12

Mãos na massa

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

Na última aula, o código acabou ficando sem herança, mas perdeu algumas funcionalidades do Python, como poder usar o `for in`, `in` e o `len`.

1) Para conseguir continuar com as vantagens do polimorfismo sem precisar herdar a classe `list`, faça o seguinte, altere a classe `Playlist` para:

```
class Playlist():
    def __init__(self, nome, programas):
        self.nome = nome
        self._programas = programas

    def __getitem__(self, item)
        return self._programas[item]

    @property
    def listagem(self):
        return self._programas

    @property
    def tamanho(self):
        return len(self._programas)
```

2) Apenas com esta alteração, você consegue agora iterar sobre a `Playlist`, sem acessar a listagem como antes:

```
for programa in minha_playlist:
    print(programa)
```

No Python, não é preciso herdar de uma classe específica pra ter polimorfismo. O que é importante no Python é: se você quer um iterável, devo se preocupar com o que um iterável deve fazer.

O nome dessa característica é ***duck typing***.

3) Será que dá para alterar a classe de outra forma para suportar também o `len`, ao invés de ter que acessar o método/*property* `tamanho`? Sim! Faça o seguinte:

```
class Playlist():
    def __init__(self, nome, programas):
        self.nome = nome
        self._programas = programas

    def __getitem__(self, item)
        return self._programas[item]

    def __len__(self):
        return len(self._programas)
```

Desta forma, você consegue chamar `len(minha_playlist)`, já que `Playlist` implementa `__len__`. Isso torna a `Playlist` um `Sized` (alguém que implementa `__len__`).

Observação: Você não precisa mais dos métodos/propriedades `listagem` e `tamanho`, então pode apagar :)

4) Com *duck typing*, você ganha muita flexibilidade ao não precisar ficar preso aos tipos dos objetos, só que em alguns momentos você pode querer ter restrições, como de uma garantia que uma classe implemente os métodos que você quer.

Em outras linguagens, há a ideia de classes e métodos abstratos, que forçam as classes filhas a implementar alguns métodos.

Para fazer o mesmo no Python, é possível usar classes abstratas, as chamadas **ABC** (***A**bstract **B**ase **C**lasses*). Existem classes já prontas que ajudam nessa ideia.

Para exemplificar, crie um novo arquivo com o nome **testeAbc.py**, para teste, com o seguinte conteúdo e execute-o:

```
from collections.abc import MutableSequence

class MinhaListinhaMutavel(MutableSequence):
    pass
```

Você vai perceber que não acontece nada quando apenas este código é executado.

5) A classe `MinhaListinhaMutavel` herda de `MutableSequence`, ou seja, é desejável que o Python avise que tem que implementar todos os métodos abstratos de uma `MutableSequence`. Só que parece que não funcionou.

Como Python é uma linguagem de tipagem dinâmica, não dá pra ter essa garantia só percorrendo o código de definição da classe. O que dá pra fazer é validar os métodos em tempo de **instanciação**. Adicione o código abaixo e teste novamente:

```
objetoValidado = MinhaListinhaMutavel()
print(objetoValidado)
```

Agora dá um erro, dizendo que você esqueceu de implementar todos os métodos necessários para tornar a classe uma `MutableSequence`.

Sempre que você quiser garantir a implementação de alguns métodos, pode recorrer às classes já existentes em `collections.abc` e outros pacotes também.