

Introdução à Programação

Aula 13: Dicionários, Função zip() e Funções para Converter Containers

Prof. Eduardo Corrêa Gonçalves

21/05/2024

Sumário

Introdução

O que é Dicionário?

Criação

Propriedades e Operações

Operações Básicas

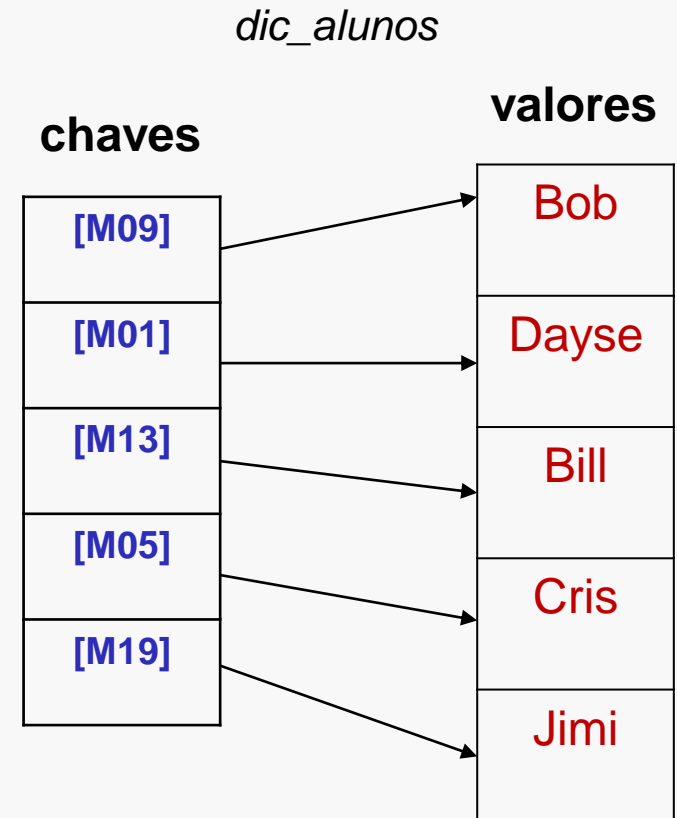
Métodos de Dicionários

Tópicos Extras: Dictionary Comprehension e Função zip()

Dicionário (1/2)

- Dicionário (*Dictionary*) – Definição:

- ED em que elementos são **pares chave:valor**.
 - A chave (*key*) identifica um item e o valor armazena o conteúdo do mesmo.
 - Qualquer valor pode ser recuperado de forma **extremamente rápida** por sua chave
 - A chave é *hasheada*...



- Características:

- As **chaves** devem ser **únicas**, mas podem existir **valores duplicados**.
- **Mutável** – pode ser alterado.
- **Iterável** – capaz de retornar seus elementos um por vez em um laço.
- **Não é sequência** – elemento não pode ser indexado por posição.

Dicionário (2/2)

- **Aplicações Práticas**

- Devido ao seu enorme número de **aplicações práticas**, é considerada uma das mais importantes EDs. Alguns exemplos:
 - Um **sistema acadêmico** utiliza a matrícula do estudante como chave para o seu registro associado (nome, endereço, turmas em que está inscrito, etc.).
 - Na área de redes, o sistema de nomes de domínios (**DNS**), mapeia um **nome de host**, como www.ence.ibge.gov.br para um **endereço IP** como 208.215.179.146.
 - Um **sistema gráfico** de computador mapeia um **nome de cor**, como “**RoyalBlue**”, para uma tripla de números que descrevem a sua representação **RGB**, como (**65, 105, 225**).
 - A notação **JSON** utiliza uma sintaxe baseada em chaves e valores para representar informações e, virtualmente, qualquer tipo de base de dados.
 - A linguagem Python utiliza dicionários para **mapear constantes e valores** (como *math.pi* associado a 3.14159)

Criação de Dicionários (1/4)

- Criando Dicionários

- Devemos especificar, entre chaves, uma relação de elementos do tipo chave:valor separados por vírgula:

- `dic_alunos = {"M09":"Bob",
 "M01":"Dayse",
 "M13":"Bill",
 "M05":"Cris",
 "M19":"Jimi"}`

- `dic_titulos = {'Portela':22,'Mangueira':19,'Beija-Flor':14}`

- `dic_vazio = dict()` # Também pode ser `dic_vazio = { }`

Criação de Dicionários (2/4)

- Criando Dicionários

- Se tentarmos criar um dicionário com uma chave repetida, o Python manterá apenas o último valor:

```
>>> d = {1:"A", 2:"B", 1:"C", 2:"D", 3:"E"}
```

```
>>> d
```

```
{1: 'C', 2: 'D', 3: 'E'}
```

- Com o construtor **dict()**, podemos criar dicionários a partir de tuplas 2d ou listas 2d.

```
>>> dict( [ ('FR', 'Euro'), ('BR', 'Real'), ('AR', 'Peso') ] )
```

```
{'FR': 'Euro', 'BR': 'Real', 'AR': 'Peso'}
```

Criação de Dicionários (3/4)

- Criando Dicionários

- Também é possível criar dicionários aninhados. Neste exemplo, cada chave (f1, f11 e f8) está associada a uma tupla com 5 elementos.

```
>>> filmes= {  
    "f1": ( "O Filho da Nova", 2001, "AR", 123, 7.9),  
    "f11": ( "Orgulho e Preconceito", 2005, "UK", 129, 7.8),  
    "f8": ( "Um Conto Chinês", 2011, "AR", 93, 7.3)  
}
```

- Chaves devem ser únicas, mas valores podem ser duplicados.

```
>>> food = {"bacon" : "yes", "egg" : "yes", "spam" : "no" }
```

Criação de Dicionários (4/4)

- Criando Dicionários
 - Quando as **chaves são strings**, podemos utilizar o construtor **dict()** usando a sintaxe **nome=valor** para os pares.

```
>>> dict(FR='Euro', BR='Real', AR='Peso')  
{'FR': 'Euro', 'BR': 'Real', 'AR': 'Peso'}
```


Propriedades e Operações (1/11)

- Operações básicas:
 - Recuperar o valor de uma chave
 - Adicionar **entrada** (*par chave:valor*)
 - Modificar o valor de uma chave.
 - Pesquisar se chave existe
 - Remover **entrada** (*par chave:valor*)
 - Recuperar todas as chaves ou todos os valores.
 - Iterar pelas chaves, valores ou pares chave:valor
- **Obs1: Não temos indexação e nem fatiamento**, pois os elementos dos dicionários são indexados por chaves e não por posição.
- **Obs2:** O termo **entrada** (*entry*) também é utilizado para designar um **par chave:valor**.
 - Esse termo é originário dos dicionários tradicionais.
<https://www.macmillandictionary.com/learn/dictionary-entry.html>

Propriedades e Operações (2/11)

- Recuperando o valor de uma chave

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
dic["mae"]          # 'Yoko'
```

```
dic['filho']         #'Sean'
```

```
>>> dic["primo"]    # Ocorre erro se especifico chave inexistente.
```

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

```
File "<stdin>", line 1, in <module>
```

```
KeyError: 'primo'
```

- Adicionando uma nova entrada (elemento chave:valor)

```
dic["enteado"] = "Julian"  #{'mae': 'Yoko', 'pai': 'John', 'filho': 'Sean', 'enteado': 'Julian'}
```

- Modificando o valor de uma chave

```
dic["pai"] = "John Lennon"  #{'mae': 'Yoko', 'pai': 'John Lennon', 'filho': 'Sean',  
                             'enteado': 'Julian'}
```

Propriedades e Operações (3/11)

- Checando se uma chave existe

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
"mae" in dic           # True      (in = a chave pertence ao dicionário?)
```

```
"primo" in dic         # False
```

```
"primo" not in dic     # True      (not in = a chave não pertence ao dicionário?)
```

- Obtendo as propriedades de um dicionário

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
# tipo do objeto dicionário
```

```
type(dic)              # <class 'dict'>
```

```
# número de elementos do dicionário
```

```
len(dic)               # 3
```

Propriedades e Operações (4/11)

• Métodos disponíveis para Dicionários

- **dic.get(*k*)**: retorna o valor associado à chave *k* do dicionário. Se a chave não existir, não retorna erro.
- **dic.pop(*k*)**: remove o par referente à chave *k* e retorna o valor que foi removido.
- **dic.popitem()**: remove o último par inserido e o retorna em uma tupla (chave, valor)
- **dic.setdefault(*k*, *v*)**: retorna o valor associado à chave *k*. Caso essa chave não exista no dicionário, o método vai inserir a chave *k* associada ao valor *v*.
- **dic.clear()**: esvazia o dicionário
- **dic.copy()**: faz uma shallow copy do dicionário. Para uma deep copy, é preciso usar o módulo copy.
- **dic.update(*iter*)**: insere todos os elementos do iterável *iter* em *dic*. Caso alguma chave já exista, o valor será sobrescrito.
- **dic.fromkeys(*lst_keys*, *v*)**: cria um dicionário contendo todas as chaves especificadas em *lst_keys*. Se *v* for especificados, todas estarão associadas ao valor *v*. Caso contrário, todas estarão associadas ao valor None.
- **dic.items()**: retorna uma visão contendo os pares chave-valor do dicionário como uma lista de tuplas no formato [(*k1*, *v1*), (*k2*, *v2*), ..., (*kn*, *vn*)]
- **dic.keys()**: retorna uma visão contendo as chaves do dicionário como uma lista.
- **dic.values()**: retorna uma visão contendo os valores do dicionário como uma lista

Propriedades e Operações (5/11)

- Inserindo e removendo entradas

```
dic1 = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
dic2 = {"bateria": "Ringo", "baixo": "Paul", "guitarra1": "George"}
```

```
dic1.pop("mae")      # remove "mae": {'pai': 'John', 'filho': 'Sean'}
```

```
del dic1["filho"]    # com del posso também apagar 1 entrada : {'pai': 'John'}
```

```
dic1.update(dic2)    # insere os elementos de dic2 em dic1:
                    # {'pai': 'John', 'bateria': 'Ringo', 'baixo': 'Paul', 'guitarra1': 'George'}
```

```
dic1["guitarra2"] = "John" # insere nova entrada: {'pai': 'John', 'bateria': 'Ringo',
                    # 'baixo': 'Paul', 'guitarra1': 'George', 'guitarra2': 'John'}
```

```
dic1.pop("pai")      # remove a entrada "pai": {'bateria': 'Ringo', 'baixo': 'Paul',
                    # 'guitarra1': 'George', 'guitarra2': 'John'}
```

```
dic1.clear()         # esvazia o dicionário
```

Propriedades e Operações (6/11)

- Recuperando o valor de uma chave (revisitado...)

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
>>> dic["primo"]    # Usando colchetes, ocorre erro se especifico chave inexistente.
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'primo'
```

```
>>> dic.get("primo") # Porém com o método get(), não ocorre erro.
```

```
>>> # Ele apenas retorna None...
```

Propriedades e Operações (7/11)

- Checando se uma chave existe (já havíamos mostrado)

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
"mae" in dic          # True    (in = a chave pertence ao dicionário?)
```

```
"sogra" in dic        # False
```

- Checando se um valor existe (não havíamos mostrado...)

```
>>> "Yoko" in dic.values()
```

```
True
```

```
>>> "Yoki" in dic.values()
```

```
False
```

Propriedades e Operações (8/11)

- Iterando apenas sobre as chaves

- Para iterar sobre as chaves, há duas formas possíveis:

1. Não usar método nenhum
2. Usar o método **keys()**, que retorna uma visão contendo a lista de chaves.

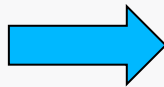
```
dic = {"mae":"Yoko", "pai":"John", "filho":"Sean"}
```

```
for chave in dic:  
    print(chave)
```



```
mae  
pai  
filho
```

```
for chave in dic.keys():  
    print(chave)
```



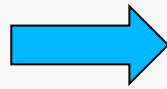
```
mae  
pai  
filho
```


Propriedades e Operações (9/11)

- Iterando apenas sobre os valores
 - Para iterar sobre os valores, usa-se o método **values()**, que retorna uma visão contendo a lista de valores.

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
for valor in dic.values():  
    print(valor)
```



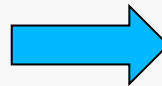
Yoko
John
Sean

Propriedades e Operações (10/11)

- Iterando sobre chaves e valores ao mesmo tempo
 - Mais interessante, as chaves e seus respectivos valores podem ser recuperados ao mesmo tempo usando o método **items()**
 - Com a técnica de **desempacotamento**, colocamos a chave e o valor em duas variáveis distintas.

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
for chave, valor in dic.items():
    print(chave, valor)
```



```
mae Yoko
pai John
filho Sean
```

- **Obs.:** Até o Python 3.5, resultados poderiam vir em qualquer ordem. A partir do Python 3.6, eles vêm na ordem em que foram inseridos.
 - *Detalhes na parte final da aula*

Propriedades e Operações (11/11)

- **Transformações**

- É possível gerar listas, tuplas e conjuntos a partir de dicionários. Basta usar os construtores dessas EDs.

```
dic = {"mae": "Yoko", "pai": "John", "filho": "Sean"}
```

```
>>> list(dic.values())      # gera lista com os valores do dicionário
```

```
['Yoko', 'John', 'Sean']
```

```
>>> set(dic)               # gera conjunto com as chaves do dicionário
```

```
{'mae', 'filho', 'pai'}
```

```
>>> tuple(dic.items())     # gera tupla 2d com os pares chave:valor
```

```
(('mae', 'Yoko'), ('pai', 'John'), ('filho', 'Sean'))
```

#**sorted()** processa qualquer iterável, gerando uma **lista ordenada** como saída

```
>>> sorted(dic.values())
```

```
['John', 'Sean', 'Yoko']
```

Tópicos Extras (1/4)

- **Dict Comprehension**

- Similar ao list comprehension, porém para criar dicionários.
 - A definição deverá estar entre chaves.
 - É preciso especificar quem será a chave e quem será o valor.

Exemplo 1 – básico:

```
>>> { x: x**2 for x in (2, 4, 6) }  
{2: 4, 4: 16, 6: 36}
```

Exemplo 2 – operação matemática:

```
>>> d = {'leite': 4.10, 'café': 8.99, 'pão': 3.50}  
>>> reajuste = 1.1  
>>> d = {produto: round(preco * reajuste,2) for (produto, preco) in d.items()}  
  
>>> d  
{'leite': 4.51, 'café': 9.89, 'pão': 3.85}
```

Tópicos Extras (2/4)

- Dict Comprehension

Exemplo 3 –filtragem:

```
>>> original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
>>> {k: v for (k, v) in original_dict.items() if v > 40}
{'michael': 48, 'guido': 57}
```

Exemplo 4 – transformação:

```
>>> original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
>>> {k: ('senior' if v > 40 else 'junior') for (k, v) in original_dict.items()}
{'jack': 'junior', 'michael': 'senior', 'guido': 'senior', 'john': 'junior'}
```

Conceitos (3/5)

- Conceitos Importantes do Python

- **Iterável (*Iterable*)**: é o tipo de container que possibilita iterar por seus elementos.
 - Iterar = percorrer um por um, em um laço.

```
for cidade in ('Rio','Recife','Goiania','Niteroi'):  
    print(cidade)
```

Rio

Recife

Goiania

Niteroi

- Caso o container seja uma sequência (ex: tupla e lista), a iteração pode ser feita em uma ordem definida pelo programador (ex: do menor índice para o maior). Caso contrário (ex: conjunto), não há como definir a ordem.
- Nem toda ED é iterável. Por exemplo, a **pilha não é iterável** (só dá para acessar o elemento do topo).

zip() – (1/11)

- Definição

- Função que recebe como entrada um ou mais iteráveis e retorna um **iterador** (*iterator*).
 - O iterator retornado pela função zip() é capaz de gerar uma sequência de **tuplas** combinando elementos dos iteráveis de entrada.

- Sintaxe: **zip(*iterables)**

- zip() é uma função padrão (*built-in*) do Python. Você pode utilizá-la sem precisar importar nenhum módulo.
- Aceita qualquer tipo de iterável, como arquivos, listas, tuplas, dicionários, conjuntos e outros.

zip() – (2/11)

- Mas o que é um iterator?
 - Um iterator é um objeto preparado para retornar uma sequência de valores.
 - Ele **não é** a sequência de valores... ele é um objeto que está preparado e irá gerar a sequência de valores, caso você queira.
 - Um iterator possui **duas propriedades** principais:
 1. Gera elementos **sob demanda**;
 2. Só pode ser **percorrido** uma **única vez**. Ao gerar o último elemento da sequência, o iterator nada mais pode fazer.

zip() – (3/11)

- Utilização básica

```
>>> nums = [1,2,3]
```

```
>>> letras = ["a", "b", "c"]
```

```
# cria um iterator, considerando a ordem:
```

```
# [(nums[0], letras[0]), (nums[1], letras[1]), (nums[2], letras[2])]
```

```
>>> zipado = zip(nums, letras)
```

```
>>>
```

```
>>> zipado
```

```
<zip object at 0x000002859B18A900> # a função zip() cria um iterator:
```

```
>>> # objeto “preparado” para virar um iterável
```

```
>>> list(zipado)
```

```
[(1, 'a'), (2, 'b'), (3, 'c')] # com list(), podemos “consumir” o iterator
```

```
# produzindo uma lista.
```

zip() – (4/11)

- Utilização básica

exemplo passando 3 iteráveis como entrada

```
>>> inteiros = [1,2,3]
```

```
>>> letras = ('a','b','c')
```

```
>>> reais = [1.5, 2.5, 3.5]
```

```
>>> z = zip(inteiros, letras, reais)
```

```
>>>
```

```
>>> list(z)
```

```
[(1, 'a', 1.5), (2, 'b', 2.5), (3, 'c', 3.5)]
```

zip() – (5/11)

- **Casando Elementos de Tamanhos Diferentes**

- Se iteráveis de tamanhos diferentes são passados como entrada, o resultado será limitado ao que possuir menos elementos.

```
>>> list(zip(range(5), range(100)))
```

```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)] # 95 elementos do segundo range
# não foram casados
```

- Detalhes sobre como contornar essa situação podem ser obtidos em <https://realpython.com/python-zip-function>

- **Criando um iterator vazio**

```
>>> z = zip() # cria um iterator vazio
```

```
>>> z
```

```
<zip object at 0x000002859B182640>
```

zip() – (6/11)

- **Avaliação Lazy**

- No Python 3, o iterator produz as tuplas sob demanda (abordagem *lazy*) e só pode ser percorrido uma vez.
- É possível gerar elemento por elemento com a função **next()**.
 - Com isso, em vez jogar toda uma nova lista em memória, ele joga um elemento de cada vez. A cada `next()`, o elemento gerado no passo anterior pode ser descartado.
 - Ao dar `next()` após o último elemento da sequência ter sido gerado, ocorre uma exceção do tipo *StopIteration*.
 - *Aprenderemos a tratar exceções em aula futura deste curso.*
- De maneira oposta, quando usamos funções como **list()**, o iterator é inteiramente percorrido, gerando toda a sequência de elementos em memória.

zip() – (7/11)

- Avaliação Lazy- exemplo next()

```
>>> z=zip(range(3), ('A','E','I'))
```

```
>>> next(z)
```

```
(0, 'A')
```

```
>>> next(z)
```

```
(1, 'E')
```

```
>>> next(z)
```

```
(2, 'I')
```

```
>>> next(z)
```

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

```
  File "<pyshell#44>", line 1, in <module>
```

```
    next(z)
```

```
StopIteration
```

zip() – (8/11)

- **Loop sobre Múltiplos Objetos**
 - É uma das principais aplicações da função zip()

```
>>> nomes=['Elizabeth','Darcy','Bingley','Jane']
```

```
>>> salarios=[5000,6500,3200,1700]
```

```
>>> for n,s in zip(nomes, salarios):
    print("{} ganha ${}".format(n,s))
```

Elizabeth ganha \$5000

Darcy ganha \$6500

Bingley ganha \$3200

Jane ganha \$1700

zip() – (9/11)

- **Desempacotando uma Sequência**

- Para desempacotar uma sequência de tuplas, usamos zip() em conjunto com asterisco * (operador de desempacotamento)

```
>>> pares=[(0, 'A'), (1, 'E'), (2, 'I'), (3, 'O'), (4, 'U')]
```

```
>>> numeros, letras = zip(*pares)
```

```
>>> numeros
```

```
(0, 1, 2, 3, 4)
```

```
>>> letras
```

```
('A', 'E', 'I', 'O', 'U')
```

zip() – (10/11)

- **Ordenação**

- Ao combinar duas listas com o `zip()`, você pode utilizar o método `sort()` para ordená-la por critérios distintos.
 - A ordenação se dará considerando a lista que for especificada como primeiro parâmetro.

```
>>> letras = ['b', 'a', 'd', 'c']
>>> numeros = [2, 4, 3, 1]
>>> lst = list(zip(letras, numeros))
>>> lst
[('b', 2), ('a', 4), ('d', 3), ('c', 1)]
>>>
>>> lst.sort()  # ordena por letras
>>> lst
[('a', 4), ('b', 2), ('c', 1), ('d', 3)]
```


zip() – (11/11)

- **Recuperando Colunas Inteiras de uma Lista 2d**

- A função zip() nos permite pegar uma “coluna” inteira em uma lista 2d
 - Veja o exemplo abaixo:

```
>>> m = [[1, 0, 7],  
          [4, 5, 5],  
          [3, 10, 2]]
```

```
>>> colunas = [c for c in zip(*m)]    # zip(*m) é o mesmo que zip(m[0], m[1], m[2])
```

```
>>> colunas
```

```
[(1, 4, 3), (0, 5, 10), (7, 5, 2)]
```

obtendo a soma de todas as colunas

```
>>> [sum(c) for c in zip(*m)]
```

```
[8, 15, 14]
```

Tópicos Extras (3/4)

- **zip()**
 - Utilizando a função **zip()**, podemos criar um dicionário a partir de duas listas:
 - Na primeira passamos as chaves.
 - Na segunda os valores

```
>>> d = dict(zip([1, 2, 3], ["A", "B", "C"]))
```

```
>>>
```

```
>>> d
```

```
{1: 'A', 2: 'B', 3: 'C'}
```