FORMAÇÃO COMPLEMENTAR

Mentor: Felipe Moreira de Assunção
Conteúdo para formação complementar do Módulo de Programação Orientada a Objetos

INFORMAÇÕES DO MÓDULO

Descrição

Conheça os conceitos básicos sobre Programação Orientada a Objetos em Python

Objetivos do Ensino:

Espera-se que o aluno consiga, ao final do módulo:

- Conceitos de POO
- Classes e objetos
- Herança
- Polimorfismo
- Exemplo de uma aplicação real

Demonstração das Ferramentas

Utilização do Anaconda com Jupyter Notebook e Python ou outro editor de código de sua preferência

DICAS

Na execução das vídeo aulas, você pode regular a velocidade do vídeo, de forma a melhorar a sua compreensão na absorção do conteúdo e produtividade.

Para algumas pessoas, uma fala mais lenta e pausada pode ajudar a assimilar melhor o conteúdo e para outras, uma fala mais rápida pode tornar o conteúdo mais dinâmico e aumentar a produtividade. Dessa forma, podemos nos adaptar melhor as ferramentas de ensino, encontrando um caminho melhor, individualmente.

PROOGRAMAÇÃO ORIENTADA A OBJETOS

Vamos discutir um pouco mais sobre POO? Nesta formação complementar, gostaria de complementar alguns conceitos que considero importantes. Vamos começar?

1. **Objetos**
   - Objetos são abstrações do Python para dados.
   - Todos os dados em um programa Python são representados por objetos ou pelas relações entre esses objetos
   - Todo objeto tem uma identidade, um tipo e um valor
   - A identidade de um objeto nunca muda depois de criado (como endereço de objetos em memória)

2. Classes e métodos

O construtor é um método que geralmente é responsável pela alocação de recursos necessários ao funcionamento do objeto além da definição inicial das variáveis de estado (atributos).

- self: A palavra 'self' é usada para representar a instância de uma classe. Usando a palavra-chave "self", acessamos os atributos e métodos da classe em Python.
- método__init__: "__init__" é um método reservado em classes Python. É chamado de construtor na terminologia orientada a objetos. Este método é chamado quando um objeto é criado a partir de uma classe e permite que a classe inicialize os atributos da classe.

Classes podem ser instanciadas mais de uma vez, diminuindo linhas de código, tornando a programação mais dinâmica

3. Alguns conceitos importantes de POO

Nesta sessão, veremos alguns conceitos importantes da POO, que são a principal discussão do nosso módulo.

Figura 1 - Os quatro pilares da POO

3.1 Abstração

De modo geral, abstração para a POO, nada mais é do que você observar comportamentos e estruturas do dia-dia, e transformá-los em uma linguagem computacional. Ao utilizar a abstração do jeito certo, você conseguirá com menos esforço e mais qualidade, criar sistemas computacionais que mais se aproximarão à expectativa do usuário. Quando bem feita reforça o paradigma de virtualizar o mundo real.

Uma maneira de como conseguir a abstração na prática é observando estados e comportamentos similares entre dois ou mais objetos. Esse processo pode ser árduo visto que muitos desenvolvedores ainda tem dificuldade de dominar esse processo com maestria pois envolve o conjunto de muita prática e estudo. Além do mais, muitos sistemas se encontrariam muito mais robustos, eficazes, eficientes, se as técnicas de abstração, em conjunto com as demais técnicas de POO, fossem aplicadas corretamente.

3.2 Herança Simples e Múltipla

Neste tópico gostaria de ampliar um pouco mais os conceitos apresentados em nossas aulas sobre Herança. Experimente executar cada um dos códigos para verificar as saídas e tente fazer algumas modificações para entender melhor os exemplos.

Em nossas aulas, trabalhos com herança simples, em que a partir de uma super classe (ou classe pai), a subclasse (ou classe filha) herda métodos e atributos da classe super classe. Podemos exemplificar a herança simples através do seguinte código

```
class Primeira:
    def__init__(self, nome, idade):
        self.idade =  idade
        self.nome = nome

    def processo(self):
        print('Primeiro processso()')

class Segunda(Primeira):
    def processo(self):
        print('Segundo processso()')
    pass
```

Para verificar os resultados a partir do código acima, digite as instruções abaixo e execute linha por linha para confirmar cada um dos atributos e a herança:

```
obj = Primeira("José", 45)
obj.nome
obj.idade
obj.processo()

obj2 = Segunda("Maria", 25)
obj2.nome
obj2.idade
obj2.processo()
```

Antes de discutirmos sobre herança múltipla, vamos ao conceito de *Method Resolution Order* (MRO) ou Ordem de Resolução de Método (ORM) que é a ordem em que o Python procura um método em uma hierarquia de classes. Especialmente, ele desempenha um papel vital no contexto de herança múltipla, pois um único método pode ser encontrado em várias superclasses. Alem disso, devemos considerar o uso do `super().`

O `super()` nos permite sobrescrever métodos e alterar comportamentos. Quando precisamos importar parâmetros da super classe dentro de um método, usamos o `super()` para indicar o que queremos importar.

Agora, vamos considerar o seguinte código em que exemplificamos o uso de herança múltipla e o MRO (considere o funcionamento tanto para Python 2 e 3):

```
class Primeira:
    def__init__(self):
        super(Primeira, self).__init__()
        print ("primeira")

class Segunda:
    def__init__(self):
        super(Segunda, self).__init__()
        print ("segunda")

class Terceira(Primeira, Segunda):
    def__init__(self):
        super(Terceira, self).__init__()
        print ("terceira")

obj = Terceira()    # instancia um objeto obj da classe Terceira
obj.__init__()      # inicializa o método “processo” herdado da classe Primeira
print(Terceira.mro())       # imprime a ordem de execução
```

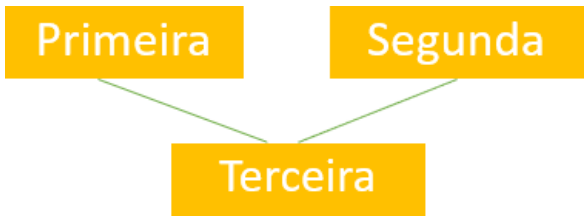

Figura 2 - Representação de uma herança múltipla

Neste exemplo, a classe Terceira ligará `Primeira._init_.` O Python procura cada atributo nos pais da classe, conforme listados da esquerda para a direita. Neste caso, estamos procurando `_init_.` Então, se você definir...

```
class Terceira(Primeira, Segunda):
    ...
```

... o Python começará examinando a classe `Primeira` e se a classe `Primeira` não tiver o atributo ou método, será analisado a classe `Segunda`.

Podemos chamar a classe `Terceira,` como uma “subclasse cooperativa”. Em relação ao resultado do console, podemos verificar o seguinte:

```
In [21]: obj = Terceira()    # instancia um objeto obj da classe Terceira
segunda
primeira
terceira

In [22]: obj.__init__()     # inicializa o método "processo" herdado da classe Primeira
segunda
primeira
terceira

In [23]: print(Terceira.mro())    # imprime a ordem de execução
[<class '__main__.Terceira'>, <class '__main__.Primeira'>, <class '__main__.Segunda'>, <class
'object'>]
```

Em poucas palavras, o Python tentará manter a ordem em que cada classe aparece na lista de herança, começando com a própria classe filho(Segunda), passando pela classe pai (Primeira) e pela classe (Terceira). À primeira vista, pode parecer difícil entender, então vamos a mais um exemplo:

```
class Primeira:
    def__init__(self):
        print ("Primeira entrada")
        super(Primeira, self)._init_()
        print ("Primeira saída")

class Segunda:
    def__init__(self):
        print ("Segunda entrada")
        super(Segunda, self).__init__()
        print ("Segunda saída")

class Terceira(Primeira, Segunda):
    def__init__(self):
        print ("Terceira entrada")
        super(Terceira, self).__init__()
        print ("Terceira saída")

obj = Terceira()    # instancia um objeto obj da classe Terceira
print(Terceira.mro())       # imprime a ordem de execução
```

Vamos continuar entendendo o que acontece quando instanciamos uma instância da classe `Terceira`, por exemplo `obj = Terceira()`.

1. De acordo com MRO `Terceira.__init__`executa.
    - impressões `Terceira entrada`
    - depois `super(Terceira, self)._init_()`executa e retorna MRO `Primeira.__init_`que é chamado.
2. `Primeira.__init__`executa.
    - impressões `Primeira entrada`
    - depois `super(Primeira, self)._init_()`executa e retorna MRO `Segunda.__init_`que é chamado.
3. `Segunda.__init__`executa.

   - impressões `Segunda entrada`
   - depois `super(Segunda, self).__init__()`executa e retorna MRO `object.__init__`que é chamado.
4. `object.__init__`executa (não há instruções de impressão no código)
5. execução remonta à `Segunda.__init__`qual, em seguida, imprime `Segunda saída`
6. execução remonta à `Primeira.__init__`qual, em seguida, imprime `Primeira saída`
7. execução remonta à `Terceira.__init__`qual, em seguida, imprime `Terceira saída`

Isso detalha por que a instanciação do Terceira() resulta em:

```
Terceira entrada
Primeira entrada
Segunda entrada
Segunda saída
Primeira saída
Terceira saída
```

O algoritmo MRO foi aprimorado a partir do Python 2.3 em diante para funcionar bem em casos complexos, mas muitos desenvolvedores acreditam que a herança em Python eventualmente tende a ser confusa e precisa ser melhorado. Alguns pesquisadores sugerem melhorias nos códigos de MRO de modo que as instruções se tornem mais simples e isso tem sido feito ao longo do tempo, atestado por pesquisas científicas. Você pode se aprofundar um pouco mais nesses assuntos, mas por hora, podemos parar por aqui para que você possa ir absorvendo o conteúdo básico necessário para os seus primeiros programas.

3.3 Polimorfismo

Podemos considerar o polimorfismo como uma técnica que permite ter algo único em vários lugares. Por exemplo, podemos utilizar o polimorfismo em classes distintas, compartilhando métodos e/ou funções que são comuns a todas elas. Vale dizer que nas classes derivadas (subclasses) existe o compartilhamento de uma relação comum: as instancias daquelas classes são utilizadas da mesma maneira.

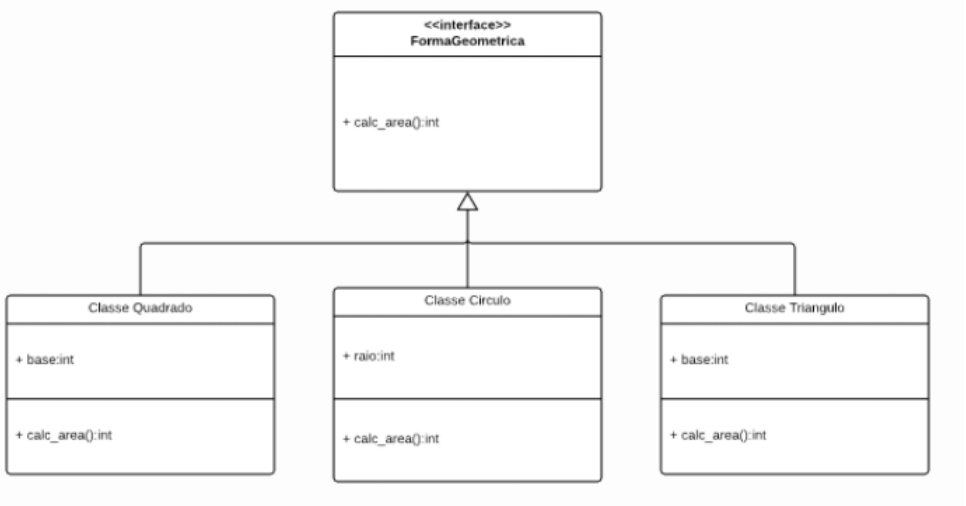

Figura 3 - Diagrama de Classe representando um caso de polimorfismo

Como podemos observar nesse caso, as subclasses Quadro, Circulo e Triangulo herdam da super classe FormaGeometrica o método cal_area() que calcula a área de maneira específica para cada uma das formas representadas nas subclasses.

Obs: Em alguns casos, de maneira intermediária a avançada para programação em Python, podemos considerar a utilização do comando super() para delimitar melhor a ordem e forma com que as classes se relacionam.

3.4 Encapsulamento

O encapsulamento é realizado quando cada objeto mantém seu estado privado dentro de uma classe. Outros objetos não podem acessar diretamente este estado, em vez disso, eles podem apenas chamar as funções públicas. O objeto gerencia seu próprio estado por meio dessas funções e nenhuma outra classe pode alterar o estado, a menos que seja explicitamente permitido. Se você deseja se comunicar com o objeto, você deve usar os métodos fornecidos. Mas você não pode alterar diretamente o estado.

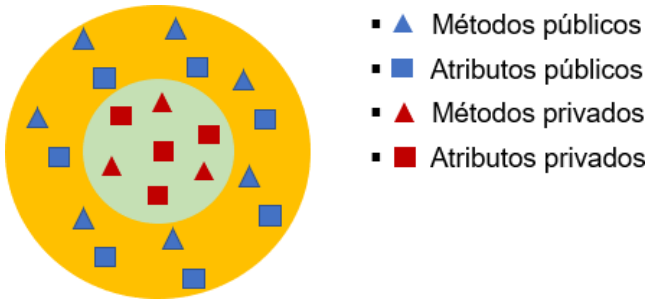

Figura 4 - Relação entre métodos e atributos públicos e privados no escopo de um projeto

Em programação, ocultar os detalhes de implementação de um objeto é chamado de encapsulamento. Para criar um membro protegido em Python, basta seguir a **convenção** de prefixar o nome do membro com **um único sublinhado** ou no máximo dois sublinhados.

ASSUNTOS PARA DISCUSSÃO NO FÓRUM

Tópico 1: Além da orientação a objetos, existem outros paradigmas importantes na programação? Justifique.

Tópico 2: Qual a importância da POO em relação a programação procedural? Discuta sobre as possibilidades, vantagens e desvantagens deste paradigma.

LINKS INTERESSANTES

Saiba um pouco mais sobre heranças e MRO em Python: http://www.srikanthtechnologies.com/blog/python/mro.aspx#:~:text=Method%20Resolution%20Order%20(MRO)%20is,found%20in%20multiple%20super%20classes.

BIBLIOGRAFIA

- Python Books: https://wiki.python.org/moin/PythonBooks
- Introdução à Programação com Python: Algoritmos e Lógica de Programação Para Iniciantes
- Python Fluente: Programação Clara, Concisa e Eficaz