

Tipos Estruturados Não Sequenciais

Hashable: o que é isso?

Antes de começar, duas considerações: este é um assunto que poderia ter sido tratado anteriormente, mas foi deixado para esta aula para evitar que o programador iniciante fosse abarrotado com conceitos específicos e aprofundados de Python antes de dominar seus aspectos mais básicos; o termo técnico *hashable*, do inglês, não tem uma tradução em português, de modo que esse será o termo utilizado daqui para a frente.

Diz-se que um objeto é *hashable* quando ele tem um valor *hash* que não se altera durante o ciclo de vida do objeto e que pode ser comparado ao *hash* de outros objetos.

Neste contexto, *hash* é um número inteiro calculado a partir do conteúdo de um objeto e pode ser verificado com o uso da função `hash`, como mostrado no Exemplo 6.1. Nele, são criadas duas variáveis string (poderia ser qualquer outro tipo, com qualquer conteúdo) contendo o mesmo conjunto de caracteres. Os números id de cada objeto são exibidos com o propósito de mostrar que são ids diferentes. E a função `hash` é usada com os dois objetos para mostrar que ambos têm o mesmo valor de *hash*. Isto ocorre porque esse número é calculado a partir do conteúdo do objeto. Quando se escreve uma condição de igualdade (`T==Z`) ou diferença (`T!=Z`), o interpretador usa os valores de *hash* de cada um dos objetos para avaliar o resultado.

Da definição dada anteriormente e considerando que todos os tipos imutáveis não podem ter seu valor alterado, então, conclui-se que eles são *hashable*, ao passo que os objetos mutáveis não o são. A única exceção é que as tuplas que contêm tipos mutáveis não são *hashable*.

Exemplo 6.1 Exemplo de *hash* de objetos

```
>>> T = 'Texto'
```

```
>>> id(T) 48527360
```

```
>>> hash(T)
```

```
-1151134949
```

```
>>> Z = 'Texto'
```



```
>>> id(Z) 48527360
```

```
>>> hash(Z)
```

```
-1151134949
```

```
>>> T == Z
```

```
True
```

```
>>> x = (3, 6, 9) # tuplas são hashable
```

```
>>> type(x)
```

```
<class 'tuple'>
```

```
>>> hash(x)
```

```
-149728741
```

```
>>> y = (3, 6, 9, [2, 4]) # tuplas que contêm listas não são
```

```
>>> type(y) # hashable
```

```
<class 'tuple'>
```

```
>>> hash(y)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#141>", line 1, in <module> hash(y)
```

```
TypeError: unhashable type: 'list'
```



```
>>> L = [1, 2, 3] # listas não são hashable
```

```
>>> hash(L)
```

Traceback (most recent call last):

File "<pyshell#143>", line 1, in <module> hash(L)

TypeError: unhashable type: 'list'

Conjuntos

O tipo conjunto é uma coleção não ordenada de elementos não repetidos. Esse tipo suporta as operações características da Teoria dos Conjuntos, um ramo da matemática, tais como união, interseção e diferença.

Por definição, dentro do conjunto, só haverá uma ocorrência de cada elemento, independentemente de quantas vezes se tente adicioná-lo. Um conjunto pode ser criado de duas maneiras: utilizando dados entre chaves, { } ou a função set. É possível existir um conjunto vazio. E apenas objetos *hashable* podem ser membros de um conjunto.

Em Python existem dois tipos de conjunto: o set e o frozenset. Em praticamente tudo eles são iguais, a única e fundamental diferença entre ambos é que o frozenset é imutável e, uma vez criado, não pode ter seus membros alterados, incluídos ou removidos.

O Exemplo 6.2 ilustra diversas situações possíveis para a criação de conjuntos. No caso 1, foram utilizadas as chaves e, dentro delas, valores numéricos. Com isso, o conjunto "a" resultante contém objetos numéricos. No caso 2 foi utilizada a função set com um string de parâmetro e o resultado obtido é o desmembramento do string, de modo que cada caractere seja um elemento do conjunto. No caso 3 foi utilizado um string entre chaves e não houve desmembramento, resultando em um conjunto com um único elemento.

Exemplo 6.2 Criação de conjunto

```
>>> a = {1, 2, 3, 4, 5} # caso 1
```



```
>>> a
```

```
{1, 2, 3, 4, 5} # cria um set com cinco elementos
```

```
>>> type(a)
```

```
<class 'set'>
```

```
>>> len(a) # a função len() pode ser usada 5
```

```
>>> b = set('12345') # caso 2
```

```
>>> b
```

```
{'1', '4', '5', '3', '2'} # cria um novo set com cinco elementos
```

```
>>> type(b)
```

```
<class 'set'>
```

```
>>> c = {'12345'} # caso 3
```

```
>>> c
```

```
{'12345'} # cria um conjunto com um elemento
```

```
>>> type(c)
```

```
<class 'set'>
```

```
>>> d = {} # este comando não cria um conjunto
```

```
>>> type(d) # vazio, mas sim um dicionário
```



```
<class 'dict'>
```

```
>>> d = set() # conjuntos vazios são criados assim  
>>> type(d)
```

```
<class 'set'>
```

É possível criar um conjunto vazio e adicionar elementos posteriormente, porém, deve-se tomar um cuidado. Conjuntos vazios devem ser criados com a função `set` sem passar qualquer parâmetro. O uso de chaves sem conteúdo criará um dicionário vazio, e não um conjunto, como já mostrado.

Conteúdos de listas e tuplas podem ser utilizados para produzir conjuntos, e vice-versa. O Exemplo 6.3 mostra a conversão de uma lista para conjunto. Note que, ao fazer essa operação, os elementos repetidos presentes na lista foram eliminados no conjunto. Essa é uma maneira conveniente de eliminar repetições de valores em listas e tuplas, pois é possível converter de volta o conjunto para uma lista ou tupla.

Exemplo 6.3 Conversão entre listas, tuplas e conjuntos

```
>>> L = [3, 7, 9, 16, 3, 8, 14, 9, 25] # lista c/ elem. repetidos
```

```
>>> a = set(L) # conversão da lista em conjunto
```

```
>>> a
```

```
{3, 7, 8, 9, 14, 16, 25} # apenas uma ocorrência de cada valor
```

```
>>> L = list(a) # converte de volta o conjunto para
```

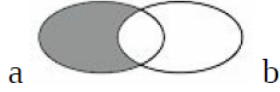

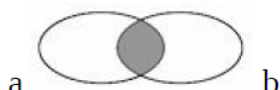
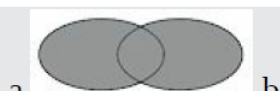
```
>>> L # lista
```

```
[3, 7, 8, 9, 14, 16, 25]
```



Operações com conjuntos

O Quadro 6.1 mostra os operadores aplicáveis aos conjuntos e o Exemplo 6.4 ilustra seu uso.

Operação	Nome	O que retorna (hachurado)
$a - b$	Diferença	
$a b$	União	
$a \& b$	Interseção	
$a \wedge b$	Diferença simétrica	
valor in a	Pertence	Valor está contido no conjunto a
valor not in a	Não pertence	Valor não está contido no conjunto a

Quadro 6.1 Operações aplicáveis aos conjuntos.

Exemplo 6.4 Operações com conjuntos

```
>>> a = set('abacaxi')
```

```
>>> a
```

```
{'a', 'x', 'i', 'b', 'c'} # não há objetos repetidos
```

```
>>> b = set('abacate')
```

```
>>> b
```

```
{'t', 'e', 'a', 'b', 'c'} # não há objetos repetidos
```

```
>>> a - b # elementos em a, porém, não em b
```



```
{'x', 'i'}
```

```
>>> a | b # elementos em a, em b ou em ambos
```

```
{'t', 'e', 'a', 'x', 'i', 'b', 'c'}
```

```
>>> a & b # elementos simultâneos em a e b
```

```
{'c', 'b', 'a'}
```

```
>>> a ^ b # elementos em a ou b, mas não em ambos
```

```
{'x', 'i', 't', 'e'}
```

```
>>> 'x' in a # 'x' está em a True
```

```
>>> 'r' in a # 'r' não está em a False
```

```
>>> 'r' not in a True
```

Métodos disponíveis

Em adição aos operadores, os conjuntos contam com um significativo número de métodos, que são utilizados para efetuar diversas tarefas. Os conjuntos podem ter elementos adicionados e removidos, pode ser zerado e atualizado, e o Quadro 6.2 mostra alguns deles.

Considere-se disponível a lista C

Método	Descrição
C.add(...)	Acrescenta um objeto ao conjunto.
C.clear()	Remove todos os elementos do conjunto.
C.copy()	Retorna uma cópia do conjunto.
C.difference(...)	Retorna um novo conjunto contendo a diferença de dois ou mais conjuntos.



C.difference_update(...)	Atualiza o conjunto C, removendo de seus elementos os que estejam no conjunto passado como parâmetro.
C.discard(...)	Se parâmetro estiver presente no conjunto, então o remove, caso não esteja não faz nada.
C.intersection(...)	Retorna um novo conjunto contendo a interseção de dois ou mais conjuntos.
C.intersection_update(...)	Atualiza o conjunto C com a interseção entre seus elementos e o conjunto passado como parâmetro.
C.isdisjoint(...)	Retorna <i>True</i> se os dois conjuntos têm interseção vazia e <i>False</i> , caso contrário.
C.issubset(...)	Retorna <i>True</i> se C é um subconjunto do conjunto passado como parâmetro.
C.issuperset(...)	Retorna <i>True</i> se C está contido no conjunto passado como parâmetro.
C.pop()	Remove e retorna um elemento arbitrário do conjunto C. Se o conjunto estiver vazio, gera uma exceção <i>KeyError</i> .
C.remove(...)	Remove do conjunto C um elemento que seja seu membro. Caso contrário, gera uma exceção <i>KeyError</i> .
	Considere-se disponível a lista C
Método	Descrição
C.symmetric_difference(...)	Atualiza o conjunto C com a diferença simétrica de seus elementos com o conjunto passado como parâmetro.



<code>C.symmetric_difference_update(...)</code>	Atualiza o conjunto C com a diferença simétrica de seus elementos com o conjunto passado como parâmetro.
<code>C.union(...)</code>	Retorna um novo conjunto com a união de C e o conjunto passado como parâmetro.
<code>C.update(...)</code>	Atualiza o conjunto C com a união de sim, mesmo com o conjunto passado como parâmetro.

Quadro 6.2 Métodos da classe “set” disponíveis ao programador.

O Exemplo 6.5 ilustra diversas situações de usos dos métodos listados no Quadro 6.2. Na maioria dos casos mostrados os métodos estão sendo usados com os conjuntos a e b. No entanto, os parâmetros passados para os métodos também podem ser listas e tuplas, como mostrado no final do exemplo.

Exemplo 6.5 Usos de conjuntos

```
>>> a = {1, 2, 3, 4, 5, 6} # define o conjunto a
```

```
>>> a.add(7) # adiciona elemento
```

```
>>> a.add(8) # adiciona outro elemento
```

```
>>> a.add(3) # esta adição não tem efeito, pois 3
```

```
>>> a # já está na lista
```

```
{1, 2, 3, 4, 5, 6, 7, 8} # conjunto ampliado
```

```
>>> b = {2, 4, 6}
```

```
>>> b.issubset(a) # o conjunto b é subconjunto de a
True
```



```
>>> a.issubset(b) # o conjunto a não é subconjunto de b False
```

```
>>> a.issuperset(b) # b está contido em a True
```

```
>>> b.add(10) # adiciona novo elemento em b
```

```
>>> b # não se tem controle sobre a posição
```

```
{2, 10, 4, 6} # onde o novo elemento é inserido
```

```
>>> a.issubset(b) # b não é mais subconjunto de a False
```

```
>>> c = a.difference(b) # gera o conjunto c contendo a-b
```

```
>>> c
```

```
{1, 3, 5, 7, 8}
```

```
>>> c = b.difference(a) # gera o conjunto c contendo b-a
```

```
>>> c
```

```
{10}
```

```
>>> >>> a.difference_update(b) # atualiza a com a-b
```

```
>>> a
```

```
{1, 3, 5, 7, 8}
```

```
>>> x = 5
```

```
>>> a.remove(x) # remove de a o valor contido em x
```

```
>>> a
```



```
{1, 3, 7, 8}
```

```
>>> a.pop() # retorna e remove algum elemento de a 1
```

```
>>> a
```

```
{3, 7, 8}
```

```
>>> b
```

```
{2, 10, 4, 6}
```

```
>>> a.isdisjoint(b) # a interseção de a e b é vazia True # e
```

```
>>> a.intersection(b) # este método comprova isso set()
```

```
>>> a.issubset(range(1, 10)) # testa se a é subconjunto de um  
True # range
```

```
>>> a = {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
>>> L = [5, 9, 'a']
```

```
>>> a.union(L) # é possível usar os métodos com listas
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 'a'} # e tuplas
```

Contando com esses recursos, os conjuntos da linguagem Python são poderosos e flexíveis, mas apresentam uma limitação. Só podem ser membros de um conjunto os tipos de objetos imutáveis, ou seja, listas e dicionários, por serem mutáveis, não podem fazer parte de conjuntos.

Uso de conjunto como iterador

Os conjuntos podem ser utilizados como iteradores, como ilustrado no Exemplo 6.6. A cada repetição, o objeto x assume um dos elementos de c.



Exemplo 6.6 Uso de conjunto como iterador print("Início do Programa")

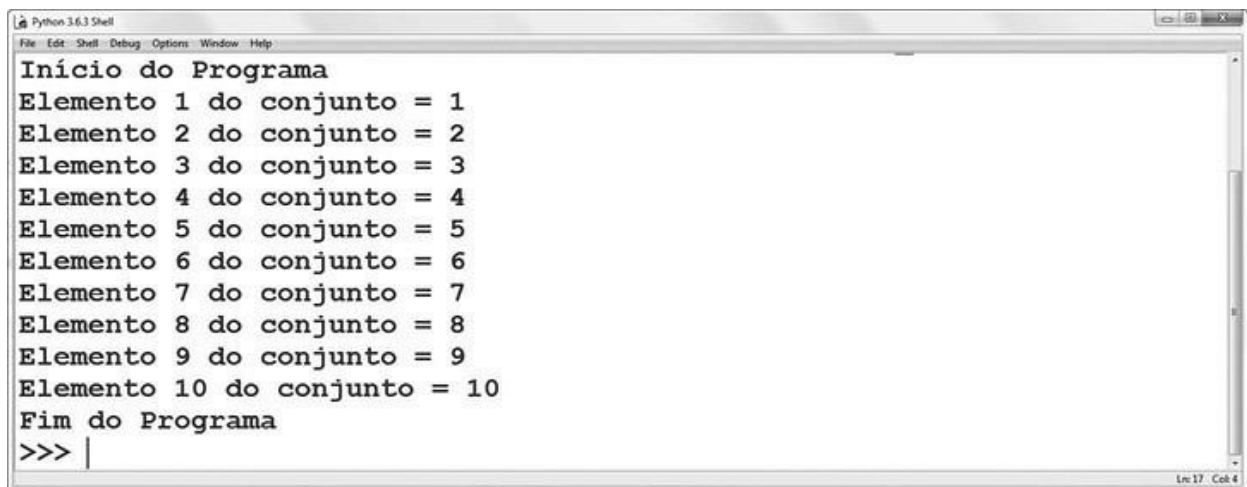
```
cont = 1
```

```
c = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} # c é um set
```

```
for x in c:
```

```
    print("Elemento {} do conjunto = {}".format(cont, x)) cont += 1
```

```
print("Fim do Programa")
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
Elemento 1 do conjunto = 1
Elemento 2 do conjunto = 2
Elemento 3 do conjunto = 3
Elemento 4 do conjunto = 4
Elemento 5 do conjunto = 5
Elemento 6 do conjunto = 6
Elemento 7 do conjunto = 7
Elemento 8 do conjunto = 8
Elemento 9 do conjunto = 9
Elemento 10 do conjunto = 10
Fim do Programa
>>> |
```

Dicionários

Os dicionários são, junto com as listas, os tipos de objetos mais flexíveis em Python. De um lado, as listas são sequenciais e o programador dispõe do índice para ter acesso individualizado a seus elementos. Por sua vez, os dicionários são não sequenciais, porém, permitem que um elemento individual seja acessado por meio de uma chave. Essa chave pode ser qualquer objeto *hashable*. A cada chave deve ser associado um conteúdo que pode ser qualquer objeto Python, imutável ou mutável.

O programador pode dispor dos dicionários da maneira que necessitar. É possível acrescentar e remover elementos, alterar o conteúdo de um elemento, verificar se um elemento está ou não presente, iterar sobre os membros do dicionário etc.

A sintaxe para uso dos dicionários assemelha-se muito à sintaxe utilizada para as listas, com a diferença de que, no lugar do índice, utiliza-se a chave. Diferentes



membros de um dicionário podem ter diferentes tipos de chaves. No Exemplo 6.7 é criado um dicionário `d` contendo dois elementos: o primeiro tem chave numérica 442 e conteúdo string “Elemento 442”; o segundo tem chave numérica 513 e conteúdo string “Elemento 513”. Em seguida, é adicionado um novo membro com chave 377 e conteúdo “Elemento 377”. Para essa adição de novo elemento basta escrever uma linha com a seguinte estrutura: `ObjetoDicionário[chave] = conteúdo`.

Exemplo 6.7 Primeiros usos de dicionário

```
>>> d = {442:"Elemento 442", 513:"Elemento 513"}
```

```
>>> d
```

```
{442: 'Elemento 442', 513: 'Elemento 513'}
```

```
>>> d[442] # exibe o conteúdo cuja chave é 442 'Elemento 442'
```

```
>>> d[513] # exibe o conteúdo cuja chave é 513 'Elemento 513'
```

```
>>> d[377] = "Elemento 377" # adiciona um novo elemento
```

```
>>> d
```

```
{442: 'Elemento 442', 513: 'Elemento 513', 377: 'Elemento 377'}
```

```
>>> d['ab'] = (2, 4, 6)
```

```
>>> d
```

```
{442: 'Elemento 442', 513: 'Elemento 513', 377: 'Elemento 377', 'ab':
```

```
(2, 4, 6)}
```

Por fim, no Exemplo 6.7 propositalmente foi adicionado um quarto membro com uma chave que foge ao padrão que vinha sendo usado, justamente para mostrar que não há padrão necessário. Nesse quarto elemento a chave é o string “ab” e o conteúdo é a tupla (2, 4, 6).



Métodos do tipo dicionário

O tipo dicionário apresenta diversos métodos que estão descritos no Quadro 6.3 e exemplificados em seguida.

Considere-se disponível o dicionário D	
Método	Descrição
D.clear()	Remove todos os elementos do dicionário.
D.copy()	Retorna uma cópia do dicionário.
D.fromkeys(i, v=None)	Recebe o iterável i como parâmetro e retorna um novo dicionário tendo os elementos do iterável como chave. Se o segundo parâmetro (opcional) for fornecido, cada valor será inicializado com ele.
D.get(k [,d])	Retorna o valor associado com a chave k passada como parâmetro. Caso a chave não esteja presente, retorna o segundo parâmetro (opcional) e, na ausência deste, retorna <i>None</i> (na prática não faz nada).
D.items()	Retorna um conjunto contendo os itens do dicionário (o par chave:valor). Esse retorno é do tipo dict_items e se assemelha ao tipo set, podendo ser utilizado como tal.
D.keys()	Retorna um conjunto contendo as chaves do dicionário. Esse retorno é do tipo dict_keys e se assemelha ao tipo set, podendo ser utilizado como tal.
D.pop(k [,d])	Remove a chave k e retorna o valor a ela associado. Caso k não esteja presente, o segundo parâmetro (opcional) é retornado e, na ausência deste, gera a exceção <i>KeyError</i> .
D.popitem()	Remove do dicionário e retorna um par chave:valor arbitrário. Se o dicionário estiver vazio, gera a exceção <i>KeyError</i> .
D.setdefault(k [,d])	Se a chave k estiver presente, seu retorno funciona como um get(k, d). Caso contrário, inclui o par k:d no dicionário, em que d é opcional e, em sua ausência, é utilizado <i>None</i> . Esse método representa uma maneira alternativa de inicializar um dicionário.
D.update(E)	Atualiza o dicionário D a partir dos itens contidos no dicionário E. Se algum item de E não estiver em D, então, será incluído, e caso esteja será atualizado. O parâmetro E também pode ser uma lista ou tupla que contenha seus elementos na forma (e1, e2).
D.values()	Retorna um conjunto contendo os valores do dicionário. Esse retorno é do tipo dict_values e se assemelha ao tipo set, podendo ser utilizado como tal.

Quadro 6.3 Métodos da classe “dict” disponíveis ao programador.



O modo mais elementar de inserir um par chave:valor em um dicionário é criar uma expressão `D[chave] = valor`. Se a chave não existir, ocorrerá a inserção. Essas operações estão feitas no bloco 1 do Exemplo 6.8. No bloco 2 foi utilizado o método `fromkeys` para gerar um dicionário a partir de uma tupla preexistente. Cada elemento da tupla foi utilizado como chave e, se o segundo parâmetro opcional for utilizado, ele será o valor atribuído a todos os membros. No exemplo, `fromkeys` é utilizado duas vezes, sem e com o segundo parâmetro.

Exemplo 6.8 Operações básicas com dicionários # bloco 1

```
>>> D = {} # define o dicionário D vazio
```

```
>>> D['aa'] = 'Valor 1' # adiciona um par chave:valor
```

```
>>> D['ab'] = 'qq.coisa' # adiciona um par chave:valor
```

```
>>> D
```

```
{'aa': 'Valor 1', 'ab': 'qq.coisa'}
```

```
>>> D['ab'] = 'Valor 2' # altera o valor da chave = 'ab'
```

```
>>> D
```

```
{'aa': 'Valor 1', 'ab': 'Valor 2'}
```

```
# bloco 2
```

```
>>> T = (16, 12, 25, 14) # define-se uma tupla T
```

```
>>> A = dict.fromkeys(T) # o método fromkeys pode ser
```

```
>>> A # usado para gerar um dicionário
```

```
{16: None, 12: None, 25: None, 14: None} # sem o segundo parâmetro
```



```
>>> A = dict.fromkeys(T, 0)
```

```
>>> A
```

```
{16: 0, 12: 0, 25: 0, 14: 0} # com o segundo parâmetro
```

```
# bloco 3
```

```
>>> B = {'Nome': 'Pedro', 'Idade': 32, 'Profissão': 'Professor'}
```

```
>>> B
```

```
{'Nome': 'Pedro', 'Idade': 32, 'Profissão': 'Professor'}
```

```
>>> B.keys()
```

```
dict_keys(['Nome', 'Idade', 'Profissão'])
```

```
>>> B.values()
```

```
dict_values(['Pedro', 32, 'Professor']) >>> B.items() dict_items([('Nome',  
'Pedro'), ('Idade', 32), ('Profissão', 'Professor')])
```

```
# bloco 4
```

```
>>> E = {'Local': 'FATEC-SP', 'Cidade': 'São Paulo', 'Idade': 42}
```

```
>>> E
```

```
{'Local': 'FATEC-SP', 'Cidade': 'São Paulo', 'Idade': 42}
```

```
>>> B.update(E)
```

```
>>> B
```




```
{'Nome': 'Pedro', 'Idade': 42, 'Profissão': 'Professor', 'Local': 'FATEC-SP',  
'Cidade': 'São Paulo'}
```

No bloco 3 foi definido um novo dicionário para exemplificar o uso dos métodos `keys`, `values` e `items`, que, respectivamente, retornam um conjunto de chaves, de valores e de tuplas formadas pelo par chave:valor. No bloco 4 foi definido um novo dicionário **E**, e este foi usado para atualizar o dicionário **B**. Note a atualização da `Idade`, que em **E** tem um valor diferente.

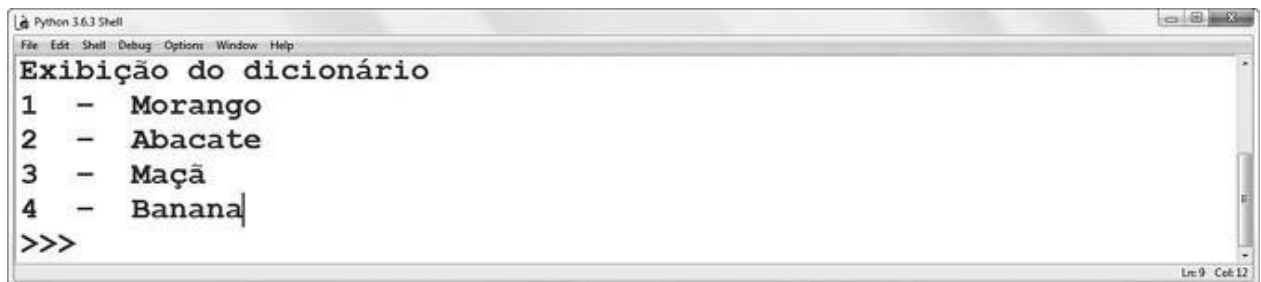
Uso de dicionário como iterador

Os dicionários também podem ser usados como iteradores em um comando `for`, sendo que, no caso dos dicionários, há diversas possibilidades para isso, conforme mostrado a seguir.

Caso 1: a cada repetição, o objeto de controle `x` assume a chave de um item do dicionário. O valor pode ser acessado por meio de `D[x]`. `D = {1:'Morango', 2:'Abacate', 3:'Maçã', 4:'Banana'}` `print("Exibição do dicionário")`

```
for x in D: # iteração é feita com D
```

```
print(x, ' - ', D[x])
```



```
Python 3.6.3 Shell  
File Edit Shell Debug Options Window Help  
Exibição do dicionário  
1 - Morango  
2 - Abacate  
3 - Maçã  
4 - Banana  
>>>
```

Caso 2: as iterações são feitas com o retorno do método `keys`. O resultado é exatamente o mesmo exibido anteriormente.

```
D = {1:'Morango', 2:'Abacate', 3:'Maçã', 4:'Banana'} print("Exibição do  
dicionário")
```

```
for x in D.keys(): # iteração é feita com D.keys() print(x, ' - ', D[x])
```

Os casos 1 e 2, na prática, são um caso só. Quando o método `keys` é omitido o interpretador o assume por padrão.

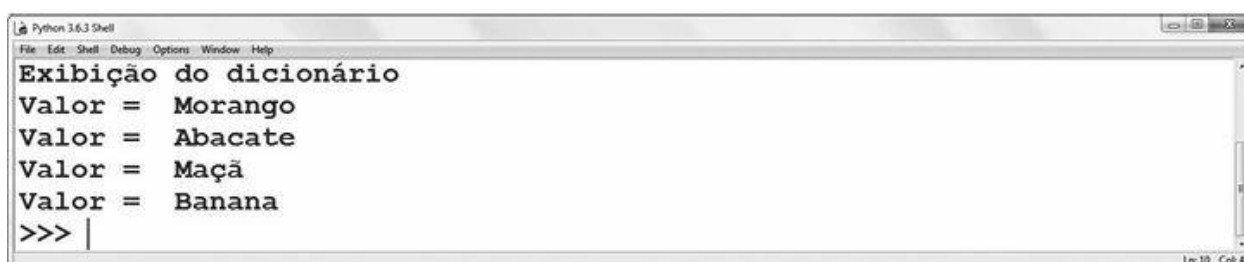


Caso 3: as iterações são feitas com o retorno do método values. Assim, cada repetição o objeto de controle x assume o valor de um item do dicionário. Neste caso, não é possível acessar a chave.

```
D = {1:'Morango', 2:'Abacate', 3:'Maçã', 4:'Banana'} print("Exibição do dicionário")
```

```
for x in D.values():
```

```
    print('Valor = ', x)
```



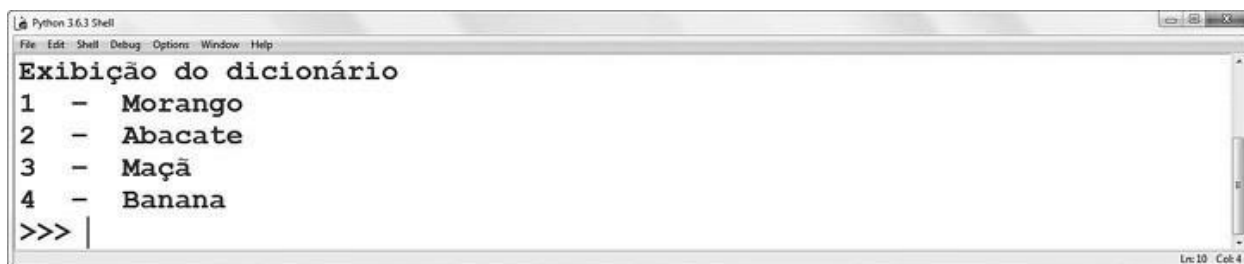
```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Exibição do dicionário
Valor = Morango
Valor = Abacate
Valor = Maçã
Valor = Banana
>>> |
```

Caso 4: as iterações são feitas com o retorno do método items. Como esse método retorna tuplas contendo (chave, valor), então, podem-se utilizar dois objetos para receber cada um dos elementos da tupla. Esse programa utiliza um recurso diferente, porém, o resultado produzido é igual aos casos 1 e 2.

```
D = {1:'Morango', 2:'Abacate', 3:'Maçã', 4:'Banana'} print("Exibição do dicionário")
```

```
for numero, nome in D.items():
```

```
    print(numero, ' - ', nome)
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Exibição do dicionário
1 - Morango
2 - Abacate
3 - Maçã
4 - Banana
>>> |
```

