Notas de aula de programação em Python

Encontro 8 - Funções parte 2

Prof. Louis Augusto
louis.augusto@ifsc.edu.br

Instituto Federal de Santa Catarina
Campus São José

Índice

Sumário

Definindo uma função

Uma função representa um bloco de código que quer-se guardar para reuso.
Reveja o código abaixo (do encontro 1):

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

Este código permite repetir quantas vezes quiser, em qualquer momento, esta operação. Observe que o retorno é feito com três variáveis soltas.

Pode-se receber a função também com 3 variáeis soltas, por exemplo:

```
r1,r2,delta = eq2grau(2,-5,7)
print(r1,r2,delta)
```

Ou encapsular o retorno em uma tupla (vetor imutável):

```
saida = eq2grau(2,-5,7)
print(saida[0], saida[1], saida[2])
```

Definindo uma função

Uma função representa um bloco de código que quer-se guardar para reuso. Reveja o código abaixo (do encontro 1):

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

Este código permite repetir quantas vezes quiser, em qualquer momento, esta operação. Observe que o retorno é feito com três variáveis soltas.

Pode-se receber a função também com 3 variáeis soltas, por exemplo:

```
r1,r2,delta = eq2grau(2,-5,7)
print(r1,r2,delta)
```

Ou encapsular o retorno em uma tupla (vetor imutável):

```
saida = eq2grau(2,-5,7)
print(saida[0], saida[1], saida[2])
```

Definindo uma função

Uma função representa um bloco de código que quer-se guardar para reuso. Reveja o código abaixo (do encontro 1):

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

Este código permite repetir quantas vezes quiser, em qualquer momento, esta operação. Observe que o retorno é feito com três variáveis soltas.

Pode-se receber a função também com 3 variáeis soltas, por exemplo:

```
r1,r2,delta = eq2grau(2,-5,7)
print(r1,r2,delta)
```

Ou encapsular o retorno em uma tupla (vetor imutável):

```
saida = eq2grau(2,-5,7)
print(saida[0], saida[1], saida[2])
```

Definindo uma função

Uma função representa um bloco de código que quer-se guardar para reuso. Reveja o código abaixo (do encontro 1):

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

Este código permite repetir quantas vezes quiser, em qualquer momento, esta operação. Observe que o retorno é feito com três variáveis soltas.

Pode-se receber a função também com 3 variáeis soltas, por exemplo:

```
r1,r2,delta = eq2grau(2,-5,7)
print(r1,r2,delta)
```

Ou encapsular o retorno em uma tupla (vetor imutável):

```
saida = eq2grau(2,-5,7)
print(saida[0], saida[1], saida[2])
```

Definindo uma função

Uma função representa um bloco de código que quer-se guardar para reuso. Reveja o código abaixo (do encontro 1):

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

Este código permite repetir quantas vezes quiser, em qualquer momento, esta operação. Observe que o retorno é feito com três variáveis soltas.

Pode-se receber a função também com 3 variáeis soltas, por exemplo:

```
r1,r2,delta = eq2grau(2,-5,7)
print(r1,r2,delta)
```

Ou encapsular o retorno em uma tupla (vetor imutável):

```
saida = eq2grau(2,-5,7)
print(saida[0], saida[1], saida[2])
```

Definindo uma função

Uma função representa um bloco de código que quer-se guardar para reuso. Reveja o código abaixo (do encontro 1):

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

Este código permite repetir quantas vezes quiser, em qualquer momento, esta operação. Observe que o retorno é feito com três variáveis soltas.

Pode-se receber a função também com 3 variáeis soltas, por exemplo:

```
r1,r2,delta = eq2grau(2,-5,7)
print(r1,r2,delta)
```

Ou encapsular o retorno em uma tupla (vetor imutável):

```
saida = eq2grau(2,-5,7)
print(saida[0], saida[1], saida[2])
```

Definindo uma função

Uma curiosidade é que podemos forçar o retorno de um vetor (mutável ou imutável).

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return (x1, x2, Delta) #Caso imutável, uso de ()
```

A última linha poderia ser trocada para

```
return [x1, x2, Delta] #Caso mutável, uso de []
```

Observe as possíveis saídas do código abaixo, com ( ) e [ ]:

```
resposta = eq2grau(2,5,-2444)
print(type(resposta))
print(resposta)
```

Caso imutável:

```
<class 'tuple'>
(-52.0, 47.0, 9801)
```

Caso mutável:

```
<class 'list'>
[-52.0, 47.0, 9801]
```

Definindo uma função

Uma curiosidade é que podemos forçar o retorno de um vetor (mutável ou imutável).

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return (x1, x2, Delta) #Caso imutável, uso de ()
```

A última linha poderia ser trocada para

```
return [x1, x2, Delta] #Caso mutável, uso de []
```

Observe as possíveis saídas do código abaixo, com ( ) e [ ]:

```
resposta = eq2grau(2,5,-2444)
print(type(resposta))
print(resposta)
```

Caso imutável:

```
<class 'tuple'>
(-52.0, 47.0, 9801)
```

Caso mutável:

```
<class 'list'>
[-52.0, 47.0, 9801]
```

Definindo uma função

Uma curiosidade é que podemos forçar o retorno de um vetor (mutável ou imutável).

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return (x1, x2, Delta) #Caso imutável, uso de ()
```

A última linha poderia ser trocada para

```
return [x1, x2, Delta] #Caso mutável, uso de []
```

Observe as possíveis saídas do código abaixo, com ( ) e [ ]:

```
resposta = eq2grau(2,5,-2444)
print(type(resposta))
print(resposta)
```

Caso imutável:

```
<class 'tuple'>
(-52.0, 47.0, 9801)
```

Caso mutável:

```
<class 'list'>
[-52.0, 47.0, 9801]
```

Definindo uma função

Uma curiosidade é que podemos forçar o retorno de um vetor (mutável ou imutável).

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return (x1, x2, Delta) #Caso imutável, uso de ()
```

A última linha poderia ser trocada para

```
return [x1, x2, Delta] #Caso mutável, uso de []
```

Observe as possíveis saídas do código abaixo, com ( ) e [ ]:

```
resposta = eq2grau(2,5,-2444)
print(type(resposta))
print(resposta)
```

Caso imutável:

```
<class 'tuple'>
(-52.0, 47.0, 9801)
```

Caso mutável:

```
<class 'list'>
[-52.0, 47.0, 9801]
```

Definindo uma função

Uma curiosidade é que podemos forçar o retorno de um vetor (mutável ou imutável).

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return (x1, x2, Delta) #Caso imutável, uso de ()
```

A última linha poderia ser trocada para

```
return [x1, x2, Delta] #Caso mutável, uso de []
```

Observe as possíveis saídas do código abaixo, com ( ) e [ ]:

```
resposta = eq2grau(2,5,-2444)
print(type(resposta))
print(resposta)
```

Caso imutável:

```
<class 'tuple'>
(-52.0, 47.0, 9801)
```

Caso mutável:

```
<class 'list'>
[-52.0, 47.0, 9801]
```

Sumário

Argumentos padrão e com palavras-chave

Podemos usar além dos argumentos de função até agora usados, os argumentos com palavra-chave.

Estes argumentos já trazem consigo um valor default, mas que podem ser alterados durante a execução do programa.

Vamos pensar numa função que calcule a média de quatro notas de um aluno, a função deve permitir que se coloque o nome do aluno e idade, mas de forma opcional.

```
def MediaAluno(p1, p2, p3, p4, Nome = None, Idade = None):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    if Idade!=None:
        print("A idade do aluno é", Idade)
    return media
media = MediaAluno(2,5,7,8, Nome = "Claudio Peixoto", Idade = 19)
```

Saída:

```
A média do aluno Claudio Peixoto é 5.5
A idade do aluno é 19
```

Argumentos padrão e com palavras-chave

Podemos usar além dos argumentos de função até agora usados, os argumentos com palavra-chave.

Estes argumentos já trazem consigo um valor default, mas que podem ser alterados durante a execução do programa.

Vamos pensar numa função que calcule a média de quatro notas de um aluno, a função deve permitir que se coloque o nome do aluno e idade, mas de forma opcional.

```
def MediaAluno(p1, p2, p3, p4, Nome = None, Idade = None):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    if Idade!=None:
        print("A idade do aluno é", Idade)
    return media
media = MediaAluno(2,5,7,8, Nome = "Claudio Peixoto", Idade = 19)
```

Saída:

```
A média do aluno Claudio Peixoto é 5.5
A idade do aluno é 19
```

Argumentos padrão e com palavras-chave

Podemos usar além dos argumentos de função até agora usados, os argumentos com palavra-chave.

Estes argumentos já trazem consigo um valor default, mas que podem ser alterados durante a execução do programa.

Vamos pensar numa função que calcule a média de quatro notas de um aluno, a função deve permitir que se coloque o nome do aluno e idade, mas de forma opcional.

```
def MediaAluno(p1, p2, p3, p4, Nome = None, Idade = None):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    if Idade!=None:
        print("A idade do aluno é", Idade)
    return media
media = MediaAluno(2,5,7,8, Nome = "Claudio Peixoto", Idade = 19)
```

Saída:

```
A média do aluno Claudio Peixoto é 5.5
A idade do aluno é 19
```

Argumentos padrão e com palavras-chave

Podemos usar além dos argumentos de função até agora usados, os argumentos com palavra-chave.

Estes argumentos já trazem consigo um valor default, mas que podem ser alterados durante a execução do programa.

Vamos pensar numa função que calcule a média de quatro notas de um aluno, a função deve permitir que se coloque o nome do aluno e idade, mas de forma opcional.

```
def MediaAluno(p1, p2, p3, p4, Nome = None, Idade = None):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    if Idade!=None:
        print("A idade do aluno é", Idade)
    return media
media = MediaAluno(2,5,7,8, Nome = "Claudio Peixoto", Idade = 19)
```

Saída:

```
A média do aluno Claudio Peixoto é 5.5
A idade do aluno é 19
```

Argumentos padrão e com palavras-chave

Podemos usar além dos argumentos de função até agora usados, os argumentos com palavra-chave.

Estes argumentos já trazem consigo um valor default, mas que podem ser alterados durante a execução do programa.

Vamos pensar numa função que calcule a média de quatro notas de um aluno, a função deve permitir que se coloque o nome do aluno e idade, mas de forma opcional.

```
def MediaAluno(p1, p2, p3, p4, Nome = None, Idade = None):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    if Idade!=None:
        print("A idade do aluno é", Idade)
    return media
media = MediaAluno(2,5,7,8, Nome = "Claudio Peixoto", Idade = 19)
```

Saída:

```
A média do aluno Claudio Peixoto é 5.5
A idade do aluno é 19
```

Argumentos padrão e com palavras-chave

Poderíamos ter rodado o código anterior sem nenhum argumento com palavra-chave, e como retorno teríamos somente a média encontrada.

Considere a linha após a função:

```
print("Média =", MediaAluno(2,5,7,8))
```

O que gera a saída: `Média = 5.5`

A palavra chave pode ser um valor diferente de None:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8)
```

O que gera a saída:

```
A média do aluno Caio é 5.5
```

Argumentos padrão e com palavras-chave

Poderíamos ter rodado o código anterior sem nenhum argumento com palavra-chave, e como retorno teríamos somente a média encontrada.

Considere a linha após a função:

```
print("Média =", MediaAluno(2,5,7,8))
```

O que gera a saída: `Média = 5.5`

A palavra chave pode ser um valor diferente de None:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8)
```

O que gera a saída:

```
A média do aluno Caio é 5.5
```

Argumentos padrão e com palavras-chave

Poderíamos ter rodado o código anterior sem nenhum argumento com palavra-chave, e como retorno teríamos somente a média encontrada.

Considere a linha após a função:

```
print("Média =", MediaAluno(2,5,7,8))
```

O que gera a saída: `Média = 5.5`

A palavra chave pode ser um valor diferente de None:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8)
```

O que gera a saída:

```
A média do aluno Caio é 5.5
```

Argumentos padrão e com palavras-chave

Poderíamos ter rodado o código anterior sem nenhum argumento com palavra-chave, e como retorno teríamos somente a média encontrada.

Considere a linha após a função:

```
print("Média =", MediaAluno(2,5,7,8))
```

O que gera a saída: `Média = 5.5`

A palavra chave pode ser um valor diferente de None:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8)
```

O que gera a saída:

```
A média do aluno Caio é 5.5
```

Argumentos padrão e com palavras-chave

Poderíamos ter rodado o código anterior sem nenhum argumento com palavra-chave, e como retorno teríamos somente a média encontrada.

Considere a linha após a função:

```
print("Média =", MediaAluno(2,5,7,8))
```

O que gera a saída: `Média = 5.5`

A palavra chave pode ser um valor diferente de None:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8)
```

O que gera a saída:

```
A média do aluno Caio é 5.5
```

Argumentos padrão e com palavras-chave

Argumentos com palavras-chave podem ser substituídos na chamada da função. Considere o mesmo bloco de código anterior, e a chamada a seguir:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8, Nome = "Carlos")
```

O que gera a saída:

```
A média do aluno Carlos é 5.5
```

OBS: Depois do primeiro argumento com palavra-chave ser definido no escopo da função nenhum argumento padrão pode ser definido.
Gera erro a definição:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio", p5)
```

porque `Nome` é um argumento com palavra-chave e `p5` é variável padrão, que está definida depois de um argumento com palavra-chave.

Argumentos padrão e com palavras-chave

Argumentos com palavras-chave podem ser substituídos na chamada da função. Considere o mesmo bloco de código anterior, e a chamada a seguir:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8, Nome = "Carlos")
```

O que gera a saída:

```
A média do aluno Carlos é 5.5
```

OBS: Depois do primeiro argumento com palavra-chave ser definido no escopo da função nenhum argumento padrão pode ser definido.

Gera erro a definição:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio", p5)
```

porque `Nome` é um argumento com palavra-chave e `p5` é variável padrão, que está definida depois de um argumento com palavra-chave.

Argumentos padrão e com palavras-chave

Argumentos com palavras-chave podem ser substituídos na chamada da função. Considere o mesmo bloco de código anterior, e a chamada a seguir:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio"):
    media = (p1+p2+p3+p4)/4
    if Nome!=None:
        print("A média do aluno", Nome," é ", media)
    return media
MediaAluno(2,5,7,8, Nome = "Carlos")
```

O que gera a saída:

```
A média do aluno Carlos é 5.5
```

OBS: Depois do primeiro argumento com palavra-chave ser definido no escopo da função nenhum argumento padrão pode ser definido.
Gera erro a definição:

```
def MediaAluno(p1, p2, p3, p4, Nome = "Caio", p5)
```

porque `Nome` é um argumento com palavra-chave e `p5` é variável padrão, que está definida depois de um argumento com palavra-chave.

Sumário

Empacotamento e desempacotamento de vetores

Observe os códigos abaixo:

```
lista = [1,2,3,4,5]
n1,n2, *n = lista
print(n1,n2, n)
```

que tem como saída:

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

```
lista = [1,2,3,4,5]
n1,n2, *n, n4 = lista
print(n1,n2, n, n4)
```

que tem como saída:

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

Este procedimento é chamado desempacotamento de vetores. As variáveis `n1` e `n2` foram desempacotadas da lista, enquanto `*n` recebe o restante da lista, no primeiro caso, e no segundo caso o último valor é desempacotado.

No caso a função abaixo empacota as variáveis `x1, x2, Delta` no procedimento return.

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

O retorno é uma tupla de três valores.

Empacotamento e desempacotamento de vetores

Observe os códigos abaixo:

```
lista = [1,2,3,4,5]
n1,n2, *n = lista
print(n1,n2, n)
```

que tem como saída:

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

```
lista = [1,2,3,4,5]
n1,n2, *n, n4 = lista
print(n1,n2, n, n4)
```

que tem como saída:

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

Este procedimento é chamado desempacotamento de vetores. As variáveis `n1` e `n2` foram desempacotadas da lista, enquanto `*n` recebe o restante da lista, no primeiro caso, e no segundo caso o último valor é desempacotado.

No caso a função abaixo empacota as variáveis `x1, x2, Delta` no procedimento return.

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

O retorno é uma tupla de três valores.

Empacotamento e desempacotamento de vetores

Observe os códigos abaixo:

```
lista = [1,2,3,4,5]
n1,n2, *n = lista
print(n1,n2, n)
```

que tem como saída:

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

```
lista = [1,2,3,4,5]
n1,n2, *n, n4 = lista
print(n1,n2, n, n4)
```

que tem como saída:

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

Este procedimento é chamado desempacotamento de vetores. As variáveis `n1` e `n2` foram desempacotadas da lista, enquanto `*n` recebe o restante da lista, no primeiro caso, e no segundo caso o último valor é desempacotado.

No caso a função abaixo empacota as variáveis `x1, x2, Delta` no procedimento return.

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

O retorno é uma tupla de três valores.

Empacotamento e desempacotamento de vetores

Observe os códigos abaixo:

```
lista = [1,2,3,4,5]
n1,n2, *n = lista
print(n1,n2, n)
```

que tem como saída:

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

```
lista = [1,2,3,4,5]
n1,n2, *n, n4 = lista
print(n1,n2, n, n4)
```

que tem como saída:

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

Este procedimento é chamado desempacotamento de vetores. As variáveis `n1` e `n2` foram desempacotadas da lista, enquanto `*n` recebe o restante da lista, no primeiro caso, e no segundo caso o último valor é desempacotado.

No caso a função abaixo empacota as variáveis `x1, x2, Delta` no procedimento return.

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

O retorno é uma tupla de três valores.

Empacotamento e desempacotamento de vetores

Observe os códigos abaixo:

```
lista = [1,2,3,4,5]
n1,n2, *n = lista
print(n1,n2, n)
```

que tem como saída:

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

```
lista = [1,2,3,4,5]
n1,n2, *n, n4 = lista
print(n1,n2, n, n4)
```

que tem como saída:

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

Este procedimento é chamado desempacotamento de vetores. As variáveis `n1` e `n2` foram desempacotadas da lista, enquanto `*n` recebe o restante da lista, no primeiro caso, e no segundo caso o último valor é desempacotado.

No caso a função abaixo empacota as variáveis `x1, x2, Delta` no procedimento return.

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

O retorno é uma tupla de três valores.

Empacotamento e desempacotamento de vetores

Observe os códigos abaixo:

```
lista = [1,2,3,4,5]
n1,n2, *n = lista
print(n1,n2, n)
```

que tem como saída:

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

```
lista = [1,2,3,4,5]
n1,n2, *n, n4 = lista
print(n1,n2, n, n4)
```

que tem como saída:

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

Este procedimento é chamado desempacotamento de vetores. As variáveis `n1` e `n2` foram desempacotadas da lista, enquanto `*n` recebe o restante da lista, no primeiro caso, e no segundo caso o último valor é desempacotado.

No caso a função abaixo empacota as variáveis `x1, x2, Delta` no procedimento return.

```
def eq2grau(a, b, c):
    Delta = b**2-4*a*c
    x1 = (-b-Delta**0.5)/(2*a)
    x2 = (-b+Delta**0.5)/(2*a)
    return x1, x2, Delta
```

O retorno é uma tupla de três valores.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Observe o código, extremamente simples, abaixo:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(*args)    #Imprime os valores desempacotados.
explo_funcao(1,2,3,4,5)
```

Esta função:

- empacota os valores numa tupla;
- imprime a tupla;
- desempacota a tupla;
- imprime os valores desempacotados.

Observe a saída no terminal:

```
(1,2,3,4,5)
1,2,3,4,5
```

A última linha equivale a fazer:

```
print(1,2,3,4,5)
```

O mesmo acontece no fragmento de código abaixo:

```
lista = [1,2,3,4,5]
print(lista)
print(*lista)
```

que tem como saída:

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

Temos no primeiro caso a lista e no segundo os valores desempacotados da lista.

Empacotamento e desempacotamento de vetores

Voltando à função do slide anterior, podemos fazer impressões de valores individuais que foram empacotados.

Ao executar:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
explo_funcao(1,2,3,4,5)
```

obtemos a saída:

```
(1, 2, 3, 4, 5)
1
5
```

Como `args` dentro da função é uma tupla, os procedimentos que podem ser utilizados em vetores continuam válidos, como `len(args)`.
Lembrando que os dados numa tupla são imutáveis, mas podem ser convertidos em lista, fazendo:

```
args = list(args)
```

Agora os dados passam a ser mutáveis.

Empacotamento e desempacotamento de vetores

Voltando à função do slide anterior, podemos fazer impressões de valores individuais que foram empacotados.

Ao executar:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
explo_funcao(1,2,3,4,5)
```

obtemos a saída:

```
(1, 2, 3, 4, 5)
1
5
```

Como `args` dentro da função é uma tupla, os procedimentos que podem ser utilizados em vetores continuam válidos, como `len(args)`.
Lembrando que os dados numa tupla são imutáveis, mas podem ser convertidos em lista, fazendo:

```
args = list(args)
```

Agora os dados passam a ser mutáveis.

Empacotamento e desempacotamento de vetores

Voltando à função do slide anterior, podemos fazer impressões de valores individuais que foram empacotados.

Ao executar:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
explo_funcao(1,2,3,4,5)
```

obtemos a saída:

```
(1, 2, 3, 4, 5)
1
5
```

Como `args` dentro da função é uma tupla, os procedimentos que podem ser utilizados em vetores continuam válidos, como `len(args)`.

Lembrando que os dados numa tupla são imutáveis, mas podem ser convertidos em lista, fazendo:

```
args = list(args)
```

Agora os dados passam a ser mutáveis.

Empacotamento e desempacotamento de vetores

Voltando à função do slide anterior, podemos fazer impressões de valores individuais que foram empacotados.

Ao executar:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
explo_funcao(1,2,3,4,5)
```

obtemos a saída:

```
(1, 2, 3, 4, 5)
1
5
```

Como `args` dentro da função é uma tupla, os procedimentos que podem ser utilizados em vetores continuam válidos, como `len(args)`.
Lembrando que os dados numa tupla são imutáveis, mas podem ser convertidos em lista, fazendo:

```
args = list(args)
```

Agora os dados passam a ser mutáveis.

Empacotamento e desempacotamento de vetores

Voltando à função do slide anterior, podemos fazer impressões de valores individuais que foram empacotados.

Ao executar:

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
explo_funcao(1,2,3,4,5)
```

obtemos a saída:

```
(1, 2, 3, 4, 5)
1
5
```

Como `args` dentro da função é uma tupla, os procedimentos que podem ser utilizados em vetores continuam válidos, como `len(args)`.
Lembrando que os dados numa tupla são imutáveis, mas podem ser convertidos em lista, fazendo:

```
args = list(args)
```

Agora os dados passam a ser mutáveis.

Empacotamento e desempacotamento de vetores

Observe agora a confusão que pode ser feita chamando a função não com uma sequência de números, mas com uma lista.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
explo_funcao(lista)
```

Temos como saída:

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

Se a chamada fosse feita com:

```
explo_funcao(lista, 8)
```

Teríamos:

```
([1, 2, 3, 4, 5],8)
[1, 2, 3, 4, 5]
8
```

Acontece algo interessante nesse fragmento de código, na primeira chamada :

1. Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
2. O primeiro e o último elemento são iguais, pois a tupla é unitária.

Empacotamento e desempacotamento de vetores

Observe agora a confusão que pode ser feita chamando a função não com uma sequência de números, mas com uma lista.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
explo_funcao(lista)
```

Temos como saída:

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

Acontece algo interessante nesse fragmento de código, na primeira chamada :

1. Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
2. O primeiro e o último elemento são iguais, pois a tupla é unitária.

Se a chamada fosse feita com:

```
explo_funcao(lista, 8)
```

Teríamos:

```
([1, 2, 3, 4, 5],8)
[1, 2, 3, 4, 5]
8
```

Empacotamento e desempacotamento de vetores

Observe agora a confusão que pode ser feita chamando a função não com uma sequência de números, mas com uma lista.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
explo_funcao(lista)
```

Temos como saída:

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

Se a chamada fosse feita com:

```
explo_funcao(lista, 8)
```

Teríamos:

```
([1, 2, 3, 4, 5],8)
[1, 2, 3, 4, 5]
8
```

Acontece algo interessante nesse fragmento de código, na primeira chamada :

1. Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
2. O primeiro e o último elemento são iguais, pois a tupla é unitária.

Empacotamento e desempacotamento de vetores

Observe agora a confusão que pode ser feita chamando a função não com uma sequência de números, mas com uma lista.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
explo_funcao(lista)
```

Temos como saída:

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

Se a chamada fosse feita com:

```
explo_funcao(lista, 8)
```

Teríamos:

```
([1, 2, 3, 4, 5],8)
[1, 2, 3, 4, 5]
8
```

Acontece algo interessante nesse fragmento de código, na primeira chamada :

1. Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
2. O primeiro e o último elemento são iguais, pois a tupla é unitária.

Empacotamento e desempacotamento de vetores

Observe agora a confusão que pode ser feita chamando a função não com uma sequência de números, mas com uma lista.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
explo_funcao(lista)
```

Temos como saída:

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

Se a chamada fosse feita com:

```
explo_funcao(lista, 8)
```

Teríamos:

```
([1, 2, 3, 4, 5],8)
[1, 2, 3, 4, 5]
8
```

Acontece algo interessante nesse fragmento de código, na primeira chamada :

1. Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
2. O primeiro e o último elemento são iguais, pois a tupla é unitária.

Empacotamento e desempacotamento de vetores

Observe agora a confusão que pode ser feita chamando a função não com uma sequência de números, mas com uma lista.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
explo_funcao(lista)
```

Temos como saída:

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

Se a chamada fosse feita com:

```
explo_funcao(lista, 8)
```

Teríamos:

```
([1, 2, 3, 4, 5],8)
[1, 2, 3, 4, 5]
8
```

Acontece algo interessante nesse fragmento de código, na primeira chamada :

1. Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
2. O primeiro e o último elemento são iguais, pois a tupla é unitária.

Empacotamento e desempacotamento de vetores

Observe agora a confusão que pode ser feita chamando a função não com uma sequência de números, mas com uma lista.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)     #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
explo_funcao(lista)
```

Temos como saída:

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

Se a chamada fosse feita com:

```
explo_funcao(lista, 8)
```

Teríamos:

```
([1, 2, 3, 4, 5],8)
[1, 2, 3, 4, 5]
8
```

Acontece algo interessante nesse fragmento de código, na primeira chamada :

1. Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
2. O primeiro e o último elemento são iguais, pois a tupla é unitária.

Empacotamento e desempacotamento de vetores

Vamos elucidar mais um pouco o que ocorre quando executamos a função `explo_funcao(*args)`.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)    #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
print("Primeira Chamada")
explo_funcao(*lista)
print("Segunda Chamada")
explo_funcao(*lista,10,20,30,40,50)
print("Terceira Chamada")
explo_funcao(lista,10,20,30,40,50)
```

E temos as saídas:

```
Primeira Chamada
(1, 2, 3, 4, 5)
1
5
Segunda Chamada
(1, 2, 3, 4, 5, 10, 20, 30, 40, 50)
1
50
Terceira Chamada
([1, 2, 3, 4, 5], 10, 20, 30, 40, 50)
[1, 2, 3, 4, 5]
50
```

Pense como seria a saída da chamada:

```
lista = [1,2,3,4,5]
lista2 = [10,20,30,4,50]
explo_funcao(*lista,*lista2)
```

Empacotamento e desempacotamento de vetores

Vamos elucidar mais um pouco o que ocorre quando executamos a função `explo_funcao(*args)`.

```
def explo_funcao(*args): #Empacota os valores.
    print(args)    #Imprime os valores empacotados.
    print(args[0]) #Imprime o primeiro valor da tupla.
    print(args[-1])   #Imprime o último valor da tupla.
lista = [1,2,3,4,5]
print("Primeira Chamada")
explo_funcao(*lista)
print("Segunda Chamada")
explo_funcao(*lista,10,20,30,40,50)
print("Terceira Chamada")
explo_funcao(lista,10,20,30,40,50)
```

E temos as saídas:

```
Primeira Chamada
(1, 2, 3, 4, 5)
1
5
Segunda Chamada
(1, 2, 3, 4, 5, 10, 20, 30, 40, 50)
1
50
```

```
Terceira Chamada
([1, 2, 3, 4, 5], 10, 20, 30, 40, 50)
[1, 2, 3, 4, 5]
50
```

Pense como seria a saída da chamada:

```
lista = [1,2,3,4,5]
lista2 = [10,20,30,4,50]
explo_funcao(*lista,*lista2)
```

Sumário

Funções recursivas

Funções recursivas são as funções que chamam a si mesmas.

Talvez a função recursiva mais simples de se entender é a que gera o fatorial de um número inteiro positivo.

Definimos $n! = n.(n-1).(n-2).....3.2.1$, para $n > 1$.[1]

Como exemplo, $5! = 5.4.3.2.1 = 120$,

Uma função recursiva deve retornar se uma dada condição ocorrer, chamado caso base, caso contrário ficará infinitamente se chamando até quebrar a memória da máquina.
Vamos ao código:

```
def fat(n):
    if n==1: #Caso base
        return 1
    else: #Caso recursivo
     return n*fat(n-1) #Multiplica o número pelo seu antecessor.
print(fat(5))
```

[1] $n!$ lê-se n fatorial

Funções recursivas

Funções recursivas são as funções que chamam a si mesmas.

Talvez a função recursiva mais simples de se entender é a que gera o fatorial de um número inteiro positivo.

Definimos $n! = n.(n-1).(n-2).....3.2.1$, para $n > 1$.[1]

Como exemplo, $5! = 5.4.3.2.1 = 120$,

Uma função recursiva deve retornar se uma dada condição ocorrer, chamado caso base, caso contrário ficará infinitamente se chamando até quebrar a memória da máquina.
Vamos ao código:

```
def fat(n):
    if n==1: #Caso base
        return 1
    else: #Caso recursivo
     return n*fat(n-1) #Multiplica o número pelo seu antecessor.
print(fat(5))
```

[1] $n!$ lê-se n fatorial

Funções recursivas

Funções recursivas são as funções que chamam a si mesmas.

Talvez a função recursiva mais simples de se entender é a que gera o fatorial de um número inteiro positivo.

Definimos $n! = n.(n-1).(n-2).....3.2.1$, para $n > 1$.[1]

Como exemplo, $5! = 5.4.3.2.1 = 120$,

Uma função recursiva deve retornar se uma dada condição ocorrer, chamado caso base, caso contrário ficará infinitamente se chamando até quebrar a memória da máquina.
Vamos ao código:

```
def fat(n):
    if n==1: #Caso base
        return 1
    else: #Caso recursivo
     return n*fat(n-1) #Multiplica o número pelo seu antecessor.
print(fat(5))
```

[1] $n!$ lê-se n fatorial

Funções recursivas

Funções recursivas são as funções que chamam a si mesmas.

Talvez a função recursiva mais simples de se entender é a que gera o fatorial de um número inteiro positivo.

Definimos $n! = n.(n-1).(n-2).....3.2.1$, para $n > 1$.[1]

Como exemplo, $5! = 5.4.3.2.1 = 120$,

Uma função recursiva deve retornar se uma dada condição ocorrer, chamado caso base, caso contrário ficará infinitamente se chamando até quebrar a memória da máquina.
Vamos ao código:

```
def fat(n):
    if n==1: #Caso base
        return 1
    else: #Caso recursivo
     return n*fat(n-1) #Multiplica o número pelo seu antecessor.
print(fat(5))
```

[1] $n!$ lê-se n fatorial

Funções recursivas

Funções recursivas são as funções que chamam a si mesmas.

Talvez a função recursiva mais simples de se entender é a que gera o fatorial de um número inteiro positivo.

Definimos $n! = n.(n-1).(n-2).....3.2.1$, para $n > 1$.[1]

Como exemplo, $5! = 5.4.3.2.1 = 120$,

Uma função recursiva deve retornar se uma dada condição ocorrer, chamado caso base, caso contrário ficará infinitamente se chamando até quebrar a memória da máquina.

Vamos ao código:

```
def fat(n):
    if n==1: #Caso base
        return 1
    else: #Caso recursivo
     return n*fat(n-1) #Multiplica o número pelo seu antecessor.
print(fat(5))
```

[1] $n!$ lê-se n fatorial

Funções recursivas

Funções recursivas são as funções que chamam a si mesmas.

Talvez a função recursiva mais simples de se entender é a que gera o fatorial de um número inteiro positivo.

Definimos $n! = n.(n-1).(n-2).....3.2.1$, para $n > 1$.[1]

Como exemplo, $5! = 5.4.3.2.1 = 120$,

Uma função recursiva deve retornar se uma dada condição ocorrer, chamado caso base, caso contrário ficará infinitamente se chamando até quebrar a memória da máquina.
Vamos ao código:

```
def fat(n):
    if n==1: #Caso base
        return 1
    else: #Caso recursivo
     return n*fat(n-1) #Multiplica o número pelo seu antecessor.
print(fat(5))
```

[1] $n!$ lê-se n fatorial

Funções recursivas

Funções recursivas são as funções que chamam a si mesmas.

Talvez a função recursiva mais simples de se entender é a que gera o fatorial de um número inteiro positivo.

Definimos $n! = n.(n-1).(n-2).....3.2.1$, para $n > 1$.[1]

Como exemplo, $5! = 5.4.3.2.1 = 120$,

Uma função recursiva deve retornar se uma dada condição ocorrer, chamado caso base, caso contrário ficará infinitamente se chamando até quebrar a memória da máquina.
Vamos ao código:

```
def fat(n):
    if n==1: #Caso base
        return 1
    else: #Caso recursivo
     return n*fat(n-1) #Multiplica o número pelo seu antecessor.
print(fat(5))
```

[1] $n!$ lê-se n fatorial

Funções recursivas

Deve-se entender uma chamada recursiva como se fosse uma chamada de uma nova função (ela ser a mesma é só um detalhe).

Quando a execução do código está no bloco de uma função e neste bloco há a chamada de outra função, a função interna deve ser executada até seu retorno, e então continua a execução do bloco da função externa.
Observe:

```
def Delta(a,b,c):
    return b**2-4*a*c
def eq2grau(a,b,c):
    discriminante = Delta(a,b,c)
    x1 = (-b+discriminante**0.5)/(2*a)
    x2 = (-b-discriminante**0.5)/(2*a)
    return x1,x2
print(eq2grau(1,-4,3))
```

Para executar o código, a função eq2grau() chama a função Delta(), e a função eq2grau() só continua a execução depois que Delta() retorna.
O mesmo ocorre com a função `fat(n)`, a função que chamou somente prossegue depois que a função chamada retorna

Funções recursivas

Deve-se entender uma chamada recursiva como se fosse uma chamada de uma nova função (ela ser a mesma é só um detalhe).

Quando a execução do código está no bloco de uma função e neste bloco há a chamada de outra função, a função interna deve ser executada até seu retorno, e então continua a execução do bloco da função externa.
Observe:

```
def Delta(a,b,c):
    return b**2-4*a*c
def eq2grau(a,b,c):
    discriminante = Delta(a,b,c)
    x1 = (-b+discriminante**0.5)/(2*a)
    x2 = (-b-discriminante**0.5)/(2*a)
    return x1,x2
print(eq2grau(1,-4,3))
```

Para executar o código, a função eq2grau() chama a função Delta(), e a função eq2grau() só continua a execução depois que Delta() retorna.
O mesmo ocorre com a função `fat(n)`, a função que chamou somente prossegue depois que a função chamada retorna

Funções recursivas

Deve-se entender uma chamada recursiva como se fosse uma chamada de uma nova função (ela ser a mesma é só um detalhe).

Quando a execução do código está no bloco de uma função e neste bloco há a chamada de outra função, a função interna deve ser executada até seu retorno, e então continua a execução do bloco da função externa.
Observe:

```
def Delta(a,b,c):
    return b**2-4*a*c
def eq2grau(a,b,c):
    discriminante = Delta(a,b,c)
    x1 = (-b+discriminante**0.5)/(2*a)
    x2 = (-b-discriminante**0.5)/(2*a)
    return x1,x2
print(eq2grau(1,-4,3))
```

Para executar o código, a função eq2grau() chama a função Delta(), e a função eq2grau() só continua a execução depois que Delta() retorna.
O mesmo ocorre com a função `fat(n)`, a função que chamou somente prossegue depois que a função chamada retorna

Funções recursivas

Deve-se entender uma chamada recursiva como se fosse uma chamada de uma nova função (ela ser a mesma é só um detalhe).

Quando a execução do código está no bloco de uma função e neste bloco há a chamada de outra função, a função interna deve ser executada até seu retorno, e então continua a execução do bloco da função externa.
Observe:

```
def Delta(a,b,c):
    return b**2-4*a*c
def eq2grau(a,b,c):
    discriminante = Delta(a,b,c)
    x1 = (-b+discriminante**0.5)/(2*a)
    x2 = (-b-discriminante**0.5)/(2*a)
    return x1,x2
print(eq2grau(1,-4,3))
```

Para executar o código, a função eq2grau() chama a função Delta(), e a função eq2grau() só continua a execução depois que Delta() retorna.
O mesmo ocorre com a função `fat(n)`, a função que chamou somente prossegue depois que a função chamada retorna

Funções recursivas

Deve-se entender uma chamada recursiva como se fosse uma chamada de uma nova função (ela ser a mesma é só um detalhe).

Quando a execução do código está no bloco de uma função e neste bloco há a chamada de outra função, a função interna deve ser executada até seu retorno, e então continua a execução do bloco da função externa.
Observe:

```
def Delta(a,b,c):
    return b**2-4*a*c
def eq2grau(a,b,c):
    discriminante = Delta(a,b,c)
    x1 = (-b+discriminante**0.5)/(2*a)
    x2 = (-b-discriminante**0.5)/(2*a)
    return x1,x2
print(eq2grau(1,-4,3))
```

Para executar o código, a função eq2grau() chama a função Delta(), e a função eq2grau() só continua a execução depois que Delta() retorna.
O mesmo ocorre com a função `fat(n)`, a função que chamou somente prossegue depois que a função chamada retorna

Funções recursivas

A função recursiva trabalha da mesma forma, a função que chama fica esperando a função chamada retornar, ou seja, as funções ficam se empilhando até que haja um retorno, aí todas retornam em cascata.

Python possui um limite de chamadas recursivas, que é 999. Observe que o script funciona até `fat(998)`, mas não `fat(999)`.

Um outro exemplo interessante é a sequência de Fibonacci:

A sequência de Fibonacci

A sequência de Fibonacci é a sequência tal que qualquer número, exceto o primeiro (zero) e o segundo (um) é a soma dos dois números anteriores.

A sequência é dada por: 0,1,1,2,3,5,8,13,21, . . .

O código recursivo é extremamente simples.

Funções recursivas

A função recursiva trabalha da mesma forma, a função que chama fica esperando a função chamada retornar, ou seja, as funções ficam se empilhando até que haja um retorno, aí todas retornam em cascata.

Python possui um limite de chamadas recursivas, que é 999. Observe que o script funciona até `fat(998)`, mas não `fat(999)`.

Um outro exemplo interessante é a sequência de Fibonacci:

A sequência de Fibonacci

A sequência de Fibonacci é a sequência tal que qualquer número, exceto o primeiro (zero) e o segundo (um) é a soma dos dois números anteriores.

A sequência é dada por: 0,1,1,2,3,5,8,13,21, . . .

O código recursivo é extremamente simples.

Funções recursivas

A função recursiva trabalha da mesma forma, a função que chama fica esperando a função chamada retornar, ou seja, as funções ficam se empilhando até que haja um retorno, aí todas retornam em cascata.

Python possui um limite de chamadas recursivas, que é 999. Observe que o script funciona até `fat(998)`, mas não `fat(999)`.

Um outro exemplo interessante é a sequência de Fibonacci:

A sequência de Fibonacci

A sequência de Fibonacci é a sequência tal que qualquer número, exceto o primeiro (zero) e o segundo (um) é a soma dos dois números anteriores.

A sequência é dada por: 0,1,1,2,3,5,8,13,21, ...

O código recursivo é extremamente simples.

Funções recursivas

A função recursiva trabalha da mesma forma, a função que chama fica esperando a função chamada retornar, ou seja, as funções ficam se empilhando até que haja um retorno, aí todas retornam em cascata.

Python possui um limite de chamadas recursivas, que é 999. Observe que o script funciona até `fat(998)`, mas não `fat(999)`.

Um outro exemplo interessante é a sequência de Fibonacci:

A sequência de Fibonacci

A sequência de Fibonacci é a sequência tal que qualquer número, exceto o primeiro (zero) e o segundo (um) é a soma dos dois números anteriores.

A sequência é dada por: 0,1,1,2,3,5,8,13,21, ...

O código recursivo é extremamente simples.

Funções recursivas

A função recursiva trabalha da mesma forma, a função que chama fica esperando a função chamada retornar, ou seja, as funções ficam se empilhando até que haja um retorno, aí todas retornam em cascata.

Python possui um limite de chamadas recursivas, que é 999. Observe que o script funciona até `fat(998)`, mas não `fat(999)`.

Um outro exemplo interessante é a sequência de Fibonacci:

A sequência de Fibonacci

A sequência de Fibonacci é a sequência tal que qualquer número, exceto o primeiro (zero) e o segundo (um) é a soma dos dois números anteriores.

A sequência é dada por: 0,1,1,2,3,5,8,13,21, ...

O código recursivo é extremamente simples.

Funções recursivas

A função recursiva trabalha da mesma forma, a função que chama fica esperando a função chamada retornar, ou seja, as funções ficam se empilhando até que haja um retorno, aí todas retornam em cascata.

Python possui um limite de chamadas recursivas, que é 999. Observe que o script funciona até `fat(998)`, mas não `fat(999)`.

Um outro exemplo interessante é a sequência de Fibonacci:

A sequência de Fibonacci

A sequência de Fibonacci é a sequência tal que qualquer número, exceto o primeiro (zero) e o segundo (um) é a soma dos dois números anteriores.

A sequência é dada por: 0,1,1,2,3,5,8,13,21, ...

O código recursivo é extremamente simples.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

Que gera o oitavo item da sequência de Fibonacci.

O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.
termo 5: 15 chamadas.
⋮
termo 14: 1219 chamadas.
⋮
termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.
O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

Que gera o oitavo item da sequência de Fibonacci.

O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.

termo 5: 15 chamadas.

⋮

termo 14: 1219 chamadas.

⋮

termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.
O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

Que gera o oitavo item da sequência de Fibonacci.

O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.

termo 5: 15 chamadas.

⋮

termo 14: 1219 chamadas.

⋮

termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.
O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

Que gera o oitavo item da sequência de Fibonacci.

O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.

termo 5: 15 chamadas.

⋮

termo 14: 1219 chamadas.

⋮

termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.
O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

Que gera o oitavo item da sequência de Fibonacci.

O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.

termo 5: 15 chamadas.

⋮

termo 14: 1219 chamadas.

⋮

termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.
O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

Que gera o oitavo item da sequência de Fibonacci.

O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.

termo 5: 15 chamadas.

⋮

termo 14: 1219 chamadas.

⋮

termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.
O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

Que gera o oitavo item da sequência de Fibonacci.

O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.

termo 5: 15 chamadas.

⋮

termo 14: 1219 chamadas.

⋮

termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.

O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas

```
def fibo(n):
    if n<2:
     return n
    return fibo(n-1)+fibo(n-2)
print(fibo(7))
```

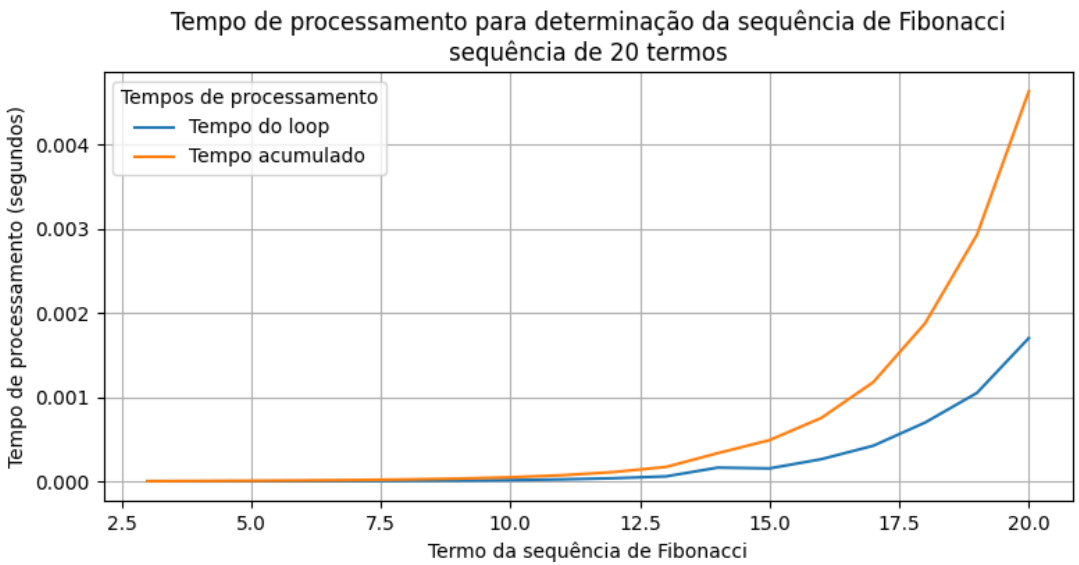
Que gera o oitavo item da sequência de Fibonacci.

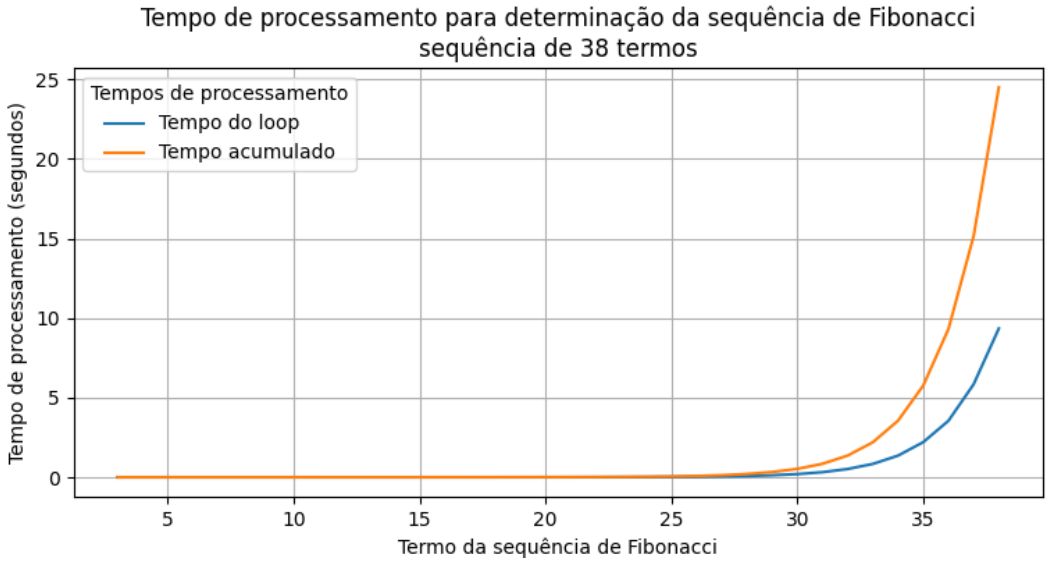
O problema que emerge é a quantidade de chamadas à função `fibo(n)`. Tente completar o quadro abaixo, com o termo da sequência e as quantidades de chamadas recursivas necessárias:

termo 4: 9 chamadas.
termo 5: 15 chamadas.
⋮
termo 14: 1219 chamadas.
⋮
termo 20: 21.831 chamadas.

Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.
O próximo slide mostra o tempo de processamento para calcular até o termo 20 e depois para calcular até o termo 38.

Funções recursivas





Funções recursivas

Para finalizar com as funções recursivas, vale lembrar que todo código com função recursiva pode ser resolvido sem recursividade.

Faça um código sem recursividade para resolver o problema do fatorial e da sequência de Fibonacci.

Funções recursivas

Para finalizar com as funções recursivas, vale lembrar que todo código com função recursiva pode ser resolvido sem recursividade.

Faça um código sem recursividade para resolver o problema do fatorial e da sequência de Fibonacci.

Funções recursivas

Para finalizar com as funções recursivas, vale lembrar que todo código com função recursiva pode ser resolvido sem recursividade.

Faça um código sem recursividade para resolver o problema do fatorial e da sequência de Fibonacci.

Sumário

Argumentos padrão desconhecidos

Às vezes queremos fazer uma função com argumentos padrão desconhecidos ou até inexistentes.

Suponha que queiramos calcular a média de um aluno, mas não sabemos quantas provas houve. Queremos uma função que calcule a média, independentemente da quantidade de provas.

Temos duas possibilidades, passar um vetor para a função ou utilizar o argumento `*args`:

```
def MediaAluno(notas):
    media = 0
    n = len(notas)
    for i in range(n):
      media+=notas[i]
    return media/=n
notas = [2,5,7,8]
print(MediaAluno(notas))
```

```
def MediaAluno(*args):
    media = 0
    n = len(args)
    for i in range(n):
      media+=args[i]
    return media/=n
print(MediaAluno(2,5,7,8))
```

Observe a diferença de como se chamar a função MediaAluno() em cada caso.

Argumentos padrão desconhecidos

Às vezes queremos fazer uma função com argumentos padrão desconhecidos ou até inexistentes.

Suponha que queiramos calcular a média de um aluno, mas não sabemos quantas provas houve. Queremos uma função que calcule a média, independentemente da quantidade de provas.

Temos duas possibilidades, passar um vetor para a função ou utilizar o argumento `*args`:

```
def MediaAluno(notas):
    media = 0
    n = len(notas)
    for i in range(n):
      media+=notas[i]
    return media/=n
notas = [2,5,7,8]
print(MediaAluno(notas))
```

```
def MediaAluno(*args):
    media = 0
    n = len(args)
    for i in range(n):
      media+=args[i]
    return media/=n
print(MediaAluno(2,5,7,8))
```

Observe a diferença de como se chamar a função MediaAluno() em cada caso.

Argumentos padrão desconhecidos

Às vezes queremos fazer uma função com argumentos padrão desconhecidos ou até inexistentes.

Suponha que queiramos calcular a média de um aluno, mas não sabemos quantas provas houve. Queremos uma função que calcule a média, independentemente da quantidade de provas.

Temos duas possibilidades, passar um vetor para a função ou utilizar o argumento `*args`:

```
def MediaAluno(notas):
    media = 0
    n = len(notas)
    for i in range(n):
      media+=notas[i]
    return media/=n
notas = [2,5,7,8]
print(MediaAluno(notas))
```

```
def MediaAluno(*args):
    media = 0
    n = len(args)
    for i in range(n):
      media+=args[i]
    return media/=n
print(MediaAluno(2,5,7,8))
```

Observe a diferença de como se chamar a função MediaAluno() em cada caso.

Argumentos padrão desconhecidos

Às vezes queremos fazer uma função com argumentos padrão desconhecidos ou até inexistentes.

Suponha que queiramos calcular a média de um aluno, mas não sabemos quantas provas houve. Queremos uma função que calcule a média, independentemente da quantidade de provas.

Temos duas possibilidades, passar um vetor para a função ou utilizar o argumento `*args`:

```
def MediaAluno(notas):
    media = 0
    n = len(notas)
    for i in range(n):
      media+=notas[i]
    return media/=n
notas = [2,5,7,8]
print(MediaAluno(notas))
```

```
def MediaAluno(*args):
    media = 0
    n = len(args)
    for i in range(n):
      media+=args[i]
    return media/=n
print(MediaAluno(2,5,7,8))
```

Observe a diferença de como se chamar a função MediaAluno() em cada caso.

Argumentos padrão desconhecidos

Às vezes queremos fazer uma função com argumentos padrão desconhecidos ou até inexistentes.

Suponha que queiramos calcular a média de um aluno, mas não sabemos quantas provas houve. Queremos uma função que calcule a média, independentemente da quantidade de provas.

Temos duas possibilidades, passar um vetor para a função ou utilizar o argumento `*args`:

```
def MediaAluno(notas):
    media = 0
    n = len(notas)
    for i in range(n):
      media+=notas[i]
    return media/=n
notas = [2,5,7,8]
print(MediaAluno(notas))
```

```
def MediaAluno(*args):
    media = 0
    n = len(args)
    for i in range(n):
      media+=args[i]
    return media/=n
print(MediaAluno(2,5,7,8))
```

Observe a diferença de como se chamar a função MediaAluno() em cada caso.

Sumário

Argumentos com palavra chave desconhecidos

Para início de conversa, `kwargs` significa `keyword arguments`, isto é, argumentos com palavras-chave.

Vamos alterar a função `explo_funcao(*args)` para incluir argumentos com palavras-chave.

```
def explo_funcao(*args, **kwargs):
    print(args)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
```

Esta função recebe argumentos com palavras-chave, que são referenciados por `**kwargs`, mas observe que não se pediu que imprimisse estes valores dos argumentos com palavras-chave.

Argumentos com palavra chave desconhecidos

Para início de conversa, `kwargs` significa `keyword arguments`, isto é, argumentos com palavras-chave.

Vamos alterar a função `explo_funcao(*args)` para incluir argumentos com palavras-chave.

```
def explo_funcao(*args, **kwargs):
    print(args)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
```

Esta função recebe argumentos com palavras-chave, que são referenciados por `**kwargs`, mas observe que não se pediu que imprimisse estes valores dos argumentos com palavras-chave.

Argumentos com palavra chave desconhecidos

Para início de conversa, `kwargs` significa `keyword arguments`, isto é, argumentos com palavras-chave.

Vamos alterar a função `explo_funcao(*args)` para incluir argumentos com palavras-chave.

```
def explo_funcao(*args, **kwargs):
    print(args)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
```

Esta função recebe argumentos com palavras-chave, que são referenciados por `**kwargs`, mas observe que não se pediu que imprimisse estes valores dos argumentos com palavras-chave.

Argumentos com palavra chave desconhecidos

Para início de conversa, `kwargs` significa `keyword arguments`, isto é, argumentos com palavras-chave.

Vamos alterar a função `explo_funcao(*args)` para incluir argumentos com palavras-chave.

```
def explo_funcao(*args, **kwargs):
    print(args)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
```

Esta função recebe argumentos com palavras-chave, que são referenciados por `**kwargs`, mas observe que não se pediu que imprimisse estes valores dos argumentos com palavras-chave.

Argumentos com palavra chave desconhecidos

Vamos agora utilizar os argumentos com palavras-chave da função:

```
def explo_funcao(*args, **kwargs):
    print(args)
    print(kwargs)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
{'nome': 'Caio', 'idade': 19}
```

Observe que os `kwargs` é transformado num dicionário, cuja chave e valor são obtidos na lista de argumentos da função.

Talvez uma melhor forma de imprimir os argumentos com palavra-chave seria substituir a linha `print(kwargs)` por `print(kwargs['nome'], kwargs['sobrenome'])`.

Argumentos com palavra chave desconhecidos

Vamos agora utilizar os argumentos com palavras-chave da função:

```
def explo_funcao(*args, **kwargs):
    print(args)
    print(kwargs)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
{'nome': 'Caio', 'idade': 19}
```

Observe que os `kwargs` é transformado num dicionário, cuja chave e valor são obtidos na lista de argumentos da função.

Talvez uma melhor forma de imprimir os argumentos com palavra-chave seria substituir a linha `print(kwargs)` por `print(kwargs['nome'], kwargs['sobrenome'])`.

Argumentos com palavra chave desconhecidos

Vamos agora utilizar os argumentos com palavras-chave da função:

```
def explo_funcao(*args, **kwargs):
    print(args)
    print(kwargs)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
{'nome': 'Caio', 'idade': 19}
```

Observe que os `kwargs` é transformado num dicionário, cuja chave e valor são obtidos na lista de argumentos da função.

Talvez uma melhor forma de imprimir os argumentos com palavra-chave seria substituir a linha `print(kwargs)` por `print(kwargs['nome'], kwargs['sobrenome'])`.

Argumentos com palavra chave desconhecidos

Vamos agora utilizar os argumentos com palavras-chave da função:

```
def explo_funcao(*args, **kwargs):
    print(args)
    print(kwargs)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
{'nome': 'Caio', 'idade': 19}
```

Observe que os `kwargs` é transformado num dicionário, cuja chave e valor são obtidos na lista de argumentos da função.

Talvez uma melhor forma de imprimir os argumentos com palavra-chave seria substituir a linha `print(kwargs)` por `print(kwargs['nome'], kwargs['sobrenome'])`.

Argumentos com palavra chave desconhecidos

Vamos agora utilizar os argumentos com palavras-chave da função:

```
def explo_funcao(*args, **kwargs):
    print(args)
    print(kwargs)
lista = [1,2,3]
lista2 = [10,20,30]
explo_funcao(lista,*lista2, nome = "Caio", idade = 19)
```

Vamos encontrar a saída:

```
([1, 2, 3], 10, 20, 30)
{'nome': 'Caio', 'idade': 19}
```

Observe que os `kwargs` é transformado num dicionário, cuja chave e valor são obtidos na lista de argumentos da função.

Talvez uma melhor forma de imprimir os argumentos com palavra-chave seria substituir a linha `print(kwargs)` por `print(kwargs['nome'], kwargs['sobrenome'])`.

Argumentos com palavra chave desconhecidos

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como arguemento com palavra-chave, mas quero verificar se uma dada chave foi enviada.

Como exemplo, em `explo_funcao(*args, **kwargs)`, usando como driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", idade = 19)
```

e desejamos saber se foi incluído o sobrenome na chamada.
Vamos alterar a função para isto.

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
     print(sbr)
```

Se não houver sobrenome na chamada, a função imprimirá somente os args, caso contrário imprimirá também o sobrenome.

Argumentos com palavra chave desconhecidos

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como arguemento com palavra-chave, mas quero verificar se uma dada chave foi enviada.

Como exemplo, em `explo_funcao(*args, **kwargs)`, usando como driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", idade = 19)
```

e desejamos saber se foi incluído o sobrenome na chamada.
Vamos alterar a função para isto.

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
     print(sbr)
```

Se não houver sobrenome na chamada, a função imprimirá somente os args, caso contrário imprimirá também o sobrenome.

Argumentos com palavra chave desconhecidos

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como arguemento com palavra-chave, mas quero verificar se uma dada chave foi enviada.

Como exemplo, em `explo_funcao(*args, **kwargs)`, usando como driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", idade = 19)
```

e desejamos saber se foi incluído o sobrenome na chamada.
Vamos alterar a função para isto.

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
     print(sbr)
```

Se não houver sobrenome na chamada, a função imprimirá somente os args, caso contrário imprimirá também o sobrenome.

Argumentos com palavra chave desconhecidos

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como arguemento com palavra-chave, mas quero verificar se uma dada chave foi enviada.

Como exemplo, em `explo_funcao(*args, **kwargs)`, usando como driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", idade = 19)
```

e desejamos saber se foi incluído o sobrenome na chamada.
Vamos alterar a função para isto.

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
     print(sbr)
```

Se não houver sobrenome na chamada, a função imprimirá somente os args, caso contrário imprimirá também o sobrenome.

Argumentos com palavra chave desconhecidos

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como arguemento com palavra-chave, mas quero verificar se uma dada chave foi enviada.

Como exemplo, em `explo_funcao(*args, **kwargs)`, usando como driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", idade = 19)
```

e desejamos saber se foi incluído o sobrenome na chamada.
Vamos alterar a função para isto.

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
     print(sbr)
```

Se não houver sobrenome na chamada, a função imprimirá somente os args, caso contrário imprimirá também o sobrenome.

Argumentos com palavra chave desconhecidos

Com a função em mente:

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
        print(sbr)
```

Alterando o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

Temos a saída:

```
([1, 2, 3],)
Martins
```

Deve-se observar que há somente um argumento padrão e um argumento com palavra-chave neste caso.

Argumentos com palavra chave desconhecidos

Com a função em mente:

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
        print(sbr)
```

Alterando o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

Temos a saída:

```
([1, 2, 3],)
Martins
```

Deve-se observar que há somente um argumento padrão e um argumento com palavra-chave neste caso.

Argumentos com palavra chave desconhecidos

Com a função em mente:

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
        print(sbr)
```

Alterando o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

Temos a saída:

```
([1, 2, 3],)
Martins
```

Deve-se observar que há somente um argumento padrão e um argumento com palavra-chave neste caso.

Argumentos com palavra chave desconhecidos

Com a função em mente:

```
def explo_funcao(*args, **kwargs):
    print(args)
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
        print(sbr)
```

Alterando o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

Temos a saída:

```
([1, 2, 3],)
Martins
```

Deve-se observar que há somente um argumento padrão e um argumento com palavra-chave neste caso.

Argumentos com palavra-chave desconhecidos

Quando não se tem certeza se uma chave nos argumentos com palavra-chave existe ou não deve-se usar o procedimento anterior com `dic.get(k)` ou uma sequência `try-except`.

Caso se chame uma chave diretamente que não existe ocorre erro.

```
def explo_funcao(*args, **kwargs):
    print(args)
    peso = kwargs['peso']
```

gera erro com o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

porque não há a palavra-chave `peso` na chamada da função, gerando um erro `KeyError`, que pode ser sanado pela evocação de uma exceção.

```
def explo_funcao(*args, **kwargs):
    print(args)
    try:
        peso = kwargs['peso']
    except KeyError:
        pass
```

Argumentos com palavra-chave desconhecidos

Quando não se tem certeza se uma chave nos argumentos com palavra-chave existe ou não deve-se usar o procedimento anterior com `dic.get(k)` ou uma sequência `try-except`.
Caso se chame uma chave diretamente que não existe ocorre erro.

```
def explo_funcao(*args, **kwargs):
    print(args)
    peso = kwargs['peso']
```

gera erro com o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

porque não há a palavra-chave `peso` na chamada da função, gerando um erro `KeyError`, que pode ser sanado pela evocação de uma exceção.

```
def explo_funcao(*args, **kwargs):
    print(args)
    try:
        peso = kwargs['peso']
    except KeyError:
        pass
```

Argumentos com palavra-chave desconhecidos

Quando não se tem certeza se uma chave nos argumentos com palavra-chave existe ou não deve-se usar o procedimento anterior com `dic.get(k)` ou uma sequência `try-except`.
Caso se chame uma chave diretamente que não existe ocorre erro.

```
def explo_funcao(*args, **kwargs):
    print(args)
    peso = kwargs['peso']
```

gera erro com o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

porque não há a palavra-chave `peso` na chamada da função, gerando um erro `KeyError`, que pode ser sanado pela evocação de uma exceção.

```
def explo_funcao(*args, **kwargs):
    print(args)
    try:
        peso = kwargs['peso']
    except KeyError:
        pass
```

Argumentos com palavra-chave desconhecidos

Quando não se tem certeza se uma chave nos argumentos com palavra-chave existe ou não deve-se usar o procedimento anterior com `dic.get(k)` ou uma sequência `try-except`.
Caso se chame uma chave diretamente que não existe ocorre erro.

```
def explo_funcao(*args, **kwargs):
    print(args)
    peso = kwargs['peso']
```

gera erro com o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

porque não há a palavra-chave `peso` na chamada da função, gerando um erro `KeyError`, que pode ser sanado pela evocação de uma exceção.

```
def explo_funcao(*args, **kwargs):
    print(args)
    try:
        peso = kwargs['peso']
    except KeyError:
        pass
```

Argumentos com palavra-chave desconhecidos

Quando não se tem certeza se uma chave nos argumentos com palavra-chave existe ou não deve-se usar o procedimento anterior com `dic.get(k)` ou uma sequência `try-except`.
Caso se chame uma chave diretamente que não existe ocorre erro.

```
def explo_funcao(*args, **kwargs):
    print(args)
    peso = kwargs['peso']
```

gera erro com o driver:

```
lista = [1,2,3]
explo_funcao(lista, nome = "Caio", sobrenome = "Martins",
    idade = 19)
```

porque não há a palavra-chave `peso` na chamada da função, gerando um erro `KeyError`, que pode ser sanado pela evocação de uma exceção.

```
def explo_funcao(*args, **kwargs):
    print(args)
    try:
        peso = kwargs['peso']
    except KeyError:
        pass
```