# Estruturas de Dados em Python

Carlos Camarão

20 de Março de 2017

## 1 Introdução

Na segunda parte do curso vamos aprender a programar com estruturas (ou coleções) de dados, em Python.

Python é uma linguagem orientada a objetos. Toda variável em Python é de fato uma referência, para o valor propriamente dito. Por exemplo, o comando de atribuição:

```
x = 10
```

atribui o valor 10 à variável `x`, que é uma referência a um valor inteiro. Quando temos uma outra atribuição, como por exemplo:

```
x = 11
```

não é o valor inteiro 10 que muda; não é o valor 10 que é armazenado em `x`, diretamente. O valor que é armazenado em `a` é uma referência. É essa referência que muda. O valor armazenado em `x` passa a fazer referência ao valor 11, em vez de fazer referência ao valor 10. A referência contida em uma variável pode ser obtida, por meio da função pré-definida *id*. Por exemplo, `id(x)` é diferente, antes e após uma atribuição de valores diferentes a `x` (experimente usar o interpretador para verificar o valor de `id(x)` antes e depois de atribuições como acima).

Uma distinção básica de dados em Python é o fato de eles poderem ser ou não modificados: valores que podem ser modificados são chamados de *mutáveis* e valores que não podem de *imutáveis*.

Vamos abordar, nas seções seguintes: tuplas, cadeias de caracteres (*strings*), que são estruturas de dados imutáveis em Python, e listas, arquivos, dicionários e conjuntos, que são estruturas de dados mutáveis em Python.

Tipos têm características e funções e operações próprias, que vamos aprender a usar para resolver problemas que envolvem o uso de dados compostos, ou estruturados. Esses dados são compostos sempre por valores básicos, como inteiros, valores de ponto flutuante, booleanos e caracteres.

Chamamos de *operação* uma função que não é usada (chamada) com um nome seguido de argumentos entre parênteses, seu uso é feito com símbolos, usualmente tendo argumentos escritos entre os símbolos.

Por exemplo, uma coleção, como uma cadeia de caracteres ou uma lista, tem um operador de indexação: por exemplo a operação de indexar tal coleção `x` na posição ou índice `i` é representada por `x[i]`. É usado o operador `[]`, e o argumemto `i` é inserido entre os colchetes que compõem o operador.

Iniciamos cada seção com funções, operações e ações básicas, e depois mostramos funções, operações e ações adicionais sobre cada uma das estruturas de dados.

Chamamos de ação uma modificação de *estado* (função que associa cada variável do programa ao valor nela contido), ou seja, uma ação é tipicamente uma modificação no valor de alguma variável.

Em uma linguagem orientada por objetos, funções e ações podem ser chamadas de *métodos*, que são chamados tendo um *objeto alvo* da chamada: em vez de usar `f(x)` para uma função `f` e um argumento `x`, uma chamada de método `f` usa a notação `(x.f()`, sendo `x` o objeto alvo da chamada. Similarmente, para chamadas `f(x,y)`, em que `f` tem mais de um parâmetro, uma chamada de método `f` usa a notação `x.f(y)`, e assim por diante para casos em que `f` tem

mais parâmetros (similarmente, em uma chamada `x.f(y)`, `x` é o objeto alvo da chamada e `y` é argumento da chamada).

Todas as estruturas de dados que abordaremos (tuplas, cadeias de caracteres, listas, arquivos, dicionários e conjuntos) são casos de *iteradores*, como são chamados, em Python e em outras linguagens de programação, estruturas de dados que permitem que se retorne cada um de seus componentes, um de cada vez. Devido a essa característica, tais estruturas de dados podem ser usadas em comandos **for**, como ilustrado abaixo:

```
for elemento in x:
    c
```

Nesse comando, a estrutura de dados `x` deve gerar um iterador, como ocorre no caso de uma tupla, uma cadeia de caracteres, uma lista, um arquivo, um dicionário ou um conjunto. Todos essas estruturas de dados têm um iterador; `elemento` é um elemento distinto da estrutura de dados, retornado pelo iterador a cada iteração do comando **for**, podendo ser usado no corpo (`c`) do comando **for**.

Um iterador é retornado pela função **range** em Python, que pode ser usada com 1, 2 ou 3 argumentos. Podemos descrever a sintaxe como a seguir:

```
range([inic,]fim[,passo])
```

**range** retorna um iterador para um intervalo de valores inteiros; o intervalo não inclui o valor dado por `fim` e vai de `inic` até (exclusive) `fim` , se `inic` for especificado, senão de 0 até `fim`, de 1 em 1 se `passo` não for espeficicado ou se `inic` for positivo e menor que `fim`, de -1 em -1 se `passo` não for especificado ou se `inic` for negativo e menor que `fim`, e de `passo` em `passo`, se `passo` for especificado.

Os símbolos `>>>` e `...` em exmplos indicam interação com (entrada para) o interpretador Python. Uma linha sem os símbolos `>>>` após uma linha com estes símbolos indica que se trata de uma resposta do interpretador. Por exemplo:

```
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> for i in range(1, 3):
...     print(i)
...
1
2
>>> for i in range(1, 5, 2):
...     print(i)
...
1
3
>>> for i in range(0, -6, -2):
...     print(i)
...
0
-2
-4
```

## 2 Tuplas

Uma tupla (dupla, tripla, quádrupla etc.) é uma sequência de valores distinguidos de acordo com a posição em que aparecem na tupla. Em uma tupla, os valores componentes são separados por vírgula.

Por exemplo:

```
dupla  = ("abc", 3)
tripla = (1, True, "1")
```

O teste de pertinência (**in** em Python) pode ser usado com tuplas em Python (e com qualquer coleção: listas, cadeias de caracteres, dicionários). Por exemplo:

```
3 in dupla
```

retorna True.

Em Python, tuplas são sequências de valores quaisquer separados por ",". Não é necessário usar parênteses. Por exemplo:

```
a, b = b, a
```

é um modo sucinto de trocar os valores armazenados nas variáveis `a` e `b` em Python, sem necessidade de usar explicitamente uma variável temporária.

Para acesso aos elementos de uma tupla, é usado em geral casamento de padrões, como no seguinte exemplo:

```
def primeiro(a, b):
    return a
```

No entanto, podemos também obter acesso aos elementos de uma tupla com indexação, sendo que os índices iniciam com 0. Por exemplo: `dupla [0]` retorna o primeiro elemento de `dupla`.

# 3 Cadeias de caracteres

Um tipo cadeia de caracteres (ou simplesmente cadeia, em inglês *string*) é um tipo pré-definido em Python.

Seus componentes são caracteres. No entanto, em Python não existe um tipo *caractere*: em Python um caractere não pode ser distinguido de uma cadeia de tamanho 1 (i.e. uma cadeia com um único elemento).

Uma cadeia tem um certo tamanho (número de elementos): uma cadeia $c$ de tamanho $n$ tem $n$ elementos que podem ser usados por meio de índices, que vão de `0` até $n$`-1`. Uma cadeia de caracteres é imutável em Python (um caractere de uma cadeia não pode ser modificado).

Um valor *literal* de uma cadeia de caracteres é denotado entre aspas; podem ser entre aspas simples, entre aspas duplas ou, no caso de precisarem ocupar mais de uma linha, entre três aspas simpes ou entre três aspas duplas. Por exemplo, `'abc'` e `"abc"` são cadeias de caracteres de tamanho 3 (i.e. com 3 caracteres). Exemplos de cadeias de caracteres que podem ser escritos em mais de uma linha:

```
c1 = '''abc
def'''
c2 = """abc
def"""
```

Após as atribuições acima, `c1` e `c2` contêm a cadeia `'abc\ndef'`.

Considere $c$ uma cadeia de caracteres de tamanho $n$. As operações básicas predefinidas em Python sobre cadeias de caracteres são:

1. Indexação (acesso a um elemento): $c$`[`$i$`]` retorna o caractere na $i$-ésima posição da cadeia, se $i$ denota um índice entre 0 e $n-1$ (lembre-se: "caractere" em Python significa "cadeia-de-caracteres de tamanho 1").

   Por exemplo: `"abc"[1]` retorna `"b"`.

   O índice $i$ pode também ser negativo, de $-1$ até $-n$, sendo $c$`[-1]` o último elemento, $c$`[-2]` o penúltimo, assim por diante, até $c$`[`$-n$`]` que é o primeiro elemento.

   Se o índice $i$ for maior que $n-1$ ou menor que $-n$, ocorre uma exceção (exceções não serão abordadas neste curso).

2. Concatenação (justaposição): a cadeia $c$ `+` $d$ consiste dos caracteres de $c$ seguidos dos caracteres de $d$.

   Por exemplo: `"abc" + "de"` é igual a `"abcde"`.

Algumas operações adicionais em cadeias de caracteres são:

1. Fatiamento (extração de sub-cadeia): $c$`[`$i$`:`$j$ retorna a cadeia de caracteres do índice $i$ ao índice $j-1$. Os valores são opcionais: o índice inicial $i$ é igual a 0 se não especificado e o índice final $j$ é igual a $n-1$ se não especificado.

   Por exemplo:
   ```
   "abcde"[2:4]   é igual a "cd"
   "abcde"[:2]    é igual a "ab"
   "abcde"[2:]    é igual a "cde"
   ```

   Um passo diferente de 1 pode ser especificado. A cadeia $c$`[`$i$`:`$j$`:`$k$`]` retorna a cadeia que inicia no índice $i$ de $c$, e vai até $j-1$ de $k$ em $k$ elementos, se $k > 0$.

   Por exemplo, `"abcde"[1:4:2]` é igual a `"bd"`.

   Se $k < 0$, a cadeia é percorrida no sentido inverso, de $j$ até $i$`+1` de $|k|$ em $|k|$ (sendo $|k|$ o valor absoluto de $k$).

   Por exemplo, `"abcde"[4:1:2]` é igual a `"ec"`.

2. Cópia (multiplicação): $n$ `*` $c$, ou $c$ `*` $n$, retorna a cadeia que consiste em $n$ cópias da cadeia $c$, concatenadas.

   Por exemplo: `2 * "abc"` é igual a `"abcabc"`.

3. Teste de existência de sub-cadeia: `d` **`in`** `c` retorna `True` se e somente se `d` é uma sub-cadeia de `c`.

   `d` **`not in`** `c` retorna `True` se e somente se `d` não é uma sub-cadeia de `c`.

   Por exemplo:
   ```
   "cd" in "abcde"       é igual a True
   "ce" in "abcde"       é igual a False
   "ce" not in "abcde"   é igual a True
   ```

# 4 Listas

Listas em Python são similares a tuplas, pois podem conter elementos de tipos distintos, embora usualmente os elementos de listas tenham todos o mesmo tipo.

A diferença principal entre listas e tuplas em Python é que listas são mutáveis, enquanto tuplas são imutáveis. Por exemplo:

```
>>> x = [1,2,3]
>>> x[0] = 4
```

A indexação de uma lista com índice $i$ fornece o $i$-ésimo componente da lista (índices começam de 0 e vão até $n-1$, onde $n$ é o tamanho da lista). No exemplo, a primeira posição (de índice 0) de `x` é modificada, passando a conter, após a atribuição, o valor 4.

Outra diferença entre tuplas e listas é que literais de listas são escritos entre colchetes, sendo separados também por vírgulas, como no caso de tuplas.

Algumas funções, operações e ações pré-definidas em Python sobre listas são:

- `x.insert(i,e)`: insere `e` na posição `i` de `x`. Por exemplo:

  ```
  >>> x = [1,2,3]
  >>> x.insert(1,4)
  >>> x
  [1,4,2,3]
  ```

- `x.append(e)`: insere `e` no fim de `x`. Por exemplo:

```
>>> x = [1,2,3]
>>> x.append(4)
>>> x
[1,2,3,4]
```

- `x+y`, `x.extend(y)`: concatenação de `y` a (no final de) `x`, no primeiro caso apenas o resultado, no segundo modificação do valor de `x`.

  Ou seja, `x.append(y)` é o mesmo que `x = x + y` ou, abreviadamente, `x += y`.

  Por exemplo:

```
>>> x = [1,2,3]
>>> x+[4,5]
[1,2,3,4,5]
>>> x
[1,2,3]
>>> x.extend([4,5])
>>> x
[1, 2, 3, 4, 5]
```

- `x.pop(i)`: remove e retorna o elemento de índice `i` de `x`; o índice pode não existir (i.e. pode-se usar `x.pop()`), e nesse caso o último elemento de `x` é removido de `x` e retornado. Por exemplo:

```
>>> x = [1,2,3]
>>> x.pop()
3
>>> x
[1, 2]
>> x.pop(0)
1
>> x
[2]
```

- `x.remove(e)`: remove a primeira ocorrência do elemento `e` em `x`; ocorre exceção `ValueError` se `e` não ocorrem em `x` (exceções não serão abordadas neste curso). Por exemplo:

```
>>> x = [1,2,3,1]
>>> x.remove(1)
>>> x
[2,3,1]
```

Temos também as seguintes funções e ações:

- `x.count(e)`: retorna o número de ocorrências de `e` em `x`. Por exemplo:

```
>>> x = [1,2,3,1]
>>> x.count(1)
2
>>> x.count(4)
0
```

- `x.index(e)`: retorna o índice de `e` em `x`; ocorre exceção `ValueError` se `e` não ocorre em `x` (exceções não serão abordadas neste curso). Por exemplo:

```
>>> x = [1,2,3,1]
>>> x.index(1)
0
>>> x.pop(0)
>>> x
[2,3,1]
>>> x.index(1)
2
```

- `x.reverse()`: inverte a ordem dos elementos em `x`. Por exemplo:

```
>>> x = [1,2,3]
>>> x.reverse()
>>> x
[3,2,1]
```

- **`sorted(x)`**, `x.sort()`: retorna `x` em ordem crescente, ordena em ordem crescente os elementos em `x` (equivalente a `x = sorted(x)`); é preciso que exista uma operação de comparação definida entre os elementos da lista, caso contrário ocorre um erro de tipo durante a execução. Por exemplo:

```
>>> x = [4,2,3,1]
>>> sorted(x)
[1,2,3,4]
>>> x
[4,2,3,1]
>>> x.sort()
>>> x
[1,2,3,4]
```

- **`min(x)`**,**`max(x)`**: retorna o menor (maior) elemento de `x`; é preciso que exista uma operação de comparação definida entre os elementos da lista, caso contrário ocorre um erro de tipo durante a execução. Por exemplo:

```
>>> x = [2,1,3]
>>> min(x)
1
>>> max(x)
3
```

- **`sum(x)`**: soma dos valores em `x`: soma dos valores em `x`; é preciso que exista uma operação de soma (`+`) entre os valores de `x`, caso contrário ocorre um erro de tipo durante a execução. Por exemplo:

```
>>> x = [1,2,3]
>>> sum(x)
6
```

- **`len(x)`**: número de elementos em `x`. Por exemplo:

```
>>> x = [1,2,3]
>>> len(x)
3
```

- `x * k`, `k * x`: retorna k cópias de `x`. Por exemplo:

```
>>> x = [1,2,3]
>>> 3 * x
[1,2,3,1,2,3,1,2,3]
```

# 5 Conjuntos

A linguagem Python provê dois tipos de conjuntos, um imutável, **frozenset**, e o outro mutável, **set**. Conjuntos são tipos abstratos, isto é, cujas características são determinadas apenas pelo conjunto de funções e operações definidas sobre valores do tipo, e para o qual a representação dos valores não é conhecida.

Conjuntos podem ser criados a partir de outros conjuntos usando o método `add`. O exemplo a seguir mostra um conjunto com um único elemento criado a partir do conjunto vazio, denotado em Python por `set()`:

```
>>> x = set()
>>> x.add(1)
>>> x
{1}
```

Um conjunto pode ser criado especificando seus elementos entre chaves (a ordem dos elementos não é significativa em um conjunto, e não há mais de uma ocorrência de um elemento), ou a partir de uma lista (ou tupla, ou qualquer coleção que tenha um iterador), usando **set**. Por exemplo:

```
>>> x = {3,1,2,3}
>>> x
{1,2,3}
>>> y = [4,5,6,5]
>>> x = set(y)
>>> x
{4,5,6}
```

Temos ainda as seguintes ações e funções com conjuntos:

- `x.remove(e)`: remove `e` de elemento presente no conjunto `x`; ocorre uma exceção (`KeyError`) se `e` não for elemento de `x`. Por exemplo:

```
>>> x = {1,2,3}
>>> x.remove(2)
>>> x
{1,3}
```

- `e in x`: teste de pertinência, i.e. retorna `True` se e somente se `e` é elemento de `x`. Por exemplo:

```
>>> x = {1,2,3}
>>> 2 in x
True
>>> 4 in x
False
```

- `x | y`, `x & y`, `x - y`: operação de união, intersecção e diferença entre conjuntos. Por exemplo:

```
>>> x = {1,2,3}
>>> y = {2,4,6}
>>> x | y
{1,2,3,4,6}
>>> x & y
{2}
>>> x - y
{1,3}
```

Conjuntos imutáveis são similares: são criados com **frozenset** em vez de **set**, e não podem ter elementos adicionados (não se pode usar `add`), nem removidos (não se pode usar `remove`). Operações de união, intersecção e diferença entre conjuntos imutáveis são mais eficientes do que as correspondentes com conjuntos mutáveis.

# 6 Dicionários

Um dicionário é uma estrutura de dados que mapeia valores de entrada, usualmente chamados de *chaves*, a valores de saída, onde para cada chave corresponde um único valor de saída.

Em Python qualquer valor imutável pode funcionar como chave em um dicionário; isso é feito em Python obtendo o valor fornecido pelo método `__hash__` com o valor de entrada como objeto alvo da chamada (sem parâmetros na chamada), automaticamente, sem necessidade de chamada do método.

A criação de dicionários pode ser feita por meio de diversos modos, usando listas ou conjuntos, de pares que associam um primeiro componente que é o valor da chave a seu valor correspondente. Por exemplo, `x1` a `x4` definidos a seguir denotam o mesmo dicionário (a função **zip** recebe um par de listas e retorna um iterador de pares de valores, tomandos elementos de cada lista até o fim de uma das listas):

```
>>> x1 = {'a':1, 'b':2,'c':3}
>>> x2 = {'c':3, 'b':2,'a':1}
>>> x3 = dict(zip('aabc', [0,1,2,3]))
>>> x4 = dict(a=1, b=2, c=3)
```

A segunda ocorrência do par `('a':1)` no iterador retornado por **zip** sobrepõe o par `('a':0)` anterior na definição do dicionário (apenas um valor ocorre para cada chave em um dicionário).

O iterador retornado pela função **zip** pode ser usado, com várias listas, em comandos **for** em Python, como exemplificado a seguir (note separação dos campos de `dado` em três componentes na iteração do comando **for** a seguir e o uso de um argumento com nome do parâmetro `sep` da função **print**):

```
>>> x = {'Joao', 'Maria', 'Jose'}
>>> y = {'Rio de Janeiro', 'Sao Paulo', 'Belo Horizonte'}
>>> z = {20, 30, 40}
>>> for dado in zip (x,y,z):
        nome, info1, info2 = dado
        print (nome, info1, info2, sep='\t')
```

A operação básica sobre dicionários é a indexação. Um dicionário indexado por uma chave fornece o valor correspondente à chave.

Se usado como uma expressão (para obter um valor), ocorre uma exceção (`KeyError`) se o valor fornecido não for uma chave do dicionário, mas se usado como uma variável, do lado esquerdo de um comando de atribuição, a indexação permite adicionar nova chave ao dicionário. Por exemplo:

```
>>> x = {'a':1, 'b':2,'c':3}
>>> x['b']
2
>>> x['d'] = 4
>>> x
{'c': 3, 'd': 4, 'b': 2, 'a':1}
```

O método `get` com um dicionário como objeto alvo permite que seja especificado, opcionalmente, um valor default, como segundo argumento, que é retornado se o primeiro argumento não é uma chave do dicionário; se um valor default não for especificado e o primeiro argumento não for uma chave do dicionário, o valor retornado é `None`. O valor `None` é o valor de uma referência nula em Python.

Por exemplo:

```
>>> x = {'a':1, 'b':2,'c':3}
>>> x.get('d', 0)
0
```

Temos ainda as seguintes ações e métodos sobre dicionários em Python:

- `x.pop(k)`: retorna e remove entrada com chave `k` de `x`; ocorre uma exceção (`KeyError`) se `k` não for uma chave de `x`. Por exemplo:

```
>>> x = {'a':1, 'b':2,'c':3}
>>> x.pop('b')
2
>>> x
{'c': 3, 'a':1}
```

  Um valor default pode ser especificado, como segundo argumento de `pop`, como valor retornado se o primeiro argumento não for uma chave do dicionário. Por exemplo:

```
>>> x = {'a':1, 'b':2,'c':3}
>>> x.pop('d',0)
0
```

- *x.update*({*k* : *v*}): o mesmo que `x[k] = v`. Por exemplo:

```
>>> x = {'a':1, 'b':2,'c':3}
>>> x.update({'b':0})
>>> x
{'c': 3, 'b': 0, 'a':1}
```

- `x.setdefault(k,v)`: retorna `v` e atualiza o dicionário `x` se `k` não for uma chave de `x`, do contrário funciona como `get` (i.e. apenas retorna o valor da entrada `k` de `x`). Por exemplo:

```
>>> x = {'a':1, 'b':2,'c':3}
>>> x.setdefault({'d':4})
4
>>> x
{'c': 3, 'd':4, 'b': 2, 'a':1}
>>> x.setdefault({'d':5})
4
>>> x
{'c': 3, 'd':4, 'b': 2, 'a':1}
```