x, double y ) {
    cout << "values: " << x << ", " << x << endl;
}


int main()
{
    f(42, 43);
    f(42, 43.7);
    f(42.3,43);
    f(42.0, 43.9);
}
```

Sobrescrita, sobrecarga, substituição

Relacionadas ao polimorfismo, temos algumas variações são bem similares, veja:

- Sobrescrita
  - Método com mesmo nome, quantidade de parâmetros (e/ou tipos de parâmetros)
- Sobrecarga
  - Método com mesmo nome, mas quantidades e tipos de parâmetros diferentes
- Substituição
  - Quando uma subclasse reescreve um método definido na superclasse

Vamos focar na sobrecarga de métodos e operadores

Sobrecarga

- O mecanismo chamado de sobrecarga (*overloading*) é utilizando quando se deseja que dois métodos de uma mesma classe possam ter o mesmo nome, desde que suas listas de parâmetros sejam diferentes, constituindo assim uma assinatura diferente de cada método.
- A sobrecarga não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método.
- A sobrecarga é uma aplicação do polimorfismo na Orientação a Objetos.

Sobrecarga de operadores

```
>>> 4 + 5
9
 >>> 3.8 + 9
12.8
>>> "Peter" + " " + "Pan"
'Peter Pan'
>>> [3,6,8] + [7,11,13]
[3, 6, 8, 7, 11, 13]
```

Em Python, o operador ’+’ é utilizado tanto para somar inteiros e ponto-flutuante e concaternar strings e listas

Métodos *mágicos*, em Python

- Em Python, um inteiro, ponto-flutuante, string, lista, dicionários, etc. são tratados como objetos. Inclusive, essas ’classes base’ podem ser herdadas, como veremos mais adiante.
- Portanto, vários operadores pode ser sobrecarregados para operar em todos esse tipos de dados e, inclusive, sobre objetos criados por um programador.
- Isso é feito através dos métodos mágicos.
- Veja a tabela a seguir, que lista os operadores, tanto aritméticos quanto lógicos, e seus respectivos métodos mágicos para sobrecarga.

Sobrecarga de operadores

Operador	Método mágico
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Sobrecarga de operadores

Considere a classe Length, ao lado, e a sobrecarga do operador '+', pelo método __add__. Os métodos mágicos __str__ e __repr__ também foram utilizados

Assim, podemos instanciar um objeto da classe Length. Ao printar o objeto o método __str__ é invocado

```
x = Length(4)
print(x)
y = eval(repr(x))

z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

```
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }

    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"
```

Sobrecarga de operadores

Ao utilizar a função repr, o método __repr__ é invocado e retorna uma String que, quando passada para a função eval, causa a instanciação de um novo objeto

```
x = Length(4)
print(x)
y = eval(repr(x))

z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

```
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }

    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"
```

Sobrecarga de operadores

Com a sobrecarga do operador '+', pelo método __add__, pode-se somar dois objetos da classe Length, como se soma dois número inteiros ou em ponto-flutuante

```
x = Length(4)
print(x)
y = eval(repr(x))

z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

```
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }

    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"
```

Atributos e métodos de um objeto

Considere a classe Conta, com um construtor e um método set_saldo, que lança uma exceção caso se atribua saldo negativo.

```
class Conta:
    def __init__(self):
        self.__saldo = 0
    def set_saldo(self, saldo):
        if saldo < 0:
            raise Exception('Error')
        self.__saldo = saldo
```

Instanciação de um objeto da classe Conta.

```
c1 = Conta()
```

Acesso direto a um atributo privado da classe Conta. Nesse caso não se utilizará a função set_saldo. Como evitar isso?

```
c1._Conta.__saldo = 100000000
c1.titular = 'Fulano Ciclano'
```

Atributos e métodos de um objeto

Considere a classe Conta, com um construtor e um método set_saldo, que lança uma exceção caso se atribua saldo negativo.

```
class Conta:
    def __init__(self):
        self.__saldo = 0
    def set_saldo(self, saldo):
        if saldo < 0:
            raise Exception('Error')
        self.__saldo = saldo
```

Instanciação de um objeto da classe Conta.

```
c1 = Conta()
```

Agora, adiciona-se um novo atributo a um objeto já instanciado. É possível isso em Java ou C++?

```
c1._Conta.__saldo = 100000000
c1.titular = 'Fulano Ciclano'
```

Definindo e limitando atributos

O uso de __slots__ faz a restrição dos atributos do objeto, não permitindo a criação de novos atributos

```
class Conta:
    __slots__ = [_Conta__saldo]
    def __init__(self):
        self.__saldo = 0
    def set_saldo(self, saldo):
        if saldo < 0:
            raise Exception('Error')
        self.__saldo = saldo
```

Não é permitida a criação de novos atributos nos objetos da classe Conta

```
c1 = Conta()
c1.new_saldo = 100000000 # Erro
```

O comando *dir()*

Retornará todos os atributos do objeto.
No exemplo, retornará:

```
['_Conta__saldo', '__class__',
  '__delattr__', '__dict__',
    '__dir__', '__doc__',
   '__eq__', '__format__',
'__ge__', '__getattribute__',
    '__gt__', '__hash__',
          '__init__',
'__init_subclass__', '__le__',
   '__lt__', '__module__',
    '__ne__', '__new__',
'__reduce__', '__reduce_ex__',
  '__repr__', '__setattr__',
   '__sizeof__', '__str__',
     '__subclasshook__',
 '__weakref__', 'set_saldo']
```

```
class Conta:
        def __init__(self):
            self.__saldo = 0
        def set_saldo(self, saldo):
            self.__saldo = saldo

c1 = Conta()
dir(c1)
```

Exercício:
Crie uma variável qualquer, x por exemplo, faça dir(x) e veja os atributos

O método mágico `__call__`

Permite que um objeto seja chamado diretamente, sem a necessidade de invocar um método específico

```python
class Polynomial:

    def __init__(self, *coefficients):
        self.coefficients = coefficients[::-1]

    def __call__(self, x):
        res = 0
        for index, coeff in enumerate(self.coefficients):
            res += coeff * x** index
        return res
```

```python
# a constant function
p1 = Polynomial(42)

# a straight Line
p2 = Polynomial(0.75, 2)

# a third degree Polynomial
p3 = Polynomial(1, -0.5, 0.75, 2)

for i in range(1, 10):
    print(i, p1(i), p2(i), p3(i))
```

os objetos da classe Polynomial são chamado diretamente.

Herança de classes base

As listas, em Python, têm o método pop(), mas não têm o método push(). Na realidade o método push é chamado de append(). Caso se queira que uma lista tenha explicitamente um método push, pode-se herdar a classe base de listas (list) e adicionar esse método

```
class Plist(list):

    def __init__(self, l):
        list.__init__(self, l)

    def push(self, item):
        self.append(item)
```

```
y = [3,4]
dir(y)
x = Plist([3,4])
x.push(47)
print(x)
dir(x)
```

Agora, pode-se chamar o método push() para adicionar um elemento à lista.

Referência Bibliográfica

Sebesta, R. W. (2011). *Conceitos de Linguagens de Programação*. 9 ed. Bookman.

Capítulo 12

Perkovic, L. *Introdução à Computação Usando Python - Um Foco no Desenvolvimento de Aplicações*. Editora LTC, 1. ed., 2016.

Capítulos 8 e 9.