MAC 115 — Introdução à Ciência da Computação

Aula 18

Nelson Lago

IF noturno – 2023

Previously on MAC 115...

Repetições com coleções

```
primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
n = 0
while n < len(primos):
    print("O número", primos[n], "é primo")
    n += 1
```

Repetições com coleções

```
primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
n = 0
while n < len(primos):
    print("O número", primos[n], "é primo")
    n += 1
```

```
primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
for n in range(len(primos)):
    print("O número", primos[n], "é primo")
```

Repetições com coleções

```
primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
n = 0
while n < len(primos):
    print("O número", primos[n], "é primo")
    n += 1
```

```
primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
for n in range(len(primos)):
    print("O número", primos[n], "é primo")
```

```
primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
for p in primos:
    print("O número", p, "é primo")
```

Tipos de repetição

- `while` pode ser usado para todos os tipos de repetição que vimos

Tipos de repetição

- `while` pode ser usado para todos os tipos de repetição que vimos
- MAS...

Tipos de repetição

- `while` pode ser usado para todos os tipos de repetição que vimos
- MAS...
- Tipos de laços diferentes "combinam melhor" com sintaxes diferentes

Tipos de repetição

- Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:

Tipos de repetição

- Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:
  - `while not achei:`

Tipos de repetição

- Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:
  - `while not achei:`
  - `while encontrados < 10:`

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`
- **Quando queremos manipular os elementos de uma coleção, `for` é uma boa escolha:**

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`
- **Quando queremos manipular os elementos de uma coleção, `for` é uma boa escolha:**
  - `for p in primos:`

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`
- **Quando queremos manipular os elementos de uma coleção, `for` é uma boa escolha:**
  - `for p in primos:`
  - `for canção in canções:`

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`
- **Quando queremos manipular os elementos de uma coleção, `for` é uma boa escolha:**
  - `for p in primos:`
  - `for canção in canções:`
- **Quando queremos manipular uma lista pré-definida de números *ou* os *índices* dos elementos de uma coleção, `for`...`range()` é uma boa escolha:**

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`
- **Quando queremos manipular os elementos de uma coleção, `for` é uma boa escolha:**
  - `for p in primos:`
  - `for canção in canções:`
- **Quando queremos manipular uma lista pré-definida de números *ou* os *índices* dos elementos de uma coleção, `for`...`range()` é uma boa escolha:**
  - `for n in range(10):`

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`
- **Quando queremos manipular os elementos de uma coleção, `for` é uma boa escolha:**
  - `for p in primos:`
  - `for canção in canções:`
- **Quando queremos manipular uma lista pré-definida de números *ou* os *índices* dos elementos de uma coleção, `for`...`range()` é uma boa escolha:**
  - `for n in range(10):`
  - `for n in range(len(minha_lista)):`

Tipos de repetição

- **Quando a quantidade de repetições é desconhecida, `while` é uma boa escolha:**
  - `while not achei:`
  - `while encontrados < 10:`
  - `while usuárioQuerJogar:`
- **Quando queremos manipular os elementos de uma coleção, `for` é uma boa escolha:**
  - `for p in primos:`
  - `for canção in canções:`
- **Quando queremos manipular uma lista pré-definida de números *ou* os *índices* dos elementos de uma coleção, `for`...`range()` é uma boa escolha:**
  - `for n in range(10):`
  - `for n in range(len(minha_lista)):`
  - `for i in range(início,final):`

A função range()

```
range(início, final, passo)
```

A função `range()`

```
range(início, final, passo)
```

- **O início pode ser omitido; o padrão é zero**

A função `range()`

```
range(início, final, passo)
```

- **O início pode ser omitido; o padrão é zero**
- **O passo pode ser omitido; o padrão é um**

A função `range()`

```
range(início, final, passo)
```

- O início pode ser omitido; o padrão é zero
- O passo pode ser omitido; o padrão é um
- Se há dois parâmetros, eles são `início` e `final`

A função `range()`

```
range(início, final, passo)
```

- **O início pode ser omitido; o padrão é zero**
- **O passo pode ser omitido; o padrão é um**
- **Se há dois parâmetros, eles são `início` e `final`**

O intervalo é sempre

fechado no início e aberto no final

A função `range()`

```
range(início, final, passo)
```

- **O início pode ser omitido; o padrão é zero**
- **O passo pode ser omitido; o padrão é um**
- **Se há dois parâmetros, eles são `início` e `final`**

O intervalo é sempre

fechado no início e aberto no final

primeiro = início

A função `range()`

```
range(início, final, passo)
```

- **O início pode ser omitido; o padrão é zero**
- **O passo pode ser omitido; o padrão é um**
- **Se há dois parâmetros, eles são `início` e `final`**

O intervalo é sempre

fechado no início e aberto no final

$$\text{primeiro} = \text{início}$$

$$|\text{último}| < |\text{final}|$$

A função `range()`

```
range(início, final, passo)
```

- **O início pode ser omitido; o padrão é zero**
- **O passo pode ser omitido; o padrão é um**
- **Se há dois parâmetros, eles são `início` e `final`**

O intervalo é sempre

fechado no início e aberto no final

primeiro = início

|último| < |final|

Brincando com listas

Brincando com listas

```
lista + outra_lista
```

Brincando com listas

```
lista + outra_lista
lista * 5
```

Brincando com listas

```
lista + outra_lista
lista * 5
lista[2:5]
```

Brincando com listas

```
lista + outra_lista
lista * 5
lista[2:5]

lista.append("blah")
```

Brincando com listas

```
lista + outra_lista
lista * 5
lista[2:5]

lista.append("blah")
lista.extend(["blah", "blá"])
```

Brincando com listas

```
lista + outra_lista
lista * 5
lista[2:5]

lista.append("blah")
lista.extend(["blah", "blá"])
lista.sort()
```

Brincando com listas

```
lista + outra_lista
lista * 5
lista[2:5]

lista.append("blah")
lista.extend(["blah", "blá"])
lista.sort()
del lista[2:5]
```

Brincando com listas

```
lista + outra_lista
lista * 5
lista[2:5]

lista.append("blah")
lista.extend(["blah", "blá"])
lista.sort()
del lista[2:5]

if elemento in lista:
```

Nomes e memória

- **Em python, só existe quem tem nome:**

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

- **O número `1` *continua existindo* na memória após a terceira linha ser executada**

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

- **O número `1` *continua existindo* na memória após a terceira linha ser executada**
  - Mas o que acontece na segunda linha?!?!?

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

- **O número `1` *continua existindo* na memória após a terceira linha ser executada**
  - Mas o que acontece na segunda linha?!?!?
    - » *Temos duas "cópias" do número `1` na memória?*

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

- **O número `1` *continua existindo* na memória após a terceira linha ser executada**
  - Mas o que acontece na segunda linha?!?!?
    - » *Temos duas "cópias" do número `1` na memória?*
- **DEPENDE**

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

- **O número `1` *continua existindo* na memória após a terceira linha ser executada**
  - Mas o que acontece na segunda linha?!?!?
    - » *Temos duas "cópias" do número `1` na memória?*
- **DEPENDE**
  - Em python, não!

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

- **O número `1` *continua existindo* na memória após a terceira linha ser executada**
  - Mas o que acontece na segunda linha?!?!?
    - » *Temos duas "cópias" do número `1` na memória?*
- **DEPENDE**
  - Em python, não!
    - » *O número `1` simplesmente tem dois nomes (`x` e `y`)*

Nomes e memória

- **Em python, só existe quem tem nome:**

```
x = 1
y = x
x += 1
```

- **O número `1` *continua existindo* na memória após a terceira linha ser executada**
  - Mas o que acontece na segunda linha?!?!?
    - » *Temos duas "cópias" do número `1` na memória?*
- **DEPENDE**
  - Em python, não!
    - » *O número `1` simplesmente tem dois nomes (`x` e `y`)*
    - » *Na terceira linha, ele volta a ter apenas um nome (`y`)*

O que são “variáveis”?

O que são “variáveis”?

- **Nomes que usamos para acessar dados armazenados em algum lugar na memória do computador?**

O que são “variáveis”?

- **Nomes que usamos para acessar dados armazenados em algum lugar na memória do computador?**
- **Conceitos abstratos que representam uma informação relevante para a computação que estamos realizando?**

O que são “variáveis”?

- **Nomes que usamos para acessar dados armazenados em algum lugar na memória do computador?**
- **Conceitos abstratos que representam uma informação relevante para a computação que estamos realizando?**

Ambos

O que são “variáveis”?

- **Nomes que usamos para acessar dados armazenados em algum lugar na memória do computador?**
- **Conceitos abstratos que representam uma informação relevante para a computação que estamos realizando?**

Ambos

- **Mas cada linguagem dá mais ênfase a este ou aquele ponto de vista**

igualdade e identidade

- **Em python**
  - `x += 1` não altera o número `1` armazenado na memória, mas sim o valor associado à variável `x` (o número `2` "nasce" e recebe o nome `x`, enquanto o número `1` é "jogado fora")

igualdade e identidade

- **Em python**
  - `x += 1` não altera o número `1` armazenado na memória, mas sim o valor associado à variável `x` (o número `2` "nasce" e recebe o nome `x`, enquanto o número `1` é "jogado fora")
    - » *Pensando em variáveis como conceitos, isso faz sentido: o valor da variável é o conceito, e ele foi modificado; o número armazenado é* ***outro*** *número*

igualdade e identidade

- **Em python**
  - `x += 1` não altera o número `1` armazenado na memória, mas sim o valor associado à variável `x` (o número `2` "nasce" e recebe o nome `x`, enquanto o número `1` é "jogado fora")
    - » *Pensando em variáveis como conceitos, isso faz sentido: o valor da variável é o conceito, e ele foi modificado; o número armazenado é* ***outro*** *número*
- **Em outras linguagens, como C, as coisas são diferentes:**

igualdade e identidade

- **Em python**
  - `x += 1` não altera o número `1` armazenado na memória, mas sim o valor associado à variável `x` (o número `2` "nasce" e recebe o nome `x`, enquanto o número `1` é "jogado fora")
    - » *Pensando em variáveis como conceitos, isso faz sentido: o valor da variável é o conceito, e ele foi modificado; o número armazenado é* ***outro*** *número*
- **Em outras linguagens, como C, as coisas são diferentes:**
  - `x = y` guarda duas "cópias" do mesmo número em lugares diferentes da memória

igualdade e identidade

- **Em python**
  - `x += 1` não altera o número `1` armazenado na memória, mas sim o valor associado à variável `x` (o número `2` "nasce" e recebe o nome `x`, enquanto o número `1` é "jogado fora")
    - » *Pensando em variáveis como conceitos, isso faz sentido: o valor da variável é o conceito, e ele foi modificado; o número armazenado é* ***outro*** *número*
- **Em outras linguagens, como C, as coisas são diferentes:**
  - `x = y` guarda duas "cópias" do mesmo número em lugares diferentes da memória
  - `x += 1` procura o lugar na memória em que a variável `x` está armazenada e altera o valor que está ali

igualdade e identidade

- **Em python**
  - `x += 1` não altera o número `1` armazenado na memória, mas sim o valor associado à variável `x` (o número `2` "nasce" e recebe o nome `x`, enquanto o número `1` é "jogado fora")
    - » *Pensando em variáveis como conceitos, isso faz sentido: o valor da variável é o conceito, e ele foi modificado; o número armazenado é* ***outro*** *número*
- **Em outras linguagens, como C, as coisas são diferentes:**
  - `x = y` guarda duas "cópias" do mesmo número em lugares diferentes da memória
  - `x += 1` procura o lugar na memória em que a variável `x` está armazenada e altera o valor que está ali
    - » *Pensando em variáveis como nomes para um dado na memória, isso faz sentido: o conteúdo daquele espaço de memória foi modificado*

igualdade e identidade

E por que eu preciso me preocupar com isso?

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
```

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
compras = ["lampião de gás", "lenha"]
adiciona(compras)
```

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
compras = ["lampião de gás", "lenha"]
adiciona(compras)
```

Qual item você quer adicionar à lista?

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
compras = ["lampião de gás", "lenha"]
adiciona(compras)
```

Qual item você quer adicionar à lista? querosene

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
compras = ["lampião de gás", "lenha"]
adiciona(compras)
print(compras)
```

```
Qual item você quer adicionar à lista? querosene
```

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
compras = ["lampião de gás", "lenha"]
adiciona(compras)
print(compras)
```

```
Qual item você quer adicionar à lista? querosene
['lampião de gás', 'lenha', 'querosene']
```

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
compras = ["lampião de gás", "lenha"]
adiciona(compras)
print(compras)
```

```
Qual item você quer adicionar à lista? querosene
['lampião de gás', 'lenha', 'querosene']
```

- **Na chamada de função, a lista `compras` passa a ter (temporariamente) dois nomes: `compras` e `l`**

igualdade e identidade

E por que eu preciso me preocupar com isso?

```
def adiciona(l):
    x = input("Qual item você quer adicionar à lista? ")
    l.append(x)
compras = ["lampião de gás", "lenha"]
adiciona(compras)
print(compras)
```

```
Qual item você quer adicionar à lista? querosene
['lampião de gás', 'lenha', 'querosene']
```

- **Na chamada de função, a lista `compras` passa a ter (temporariamente) dois nomes: `compras` e `l`**
  - Como `append()` modifica a lista, o que acontece dentro da função se reflete fora da função

igualdade e identidade

- **Na prática:**

igualdade e identidade

- **Na prática:**
  - Alterações feitas pelo operador de atribuição (=) nunca se "propagam" para fora do escopo atual

igualdade e identidade

- **Na prática:**
  - Alterações feitas pelo operador de atribuição (=) nunca se "propagam" para fora do escopo atual
    - » *Mas, com listas,* `+=` *é equivalente a* `lista.extend()` 🤦

igualdade e identidade

- **Na prática:**
  - Alterações feitas pelo operador de atribuição (=) nunca se "propagam" para fora do escopo atual
    - » *Mas, com listas,* `+=` *é equivalente a* `lista.extend()` 
  - Alterações feitas por chamadas de métodos (`variável.método()`) geralmente se "propagam" para fora do escopo atual

igualdade e identidade

- **Na prática:**
  - Alterações feitas pelo operador de atribuição (`=`) nunca se "propagam" para fora do escopo atual
    - » *Mas, com listas,* `+=` *é equivalente a* `lista.extend()` 🤦
  - Alterações feitas por chamadas de métodos (`variável.método()`) geralmente se "propagam" para fora do escopo atual
    - » *Mas não com tipos imutáveis, como strings* 🙄

igualdade e identidade

- **Na prática:**
  - Alterações feitas pelo operador de atribuição (=) nunca se "propagam" para fora do escopo atual
    - » *Mas, com listas,* `+=` *é equivalente a* `lista.extend()` 🤦
  - Alterações feitas por chamadas de métodos (`variável.método()`) geralmente se "propagam" para fora do escopo atual
    - » *Mas não com tipos imutáveis, como strings* 🙄
  - Alterações feitas com alguns (poucos) outros operadores específicos, como **`del`**, se "propagam" para fora do escopo atual

igualdade e identidade

- **Na prática:**
  - Alterações feitas pelo operador de atribuição (=) nunca se "propagam" para fora do escopo atual
    - » *Mas, com listas,* `+=` *é equivalente a* `lista.extend()` 🤦
  - Alterações feitas por chamadas de métodos (`variável.método()`) geralmente se "propagam" para fora do escopo atual
    - » *Mas não com tipos imutáveis, como strings* 🙄
  - Alterações feitas com alguns (poucos) outros operadores específicos, como **`del`**, se "propagam" para fora do escopo atual

igualdade e identidade

O que acontece aqui?

igualdade e identidade

O que acontece aqui?

```
lista1 = ["Beethoven", "Bach", "Berio"]
lista2 = lista1
lista1[0] = "Beatles"
print(lista2)
```

igualdade e identidade

O que acontece aqui?

```
lista1 = ["Beethoven", "Bach", "Berio"]
lista2 = lista1
lista1[0] = "Beatles"
print(lista2)
```

['Beatles', 'Bach', 'Berio']

igualdade e identidade

O que acontece aqui?

```
lista1 = ["Beethoven", "Bach", "Berio"]
lista2 = lista1
lista1[0] = "Beatles"
print(lista2)
```

['Beatles', 'Bach', 'Berio']

Embora tenhamos usado o operador de atribuição (=), ele foi usado para alterar *um elemento* da lista, ou seja, o conteúdo da lista, e não a lista em si.

igualdade e identidade

E se eu quiser fazer uma cópia “de verdade” (um *clone*)?

igualdade e identidade

E se eu quiser fazer uma cópia “de verdade” (um *clone*)?

```
def clone(l1):
    l2 = []
    for item in l1:
        l2.append(item)
    return l2
```

igualdade e identidade

E se eu quiser fazer uma cópia “de verdade” (um *clone*)?

```
def clone(l1):
    l2 = []
    for item in l1:
        l2.append(item)
    return l2
```

```
l2 = l1[:]
```

Oncotô?

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo
- **Essas partes também podem interagir com o "mundo"**

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo
- **Essas partes também podem interagir com o "mundo"**
  - `input()` e `print()`

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo
- **Essas partes também podem interagir com o "mundo"**
  - `input()` e `print()`
- **Ou podem interagir com as outras partes do programa**

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo
- **Essas partes também podem interagir com o "mundo"**
  - `input()` e `print()`
- **Ou podem interagir com as outras partes do programa**
  - *recebendo* e *devolvendo* dados — parâmetros e **`return`**

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo
- **Essas partes também podem interagir com o "mundo"**
  - `input()` e `print()`
- **Ou podem interagir com as outras partes do programa**
  - *recebendo* e *devolvendo* dados — parâmetros e **`return`**

```
def média(a, b):
    return (a + b) / 2
```

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo
- **Essas partes também podem interagir com o "mundo"**
  - `input()` e `print()`
- **Ou podem interagir com as outras partes do programa**
  - *recebendo* e *devolvendo* dados — parâmetros e **`return`**

```
def média(a, b):
    return (a + b) / 2
```

  - Modificando uma variável global — **`global`**

Oncotô?

- **Dividir o programa em partes com *nomes* — funções**
  - Cada parte funciona de maneira independente, pois as variáveis "dentro" de cada uma normalmente não são visíveis para as outras — escopo
- **Essas partes também podem interagir com o "mundo"**
  - `input()` e `print()`
- **Ou podem interagir com as outras partes do programa**
  - *recebendo* e *devolvendo* dados — parâmetros e **`return`**

```
def média(a, b):
    return (a + b) / 2
```

  - Modificando uma variável global — **`global`**
  - **Modificando o conteúdo de uma variável (mutável) recebida como parâmetro**

And now for something completely different

Coleções de coleções

- Os itens de uma coleção/lista podem ser de qualquer tipo

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
```

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

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

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
outra_lista = [4, 5, 6]
lista.extend(outra_lista)
```

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
outra_lista = [4, 5, 6]
lista.extend(outra_lista)
print(lista)
```

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
outra_lista = [4, 5, 6]
lista.extend(outra_lista)
print(lista)
```

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

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
outra_lista = [4, 5, 6]
lista.extend(outra_lista)
print(lista)
lista = [1, 2, 3]
```

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

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
outra_lista = [4, 5, 6]
lista.extend(outra_lista)
print(lista)
lista = [1, 2, 3]
lista.append(outra_lista)
```

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

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
outra_lista = [4, 5, 6]
lista.extend(outra_lista)
print(lista)
lista = [1, 2, 3]
lista.append(outra_lista)
print(lista)
```

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

Coleções de coleções

- **Os itens de uma coleção/lista podem ser de qualquer tipo**
  - (E, em python, podem inclusive ser de tipos diferentes)
- **Um item pode, inclusive, ser uma lista!**

```
lista = [1, 2, 3]
outra_lista = [4, 5, 6]
lista.extend(outra_lista)
print(lista)
lista = [1, 2, 3]
lista.append(outra_lista)
print(lista)
```

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

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

Coleções de coleções

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
```

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
```

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
```

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
```

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

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
lista1.append(lista2)
```

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

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
lista1.append(lista2)
lista2.append("olá")
```

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

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
lista1.append(lista2)
lista2.append("olá")
print(lista2)
print(lista1)
print(juntas)
```

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

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
lista1.append(lista2)
lista2.append("olá")
print(lista2)
print(lista1)
print(juntas)
```

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

[4, 5, 6, 'olá']

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
lista1.append(lista2)
lista2.append("olá")
print(lista2)
print(lista1)
print(juntas)
```

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

[4, 5, 6, 'olá']

[1, 2, 3, [4, 5, 6, 'olá']]

Coleções de coleções

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
juntas = [lista1, lista2]
print(juntas)
lista1.append(lista2)
lista2.append("olá")
print(lista2)
print(lista1)
print(juntas)
```

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

[4, 5, 6, 'olá']

[1, 2, 3, [4, 5, 6, 'olá']]

[[1, 2, 3, [4, 5, 6, 'olá']], [4, 5, 6, 'olá']]

Mas para que fazer um rolo desses?!?

Mas para que fazer um rolo desses?!?

ABSTRAÇÕES

Abstrações

- As linguagens de programação oferecem “peças” que usamos para construir nossos programas

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções
  - coleções

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções
  - coleções
  - ...

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções
  - coleções
  - ...
- **Com essas "peças", construímos abstrações que se aproximam dos problemas reais que queremos solucionar**

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções
  - coleções
  - ...
- **Com essas "peças", construímos abstrações que se aproximam dos problemas reais que queremos solucionar**
  - Assim como as peças de um Lego podem ser usadas para construir formas diversas

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções
  - coleções
  - ...
- **Com essas "peças", construímos abstrações que se aproximam dos problemas reais que queremos solucionar**
  - Assim como as peças de um Lego podem ser usadas para construir formas diversas
    - » *Por exemplo, nosso exercício com polinômios*

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções
  - coleções
  - ...
- **Com essas "peças", construímos abstrações que se aproximam dos problemas reais que queremos solucionar**
  - Assim como as peças de um Lego podem ser usadas para construir formas diversas
    - » *Por exemplo, nosso exercício com polinômios*
- **Podemos usar uma coleção de coleções para criar uma nova abstração:**

Abstrações

- **As linguagens de programação oferecem "peças" que usamos para construir nossos programas**
  - variáveis
  - funções
  - coleções
  - ...
- **Com essas "peças", construímos abstrações que se aproximam dos problemas reais que queremos solucionar**
  - Assim como as peças de um Lego podem ser usadas para construir formas diversas
    - » *Por exemplo, nosso exercício com polinômios*
- **Podemos usar uma coleção de coleções para criar uma nova abstração: Matrizes**

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
print(tabela_de_preços)
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
print(tabela_de_preços)
```

[['item', 'à vista', 'a prazo'], ['inhame', 10, 11], ['batata', 13, 15], ['aipim', 12, 14]]

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
for line in tabela_de_preços:
    for item in line:
        print(item, end="\t")
    print()
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
for line in tabela_de_preços:
    for item in line:
        print(item, end="\t")
    print()
```

```
item    à vista a prazo
inhame  10      11
batata  13      15
aipim   12      14
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
for line in tabela_de_preços:
    for item in line:
        print(item, end="\t")
    print()
```

```
item    à vista a prazo
inhame  10      11
batata  13      15
aipim   12      14
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
for line in tabela_de_preços:
    for item in line:
        print(item, end="\t")
    print()
```

```
item    à vista a prazo
inhame  10      11
batata  13      15
aipim   12      14
```

Matrizes

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} = (a_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}$$

Matrizes

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{1,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix} = (a_{i,j})_{0 \leq i < m, 0 \leq j < n}$$

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
l2 = tabela_de_preços[2]
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
l2 = tabela_de_preços[2]
print(l2[0])
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
l2 = tabela_de_preços[2]
print(l2[0])
```

```
batata
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
l2 = tabela_de_preços[2]
print(l2[0])
print(tabela_de_preços[2][0])
```

```
batata
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
l2 = tabela_de_preços[2]
print(l2[0])
print(tabela_de_preços[2][0])
```

```
batata
batata
```

Matrizes

```
tabela_de_preços = [["item", "à vista", "a prazo"]]
tabela_de_preços.append(["inhame", 10, 11])
tabela_de_preços.append(["batata", 13, 15])
tabela_de_preços.append(["aipim", 12, 14])
l2 = tabela_de_preços[2]
print(l2[0])
print(tabela_de_preços[2][0])
```

```
batata
batata
```

Nada obriga que todas as linhas tenham o mesmo número de elementos

Nada obriga que todas as linhas tenham o mesmo número de elementos

Matrizes desse tipo funcionam por *convenção*

Matrizes

Também podemos criar uma matriz assim:

```
MATRIZ = [[4, 1, 8, 3],
          [2, 5, 7, 0],
          [6, 9, 0, 3]]
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
```

0

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
```

0

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
matriz[1][1] = 3
```

0

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
matriz[1][1] = 3
print(matriz[1][1])
```

```
0
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
matriz[1][1] = 3
print(matriz[1][1])
```

```
0
3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
matriz[1][1] = 3
print(matriz[1][1])
```

```
0
3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
matriz[1][1] = 3
print(matriz[1][1])
print(matriz[2][1])
```

```
0
3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
matriz[1][1] = 3
print(matriz[1][1])
print(matriz[2][1])
```

```
0
3
3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
print(matriz[2][2])
matriz[1][1] = 3
print(matriz[1][1])
print(matriz[2][1])
```

```
0
3
3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

[[0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0]]

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

[[0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0]]

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

[[0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0]]

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = []
for i in range(5):
    matriz.append(linha[:])
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = []
for i in range(5):
    matriz.append(linha[:])
matriz[1][1] = 3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = []
for i in range(5):
    matriz.append(linha[:])
matriz[1][1] = 3
print(matriz[1][1])
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = []
for i in range(5):
    matriz.append(linha[:])
matriz[1][1] = 3
print(matriz[1][1])
```

```
3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = []
for i in range(5):
    matriz.append(linha[:])
matriz[1][1] = 3
print(matriz[1][1])
print(matriz[2][1])
```

```
3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = []
for i in range(5):
    matriz.append(linha[:])
matriz[1][1] = 3
print(matriz[1][1])
print(matriz[2][1])
```

```
3
0
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = []
for i in range(5):
    matriz.append(linha[:])
matriz[1][1] = 3
print(matriz[1][1])
print(matriz[2][1])
```

```
3
0
```

Matrizes

```
def cria_matriz(linhas, colunas, valor_inicial):
```

Matrizes

```
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
```

Matrizes

```
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
    matriz = []
```

Matrizes

```
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
    matriz = []
    for l in range(linhas):
```

Matrizes

```
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
    matriz = []
    for l in range(linhas):
        matriz.append(umalinha[:])
```

Matrizes

```
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
    matriz = []
    for l in range(linhas):
        matriz.append(umalinha[:])
    return matriz
```

Matrizes

```
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
    matriz = []
    for l in range(linhas):
        matriz.append(umalinha[:])
    return matriz
```

Matrizes

```python
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
    matriz = []
    for l in range(linhas):
        matriz.append(umalinha[:])
    return matriz
```

```python
def cria_matriz(linhas, colunas, valor_inicial):
    matriz = []
    for i in range(linhas):
        linha = []
        for j in range(colunas):
            linha.append(valor_inicial)
        matriz.append(linha)
    return matriz
```

Matrizes

```python
def cria_matriz(linhas, colunas, valor_inicial):
    umalinha = [valor_inicial] *colunas
    matriz = []
    for l in range(linhas):
        matriz.append(umalinha[:])
    return matriz
```

```python
def cria_matriz(linhas, colunas, valor_inicial):
    matriz = []
    for i in range(linhas):
        linha = []
        for j in range(colunas):
            linha.append(valor_inicial)
        matriz.append(linha)
    return matriz
```

Impressão de matrizes

```
def imprime_matriz(A):
```

Impressão de matrizes

```
def imprime_matriz(A):
    for linha in A:
```

Impressão de matrizes

```
def imprime_matriz(A):
    for linha in A:
        for col in linha:
```

Impressão de matrizes

```
def imprime_matriz(A):
    for linha in A:
        for col in linha:
            print("{:4}".format(col), end="")
```

Impressão de matrizes

```
def imprime_matriz(A):
    for linha in A:
        for col in linha:
            print("{:4}".format(col), end="")
        print()
```

Soma de matrizes

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{bmatrix} + \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \end{bmatrix} = \begin{bmatrix} a_{0,0} + b_{0,0} & a_{0,1} + b_{0,1} & a_{0,2} + b_{0,2} \\ a_{1,0} + b_{1,0} & a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} \end{bmatrix}$$

Soma de matrizes

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{bmatrix} + \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \end{bmatrix} = \begin{bmatrix} a_{0,0} + b_{0,0} & a_{0,1} + b_{0,1} & a_{0,2} + b_{0,2} \\ a_{1,0} + b_{1,0} & a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 400 & 500 & 600 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 404 & 505 & 606 \end{bmatrix}$$

Soma de matrizes

```
def soma_matrizes(A, B):









    return C
```

Soma de matrizes

```
def soma_matrizes(A, B):
    nlinhas = len(A)









    return C
```

Soma de matrizes

```
def soma_matrizes(A, B):
    nlinhas = len(A)
    ncols = len(A[0])




    return C
```

Soma de matrizes

```
def soma_matrizes(A, B):
    nlinhas = len(A)
    ncols = len(A[0])
    C = cria_matriz(nlinhas, ncols, 0)




    return C
```

Soma de matrizes

```
def soma_matrizes(A, B):
    nlinhas = len(A)
    ncols = len(A[0])
    C = cria_matriz(nlinhas, ncols, 0)
    for l in range(nlinhas):



    return C
```

Soma de matrizes

```
def soma_matrizes(A, B):
    nlinhas = len(A)
    ncols = len(A[0])
    C = cria_matriz(nlinhas, ncols, 0)
    for l in range(nlinhas):
        for c in range(ncols):

    return C
```

Soma de matrizes

```
def soma_matrizes(A, B):
    nlinhas = len(A)
    ncols = len(A[0])
    C = cria_matriz(nlinhas, ncols, 0)
    for l in range(nlinhas):
        for c in range(ncols):
            C[l][c] = A[l][c] + B[l][c]
    return C
```

Soma de matrizes

Soma de matrizes

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

Soma de matrizes

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

B = [[100, 200, 300],
     [400, 500, 600]]
```

Soma de matrizes

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

B = [[100, 200, 300],
     [400, 500, 600]]

C = soma_matrizes(A, B)
```

Soma de matrizes

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

B = [[100, 200, 300],
     [400, 500, 600]]

C = soma_matrizes(A, B)
imprime_matriz(C)
```

Soma de matrizes

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

B = [[100, 200, 300],
     [400, 500, 600]]

C = soma_matrizes(A, B)
imprime_matriz(C)
```

---

101 202 303

404 505 606

Exercícios

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):

        for j in range(len(A[0])):









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):
        sóZeros = True
        for j in range(len(A[0])):









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):
        sóZeros = True
        for j in range(len(A[0])):
            if A[i][j] != 0:
                sóZeros = False









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):
        sóZeros = True
        for j in range(len(A[0])):
            if A[i][j] != 0:
                sóZeros = False
        if sóZeros:
            linhas_vazias.append(i)









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):
        sóZeros = True
        for j in range(len(A[0])):
            if A[i][j] != 0:
                sóZeros = False
        if sóZeros:
            linhas_vazias.append(i)
    for j in range(len(A[0])):
        sóZeros = True









    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):
        sóZeros = True
        for j in range(len(A[0])):
            if A[i][j] != 0:
                sóZeros = False
        if sóZeros:
            linhas_vazias.append(i)
    for j in range(len(A[0])):
        sóZeros = True
        for i in range(len(A)):
            if A[i][j] != 0:
                sóZeros = False


    return linhas_vazias, colunas_vazias
```

Exercício

Escreva uma função que recebe uma matriz A e devolve duas listas: a lista das linhas e a lista das colunas em que todos os elementos são zero

```
def encontra_zeros(A):
    linhas_vazias = []
    colunas_vazias = []
    for i in range(len(A)):
        sóZeros = True
        for j in range(len(A[0])):
            if A[i][j] != 0:
                sóZeros = False
        if sóZeros:
            linhas_vazias.append(i)
    for j in range(len(A[0])):
        sóZeros = True
        for i in range(len(A)):
            if A[i][j] != 0:
                sóZeros = False
        if sóZeros:
            colunas_vazias.append(j)
    return linhas_vazias, colunas_vazias
```

Exercício

Dizemos que uma matriz quadrada A de dimensão $n \times n$ é um quadrado latino de ordem n se, em cada linha e em cada coluna, aparecem todos os inteiros $1, 2, 3, \ldots, n$ (ou seja, cada linha e coluna é permutação dos inteiros $1, 2, \ldots, n$). Por exemplo:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

Exercício

1. Escreva uma função que recebe como parâmetro uma lista L com n inteiros e verifica se em L ocorrem todos os inteiros de 1 a n.

Exercício

1. **Escreva uma função que recebe como parâmetro uma lista L com n inteiros e verifica se em L ocorrem todos os inteiros de 1 a n.**

```
def checa_permutação(l):
```

Exercício

1. Escreva uma função que recebe como parâmetro uma lista L com n inteiros e verifica se em L ocorrem todos os inteiros de 1 a n.

```
def checa_permutação(l):
    ok = True
```

Exercício

1. **Escreva uma função que recebe como parâmetro uma lista L com n inteiros e verifica se em L ocorrem todos os inteiros de 1 a n.**

```
def checa_permutação(l):
    ok = True



    return ok
```

Exercício

1. **Escreva uma função que recebe como parâmetro uma lista L com n inteiros e verifica se em L ocorrem todos os inteiros de 1 a n.**

```
def checa_permutação(l):
    ok = True
    for i in range(1, len(l) +1):



    return ok
```

Exercício

1. **Escreva uma função que recebe como parâmetro uma lista L com n inteiros e verifica se em L ocorrem todos os inteiros de 1 a n.**

```
def checa_permutação(l):
    ok = True
    for i in range(1, len(l) +1):
        if not i in l:
            ok = False
    return ok
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):




    return True
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
    for linha in A:









    return True
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
    for linha in A:
        if not checa_permutação(linha):
            return False









    return True
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
    for linha in A:
        if not checa_permutação(linha):
            return False
    for j in range(len(A[0])):









    return True
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
    for linha in A:
        if not checa_permutação(linha):
            return False
    for j in range(len(A[0])):
        col = []




    return True
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
    for linha in A:
        if not checa_permutação(linha):
            return False
    for j in range(len(A[0])):
        col = []
        for i in range(len(A)):




    return True
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
    for linha in A:
        if not checa_permutação(linha):
            return False
    for j in range(len(A[0])):
        col = []
        for i in range(len(A)):
            col.append(A[i][j])


    return True
```

Exercício

2. **Usando essa função, verifique se uma dada matriz inteira A de dimensão $n \times n$ é um quadrado latino de ordem n.**

```
def checa_latino(A):
    for linha in A:
        if not checa_permutação(linha):
            return False
    for j in range(len(A[0])):
        col = []
        for i in range(len(A)):
            col.append(A[i][j])
        if not checa_permutação(col):
            return False
    return True
```

Matrizes

Este é um assunto novo?

Matrizes

Este é um assunto novo?

Não!

Matrizes

Este é um assunto novo?

Não! — Listas e laços encaixados

Matrizes

Este é um assunto novo?

Não! — Listas e laços encaixados

Sim!

Matrizes

Este é um assunto novo?

Não! — Listas e laços encaixados

Sim! — Matrizes são uma abstração diferente

Strings vs Listas

Strings vs listas

- **Strings são bastante similares a listas**

Strings vs listas

- **Strings são bastante similares a listas**
  - `len()`, `string[]`, **`for`** etc. funcionam como esperado

Strings vs listas

- **Strings são bastante similares a listas**
  - `len()`, `string[]`, **`for`** etc. funcionam como esperado
- **MAS!**

Strings vs listas

- **Strings são bastante similares a listas**
  - `len()`, `string[]`, **`for`** etc. funcionam como esperado
- **MAS!**
- **Strings são imutáveis**

Strings vs listas

- **Strings são bastante similares a listas**
  - `len()`, `string[]`, **`for`** etc. funcionam como esperado
- **MAS!**
- **Strings são imutáveis**
  - `append()`, `string[x] = "a"` etc. não existem ou não funcionam como esperado

Strings vs listas

- **Strings têm vários métodos úteis**

Strings vs listas

- **Strings têm vários métodos úteis**
  - `.format()`

Strings vs listas

- **Strings têm vários métodos úteis**
  - `.format()`
  - `.strip()`

Strings vs listas

- **Strings têm vários métodos úteis**
  - `.format()`
  - `.strip()`
  - `.upper()` / `.lower()` / `.casefold()`

Strings vs listas

- **Strings têm vários métodos úteis**
  - `.format()`
  - `.strip()`
  - `.upper()` / `.lower()` / `.casefold()`
  - `.find()` / `.replace()`

Strings vs listas

- **Strings têm vários métodos úteis**
  - `.format()`
  - `.strip()`
  - `.upper()` / `.lower()` / `.casefold()`
  - `.find()` / `.replace()`
  - `.split()` / `.join()`

Strings vs listas

- **Strings têm vários métodos úteis**
  - `.format()`
  - `.strip()`
  - `.upper()` / `.lower()` / `.casefold()`
  - `.find()` / `.replace()`
  - `.split()` / `.join()`

https://docs.python.org/3/library/stdtypes.html#string-methods

Strings vs listas

- **Strings têm vários métodos úteis**
  - `.format()`
  - `.strip()`
  - `.upper()` / `.lower()` / `.casefold()`
  - `.find()` / `.replace()`
  - `.split()` / `.join()`

https://docs.python.org/3/library/stdtypes.html#string-methods

Como strings são imutáveis, todos esses devolvem uma nova string

(exceto `.split()`, que devolve uma lista)