

Listas, Tuplas e Dicionários



Prof. André Backes | @progdescomplicada

Listas

Por que usar listas?

- As variáveis declaradas até agora são capazes de armazenar um único valor por vez.
 - Sempre que tentamos armazenar um novo valor dentro de uma variável, o valor antigo é sobrescrito e, portanto, perdido

```
>>>  
>>> x = 10  
>>> x  
10  
>>> x = 20  
>>> x  
20  
>>>
```

Listas

- A lista é a forma mais familiar de dados estruturados
 - Basicamente, uma lista é uma sequência de elementos, onde cada elemento é identificado por um índice
 - Diferente dos arrays usados em outras linguagens, em Python os elementos de uma lista podem possuir qualquer tipo

Listas | Problema

- Imagine o seguinte problema
 - leia as notas de uma turma de cinco estudantes e depois imprima as notas que são maiores do que a média da turma
- Um algoritmo para esse problema poderia ser o mostrado ao lado

```
n1 = int(input("Nota do aluno 1: "))
n2 = int(input("Nota do aluno 2: "))
n3 = int(input("Nota do aluno 3: "))
n4 = int(input("Nota do aluno 4: "))
n5 = int(input("Nota do aluno 5: "))

media = (n1 + n2 + n3 + n4 + n5)/5
if n1 > media:
    print("Aprovado: ",n1)
if n2 > media:
    print("Aprovado: ",n2)
if n3 > media:
    print("Aprovado: ",n3)
if n4 > media:
    print("Aprovado: ",n4)
if n5 > media:
    print("Aprovado: ",n5)
```

Listas

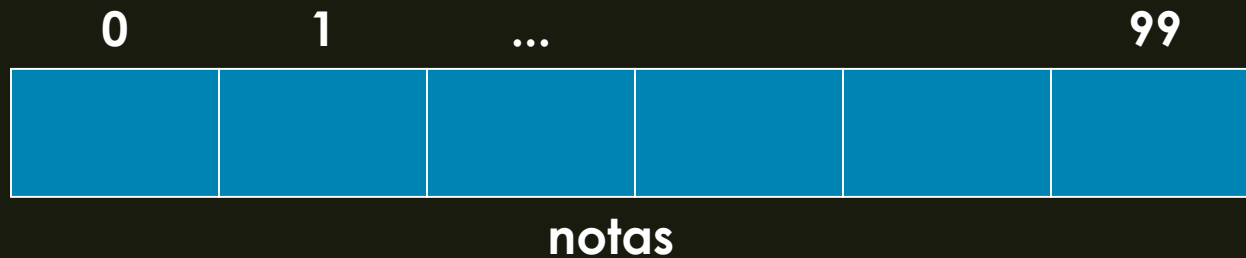
- O algoritmo anterior apresenta uma solução possível para o problema apresentado
- Porém, essa solução é inviável para grandes quantidades de alunos
 - Imagine se tivéssemos de processar os dados de 100 alunos

Listas

- Para 100 alunos, precisamos de:
 - Uma variável para armazenar a nota de cada aluno: 100 variáveis
 - Um comando de leitura para cada nota: 100 `input()`
 - Um somatório de 100 notas
 - Um comando de teste para cada aluno: 100 comandos `if`
 - Um comando de impressão na tela para cada aluno: 100 `print()`

Listas | Definição

- As variáveis têm relação entre si
 - todas armazenam notas de alunos
- Podemos declará-las usando um ÚNICO nome para todos os 100 alunos
 - notas = conjunto de 100 números acessados por um índice
 - **isso é uma lista!**



Listas | Declaração

- Existem várias maneiras de criar uma lista
 - Uma das mais simples é definir os elementos da lista entre colchetes
 - Os elementos da lista devem ser separados por vírgulas
 - Se nenhum elemento for definido, temos uma lista vazia
 - Também podemos usar a função **range()** para gerar uma lista de valores

```
>>>
>>> compras = ["Miojo", "Ovo", "Leite", "Pão"]
>>> compras
['Miojo', 'Ovo', 'Leite', 'Pão']
>>>
>>> compras = []
>>> compras
[]
>>>
>>>
>>> list(range(5))
[0, 1, 2, 3, 4]
>>>
```

Listas | Declaração

- Em uma lista, os elementos são acessados especificando o índice desejado entre colchetes. A numeração começa sempre do zero
 - Isto significa que uma lista de 4 elementos terá índices de 0 a 3: `compras[0]`, `compras[1]`, `compras[2]` e `compras[3]`
 - A função **len()** retorna o número de elementos da lista

```
>>>  
>>> compras = ["Miojo", "Ovo", "Leite", "Pão"]  
>>> compras[0]  
'Miojo'  
>>> compras[1]  
'Ovo'  
>>> len(compras)  
4  
>>>
```

0	1	2	3
Miojo	Ovo	Leite	Pão

compras

Lista = variável

- Cada elemento da lista tem todas as características de uma variável e pode aparecer em expressões e atribuições (respeitando os seus tipos)
 - `notas[2] = x + notas[3]`
 - `if notas[3] > 60:`
- Exemplo
 - Somar todos os elementos de uma lista

```
lista = [3, 5, 10, 2, 4]
soma = 0
for x in lista:
    soma = soma + x

print("Soma = ",soma)
```

Percorrendo uma lista

- Podemos usar um comando de repetição (preferencialmente o for) para percorrer a lista de 2 formas
- Forma 1
 - Percorrer os índices e elementos

```
compras = ["Miojo", "Ovo", "Leite", "Pão"]  
for i in range(len(compras)):  
    print(i, compras[i])
```

```
>>>  
0 Miojo  
1 Ovo  
2 Leite  
3 Pão  
>>>
```

Percorrendo uma lista

- Podemos usar um comando de repetição (preferencialmente o for) para percorrer a lista de 2 formas
- Forma 2
 - Percorrer apenas os elementos. Isso evita manipular explicitamente o índice

```
compras = ["Miojo", "Ovo", "Leite", "Pão"]  
for x in compras:  
    print(x)
```

```
>>>  
Miojo  
Ovo  
Leite  
Pão  
>>>
```

Listas | Problema

- Voltando ao problema anterior
 - leia as notas de uma turma de cinco estudantes e depois imprima as notas que são maiores do que a média da turma

Listas | Solução

- Um algoritmo para esse problema usando listas

```
N = 5
notas = []
for i in range(1,N+1):
    x = int(input("Nota do aluno: "))
    notas.append(x)

media = 0
for n1 in notas:
    media = media + n1
media = media / N

for n1 in notas:
    if n1 > media:
        print("Aprovado: ",n1)
```

Listas | Solução

- Se ao invés de 5, fossem 100 alunos?

```
N = 100
notas = []
for i in range(1, N+1):
    x = int(input("Nota do aluno: "))
    notas.append(x)

media = 0
for n1 in notas:
    media = media + n1
media = media / N

for n1 in notas:
    if n1 > media:
        print("Aprovado: ", n1)
```


Listas | Exercício

- Para uma lista contendo 5 números inteiros, formular um algoritmo que determine o maior elemento desta lista

Listas | Exercício

- Uma possível solução para o problema

```
lista = [3, 5, 10, 2, 4]
ma = lista[0]
for x in lista:
    if ma < x:
        ma = x

print("Maior = ",ma)
```

Acessando seus elementos

- As listas suportam acesso a sub-listas, isto é, a certos conjuntos de índices
 - `lista[i:j]`
 - seleciona a sub-lista dos índices `i` até `j-1`
 - `lista[i:]`
 - seleciona a sub-lista dos índices `i` até o final
 - `lista[:j]`
 - seleciona a sub-lista do início até o índice `j-1`
 - `lista[i:j:k]`
 - seleciona a sub-lista dos índices `i` até `j-1`, indo de `k` em `k`
 - `i, i+k, i+2k, ..., j-1`

Acessando seus elementos

- Selecionando sub-listas de elementos

```
>>>
>>> nros = [10,20,30,40,50]
>>> nros[2:4]
[30, 40]
>>> nros[2:]
[30, 40, 50]
>>> nros[:3]
[10, 20, 30]
>>> nros[:]
[10, 20, 30, 40, 50]
>>> nros[0:5:2]
[10, 30, 50]
>>> nros[4:0:-1]
[50, 40, 30, 20]
>>>
```

Acessando seus elementos

- Listas são objetos *mutáveis*
 - Podemos acessar qualquer elemento ou sequência de elementos e modificá-los
 - Os elementos de uma lista podem possuir qualquer tipo

```
>>>  
>>> lista = [1, 2, 3]  
>>> lista[0] = "novo"  
>>> lista  
['novo', 2, 3]  
>>>  
>>> lista[2:3] = [10, 20]  
>>> lista  
['novo', 2, 10, 20]  
>>>
```

Acessando seus elementos

- Podemos também associar o valor de cada posição da lista a uma variável sem precisar especificar os índices usando o recurso de ***unpacking***
 - Basta definir uma lista de variáveis, entre colchetes, que receberá o conteúdo da lista

```
>>> pontos = [1, 2, 3]
>>> pontos
[1, 2, 3]
>>>
>>> [x, y, z] = pontos
>>> x
1
>>> y
2
>>> z
3
>>>
```

Listas | Concatenação e repetição

- Podemos unir/concatenar duas listas para formar uma nova usando o operador de +
- Podemos criar uma lista a partir da repetição de outra usando o operador de *

```
>>>
>>> lista1 = [1, 2, 3]
>>> lista2 = [4, 5, 6]
>>> lista1 + lista2
[1, 2, 3, 4, 5, 6]
>>> lista3 = lista1 + lista2
>>> lista3
[1, 2, 3, 4, 5, 6]
>>>
>>> lista = 3 * lista1
>>> lista
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

Listas | Remoção de elementos

- Podemos remover um ou mais elementos de uma lista de duas maneiras
 - Usando o operador **del**
 - Atribuindo uma lista vazia àquela posição

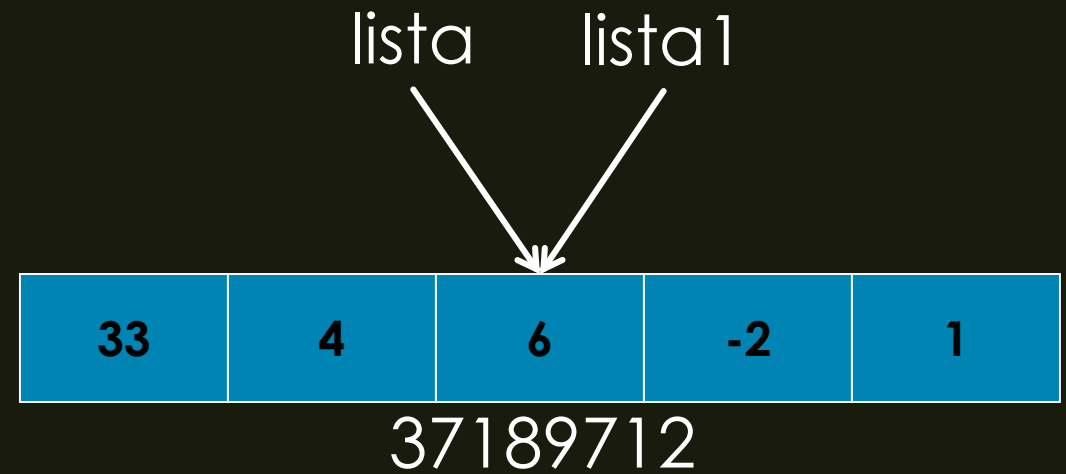
```
>>>
>>> lista = [1, 2, 3, 4, 5, 6]
>>> lista
[1, 2, 3, 4, 5, 6]
>>>
>>> del lista[1]
>>> lista
[1, 3, 4, 5, 6]
>>>
>>> lista = [1, 2, 3, 4, 5, 6]
>>> lista
[1, 2, 3, 4, 5, 6]
>>>
>>> lista[1:2] = []
>>> lista
[1, 3, 4, 5, 6]
>>>
```


Listas | Cópia de listas

- A operação de atribuição não cria uma cópia do objeto!
 - Em Python, listas são objetos. Se atribuirmos uma lista a outra, ambas irão se referir ao mesmo objeto

Listas | Cópia de listas

```
>>>  
>>> lista = [33, 4, 6, -2, 1]  
>>> lista  
[33, 4, 6, -2, 1]  
>>> lista1 = lista  
>>> lista1  
[33, 4, 6, -2, 1]  
>>>  
>>> id(lista)  
37189712  
>>> id(lista1)  
37189712  
>>>
```



Listas | Cópia de listas

- Para copiar uma lista, o correto é utilizar `[:]` na operação de atribuição

Listas | Cópia de listas

```
>>>  
>>> lista = [33, 4, 6, -2, 1]  
>>> lista  
[33, 4, 6, -2, 1]  
>>> lista1 = lista[:]  
>>> lista1  
[33, 4, 6, -2, 1]  
>>>  
>>> id(lista)  
36998160  
>>> id(lista1)  
37415712  
>>>
```

lista

33	4	6	-2	1
----	---	---	----	---

36998160

lista1

33	4	6	-2	1
----	---	---	----	---

37415712

Listas | Procurando um elemento

- O operador **in** permite verificar se um determinado elemento está presente ou não em uma lista

```
lista = [1,2,3,4,5,6]

if 2 in lista:
    print("Valor existe na lista!")
else:
    print("Valor não existe na lista")
```

Métodos sobre listas

- Uma lista é uma classe e, portanto, possui diversos métodos já definidos
 - Um dos jeitos mais simples de manipular listas é utilizar os métodos que já fazem parte dela
- Esses métodos permitem executar diversas tarefas
 - Ordenação, inserção, remoção, etc
 - Esses métodos modificam o conteúdo original da lista

Métodos sobre listas

- Forma geral de uso dos métodos
 - **lista.nome-método()**
- Alguns métodos
 - `sort()`: ordena os elementos da lista
 - `append(x)`: insere um elemento **x** no final da lista
 - `insert(pos,x)`: insere um elemento **x** na posição **pos**
 - `remove(x)`: remove o elemento **x** da lista
 - `pop(pos)`: remove e retorna o elemento da posição **pos**

Métodos sobre listas | Exemplos

```
>>>
>>> lista = list(range(5))
>>> lista
[0, 1, 2, 3, 4]
>>>
>>> lista.append(5)
>>> lista
[0, 1, 2, 3, 4, 5]
>>>
>>> lista.insert(0,-1)
>>> lista
[-1, 0, 1, 2, 3, 4, 5]
>>>
>>> lista.insert(2,"meio")
>>> lista
[-1, 0, 'meio', 1, 2, 3, 4, 5]
>>>
>>> lista.remove(1)
>>> lista
[-1, 0, 'meio', 2, 3, 4, 5]
```

```
>>>
>>> lista = [33, 4, 6, -2, 1]
>>> lista
[33, 4, 6, -2, 1]
>>>
>>> lista.sort()
>>> lista
[-2, 1, 4, 6, 33]
>>>
>>> x = lista.pop(2)
>>> x
4
>>> lista
[-2, 1, 6, 33]
>>>
```


Lista aninhadas

- Uma lista pode armazenar qualquer tipo de dado
 - As listas podem inclusive conter outras listas
 - Permite representar tabelas ou matrizes

	0	1	...	49
0		99		
1				
...				
99				

Lista aninhadas

- Para criar uma lista aninhada, basta definir cada elemento como uma nova lista
 - Os elementos são acessados especificando um par de colchetes e índice para cada dimensão da lista
 - A numeração começa sempre do zero

```
>>>
>>> matriz = [[1,2,3],[4,5,6],[7,8,9]]
>>> matriz
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
>>> matriz[0][1]
2
>>> matriz[0][1] = "oi"
>>> matriz
[[1, 'oi', 3], [4, 5, 6], [7, 8, 9]]
>>>
```

	0	1	2
0	1	'oi'	3
1	4	5	6
2	7	8	9

Lista aninhadas

- Podemos também incluir novas linhas e colunas em cada lista
 - Recomenda-se, neste caso, usar os métodos ao invés do operador de concatenação
 - Nas listas aninhadas, não é necessários que as linhas tenham sempre o mesmo número de colunas

```
>>>  
>>> matriz = [[1,2,3],[4,5,6],[7,8,9]]  
>>> matriz.append([10,11,12])  
>>> matriz  
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]  
>>>
```

Lista aninhadas

- Podemos remover um ou mais elementos de uma lista aninhada de maneira similar as listas tradicionais
 - 1 índice: a linha inteira é removida
 - 2 índices: o elemento da linha e coluna é removido

```
>>> matriz
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
>>>
>>>
>>> del matriz[0][0]
>>> matriz
[[2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
>>>
>>> del matriz[1]
>>> matriz
[[2, 3], [7, 8, 9], [10, 11, 12]]
>>>
```

Lista aninhadas

- Todas as operações feitas em listas aninhadas devem considerar o fato de que temos agora uma lista dentro de outra

```
#impressão
matriz = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
for linha in matriz:
    for elem in linha:
        print(elem)

#soma
matriz = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
soma = 0
for linha in matriz:
    for elem in linha:
        soma = soma + elem

print("Soma = ",soma)
```

Compreensão de lista

- Trata-se de uma construção sintática para criação de listas
 - Segue a forma da notação de definição de conjunto matemática
 - É uma maneira de fazer caber uma repetição (**for**), uma condição (**if**) e uma tarefa, em uma única linha
 - Permite mapear e filtrar uma lista em uma única expressão

Compreensão de lista

- Imagine que queremos criar uma lista com os quadrados de alguns números
- Podemos fazer um laço **for** ou usar compreensão de lista

```
quadrados = []  
for x in range(10):  
    quadrados.append(x**2)
```

```
quadrados = [x**2 for x in range(10)]  
quadrados  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Compreensão de lista

- Imagine agora que queremos filtrar os elementos de uma lista já existente
- Podemos fazer um laço **for** ou usar compreensão de lista

```
numeros = [1,2,3,4,5]
menores = []
for n in numeros:
    if n < 4:
        menores.append(n)
```

```
numeros = [1,2,3,4,5]
menores = [n for n in numeros if n < 4]
menores
[1, 2, 3]
```


Compreensão de lista

- Podemos também combinar as tarefas de criação e filtragem

```
numeros = [1,2,3,4,5]
quadrados = []
for n in numeros:
    if n < 4:
        quadrados.append(n**2)

numeros = [1,2,3,4,5]
quadrados = [n**2 for n in numeros if n < 4]
quadrados
[1, 4, 9]
```

Tuplas

Tuplas | Definição

- As tuplas se comportam como as listas
 - São uma sequência arbitrária de elementos
 - Cada elemento pode ser acessado por um índice inteiro
- Diferem das listas por serem definidas utilizando parênteses e não colchetes

```
>>>  
>>> aluno = ("Ricardo", 1.74, 33)  
>>> aluno  
(  
'Ricardo', 1.74, 33)  
>>>  
>>>  
>>> aluno[0]  
'Ricardo'  
>>> aluno[1]  
1.74  
>>> aluno[2]  
33  
>>>
```

Tuplas | Definição

- No entanto, diferente das listas, os elementos de uma tupla são **imutáveis**, ou seja, não podem ser modificados

```
>>>
>>> aluno = ("Ricardo", 1.74, 33)
>>> aluno
('Ricardo', 1.74, 33)
>>>
>>> aluno[0] = "André"
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    aluno[0] = "André"
TypeError: 'tuple' object does not support item assignment
>>>
```

Tuplas | Concatenação e repetição

- Podemos acessar os elementos de uma tupla, mas não podemos modificá-los
 - Felizmente, podemos construir uma outra tupla via concatenação e repetição

```
>>>
>>> aluno1 = ("Ricardo", 1.74, 33)
>>> aluno2 = ("Bianca", 1.60, 34)
>>> alunos = aluno1 + aluno2
>>> alunos
('Ricardo', 1.74, 33, 'Bianca', 1.6, 34)
>>>
>>> alunos = 2 * aluno1
>>> alunos
('Ricardo', 1.74, 33, 'Ricardo', 1.74, 33)
>>>
```

Tuplas | Inicialização

- Um problema com a construção de uma tupla é quando ela tem ZERO ou UM elemento
 - Tuplas vazias são definidas por um par de parênteses vazio
 - Tuplas com 1 elemento devem possuir uma vírgula depois do elemento para serem consideradas tuplas

```
>>>  
>>> tupla = ()  
>>> type(tupla)  
<class 'tuple'>  
>>>  
>>> tupla = (1)  
>>> type(tupla)  
<class 'int'>  
>>>  
>>> tupla = (1,)  
>>> type(tupla)  
<class 'tuple'>  
>>>
```

Tuplas | Acessando seus elementos

- Como nas listas, nas tuplas também podemos associar o valor de cada posição a uma variável sem precisar especificar os índices usando o recurso de ***unpacking***
 - Basta definir uma lista de variáveis, entre parênteses, que receberá o conteúdo da tupla

```
>>>
>>> aluno = ("Ricardo", 1.74, 33)
>>>
>>> (nome, altura, idade) = aluno
>>> nome
'Ricardo'
>>> altura
1.74
>>> idade
33
>>>
```

Por que usar tuplas?

- Apesar de listas e tuplas serem semelhantes, elas são normalmente utilizadas com propósitos diferentes
 - Tuplas são imutáveis e normalmente contém elementos de tipos diferentes que são acessados via **unpacking** ou indexação
 - Listas são mutáveis e normalmente contém elementos do mesmo tipo que são acessados via iteração

Dicionários

Dicionários | Definição

- São estruturas de dados que implementam associações entre pares de valores
 - O primeiro elemento do par é chamado de **chave** e o outro de **conteúdo**
 - Cada chave é associada a um (e um só) conteúdo

Dicionários | Definição

- Exemplo: inventário
 - Um inventário que associa produtos a quantidades disponíveis
 - Miojo: 10
 - Ovo: 12
 - Leite: 2
 - Pão: 5
 - Esse problema poderia ser representado por uma lista de tuplas. Porém, dicionários permitem busca indexada pela **chave**
 - Não é necessário percorrer a lista procurando um item

Dicionários | Definição

- Um dicionário é definido utilizando chaves { } e cada par chave-valor é separado por dois pontos :
 - Diferentes pares de chave-valor são separados por vírgula
- Um dicionário vazio é definido por um par de chaves vazio

```
diccionario = { chave1:conteúdo1, ..., chaveN:conteúdoN }  
  
>>>  
>>> dic = {}  
>>> dic  
{ }  
>>>
```

Dicionários | Definição

- Exemplos: dicionário de palavras em português e inglês
 - Note que as chaves do dicionário não são armazenadas em qualquer ordem específica

```
>>> dic = {"um": "one", "dois": "two", "tres": "three"}
>>> dic
{'tres': 'three', 'dois': 'two', 'um': 'one'}
>>>
```

Dicionários | Acessando seus elementos

- Cada elemento pode ser acessado por indexação usando a chave
 - Podemos alterar o conteúdo associado a uma chave
- Novos itens podem ser adicionados a um dicionário
 - Basta fazer a atribuição a uma chave ainda não definida

```
>>>
>>> dic = {"um": "one", "dois": "two", "tres": "three"}
>>> dic
{'tres': 'three', 'dois': 'two', 'um': 'one'}
>>> dic["dois"]
'two'
>>>
>>> dic["dois"] = 2
>>> dic
{'tres': 'three', 'dois': 2, 'um': 'one'}
>>>
```

Métodos sobre dicionários

- Um dicionário é uma classe e, portanto, possui diversos métodos já definidos
 - Um dos jeitos mais simples de manipular dicionários é utilizar os métodos que já fazem parte dele
- Forma geral de uso dos métodos
 - **dicionário.nome-método()**

Métodos sobre dicionários

- **clear()**: remove todos os elementos do dicionário
- **copy()**: cria uma cópia do dicionário (atribuição não cria cópia)

```
>>>
>>> dic = {"um": "one", "dois": "two", "tres": "three"}
>>> dic.clear()
>>> dic
{}

>>> dic = {"um": "one", "dois": "two", "tres": "three"}
>>> dic1 = dic
>>> dic2 = dic.copy()
>>> id(dic)
34433864
>>> id(dic1)
34433864
>>> id(dic2)
34434104
```


Métodos sobre dicionários

- **get(chave,valor)**: obtém o conteúdo de chave. Caso a chave não exista, retorna **valor**

```
>>>  
>>> dic = {"um": "one", "dois": "two", "tres": "three"}  
>>> dic.get("um",0)  
'one'  
>>> dic.get("cinco",0)  
0  
>>>
```

Métodos sobre dicionários

- **items()**: retorna uma lista com todos os pares chave/conteúdo do dicionário
- **keys()**: retorna uma lista com todas as chaves do dicionário
- **values()**: retorna uma lista com todos os valores do dicionário

```
>>>  
>>> dic = {"um": "one", "dois": "two", "tres": "three"}  
>>> dic.items()  
dict_items([('tres', 'three'), ('dois', 'two'), ('um', 'one')])  
>>> dic.keys()  
dict_keys(['tres', 'dois', 'um'])  
>>> dic.values()  
dict_values(['three', 'two', 'one'])  
>>>
```

Métodos sobre dicionários

- **pop(chave)**: obtém o valor correspondente a chave e remove o par chave/valor do dicionário
- **popitem()**: retorna e remove um par chave/valor aleatório do dicionário. Pode ser usado para iterar sobre todos os elementos do dicionário

```
>>>  
>>> dic = {"um": "one", "dois": "two", "tres": "three"}  
>>> dic.pop("um")  
'one'  
>>> dic.popitem()  
( 'tres', 'three' )  
>>> dic  
{ 'dois': 'two' }  
>>>
```

Material Complementar

- Vídeo Aulas
 - Aula 16 - Listas
 - <https://youtu.be/Y05YeBAFCrs>
 - Aula 17 - Listas: criando sub-listas
 - <https://youtu.be/0sdtkjlSZIE>
 - Aula 18 - Listas: operações e métodos
 - <https://youtu.be/ZMkN0BXd-3o>
 - Aula 19 - Listas aninhadas
 - https://youtu.be/eJHpe_-izj4

Material Complementar

- Vídeo Aulas
 - Aula 20 - Compreensão de listas
 - <https://youtu.be/oui2ObnvYCY>
 - Aula 21 - Tuplas
 - <https://youtu.be/fhekxMRM4Ts>
 - Aula 22 - Dicionários
 - <https://youtu.be/AZyH-ueMvhY>
 - Aula 23 - Dicionários: operações e métodos
 - <https://youtu.be/EXMr1u2bnjc>