Dicionários

Você provavelmente já viu diversas tabelas onde cada coluna representa uma informação diferente sobre uma pessoa, evento ou objeto. Por exemplo, cada linha pode representar um aluno. A primeira coluna pode ser o nome do aluno, as quatro colunas seguintes podem ser suas notas em provas, a sexta coluna pode ser sua média, a sétima pode ser suas presenças, a oitava pode ser seu status (aprovado ou reprovado) e a nona pode conter algumas observações.

Em programação, as linhas de uma tabela poderiam ser representadas por listas. Neste caso, o índice 0 de cada linha seria sempre o nome, os índices 1 a 4 seriam as notas, o índice 5 seria a média, e assim sucessivamente. Porém, isso nem sempre convém, porque:

- expressões como `lista[1] + lista[2] + lista[3] + lista[4]` nos trazem pouca informação sobre o que significam os dados envolvidos na operação;
- caso seja necessário alterar a estrutura de dados para inserir, por exemplo, mais notas, seria necessário adaptar todo o código alterando todos os *números mágicos* representando cada coluna;

Seria preferível referenciarmos o nome do aluno por 'nome', suas notas poderiam ser uma lista chamada 'notas', sua média poderia ser chamada 'media', e assim sucessivamente. Aqui entram os **dicionários**, também conhecidos em outras linguagens como **tabelas hash**, **hash maps**, entre outros.

1. Dicionários em Python

Quando utilizamos um dicionário (o de papel, com definições), não temos o hábito de procurar pela palavra que está em uma determinada posição. Ao invés disso, nós buscamos pela palavra em si, e ao encontrá-la ela contém uma definição.

A estrutura **dicionário** em Python é uma coleção de dados. Porém, ela não é indexada. Ao adicionarmos elementos em um dicionário, sempre o fazemos aos pares: todo elemento terá uma **chave** e um **valor**.

A chave será uma *string* que utilizaremos como se fosse o índice. É como se fosse a palavra que buscamos em um dicionário de papel.

O valor pode ser qualquer dado: um `int`, um `float`, uma `str`, um `bool`, uma lista, uma tupla, outro dicionário. Ele é como se fosse a definição que encontramos vinculada à palavra que buscamos no dicionário de papel.

1.1. Criando um dicionário

Separamos chave e valor utilizando dois pontos (`:`), e separamos um par de outro utilizando vírgula. Utilizamos o símbolo chave (`{` e `}`) para representar um dicionário. O exemplo abaixo representa um aluno como descrito no início do capítulo.

```
aluno = {'nome':'Mario', 'notas':[7, 9, 5, 6], 'presencas':0.8}
```

Podemos acessar uma informação de um dicionário utilizando a sua chave da mesma maneira que utilizamos um índice em uma lista:

```
print('Aluno:', aluno['nome']) # Aluno: Mario
print('Notas:', aluno['notas']) # Notas: [7, 9, 5, 6]
```

Também é possível criar dicionários através da função `dict`. Ela pode ser utilizada de diferentes maneiras. Uma delas é passando parâmetros com nomes. Os nomes dos parâmetros se tornarão chaves, e os valores associados serão valores:

```
notas = dict(Ana = 7, Brenda = 10, Carlos = 8)
print(notas) # resultado: {'Ana': 7, 'Brenda': 10, 'Carlos': 8}
```

Outra possibilidade é utilizar uma coleção (como uma lista ou uma tupla) contendo, internamente, outras coleções com exatamente 2 elementos. O primeiro elemento será chave, o segundo será valor. O exemplo abaixo cria o mesmo dicionário do exemplo anterior:

```
lista = [['Ana', 7], ['Brenda', 10], ['Carlos', 8]]
dicionario = dict(lista)
print(dicionario)
```

Caso você tenha suas chaves e valores em coleções separadas, uma maneira fácil de explorar a possibilidade anterior é utilizar um `zip`:

```
nomes = ['Ana', 'Brenda', 'Carlos']
notas = [7, 10, 8]
dicionario_notas = dict(zip(nomes, notas))
```

1.2. Adicionando elementos em um dicionário

Para adicionar elementos, não precisamos de uma função pronta (como o *append* das listas). Basta "acessar" a nova chave e atribuir um novo valor.

```
aluno['media'] = sum(aluno['notas'])/len(aluno['notas'])

aluno['aprovado'] = aluno['media'] >= 6.0 and aluno['presencas'] >= 0.7

print(aluno)
```

Saída na tela:

```
{'nome': 'Mario', 'notas': [7, 9, 5, 6], 'presencas': 0.8, 'media': 6.75, 'aprovado': True}
```

1.3. Percorrendo um dicionário

Dicionários podem ser percorridos com um *for*. Ao fazer isso, as **chaves** serão percorridas, não os valores. Porém, a partir da chave obtém-se o valor:

```
for chave in aluno:
    print(chave, '--->', aluno[chave])
```

Saída na tela:

```
nome ---> Mario
notas ---> [7, 9, 5, 6]
presencas ---> 0.8
media ---> 6.75
aprovado ---> True
```

1.4. Testando a existência de uma chave

Antes de criar uma chave nova em um dicionário, convém testar se ela já existe, para evitar sobrescrever um valor. Podemos fazer isso com o operador *in* (sim, o mesmo que usamos no *for*!). Neste contexto, ele retornará `True` se a chave existir e `False` caso contrário.

Vamos supor que, no exemplo abaixo, o valor de 'cursos' seja uma lista com todos os cursos que o usuário está fazendo.

```
dicionario = {'escola':"Ada", 'unidade':'Faria Lima'}

# Neste caso, 'cursos' ainda não existe.
# Cairemos no else e será criada uma lista com a string 'Python'.

if 'cursos' in dicionario:
```

```
    dicionario['cursos'].append('Python')
else:
    dicionario['cursos'] = ['Python']

# Agora a chave já existe.
# Portanto, será adicionado 'Data Science' à lista.
if 'cursos' in dicionario:
    dicionario['cursos'].append('Data Science')
else:
    dicionario['cursos'] = ['Data Science']

print(dicionario)
```

Saída na tela:

```
{'escola': "Ada", 'unidade': 'Faria Lima', 'cursos': ['Python', 'Data Science']}
```

2. Métodos de dicionários

Você deve ter notado que algumas coisas que fazíamos em listas utilizando métodos (funções), em dicionários fazemos de maneira mais direta. Mas dicionários também possuem seus próprios métodos. Iremos estudar alguns bastante utilizados, mas você pode acessar essa página caso queira conhecer mais.

2.1. Acessando valores de maneira segura

Quando tentamos acessar uma chave que não existe em um dicionário ocorre um erro. Vimos que uma forma de driblar isso é testar a existência dela utilizando o `in`. Alguns métodos nos ajudam a "economizar" este teste.

2.1.1. get

O método `get` permite acessar uma chave sem a ocorrência de erro. Caso uma chave não exista, ele irá retornar `None`, a constante nula denotando a ausência de valor.

```
nomes = ['Ana', 'Brenda', 'Carlos']
notas = [7, 10, 8]
dicionario_notas = dict(zip(nomes, notas))

nota_daniel = dicionario_notas.get('Daniel') # valor de nota_daniel: None
nota_eliza = dicionario_notas['Eliza'] # erro de chave inexistente
```

O `get` aceita como parâmetro opcional um valor padrão que será retornado ao invés de `None` caso a chave não exista:

```
nota_brenda = dicionario_notas.get('Brenda', 0) # valor de nota_brenda: 10
nota_daniel = dicionario_notas.get('Daniel', 0) # valor de nota_daniel: None
```

2.1.2. setdefault

Um caso específico que vimos foi quando desejamos inserir uma chave caso ela não exista ou acessar seu valor caso ela exista. O método `setdefault` faz exatamente isso. Passamos uma chave e um valor. Se a chave for encontrada, seu valor é retornado. Caso contrário, ela é inserida com o valor passado. Vamos refazer o exemplo do `in` utilizando este método:

```
dicionario = {'escola':"Ada", 'unidade':'Faria Lima'}
cursos = dicionario.setdefault('cursos', ['Python'])
print(cursos) # resultado na tela: ['Python']
cursos.append('Data Science')
print(dicionario['cursos']) # resultado na tela: ['Python', 'Data Science']
```

2.2. Copiando dicionários

2.2.1. Criando um novo dicionário

Quando você já possui um dicionário e gostaria de copiar todo o seu conteúdo para outro dicionário, assim como no caso da lista, você não deve fazer uma atribuição direta, pois não houve cópia, e sim duas variáveis referenciando o mesmo dicionário na memória:

```
nomes = ['Ana', 'Brenda', 'Carlos']
notas = [7, 10, 8]
dicionario_notas = dict(zip(nomes, notas))
dicionario_notas_copia = dicionario_notas

dicionario_notas_copia['Ana'] = 0
print(dicionario_notas['Ana']) # resultado: 0
```

Para copiar de fato o dicionário você pode utilizar o método `copy`:

```
nomes = ['Ana', 'Brenda', 'Carlos']
notas = [7, 10, 8]
dicionario_notas = dict(zip(nomes, notas))
dicionario_notas_copia = dicionario_notas.copy()

dicionario_notas_copia['Ana'] = 0
print(dicionario_notas['Ana']) # resultado: 7
print(dicionario_notas_copia['Ana']) # resultado: 0
```

2.2.2. Copiar um dicionário para dicionário já existente

Imagine que você já possui dois dicionários distintos e gostaria de uni-los, copiando os pares chave-valor de um deles para o outro. Você pode fazer isso utilizando o método `update`.

```
escola = {'escola':"Ada", 'unidade':'Faria Lima'}
mais_escola = {'trilhas':['Data Science', 'Web Full Stack'], 'formato':'online'}

escola.update(mais_escola)

print(escola)
```

Saída na tela:

```
{'escola': "Ada", 'unidade': 'Faria Lima', 'trilhas': ['Data Science', 'Web Full Stack'], 'formato': 'online'}
```

2.3. Removendo elementos de um dicionário

Você pode remover um elemento de um dicionário através do método `pop`. Você deve passar a chave a ser removida.

```
aluno = {'nome':'Mario', 'notas':[7, 9, 5, 6], 'presencas':0.8}
aluno.pop('presencas')
print(aluno) # resultado: {'nome': 'Mario', 'notas': [7, 9, 5, 6]}
```

2.4. Separando chaves e valores

O Python possui funções para obter, separadamente, todas as chaves ou todos os valores de um dicionário. Elas são, respectivamente, *keys* e *values*. Podemos transformar o retorno dessa função em uma lista ou tupla.

```
aluno = {'nome':'Mario', 'notas':[7, 9, 5, 6], 'presencas':0.8}

chaves = list(aluno.keys())
valores = list(aluno.values())

print('Chaves: ', chaves)
print('Valores:', valores)
```

Saída na tela:

```
Chaves:  ['nome', 'notas', 'presencas']
Valores: ['Mario', [7, 9, 5, 6], 0.8]
```

Combinando essas funções com o `zip` estudado em um capítulo anterior, podemos iterar chaves e valores simultaneamente de maneira bastante *pythonica*:

```
for chave, valor in zip(aluno.keys(), aluno.values()):
    # alguma operação aqui
```

Note que a construção acima também pode ser substituída por um método. O método `items` retorna uma coleção de tuplas, onde cada tupla contém um par chave-valor do dicionário:

```
print(aluno.items())
```

Saída na tela:

```
dict_items([('nome', 'Mario'), ('notas', [7, 9, 5, 6]), ('presencas', 0.8)])
```

Portanto, podemos fazer:

```
for chave, valor in aluno.items():
    # alguma operação aqui
```

Após o capítulo sobre tuplas, onde foram abordadas questões de desempenho, você pode estar se perguntando quão eficiente ou ineficiente é um dicionário. Surpreendentemente, ele é uma estrutura bastante rápida para consulta. Isso se deve à forma como ele é implementado.

O motivo para um de seus nomes ser *tabela hash* é que ele utiliza o conceito de *hash*. De maneira simplificada, *hash* é um valor numérico obtido a partir de um dado quando realizamos uma sequência de operações sobre esse dado. Essas operações devem ser tais que se o dado mudar, o valor numérico também deve mudar. Quando dois dados diferentes podem gerar o mesmo número, chamamos isso de *colisão de hash*, e isso é bastante indesejável.

Quando o dicionário é criado, uma faixa tamanho razoável de memória é alocada para ele. Quando passamos uma chave, o computador calcula o *hash* dessa chave e utiliza o *hash* como índice para acessar o dado na memória! . Inclusive, quando citamos acima que uma chave deve ser uma string, isso foi uma simplificação. Qualquer tipo de dado imutável (como uma tupla ou uma constante numérica) pode servir como chave. Objetos personalizados também podem, desde que eles sejam *hashable*. . Você pode consultar um tutorial mais básico sobre dicionários aqui, focado em diferentes funções úteis. Já aqui temos um tutorial mais avançado, que ensina você a criar uma estrutura semelhante a um dicionário do zero, explicando os conceitos envolvidos em cada passo.