MAC 110 — Introdução à Ciência da Computação

Aula 15

Nelson Lago

BMAC – 2024

Previously on MAC 110...

Coleções

A principal coleção em python é a *lista*:

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

```
print(cores[0])
```

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

```
print(cores[0])
```

---

vermelho

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

```
print(cores[0])
print(cores[2])
```

---

vermelho

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

```
print(cores[0])
print(cores[2])
```

---

vermelho

amarelo

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

```
print(cores[0])
print(cores[2])
```

---

vermelho

amarelo

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

```
print(cores[0])
print(cores[2])
```

---

vermelho

amarelo

Coleções

A principal coleção em python é a *lista*:

```
cores = ["vermelho", "azul", "amarelo"]
```

Uma lista é um conjunto ordenado:

```
print(cores[0])
print(cores[2])
```

---

vermelho

amarelo

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
```

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 p in primos:
    print("O número", p, "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 p in primos:
    print("O número", p, "é primo")
```

Repetições com coleções

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

Repetições com coleções

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

```
primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
n = 1
for p in primos:
  print("O {}o primo é {}".format(n, p))
  n += 1
```

Comandos básicos com listas

- `lista = [a, b, c]`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`

And now for something slightly different

Tipos de repetição

- **Dois tipos fundamentais de repetição**
  1. Repetições até atingir um resultado
     - » *Encontrar o próximo primo*
     - » *Reiniciar o jogo até o usuário escolher "sair"*
     - » *...*
  2. Repetições sobre os elementos de um conjunto
     - » *Apresentar todos os pixels de uma foto na tela*
     - » *Trocar todas as letras de um texto para maiúsculas*
     - » *...*

Tipos de repetição

- **De maneira geral:**

Tipos de repetição

- **De maneira geral:**
  - Repetições até atingir um resultado ⟶ `while`

Tipos de repetição

- **De maneira geral:**
  - Repetições até atingir um resultado ⟶ `while`
  - Repetições sobre os elementos de um conjunto ⟶ `for`

Tipos de repetição

- **De maneira geral:**
  - Repetições até atingir um resultado ⟶ `while`
  - Repetições sobre os elementos de um conjunto ⟶ `for`

mas quando fazemos repetições sobre um conjunto de inteiros, sempre usamos **`while`** também 😢

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

```
def naturais(n):
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

```
def naturais(n):




    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

```
def naturais(n):
    lista = []




    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

```
def naturais(n):
    lista = []
    i = 0
    while i < n:

        i += 1
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

```
def naturais(n):
    lista = []
    i = 0
    while i < n:
        lista.append(i)
        i += 1
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

```
def naturais(n):
    lista = []

    for i in [lista de inteiros até n]:
        lista.append(i)

    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais

```
def naturais(n):
    lista = []

    for i in range(n):
        lista.append(i)

    return lista
```

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)
```

A função `range()`

```
range(início, final, passo)
```

O intervalo é sempre

fechado no início e aberto no final

A função `range()`

```
range(início, final, passo)
```

O intervalo é sempre

fechado no início e aberto no final

`range(4)` → `range(0, 4)` → de zero a três! (quatro elementos)

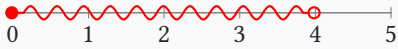
A função `range()`

`range(início, final, passo)`

O intervalo é sempre

fechado no início e aberto no final

`range(4)` → `range(0, 4)` → de zero a três! (quatro elementos)



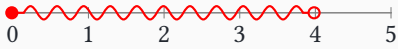
A função `range()`

`range(início, final, passo)`

O intervalo é sempre

fechado no início e aberto no final

`range(4)` → `range(0, 4)` → de zero a três! (quatro elementos)



`range(2, 7)` → de dois a seis! (cinco elementos)

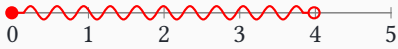
A função `range()`

`range(início, final, passo)`

O intervalo é sempre

fechado no início e aberto no final

`range(4)` → `range(0, 4)` → de zero a três! (quatro elementos)



`range(2, 7)` → de dois a seis! (cinco elementos)

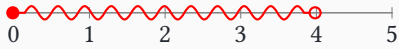
`range(1, 10, 2)` → de um a nove! (cinco elementos)

A função range()

`range(início, final, passo)`

O intervalo é sempre fechado no início e aberto no final

`range(4)` → `range(0, 4)` → de zero a três! (quatro elementos)



`range(2, 7)` → de dois a seis! (cinco elementos)

`range(1, 10, 2)` → de um a nove! (cinco elementos)

`range(1, 11, 2)` → de um a nove! (cinco elementos)

A função range()

```
range(início, final, passo)
```

A função `range()`

```
range(início, final, passo)
```

O intervalo é sempre

fechado no início e aberto no final

A função `range()`

`range(início, final, passo)`

O intervalo é sempre
fechado no início e aberto no final

O número total de elementos é $\left\lceil \frac{\text{final} - \text{início}}{\text{passo}} \right\rceil$

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais estritamente positivos

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais estritamente positivos

```
def naturais(n):
    lista = []
    for i in range(n):
        lista.append(i)
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais estritamente positivos

```
def naturais(n):
    lista = []
    for i in range(n):
        lista.append(i+1)
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros naturais estritamente positivos

```
def naturais(n):
    lista = []
    for i in range(n):
        lista.append(i+1)
    return lista
```

```
def naturais(n):
    lista = []
    for i in range(1, n+1):
        lista.append(i)
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros ímpares

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros ímpares

```
def ímpares(n):
    lista = []
    for i in range(         ):
        lista.append(i)
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros ímpares

```
def ímpares(n):
    lista = []
    for i in range(1,       ):
        lista.append(i)
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros ímpares

```
def ímpares(n):
    lista = []
    for i in range(1,    , 2):
        lista.append(i)
    return lista
```

Coleções

Escreva uma função que recebe um número natural n como parâmetro e devolve uma lista com os n primeiros ímpares

```
def ímpares(n):
    lista = []
    for i in range(1, 2*n, 2):
        lista.append(i)
    return lista
```

Coleções

Escreva uma função que recebe um número natural n e imprime uma contagem regressiva de n até zero

Coleções

Escreva uma função que recebe um número natural n e imprime uma contagem regressiva de n até zero

```
def regressiva(n):
```

Coleções

Escreva uma função que recebe um número natural n e imprime uma contagem regressiva de n até zero

```
def regressiva(n):

        print(i)
```

Coleções

Escreva uma função que recebe um número natural n e imprime uma contagem regressiva de n até zero

```
def regressiva(n):
    for i in range(         ):
        print(i)
```

Coleções

Escreva uma função que recebe um número natural n e imprime uma contagem regressiva de n até zero

```
def regressiva(n):
    for i in range(n,       ):
        print(i)
```

Coleções

Escreva uma função que recebe um número natural n e imprime uma contagem regressiva de n até zero

```
def regressiva(n):
    for i in range(n,   , -1):
        print(i)
```

Coleções

Escreva uma função que recebe um número natural n e imprime uma contagem regressiva de n até zero

```
def regressiva(n):
    for i in range(n, -1, -1):
        print(i)
```

Coleções

Escreva uma função que recebe duas listas e devolve **`True`** caso elas sejam “iguais” (ou seja, têm o mesmo tamanho e os mesmos elementos nas mesmas posições) e **`False`** caso contrário

Coleções

Escreva uma função que recebe duas listas e devolve **`True`** caso elas sejam “iguais” (ou seja, têm o mesmo tamanho e os mesmos elementos nas mesmas posições) e **`False`** caso contrário

```
def compara_listas(l1, l2):
```

Coleções

Escreva uma função que recebe duas listas e devolve **`True`** caso elas sejam "iguais" (ou seja, têm o mesmo tamanho e os mesmos elementos nas mesmas posições) e **`False`** caso contrário

```
def compara_listas(l1, l2):
    if len(l1) != len(l2):
        return False
```

Coleções

Escreva uma função que recebe duas listas e devolve **`True`** caso elas sejam “iguais” (ou seja, têm o mesmo tamanho e os mesmos elementos nas mesmas posições) e **`False`** caso contrário

```
def compara_listas(l1, l2):
    if len(l1) != len(l2):
        return False
    for i in range(len(l1)):
```

Coleções

Escreva uma função que recebe duas listas e devolve **`True`** caso elas sejam “iguais” (ou seja, têm o mesmo tamanho e os mesmos elementos nas mesmas posições) e **`False`** caso contrário

```
def compara_listas(l1, l2):
    if len(l1) != len(l2):
        return False
    for i in range(len(l1)):
        if l1[i] != l2[i]:
            return False
```

Coleções

Escreva uma função que recebe duas listas e devolve **`True`** caso elas sejam "iguais" (ou seja, têm o mesmo tamanho e os mesmos elementos nas mesmas posições) e **`False`** caso contrário

```
def compara_listas(l1, l2):
    if len(l1) != len(l2):
        return False
    for i in range(len(l1)):
        if l1[i] != l2[i]:
            return False
    return True
```

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 p in primos:
    print("O número", p, "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 p in primos:
    print("O número", p, "é primo")
```

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

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`...`in` 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...in` 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`...`in` 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`...`in` 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`...`in` `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`...`in` 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`...`in` `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`...`in` 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`...`in` `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`...`in` 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`...`in` `range()` é uma boa escolha:**
  - `for n in range(10):`
  - `for n in range(len(minha_lista)):`
  - `for i in range(início,final):`

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

- **Imagine que queremos representar uma pessoa como um conjunto de dados (nome, RG, endereço, peso, altura...)**

Abstrações

- Imagine que queremos representar uma pessoa como um conjunto de dados (nome, RG, endereço, peso, altura...)
- Podemos criar uma variável para cada uma dessas coisas, mas e se houver várias pessoas?

Abstrações

- Imagine que queremos representar uma pessoa como um conjunto de dados (nome, RG, endereço, peso, altura...)
- Podemos criar uma variável para cada uma dessas coisas, mas e se houver várias pessoas?
- O ideal é representar todos esses dados como um **conjunto**

Abstrações

- Imagine que queremos representar uma pessoa como um conjunto de dados (nome, RG, endereço, peso, altura...)
- Podemos criar uma variável para cada uma dessas coisas, mas e se houver várias pessoas?
- O ideal é representar todos esses dados como um **conjunto**
- Um tipo de conjunto que já conhecemos é... a lista!

Abstrações

- Você pode usar uma lista para armazenar, por exemplo, o RG de uma pessoa na posição zero e seu nome na posição 1

Abstrações

- Você pode usar uma lista para armazenar, por exemplo, o RG de uma pessoa na posição zero e seu nome na posição 1
- Uma lista desse tipo representa um **conceito** composto (por exemplo, "pessoa") — ou seja, uma **abstração**

Abstrações

- **Você pode usar uma lista para armazenar, por exemplo, o RG de uma pessoa na posição zero e seu nome na posição 1**
- **Uma lista desse tipo representa um conceito composto (por exemplo, "pessoa") — ou seja, uma abstração**
  - Abstração: representação de algum conceito apenas com seus aspectos que são relevantes para o que queremos fazer

Abstrações

- **Você pode usar uma lista para armazenar, por exemplo, o RG de uma pessoa na posição zero e seu nome na posição 1**
- **Uma lista desse tipo representa um conceito composto (por exemplo, "pessoa") — ou seja, uma abstração**
  - Abstração: representação de algum conceito apenas com seus aspectos que são relevantes para o que queremos fazer
- **Nesse caso, em geral não faz muito sentido percorrer a lista**

Abstrações

- **Você pode usar uma lista para armazenar, por exemplo, o RG de uma pessoa na posição zero e seu nome na posição 1**
- **Uma lista desse tipo representa um conceito composto (por exemplo, "pessoa") — ou seja, uma abstração**
  - Abstração: representação de algum conceito apenas com seus aspectos que são relevantes para o que queremos fazer
- **Nesse caso, em geral não faz muito sentido percorrer a lista**
  - os elementos não são "itens"

Abstrações

- **Você pode usar uma lista para armazenar, por exemplo, o RG de uma pessoa na posição zero e seu nome na posição 1**
- **Uma lista desse tipo representa um conceito composto (por exemplo, "pessoa") — ou seja, uma abstração**
  - Abstração: representação de algum conceito apenas com seus aspectos que são relevantes para o que queremos fazer
- **Nesse caso, em geral não faz muito sentido percorrer a lista**
  - os elementos não são "itens"
  - o que queremos é acessar seus elementos usando os índices que representam cada aspecto do conceito ("RG", "nome" etc.)

Coleções

- **Em python, os elementos de uma lista não precisam ser todos do mesmo tipo**

Coleções

- **Em python, os elementos de uma lista não precisam ser todos do mesmo tipo**
  - (Em algumas outras linguagens sim!)

Coleções

- **Em python, os elementos de uma lista não precisam ser todos do mesmo tipo**
  - (Em algumas outras linguagens sim!)
- **uma lista pode inclusive conter uma outra lista como um de seus elementos**

Coleções

- **Em python, os elementos de uma lista não precisam ser todos do mesmo tipo**
  - (Em algumas outras linguagens sim!)
- **uma lista pode inclusive conter uma outra lista como um de seus elementos**
  - Então, dada a nossa abstração "pessoa" (que é uma lista), podemos fazer uma lista de pessoas (uma lista em que cada elemento é uma outra lista)

Coleções

Imagine um sistema de *login* com múltiplos usuários, por exemplo:

- **Alan Turing** — UID **`turing`**, senha **`tmachine`**
- **Ada Lovelace** — UID **`llace`**, senha **`anengine`**
- **Grace Hopper** — UID **`hopper`**, senha **`business`**
- **Charles Babbage** — UID **`cbb`**, senha **`analytical`**

Crie um sistema que lê o UID (*login*) e a senha do usuário e, se os dados estiverem corretos, escreve “Bem-vindo, [nome]!”; caso contrário, o sistema escreve “Login ou senha incorreto”.

Coleções

```
def main():
    uid = input("username: ")
    senha = input("senha: ")
    nome = checa_login(uid, senha)
    if nome == "":
        print("Login ou senha incorreto")
    else:
        print("Bem-vindo, {}!".format(nome))
```

Coleções

```
def main():
    uid = input("username: ")
    senha = input("senha: ")
    nome = checa_login(uid, senha)
    if nome == "":
        print("Login ou senha incorreto")
    else:
        print("Bem-vindo, {}!".format(nome))
```

```
def checa_login(login, pwd):
    name = ""
    if login == "turing" and pwd == "tmachine":
        name = "Alan Turing"
    elif login == "llace" and pwd == "anengine":
        name = "Ada Lovelace"
    elif login == "hopper" and pwd == "business":
        name = "Grace Hopper"
    return name
```

Coleções

- **users é uma lista com um número qualquer de usuários (neste exemplo, quatro)**
- **Cada usuário (elemento da lista users) é representado por uma lista com login, senha e nome**
  - (uma lista dentro da outra)

```
users = []
users.append(["turing", "tmachine", "Alan Turing"])
users.append(["llace", "anengine", "Ada Lovelace"])
users.append(["hopper", "business", "Grace Hopper"])
users.append(["cbb", "analytical", "Charles Babbage"])
```

Coleções

- **users é uma lista com um número qualquer de usuários (neste exemplo, quatro)**
- **Cada usuário (elemento da lista users) é representado por uma lista com login, senha e nome**
  - (uma lista dentro da outra)

```
users = []
users.append(["turing", "tmachine", "Alan Turing"])
users.append(["llace", "anengine", "Ada Lovelace"])
users.append(["hopper", "business", "Grace Hopper"])
users.append(["cbb", "analytical", "Charles Babbage"])
```

```
def checa_login(login, pwd, users):
    name = ""



    return name
```

Coleções

- **users é uma lista com um número qualquer de usuários (neste exemplo, quatro)**
- **Cada usuário (elemento da lista users) é representado por uma lista com login, senha e nome**
  - (uma lista dentro da outra)

```
users = []
users.append(["turing", "tmachine", "Alan Turing"])
users.append(["llace", "anengine", "Ada Lovelace"])
users.append(["hopper", "business", "Grace Hopper"])
users.append(["cbb", "analytical", "Charles Babbage"])
```

```
def checa_login(login, pwd, users):
    name = ""
    for u in users:



    return name
```

Coleções

- **users é uma lista com um número qualquer de usuários (neste exemplo, quatro)**
- **Cada usuário (elemento da lista users) é representado por uma lista com login, senha e nome**
  - (uma lista dentro da outra)

```
users = []
users.append(["turing", "tmachine", "Alan Turing"])
users.append(["llace", "anengine", "Ada Lovelace"])
users.append(["hopper", "business", "Grace Hopper"])
users.append(["cbb", "analytical", "Charles Babbage"])
```

```
def checa_login(login, pwd, users):
    name = ""
    for u in users:
        if login == u[0] and pwd == u[1]:

    return name
```

Coleções

- **users é uma lista com um número qualquer de usuários (neste exemplo, quatro)**
- **Cada usuário (elemento da lista users) é representado por uma lista com login, senha e nome**
  - (uma lista dentro da outra)

```
users = []
users.append(["turing", "tmachine", "Alan Turing"])
users.append(["llace", "anengine", "Ada Lovelace"])
users.append(["hopper", "business", "Grace Hopper"])
users.append(["cbb", "analytical", "Charles Babbage"])
```

```
def checa_login(login, pwd, users):
    name = ""
    for u in users:
        if login == u[0] and pwd == u[1]:
            name = u[2]
    return name
```

Coleções

- **users é uma lista com um número qualquer de usuários (neste exemplo, quatro)**
- **Cada usuário (elemento da lista users) é representado por uma lista com login, senha e nome**
  - (uma lista dentro da outra)

```
users = []
users.append(["turing", "tmachine", "Alan Turing"])
users.append(["llace", "anengine", "Ada Lovelace"])
users.append(["hopper", "business", "Grace Hopper"])
users.append(["cbb", "analytical", "Charles Babbage"])
```

```
def checa_login(login, pwd, users):

    for u in users:
        if login == u[0] and pwd == u[1]:
            return u[2]
    return ""
```

Polinômio

Um polinômio de uma variável ($3x^4 + 2x^2 + x + 5$) pode ser representado pela lista de seus coeficientes: `polinômio = [5, 1, 2, 0, 3]`

Nesta representação:

- **O tamanho da lista é o grau do polinômio +1**
- **O número na posição k da lista é coeficiente do monômio de grau k**
- **O último item da lista é sempre diferente de zero**

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```
def calcula_polinômio(p, x):
```

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```
def calcula_polinômio(p, x):



    return resultado
```

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```
def calcula_polinômio(p, x):
    resultado = 0



    return resultado
```

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```
def calcula_polinômio(p, x):
    resultado = 0
    for i in range(len(p)):

    return resultado
```

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```python
def calcula_polinômio(p, x):
    resultado = 0
    for i in range(len(p)):
        resultado += p[i] * x**i
    return resultado
```

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```python
def calcula_polinômio(p, x):
    resultado = 0
    for i in range(len(p)):
        resultado += p[i] * x**i
    return resultado
```

- **Usamos uma lista para criar uma abstração: polinômios**

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```python
def calcula_polinômio(p, x):
    resultado = 0
    for i in range(len(p)):
        resultado += p[i] * x**i
    return resultado
```

- **Usamos uma lista para criar uma abstração: polinômios**
- **Neste caso, nós iteramos por essa lista!**

Polinômio

Escreva uma função `calcula_polinômio(p, x)` em python que recebe um polinômio `p` como definido anteriormente e um número real `x` e devolve $p(x)$.

```python
def calcula_polinômio(p, x):
    resultado = 0
    for i in range(len(p)):
        resultado += p[i] * x**i
    return resultado
```

- **Usamos uma lista para criar uma abstração: polinômios**
- **Neste caso, nós iteramos por essa lista!**
  - Porque faz sentido iterar neste caso, ou seja, nesta abstração, não porque se trata de uma "lista de itens"

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`
  - (de trás para a frente não começa do zero!)

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`
  - (de trás para a frente não começa do zero!)
- `pedaço = lista[2:5]`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`
  - (de trás para a frente não começa do zero!)
- `pedaço = lista[2:5]`
  - mesmo formato de `range()`: `lista[início:final:passo]`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`
  - (de trás para a frente não começa do zero!)
- `pedaço = lista[2:5]`
  - mesmo formato de `range()`: `lista[início:final:passo]`
- `del l[3]`, `del l[2:5]`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`
  - (de trás para a frente não começa do zero!)
- `pedaço = lista[2:5]`
  - mesmo formato de `range()`: `lista[início:final:passo]`
- `del l[3]`, `del l[2:5]`
- `lista_com_tudo = lista1 + lista2`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`
  - (de trás para a frente não começa do zero!)
- `pedaço = lista[2:5]`
  - mesmo formato de `range()`: `lista[início:final:passo]`
- `del l[3]`, `del l[2:5]`
- `lista_com_tudo = lista1 + lista2`
- `váriosAs = ['A'] * 5`

Comandos básicos com listas

- `lista = [a, b, c]`
- `len(lista)`
- `lista.append(blah)`
- `lista[X]`
- `for item in lista:`
- `if item in lista:`
- `penúltimo = lista[-2]`
  - (de trás para a frente não começa do zero!)
- `pedaço = lista[2:5]`
  - mesmo formato de `range()`: `lista[início:final:passo]`
- `del l[3]`, `del l[2:5]`
- `lista_com_tudo = lista1 + lista2`
- `váriosAs = ['A'] * 5` → `['A', 'A', 'A', 'A', 'A']`

Extremos

Escreva uma função que recebe uma lista de números ordenada e devolve o menor e o maior números da lista

Extremos

Escreva uma função que recebe uma lista de números ordenada e devolve o menor e o maior números da lista

```
def extremos_da_lista(l):

```

Extremos

Escreva uma função que recebe uma lista de números ordenada e devolve o menor e o maior números da lista

```
def extremos_da_lista(l):
    return l[0]
```

Extremos

Escreva uma função que recebe uma lista de números ordenada e devolve o menor e o maior números da lista

```
def extremos_da_lista(l):
    return l[0],
```

Extremos

Escreva uma função que recebe uma lista de números ordenada e devolve o menor e o maior números da lista

```
def extremos_da_lista(l):
    return l[0], l[-1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):









        return l[primeiro_positivo:último_positivo]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):









        return l[primeiro_positivo:último_positivo +1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
    primeiro_positivo, último_positivo = -1, -1









        return l[primeiro_positivo:último_positivo +1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
    primeiro_positivo, último_positivo = -1, -1









    if primeiro_positivo < 0:


        return l[primeiro_positivo:último_positivo +1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
    primeiro_positivo, último_positivo = -1, -1









    if primeiro_positivo < 0:
        return []
    else:
        return l[primeiro_positivo:último_positivo +1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
    primeiro_positivo, último_positivo = -1, -1
    for i in range(len(l)):









    if primeiro_positivo < 0:
        return []
    else:
        return l[primeiro_positivo:último_positivo +1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
    primeiro_positivo, último_positivo = -1, -1
    for i in range(len(l)):
        if l[i] > 0:




    if primeiro_positivo < 0:
        return []
    else:
        return l[primeiro_positivo:último_positivo +1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
    primeiro_positivo, último_positivo = -1, -1
    for i in range(len(l)):
        if l[i] > 0:



            último_positivo = i
    if primeiro_positivo < 0:
        return []
    else:
        return l[primeiro_positivo:último_positivo +1]
```

Sequência de inteiros

Escreva uma função que recebe uma lista de números inteiros e devolve o pedaço da lista entre o primeiro e o último números estritamente positivos (inclusive)

```
def sequência_positivos(l):
    primeiro_positivo, último_positivo = -1, -1
    for i in range(len(l)):
        if l[i] > 0:
            if primeiro_positivo < 0:
                primeiro_positivo = i
            último_positivo = i
    if primeiro_positivo < 0:
        return []
    else:
        return l[primeiro_positivo:último_positivo +1]
```