02

Tuples e dictionary

Tuples

No capítulo anterior vimos as listas, que nada mais são do que uma estrutura de dados que guardam valores de qualquer tipo conhecido do Python. Aprendemos que elas podem crescer ou diminuir conforme nossa necessidade.

Vamos criar uma lista que contém todos os tipos de convite do nosso sistema:

```
>>> tipos_convite = ['vip', 'normal', 'meia', 'cortesia']
```

Criar uma lista não é novidade para nós. Porém, durante o levantamento de requisito do sistema, nosso cliente foi bem taxativo alegando que só trabalhará com esses tipos e ninguém tem autorização para adicionar novos. Será que nossa inocente lista reflete a regra de negócio? Vejamos:

```
>>> tipos_convite.append('penetra')
>>> tipos_convite
['vip', 'normal', 'meia', 'cortesia', 'penetra']
```

Com certeza nosso cliente não gostará de ver passeando por aí um convite do tipo `penetra`. O problema é que a lista é mutável. Há conjuntos de dados que nunca precisam mudar de tamanho. A lista de meses ou estações do ano são exemplos clássicos disso. Nesse caso pode fazer sentido usar uma estrutura de dados similar à lista que vimos, mas que não seja possível adicionar ou remover valores. No Python, essa estrutura de dados se chama **Tuple**.

Vamos realizar uma ligeira modificação na declaração de nossa lista:

```
>>> tipos_convite = ('vip', 'normal', 'meia', 'cortesia')
>>> tipos_convite
('vip', 'normal', 'meia', 'cortesia')
```

Repare que nossa lista é declarada com `(...)` (parênteses) no lugar do `[...]` (colchetes). E agora? Será que conseguimos adicionar um novo tipo de convite?

```
>>> tipos_convite.append('Chocalho')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Recebemos uma mensagem de erro alegando a inexistência da função 'append'. Você ficou magoado? Esperamos que não, pois é justamente o que queremos: evitar que alguém altere a lista depois de criada.

Acessamos os elementos da nossa tupla utilizando o mesmo operador *slice* presente nas listas:

```
>>> tipos_convite[1]
'normal'
```

```
>>>tipos_convite[0:2]
 ['vip', 'normal']
```

Dicionários

Uma característica comum entre listas e tuples é que eles guardam uma sequência de valores. E se quisermos guardar o tipo do convite com o seu valor? Repare que uma lista ou tuple não serve muito bem para este propósito:

```
>>> convite = ('vip', 60, 'normal', 40, 'meia', 30, 'cortesia',0)
```

Se quisermos recuperar o tipo de convite com o seu valor precisamos utilizar o operador slice, algo complicado para operações do dia-a-dia:

```
>>> convite[0:2]
('vip', 60)
```

Para facilitar a vida de desenvolvedor, existe uma estrutura de dados chamada de **Dictionary**. Ela nos permite criar facilmente uma associação entre o tipo do convite e seu valor. Para declararmos um dicionário, usaremos um par de chaves (`{ … }`) com os elementos separados por vírgula. Cada um dos elementos possuirá uma **chave** e um **valor**. Estes ficam separados pelo símbolo `:` (dois pontos).

Nesse exemplo associamos cada convite com o seu valor:

```
>>> convite_com_valor = {'vip' : 60 , 'normal' : 40 , 'meia': 30 , 'cortesia':0}
```

Recuperando valores e alterando valores

Para recuperarmos o valor utilizaremos a sua chave. Por exemplo, para sabermos qual é o valor do convite `vip` usaremos:

```
>>> convite_com_valor['vip']
60
```

Podemos alterar um valor baseado na sua chave, basta atribuir um novo valor:

```
>>> convite_com_valor['meia'] = 50
```

Imprimindo as chaves e os valores

Se quisermos imprimir todas as chaves ou todos os valores, o dicionário oferece dois métodos: `keys` e `values` :

```
>>> convite_com_valor.keys()
['vip', 'cortesia', 'meia', 'normal']
```

E

```
>>> convite_com_valor.values()
[60, 0, 30, 40]
```

Nos exercícios veremos mais funcionalidades interessantes com tuples e dicionários! Até lá!