Atividade 3 - Paradigmas

Vitor Della Torre dos Santos
18206049

- Declaração e instanciação de classes (métodos, atributos, construtor)

```
class Stack:

    def __init__(self) -> None:
        self.elements = list()

    def is_empty(self) -> bool:
        return True if not self.elements else False

    def push(self, element):
        self.elements.append(element)

    def pop(self):
        return self.elements.pop()



stack = Stack()
stack.push(0)
stack.push(1)
stack.push(2000)
print(stack.elements)
print(stack.is_empty())
```

Para realizar a instanciação de uma Classe em Python, é necessário, inicialmente, escolher o nome da variável. Supõe-se uma variável "a". Como queremos que esta seja representante de uma Classe, ou seja, uma instância, não basta dar um valor *built-in* para ela, como seria o caso com um número inteiro ou uma sequência de caracteres: devemos chamar o construtor de uma dada classe para apontar àquela variável recém criada e declarar que ela é do tipo "Classe X", sendo esta última chamada pelo seu método de construção.

`a = classe_x()` (Vale lembrar que, em versões maiores ou iguais a Python 3.6, pode-se também indicar visualmente o tipo de uma dada variável, como `a: classe_x = classe_x()`).

No caso de não existir uma implementação de um construtor, isso não impede o ambiente de desenvolvimento de gerar um objeto: o interpretador nos dará o objeto que queremos, e teremos um construtor não visível ao programador, sem parâmetros.

Em seguida, após criado o objeto, podemos realizar operações com este, as quais são chamadas de métodos. Um método é uma função atribuída a um conjunto de objetos,

ou melhor, às instâncias de uma classe; é quase como uma coisa inerente ao objeto criado: da mesma forma que pássaros voam, e isso é natural, a pilha representada acima possui formas de adicionar e remover elementos de modo inerente.

Indo por este mesmo raciocínio, temos os atributos, que são características não funcionais de um objeto: separando métodos e atributos por suas respectivas funcionalidades, métodos executam tarefas, manuseiam variáveis, até atributos, e podem retornar resultados para diferentes operações, enquanto atributos são entidades que não propriamente executam coisas: são valores, e são usados para, por exemplo, fazer a distinção entre objetos ou controlar o fluxo de execução.

- Herança (simples e múltipla) e polimorfismo

Tendo-se duas classes A e B, em que A *herda* atributos e funções de B, estabelece-se uma relação de *Herança* entre ambas: A é **subclasse** de B.

```
class A(B):
      pass
```

Nesse sentido, A é uma *especificação* de B:

1. A é do tipo A;
2. A é do tipo B;
3. B é **não** é do tipo A;
4. B é do tipo B.

Um exemplo lúdico seria pensar num semideus.

```
class greek_god:

   def __init__(self, name: str) -> None:
       self.name = name

class human:

   def __init__(self, name: str) -> None:
       self.name = name

class demigod(greek_god, human):

   def __init__(self, greek_god, human) -> None:
       super().__init__(greek_god.name)
       self.human_parent = human.name

father = greek_god('Zeus')
mother = human('danae')

perseu: demigod = demigod(father, mother)
print(perseu.name)  # Isso vai gerar um resultado diferente do
                    # que queremos.
```

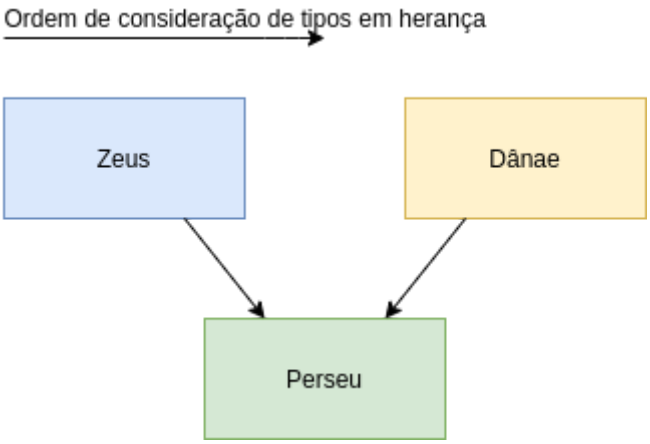
`No exemplo acima, temos, comparando` *danae* com *perseu*:

1. Perseu é humano;
2. Perseu é um deus;
3. Dânae é humana; mas
4. Dânae **não** é uma deusa.

Analogamente, Zeus é deus, mas não é humano.

Portanto, Perseu herda atributos e propriedades de seus familiares: Zeus e Dânae: um semideus é tanto um deus quanto um humano.

No entanto, vale notar que criou-se métodos propositalmente com nomes iguais para ambas as superclasses da classe *demigod*. Como é gerenciada a decisão de qual método será chamado?




```
class demigod(greek_god,
human):
  # ...
```

De uma forma simplista, isso é decidido dando prioridade à classe nomeada mais à esquerda. Neste caso, foi *greek_god.* Portanto, ao verificar o atributo *name* de *perseu*, veríamos que seria impresso “Zeus”, pois a classe *demigod* não tem um atributo para representar um nome.

Agora, o que aconteceria se tivéssemos o caso contrário? Tomemos outro exemplo para esclarecer melhor o próximo tópico:

```
class mammal:

    def __init__(self, name: str) -> None:
        self.name = name

    def do_noise(self):
        print('AAAAA')

    def give_name(self):
        print('My name is ' + self.name)


class dog(mammal):

    def __init__(self, name: str) -> None:
        super().__init__(name)

    def do_noise(self):
```

```
        print('Woof')


dog1 = mammal('Albert')
dog1.do_noise()  # AAAAA

dog2 = dog('Albert')
dog2.do_noise()  # Woof
dog2.give_name()
```

Vê-se que há uma classe para os mamíferos como uma superclasse e, para a subclasse, temos cachorros. Se criamos uma variável, como *dog2*, esta pode também chamar o método *give_name()*, mesmo que este não esteja declarado dentro de sua classe. A isto dá-se o nome de *polimorfismo*, pois mesmo que um cachorro seja um cachorro, ele ainda é um mamífero e, portanto, pode fazer tudo o que um mamífero é capaz de realizar.

- Composição e Agregação

Neste contexto de criação de classes e, consequentemente, objetos, vale a pena discutir o que é Composição e Agregação.

1. Composição: quando um objeto estabelece uma relação de dependência obrigatória com outro, de modo que é necessária a criação de um para que possa-se ter a criação de outro. Se tivéssemos que construir uma lista encadeada, não poderíamos fazer isso sem nodos, imaginando que estes representam uma classe; ou seja, sem a criação de nodos, não temos como gerar uma lista encadeada, pois esta é **composta** de nodos.
2. Agregação: quando um objeto estabelece uma relação com outro sem que um seja necessário para o ciclo do outro. É quase como uma generalização do que é a Composição. Por exemplo, numa classe que representa um shopping, esta se relaciona de modo agregado aos clientes e outros transeuntes: o shopping ainda existirá quando estiver vazio, assim como as pessoas dentro deste ainda existirão após a saída. Diz-se, portanto, que shopping e pessoa, neste contexto, estão **agregados**.

- Métodos abstratos e estáticos

```
class abstract_list:

   def __init__(self) -> None:
       pass

   def add(element):
       pass

   def remove(element):
       pass
```

```
class concrente_queue(abstract_list):

   def __init__(self) -> None:
       super().__init__()
       self.elements = list()

   def add(self, element):
       self.elements.append(element)

   def remove(self, element):
       return self.elements.pop(0)
```

Métodos abstratos servem de *template* a outras classes: são declarações com o nome de uma dada função, seu tipo de retorno e parâmetros, mas sem implementação, a qual fica a critério da classe que herdar este mesmo método abstrato, como é demonstrado acima.

A classe representante da lista abstrata não implementa de fato nenhum de seus métodos, apenas os declara. A implementação recai sobre os ombros de nossa pilha.

Em Orientação a Objetos, considera-se isso uma boa prática, pois simplifica a construção de classes novas que herdam atributos e outros métodos de uma ou mais superclasses abstratas, de modo que não precisa-se saber as partes mais intrínsecas do funcionamento de métodos entre tipos que herdam de uma mesma superclasse, pois retornarão objetos de mesmo tipo, apesar da geração de resultado se dar de forma diferente.

Em paralelo a isso, temos métodos estáticos.

```
class Printer:

   @abstractmethod
   def show(text: str):
       print(text)


Printer.show("Uhuu")
```

Métodos abstratos são métodos restritos às classes: instâncias não os chamam, pois estes não representam instância alguma. São úteis para preservar um estilo de programação mais voltado à Programação Estruturada.

Em muitos Design Patterns diferentes, há a presença de métodos estáticos, como no padrão de projeto Builder, o qual possuirá um método que constrói uma instância de outra classe.

```
class PrinterBuilder:
```

```
    @abstractmethod
    def build():
        return Printer()


class Printer:

    def __init__(self) -> None:
        self.word = "AAAAA"

printer: Printer = PrinterBuilder.build()
print(printer.word)
```