



**CIÊNCIAS
EMPRESARIAIS**
ESCOLA SUPERIOR
POLITÉCNICO SETÚBAL

Fundamentos de Programação

Aula 6

Mestrado em Ciência de Dados para Empresas

David Simões

VARIÁVEIS GLOBAIS: MAU ESTILO

```
# Constante – visível a todas as funções
```

```
NUM_DIAS_SEMANA = 7
```

```
# Variável global – visível a todas as funções
```

```
saldo = 0
```

```
def main():
```

```
    saldo = int(input("Saldo inicial: "))
```

```
    while True:
```

```
        valor = int(input("Depósito (0 para sair): "))
```

```
        if valor == 0:
```

```
            break
```

```
            depositar(valor)
```

```
def depositar(valor):
```

```
    saldo += valor
```

Variáveis diferentes com o mesmo nome!
Ou usar a variável global? Super confuso!

Além disso, muito MAU estilo

Tão mau que o Python não deixa usar
a variável global a não ser que
adicionemos um comando a dizer
basicamente “quero ter mau estilo”

Não vamos ver esse comando aqui 😊

USAR PARÂMETROS: BOM ESTILO



Não queremos que usar a torradeira tenha impacto no frigorífico!



```
def main():
    saldo = int(input("Saldo inicial: "))
    while True:
        valor = int(input("Depósito (0 para sair): "))
        if valor == 0:
            break
        saldo = depositar(saldo, valor)

def depositar(saldo, valor):
    saldo += valor
    return saldo
```

PRINCÍPIO DE ENCAPSULAMENTO
Os dados usados por uma função devem estar encapsulados na função ou ser recebidos por parâmetros

A CONSOLA PYTHON

- Podemos correr o Python interativamente através da “consola”
 - No PyCharm, clicar na tab “Python Console”
 - No Terminal, correr o Python (com “py” ou “python3”, dependendo da plataforma) para obter a consola
- A consola tem a prompt: >>>
 - Podemos escrever e executar comandos Python (e ver os resultados)
 - Exemplo:

```
>>> x = 5
>>> x
5
```
- Forma fácil de experimentar coisas para esclarecer alguma questão
- Para sair, usar **exit()**

Vamos experimentar a consola

NENHUM VALOR

- O termo **None** é usado no Python para descrever “sem valor”

- Por exemplo, é o valor que receberíamos de uma função que não retorna nada
 - WHAT?!
 - Exemplo:

```
>>> x = print("olá")
```

```
olá
```

```
>>> print(x)
```

```
None
```

- Comparar qualquer coisa com **None** (excepto **None**) é Falso
- Porque é que o **None** existe?
 - Denota quando a mala de uma variável não tem “nada” lá dentro

OBJETIVOS DE APRENDIZAGEM

1. Aprender listas em Python
2. Escrever código que usa listas
3. Perceber como é que as listas são passadas como parâmetros

PARTE I



Listas

O QUE É UMA LISTA?

- Uma **lista** é uma forma de gerir uma *coleção ordenada* de itens
 - Os itens de uma lista chamam-se “elementos”
 - Coleção: uma lista pode conter múltiplos elementos
 - Ordenados: podemos referir os elementos pela sua posição
- A lista ajusta o seu tamanho dinamicamente à medida que os elementos são adicionados ou removidos
- As listas têm muitas funcionalidades incluídas para ser mais fácil usá-las

MOSTREM-ME AS LISTAS!

- Criar listas

- As listas começam/acabam com parêntesis retos. Os elementos separam-se com vírgulas.

```
minha_lista = [1, 2, 3]
reais = [4.7, -6.0, 0.22, 1.6]
strs = ['muitas', 'strings', 'na', 'lista']
mix = [4, 'olá', -3.2, True, 6]
lista_vazia = []
```

- Lista com um elemento não é o mesmo que o elemento

- Podemos experimentar na consola

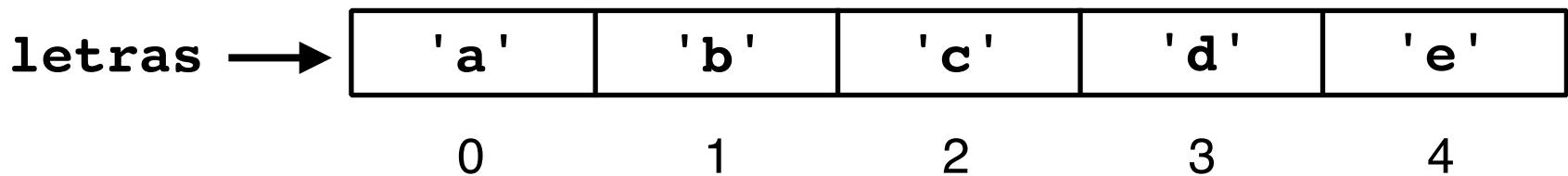
```
>>> lista_um = [1]
>>> um = 1
>>> lista_um == um
False
```

ACEDER AOS ELEMENTOS DE UMA LISTA

- Consideremos a seguinte lista:

```
letras = [ 'a', 'b', 'c', 'd', 'e' ]
```

- Podemos pensar na lista como uma sequência de variáveis que estão indexadas
 - Os índices começam de 0



- Aceder a elementos individuais:

letras[0] é 'a'

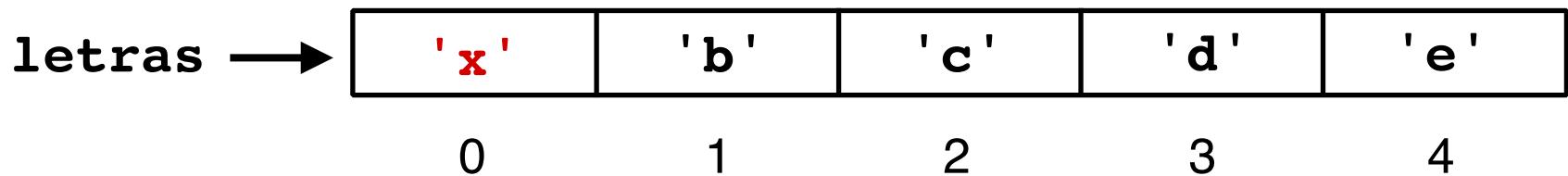
letras[4] é 'e'

ACEDER AOS ELEMENTOS DE UMA LISTA

- Consideremos a seguinte lista:

```
letras = [ 'a', 'b', 'c', 'd', 'e' ]
```

- Podemos pensar na lista como uma sequência de variáveis que estão indexadas
 - Os índices começam de 0



- Aceder a elementos individuais:

letras[0] é 'a'

letras[4] é 'e'

- Podemos atribuir valores aos elementos individuais:

letras[0] = 'x'

OBTER O COMPRIMENTO DE UMA LISTA

- Consideremos a seguinte lista:

```
letras = ['a', 'b', 'c', 'd', 'e']
```

- Podemos obter o comprimento de uma lista com a função `len`:

`len(letras)` é 5

- Os elementos de uma lista são indexados desde 0 até `length - 1`

- Exemplo:

```
for i in range(len(letras)):  
    print(i, "->", letras[i])
```

0	→	a
1	→	b
2	→	c
3	→	d
4	→	e

COMPRIMENTO DE UMA LISTA: CURSO AVANÇADO

- Voltemos aos nossos amigos:

```
minha_lista = [1, 2, 3]
reais = [4.7, -6.0, 0.22, 1.6]
strs = ['muitas', 'strings', 'na', 'lista']
mix = [4, 'olá', -3.2, True, 6]
lista_vazia = []
```

- Pop quiz!

<code>len(minha_lista)</code>	= 3
<code>len(reais)</code>	= 4
<code>len(strs)</code>	= 4
<code>len(mix)</code>	= 5
<code>len(lista_vazia)</code>	= 0

A ESTRANHEZA DA INDEXAÇÃO

- Podemos usar índices negativos para contar para trás a partir do fim da lista
 - What!?

```
letras = ['a', 'b', 'c', 'd', 'e']
```

- Vamos à estranheza!

`letras[-1]` é 'e'

`letras[-2]` é 'd'

`letras[-5]` é 'a'

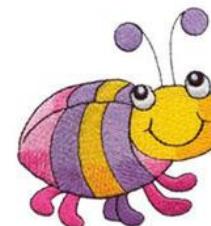
- Para os índices, `-x` é o mesmo que `len(lista)-x`

`letras[-1]` é o mesmo que `letras[len(letras)-1]`

- Então e isto?

```
letras[6]
```

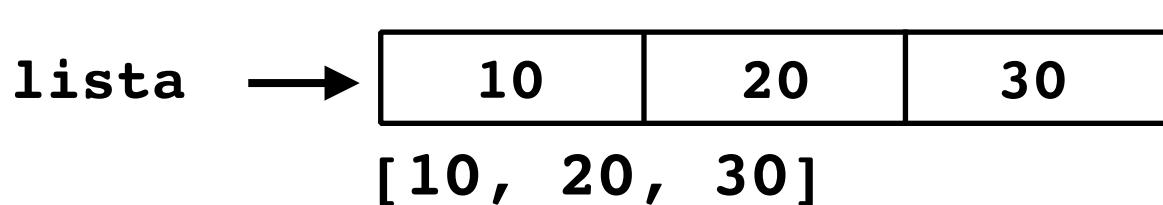
`IndexError: list index out of range`



CONSTRUINDO LISTAS

- Podemos adicionar elementos ao final duma lista com `.append`

```
lista = [10, 20, 30]
```

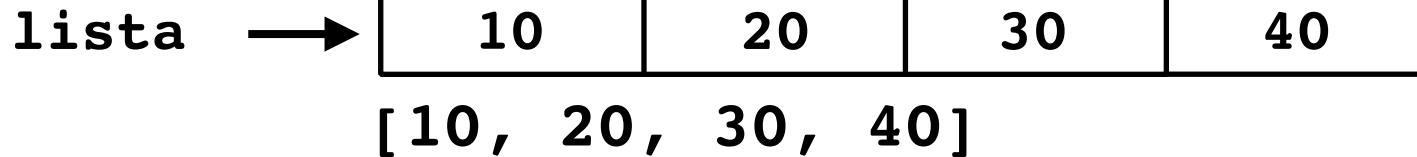


CONSTRUINDO LISTAS

- Podemos adicionar elementos ao final duma lista com `.append`

```
lista = [10, 20, 30]
```

```
lista.append(40)
```



CONSTRUINDO LISTAS

- Podemos adicionar elementos ao final duma lista com `.append`

```
lista = [10, 20, 30]
```

```
lista.append(40)
```

```
lista.append(50)
```

lista →

10	20	30	40	50
----	----	----	----	----

[10, 20, 30, 40, 50]

CONSTRUINDO LISTAS

- Podemos adicionar elementos ao final duma lista com `.append`

```
lista = [10, 20, 30]
```

```
lista.append(40)
```

```
lista.append(50)
```

```
outra_lista = []
```

outra_lista → *lista vazia*

[]

lista

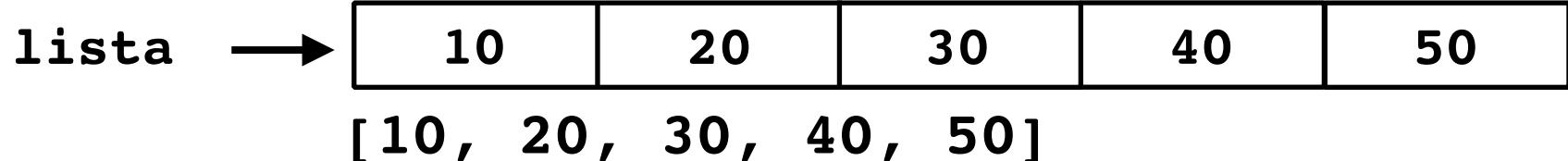
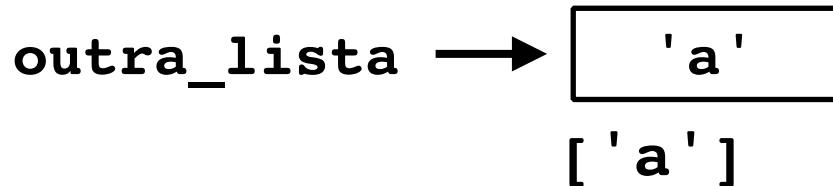
→

10	20	30	40	50
[10, 20, 30, 40, 50]				

CONSTRUINDO LISTAS

- Podemos adicionar elementos ao final duma lista com `.append`

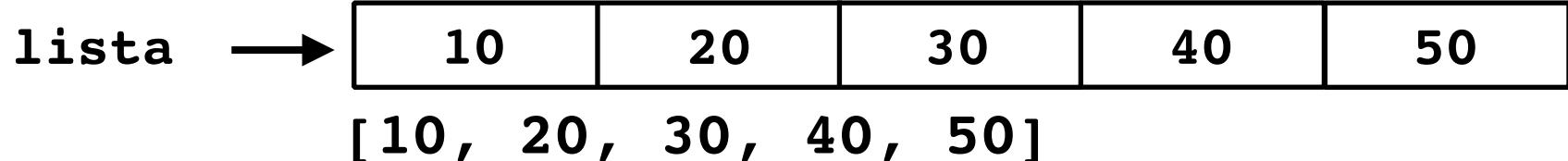
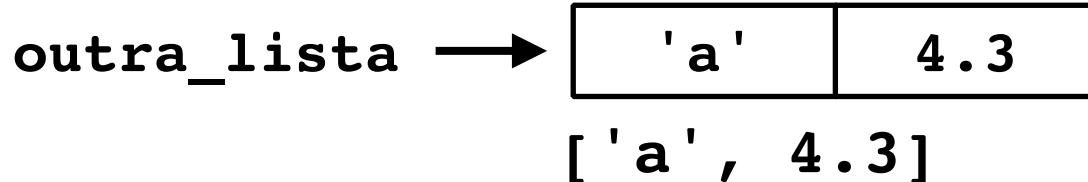
```
lista = [10, 20, 30]
lista.append(40)
lista.append(50)
outra_lista = []
outra_lista.append('a')
```



CONSTRUINDO LISTAS

- Podemos adicionar elementos ao final duma lista com `.append`

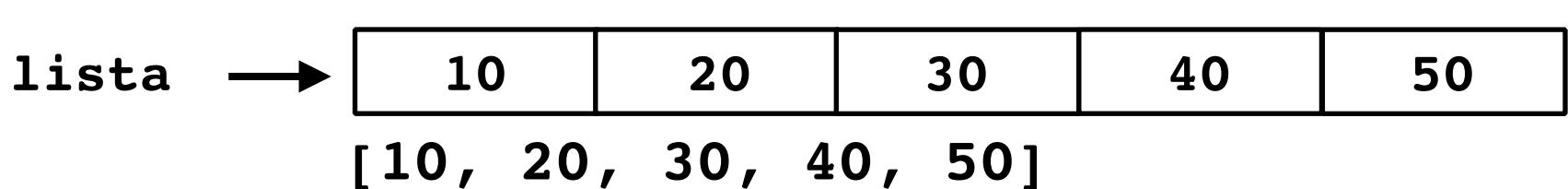
```
lista = [10, 20, 30]
lista.append(40)
lista.append(50)
outra_lista = []
outra_lista.append('a')
outra_lista.append(4.3)
```



REMOVENDO ELEMENTOS DE LISTAS

- Podemos remover elementos do final duma lista com `.pop`
 - Remove o último elemento da lista e retorna-o

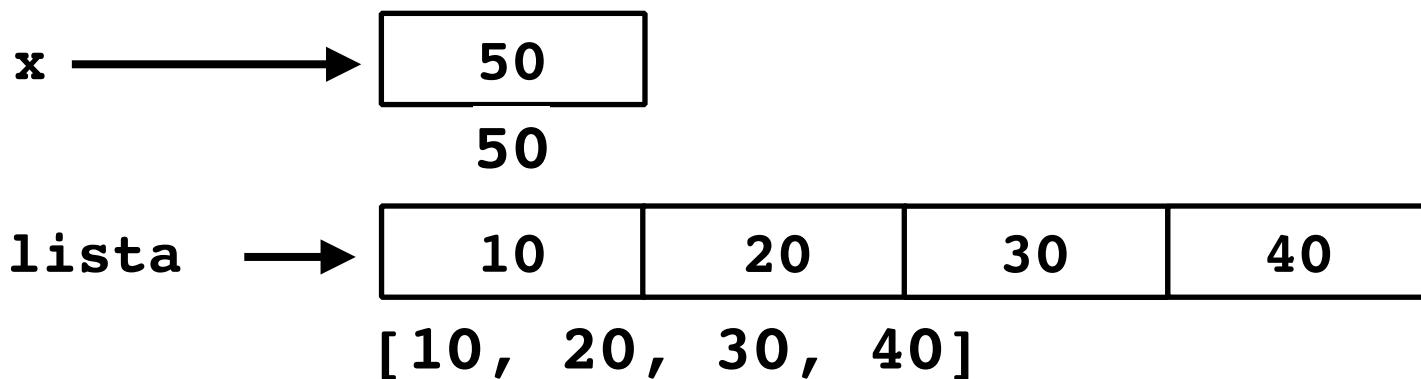
```
lista = [10, 20, 30, 40, 50]
```



REMOVENDO ELEMENTOS DE LISTAS

- Podemos remover elementos do final duma lista com `.pop`
 - Remove o último elemento da lista e retorna-o

```
lista = [10, 20, 30, 40, 50]  
x = lista.pop()
```



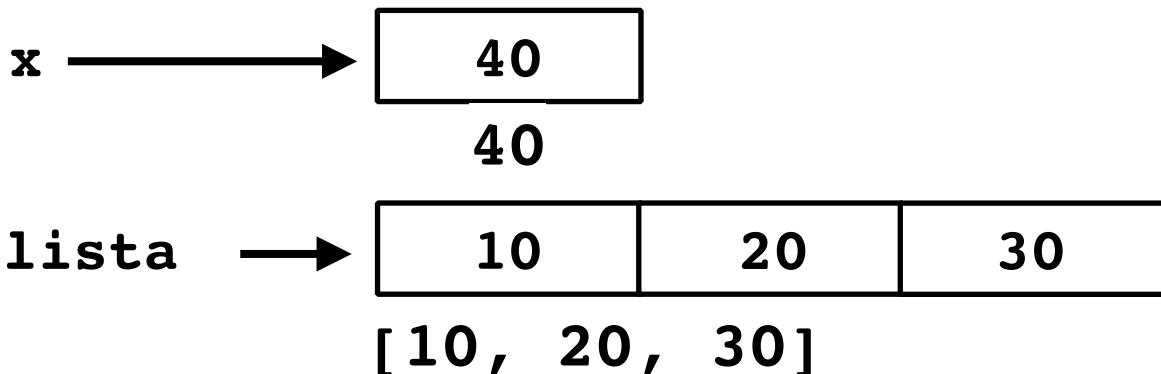
REMOVENDO ELEMENTOS DE LISTAS

- Podemos remover elementos do final duma lista com `.pop`
 - Remove o último elemento da lista e retorna-o

```
lista = [10, 20, 30, 40, 50]
```

```
x = lista.pop()
```

```
x = lista.pop()
```



REMOVENDO ELEMENTOS DE LISTAS

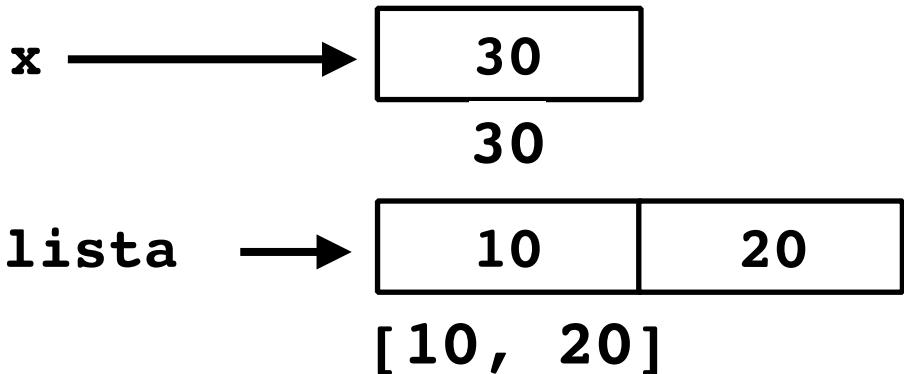
- Podemos remover elementos do final duma lista com `.pop`
 - Remove o último elemento da lista e retorna-o

```
lista = [10, 20, 30, 40, 50]
```

```
x = lista.pop()
```

```
x = lista.pop()
```

```
x = lista.pop()
```

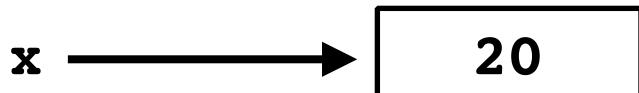


REMOVENDO ELEMENTOS DE LISTAS

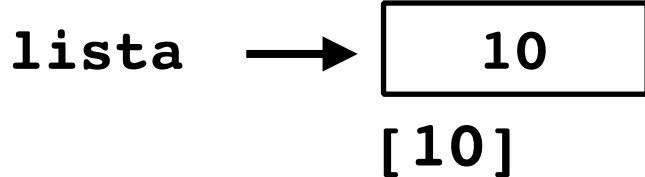
- Podemos remover elementos do final duma lista com `.pop`
 - Remove o último elemento da lista e retorna-o

```
lista = [10, 20, 30, 40, 50]
```

```
x = lista.pop()
```



20

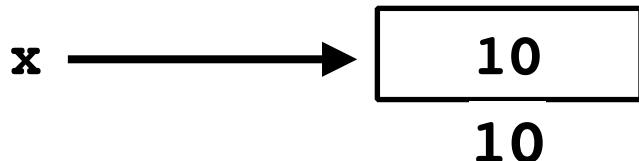


REMOVENDO ELEMENTOS DE LISTAS

- Podemos remover elementos do final duma lista com `.pop`
 - Remove o último elemento da lista e retorna-o

```
lista = [10, 20, 30, 40, 50]
```

```
x = lista.pop()
```



`lista` → *lista vazia*

[]

REMOVENDO ELEMENTOS DE LISTAS

- Podemos remover elementos do final duma lista com `.pop`
 - Remove o último elemento da lista e retorna-o

```
lista = [10, 20, 30, 40, 50]
```

```
x = lista.pop()
```

E se fizermos mais um?

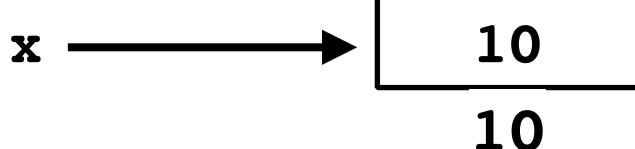
```
x = lista.pop()
```

```
x = lista.pop()
```

```
x = lista.pop()
```

IndexError: pop from empty list

```
x = lista.pop()
```



`lista` → *lista vazia*
[]



MAIS DIVERSÃO COM LISTAS

- Posso ter mais umas listas, sff?

```
lista_num = [1, 2, 3, 4]
```

```
lista_str = ['Maria', 'Pedro', 'João']
```

- Imprimir listas (exemplo na consola)

```
>>> print(lista_num)
```

```
[1, 2, 3, 4]
```

```
>>> print(lista_str)
```

```
['Maria', 'Pedro', 'João']
```

- Verificar se uma lista está vazia (uma lista vazia é como “False”)

```
if lista_num:
```

```
    print("lista_num não está vazia")
```

```
else:
```

```
    print("lista_num está vazia")
```

AINDA MAIS DIVERSÃO COM LISTAS

- Posso ter mais umas listas, sff?

```
lista_num = [1, 2, 3, 4]
```

```
lista_str = ['Maria', 'Pedro', 'João']
```

- Verificar se uma lista contém um elemento:

```
x = 1  
if x in lista_num:  
    # fazer qualquer coisa
```

- Forma geral do teste (avalia para um Booleano)

elemento in lista

- Retorna **True** se *elemento* é um valor na *lista*, **Falso** caso contrário
- Também podemos usar como teste num ciclo **while**

LISTAS: FUNÇÕES SECRETAS! (PARTE 1)

- Função: ***lista.pop(indice)*** # pop aceita parâmetro

- Remove (e retorna) um elemento no índice especificado

```
>>> lista_top = ['a', 'b', 'c', 'd']
>>> lista_top.pop(2)
'c'
>>> lista_top
['a', 'b', 'd']
```

- Função: ***lista.remove(elem)***

- Remove a primeira ocorrência do elemento na lista

```
>>> outra_lista = ['a', 'b', 'b', 'c']
>>> outra_lista.remove('b')
>>> outra_lista
['a', 'b', 'c']
```

- **ValueError** se tentarmos remover um elemento que não está na lista

LISTAS: FUNÇÕES SECRETAS! (PARTE 2)

- Função: **lista.extend(outra_lista)**

- Adiciona todos os elementos da outra lista à lista donde chamamos a função

```
>>> lista1 = [1, 2, 3]
>>> lista2 = [4, 5]
>>> lista1.extend(lista2)
>>> lista1
[1, 2, 3, 4, 5]
```

- **append** não é o mesmo que **extend**

- O append adiciona um elemento, o extend junta uma lista a outra

```
>>> lista1 = [1, 2, 3]
>>> lista2 = [4, 5]
>>> lista1.append(lista2)
[1, 2, 3, [4, 5]]
```

LISTAS: FUNÇÕES SECRETAS! (PARTE 3)

- Usar o operador `+` em listas funciona como o `extend`, mas cria uma nova lista. As listas originais não são alteradas.

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

- Podemos usar o operador `+=` tal como o `extend`

```
>>> lista1 = [1, 2, 3]
>>> lista2 = [4, 5]
>>> lista1 += lista2
>>> lista1
[1, 2, 3, 4, 5]
```

LISTAS: FUNÇÕES SECRETAS! (PARTE 4)

- Função: ***lista.index(elem)***

- Retorna o índice do primeiro elemento na lista igual ao parâmetro elem

```
>>> lista = ['a', 'b', 'b', 'c']
>>> i = lista.index('b')
>>> i
1
```

- **ValueError** se pedirmos um elemento que não está na lista

- Função: ***lista.insert(indice, elem)***

- Insere elem no índice dado. Empurra para trás todos os outros elementos.

```
>>> jedi = ['luke', 'rey', 'obiwan']
>>> jedi.insert(1, 'david')
>>> jedi
['luke', 'david', 'rey', 'obiwan']
```

LISTAS: FUNÇÕES SECRETAS! (PARTE 5)

- Função: **lista.copy()**

- Devolve uma cópia da lista

```
>>> jedi_genuinos = ['luke', 'rey', 'obiwan']
>>> faz_de_conta = jedi_genuinos.copy()
>>> faz_de_conta
['luke', 'rey', 'obiwan']
>>> faz_de_conta.insert(1, 'david')
>>> faz_de_conta
['luke', 'david', 'rey', 'obiwan']
>>> jedi_genuinos
['luke', 'rey', 'obiwan']
```

LISTAS: FUNÇÕES SECRETAS! (PARTE 6)

```
reais = [3.6, 2.9, 8.0, -3.2, 0.5]
```

- Função: **max(lista)**

- Devolve o valor máximo na lista

```
>>> max(reais)
```

```
8.0
```

- Função: **min(lista)**

- Devolve o valor mínimo na lista

```
>>> min(reais)
```

```
-3.2
```

- Função: **sum(lista)**

- Devolve a soma dos valores na lista

```
>>> sum(reais)
```

```
11.8
```

CICLOS NOS ELEMENTOS DE UMA LISTA

```
lista_str = ['Maria', 'Pedro', 'João']
```

- Ciclo **for** utilizando o **range**:

```
for i in range(len(lista_str)):  
    elem = lista_str[i]  
    print(elem)
```

Output:

```
Maria  
Pedro  
João
```

- Podemos usar um novo tipo de ciclo chamado ciclo “for-each”

```
for elem in lista_str:  
    print(elem)
```

- Ambos os ciclos iteram sobre todos os elementos da lista
 - A variável **elem** assume cada um dos valores da lista (por ordem)

CICLO FOR-EACH COM LISTAS

```
lista_str = ['Maria', 'Pedro', 'João']
```

```
for elem in lista_str:  
    # corpo do ciclo  
    # fazer algo com elem
```



Este código é repetido uma vez para cada elemento na lista

- Tal como a variável **i** no ciclo **for** com o **range()**, **elem** é uma variável que é atualizada a cada iteração do ciclo.
- Os elementos da lista são todos atribuídos a **elem**, um de cada vez (ou seja, um em cada iteração)

CICLOS NOS ELEMENTOS DE UMA LISTA

- Forma geral de um ciclo for-each:

for elemento in coleção:

fazer alguma coisa com elemento

- **elemento** pode ser qualquer variável que queremos usar para nos referirmos aos itens da **coleção**

- Em cada iteração do ciclo, **elemento** assumirá o valor do próximo item (por ordem) na **coleção**

- De novo, exemplo:

for elem in lista_str:

print(elem)

- As listas são coleções
 - Vamos ver mais tarde outros tipos de coleção

QUANDO PASSADOS COMO PARÂMETROS

Tipos que são “imutáveis”

`int`

`float`

`bool`

`string`

Tipos que são “mutáveis”

`list`

(vamos ver outros mais tarde)

- Quando atribuímos novo valor a variável, estamos a atribuir etiqueta (nome) a um novo valor (nova mala).

- Com parâmetros, o valor original da variável que passamos **não** é alterado quando a função termina.

- Quando alteramos a variável “no lugar” (conteúdo), a etiqueta não muda, mas o valor dentro da mala muda.

- Com parâmetros, o valor original da variável que passamos **é** alterado quando a função termina.

LISTAS COMO PARÂMETROS I

- Quando passamos uma lista como parâmetro estamos a passar uma referência para a própria lista
 - É como obter um URL para a lista (passagem por referência)
 - Na função, alterações aos valores na lista **persistem** após a função terminar.

```
def soma_cinco(lista_num)
    for i in range(len(lista_num)):
        lista_num[i] += 5

def main()
    valores = [5, 6, 7, 8]
    soma_cinco(valores)
    print(valores)
```

Output: [10, 11, 12, 13]

LISTAS COMO PARÂMETROS II

- Mas, cuidado se criarmos uma nova lista numa função
 - Criar uma nova lista significa que já não estamos a lidar com a lista passada como parâmetro
 - É como se o URL que estamos a usar passasse a apontar para uma página diferente. (Atribuímos a etiqueta a um novo valor dentro da função)
 - A partir desse ponto já não estamos a alterar o parâmetro passado

```
def cria_nova_lista(lista_num)
    lista_num.append(9)
    lista_num = [1, 2, 3]

def main()
    valores = [5, 6, 7, 8]
    cria_nova_lista(valores)
    print(valores)
```

Output: [5, 6, 7, 8, 9]

NOTA SOBRE CICLOS E LISTAS

- Ciclo for usando range:

```
for i in range(len(lista)):  
    lista[i] += 1 # Modifica a lista no lugar
```

- Ciclo for-each:

```
for elem in lista: # Modifica a variável local  
    elem += 1        # elem. Se elem é de um tipo  
                      # imutável, não altera a lista!
```

- Normalmente usamos um ciclo for com range quando queremos *modificar* os elementos da lista (quando os elementos são de *tipos imutáveis*)
- Normalmente usamos um ciclo for-each quando *não estamos a modificar* os elementos da lista ou quando os elementos são de *tipos mutáveis*

Exemplo completo:
averagescores.py

OBJETIVOS DE APRENDIZAGEM

1. Aprender listas em Python
2. Escrever código que usa listas
3. Perceber como é que as listas são passadas como parâmetros

PARTE I





OBJETIVOS DE APRENDIZAGEM

1. Aprender mais sobre listas
2. Aprender outras coleções: Dicionários



Slices



O QUE SÃO SLICES?

- Podemos cortar listas em “slices”
 - Slices são apenas sub-porções de listas
 - Slices também são listas
 - Slicing cria uma **nova** lista
- Exemplo



```
lista = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
lista → [ 'a' | 'b' | 'c' | 'd' | 'e' | 'f' ]  
        0   1   2   3   4   5
```

```
slice = lista[2:4]
```

```
slice → [ 'c' | 'd' ]  
        0   1
```

O QUE SÃO SLICES?

- Podemos cortar listas em “slices”
 - Slices são apenas sub-porções de listas
 - Slices também são listas
 - Slicing cria uma **nova** lista
- Exemplo



```
lista = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
lista → [ 'a' | 'b' | 'c' | 'd' | 'e' | 'f' ]  
        0   1   2   3   4   5
```

```
slice = lista[2:4]
```

```
slice → [ 'x' | 'd' ]  
        0   1
```

```
slice[0] = 'x'
```

FORMA GERAL DO SLICE

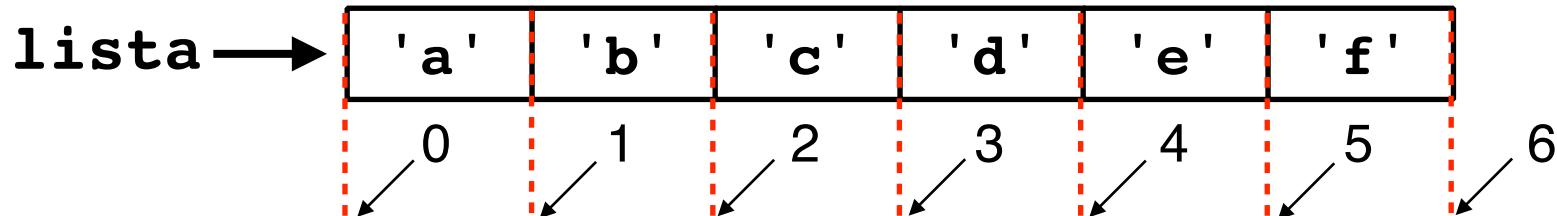
- Forma geral para obter uma slice

lista[início:fim]

- Produz uma nova lista com os elementos de **lista** a começar no índice **início** até (mas não incluindo) o índice **fim**

- Exemplo

```
lista = ['a', 'b', 'c', 'd', 'e', 'f']
```



lista[2:4] → ['c', 'd']

lista[1:6] → ['b', 'c', 'd', 'e', 'f']

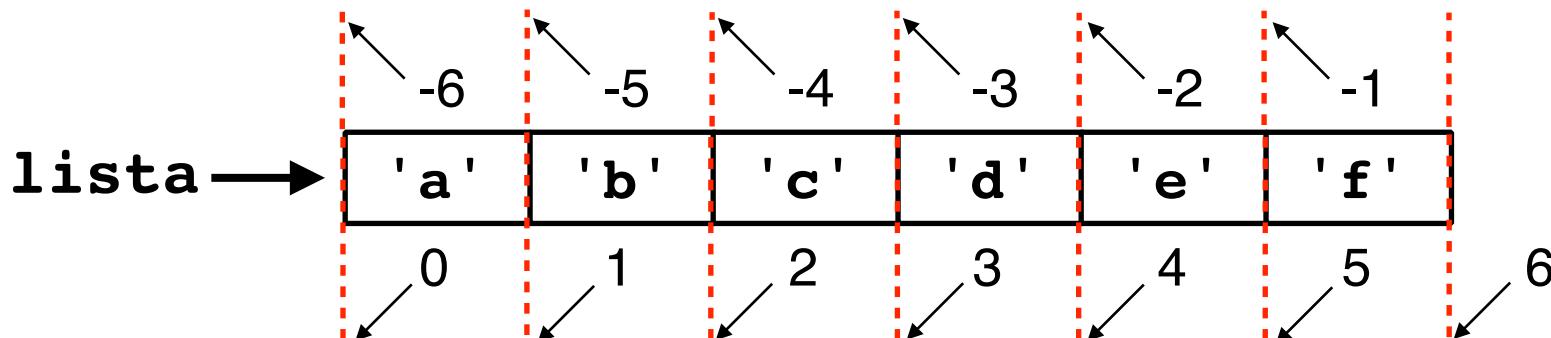
lista[0:3] → ['a', 'b', 'c']

MAIS UMA FATIA!

- Forma geral para obter uma slice

lista[início:fim]

- Se **início** estiver em falta, é usado por omissão 0 no seu lugar
- Se **fim** estiver em falta, é usado por omissão `len(lista)` no seu lugar
- Também podemos usar índices negativos para **início/fim**



`lista[2:-2]` → `['c', 'd']`

`lista[-2:]` → `['e', 'f']`

`lista[:-1]` → `['a', 'b', 'c', 'd', 'e']`

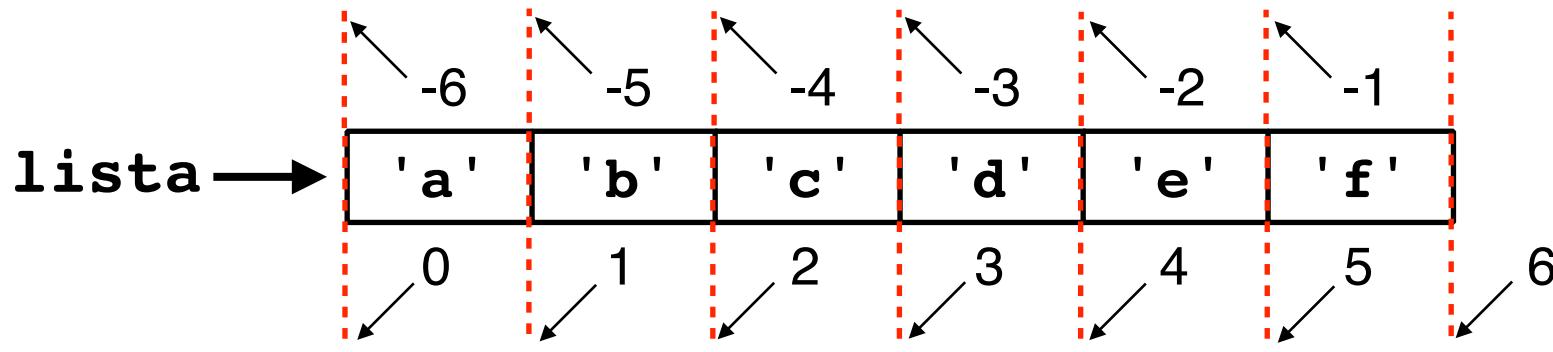
`lista[:]` → `['a', 'b', 'c', 'd', 'e', 'f']`

SLICES AVANÇADAS

- Forma geral para obter uma slice, com um salto

lista[início:fim:salto]

- Retirar uma slice de **início** até **fim**, progredindo com **salto**
- **salto** pode ser negativo (andar para trás, logo o **início/fim** são trocados)



`lista[1:5:2] → ['b', 'd']`

`lista[::-2] → ['a', 'c', 'e']`

`lista[4:1:-1] → ['e', 'd', 'c']`

`lista[1:4:-1] → []`

`lista[::-1] → ['f', 'e', 'd', 'c', 'b', 'a']`

CICLOS E SLICES

- Podemos usar um ciclo for-each com o slice
 - O slice é só uma lista, por isso podemos usá-lo tal como uma lista
 - Lembremo-nos outra vez dos ciclos com listas:

```
for i in range(len(lista)):  
    # fazer algo com lista[i]
```

```
for elem in lista:  
    # fazer algo com elem
```

CICLOS E SLICES

- Podemos usar um ciclo for-each com o slice
 - O slice é só uma lista, por isso podemos usá-lo tal como uma lista
 - Agora, para ciclos com **slices** (nota: **salto** é opcional):

```
for i in range(inicio, fim, salto):  
    # fazer algo com lista[i]
```

```
for elem in lista[inicio: fim: salto]:  
    # fazer algo com elem
```

- Não esquecer: se **salto** for negativo, então **início** deve ser maior que o **fim**

APAGAR COM SLICES

- Podemos apagar elementos de uma lista com o comando **del**
- Exemplo:

```
>>> lista_num = [50, 30, 40, 60, 90, 80]
>>> del lista_num[1]
>>> lista_num
[50, 40, 60, 90, 80]
```

- Podemos usar o **del** com a notação slice:

```
>>> lista_num = [50, 30, 40, 60, 90, 80]
>>> del lista_num[1:4]
>>> lista_num
[50, 90, 80]
```

ALTERAR UMA LISTA NO LUGAR

- O Python oferece algumas operações sobre uma lista inteira
 - Estas funções modificam a lista no lugar (sem criar uma lista nova)
- Função: ***lista.reverse()***

- Reverte a ordem dos elementos da lista

```
>>> super_lista = [6, 3, 12, 4]
>>> super_lista.reverse()
>>> super_lista
[4, 12, 3, 6]
```

- Função: ***lista.sort()***
 - Coloca os elementos da lista por ordem ascendente

```
>>> super_lista = [6, 3, 12, 4]
>>> super_lista.sort()
>>> super_lista
[3, 4, 6, 12]
```

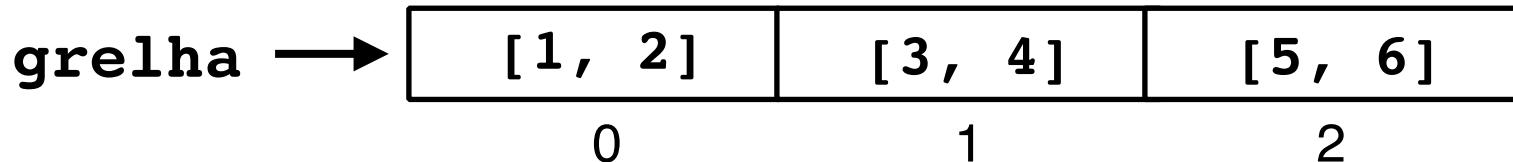
Listas Bidimensionais

LISTA 2D

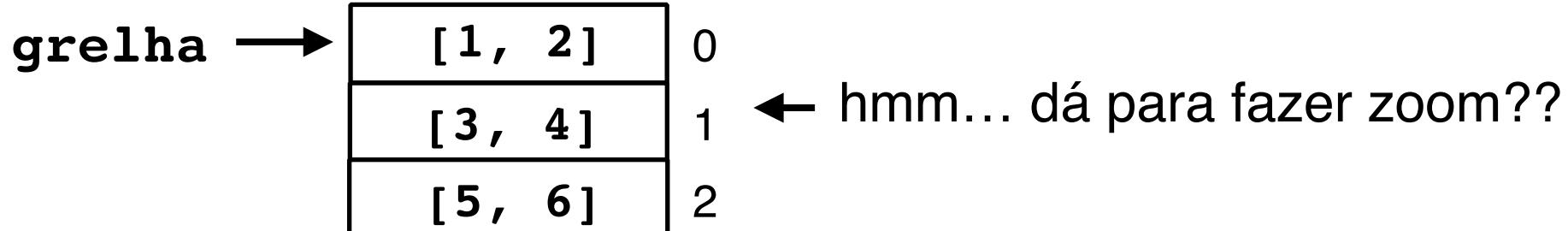
- Podemos ter uma lista de listas!
 - Cada elemento da lista de fora é apenas outra lista
 - Podemos pensar nisso como uma tabela ou grelha

- Exemplo

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



- Pode ser mais fácil pensar desta forma:



LISTA 2D

grelha →

[1, 2]	0
[3, 4]	1
[5, 6]	2

hmm... dá para fazer zoom??

grelha →

1	2	0
0	1	
3	4	1
0	1	
5	6	2
0	1	

LISTA 2D

grelha →

1	2
0	1
3	4
0	1
5	6
0	1

0

1

2

grelha[0][0]	grelha[0][1]
1	2
grelha[1][0]	grelha[1][1]
3	4
grelha[2][0]	grelha[2][1]
5	6

- Para aceder aos elementos, especificar o índice na lista “de fora”, depois o índice na lista “de dentro”

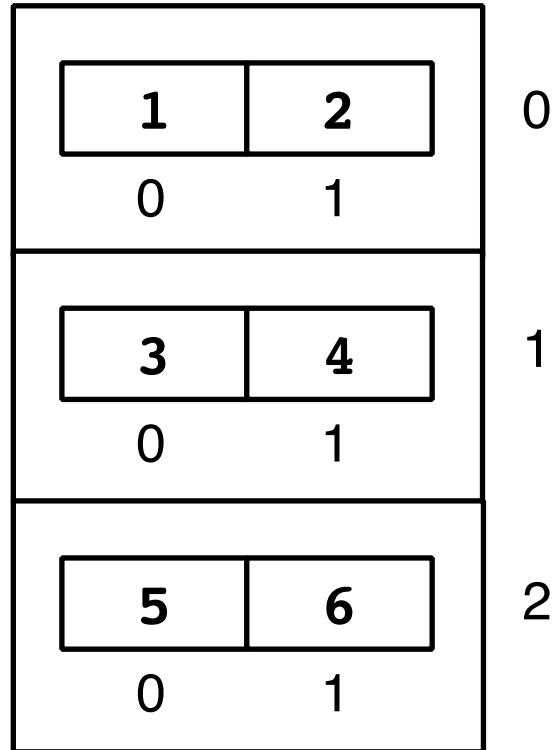
grelha[0][0] → 1

grelha[1][0] → 3

grelha[2][1] → 6

LISTA 2D

grelha →



0

1

2

- Então e se eu especificar só um índice?

grelha[0] → [1, 2]

grelha[1] → [3, 4]

grelha[2] → [5, 6]

- Grelha é só uma lista de listas

- Os elementos da lista de fora são só listas normais

DIVERSÃO COM LISTAS

- As listas de dentro têm de ter o mesmo tamanho?

- Não! Mas cuidado se não forem

desigual = [[1, 2, 3], [4], [5, 6]]

desigual[0] → [1, 2, 3]

desigual[1] → [4]

desigual[2] → [5, 6]

- Posso ter mais de duas dimensões?

- Claro! As que forem precisas!

cubo = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]

cubo[0] → [[1, 2], [3, 4]]

cubo[0][1] → [3, 4]

cubo[0][1][0] → 3

TROCA DE ELEMENTOS NUMA GRELHA

```
def troca(grelha, linha1, col1, linha2, col2):  
    temp = grelha[linha1, col1]  
    grelha[linha1, col1] = grelha[linha2, col2]  
    grelha[linha2, col2] = temp  
  
def main():  
    minha_grelha=[[10, 20, 30], [40, 50, 60]]  
    troca(minha_grelha, 0, 1, 1, 2)  
    print(minha_grelha)
```

Output: [[10, 60, 30], [40, 50, 20]]

ITERAR NUMA LISTA DE LISTAS

```
def main():
    grelha = [[10, 20], [40], [70, 80, 100]]
    linhas = len(grelha)
    for i in range(linhas):
        cols = len(grelha[i])
        for j in range(cols):
            print("grelha[" + str(i) + "][" + str(j)
                  + "] = " + str(grelha[i][j]))
```

Output:

```
grelha[0][0] = 10
grelha[0][1] = 20
grelha[1][0] = 40
grelha[2][0] = 70
grelha[2][1] = 80
grelha[2][2] = 100
```

MAIS SIMPLES COM GRELHA VERDADEIRA

```
def main():
    grelha = [[1, 2], [10, 11], [20, 21]]
    linhas = len(grelha)
    cols = len(grelha[0])
    for i in range(linhas):
        for j in range(cols):
            print("grelha[" + str(i) + "][" + str(j)
                  + "] = " + str(grelha[i][j]))
```

Output:

```
grelha[0][0] = 1
grelha[0][1] = 2
grelha[1][0] = 10
grelha[1][1] = 11
grelha[2][0] = 20
grelha[2][1] = 21
```

FOR-EACH COM LISTA 2D

```
def main():
    grelha = [[10, 20], [40], [70, 80, 100]]
    for linha in grelha:
        for elem in linha:
            print(elem)
```

Output:

```
10
20
40
70
80
100
```

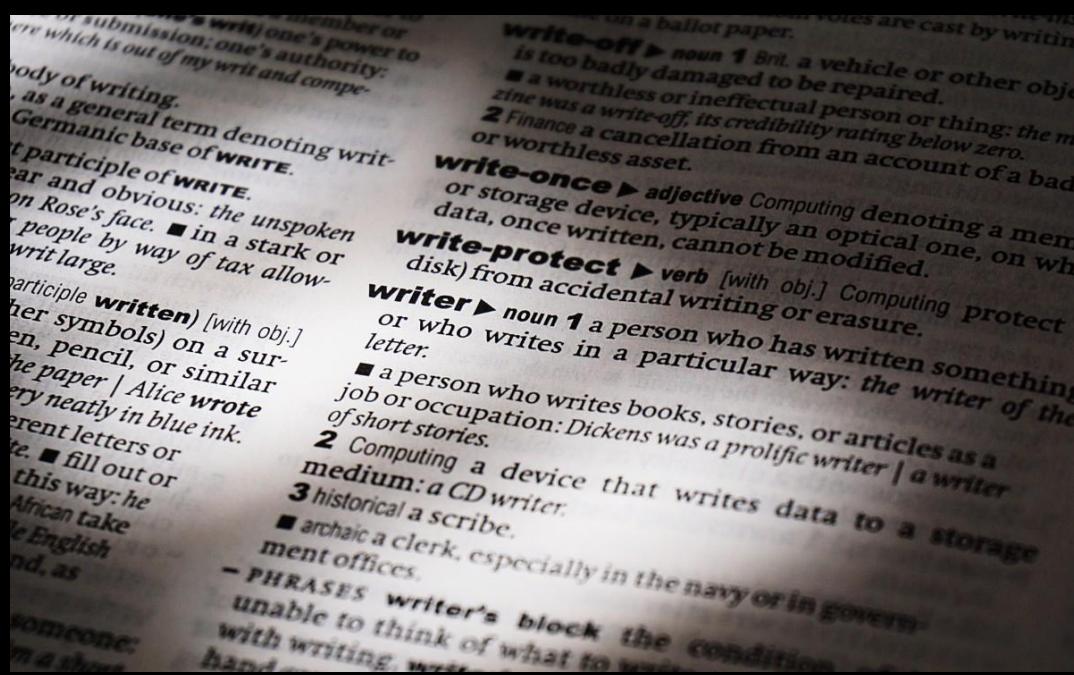
CRIAR UMA LISTA 2D

```
def criar_grelha(linhas, cols, valor):  
    grelha = []                      # Criar grelha vazia  
    for y in range(linhas):          # Fazer linhas uma a uma  
        linha = []  
        for x in range(cols):  
            linha.append(valor)  
  
        grelha.append(linha)  
  
    return grelha
```

Consola:

```
>>> criar_grelha(2, 4, 1)  
[[1, 1, 1, 1], [1, 1, 1, 1]]  
>>> criar_grelha(3, 2, 5)  
[[5, 5], [5, 5], [5, 5]]
```


Dicionários



O QUE SÃO DICIONÁRIOS?

- Os dicionários associam uma **chave** (key) a um **valor** (value)
 - Chave é um identificador *único*
 - Valor é algo que associamos àquela chave
- Exemplos reais
 - Lista telefónica
 - Chaves: nomes
 - Valores: números de telefone
 - Dicionário
 - Chaves: palavras
 - Valores: definições das palavras
 - Divisão Académica
 - Chaves: números de estudante
 - Valores: processos dos estudantes

DICIONÁRIOS EM PYTHON

- Criar dicionários

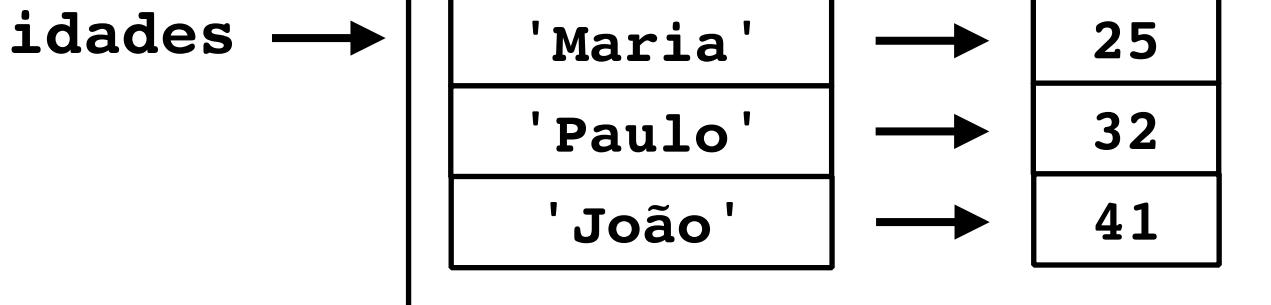
- Dicionários começam e acabam com parêntesis
- Pares chave:valor separados por dois pontos (:)
- Cada par separado por vírgula

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```

```
quadrados = {2: 4, 3: 9, 4: 16, 5: 25}
```

```
telefones = {'Ana': '913384459', 'Rui': '962938845'}
```

```
dic_vazio = {}
```



ACEDER AOS ELEMENTOS DE UM DICIONÁRIO

- Consideremos o seguinte dicionário:

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```



- Usamos a chave para aceder ao valor associado

```
idades[ 'Maria' ] é 25
```

```
idades[ 'João' ] é 41
```

ACEDER AOS ELEMENTOS DE UM DICIONÁRIO

- Consideremos o seguinte dicionário:

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```



- Usamos a chave para aceder ao valor associado

```
idades[ 'Maria' ] é 25
```

```
idades[ 'João' ] é 41
```

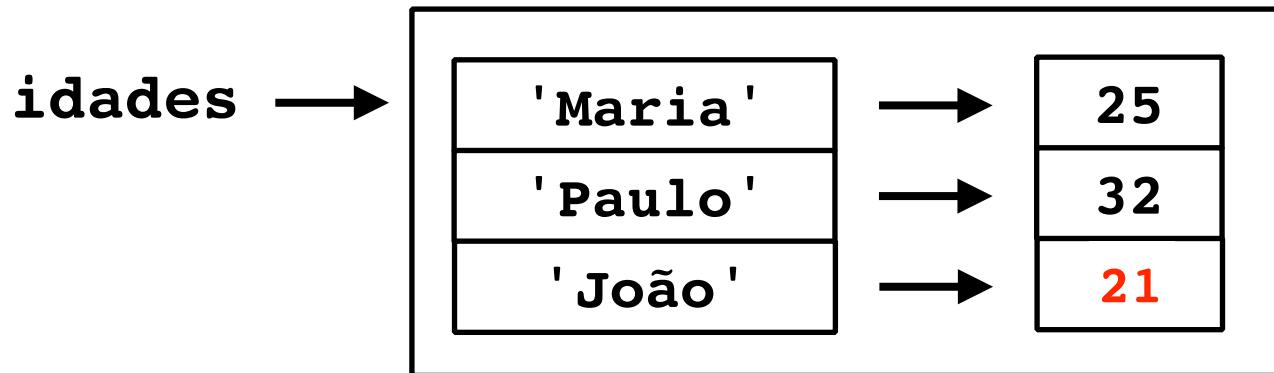
- Podemos atribuir valores como uma variável normal:

```
idades[ 'João' ] = 18
```

ACEDER AOS ELEMENTOS DE UM DICIONÁRIO

- Consideremos o seguinte dicionário:

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```



- Usamos a chave para aceder ao valor associado

```
idades[ 'Maria' ] é 25
```

```
idades[ 'João' ] é 41
```

- Podemos atribuir valores como uma variável normal:

```
idades[ 'João' ] = 18
```

```
idades[ 'João' ] += 3
```

ACEDER AOS ELEMENTOS DE UM DICIONÁRIO

- Consideremos o seguinte dicionário:

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```



- Good times e bad times com aceder aos pares:

```
>>> idade_paulo = idades['Paulo']
```

```
>>> idade_paulo
```

```
32
```

```
>>> idade_pai_natal = idades['Pai Natal']
```

```
KeyError: 'Pai Natal'
```

ACEDER AOS ELEMENTOS DE UM DICIONÁRIO

- Consideremos o seguinte dicionário:

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```



- Verificar se pertence ao dicionário:

```
>>> 'Paulo' in idades
```

```
True
```

```
>>> 'Pai Natal' not in idades
```

```
True
```

ADICIONAR ELEMENTOS AO DICIONÁRIO

- Podemos adicionar pares a um dicionário:

```
telefones = {}
```

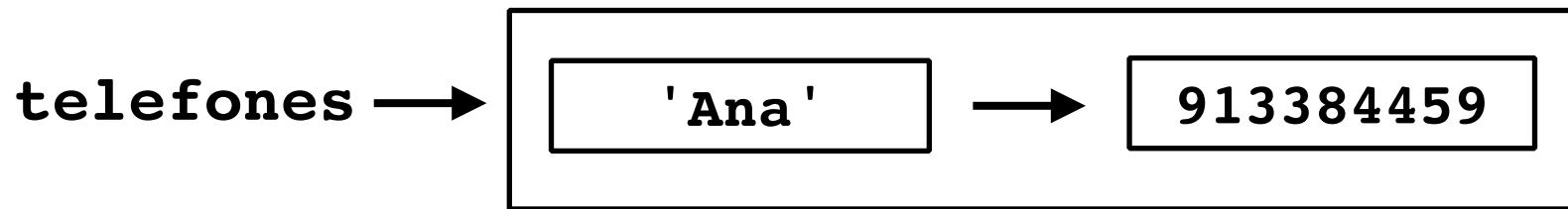
```
telefones → 'Ana' → 913384459
```

```
telefones['Ana'] = '913384459'
```

ADICIONAR ELEMENTOS AO DICIONÁRIO

- Podemos adicionar pares a um dicionário:

```
telefones = {}
```



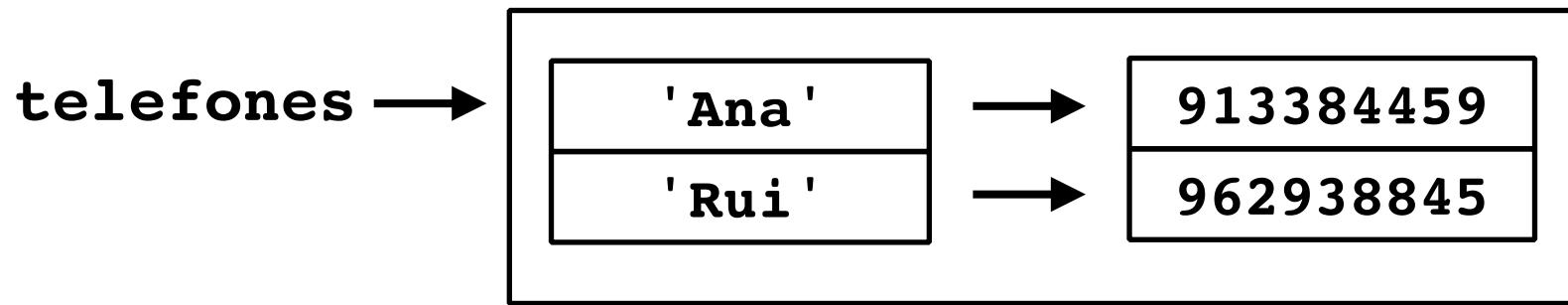
```
telefones['Ana'] = '913384459'
```

```
telefones['Rui'] = '962938845'
```

ADICIONAR ELEMENTOS AO DICIONÁRIO

- Podemos adicionar pares a um dicionário:

```
telefones = {}
```



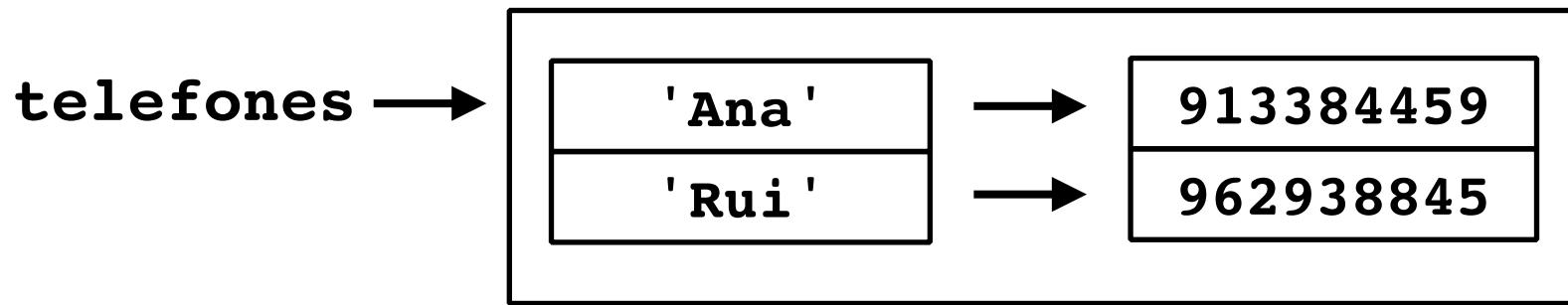
```
telefones['Ana'] = '913384459'
```

```
telefones['Rui'] = '962938845'
```

ADICIONAR ELEMENTOS AO DICIONÁRIO

- Podemos adicionar pares a um dicionário:

```
telefones = {}
```



```
telefones['Ana'] = '913384459'
```

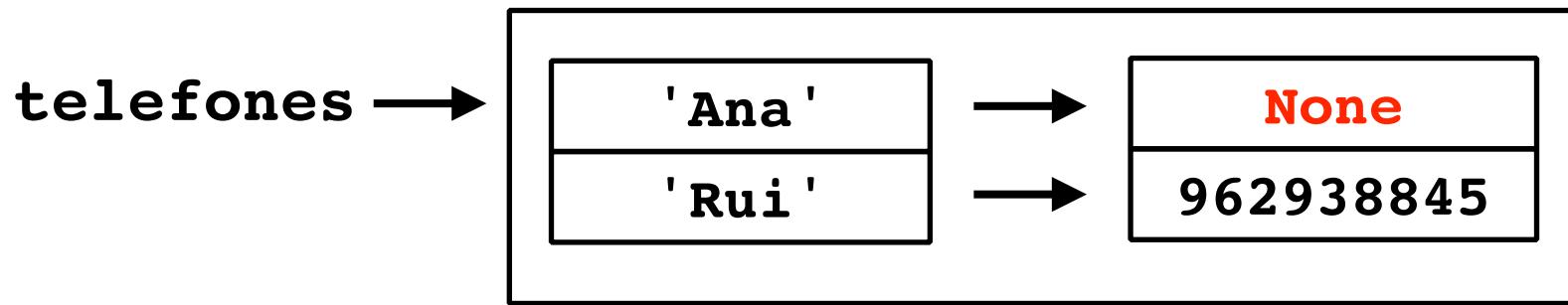
```
telefones['Rui'] = '962938845'
```

```
telefones['Ana'] = None
```

ADICIONAR ELEMENTOS AO DICIONÁRIO

- Podemos adicionar pares a um dicionário:

```
telefones = {}
```



```
telefones[ 'Ana' ] = '913384459'
```

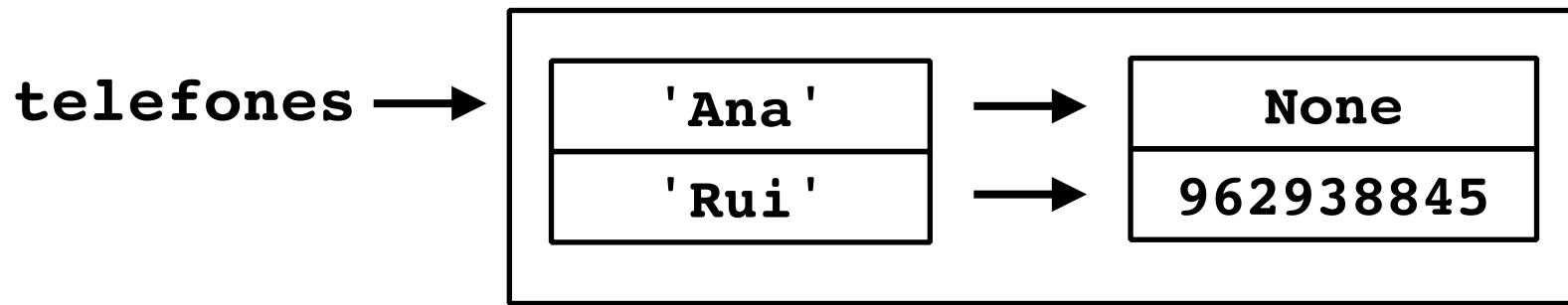
```
telefones[ 'Rui' ] = '962938845'
```

```
telefones[ 'Ana' ] = None
```

ADICIONAR ELEMENTOS AO DICIONÁRIO

- Podemos adicionar pares a um dicionário:

```
telefones = {}
```



```
telefones['Ana'] = '913384459'
```

```
telefones['Rui'] = '962938845'
```

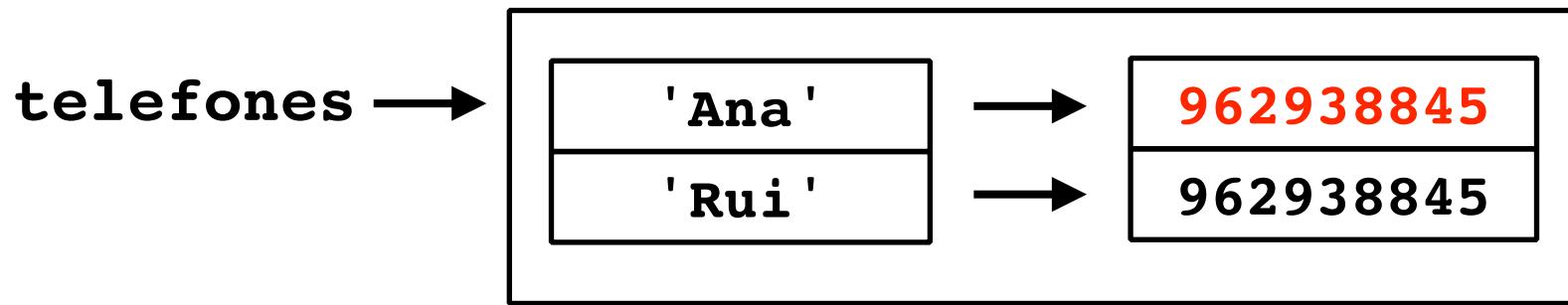
```
telefones['Ana'] = None
```

```
telefones['Ana'] = '962938845'
```

ADICIONAR ELEMENTOS AO DICIONÁRIO

- Podemos adicionar pares a um dicionário:

```
telefones = {}
```



```
telefones[ 'Ana' ] = '913384459'
```

```
telefones[ 'Rui' ] = '962938845'
```

```
telefones[ 'Ana' ] = None
```

```
telefones[ 'Ana' ] = '962938845'
```

NOTAS SOBRE CHAVES/VALORES

- As chaves têm de ser tipos imutáveis
 - Ex: int, float, string
 - As chaves não podem ser alteradas no lugar
 - Se quisermos mudar uma chave, necessário remover o par chave/valor do dicionário e de seguida adicionar o par chave/valor com a nova chave.
- Os valores podem ser tipos mutáveis ou imutáveis
 - Ex: int, float, string, listas, dicionários
 - Os valores podem ser alterados no lugar
- Os dicionários são mutáveis
 - Alterações feitas a um dicionário dentro de uma função persistem depois da função terminar

ALTERAR DICIONÁRIO NUMA FUNÇÃO

```
def faz_anos(dic, nome):
    print = ("Feliz aniversário, " + nome + "!")
    dic[nome] += 1

def main():
    idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
    print(idades)
    faz_anos(idades, 'Maria')
    print(idades)
    faz_anos(idades, 'João')
    print(idades)
```

Terminal:

```
{'Maria': 25, 'Paulo': 32, 'João': 41}
Feliz aniversário, Maria!
{'Maria': 26, 'Paulo': 32, 'João': 41}
Feliz aniversário, João!
{'Maria': 26, 'Paulo': 32, 'João': 42}
```

FESTA DO DICIONÁRIO! (PARTE 1)

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```

- Função: **dic.get(chave)**

- Devolve o valor associado à chave no dicionário. Retorna **None** se a chave não existir.

```
>>> print(idades.get('Maria'))
```

```
25
```

```
>>> print(idades.get('Pai Natal'))
```

```
None
```

- Função: **dic.get(chave, valor_omissão)**

- Devolve o valor associado à chave no dicionário. Retorna **valor_omissão** se a chave não existir.

```
>>> print(idades.get('Maria', 100))
```

```
25
```

```
>>> print(idades.get('Pai Natal', 100))
```

```
100
```

FESTA DO DICIONÁRIO! (PARTE 2)

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```

- Função: **dic.keys()**

- Devolve uma coleção com as chaves no dicionário.
- Podemos usar para iterar sobre todas as chaves do dicionário

```
for chave in idades.keys():
    print(chave + " -> " + str(idades[chave]))
```

Terminal:

```
Maria -> 25
Paulo -> 32
João -> 41
```

- Podemos transformar **keys()** numa lista, com a função **list**

```
>>> list(idades.keys())
['Maria', 'Paulo', 'João']
```

FESTA DO DICIONÁRIO! (PARTE 3)

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```

- Também podemos iterar pelo dicionário com um ciclo for-each usando apenas o nome do dicionário:

```
for chave in idades:  
    print(chave + " -> " + str(idades[chave]))
```

Terminal:

```
Maria -> 25  
Paulo -> 32  
João -> 41
```

FESTA DO DICIONÁRIO! (PARTE 4)

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```

- Função: **dic.values()**

- Devolve uma coleção com os valores no dicionário.
- Podemos usar para iterar sobre todas os valores do dicionário

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

Terminal:

```
25
32
41
```

- Podemos transformar **values()** numa lista, com a função **list**
- ```
>>> list(idades.values())
[25, 32, 41]
```

# FESTA DO DICIONÁRIO! (PARTE 5)

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```

- Função: **dic.pop(chave)**

- Remove o par chave/valor com a chave fornecida. Devolve o valor desse par chave/valor

```
>>> idades
{'Maria': 25, 'Paulo': 32, 'João': 41}
>>> idades.pop('João')
41
>>> idades
{'Maria': 25, 'Paulo': 32}
```

- Função: **dic.clear()**

- Remove todos os pares chave/valor do dicionário

```
>>> idades.clear()
>>> idades
{}
```

# FESTA DO DICIONÁRIO! (PARTE 6)

```
idades = {'Maria': 25, 'Paulo': 32, 'João': 41}
```

- Função: **len** (dic)

- Retorna o número de pares chave/valor no dicionário

```
>>> idades
```

```
{'Maria': 25, 'Paulo': 32, 'João': 41}
```

```
>>> len(idades)
```

```
3
```

- Função: **del** (dic [chave])

- Remove pares chave/valor do dicionário
- Semelhante ao **pop**, mas não devolve nada.

```
>>> idades
```

```
{'Maria': 25, 'Paulo': 32, 'João': 41}
```

```
>>> del(idades['João'])
```

```
>>> idades
```

```
{'Maria': 25, 'Paulo': 32}
```



# OBJETIVOS DE APRENDIZAGEM

1. Aprender mais sobre listas
2. Aprender outras coleções: Dicionários





# REFERÊNCIAS

- Slides adaptados de Piech and Sahami, CS106A, Stanford University
- PyCharm - Jetbrains, <https://www.jetbrains.com/pycharm/>

FP@moodle

<https://moodle.ips.pt/2223/course/view.php?id=1100>