Programação Orientada a Objetos

Linguagens e Paradigmas de Programação
Sistemas de Informação
Alexandre de Sá Carneiro Wanderley

Caique Galdino de Lima [1]
Francisco Lucas do Nascimento [2]
Marcelo de Lima Freire [3]
Renan Gustavo Carvalho Menezes [4]

Programação Orientada a Objetos com Python

[1] caiquegamer0@gmail.com
[2] lucasnasm@gmail.com
[3] marcelodelima.m@gmail.com
[4] renanzx@live.com

Sumário

Resumo: Nesta pesquisa, apresenta-se um breve histórico, conceitos e exemplos de aplicações que seguem o paradigma de programação orientado a objetos (OO). Para tanto, objetiva-se investigar como o paradigma OO aplica-se à linguagem de programação *Python*, relacionando os conceitos desse paradigma com aplicações prática construídas com *Python*. Trata-se de um estudo exploratório descritivo, com um levantamento sobre o assunto na literatura, buscando identificar os principais autores que exploraram o tema, e realizar uma abordagem técnica sobre esse paradigma.

Palavras-chave: Paradigma. Programação. Python. Computação.

1 Introdução

O crescente desenvolvimento de software e a exigência do mercado por aplicações cada vez mais complexas, levou a uma expressiva necessidade por metodologias que possibilitassem abstrair e modularizar as estruturas dos programas existentes. Assim, com base nessas necessidades, surge o paradigma de Orientado a Objetos focado em atender essas exigências. Dentre as linguagens que suportam o paradigma orientação a objetos, tem-se a linguagem Simula (marco inicial da Orientação a objetos), C++, Java, Python, PHP, Ruby, Pascal, etc.

2 Histórico

Em 1967, Kristen Nygaard e Ole-Johan Dahl, do Centro Norueguês de Computação, em Oslo, desenvolveram a linguagem Simula. Derivada do Algol, o Simula I e Simula 67 podem ser consideradas as primeiras linguagens a introduzir conceitos de orientação a objetos. Em princípio, eram usadas para realizar simulações do comportamento de partículas de gases.

Os conceitos de objetos, classes e herança nesse estágio de desenvolvimento desse paradigma, eram tratados não necessariamente da forma que se conhece hoje, por exemplo, o conceito de herança surgiu no Simula 67, pois até então falava-se apenas em *subclassing* [1]. Em 1970, Alan Kay, Dan Ingalls e Adele Goldberg, do Centro de Pesquisa da Xerox, desenvolveram a linguagem essa linguagem totalmente orientada a objetos.

Em 1979–1983, Bjarne Stroustrup, no laboratório da AT & T, desenvolveu a linguagem de programação C++, uma evolução da linguagem C, passando a suportar o paradigma orientado a objetos.

A partir de 1983 com o C++, ObjetC, 86 objectpascal java c e objectivec surgem como linguagens com suporte a OO. Um *slogan* que consolidou esse paradigma "escrava uma vez execute em qualquer lugar"proferido pela *Sun MicroSystems* como uma forma para promover o Java. A partir desse período a POO passa a ganhar mais espaço nos ambientes de desenvolvimento tornando-se amplamente adotada por grupos desenvolvedores.

3 Conceitos

Neste capítulo, tem-se uma abordagem sobre os principais conceitos e características, que cercam a programação orientada a objetos, juntamente com uma explanação sobre os pilares da programação orientada a objetos. Cabe ser colo-

cado que, as definições serão exemplificadas com aplicações implementadas com a linguagem. Ao final serão apresentadas as conclusões desse trabalho.

3.1 Abstração

A abstração pode ser compreendida como a capacidade que uma linguagem tem de permitir ao desenvolvedor omitir detalhes da implementação que não tem relevância para o que ele está desenvolvendo. Ex: para um programador desenvolvem um sistema de cadastro de carros de uma concessionária (imagine se fosse necessário implementar todos os elementos que compõem um carro), muitas informações não serão relevantes constar no cadastro (e.g., quantos parafusos prendem o motor? qual o tamanho desses parafusos? que tipo de polímero é usado no plástico que reveste a direção?). Esses são detalhes importantes para o fabricante, mas não são tão importantes para o vendedor.

Para usar um determinado objeto, não é necessário compreender todas as operações que este pode desempenhar ou mesmo como o objeto é representado [4]. Com isso, a linguagem que incorpora esse conceito tem que permitir ao programador olhar para o problema ou objeto real (e.g., carro) e abstrair do carro informações que não são relevantes para a finalidade da aplicação que ele está desenvolvendo.

Portanto, uma característica da abstração, é omitir, na linguagem, informações que não são relevantes para aquela operação. Com isso, a classe é modelada de acordo com a necessidade do negócio.

A criação de uma classe abstrata é bastante útil e serve para definir o esqueleto para uma subclasse. Contudo, em Python, a implementação de classes abstratas são definidas por meio da biblioteca ABC. A seguir, é apresentada uma simples e eficiente forma de uso do biblioteca padrão abc do Python 3.6. Cabe ser observado que, esse módulo foi adicionado ao Python 2.6 definido na proposta: PEP 3119. Segue exemplo abaixo.

```
1  from abc import ABC, abstractmethod
2
3  class AbstractOperation(ABC):
4
5          def __init__(self, operand_a, operand_b):
6                  self.operand_a = operand_a
7                  self.operand_b = operand_b
8                  super(AbstractOperation, self).__init__()
9
10         @abstractmethod
11         def execute(self):
12                 pass
```

A seguir é apresentado um exemplo mais detalhado envolvendo a passagem

```
from abc import ABC, abstractmethod

class AbstragtClass(ABC):
        def __init__(self, value):
                self.value = value


class Adulto(AbstractClass):
        def eat(self):
                return "Ingerir comidas sólidas "+ str(self.value) + " vezes ao
                    dia"


class Babies(AbstractClass):
        def eat(self):
                return "Apenas leite "+ str(self.value) + " vezes ou mais ao
                   dia"
```

3.2 Encapsulamento

Encapsulamento é a proteção dos atributos ou métodos de uma classe assim evitando que dados específicos de uma aplicação possam ser acessados diretamente.

Encapsular é fundamental para que seu sistema seja suscetível a mudanças: já que não é preciso mudar métodos e atributos em vários lugares, mas sim em apenas um único lugar, já que o código está encapsulado. Em Python existem dois tipos de modificadores de acesso para atributos e métodos: Público e Privado. Atributos ou métodos iniciados por dois sublinhados são privados e todas as outras formas são públicas.

```
class Funcionario():
        def metodo(self,valor):
                self.__atributo=valor
```

Uma maneira de ter acesso a esses atributos é utilizados métodos chamados de Getters e Setters.

```
class Funcionario():
        def setMetodo(self,valor):
                self.__atributo=valor

        def getMetodo(self):
                return self.__atributo
```

3.3 Herança

Assim como as características biológicas são passadas dos pais para os filhotes no mundo animal, como por exemplo os dentes caninos do grupo dos canídeos,

as classes em Python tem um comportamento semelhante, onde os atributos podem ser herdados de outras classes. Mas por que se faz necessário o uso de herança? imagine uma classe chamada funcionário repleta de atributos, centenas deles a serem definidos, como nome, matrícula, idade, sexo, salário, setor, etc. etc. Agora imagine que você precisa escrever várias classes semelhantes a essa(mesmos atributos). Para cada classe criada é necessário escrever cada atributo um por um. E se você esquecer algum? E se esquecer um método de alguma? E é aí que a herança entra em ação. Ao invés de escrever atributo por atributo em cada classe, você precisa apenas escrever uma classe e herdar os atributos para as classes semelhantes. Por exemplo, uma classe Funcionário tem todos os atributos padrão de um funcionário, então não preciso reescrever esses atributos em uma classe Secretária. Apenas herdar-los e escrever os atributos próprios dela, se preciso. Em Python quando uma classe tem seus atributos herdados por outras, essa é chamada de superclasse (ou "pai","mestre"ou "base"), as que herdam esses atributos são então chamadas de subclasses (ou "filha", ou "derivada"). Uma subclasse pode herdar os atributos herdados da sua superclasse. Confuso né? veja o exemplo abaixo.

```
1 class Funcionario:
2         atributos1....
3 class Supervisor(Funcionario):
4         atributos2.....
5 class Gerente(Supervisor):
6         atributos3.....
```

Quando queremos herdar de outra classe, utilizamos o parêntese para indicar a classe da qual estamos pegando os atributos, ou seja, a nossa superclasse. No exemplo acima a classe Supervisor herda atributos1 de Funcionario, e a classe Gerente herda atributos1 e atributos2 de Supervisor. Vimos que a herança pode "copiar" os atributos de uma classe para outra, mas não para por aí. Ela funciona com os métodos também, então todos os métodos da nossa superclasse serão herdados pelas subclasses dela e poderão ser utilizados sem nenhum problema. Por exemplo, uma subclasse chamada Fiat, pode herdar os métodos de sua superclasse ModeloCarro que tem os métodos ligar e acelerar. Para aprofundar mais seus conhecimentos sobre orientação a objetos em python, consulte o material de apoio do curso "Introdução a python" módulo B, por Josué Labaki e E. R. Woiski.

3.4 Polimorfismo

O polimorfismo é uma palavra que deriva do Grego que significa "algo que assume muitas formas"[2]. Um dos principais objetivos da programação orientada à objetos é prover a reutilização de código, no Python, podemos chamar isso de herança e polimorfismo, onde através de uma classe filha, podemos herdar e sobrescrever métodos ou não de uma classe pai. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de

um objeto pai. No Python podemos utilizar essa técnica da seguinte forma

```
class Mamifero:
        def som(self):
                return "Emiti som"


class Homem(Mamifero):
        def som(self):
                return "oi"


class Cachorro(Mamifero):
        def som(self):
                return "Wuff! Wuff!"


class Gato(Mamifero):
        def som(self):
                return "Miauuu!"


mamifero = Mamifero()
print(mamifero.som())


homem = Homem()
print(homem.som())

cachorro = Cachorro()
print(cachorro.som())

gato = Gato()
print(gato.som())
```

3.5 Código em Python

Exemplo de aplicação em *Python*.

```
    class Poupanca(Conta):
        def __init__ (self ,numero):
            super().__init__(numero)
            self.__rendimento = 0.0

        def consultar_rendimento(self):
            return self.__rendimento

        def gerar_rendimento(self,taxa):
            self.__rendimento += super().consultar_saldo()*taxa/100
    conta = Poupanca(1)
    conta.creditar(200.0)
    conta.gerar_rendimento(10)
    print(conta.consultar_saldo())
    print(conta.consultar_rendmento())
```

4 Conclusão

Com suporte nos recursos didáticos e a partir do que foi explanado, espera-se que com esse relatório sucinto ter dado uma visão geral da orientação a objetos na linguagem Python. Assim para um estudo mais detalhado sobre o assunto, faz-se necessário uma pesquisa mais profunda na linguagem e nos conceitos da orientação a objetos. Contudo, acredita-se que como definido, tenhamos alcançado o objetivo de explanar esse paradigma.

5 Bibliografia

Referências

[1] Andrew P Black. "Object-oriented programming: Some history, and challenges for the next fifty years". Em: *Information and Computation* 231 (2013), pp. 3–20.

[2] Charles Dierbach. *Introduction to Computer Science Using Python: A Computational Problem-Solving Focus*. Wiley Publishing, 2012.

[3] Python. *Python e Programação Orientada a Objeto*. 2018. URL: `https://wiki.python.org.br/ProgramacaoOrientadaObjetoPython`.

[4] Alan Snyder. "Encapsulation and inheritance in object-oriented programming languages". Em: *ACM Sigplan Notices*. Vol. 21. 11. ACM. 1986, pp. 38–45.