

Aula 3 - Estruturas de dados

Karl Jan Clinckspoor

2 de julho de 2018

Sumário

1	Introdução	1
2	Listas	1
2.1	Indexação	2
2.2	Seções (<i>slicing</i>)	2
2.3	Métodos de listas	4
3	Mutabilidade e Imutabilidade	5
4	Dicionários	5
4.1	Métodos de dicionários	6
5	Tuples	7

1 Introdução

Estruturas de dados é um nome chique para coisas bastante simples. As estruturas internas de maior relevância para esse curso são, em ordem decrescente de importância:

- Listas
- Dicionários
- Tuples

2 Listas

Listas são sequências mutáveis de qualquer objeto. Para criar uma lista, basta separar os objetos por vírgulas entre colchetes []. A função `list()` transforma uma estrutura de dados em uma lista.

```
In [60]: lista1 = [1, 2, 3, 4, 5]
        lista2 = ['a', 'b', 'c', 'd']
        lista3 = [lista1, lista2]
        lista4 = list('Hello world!')
```

Veja que listas podem conter listas. Isso é chamado de *nesting*. Além disso, veja que uma string é basicamente uma lista, só que exclusivamente para caracteres, e imutável. O significado de *mutável* e *imutável* se tornará mais claro adiante.

2.1 Indexação

Para obter a informação de uma posição da lista, utiliza-se a indexação, que também usa colchetes, porém juntos do nome. Algo que pode confundir bastante é que a posição das listas, em Python, começa do **zero**. Ou seja, a **terceira** posição tem número **2**! Tenha isso sempre em mente.

```
In [9]: lista1[3]
```

```
Out[9]: 4
```

```
In [11]: hello = 'Hello world!'
         hello[2]
```

```
Out[11]: 'l'
```

Também é possível acessar os elementos do final de uma lista, utilizando números negativos. **-1** é o último elemento da lista, **-2** é o penúltimo elemento, e assim por diante.

```
In [12]: lista2[-1]
```

```
Out[12]: 'd'
```

Quando há listas dentro de listas, é necessário utilizar colchetes seguidos.

```
In [13]: lista3[0][2]
```

```
Out[13]: 3
```

Aqui, foi pego o primeiro elemento da lista3, e depois o terceiro elemento do primeiro elemento.

2.2 Seções (*slicing*)

Além de acessar membros pontualmente, é possível também acessar seções de listas utilizando o conceito de *slicing*. A sintaxe de um *slice* é:

[índice inicial:índice final (não incluso):passo (opcional)]

```
In [14]: lista1[0:3]
```

```
Out[14]: [1, 2, 3]
```

```
In [16]: lista1[0:-2]
```

```
Out[16]: [1, 2, 3]
```

```
In [20]: lista1[0:5:2]
```

```
Out[20]: [1, 3, 5]
```

```
In [23]: lista1[-1:0:-1] # Passos negativos significa andar para trás.
```

```
Out[23]: [5, 4, 3, 2]
```

Quando nenhum valor é fornecido, supõe-se foi colocado "tudo". Por exemplo `[:-1]` significa "tudo até o último termo".

```
In [17]: lista1[:-1]
```

```
Out[17]: [1, 2, 3, 4]
```

```
In [18]: lista1[3:]
```

```
Out[18]: [4, 5]
```

```
In [19]: lista1[-3:5]
```

```
Out[19]: [3, 4, 5]
```

Mais abstratamente, tanto o início quanto o fim podem ser deixados em branco, colocando-se somente o passo.

```
In [25]: lista1[::2]
```

```
Out[25]: [1, 3, 5]
```

```
In [28]: lista1[::-1] # Uma maneira fácil de obter o inverso de uma lista
```

```
Out[28]: [5, 4, 3, 2, 1]
```

É importante notar que somente números inteiros podem ser utilizados na indexação. Não faz sentido o número na posição 1.5!

Um exemplo prático para o uso de listas no tratamento de dados é uma lista com os nomes dos arquivos que serão tratados. Suponha que somente deseja-se tratar metade dos arquivos de uma certa maneira, então é fácil utilizar somente a primeira metade da lista. Por exemplo:

```
In [59]: dados = ['Arq01', 'Arq02', 'Arq03', 'Arq04', 'Arq05', 'Arq06', 'Arq07', 'Arq08']
          primeira_metade = dados[:4]
          segunda_metade = dados[4:] # Já que o número final não está incluso,
                                     # é possível começar e terminar do mesmo índice sem haver r
          primeira_metade
```

```
Out[59]: ['Arq01', 'Arq02', 'Arq03', 'Arq04']
```

É possível utilizar a função `len` para descobrir o comprimento de uma lista, e de vários outros objetos.

```
In [52]: len(dados)
```

```
Out[52]: 8
```

E a partir disso, é possível separar uma lista em duas partes de maneira abstrata, sem ter que saber o comprimento real. Note que utilizou-se a divisão *floor*, pois somente números inteiros podem ser índices de listas.

```
In [58]: primeira_metade = dados[:len(dados) // 2]
        segunda_metade = dados[len(dados) // 2:]
        primeira_metade
```

```
Out[58]: ['Arq01', 'Arq02', 'Arq03', 'Arq04']
```

Também é possível obter o maior ou menor elemento de uma lista utilizando as funções **max** e **min**.

```
In [61]: max(lista1)
```

```
Out[61]: 5
```

```
In [62]: min(lista1)
```

```
Out[62]: 1
```

Da mesma maneira que é possível obter elementos de uma lista, é possível também alterá-los, utilizando a indexação, da mesma maneira que se altera uma variável.

```
In [40]: lista1[0] = 100
        lista1
```

```
Out[40]: [100, 2, 3, 4, 5]
```

Também é possível remover um elemento específico de uma lista utilizando o comando **del**. Esse comando serve também para deletar qualquer variável presente.

```
In [46]: del lista1[0]
        lista1
```

```
Out[46]: [2, 3, 4, 5]
```

2.3 Métodos de listas

Alguns dos métodos mais comumente utilizados para listas são:

- **append**, coloca um termo no final da lista
- **sort**, organiza os elementos de uma lista baseado num critério.
- **pop**, retorna o último elemento da lista e remove-o dela

Exemplo:

```
In [55]: dados.reverse()
        dados
```

```
Out[55]: ['Arq01', 'Arq02', 'Arq03', 'Arq04', 'Arq05', 'Arq06', 'Arq07', 'Arq08']
```

É possível utilizar o operador de soma para concatenar duas listas.

```
In [2]: a = [1, 2, 3]
        b = [4, 5, 6]
        c = a + b
        c
```

```
Out[2]: [1, 2, 3, 4, 5, 6]
```

3 Mutabilidade e Imutabilidade

Veja que *append* e *pop* alteram os itens da lista, colocando ou removendo. Também, vimos que elementos internos de listas podem ser alterados sabendo seus índices. Isso caracteriza as listas como sendo mutáveis. Veja o que acontece quando tentamos alterar um elemento de uma string.

```
In [45]: string1 = 'Isto é imutável'
        string1[7] = ''
```

```
-----
TypeError                                Traceback (most recent call last)
```

```
<ipython-input-45-766c15d29d75> in <module>()
      1 string1 = 'Isto é imutável'
----> 2 string1[7] = ''
```

```
TypeError: 'str' object does not support item assignment
```

Da mesma maneira, veja o que acontece quando tentamos remover um dos elementos de uma string.

```
In [48]: del string1[7]
```

```
-----
TypeError                                Traceback (most recent call last)
```

```
<ipython-input-48-41711ea8eec7> in <module>()
----> 1 del string1[7]
```

```
TypeError: 'str' object doesn't support item deletion
```

Isso pode parecer perfumaria, e geralmente esse tipo de coisa não aparece na rotina do tratamento de dados. Porém, é importante saber disso, pois certas operações podem ser realizadas somente com objetos mutáveis ou imutáveis. E note também que anteriormente havíamos somado (concatenado) duas strings, aparentemente violando esse princípio. O que aconteceu na verdade é que foi criada uma nova string a partir das duas outras, que permaneceram intactas. Outros tipos imutáveis já vistos são os ints e os floats.

4 Dicionários

Dicionários são estruturas de dados que relacionam duas coisas. Por exemplo, um dicionário, no sentido habitual, relaciona uma explicação a uma palavra. Dicionários em Python relacionam

uma chave (key) a algum outro item. Necessariamente as chaves devem ser imutáveis, tornando strings ótimos candidatos. Dicionários são criados de duas maneiras, ou utilizando-se chaves {}, com chave:item, separados por vírgulas, ou atribuindo depois da criação. Por exemplo.

```
In [65]: dict1 = {'Nome': 'Sophia', 'Idade': 12, 'Personalidade': 'Extrovertida'}
```

```
dict2 = {}
dict2['Nome'] = 'Eva'
dict2['Idade'] = 8
dict2['Personalidade'] = 'Introvertida'

dict1
```

```
Out [65]: {'Idade': 12, 'Nome': 'Sophia', 'Personalidade': 'Extrovertida'}
```

Note que na hora de representar um dicionários, a ordem aparente dos itens não é igual à ordem que os itens foram criados, como em listas. Isso depende da maneira que o Python armazena esses valores na memória, e não é muito importante para este curso.

Para acessar os elementos de um dicionário, utiliza-se colchetes com a chave dentro, da mesma maneira que valores podem ser atribuídos.

```
In [50]: dict1['Nome']
```

```
Out [50]: 'Sophia'
```

Um exemplo da utilidade de dicionários. Suponha que você tenha vários espectros de UV-Vis. Porém, cada experimento necessita somente dos dados a partir de um certo comprimento de onda. Ao invés de ter que colocar as regiões uma a uma no tratamento, é possível criar um dicionário que contém o nome do arquivo e o comprimento de onda inicial. Dessa maneira, essas informações ficam agregadas em um único local de fácil acesso.

4.1 Métodos de dicionários

Os métodos mais utilizados são:

- `.keys()` retorna as chaves de um dicionário
- `.values()` retorna os valores

Não se esqueça que é possível obter informações sobre todos os métodos presentes, e ajuda sobre cada método, com as funções **dir** e **help**.

```
In [63]: dict1.keys()
```

```
Out [63]: dict_keys(['Nome', 'Idade', 'Personalidade'])
```

```
In [64]: dict2.values()
```

```
Out [64]: dict_values(['Eva', 8, 'Introvertida'])
```

5 Tuples

Tuples são, basicamente, listas imutáveis. São criadas com parênteses, ao invés de colchetes, e retêm a ordem, ao contrário de dicionários. A principal utilidade de tuples é para passar e receber argumentos de funções, e como chaves em dicionários. O chamado *tuple assignment* é um método para atribuir valores para mais de uma variável em uma única linha.

```
In [70]: (a, b) = (3, 4)
         c, d = 7, 6

         a + b + c + d
```

```
Out[70]: 20
```

Note que nesse caso, não é necessário colocar os parênteses. Isso acontece com bastante frequência com tuples. Se você deseja explicitamente criar uma tuple com um único termo, deve colocar uma vírgula após o primeiro valor, antes do parêntese.

```
In [71]: e = (1,)
         type(e)
```

```
Out[71]: tuple
```