Arthur Pellenz Winck
Paradigmas de Programação
19201620

Orientação a Objetos em Python

Orientação a objetos é um dos mais conhecidos paradigmas de programação, e diversas linguagens possuem suporte a ele. Nesse relatório, vamos entender como trabalhamos com Python e Orientação a Objetos.

1) Declaração e Instanciação de Classes

Para declararmos uma classe, temos que utilizar a keyword **class** e definirmos um nome.

```
class Pessoa:
```

Além disso, para que ela possa ser instanciada, é necessário que possua um construtor. O construtor é definido por um **método** da classe chamado **__init__**. Para definirmos um método dentro de uma classe, só precisamos criar uma função e passar como parâmetro o keyword **self**, que indica que esse método é ligado ao objeto que criamos a partir da classe.

Dentro do construtor declaramos variáveis que serão os **atributos** de um objeto da classe Pessoa. Para declarar esses atributos usamos também o keyword **self.**

```
class Pessoa():
    def __init__(self):
        self.nome = 'João'
        self.idade = 15
```

Para que possamos criar um Pessoa com qualquer nome ou idade, adicionamos mais parâmetros na função do construtor, dessa maneira:

```
class Pessoa():
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
```

Agora, vamos instanciar um objeto da classe Pessoa, para isso, temos que atribuir a uma variável o nome da classe seguido de parênteses, e dentro deles os possíveis parâmetros da classe.

Pessoa1 = Pessoa(‘arthur’,20)

Dessa forma criamos um Pessoa que possui arthur como nome e 20 como idade.

2) Herança (Simples e Múltipla) e Polimorfismo

Para criarmos uma relação de herança, precisamos de uma classe pai (superclasse) e uma classe filha (subclasse). A classe filha **herda** todos os atributos e métodos disponíveis da classe pai. Vamos criar uma classe chamada Funcionario, que herda de Pessoa. Para que a herança funcione corretamente, é necessário chamar da classe pai o seu método construtor com Pessoa.__init__()

```
class Funcionario(Pessoa):
        def __init__(self, nome, idade, funcao, salario):
                Pessoa.__init__(self,nome,idade)
                self.funcao = funcao
                self.salario = salario
```

E podemos criar um objeto da classe Funcionario, que é subclasse de Pessoa

func = Funcionario(‘Bruno’, 28, ‘Desenvolvedor’, 12155)

Esse foi um exemplo de herança simples, para a herança múltipla, uma subclasse herda características de duas superclasses, dessa maneira

```
class A:
   def a(self):
      print('Eu sou da classe A')

class B:
   def b(self):
      print('Eu sou da classe B')


class C(A,B):
   def c(self):
      print('Eu sou da classe C')


exemplo = C()
exemplo.a()
exemplo.b()
exemplo.c()
```

Dessa maneira, quando criamos um objeto e o atribuímos a variável exemplo, esse objeto possui todos os métodos tanto das classes A e B quanto C. Assim, é possível usufruir de herança múltipla em Python.

Uma característica importante da relação de herança é a capacidade de realizar polimorfismo. Polifmorfismo é a possibilidade de dois métodos com a mesma assinatura realizarem operações diferentes dependendo de qual é o objeto que os chama.

```
class Pessoa():
        def __init__(self, nome, idade):
                self.nome = nome
                self.idade = idade

        def status(self):
                print("Apenas um cliente da loja, não tem acesso as dependências")

class Funcionario(Pessoa):
        def __init__(self, nome, idade, funcao, salario):
                Pessoa.__init__(self,nome,idade)
                self.funcao = funcao
                self.salario = salario

        def status(self):
                print("Um funcionário com acesso às dependências da loja")
```

Nesse exemplo, criamos um método chamado status() que indica se uma pessoa possui ou não acesso às dependências da loja, onde apenas os funcionários podem passar.

```
funcionario1 = Funcionario('Jorge', 30, 'Gerente', 4000)
pessoa1 = Pessoa('Sabino', 44)

funcionario1.status()
pessoa1.status()
```

Agora, chamando dois métodos que possuem a mesma assinatura, vemos que eles dão resultados diferentes, por conta de qual objeto que os chama.

3) Composição e/ou Agregação

Composição é a relação entre duas classes onde uma classe só existe se existir outra classe, em uma relação de posse.

```
class Coracao:
   def __init__(self):
```

```
        pass

class Pessoa:
    def __init__(self):
        self.coracao  = Coracao()

pessoa1 = Pessoa()
```

Dessa forma, um coração só existe quando um objeto da classe Pessoa for instanciado.

Agregação é uma relação mais fraca que a composição, já que as duas classes podem existir. Usando o mesmo exemplo, a agregação seria dessa maneira:

```
class Coracao:
    def __init__(self):
        pass

class Pessoa:
    def __init__(self, coracao):
        self.coracao  = coracao

coracao1 = Coracao()
pessoa1 = Pessoa(coracao1)
```

4) Métodos Abstratos e Estáticos

Até agora os métodos eram escritos como nome_método(self), indicando que eram ligados ao objeto criado a partir de uma classe. Porém, existem momentos em que gostaríamos que os métodos não estivessem ligados a um objeto, mas sim uma classe, esse é o caso do método estático.

```
class Calculadora:

    def multiplcar(x,y):
        print(x*y)

Calculadora.multiplcar(7,8)
```

Outro ponto é que chamamos a **classe** Calculadora, e não um objeto, já que não precisamos criar um objeto para invocar métodos estáticos.

Métodos abstratos servem para estruturar classes sem precisarmos nos preocupar em realmente implementar os métodos. Assim é possível criar modelos do que será implementado em outras classes.

```
from abc import ABC, abstractmethod
```

```
class Calculadora(ABC):

    @abstractmethod
    def integrar(funcao):
        pass

    def multiplcar(x,y):
        print(x*y)

class CalculadoraCientifica(Calculadora):

    def integrar(funcao):
        print("O resultado é...")


Calculadora.multiplcar(7,8)

CalculadoraCientifica.integrar("x+5")
```

É necessário importar uma biblioteca, a abc, e usar os módulos ABC (Classe Abstrata) e abstractmethod, um decorador que indica que o método é abstrato.