Introdução ao conceito de lista

MC102-2018s1-Aula07-180320-takeaway

Arthur J. Catto, PhD
ajcatto@g.unicamp.br

20 de março de 2018

0.1 Introdução ao conceito de *lista*

Uma *lista* é uma sequência ordenada de objetos, não necessariamente do mesmo tipo, cada um deles identificado por um *índice* indicando sua posição na *lista*.

- Uma lista vazia é representada como `[]`.

```
In [1]: lst = []
        print(type(lst), lst)

        lst = [1, 2, 3.14, 'abc']
        print(type(lst), lst)
```

```
<class 'list'> []
<class 'list'> [1, 2, 3.14, 'abc']
```

- Os *índices* de uma lista são inteiros não-negativos, consecutivos, começando por `0`.
- O item na posição `i` de uma lista `lst` é referenciado por `lst[i]`.
- O último item de uma lista `lst` também pode ser referenciado por `lst[-1]`, o penúltimo por `lst[-2]`, etc.

```
In [5]: lst = [1, 2, 3.14, 'abc']
        print(type(lst), lst)

        print(type(lst[0]), lst[0], lst[1], lst[2], lst[3])
        print(type(lst[-1]), lst[-1], lst[-2], lst[-3], lst[-4])
```

```
<class 'list'> [1, 2, 3.14, 'abc']
<class 'int'> 1 2 3.14 abc
<class 'str'> abc 3.14 2 1
```

- Um item `x` pode ser adicionado a uma lista `lst` executando-se `lst.append(x)` ou `lst = lst + [x]`.

```
In [15]: lst = [1, 2, 3.14, 'abc']
         print(type(lst), lst)

         lst.append(56)
```

```
        print(type(lst), lst)

        lst = ['xyz'] + lst + list('etc')
        print(type(lst), lst)
```

```
<class 'list'> [1, 2, 3.14, 'abc']
<class 'list'> [1, 2, 3.14, 'abc', 56]
<class 'list'> ['xyz', 1, 2, 3.14, 'abc', 56, 'e', 't', 'c']
```

- A primeira ocorrência de um valor `x` numa lista `lst` pode ser removida executando-se `lst.remove(x)`.
- O item na posição `i` da lista `lst` pode ser removido executando-se `lst.pop(i)`.

```
In [19]: lst = [1, 'abc', 2, 3.14, 'abc ']
         print(type(lst), lst)

         x = lst.remove('abc')
         print(type(lst), x, lst)

         x = lst.pop(3)
         print(type(lst), type(x), repr(x), lst)
```

```
<class 'list'> [1, 'abc', 2, 3.14, 'abc ']
<class 'list'> None [1, 2, 3.14, 'abc ']
<class 'list'> <class 'str'> 'abc ' [1, 2, 3.14]
```

0.2 O comando for

Este é um recurso poderoso e versátil para a implementação de iterações.

Sua forma básica é...

```
for variável in objeto_iterável:
    suite
```

Um `objeto_iterável` é algum objeto composto que possa fornecer itens, um a um, p.ex. listas, strings, tuplas, etc.

Um comando `for` extrai um item do `objeto_iterável`, associa esse item à `variável` e executa a `suite`.

Quando a `suite` termina, o processo se repete.

Quando o `objeto_iterável` se esgota, o comando `for` termina. Se, no início, o `objeto_iterável` já estiver vazio, o comando `for` termina sem produzir qualquer efeito.

Examine o código abaixo...

```
In [34]: nomes = [123, "Maria", 3.14, "Ana"]
         for estudante in nomes:
             # print("Olá, " + estudante + ". Você está gostando de MC102?")
             print(type(estudante), repr(estudante))
         print(nomes, estudante)
```

```
<class 'int'> 123
<class 'str'> 'Maria'
```

```
<class 'float'> 3.14
<class 'str'> 'Ana'
[123, 'Maria', 3.14, 'Ana'] Ana
```

```
In [39]: [3 + 3]

Out[39]: [6]
```

Vamos examinar a estrutura do `for` neste exemplo:

- Na linha 1, `estudante` representa a *variável* do loop.
- Os nomes entre colchetes compõem uma *lista* (que é um *objeto_iterável*).
- O operador `in` tem o significado `que faça parte de`.
- A linha 2 é a *suite*. Uma *suite* é composta por um ou mais comandos igualmente indentados e fica recuada em relação ao `for` (tipicamente 4 espaços).

Você pode ler esse comando como... > para todo `estudante` que faça parte da *lista* execute a *suite*

```
for estudante in ["José", "Maria", "Francisco", "Ana"]:
    print("Olá,", estudante + ".", "Você está gostando de MC102?")
```

Vamos entender como esse `for` funciona:

- O comando começa examinando o *objeto_iterável*, neste caso uma lista.
- Como ela não está vazia, a *variável* é associada ao primeiro item da lista, neste caso `'José'` e a *suite* é executada.
- No final de uma execução da *suite*, Python retorna ao início para ver se há mais itens a serem processados.
- Se não houver nenhum, o **for** termina e a execução do programa continua no comando seguinte à *suite*.
- Caso contrário, a *variável* é associada ao próximo item da *lista* e a *suite* é executada novamente.

Por exemplo, entenda e depois execute o código abaixo...

```
In [40]: for x in [0, 1, 2, 3, 4]:
             print(x, x ** 2)

0 0
1 1
2 4
3 9
4 16
```

É possível aninhar `for`s para gerar combinações. Por exemplo...

```
In [51]: marcas = ['Ford', 'Volkswagen', 'Kia']
         cores = ['preto', 'branco']

         for marca in marcas:
             for cor in cores:
                 # print(format(marca, '10'), cor)
                 print(f'{marca:5}  {cor}')
             print('-'*17)
         print('Fim da tabela')
```

```
Ford   preto
Ford   branco
-----------------
Volkswagen  preto
Volkswagen  branco
-----------------
Kia    preto
Kia    branco
-----------------
Fim da tabela
```

O aninhamento de `for`s também nos permite percorrer listas de listas. Por exemplo...

```
In [58]: llista = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
         for linha in llista:
             for x in linha:
                 print(x, end=' ')
             print()
```

```
11 12 13
21 22 23
31 32 33
```

Da mesma forma, poderíamos linearizar `llista`,...

```
In [60]: llista = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
         llin = []
         for linha in llista:
             for x in linha:
                 llin += [x]
         print(llin)
```

```
[11, 12, 13, 21, 22, 23, 31, 32, 33]
```

... mas neste caso dá pra fazer mais simples...

```
In [61]: llista = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
         llin = []
         for linha in llista:
             llin += linha
         print(llin)
```

```
[11, 12, 13, 21, 22, 23, 31, 32, 33]
```

O uso de uma sequência de inteiros como *objeto iterável* num `for` é frequente e mereceu uma função específica...

0.2.1 A função `range()`

Uma chamada `range(`*stop*`)` gera todos os valores inteiros pertencentes ao intervalo `[0..`*stop*`)`.

Note que o intervalo é fechado à esquerda (inclui o `0`), mas é aberto à direita (**não** inclui *stop*).

```
In [62]: for x in range(3):
             print(x)

0
1
2
```

Uma chamada `range(`*start*, *stop*`)` gera todos os valores inteiros pertencentes ao intervalo [*start* .. *stop*).

Note que o intervalo é fechado à esquerda (inclui *start*), mas é aberto à direita (**não** inclui *stop*).

```
In [66]: for x in range(4, 3):
             print(x)
```

Se *start* > *stop*, `range` não produz qualquer efeito, mas também não gera uma exceção.

```
In [ ]: for x in range(4, 3):
            print(x)
```

Finalmente, uma chamada `range(`*start*, *stop*, *step*`)` gera todos os valores inteiros no intervalo [*start*, *start* + *step*, ... *stop*), onde...

- *start*: limite inferior
- *stop*: limite superior
- *step*: tamanho do passo, isto é valor adicionado ao item atual para gerar o próximo

O intervalo é fechado à esquerda (inclui *start*), mas é aberto à direita (**não** inclui *stop*).

```
In [67]: for x in range(1, 10, 3):
             print(x)

1
4
7
```

Respeitada a definição, é possível usar valores negativos como argumentos numa chamada de `range`...

```
In [69]: for x in range(2,-4,-2):
             print(x)
```

```
2
0
-2
```

O fato de o intervalo ser fechado à esquerda e aberto à direita tem propriedades interessantes. Por exemplo, intervalos adjacentes fundem-se naturalmente...

```
In [70]: n = 8
         k = 5
         print('n =', n, '  k =', k)

n = 8   k = 5


In [71]: print('range(0, n) =', list(range(0, n)))

range(0, n) = [0, 1, 2, 3, 4, 5, 6, 7]


In [72]: print('range(0, k) =', list(range(0, k)))

range(0, k) = [0, 1, 2, 3, 4]


In [73]: print('range(k, n) =', list(range(k, n)))

range(k, n) = [5, 6, 7]


In [74]: print('range(0, k) + range(k, n) =',
               list(range(0, k)) + list(range(k, n)))

range(0, k) + range(k, n) = [0, 1, 2, 3, 4, 5, 6, 7]
```