CAPÍTULO 11

Dicionários

Dicionário é uma coleção de itens (chamados *chaves*) e seus respectivos significados (chamados de *valores*):

{chave: valor}

Cada chave do dicionário deve ser única! Ao contrário de *listas*, *dicionários*, não podem ter chaves repetidas.

Nota: As chaves devem ser únicas.

11.1 Declaração

Declaramos um dicionário colocando entre colchetes `{}` cada chave e o seu respectivo valor, da seguinte forma:

```
>>> telefones = {"ana": 123456, "yudi": 40028922, "julia": 4124492}
>>> telefones
{'ana': 123456, 'yudi': 40028922, 'julia': 4124492}
```

No caso acima, a chave `"ana"`, por exemplo, está relacionada ao valor `123456`. Cada par chave-valor é separado por uma vírgula `,` .

11.2 Função `dict()`

A função `dict()` constrói um dicionário. Existem algumas formas de usá-la:

- Com uma lista de listas:

```
>>> lista1 = ["brigadeiro", "leite condesado, achocolatado"]
>>> lista2 = ["omelete", "ovos, azeite, condimentos a gosto"]
>>> lista3 = ["ovo frito", "ovo, óleo, condimentos a gosto"]
>>> lista_receitas = [lista1, lista2, lista3]
```

(continues on next page)

(continued from previous page)

```
>>> print(lista_receitas)
[['brigadeiro', 'leite condesado, achocolatado'], ['omelete', 'ovos, azeite,␣
↪condimentos a gosto'], ['ovo frito', 'ovo, óleo, condimentos a gosto']]
>>> receitas = dict(lista_receitas)
>>> print(receitas)
{'brigadeiro': 'leite condesado, achocolatado', 'omelete': 'ovos, azeite,␣
↪condimentos a gosto', 'ovo frito': 'ovo, óleo, condimentos a gosto'}
```

- Atribuindo os valores diretamente:

```
>>> constantes = dict(pi=3.14, e=2.7, alpha=1/137)
>>> print(constantes)
{'pi': 3.14, 'e': 2.7, 'alpha': 0.0072992700729927005}
```

  Neste caso, o nome das chaves deve ser um identificador válido. As mesmas regras de *nomes de variáveis* (Página 41) se aplicam.

- Usando as chaves `{}`:

```
>>> numerinhos = dict({"um": 1, "dois": 2, "três": 3})
>>> print(numerinhos)
{'um': 1, 'dois': 2, 'três': 3}
```

  E nesse caso se não houvesse a função *dict()*, o resultado seria exatamente o mesmo...

11.3 Chaves

Acessamos um determinado valor do dicionário através de sua chave:

```
>>> capitais = {"SP": "São Paulo", "AC": "Rio Branco", "TO": "Palmas", "RJ": "Rio de␣
↪Janeiro", "SE": "Aracaju", "MG": "Belo Horizonte"}
>>> capitais["MG"]
'Belo Horizonte'
```

Até o momento, usamos apenas *strings*, mas podemos colocar todo tipo de coisa dentro dos dicionários, incluindo listas e até mesmo outros dicionários:

```
>>> numeros = {"primos": [2, 3, 5], "pares": [0, 2, 4], "ímpares": [1, 3, 5]}
>>> numeros["ímpares"]
[1, 3, 5]
```

Mesmo que os pares chave-valor estejam organizados na ordem que foram colocados, não podemos acessá-los por *índices* como faríamos em listas:

```
>>> numeros[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

O mesmo erro ocorre se tentarmos colocar uma chave que não pertence ao dicionário:

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

Assim como os valores não precisam ser do tipo *string*, o mesmo vale para as chaves:

```
>>> numeros_por_extenso = {2: "dois", 1: "um", 3: "três", 0: "zero"}
>>> numeros_por_extenso[0]
'zero'
>>> numeros_por_extenso[2]
'dois'
```

Nota: Listas e outros dicionários *não* podem ser usados como chaves por serem de tipos *mutáveis*.

11.4 Adicionando e removendo elementos

Podemos alterar o valor relacionado a uma chave da seguinte forma:

```
>>> pessoa = {"nome": "Cleiton", "idade": 34, "família": {"mãe": "Maria", "pai": "Enzo
↪"}}
>>> pessoa["idade"]
34
>>> pessoa["idade"] = 35
>>> pessoa["idade"]
35
```

Para adicionar um elemento novo à um dicionário, podemos simplesmente fazer o seguinte:

```
>>> meses = {1: "Janeiro", 2: "Fevereiro", 3: "Março"}
>>> meses[4] = "Abril"
>>> meses
{1: "Janeiro", 2: "Fevereiro", 3: "Março", 4: "Abril"}
```

Aqui nos referimos a uma chave que *não* está no dicionário e associamos um valor a ela. Desta forma, *adicionando* esse conjunto chave-valor ao dicionário.

Removemos um conjunto chave-elemento de um dicionário com o comando `del`:

```
>>> meses
{1: "Janeiro", 2: "Fevereiro", 3: "Março", 4: "Abril"}
>>> del(meses[4])
>>> meses
{1: "Janeiro", 2: "Fevereiro", 3: "Março"}
```

Para apagar *todos* os elementos de um dicionário, usamos o método `clear`:

```
>>> lixo = {"plástico": ["garrafa", "copinho", "canudo"], "papel": ["folha amassada",
↪"guardanapo"], "orgânico": ["batata", "resto do bandeco", "casca de banana"]}
>>> lixo
{"plástico": ["garrafa", "copinho", "canudo"], "papel": ["folha amassada", "guardanapo
↪"], "organico": ["batata", "resto do bandeco", "casca de banana"]}
>>> lixo.clear()
>>> lixo
{}
```

11.5 Função `list()`

A função `list()` recebe um conjunto de objetos e retorna uma lista. Ao passar um dicionário, ela retorna uma lista contendo todas as suas *chaves*:

```
>>> institutos_uspsc = {"IFSC": "Instituto de Física de São Carlos", "ICMC":
↪"Instituto de Ciências Matemáticas e de Computação", "EESC": "Escola de Engenharia␣
↪de São Carlos", "IAU": "Instituto de Arquitetura e Urbanismo", "IQSC": "Instituto␣
↪de Química de São Carlos"}
>>> list(institutos_uspsc)
['IQSC', 'IFSC', 'ICMC', 'IAU', 'EESC']
```

11.6 Função `len()`

A função `len()` retorna o número de elementos («tamanho») do objeto passado para ela. No caso de uma lista, fala quantos elementos há. No caso de dicionários, retorna o *número de chaves* contidas nele:

```
>>> institutos_uspsc
{'IQSC': 'Instituto de Química de São Carlos', 'IFSC': 'Instituto de Física de São␣
↪Carlos', 'ICMC': 'Instituto de Ciências Matemáticas e de Computação', 'IAU':
↪'Instituto de Arquitetura e Urbanismo', 'EESC': 'Escola de Engenharia de São Carlos
↪'}
>>> len(institutos_uspsc)
5
```

Você pode contar o número de elementos na lista gerada pela função `list()` para conferir:

```
>>> len(list(institutos_uspsc))
5
```

11.7 Método `get()`

O método *get(chave, valor)* pode ser usado para retornar o valor associado à respectiva chave! O segundo parâmetro <*valor*> é opcional e indica o que será retornado caso a chave desejada *não* esteja no dicionário:

```
>>> institutos_uspsc.get("IFSC")
'Instituto de Física de São Carlos'
```

Dá para ver que ele é muito parecido com fazer assim:

```
>>> institutos_uspsc["IFSC"]
'Instituto de Física de São Carlos'
```

Mas ao colocarmos uma chave que não está no dicionário:

```
>>> institutos_uspsc.get("Poli", "Não tem!")
'Não tem!'
>>> institutos_uspsc["Poli"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Poli'
```

11.8 Alguns métodos

- O método `items()` pode ser comparado com o *inverso* da função `dict()`:

```
>>> pessoa = {"nome": "Enzo", "RA": 242334, "curso": "fiscomp"}
>>> pessoa.items()
dict_items([('curso', 'fiscomp'), ('nome', 'Enzo'), ('RA', 242334)])
```

  Usando a função `list()` nesse resultado, obtemos:

```
  >>> pessoa.items()
  dict_items([('curso', 'fiscomp'), ('nome', 'Enzo'), ('RA', 242334)])
  >>> itens = list(pessoa.items())
  >>> itens
  [('curso', 'fiscomp'), ('nome', 'Enzo'), ('RA', 242334)]

Experimente usar a função ``dict()`` na lista ``itens``!
```

- O método `values()` nos retorna os *valores* do dicionário:

```
>>> pessoa.values()
dict_values(['fiscomp', 'Enzo', 242334])
>>> valores = list(pessoa.values())
>>> valores
['fiscomp', 'Enzo', 242334]
```

- O método `keys()` nos retorna as *chaves* do dicionário:

```
>>> pessoa.keys()
dict_keys(['curso', 'nome', 'RA'])
>>> chaves = list(pessoa.keys())
>>> chaves
['curso', 'nome', 'RA']
```

  Repare que nesse último obtemos o mesmo que se tivéssemos usado a função `list()`:

```
>>> list(pessoa)
['curso', 'nome', 'RA']
```

11.9 Ordem dos elementos

Dicionários não tem sequência dos seus elementos. As listas têm. Dicionários mapeiam um valor a uma chave. Veja este exemplo:

```
>>> numerinhos = dict({"um": 1, "dois": 2, "três": 3})
>>> numeritos = {"três": 3, "dois": 2, "um": 1}
>>> numerinhos == numeritos
True
>>> numeritos
{'três': 3, 'dois': 2, 'um': 1}
>>> numerinhos
{'um': 1, 'dois': 2, 'três': 3}
```

Vemos que `numerinhos` e `numeritos` têm as mesmas chaves com os mesmos valores e por isso são iguais. Mas quando imprimimos cada um, a ordem que aparece é a que os itens foram inseridos.

11.10 Está no dicionário?

Podemos checar se uma chave está ou não em um dicionário utilizando o comando `in`. Voltando para o dicionário que contem os institutos da USP São Carlos:

```
>>> institutos_uspsc
{'IQSC': 'Instituto de Química de São Carlos', 'IFSC': 'Instituto de Física de São␣
↪Carlos', 'ICMC': 'Instituto de Ciências Matemáticas e de Computação', 'IAU':
↪'Instituto de Arquitetura e Urbanismo', 'EESC': 'Escola de Engenharia de São Carlos
↪'}
>>> "IFSC" in institutos_uspsc
True
>>> "ESALQ" in institutos_uspsc
False
```

E checamos se uma chave *não está* no dicionário com o comando `not in`:

```
>>> institutos_uspsc
{'IQSC': 'Instituto de Química de São Carlos', 'IFSC': 'Instituto de Física de São␣
↪Carlos', 'ICMC': 'Instituto de Ciências Matemáticas e de Computação', 'IAU':
↪'Instituto de Arquitetura e Urbanismo', 'EESC': 'Escola de Engenharia de São Carlos
↪'}
>>> "IFSC" not in institutos_uspsc
False
>>> "ESALQ" not in institutos_uspsc
True
```

11.11 Exercícios

1. Faça um dicionário com as 5 pessoas mais perto de você, tendo o nome como chave e a cor da camisa que está usando como valor.
2. Crie um dicionário vazio `semana = {}` e o complete com uma chave para cada dia da semana, tendo como seu valor uma lista com as aulas que você tem nesse dia (sábado e domingo recebem listas vazias, ou você tem aula?).
3. Crie um dicionário vazio `filmes = {}`. Utilize o nome de um filme como chave. E, como valor, *outro* dicionário contendo o `vilão` e o `ano` em que o filme foi lançado. Preencha 5 filmes.

CAPÍTULO 12

Condicionais

O tipo de dado boleano (`bool`) refere-se a uma unidade lógica sobre a qual podemos realizar operações, particularmente úteis para o controle de fluxo de um programa.

A unidade boleana assume apenas 2 valores: Verdadeiro (`True`) e Falso (`False`).

Nota: Essa estrutura binária é a forma com a qual o computador opera (0 e 1).

```
>>> True
True
>>> type(False)
<class 'bool'>
```

Qualquer expressão lógica retornará um valor boleano:

```
>>> 2 < 3
True
>>> 2 == 5
False
```

Os operadores lógicos utilizados em programação são:

- >: maior a, por exemplo 5 > 3
- <: menor a
- >=: maior ou igual a
- <=: menor ou igual a
- ==: igual a
- !=: diferente de

Para realizar operações com expressões lógicas, existem:

- **`and` (e): opera segundo a seguinte tabela:**

Valor 1	Valor 2	Resultado
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

- **`or` (ou):**

Valor 1	Valor 2	Resultado
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

- **`not` (não):**

Valor	Resultado
Verdadeiro	Falso
Falso	Verdadeiro

```
>>> 10 > 3 and 2 == 4
False

>>> 10 > 3 or 2 == 4
True

>>> not not not 1 == 1
False
```

Assim como os operadores aritméticos, os operadores boleanos também possuem uma ordem de prioridade:

- `not` tem maior prioridade que `and` que tem maior prioridade que `or`:

```
>>> not False and True or False
True
```

CAPÍTULO 13

Estruturas de controle

As estruturas de controle servem para decidir quais blocos de código serão executados.

Exemplo:

Se estiver nublado:
    Levarei guarda-chuva
Senão:
    Não levarei

Nota: Na linguagem Python, a indentação (espaço dado antes de uma linha) é utilizada para demarcar os blocos de código, e são obrigatórios quando se usa estruturas de controle.

```
>>> a = 7
>>> if a > 3:
...     print("estou no if")
... else:
...     print("cai no else")
...
estou no if
```

Também é possível checar mais de uma condição com o `elif`. É a abreviatura para `else if`. Ou seja, se o `if` for falso, testa outra condição antes do `else`:

```
>>> valor_entrada = 10
>>> if valor_entrada == 1:
...     print("a entrada era 1")
... elif valor_entrada == 2:
...     print("a entrada era 2")
... elif valor_entrada == 3:
...     print("a entrada era 3")
... elif valor_entrada == 4:
...     print("a entrada era 4")
```

(continues on next page)

(continued from previous page)

```
... else:
...     print("o valor de entrada não era esperado em nenhum if")
...
o valor de entrada não era esperado em nenhum if
```

Note que quando uma condição for verdadeira, aquele bloco de código é executado e as demais condições (`elif` e `else`) são puladas:

```
>>> a = 1
>>> if a == 1:
...     print("é 1")
... elif a >= 1:
...     print("é maior ou igual a 1")
... else:
...     print("é qualquer outra coisa")
...
é 1
```

13.1 Exercícios

1. Escreva um programa que, dados 2 números diferentes (a e b), encontre o menor deles.
2. Para doar sangue é necessário[1]:
   - Ter entre 16 e 69 anos.
   - Pesar mais de 50 kg.
   - Estar descansado (ter dormido pelo menos 6 horas nas últimas 24 horas).

   Faça um programa que pergunte a idade, o peso e quanto dormiu nas últimas 24 h para uma pessoa e diga se ela pode doar sangue ou não.
3. Considere uma equação do segundo grau $f(x) = a \cdot x^2 + b \cdot x + c$. A partir dos coeficientes, determine se a equação possui duas raízes reais, uma, ou se não possui.

   Dica: $\Delta = b^2 - 4 \cdot a \cdot c$: se delta é maior que 0, possui duas raízes reais; se delta é 0, possui uma raiz; caso delta seja menor que 0, não possui raiz real
4. Leia dois números e efetue a adição. Caso o valor somado seja maior que 20, este deverá ser apresentado somando-se a ele mais 8; caso o valor somado seja menor ou igual a 20, este deverá ser apresentado subtraindo-se 5.
5. Leia um número e imprima a raiz quadrada do número caso ele seja positivo ou igual a zero e o quadrado do número caso ele seja negativo.
6. Leia um número inteiro entre 1 e 12 e escreva o mês correspondente. Caso o usuário digite um número fora desse intervalo, deverá aparecer uma mensagem informando que não existe mês com este número.

[1] Para mais informações sobre doação de sangue, acesse http://www.prosangue.sp.gov.br/artigos/requisitos_basicos_para_doacao.html

CAPÍTULO 14

Estruturas de repetição

As estruturas de repetição são utilizadas quando queremos que um bloco de código seja executado várias vezes.

Em Python existem duas formas de criar uma estrutura de repetição:

- O `for` é usado quando se quer iterar sobre um bloco de código um número determinado de vezes.
- O `while` é usando quado queremos que o bloco de código seja repetido até que uma condição seja satisfeita. Ou seja, é necessário que uma expressão boleana dada seja verdadeira. Assim que ela se tornar falsa, o `while` para.

Nota: Na linguagem Python a indentação é obrigatória. assim como nas estruturas de controle, as estruturas de repetição também precisam.

```
>>> # Aqui repetimos o print 3 vezes
>>> for n in range(0, 3):
...     print(n)
...
0
1
2

>>> # Aqui iniciamos o n em 0, e repetimos o print até que seu valor seja maior ou␣
↪igual a 3
>>> n = 0
>>> while n < 3:
...     print(n)
...     n += 1
...
0
1
2
```

O *loop* `for` em Python itera sobre os itens de um conjunto, sendo assim, o `range(0, 3)` precisa ser um conjunto de elementos. E na verdade ele é:

```
>>> list(range(0, 3))
[0, 1, 2]
```

Para iterar sobre uma lista:

```
>>> lista = [1, 2, 3, 4, 10]
>>> for numero in lista:
...     print(numero ** 2)
...
1
4
9
16
100
```

Isso se aplica para *strings* também:

```
>>> # Para cada letra na palavra, imprimir a letra
>>> palavra = "casa"
>>> for letra in palavra:
...     print(letra)
...
c
a
s
a
```

Em dicionários podemos fazer assim:

```
>>> gatinhos = {"Português": "gato", "Inglês": "cat", "Francês": "chat", "Finlandês":
↪"Kissa"}
>>> for chave, valor in gatinhos.items():
...     print(chave, "->", valor)
...
Português -> gato
Inglês -> cat
Francês -> chat
Finlandês -> Kissa
```

Para auxiliar as estruturas de repetição, existem dois comandos:

- `break`: É usado para sair de um *loop*, não importando o estado em que se encontra.
- `continue`: Funciona de maneira parecida com a do `break`, porém no lugar de encerrar o *loop*, ele faz com que todo o código que esteja abaixo (porém ainda dentro do *loop*) seja ignorado e avança para a próxima iteração.

Veja a seguir um exemplo de um código que ilustra o uso desses comandos. Note que há uma *string* de documentação no começo que explica a funcionalidade.

```
"""
Esse código deve rodar até que a palavra "sair" seja digitada.
* Caso uma palavra com 2 ou menos caracteres seja digitada, um aviso
  deve ser exibido e o loop será executado do início (devido ao
  continue), pedindo uma nova palavra ao usuário.
* Caso qualquer outra palavra diferente de "sair" seja digitada, um
  aviso deve ser exibido.
* Por fim, caso a palavra seja "sair", uma mensagem deve ser exibida e o
  loop deve ser encerrado (break).
```

(continues on next page)

(continued from previous page)

```
"""

>>> while True:
...     string_digitada = input("Digite uma palavra: ")
...     if string_digitada.lower() == "sair":
...         print("Fim!")
...         break
...     if len(string_digitada) < 2:
...         print("String muito pequena")
...         continue
...     print("Tente digitar \"sair\"")
...
Digite uma palavra: oi
Tente digitar "sair"
Digite uma palavra: ?
String muito pequena
Digite uma palavra: sair
Fim!
```

14.1 Exercícios

1. Calcule a tabuada do 13.
2. Ler do teclado uma lista com 5 inteiros e imprimir o menor valor.
3. Ler do teclado uma lista com 5 inteiros e imprimir `True` se a lista estiver ordenada de forma crescente ou False caso contrário.
4. Exiba em ordem decrescente todos os números de 500 até 10.
5. Ler do teclado 10 números e imprima a quantidade de números entre 10 e 50.
6. Ler do teclado a idade e o sexo de 10 pessoas, calcule e imprima:
    a) idade média das mulheres
    b) idade média dos homens
    c) idade média do grupo
7. Calcule o somatório dos números de 1 a 100 e imprima o resultado.