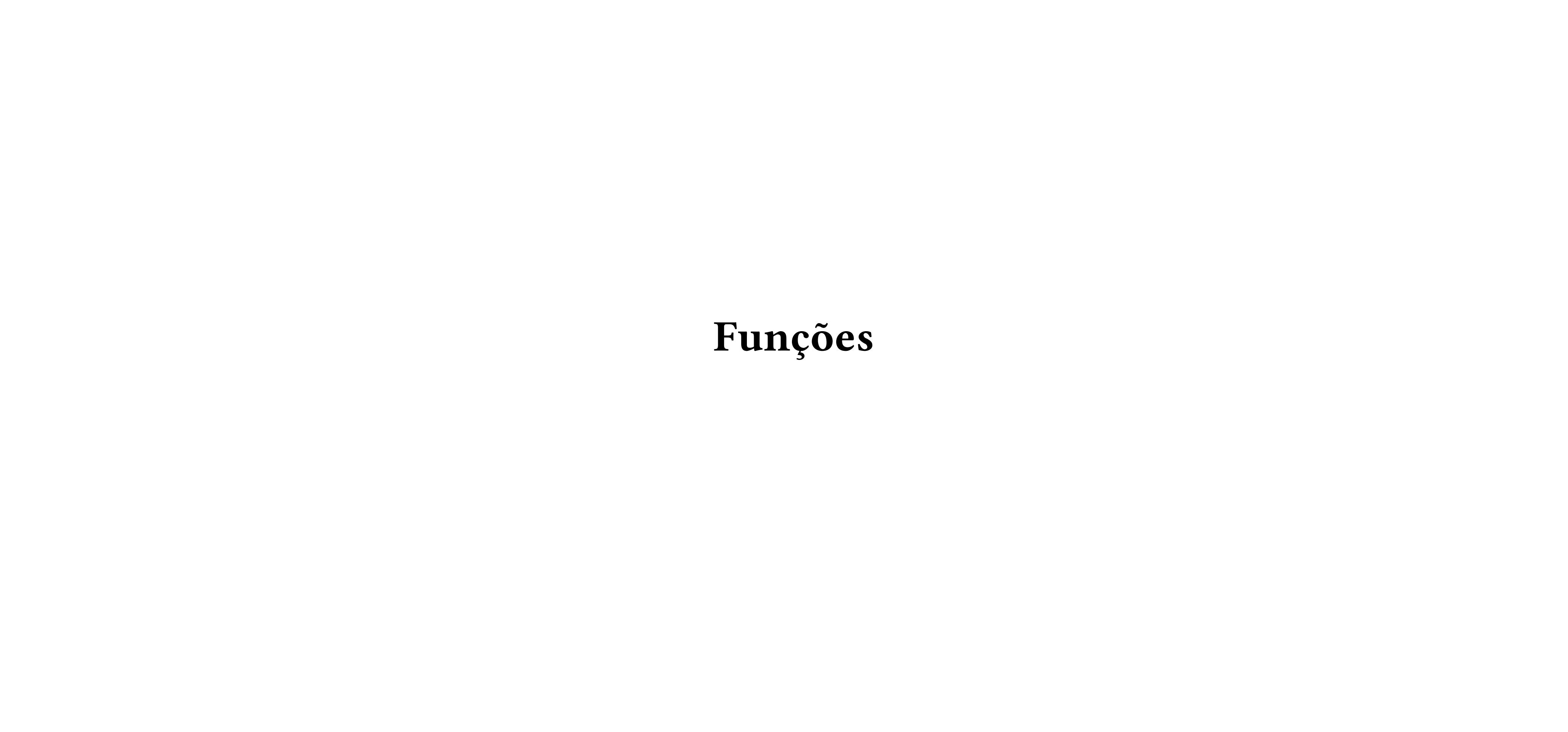
Funções, Bibliotecas e Exceções

MC102



Funções

Na Matemática vs Programação

Na Matemática: o que é uma função?

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = x_1 + x_2$

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = x_1 + x_2$
- Nem y precisa ser um único valor:

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = x_1 + x_2$
- Nem y precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = (2x_1, 3x_2)$

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = x_1 + x_2$
- Nem y precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = (2x_1, 3x_2)$
  - Mesmo assim, no final das contas, o resultado é um **único** vetor.

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = x_1 + x_2$
- Nem y precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = (2x_1, 3x_2)$
  - Mesmo assim, no final das contas, o resultado é um **único** vetor.

Informalmente, f nos diz como calcular $y = f(x)$ a partir de x.

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = x_1 + x_2$
- Nem y precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = (2x_1, 3x_2)$
  - Mesmo assim, no final das contas, o resultado é um **único** vetor.

Informalmente, f nos diz como calcular $y = f(x)$ a partir de x.

- Como obter uma saída a partir de uma entrada

Na Matemática vs Programação

Na Matemática: o que é uma função?

- Ex.: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$.
- É uma relação entre dois conjuntos X e Y.
- Para cada entrada x, temos uma única saída $y = f(x)$.
- Escrevemos: $f : X \to Y$
- Note que x não precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = x_1 + x_2$
- Nem y precisa ser um único valor:
  - Ex.: $f(x_1, x_2) = (2x_1, 3x_2)$
  - Mesmo assim, no final das contas, o resultado é um **único** vetor.

Informalmente, f nos diz como calcular $y = f(x)$ a partir de x.

- Como obter uma saída a partir de uma entrada

Na Matemática vs Programação

Na Programação: É um bloco de código que:

Na Matemática vs Programação

Na Programação: É um bloco de código que:

- recebe **parâmetros** (entrada),

Na Matemática vs Programação

Na Programação: É um bloco de código que:

- recebe **parâmetros** (entrada),
- executa instruções

Na Matemática vs Programação

Na Programação: É um bloco de código que:

$$f(x) = |x|$$

- recebe **parâmetros** (entrada),
- executa instruções
- e devolve uma **resposta** (saída).

Na Matemática vs Programação

Na Programação: É um bloco de código que:

- recebe **parâmetros** (entrada),
- executa instruções
- e devolve uma **resposta** (saída).

$$f(x) = |x|$$

```
1 Absoluto(x):
2     Se x >= 0:
3         Devolva x
4     Senão:
5         Devolva -x
```

Na Matemática vs Programação

Na Programação: É um bloco de código que:

- recebe **parâmetros** (entrada),
- executa instruções
- e devolve uma **resposta** (saída).

$f(x) = |x|$

```
Absoluto(x):
    Se x >= 0:
        Devolva x
    Senão:
        Devolva -x
```

$f(x) =$ soma dos dígitos na base 10 de x

Na Matemática vs Programação

Na Programação: É um bloco de código que:

- recebe **parâmetros** (entrada),
- executa instruções
- e devolve uma **resposta** (saída).

$f(x) = |x|$

```
Absoluto(x):
    Se x >= 0:
        Devolva x
    Senão:
        Devolva -x
```

$f(x) =$ soma dos dígitos na base 10 de x

```
Soma_dos_digitos(x):
    soma = 0
    Enquanto x > 0:
        soma = soma + x % 10
        x = x // 10  # divisão inteira
    Devolva soma
```

Na Matemática vs Programação

Na Programação: É um bloco de código que:
- recebe **parâmetros** (entrada),
- executa instruções
- e devolve uma **resposta** (saída).

$f(x) = |x|$

```
Absoluto(x):
    Se x >= 0:
        Devolva x
    Senão:
        Devolva -x
```

$f(x) = \text{soma dos dígitos na base 10 de x}$

```
Soma_dos_digitos(x):
    soma = 0
    Enquanto x > 0:
        soma = soma + x % 10
        x = x // 10  # divisão inteira
    Devolva soma
```

$f(x) = x^2$

Na Matemática vs Programação

Na Programação: É um bloco de código que:

- recebe **parâmetros** (entrada),
- executa instruções
- e devolve uma **resposta** (saída).

$f(x) = |x|$

```
Absoluto(x):
    Se x >= 0:
        Devolva x
    Senão:
        Devolva -x
```

$f(x) =$ soma dos dígitos na base 10 de x

```
Soma_dos_digitos(x):
    soma = 0
    Enquanto x > 0:
        soma = soma + x % 10
        x = x // 10  # divisão inteira
    Devolva soma
```

$f(x) = x^2$

```
Quadrado(x):
    Devolva x * x

Leia n
Imprima Quadrado(x)
```

Na Matemática vs Programação

Agora, em Python!

Na Matemática vs Programação

Agora, em Python!

$$f(x) = |x|$$

```
1 def absoluto(x):
2     if x >= 0:
3         return x
4     else:
5         return -x
```

Na Matemática vs Programação

Agora, em Python!

$$f(x) = |x|$$

```python
def absoluto(x):
    if x >= 0:
        return x
    else:
        return -x
```

$$f(x) = \text{soma dos dígitos na base 10 de x}$$

```python
def soma_dos_digitos(x):
    soma = 0
    while x > 0:
        soma = soma + x % 10
        x = x // 10
    return soma
```

Na Matemática vs Programação

Agora, em Python!

$f(x) = |x|$

```
1  def absoluto(x):
2      if x >= 0:
3          return x
4      else:
5          return -x
```

$f(x) =$ soma dos dígitos na base 10 de x

```
1  def soma_dos_digitos(x):
2      soma = 0
3      while x > 0:
4          soma = soma + x % 10
5          x = x // 10
6      return soma
```

$f(x) = x^2$

```
1  def quadrado(x):
2      return x * x  # ou x ** 2
3
4  n = int(input())
5  print(quadrado(n))
```

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

```
def nome_da_funcao(parametro_1, parametro_2):
    # instruções para computar o resultado
    resultado = parametro_1 + parametro_2
    return resultado
```

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

```
1 def nome_da_funcao(parametro_1, parametro_2):
2     # instruções para computar o resultado
3     resultado = parametro_1 + parametro_2
4     return resultado
```

Uma vez definida, podemos **chamar** a função passando valores (argumentos).

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

```
def nome_da_funcao(parametro_1, parametro_2):
    # instruções para computar o resultado
    resultado = parametro_1 + parametro_2
    return resultado
```

Uma vez definida, podemos **chamar** a função passando valores (argumentos). A execução pausa onde estava, vai para a função, calcula, e traz a resposta de volta.

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

```python
def nome_da_funcao(parametro_1, parametro_2):
    # instruções para computar o resultado
    resultado = parametro_1 + parametro_2
    return resultado
```

Uma vez definida, podemos **chamar** a função passando valores (argumentos). A execução pausa onde estava, vai para a função, calcula, e traz a resposta de volta.

```python
x = nome_da_funcao(10, 5)
print(x)
```

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

```
def nome_da_funcao(parametro_1, parametro_2):
    # instruções para computar o resultado
    resultado = parametro_1 + parametro_2
    return resultado
```

Uma vez definida, podemos **chamar** a função passando valores (argumentos). A execução pausa onde estava, vai para a função, calcula, e traz a resposta de volta.

```
x = nome_da_funcao(10, 5)
print(x)
```

```
15
```

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

```
def nome_da_funcao(parametro_1, parametro_2):
    # instruções para computar o resultado
    resultado = parametro_1 + parametro_2
    return resultado
```

Uma vez definida, podemos **chamar** a função passando valores (argumentos). A execução pausa onde estava, vai para a função, calcula, e traz a resposta de volta.

```
x = nome_da_funcao(10, 5)
print(x)
```

```
15
```

Observações:

- O valor passado para o parâmetro pode vir de uma constante, variável ou expressão

Anatomia de uma Função em Python

Para informar ao Python que uma função existe, nós a **definimos** usando a palavra-chave `def`.

```python
def nome_da_funcao(parametro_1, parametro_2):
    # instruções para computar o resultado
    resultado = parametro_1 + parametro_2
    return resultado
```

Uma vez definida, podemos **chamar** a função passando valores (argumentos). A execução pausa onde estava, vai para a função, calcula, e traz a resposta de volta.

```python
x = nome_da_funcao(10, 5)
print(x)
```

```
15
```

Observações:

- O valor passado para o parâmetro pode vir de uma constante, variável ou expressão
- A ordem importa!

Exemplo completo

```
1  def le_lista(n):
2      lista = []
3      for _ in range(n):
4          a = float(input())
5          lista.append(a)
6      return lista
7
8  def soma_valores(lista):
9      soma = 0
10     for x in lista:
11         soma += x
12     return soma
13
14 n = int(input())
15 lista = le_lista(n)
16 print(soma_valores(lista))
```

Nas linhas:

- 1 a 6, definimos uma função chamada `le_lista`

Exemplo completo

```
1  def le_lista(n):
2      lista = []
3      for _ in range(n):
4          a = float(input())
5          lista.append(a)
6      return lista
7
8  def soma_valores(lista):
9      soma = 0
10     for x in lista:
11         soma += x
12     return soma
13
14 n = int(input())
15 lista = le_lista(n)
16 print(soma_valores(lista))
```

Nas linhas:

- 1 a 6, definimos uma função chamada `le_lista`
- 8 a 12, uma função chamada `soma_valores`

Exemplo completo

```python
def le_lista(n):
    lista = []
    for _ in range(n):
        a = float(input())
        lista.append(a)
    return lista

def soma_valores(lista):
    soma = 0
    for x in lista:
        soma += x
    return soma

n = int(input())
lista = le_lista(n)
print(soma_valores(lista))
```

Nas linhas:

- 1 a 6, definimos uma função chamada `le_lista`
- 8 a 12, uma função chamada `soma_valores`
- 14 a 16, chamamos tais funções para ler e somar os números

Por que usar funções?

Usamos funções para

Por que usar funções?

Usamos funções para

- Evitar repetição de código

Por que usar funções?

Usamos funções para

- Evitar repetição de código
- Reutilizar código de outras pessoas (ou os nossos)

Por que usar funções?

Usamos funções para

- Evitar repetição de código
- Reutilizar código de outras pessoas (ou os nossos)
- Permitir que outros utilizem o nosso código

Por que usar funções?

Usamos funções para

- Evitar repetição de código
- Reutilizar código de outras pessoas (ou os nossos)
- Permitir que outros utilizem o nosso código
- Deixar o programa mais fácil de entender

Por que usar funções?

Usamos funções para

- Evitar repetição de código
- Reutilizar código de outras pessoas (ou os nossos)
- Permitir que outros utilizem o nosso código
- Deixar o programa mais fácil de entender
- Deixar o programa mais fácil de *debuggar*

Por que usar funções?

Usamos funções para

- Evitar repetição de código
- Reutilizar código de outras pessoas (ou os nossos)
- Permitir que outros utilizem o nosso código
- Deixar o programa mais fácil de entender
- Deixar o programa mais fácil de *debuggar*
- Criar conjuntos de funções (bibliotecas) úteis

Por que usar funções?

Usamos funções para

- Evitar repetição de código
- Reutilizar código de outras pessoas (ou os nossos)
- Permitir que outros utilizem o nosso código
- Deixar o programa mais fácil de entender
- Deixar o programa mais fácil de *debuggar*
- Criar conjuntos de funções (bibliotecas) úteis

Dividir para conquistar

Quando o problema é grande, a melhor estratégia é separar em partes menores e mais “mastigáveis”.

Dividir para conquistar

Quando o problema é grande, a melhor estratégia é separar em partes menores e mais “mastigáveis”.

- Cada função resolve uma tarefa específica

Dividir para conquistar

Quando o problema é grande, a melhor estratégia é separar em partes menores e mais “mastigáveis”.

- Cada função resolve uma tarefa específica
- O código fica mais simples de testar

Dividir para conquistar

Quando o problema é grande, a melhor estratégia é separar em partes menores e mais “mastigáveis”.

- Cada função resolve uma tarefa específica
- O código fica mais simples de testar
- Fica mais fácil trocar uma parte sem quebrar o resto

Dividir para conquistar

Quando o problema é grande, a melhor estratégia é separar em partes menores e mais "mastigáveis".

- Cada função resolve uma tarefa específica
- O código fica mais simples de testar
- Fica mais fácil trocar uma parte sem quebrar o resto

```
1 def processa_turma(notas):
2   validas = filtra_notas_validas(notas)
3   media = calcula_media(validas)
4   situacao = classifica(media)
5   return situacao
```

Dividir para conquistar

Quando o problema é grande, a melhor estratégia é separar em partes menores e mais “mastigáveis”.

- Cada função resolve uma tarefa específica
- O código fica mais simples de testar
- Fica mais fácil trocar uma parte sem quebrar o resto

```
1 def processa_turma(notas):
2   validas = filtra_notas_validas(notas)
3   media = calcula_media(validas)
4   situacao = classifica(media)
5   return situacao
```

Em vez de um blocão com tudo junto, temos uma sequência clara de passos.

Exemplo

```python
def eh_primo(p):
    k = 2
    while k * k < p:
        if p % k == 0:
            return False
        k += 1
    return True

p = int(input())
q = int(input())

if eh_primo(p) and eh_primo(q):
    print("Ambos são primos!")
else:
    print("Pelo menos um não é primo.")
```

Vantagens:

- Podemos chamar a função várias vezes

Exemplo

```python
def eh_primo(p):
    k = 2
    while k * k < p:
        if p % k == 0:
            return False
        k += 1
    return True

p = int(input())
q = int(input())

if eh_primo(p) and eh_primo(q):
    print("Ambos são primos!")
else:
    print("Pelo menos um não é primo.")
```

Vantagens:

- Podemos chamar a função várias vezes
- Outra pessoa poderia usar a função `primo`

Exemplo

```python
def eh_primo(p):
    k = 2
    while k * k < p:
        if p % k == 0:
            return False
        k += 1
    return True

p = int(input())
q = int(input())

if eh_primo(p) and eh_primo(q):
    print("Ambos são primos!")
else:
    print("Pelo menos um não é primo.")
```

Vantagens:

- Podemos chamar a função várias vezes
- Outra pessoa poderia usar a função `primo`
- Outra pessoa poderia implementar a função

Exemplo

```
1  def eh_primo(p):
2      k = 2
3      while k * k < p:
4          if p % k == 0:
5              return False
6          k += 1
7      return True
8
9  p = int(input())
10 q = int(input())
11
12 if eh_primo(p) and eh_primo(q):
13     print("Ambos são primos!")
14 else:
15     print("Pelo menos um não é primo.")
```

Vantagens:

- Podemos chamar a função várias vezes
- Outra pessoa poderia usar a função `primo`
- Outra pessoa poderia implementar a função
- O código é mais fácil de ler!

O `NoneType`

E se esquecermos do `return` ou criarmos uma função que só imprime coisas?

O `NoneType`

E se esquecermos do `return` ou criarmos uma função que só imprime coisas?

```python
def imprime_lista(lista):
    for x in lista:
        print(x)

valor = imprime_lista([1, 2])
print(valor)
```

```
1
2
None
```

O `NoneType`

E se esquecermos do `return` ou criarmos uma função que só imprime coisas?

```python
def imprime_lista(lista):
    for x in lista:
        print(x)

valor = imprime_lista([1, 2])
print(valor)
```

```
1
2
None
```

Toda função em Python devolve **algum** valor. Se não há `return`, ela devolve `None`.

O `NoneType`

E se esquecermos do `return` ou criarmos uma função que só imprime coisas?

```
def imprime_lista(lista):
    for x in lista:
        print(x)

valor = imprime_lista([1, 2])
print(valor)
```

```
1
2
None
```

Toda função em Python devolve **algum** valor. Se não há `return`, ela devolve `None`. O tipo `NoneType` representa o "nada", a ausência de um valor válido.

O NoneType

E se esquecermos do return ou criarmos uma função que só imprime coisas?

```python
def imprime_lista(lista):
    for x in lista:
        print(x)

valor = imprime_lista([1, 2])
print(valor)
```

```
1
2
None
```

Toda função em Python devolve **algum** valor. Se não há return, ela devolve None. O tipo NoneType representa o "nada", a ausência de um valor válido.

As vezes, utilizamos o None como um valor especial, simbolizando o "nada".

O `NoneType`

E se esquecermos do `return` ou criarmos uma função que só imprime coisas?

```python
def imprime_lista(lista):
    for x in lista:
        print(x)

valor = imprime_lista([1, 2])
print(valor)
```

```
1
2
None
```

Toda função em Python devolve **algum** valor. Se não há `return`, ela devolve `None`. O tipo `NoneType` representa o "nada", a ausência de um valor válido.

As vezes, utilizamos o `None` como um valor especial, simbolizando o "nada".

```python
a = None
```

Valores padrão

Valores padrão

Parâmetros podem ter um valor padrão, usado quando não passamos aquele argumento na chamada.

Valores padrão

Parâmetros podem ter um valor padrão, usado quando não passamos aquele argumento na chamada.

```python
def saudacao(nome, mensagem="Olá"):
    print(mensagem, nome)

saudacao("Ana")
saudacao("Bia", "Oi")
```

```
Olá Ana
Oi Bia
```

Valores padrão

Parâmetros podem ter um valor padrão, usado quando não passamos aquele argumento na chamada.

```
1 def saudacao(nome, mensagem="Olá"):
2     print(mensagem, nome)
3 
4 saudacao("Ana")
5 saudacao("Bia", "Oi")
```

```
Olá Ana
Oi Bia
```

Na definição, escrevemos `parametro=valor` depois dos parâmetros obrigatórios.

Valores padrão

Parâmetros podem ter um valor padrão, usado quando não passamos aquele argumento na chamada.

```python
def saudacao(nome, mensagem="Olá"):
    print(mensagem, nome)

saudacao("Ana")
saudacao("Bia", "Oi")
```

```
Olá Ana
Oi Bia
```

Na definição, escrevemos `parametro=valor` depois dos parâmetros obrigatórios.

- Se o argumento não for informado, a função usa o valor padrão

Valores padrão

Parâmetros podem ter um valor padrão, usado quando não passamos aquele argumento na chamada.

```python
def saudacao(nome, mensagem="Olá"):
    print(mensagem, nome)

saudacao("Ana")
saudacao("Bia", "Oi")
```

```
Olá Ana
Oi Bia
```

Na definição, escrevemos `parametro=valor` depois dos parâmetros obrigatórios.

- Se o argumento não for informado, a função usa o valor padrão
- Se ele for informado, o valor passado na chamada tem prioridade

Valores padrão

Parâmetros podem ter um valor padrão, usado quando não passamos aquele argumento na chamada.

```python
def saudacao(nome, mensagem="Olá"):
    print(mensagem, nome)

saudacao("Ana")
saudacao("Bia", "Oi")
```

```
Olá Ana
Oi Bia
```

Na definição, escrevemos `parametro=valor` depois dos parâmetros obrigatórios.

- Se o argumento não for informado, a função usa o valor padrão
- Se ele for informado, o valor passado na chamada tem prioridade
- Parâmetros com valor padrão devem vir depois dos obrigatórios

Valores padrão

Também podemos misturar argumentos por posição e por nome:

Valores padrão

Também podemos misturar argumentos por posição e por nome:

```
1 def potencia(base, expoente=2):
2     return base ** expoente
3
4 print(potencia(5))
5 print(potencia(5, 3))
6 print(potencia(expoente=4, base=2))
```

`potencia(5)` usa `expoente=2`.

Valores padrão

Também podemos misturar argumentos por posição e por nome:

```
def potencia(base, expoente=2):
    return base ** expoente

print(potencia(5))
print(potencia(5, 3))
print(potencia(expoente=4, base=2))
```

`potencia(5)` usa `expoente=2`.

Em `potencia(5, 3)`, os valores são associados pela ordem.

Valores padrão

Também podemos misturar argumentos por posição e por nome:

```
def potencia(base, expoente=2):
    return base ** expoente

print(potencia(5))
print(potencia(5, 3))
print(potencia(expoente=4, base=2))
```

`potencia(5)` usa `expoente=2`.

Em `potencia(5, 3)`, os valores são associados pela ordem.

Em `potencia(expoente=4, base=2)`, usamos os nomes dos parâmetros, então a ordem deixa de importar.

Valores padrão

Também podemos misturar argumentos por posição e por nome:

```
def potencia(base, expoente=2):
    return base ** expoente

print(potencia(5))
print(potencia(5, 3))
print(potencia(expoente=4, base=2))
```

`potencia(5)` usa `expoente=2`.

Em `potencia(5, 3)`, os valores são associados pela ordem.

Em `potencia(expoente=4, base=2)`, usamos os nomes dos parâmetros, então a ordem deixa de importar.

Quando misturamos os dois estilos, os argumentos por posição devem vir primeiro e os por nome depois.

Boas Práticas com Funções

Type hinting

Quando começamos a ver variáveis em python, eu comentei que *valores possuem tipos*.

Type hinting

Quando começamos a ver variáveis em python, eu comentei que *valores possuem tipos*. Ou seja, uma variável pode assumir qualquer valor, independente do tipo dele.

Type hinting

Quando começamos a ver variáveis em python, eu comentei que *valores possuem tipos*. Ou seja, uma variável pode assumir qualquer valor, independente do tipo dele.

Em outras linguagens de programação, como C, isso não é verdade:

Type hinting

Quando começamos a ver variáveis em python, eu comentei que *valores possuem tipos*. Ou seja, uma variável pode assumir qualquer valor, independente do tipo dele.

Em outras linguagens de programação, como C, isso não é verdade:

```
int a = 1;
```

Nela, precisamos dizer qual o tipo da variável e, portanto, ela só irá aceitar valores daquele tipo.

Type hinting

Quando começamos a ver variáveis em python, eu comentei que *valores possuem tipos*. Ou seja, uma variável pode assumir qualquer valor, independente do tipo dele.

Em outras linguagens de programação, como C, isso não é verdade:

```
int a = 1;
```

Nela, precisamos dizer qual o tipo da variável e, portanto, ela só irá aceitar valores daquele tipo.

Type hinting

Em python, podemos fazer algo similar:

Type hinting

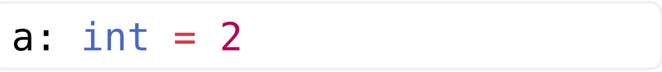
Em python, podemos fazer algo similar:

```
a: int = 2
```

Nesse caso, estamos indicando que a variável `a` só receberá valores do tipo `int` e começará com o `2`.

Type hinting

Em python, podemos fazer algo similar:



```python
a: int = 2
```

Nesse caso, estamos indicando que a variável `a` só receberá valores do tipo `int` e começará com o `2`.

Isso é chamado de **type hinting**.

Type hinting

Em python, podemos fazer algo similar:

```python
a: int = 2
```

Nesse caso, estamos indicando que a variável `a` só receberá valores do tipo `int` e começará com o `2`.

Isso é chamado de **type hinting**.

Como o nome diz, é uma "dica", então algumas versões de python não vão se importar se você desobedecer.

Type hinting

Em python, podemos fazer algo similar:

```python
a: int = 2
```

Nesse caso, estamos indicando que a variável `a` só receberá valores do tipo `int` e começará com o `2`.

Isso é chamado de **type hinting**.

Como o nome diz, é uma "dica", então algumas versões de python não vão se importar se você desobedecer.

"*Ai, professor, mas se não precisa, para quê serve?* – diz o aluno…"

Type hinting

Em python, podemos fazer algo similar:

```python
a: int = 2
```

Nesse caso, estamos indicando que a variável `a` só receberá valores do tipo `int` e começará com o `2`.

Isso é chamado de **type hinting**.

Como o nome diz, é uma “dica”, então algumas versões de python não vão se importar se você desobedecer.

“*Ai, professor, mas se não precisa, para quê serve?* – diz o aluno...”

Para impedir que cometamos erros e evitar bugs!!!

```python
a: int = 10
if a != 20:
    a = 3.14
lista[a]
```

Type hinting: funções

No caso de funções, pode ser ainda mais útil!

Type hinting: funções

No caso de funções, pode ser ainda mais útil!

```
def le_idade() -> int:
    a = int(input())
    return a
```

O `->` indica o tipo do `return` da função. Caso você tente retornar algo de outro tipo, ele vai avisar!

Type hinting: funções

No caso de funções, pode ser ainda mais útil!

```python
def le_idade() -> int:
    a = int(input())
    return a
```

```python
def soma(a: int, b: float) -> float:
    return a + b
```

O `->` indica o tipo do `return` da função. Caso você tente retornar algo de outro tipo, ele vai avisar!

Para cada parâmetro, também podemos especificar qual o tipo esperado. Isso ajuda na hora de chamar a função e garantir que não estamos errando a ordem do parâmetros.

Type hinting: funções

No caso de funções, pode ser ainda mais útil!

```python
def le_idade() -> int:
    a = int(input())
    return a
```

O `->` indica o tipo do `return` da função. Caso você tente retornar algo de outro tipo, ele vai avisar!

```python
def soma(a: int, b: float) -> float:
    return a + b
```

Para cada parâmetro, também podemos especificar qual o tipo esperado. Isso ajuda na hora de chamar a função e garantir que não estamos errando a ordem do parâmetros.

```python
def media(notas: list[float]) -> float:
    return sum(notas) / len(notas)
```

Um pouco mais avançado, mas para tipos compostos de dados (listas, tuplas, dicionários, conjuntos…) podemos especificar não só o tipo, mas também o que vai dentro dele.

Docstrings: como documentar uma função

```python
def soma_dos_digitos(x: int) -> int:
  """Calcula a soma dos dígitos de x na base 10.

  Considera x como inteiro não negativo.

  Args:
    x: Número inteiro não negativo.

  Returns:
    Soma dos dígitos de x.
  """
  soma = 0
  while x > 0:
    soma += x % 10
    x = 10
  return soma
```

Docstring é a string na primeira linha da função, entre `"""`.

Docstrings: como documentar uma função

```python
def soma_dos_digitos(x: int) -> int:
  """Calcula a soma dos dígitos de x na base 10.

  Considera x como inteiro não negativo.

  Args:
    x: Número inteiro não negativo.

  Returns:
    Soma dos dígitos de x.
  """
  soma = 0
  while x > 0:
    soma += x % 10
    x = 10
  return soma
```

Docstring é a string na primeira linha da função, entre `"""`.

- Diga em 1 linha o que a função faz

Docstrings: como documentar uma função

```
1  def soma_dos_digitos(x: int) -> int:
2    """Calcula a soma dos dígitos de x na base 10.
3
4    Considera x como inteiro não negativo.
5
6    Args:
7      x: Número inteiro não negativo.
8
9    Returns:
10     Soma dos dígitos de x.
11   """
12   soma = 0
13   while x > 0:
14     soma += x % 10
15     x = 10
16   return soma
```

Docstring é a string na primeira linha da função, entre `"""`.

- Diga em 1 linha o que a função faz
- Em docstrings longas, deixe uma linha em branco após o resumo

Docstrings: como documentar uma função

```python
def soma_dos_digitos(x: int) -> int:
  """Calcula a soma dos dígitos de x na base 10.

  Considera x como inteiro não negativo.

  Args:
    x: Número inteiro não negativo.

  Returns:
    Soma dos dígitos de x.
  """
  soma = 0
  while x > 0:
    soma += x % 10
    x = 10
  return soma
```

Docstring é a string na primeira linha da função, entre `"""`.

- Diga em 1 linha o que a função faz
- Em docstrings longas, deixe uma linha em branco após o resumo
- Documente parâmetros e retorno

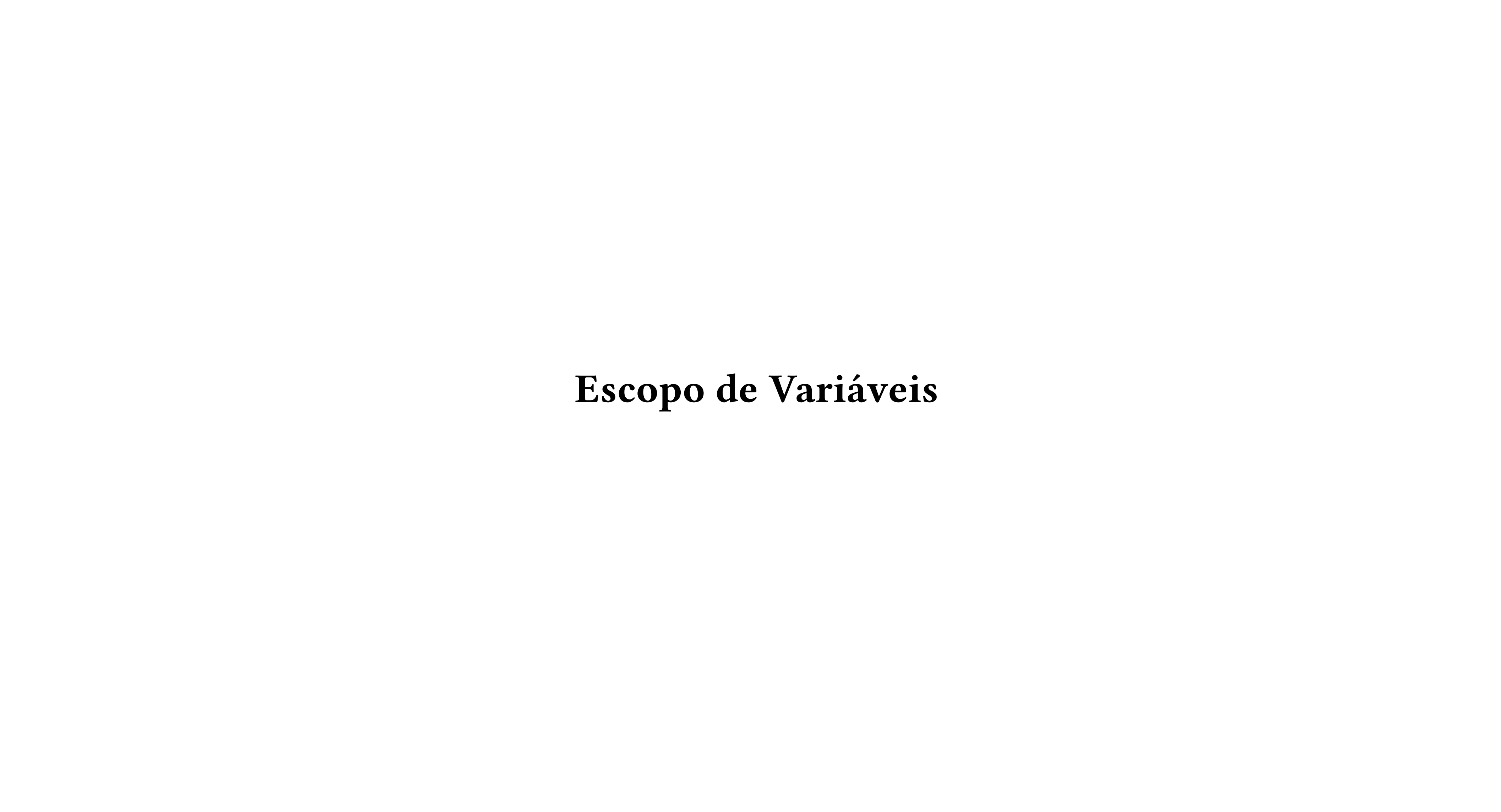
Docstrings: como documentar uma função

```python
def soma_dos_digitos(x: int) -> int:
  """Calcula a soma dos dígitos de x na base 10.

  Considera x como inteiro não negativo.

  Args:
    x: Número inteiro não negativo.

  Returns:
    Soma dos dígitos de x.
  """
  soma = 0
  while x > 0:
    soma += x % 10
    x = 10
  return soma
```

Docstring é a string na primeira linha da função, entre `"""`.

- Diga em 1 linha o que a função faz
- Em docstrings longas, deixe uma linha em branco após o resumo
- Documente parâmetros e retorno
- Use `help(nome_da_funcao)` para consultar rapidamente

```
>>> help(soma_dos_digitos)
Help on function soma_dos_digitos:
Calcula a soma dos dígitos de x na base
10.
```


Escopo de Variáveis

Variáveis Locais

Variáveis criadas **dentro** de uma função só existem lá dentro. Chamamos isso de **Escopo Local**.

Variáveis Locais

Variáveis criadas **dentro** de uma função só existem lá dentro. Chamamos isso de **Escopo Local**.

```python
def soma_um(x):
    y = x + 1
    return y

z = soma_um(10)
print(z)
print(y)
```

```
Traceback (most recent call last):
  File "arquivo.py", line 7, in <module>
    print(y)
NameError: name 'y' is not defined
```

Variáveis Locais

Variáveis criadas **dentro** de uma função só existem lá dentro. Chamamos isso de **Escopo Local**.

```python
def soma_um(x):
    y = x + 1
    return y

z = soma_um(10)
print(z)
print(y)
```

```
Traceback (most recent call last):
  File "arquivo.py", line 7, in <module>
    print(y)
NameError: name 'y' is not defined
```

O `y` morre assim que a função termina de rodar! 💀

Variáveis Globais

Variáveis criadas **fora** de qualquer função pertencem ao **Escopo Global** e podem ser acessadas por todo mundo.

Variáveis Globais

Variáveis criadas **fora** de qualquer função pertencem ao **Escopo Global** e podem ser acessadas por todo mundo.

```python
z = 10
def imprime():
    z = "batata"
    print(z)
imprime()
print(z)
```

```
batata
10
```

Variáveis Globais

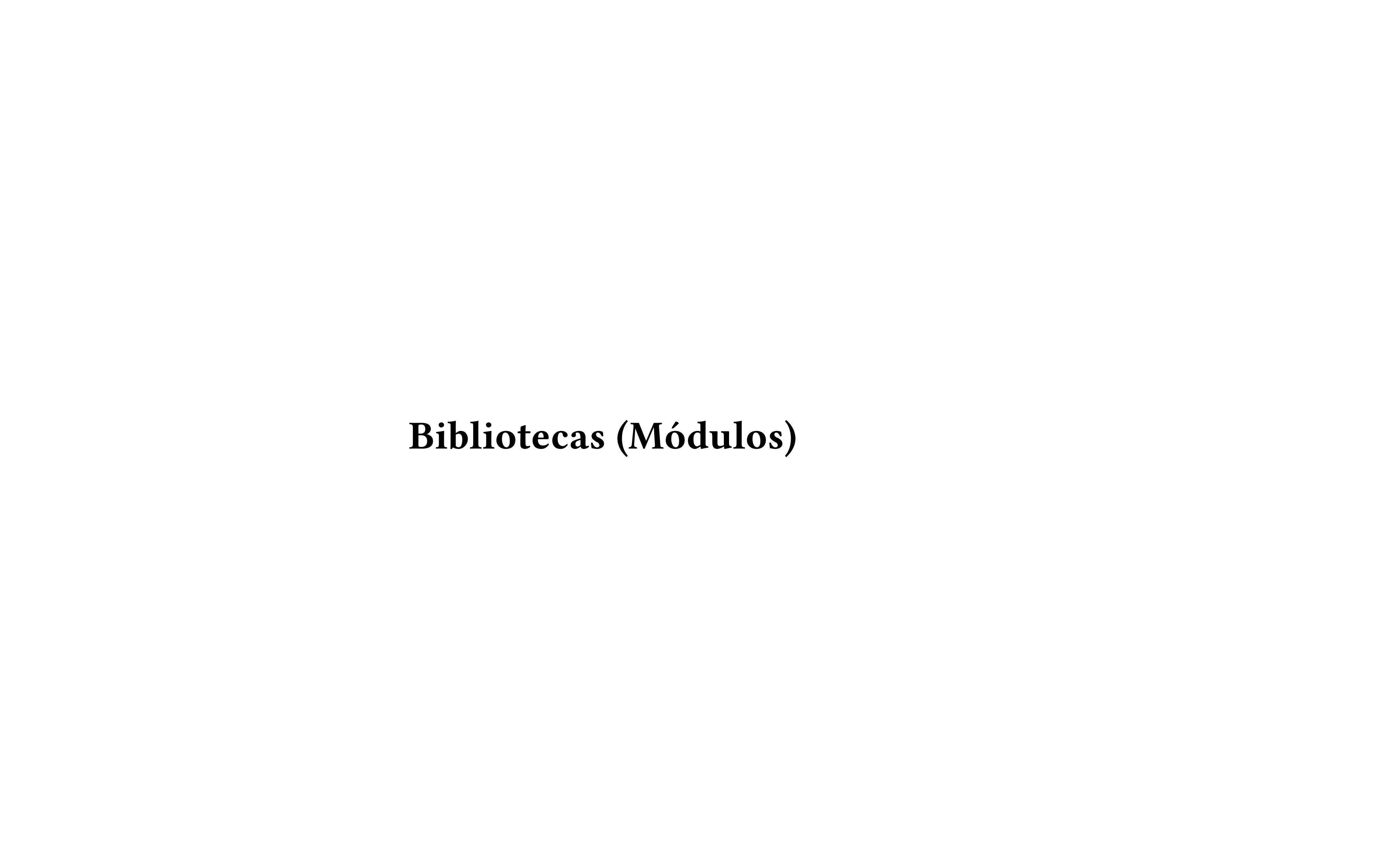
Variáveis criadas **fora** de qualquer função pertencem ao **Escopo Global** e podem ser acessadas por todo mundo.

```python
z = 10
def imprime():
    z = "batata"
    print(z)
imprime()
print(z)
```

```
batata
10
```

⚡ **Danger**

Se você tentar modificar uma variável global sem avisar o Python, ele vai criar uma **nova** variável local com o mesmo nome e você vai ficar maluco caçando o bug.


Bibliotecas (Módulos)

Não reinvente a roda

Bibliotecas são pacotes de código já escritos, otimizados e organizados para nós usarmos.

Não reinvente a roda

Bibliotecas são pacotes de código já escritos, otimizados e organizados para nós usarmos.

O comando para usar uma biblioteca é o `import`.

Não reinvente a roda

Bibliotecas são pacotes de código já escritos, otimizados e organizados para nós usarmos.

O comando para usar uma biblioteca é o `import`.

Importando toda a biblioteca:

```
1 import math
2 print(math.sin(2.3))
```

Não reinvente a roda

Bibliotecas são pacotes de código já escritos, otimizados e organizados para nós usarmos.

O comando para usar uma biblioteca é o `import`.

Importando toda a biblioteca:

```python
import math
print(math.sin(2.3))
```

Importando com apelido:

```python
import math as m
print(m.pi)
```

Não reinvente a roda

Bibliotecas são pacotes de código já escritos, otimizados e organizados para nós usarmos.

O comando para usar uma biblioteca é o `import`.

Importando toda a biblioteca:

```
1 import math
2 print(math.sin(2.3))
```

Importando apenas uma função/constante:

```
1 from math import sin, pi
2 print(sin(pi))
```

Importando com apelido:

```
1 import math as m
2 print(m.pi)
```

Não reinvente a roda

Bibliotecas são pacotes de código já escritos, otimizados e organizados para nós usarmos.

O comando para usar uma biblioteca é o `import`.

Importando toda a biblioteca:

```python
import math
print(math.sin(2.3))
```

Importando com apelido:

```python
import math as m
print(m.pi)
```

Importando apenas uma função/constante:

```python
from math import sin, pi
print(sin(pi))
```

Tip

`from math import *` importa tudo sem precisar do prefixo `math.`, mas evite isso! Polui o escopo e pode sobrescrever nomes importantes.

Revisando

O que aprendemos?

- Funções recebem entradas, executam instruções e podem devolver um valor com `return`
- Definimos funções com `def` e chamamos essas funções passando argumentos
- Funções ajudam a evitar repetição, organizar o código e dividir problemas grandes em partes menores
- Se uma função não usa `return`, o Python devolve `None`

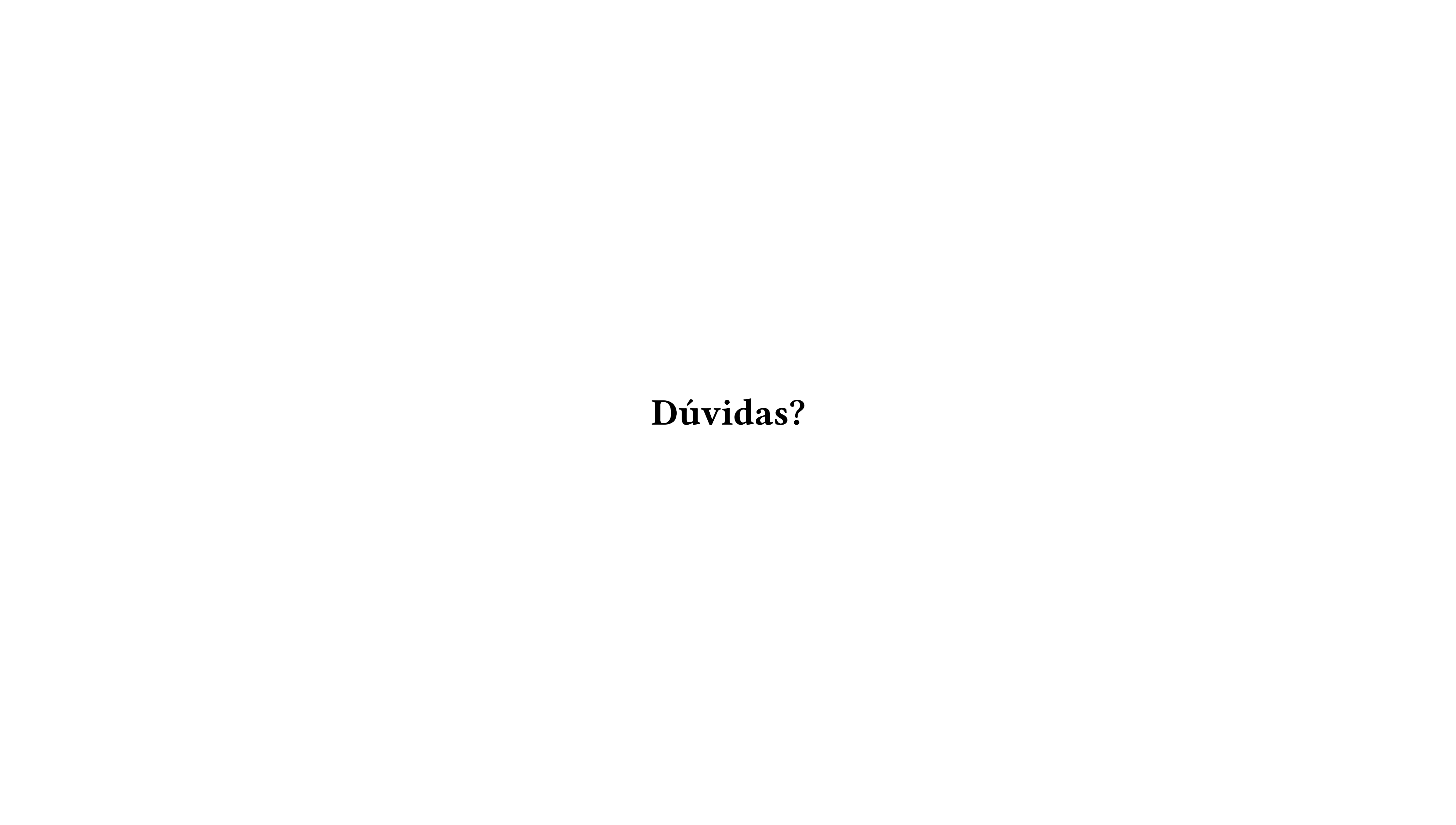
```python
def nome_funcao(arg1: int, arg2: float) -> list[str]:
    # intruções da função
    ...
    return algo
```

O que aprendemos?

- Funções recebem entradas, executam instruções e podem devolver um valor com `return`
- Definimos funções com `def` e chamamos essas funções passando argumentos
- Funções ajudam a evitar repetição, organizar o código e dividir problemas grandes em partes menores
- Se uma função não usa `return`, o Python devolve `None`

```
def nome_funcao(arg1: int, arg2: float) -> list[str]:
    # intruções da função
    ...
    return algo
```

- Podemos usar **type hints** para indicar os tipos esperados dos parâmetros e do retorno
- Docstrings servem para documentar o que a função faz, quais parâmetros recebe e o que devolve
- Variáveis locais existem só dentro da função; variáveis globais vivem fora dela
- Bibliotecas reúnem funções prontas que podemos reutilizar com `import`
- Podemos importar um módulo inteiro, ou importar nomes específicos com `from ... import ...`
- Em geral, preferimos código modular, legível, documentado e reutilizável


Dúvidas?