MAC 115 — Introdução à Ciência da Computação

Aula 15

Nelson Lago

IF noturno – 2023

cc BY

Previously on MAC 115...

Oncotô?

Nós já podemos:

Oncotô?

Nós já podemos:

- **Interagir com o “mundo”**

Oncotô?

Nós já podemos:

- **Interagir com o "mundo"**
  - *Ler* dados — `input()`

Oncotô?

Nós já podemos:

- **Interagir com o "mundo"**
  - *Ler* dados — `input()`
  - Apresentar dados — `print()`

Oncotô?

Nós já podemos:

- **Interagir com o "mundo"**
  - *Ler* dados — `input()`
  - Apresentar dados — `print()`
    - » *Lembrando que dados podem ter* ***tipos*** *diferentes*

Oncotô?

Nós já podemos:

- **Interagir com o “mundo”**
  - *Ler* dados — `input()`
  - Apresentar dados — `print()`
    - » *Lembrando que dados podem ter **tipos** diferentes*
- **Guardar dados com um *nome* — variáveis**

Oncotô?

Nós já podemos:

- **Interagir com o "mundo"**
  - *Ler* dados — `input()`
  - Apresentar dados — `print()`
    - » *Lembrando que dados podem ter **tipos** diferentes*
- **Guardar dados com um *nome* — variáveis**
- **Processá-los — expressões como `+`, `-`, `format()` etc**

Oncotô?

Nós já podemos:

- **Interagir com o "mundo"**
  - *Ler* dados — `input()`
  - Apresentar dados — `print()`
    - » *Lembrando que dados podem ter **tipos** diferentes*
- **Guardar dados com um *nome* — variáveis**
- **Processá-los — expressões como `+`, `-`, `format()` etc**
  - Que podem ser *compostas* — (5 + 7) / 2

Oncotô?

Nós já podemos:

- **Interagir com o "mundo"**
  - *Ler* dados — `input()`
  - Apresentar dados — `print()`
    - » *Lembrando que dados podem ter **tipos** diferentes*
- **Guardar dados com um *nome* — variáveis**
- **Processá-los — expressões como `+`, `-`, `format()` etc**
  - Que podem ser *compostas* — (5 + 7) / 2
  - Tomando "decisões" — `if`

Oncotô?

Nós já podemos:

- **Interagir com o "mundo"**
  - *Ler* dados — `input()`
  - Apresentar dados — `print()`
    - » *Lembrando que dados podem ter **tipos** diferentes*
- **Guardar dados com um *nome* — variáveis**
- **Processá-los — expressões como `+`, `-`, `format()` etc**
  - Que podem ser *compostas* — (5 + 7) / 2
  - Tomando "decisões" — `if`
  - Fazendo repetições — `while`

Oncotô?

Também podemos:

Oncotô?

Também podemos:

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

Oncotô?

Também podemos:

- **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?

Também podemos:

- **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?

Também podemos:

- **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?

Também podemos:

- **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?

Também podemos:

- **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?

Também podemos:

- **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?

Também podemos:

- **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`**

O que está faltando?

O que está faltando?

Coleções

Coleções

Coleções

- **Lista de clientes**

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- ...

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- ...

Faz sentido inventar nomes (variáveis) para cada um deles?

Coleções

- **Lista de clientes**
  - E, por que não, "sub-listas", como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- ...

Faz sentido inventar nomes (variáveis) para cada um deles? (dica: não 😂)

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- …

Faz sentido inventar nomes (variáveis) para cada um deles? (dica: não 😂)

O que faz sentido é tratá-los como um *conjunto* que tem um nome

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- ...

Faz sentido inventar nomes (variáveis) para cada um deles? (dica: não 😂)

O que faz sentido é tratá-los como um *conjunto* que tem um nome

- **Podemos manipular o conjunto como um todo**

Coleções

- **Lista de clientes**
  - E, por que não, "sub-listas", como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- ...

Faz sentido inventar nomes (variáveis) para cada um deles? (dica: não 😂)

O que faz sentido é tratá-los como um *conjunto* que tem um nome

- **Podemos manipular o conjunto como um todo**
- **Podemos manipular subconjuntos**

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- …

Faz sentido inventar nomes (variáveis) para cada um deles? (dica: não 😂)

O que faz sentido é tratá-los como um *conjunto* que tem um nome

- **Podemos manipular o conjunto como um todo**
- **Podemos manipular subconjuntos**
- **Podemos manipular elementos um por um**

Coleções

- **Lista de clientes**
  - E, por que não, “sub-listas”, como a lista dos clientes que atendem a um dado critério (idade, cidade em que moram etc.)
- **Lista dos divisores de um número**
- ...

Faz sentido inventar nomes (variáveis) para cada um deles? (dica: não 😂)

O que faz sentido é tratá-los como um *conjunto* que tem um nome

- **Podemos manipular o conjunto como um todo**
- **Podemos manipular subconjuntos**
- **Podemos manipular elementos um por um**
- ...

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

Por que computação?

O computador é extremamente rápido, mas é uma ferramenta com o mesmo nível de "inteligência" que um martelo

Não é mais fácil fazer manualmente?

Repetições e condições

- **“Algo” precisa acontecer para indicar que as repetições chegaram ao fim**

    (ok, às vezes queremos repetir indefinidamente, mas vamos ignorar isso por enquanto)

- **As repetições são controladas por algum tipo de *condição* baseada no estado de uma *variável***

Partes mínimas de um laço

- **Um laço correto precisa**
  - Inicializar a variável de controle antes do início do laço
  - Verificar a condição adequada a cada iteração para que as repetições aconteçam o número correto de vezes
  - Alterar o valor da variável de acordo com a lógica do programa (no mínimo, na última iteração) para garantir que o laço termine

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

- **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*
     - » *...*

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

Repetições com coleções

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

- **Um laço correto precisa**
  - Inicializar a variável de controle antes do início do laço
  - Verificar a condição adequada a cada iteração para que as repetições aconteçam o número correto de vezes
  - Alterar o valor da variável de acordo com a lógica do programa (no mínimo, na última iteração) para garantir que o laço termine

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

```python
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
```

```python
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
```

Coleções

Dicas para o DJ:

Coleções

Dicas para o DJ:

Coleções

Dicas para o DJ:

```
sugestão = "nenhuma"
while sugestão != "":
  sugestão = input("Sugira uma canção para o DJ: ")
```

Coleções

Dicas para o DJ:

```
canções = []
sugestão = "nenhuma"
while sugestão != "":
  sugestão = input("Sugira uma canção para o DJ: ")
```

Coleções

Dicas para o DJ:

```
canções = []
sugestão = "nenhuma"
while sugestão != "":
  sugestão = input("Sugira uma canção para o DJ: ")
  if sugestão != "":
    canções.append(sugestão)
```

Coleções

Dicas para o DJ:

```
canções = []
sugestão = "nenhuma"
while sugestão != "":
  sugestão = input("Sugira uma canção para o DJ: ")
  if sugestão != "":
    canções.append(sugestão)
print("No total, foram sugeridas {} canções:".format(len(canções)))
print(canções)
```

Coleções

Dicas para o DJ:

```
canções = []
sugestão = "nenhuma"
while sugestão != "":
  sugestão = input("Sugira uma canção para o DJ: ")
  if sugestão != "":
    canções.append(sugestão)
print("No total, foram sugeridas {} canções:".format(len(canções)))
n = 0
while n < len(canções):
  print(canções[n])
  n += 1
```

Coleções

Dicas para o DJ:

```
canções = []
sugestão = "nenhuma"
while sugestão != "":
  sugestão = input("Sugira uma canção para o DJ: ")
  if sugestão != "":
    canções.append(sugestão)
print("No total, foram sugeridas {} canções:".format(len(canções)))
for canção in canções:
  print(canção)
```

Exercícios

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 = []
    for i in range(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 = []
    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

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
```

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

primeiro = início

A função `range()`

```
range(início, final, passo)
```

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)
```

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)

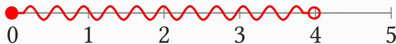
A função `range()`

`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)



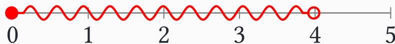
A função `range()`

`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)



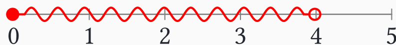
`range(2, 7)` → de dois a seis! (cinco elementos)

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)

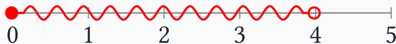
`range(1, 10, 2)` → de um a nove! (cinco elementos)

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 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, 2*n, 2):
        lista.append(i)
    return lista
```

Coleções — primos

Já criamos uma função que recebe um número natural e devolve **`True`** ou **`False`** indicando se o número é ou não é primo

Coleções — primos

Já criamos uma função que recebe um número natural e devolve **True** ou **False** indicando se o número é ou não é primo

```
def éPrimo(x):


    divisor = x -1
    primo = True
    while divisor >= 2:
        if x % divisor == 0:
            primo = False
        divisor -= 1
    return primo
```

Coleções — primos

Já criamos uma função que recebe um número natural e devolve **True** ou **False** indicando se o número é ou não é primo

```
def éPrimo(x):
    if x < 2:
        return False
    divisor = x -1
    primo = True
    while divisor >= 2:
        if x % divisor == 0:
            primo = False
        divisor -= 1
    return primo
```

Coleções

Escreva uma função que recebe uma lista de números naturais e devolve um booleano indicando se ao menos um deles é primo

Coleções

Escreva uma função que recebe uma lista de números naturais e devolve um booleano indicando se ao menos um deles é primo

```
def contémAlgumPrimo(l):
```

Coleções

Escreva uma função que recebe uma lista de números naturais e devolve um booleano indicando se ao menos um deles é primo

```
def contémAlgumPrimo(l):
    achei = False




    return achei
```

Coleções

Escreva uma função que recebe uma lista de números naturais e devolve um booleano indicando se ao menos um deles é primo

```
def contémAlgumPrimo(l):
    achei = False

        if éPrimo(n):
            achei = True
    return achei
```

Coleções

Escreva uma função que recebe uma lista de números naturais e devolve um booleano indicando se ao menos um deles é primo

```
def contémAlgumPrimo(l):
    achei = False
    for n in l:
        if éPrimo(n):
            achei = True
    return achei
```

Coleções

Escreva uma função que recebe uma lista de números naturais e devolve um booleano indicando se ao menos um deles é primo

```
def contémAlgumPrimo(l):

    for n in l:
        if éPrimo(n):
            return True
    return False
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

Coleções

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

```
def encontraPróximoPrimo(n):
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

```
def encontraPróximoPrimo(n):

  while     :
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

```
def encontraPróximoPrimo(n):

  while     :
      if éPrimo(n):
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

```
def encontraPróximoPrimo(n):

  while     :
      if éPrimo(n):
        return n
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

```
def encontraPróximoPrimo(n):

  while     :
      if éPrimo(n):
        return n
      else:
        n += 1
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

```
def encontraPróximoPrimo(n):

  while True:
      if éPrimo(n):
        return n
      else:
        n += 1
```

Escreva uma função que recebe n como parâmetro e devolve o primeiro primo maior que n.

```
def encontraPróximoPrimo(n):
  n += 1
  while True:
      if éPrimo(n):
        return n
      else:
        n += 1
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve uma coleção com os n primeiros números primos

Coleções

Escreva uma função que recebe n como parâmetro e devolve uma coleção com os n primeiros números primos

```
def primeirosPrimos(n):
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve uma coleção com os n primeiros números primos

```
def primeirosPrimos(n):
  primos = []





  return primos
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve uma coleção com os n primeiros números primos

```
def primeirosPrimos(n):
  primos = []

  while len(primos) < n:



  return primos
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve uma coleção com os n primeiros números primos

```
def primeirosPrimos(n):
  primos = []

  while len(primos) < n:


    primos.append(encontrado)
  return primos
```

Coleções

Escreva uma função que recebe n como parâmetro e devolve uma coleção com os n primeiros números primos

```
def primeirosPrimos(n):
  primos = []

  while len(primos) < n:
    encontrado = encontraPróximoPrimo(encontrado)
    primos.append(encontrado)
  return primos
```

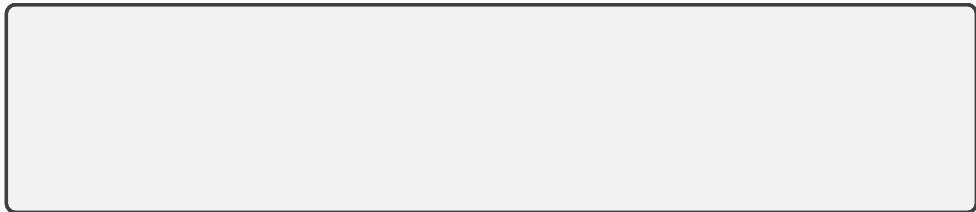
Coleções

Escreva uma função que recebe n como parâmetro e devolve uma coleção com os n primeiros números primos

```
def primeirosPrimos(n):
  primos = []
  encontrado = 0
  while len(primos) < n:
    encontrado = encontraPróximoPrimo(encontrado)
    primos.append(encontrado)
  return primos
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior valor da lista



Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior valor da lista

```
def encontraMaior(lista):
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior valor da lista

```
def encontraMaior(lista):




    return maior
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior valor da lista

```
def encontraMaior(lista):

    for n in lista:



    return maior
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior valor da lista

```
def encontraMaior(lista):

    for n in lista:
        if n > maior:
            maior = n
    return maior
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior valor da lista

```
def encontraMaior(lista):
    maior = lista[0]
    for n in lista:
        if n > maior:
            maior = n
    return maior
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior *e o menor* valores da lista

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior *e o menor* valores da lista

```
def encontraMaior(lista):
    maior = lista[0]

    for n in lista:
        if n > maior:
            maior = n



    return maior
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior *e o menor* valores da lista

```
def encontraMaior(lista):
    maior = lista[0]

    for n in lista:
        if n > maior:
            maior = n



    return maior, menor
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior *e o menor* valores da lista

```
def encontraMaior(lista):
    maior = lista[0]
    menor = lista[0]
    for n in lista:
        if n > maior:
            maior = n



    return maior, menor
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve o maior *e o menor* valores da lista

```python
def encontraMaior(lista):
    maior = lista[0]
    menor = lista[0]
    for n in lista:
        if n > maior:
            maior = n
        if n < menor:
            menor = n
    return maior, menor
```

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, -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
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve a soma de seus elementos

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve a soma de seus elementos

```
def soma_elementos(lista):
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve a soma de seus elementos

```
def soma_elementos(lista):
    soma = 0
    for i in lista:

    return soma
```

Coleções

Escreva uma função que recebe uma lista de inteiros e devolve a soma de seus elementos

```python
def soma_elementos(lista):
    soma = 0
    for i in lista:
        soma += i
    return soma
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

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

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []

    return lista
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0











    return lista
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0
    while i < len(l1) and j < len(l2):









    return lista
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0
    while i < len(l1) and j < len(l2):
        if l1[i] < l2[j]:
            lista.append(l1[i])
            i += 1









    return lista
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0
    while i < len(l1) and j < len(l2):
        if l1[i] < l2[j]:
            lista.append(l1[i])
            i += 1
        else:








    return lista
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0
    while i < len(l1) and j < len(l2):
        if l1[i] < l2[j]:
            lista.append(l1[i])
            i += 1
        else:
            lista.append(l1[j])
            j += 1







    return lista
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0
    while i < len(l1) and j < len(l2):
        if l1[i] < l2[j]:
            lista.append(l1[i])
            i += 1
        else:
            lista.append(l1[j])
            j += 1
    while i < len(l1):
        lista.append(l1[i])
        i += 1



    return lista
```

Coleções

Escreva uma função que recebe duas listas de números ordenadas e devolve uma lista com a união das duas listas, também ordenada

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0
    while i < len(l1) and j < len(l2):
        if l1[i] < l2[j]:
            lista.append(l1[i])
            i += 1
        else:
            lista.append(l1[j])
            j += 1
    while i < len(l1):
        lista.append(l1[i])
        i += 1
    while j < len(l2):
        lista.append(l2[j])
        j += 1
    return lista
```