Atividade II.2 - Maria Eduarda de Melo Hang

Introdução

Seja bem-vindo(a) ao tutorial de programação orientada a objetos de Python, nesse documento será ensinado os seguintes tópicos: Declaração e instanciação de classe(métodos, atributos e construtores), herança simples e múltipla, polimorfismo, agregação e métodos abstratos e estáticos. Além disso, ao final de cada tópico será mostrado o nome do arquivo a ser executado para a verificação do leitor das propriedades apresentadas nesse tutorial.

Declaração e instanciação de classe

Um exemplo para aplicar esse assunto seria uma pessoa e será considerada como a primeira classe desse tutorial. Não esquecendo, as características básicas de uma pessoa seriam nome e idade, correto? (sendo a cargo do leitor criar outros atributos posteriormente para testá-los) Na linguagem Python, a declaração e o construtor sem argumentos dessa classe seria desta forma:

```
class Pessoa:
    # Construtor
    def __init__(self):
        self.idade = 0
        self.nome = ''
```

Por definição da comunidade, o construtor deverá ter o nome *__init__*. A diretiva *def* é utilizada para definir métodos e o argumento *self* seria o próprio objeto que foi instanciado (não o coloque quando chamar o construtor) para poder mexer em seus atributos.

Para melhorar esse construtor, serão colocados parâmetros como idade e nome (não é necessário colocar o tipo de dado nos parâmetros já que essa linguagem contém a tipagem dinâmica, ou seja, ela mesma atribui os tipos de dados em tempo de compilação ou execução) e o construtor ficaria desta forma:

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

Caso fosse necessário ver o nome de um objeto, deveria ser criado um método para retornar o nome do objeto que o chamou e ficaria assim:

```
    def ver_nome(self):
        return self.name
```

Assim como foi comentado antes, o argumento *self* serve para acessar os atributos do objeto que chamou o método.

Após adicionar outro método, o *ver_idade,* a classe ficará deste jeito:

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

    def ver_nome(self):
        return self.name

    def ver_idade(self):
        return self.idade
```

Um objeto representa a instância de uma classe e para criá-lo no Python é necessário chamar o construtor (o nome da classe com os parâmetros), colocando os parâmetros **na ordem que foi declarado**.

```python
eduardo = Pessoa(20, "Eduardo Oliveira")
print(eduardo.ver_nome()) # Eduardo Oliveria
print(eduardo.ver_idade())) # 20
```

Execute o documento Declarao_Pessoa.py e veja as saídas dos prints.

Herança Simples

Caso fosse preciso criar uma classe Empregado com os atributos nome, idade e salário seria mais eficiente utilizar a herança para essa classe herdar os atributos nome e idade e os métodos em vez de fazer todos os métodos novamente. A forma de declarar a herança em Python seria desta forma:

```python
class Empregado(Pessoa):
    # Construtor
    def __init__(self, idade, nome, salario):
        super().__init__(idade, nome)
        self.salario = salario

    def ver_salario(self):
        return self.salario
```

Todas as heranças deverão estar após o nome da classe como foi apresentado nessa imagem. Como os atributos idade e nome se encontram na classe pessoa, pode-se utilizar o construtor dessa classe para atribuir os parâmetros através da forma *super().__init__(parâmetros)*, ou seja, utiliza-se o construtor da classe pessoa para atribuir os valores de idade e nome.

A instanciação será idêntica ao tópico anterior e não é necessário o *super()* para chamar métodos da classe que foi herdada.

```python
marcos = Empregado(35, "Marcos Silva", 570)
print(marcos.ver_nome()) # Marcos Silva
print(marcos.ver_idade()) # 35
print(marcos.ver_salario()) # 570
```

Execute o documento Heranca_Simples.py e veja as saídas dos prints.

Polimorfismo e herança múltipla

Para esse tópico, será utilizado estas classes:

```python
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def dormir(self):
        print("O(A) %s está dormindo..." % self.nome)

    def comer(self, comida):
        print("Comendo...")
```

```python
class Carnivoro(Animal):
    def __init__(self, nome):
        super().__init__(nome)

    def comer(self, comida):
        if (comida == "carne"):
            print("O(A) %s está comendo carne..." % self.nome)
        else:
            print("O(A) %s somente come carne" % self.nome)

    def cacar_animais(self):
        print("%s está caçando..." % self.nome)

class Herbivoro(Animal):
    def __init__(self, nome):
        super().__init__(nome)
    def comer(self, comida):
        if (comida == "vegetal"):
            print("O(A) %s está comendo um vegetal..." % self.nome)
        else:
            print("O(A) %s somente come vegetais" % self.nome)

    def procurar_vegetais(self):
        print("%s está procurando vegetais..." % self.nome)
```

O polimorfismo significa pegar um método herdado, nesse caso o *comer*, e implementá-lo de forma diferente como foi feito nas outra duas classes que herdam da animal. Lembrando que o método deve ter a mesma identificação, ou seja, o mesmo nome e parâmetros.

Enquanto que a herança múltipla seria herdar duas ou mais classes, ou seja, todos os atributos e métodos de ambas as classes, nesse caso carnívoro e herbívoro, ficando assim a classe onívoro:

```
class Onivoro(Carnivoro, Herbivoro):
    def __init__(self, nome):
        super().__init__(nome)

    def comer(self, comida):
        if (comida =="vegetal" or comida=="carne"):
            print("O(A) %s está comendo..." % self.nome)
        else:
            print("O(A) %s não come %s" % (self.nome, comida))
```

Instanciando os objetos de cada classe:

```
urso = Onivoro("urso")
urso.dormir()
urso.comer("carne")
urso.comer("fruta")
urso.comer("vegetal")
urso.cacar_animais()
urso.procurar_vegetais()

leao = Carnivoro("leão")
leao.dormir()
leao.comer("carne")
leao.comer("fruta")

alce = Herbivoro("alce")
alce.dormir()
alce.comer("vegetal")
alce.comer("carne")
```

Para verificar que um objeto da classe onívoro herdou todos os métodos de ambas as classes, execute o arquivo heranca_multipla_polimorfismo.py.

Agregação

A agregação representa uma relação entre classes da forma não destrutiva, ou seja, se uma determinada classe contém outra como atributo mas se a primeira for destruída a segunda se manterá no programa. Um exemplo para esse assunto seria o carrinho e os produtos contidos nele, uma vez que serão adicionados produtos ao carrinho conforme a necessidade e caso o carrinho seja destruído, os produtos não serão destruídos junto.

```python
class Produto:
    def __init__(self, nome, codigo, preco):
        self.nome = nome
        self.codigo = codigo
        self.preco = preco

    def printar_produto(self):
        print("Nome: %s Código: %s Preço: R$%.2f" % (self.nome, self.codigo, self.preco))

    def ver_preco(self):
        return self.preco

class Carrinho:
    produtos = []

    def __init__(self, produtos):
        i = 0
        self.produtos = produtos

    def adicionar_ao_carrinho(self, produto):
        produtos.append(produto)

    def contar_produtos(self):
        return len(produtos)

    def listar_produtos(self):
        for i in range(0, len(produtos)):
            produtos[i].printar_produto()
```

```python
maionese = Produto("Maionese", 1, 2.50)
ketchup = Produto("Ketchup", 2, 3.00)
pao = Produto("Pão", 3, 1.50)

produtos = [maionese, ketchup, pao]
carrinho = Carrinho(produtos)
print(carrinho.contar_produtos())
print(carrinho.listar_produtos())
print(carrinho == None)
del carrinho
print(maionese == None)
print(ketchup == None)
print(pao == None)
#print(carrinho == None)
```

Percebe-se que os produtos não são instanciados em nenhum método da classe carrinho, logo eles não serão destruídos caso a instância do carrinho for destruída.

Execute o arquivo agregacao.py e tente executar o último comando que está comentado.

Para entender sobre listas em python: https://www.programiz.com/python-programming/methods/list

Método Abstratos

Uma classe abstrata representa um **modelo** para outras classes que a herdam, tendo atributos e métodos, no entanto, os métodos **não devem ser implementados na própria**

classe e sim nas outras (caso não implemente o método, haverá um erro ao compilar o programa).

Utilizando o polígono como exemplo para esse tópico, criaremos uma classe abstrata desta forma:

```python
from abc import ABC, abstractmethod

class Poligono(ABC):

    def __init__(self, vertices):
        self.vertices = vertices
        super().__init__()

    @abstractmethod
    def diz_vertices(self):
        pass

class Triangulo(Poligono):
    pass

x = Triangulo(3)
```

Para criar uma classe abstrata é preciso importar a biblioteca *abc* (mais sobre ela: https://docs.python.org/3/library/abc.html) e colocar a herança na classe. Um método abstrato é distinguido dos outros com o *@abstractmethod* colocado antes da declaração e a diretiva *pass* serve para avisar interpretador de que deve deixar passar essa classe ou método.

Ao tentar executar esse código, encontrará este erro:

```
Traceback (most recent call last):
  File "ex.py", line 15, in <module>
    x = Triangulo(3)
TypeError: Can't instantiate abstract class Triangulo with abstract methods diz_vertices
```

E o motivo é simples, a classe triangulo, que herda a classe abstrata poligono, não implementou o método abstrato *diz_vertices* e para corrgir esse erro altere o código desta forma:

```python
from abc import ABC, abstractmethod

class Poligono(ABC):

    def __init__(self, vertices):
        self.vertices = vertices
        super().__init__()

    @abstractmethod
    def diz_vertices(self):
        pass

class Triangulo(Poligono):
    def diz_vertices(self):
        print(self.vertices)
    def calcula_area(self, base, altura):
        return (base*altura)/2

triangulo = Triangulo(3)
triangulo.diz_vertices()
print(triangulo.calcula_area(2,3))
```

Para ver o resultado, execute os arquivos erro_abstrata.py e abstrata.py.

Métodos Estáticos

Os métodos estáticos permitem que sejam acessados sem uma instância da classe, como por exemplo:

```python
class Calculadora(objetc):

    @staticmethod
    def soma(a, b):
        return a + b
    @staticmethod
    def subtracao(a, b):
        return a - b
    @staticmethod
    def multiplicacao(a, b):
        return a * b
    @staticmethod
    def divisao(a, b):
        if (b != 0):
            return a/b
        else:
            print("Não é possível fazer divisão por zero")

print(Calculadora.soma(2,3))
print(Calculadora.subtracao(2,3))
print(Calculadora.multiplicacao(2,3))
print(Calculadora.divisao(2,3))
```

Percebe-se que em nenhum momento foi instanciado um objeto da classe calculadora.

Outro exemplo de classe estática é a *math* muito utilizada em contas, uma vez que para acessar o valor de pi, basta fazer *math.pi*.

Mais sobre *math*: https://docs.python.org/3/library/math.html

Execute o arquivo estatico.py.