Live de Python #31

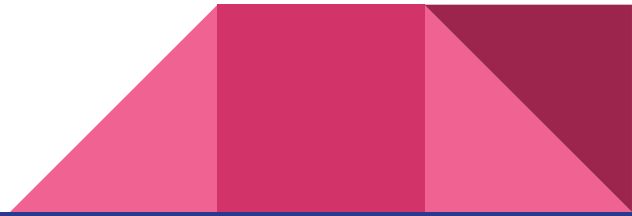
collections #4 - User(Dict|List|String) e collections.abc

Roteiro

- Uma introdução informal sobre o collections
- UserDict
- UserList
- UserString
- Uma introdução dócil aos collections.abc
  - Entendendo o que é uma sequência
  - E agora uma sequência mutável não iria mal, não é mesmo?

Uma introdução informal sobre o collections [0]

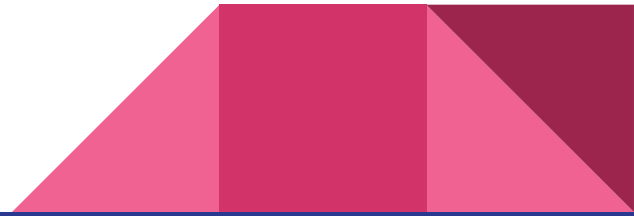
A biblioteca padrão do python é rica e tem muitos tipos de containers (dict, set, list, tuple, ...). A biblioteca collections tem objetivo de oferecer alternativas aos objetos de propósitos gerais embutidos em Python (Counter, ChainMap, deque, namedtuple, ...) para que seja possível implementar novas coisas sem reinventar a roda (ou reinventar tipos nesse caso).



Uma introdução informal sobre o collections [1]

Como sabemos, os objetos que estão contidos em collections são de grande abstração, os mesmos produzem interfaces como se fossem objetos nativos do python para que possamos estender os mesmos.

Criar um dict maluco, uma lista que sempre adiciona reversamente... etc...



UserDict [0]

Um objeto dicionário que contém um dicionário embrulhado (wrapper) e que oferece as mesmas interfaces de um dicionário da biblioteca padrão.

```
In [1]: from collections import UserDict

In [2]: xpto = UserDict({1:1})

In [3]: xpto[1]
Out[3]: 1

In [4]: xpto.data
Out[4]: {1: 1}
```

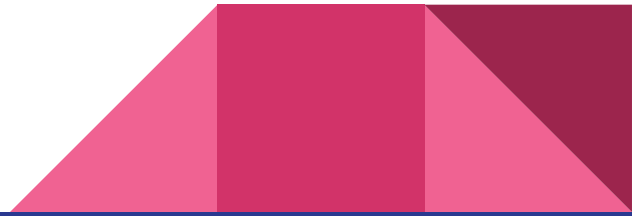
UserDict [1]

```
In [11]: set(dir(dict)).difference(dir(UserDict))
Out[11]: set()

In [12]: set(dir(UserDict)).difference(dir(dict))
Out[12]:
{'_MutableMapping__marker',
 '__abstractmethods__',
 '__dict__',
 '__module__',
 '__reversed__',
 '__slots__',
 '__weakref__',
 '_abc_cache',
 '_abc_negative_cache',
 '_abc_negative_cache_version',
 '_abc_registry'}
```

collections.abc [0]

ABC - Abstract Base Classes (Classes bases abstratas). São classes “virtuais” que não herdam de nenhuma classe builtin mas fornecem as interfaces (ou métodos) necessários para que seja possível emular o comportamento dos objetos nativos do python (Sequências, Iteráveis, Containers, Corrotinas, Invocáveis, ...).



collections.abc [1]

Por exemplo, se preciso desenvolver o comportamento de um container (ou seja, usar ‘in’, será necessário implementar o método ‘__contains__’.

```
'a' in 'eduardo'  # True
'eduardo'.__contains__('a')  # True
```

collections.abc [2]

Então nosso objeto deve implementar as interfaces necessárias para que seja possível aplicar o operador ‘in’.

```
class Container:
    def __init__(self, container):
        self.data = container

    def __contains__(self, value):
        return value in self.data
```

collections.abc [3]

Porém, o Python já proporciona uma abstração necessária para esses casos, collections.abc.Container.

```
from collections.abc import Container


class MyContainer(Container):
    pass
```

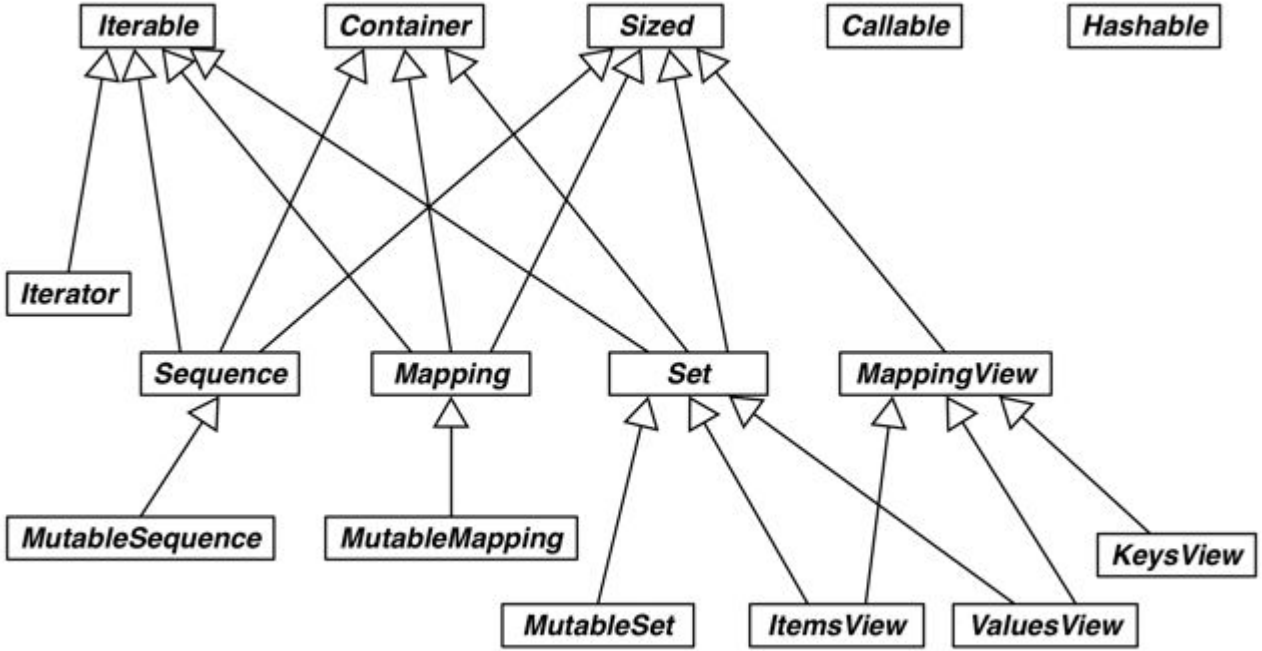
collections.abc [4]

E ao ser instanciado, nos cobra (usando abstractmethod) que essa implementação seja feita

```
In [1]: a = MyContainer()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-7d547f1c1f50> in <module>()
----> 1 a = MyContainer()

TypeError: Can't instantiate abstract class MyContainer with abstract me
thods __contains__
```

Classes de collections.abc



FONTE: FLUENT PYTHON - Luciano Ramalho - O'Reilly