

Notas de aula de programação em Python

Encontro 8 - Funções parte 2

Prof. Louis Augusto

`louis.augusto@ifsc.edu.br`



INSTITUTO FEDERAL
SANTA CATARINA

Instituto Federal de Santa Catarina
Campus São José

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

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 variáveis 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 variáveis 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 variáveis 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 variáveis 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 variáveis 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 variáveis 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]
```

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- **Funções recursivas**
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

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 o retorno é feito.

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 o retorno é feito.

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 o retorno é feito.

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 o retorno é feito.

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 o retorno é feito.

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.

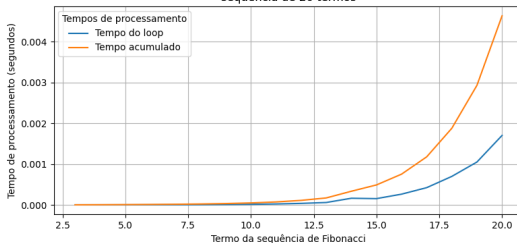
Isto faz com que o tempo de processamento aumente muito rapidamente depois de pouco tempo.

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

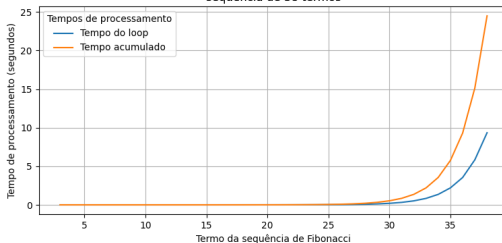


Funções recursivas

Tempo de processamento para determinação da sequência de Fibonacci
sequência de 20 termos



Tempo de processamento para determinação da sequência de Fibonacci
sequência de 38 termos



Funções recursivas

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



DESAFIO

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.

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

```
strings: print(len("Arroz"))
```

```
vetores: print(len([23, 43, 5, -9]))
```

```
Dicionários: (len({"mamão": 12, "pepino":14}))
```

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

strings: `print(len("Arroz"))`

vetores: `print(len([23, 43, 5, -9]))`

Dicionários: `(len({"mamão": 12, "pepino":14}))`

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

strings: `print(len("Arroz"))`

vetores: `print(len([23, 43, 5, -9]))`

Dicionários: `(len({"mamão": 12, "pepino":14}))`

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

strings: `print(len("Arroz"))`

vetores: `print(len([23, 43, 5, -9]))`

Dicionários: `(len({"mamão": 12, "pepino":14}))`

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

strings: `print(len("Arroz"))`

vetores: `print(len([23, 43, 5, -9]))`

Dicionários: `(len({"mamão": 12, "pepino":14}))`

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

strings: `print(len("Arroz"))`

vetores: `print(len([23, 43, 5, -9]))`

Dicionários: `(len({"mamão": 12, "pepino":14}))`

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

strings: `print(len("Arroz"))`

vetores: `print(len([23, 43, 5, -9]))`

Dicionários: `(len({"mamão": 12, "pepino":14}))`

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

Existem em python funções polimórficas *built-in*, que são funções que admitem entradas diferentes de dados.

Um exemplo típico é a função `len()`, que calcula métrica para diferentes variáveis:

strings: `print(len("Arroz"))`

vetores: `print(len([23, 43, 5, -9]))`

Dicionários: `(len({"mamão": 12, "pepino":14}))`

Observe que os tipos são diferentes, mas a função `len()` se adapta aos diferentes tipos de dados e retorna o tamanho de cada um deles (quantidade de caracteres, quantidade de elementos no vetor e no dicionário).

Uma outra função polimórfica é a função `max()` (e sua equivalente `min()`) são polimórficas. Se inserimos um vetor a função retorna o maior valor (maior número se as entradas são inteiras e letra mais adiantada no alfabeto se forem letras), mas se for um dicionário retorna a maior chave.

Vamos fazer uma função para retornar num dicionário o maior valor.

Funções polimórficas

```
def Funcao_Maior(Var):  
    if type(Var)==str:  
        print(max(Var))  
    '''  
    if isinstance(Var, str):  
        print(max(Var))  
    '''  
    if type(Var)==list:  
        print(max(Var))  
    if type(Var)==dict:  
        print("Maior chave: ", max(Var))  
        Maior = None  
        for valor in Var.values():  
            x = valor  
            if Maior == None:  
                Maior = valor  
            else:  
                if x>Maior:  
                    Maior = x  
        print("Maior valor: ",Maior)
```

```
if __name__=="__main__":  
    #Var = "Palavra"  
    #Var = [-2,4,3,-1,0,9]  
    Var = {"mamão": 12, "pepino":14, "maçã": 18}  
    Funcao_Maior(Var)
```

Funções polimórficas

```
def Funcao_Maior(Var):
    if type(Var)==str:
        print(max(Var))
    '''
    if isinstance(Var, str):
        print(max(Var))
    '''
    if type(Var)==list:
        print(max(Var))
    if type(Var)==dict:
        print("Maior chave: ", max(Var))
        Maior = None
        for valor in Var.values():
            x = valor
            if Maior == None:
                Maior = valor
            else:
                if x>Maior:
                    Maior = x
        print("Maior valor: ",Maior)
```

```
if __name__=="__main__":
    #Var = "Palavra"
    #Var = [-2,4,3,-1,0,9]
    Var = {"mamão": 12, "pepino":14, "maçã": 18}
    Funcao_Maior(Var)
```


1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- **Argumentos posicionais e nomeados**
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Argumentos posicionais e nomeados

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 posicionais e nomeados

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 posicionais e nomeados

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 posicionais e nomeados

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 posicionais e nomeados

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 posicionais e nomeados

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)  
MediaAluno(2, 5, 7, 8, Nome=None)
```

O que gera a saída:

A média do aluno Caio é 5.5

Argumentos posicionais e nomeados

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)  
MediaAluno(2, 5, 7, 8, Nome=None)
```

O que gera a saída:

A média do aluno Caio é 5.5

Argumentos posicionais e nomeados

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)  
MediaAluno(2, 5, 7, 8, Nome=None)
```

O que gera a saída:

A média do aluno Caio é 5.5

Argumentos posicionais e nomeados

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)  
MediaAluno(2, 5, 7, 8, Nome=None)
```

O que gera a saída:

A média do aluno Caio é 5.5

Argumentos posicionais e nomeados

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)  
MediaAluno(2, 5, 7, 8, Nome=None)
```

O que gera a saída:

A média do aluno Caio é 5.5

Argumentos posicionais e nomeados

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 posicionais e nomeados

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 posicionais e nomeados

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 posicionais e nomeados

Argumentos nomeados podem vir em qualquer posição depois dos nomeados, mas posicionais precisam vir na posição indicada no escopo da função.

```
def exibir_preco(nome, preco, custo = 12, valor = 14):  
    print(f"{nome} possui o preço {preco}")  
    print(f"Faturamento = {valor} e custo = {custo}")  
exibir_preco('batata', 5.5, valor = 16, custo = 14 )
```

Como o Python precisa saber qual argumento será passado para cada variável, os argumentos posicionais precisam estar em posições corretas, antes dos nomeados.

Todavia argumentos posicionais podem ter valores informados como se fossem nomeados, e neste caso podem vir em qualquer posição.

A mesma função poderia ser chamada como:

```
exibir_preco(custo=14,nome='batata',valor=16,preco=5.5)
```

E exibiria a mesma resposta.

Argumentos posicionais e nomeados

Argumentos nomeados podem vir em qualquer posição depois dos nomeados, mas posicionais precisam vir na posição indicada no escopo da função.

```
def exibir_preco(nome, preco, custo = 12, valor = 14):  
    print(f"{nome} possui o preço {preco}")  
    print(f"Faturamento = {valor} e custo = {custo}")  
exibir_preco('batata', 5.5, valor = 16, custo = 14 )
```

Como o Python precisa saber qual argumento será passado para cada variável, os argumentos posicionais precisam estar em posições corretas, antes dos nomeados.

Todavia argumentos posicionais podem ter valores informados como se fossem nomeados, e neste caso podem vir em qualquer posição.

A mesma função poderia ser chamada como:

```
exibir_preco(custo=14,nome='batata',valor=16,preco=5.5)
```

E exibiria a mesma resposta.

Argumentos posicionais e nomeados

Argumentos nomeados podem vir em qualquer posição depois dos nomeados, mas posicionais precisam vir na posição indicada no escopo da função.

```
def exibir_preco(nome, preco, custo = 12, valor = 14):  
    print(f"{nome} possui o preço {preco}")  
    print(f"Faturamento = {valor} e custo = {custo}")  
exibir_preco('batata', 5.5, valor = 16, custo = 14 )
```

Como o Python precisa saber qual argumento será passado para cada variável, os argumentos posicionais precisam estar em posições corretas, antes dos nomeados.

Todavia argumentos posicionais podem ter valores informados como se fossem nomeados, e neste caso podem vir em qualquer posição.

A mesma função poderia ser chamada como:

```
exibir_preco(custo=14,nome='batata',valor=16,preco=5.5)
```

E exibiria a mesma resposta.

Argumentos posicionais e nomeados

Argumentos nomeados podem vir em qualquer posição depois dos nomeados, mas posicionais precisam vir na posição indicada no escopo da função.

```
def exibir_preco(nome, preco, custo = 12, valor = 14):  
    print(f"{nome} possui o preço {preco}")  
    print(f"Faturamento = {valor} e custo = {custo}")  
exibir_preco('batata', 5.5, valor = 16, custo = 14 )
```

Como o Python precisa saber qual argumento será passado para cada variável, os argumentos posicionais precisam estar em posições corretas, antes dos nomeados.

Todavia argumentos posicionais podem ter valores informados como se fossem nomeados, e neste caso podem vir em qualquer posição.

A mesma função poderia ser chamada como:

```
exibir_preco(custo=14,nome='batata',valor=16,preco=5.5)
```

E exibiria a mesma resposta.

Passagem forçada com argumentos nomeados

O programador pode forçar o usuário a utilizar somente argumentos nomeados, mesmo definindo os argumentos como posicionais.

Para isto utiliza-se o operador `*` a partir do argumento posicional que se quer obrigar a usar como nomeado, e todos os outros são passados sem valor default.

Observe, vamos reescrever a função anterior sem argumentos nomeados:

```
def exibir_preco(nome, preco, custo, valor)
```

Os argumentos são posicionais e precisam estar escritos nesta ordem.

Porém se usarmos:

```
def exibir_preco(nome, preco, *, custo, valor)
```

os argumentos `custo` e `valor` precisarão ser nomeados.

```
def exibir_preco('batata', 5.5, 14, 16)
```

 gera erro de tipo.

Podemos usar, por exemplo:

```
def exibir_preco('batata', 5.5, valor = 16, custo = 14)
```

que o código irá funcionar.

Utilizando: `def exibir_preco(*, nome, preco, custo, valor)`

todos os argumentos deverão ser recebidos como nomeados.

Passagem forçada com argumentos nomeados

O programador pode forçar o usuário a utilizar somente argumentos nomeados, mesmo definindo os argumentos como posicionais.

Para isto utiliza-se o operador `*` a partir do argumento posicional que se quer obrigar a usar como nomeado, e todos os outros são passados sem valor default.

Observe, vamos reescrever a função anterior sem argumentos nomeados:

```
def exibir_preco(nome, preco, custo, valor)
```

Os argumentos são posicionais e precisam estar escritos nesta ordem.

Porém se usarmos:

```
def exibir_preco(nome, preco, *, custo, valor)
```

os argumentos `custo` e `valor` precisarão ser nomeados.

```
def exibir_preco('batata', 5.5, 14, 16)
```

 gera erro de tipo.

Podemos usar, por exemplo:

```
def exibir_preco('batata', 5.5, valor = 16, custo = 14)
```

que o código irá funcionar.

Utilizando: `def exibir_preco(*, nome, preco, custo, valor)`

todos os argumentos deverão ser recebidos como nomeados.

Passagem forçada com argumentos nomeados

O programador pode forçar o usuário a utilizar somente argumentos nomeados, mesmo definindo os argumentos como posicionais.

Para isto utiliza-se o operador `*` a partir do argumento posicional que se quer obrigar a usar como nomeado, e todos os outros são passados sem valor default.

Observe, vamos reescrever a função anterior sem argumentos nomeados:

```
def exibir_preco(nome, preco, custo, valor)
```

Os argumentos são posicionais e precisam estar escritos nesta ordem.

Porém se usarmos:

```
def exibir_preco(nome, preco, *, custo, valor)
```

os argumentos `custo` e `valor` precisarão ser nomeados.

```
def exibir_preco('batata', 5.5, 14, 16)
```

 gera erro de tipo.

Podemos usar, por exemplo:

```
def exibir_preco('batata', 5.5, valor = 16, custo = 14)
```

que o código irá funcionar.

Utilizando:

```
def exibir_preco(*, nome, preco, custo, valor)
```

todos os argumentos deverão ser recebidos como nomeados.

Passagem forçada com argumentos nomeados

O programador pode forçar o usuário a utilizar somente argumentos nomeados, mesmo definindo os argumentos como posicionais.

Para isto utiliza-se o operador `*` a partir do argumento posicional que se quer obrigar a usar como nomeado, e todos os outros são passados sem valor default.

Observe, vamos reescrever a função anterior sem argumentos nomeados:

```
def exibir_preco(nome, preco, custo, valor)
```

Os argumentos são posicionais e precisam estar escritos nesta ordem.

Porém se usarmos:

```
def exibir_preco(nome, preco, *, custo, valor)
```

os argumentos `custo` e `valor` precisarão ser nomeados.

```
def exibir_preco('batata', 5.5, 14, 16)
```

 gera erro de tipo.

Podemos usar, por exemplo:

```
def exibir_preco('batata', 5.5, valor = 16, custo = 14)
```

que o código irá funcionar.

Utilizando:

```
def exibir_preco(*, nome, preco, custo, valor)
```

todos os argumentos deverão ser recebidos como nomeados.

Passagem forçada com argumentos nomeados

O programador pode forçar o usuário a utilizar somente argumentos nomeados, mesmo definindo os argumentos como posicionais.

Para isto utiliza-se o operador `*` a partir do argumento posicional que se quer obrigar a usar como nomeado, e todos os outros são passados sem valor default.

Observe, vamos reescrever a função anterior sem argumentos nomeados:

```
def exibir_preco(nome, preco, custo, valor)
```

Os argumentos são posicionais e precisam estar escritos nesta ordem.

Porém se usarmos:

```
def exibir_preco(nome, preco, *, custo, valor)
```

os argumentos `custo` e `valor` precisarão ser nomeados.

```
def exibir_preco('batata', 5.5, 14, 16) gera erro de tipo.
```

Podemos usar, por exemplo:

```
def exibir_preco('batata', 5.5, valor = 16, custo = 14)
```

que o código irá funcionar.

Utilizando:

```
def exibir_preco(*, nome, preco, custo, valor)
```

todos os argumentos deverão ser recebidos como nomeados.

Passagem forçada com argumentos nomeados

O programador pode forçar o usuário a utilizar somente argumentos nomeados, mesmo definindo os argumentos como posicionais.

Para isto utiliza-se o operador `*` a partir do argumento posicional que se quer obrigar a usar como nomeado, e todos os outros são passados sem valor default.

Observe, vamos reescrever a função anterior sem argumentos nomeados:

```
def exibir_preco(nome, preco, custo, valor)
```

Os argumentos são posicionais e precisam estar escritos nesta ordem.

Porém se usarmos:

```
def exibir_preco(nome, preco, *, custo, valor)
```

os argumentos `custo` e `valor` precisarão ser nomeados.

```
def exibir_preco('batata', 5.5, 14, 16) gera erro de tipo.
```

Podemos usar, por exemplo:

```
def exibir_preco('batata', 5.5, valor = 16, custo = 14)
```

que o código irá funcionar.

Utilizando:

```
def exibir_preco(*, nome, preco, custo, valor)
```


todos os argumentos deverão ser recebidos como nomeados.

Passagem forçada com argumentos nomeados

O programador pode forçar o usuário a utilizar somente argumentos nomeados, mesmo definindo os argumentos como posicionais.

Para isto utiliza-se o operador `*` a partir do argumento posicional que se quer obrigar a usar como nomeado, e todos os outros são passados sem valor default.

Observe, vamos reescrever a função anterior sem argumentos nomeados:

```
def exibir_preco(nome, preco, custo, valor)
```

Os argumentos são posicionais e precisam estar escritos nesta ordem.

Porém se usarmos:

```
def exibir_preco(nome, preco, *, custo, valor)
```

os argumentos `custo` e `valor` precisarão ser nomeados.

```
def exibir_preco('batata', 5.5, 14, 16) gera erro de tipo.
```

Podemos usar, por exemplo:

```
def exibir_preco('batata', 5.5, valor = 16, custo = 14)
```

que o código irá funcionar.

Utilizando: `def exibir_preco(*, nome, preco, custo, valor)`

todos os argumentos deverão ser recebidos como nomeados.

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (`*args`)
- Argumentos nomeados dinâmicos (`**kwargs`)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

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 :

- Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
- 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 :

- Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
- 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 :

- Só há um argumento enviado para a função, a lista. Então `args` é uma tupla com somente um elemento, que é uma lista.
- 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)
```



1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (`*args`)
- Argumentos nomeados dinâmicos (`**kwargs`)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Argumentos posicionais dinâmicos

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. Rode o script e veja que `*args` é recebido na forma de tupla.

Argumentos posicionais dinâmicos

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. Rode o script e veja que `*args` é recebido na forma de tupla.

Argumentos posicionais dinâmicos

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. Rode o script e veja que `*args` é recebido na forma de tupla.

Argumentos posicionais dinâmicos

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. Rode o script e veja que `*args` é recebido na forma de tupla.

Argumentos posicionais dinâmicos

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. Rode o script e veja que `*args` é recebido na forma de tupla.

Argumentos posicionais desconhecidos

A palavra-chave `*args` é muito utilizada, mas poderia ser qualquer outra. Por exemplo:

```
def media_aluno(*notas, p, nome = None):  
    n = len(notas)+1  
    if nome==None:  
        print(f"Há {n} avaliações para o aluno.")  
    print(f"As notas do aluno são: {notas}, {p}.")  
    for nota in notas:  
        p+=nota  
    print(f"A média do aluno é {p/n}.")
```

No exemplo há um argumento posicional e outro nomeado. Todavia a passagem da função deve ser feita considerando `p` como nomeado, assim como `nome`.

```
media_aluno(2,6,4,6, p = 7, nome = "Carlos")
```

Resposta:

```
As notas do aluno são: (2, 6, 4, 6), 7.  
A média do aluno é 5.0.
```

Argumentos posicionais desconhecidos

A palavra-chave `*args` é muito utilizada, mas poderia ser qualquer outra. Por exemplo:

```
def media_aluno(*notas, p, nome = None):  
    n = len(notas)+1  
    if nome==None:  
        print(f"Há {n} avaliações para o aluno.")  
    print(f"As notas do aluno são: {notas}, {p}.")  
    for nota in notas:  
        p+=nota  
    print(f"A média do aluno é {p/n}.")
```

No exemplo há um argumento posicional e outro nomeado. Todavia a passagem da função deve ser feita considerando `p` como nomeado, assim como `nome`.

```
media_aluno(2,6,4,6, p = 7, nome = "Carlos")
```

Resposta:

```
As notas do aluno são: (2, 6, 4, 6), 7.  
A média do aluno é 5.0.
```

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Argumentos com palavras-chave dinâmicas

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 palavras-chave dinâmicas

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 palavras-chave dinâmicas

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 palavras-chave dinâmicas

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 palavras-chave desconhecidas

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.

Argumentos com palavras-chave desconhecidas

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.

Argumentos com palavras-chave desconhecidas

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.

Argumentos com palavras-chave desconhecidas

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.

Argumentos com palavras-chave desconhecidas

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como argumento 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 palavras-chave desconhecidas

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como argumento 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 palavras-chave desconhecidas

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como argumento 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 palavras-chave desconhecidas

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como argumento 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 palavras-chave desconhecidas

Podemos ainda melhorar a funcionalidade da nossa função.

Suponha que eu não saiba o que foi enviado como argumento 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
```

Argumentos com palavra-chave desconhecidos

De forma mais geral, temos a função e seu driver abaixo:

```
def explo_funcao(*args, **kwargs):
    print(args)
    print("Número de argumentos nomeados:", len(kwargs))
    print("Chaves encontradas:", end = '')
    for chave in kwargs.keys():
        print(chave, end = ' ')
    print()
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
        print("Sobrenome do usuário: ",sbr)
    try:
        peso=kwargs['peso']
    except KeyError:
        pass

#Driver:
lista = [1,2,3] #lista
lista2 = (10,20,30) #tupla
num = 78
explo_funcao(lista,lista2,num,nome="Caio",sobrenome="Martins",idade=19)
```

Observe como os elementos de `*args` se tornam elementos de uma tupla e que todos os métodos de dicionário passam a serem válidos para `**kwargs`.

Argumentos com palavra-chave desconhecidos

De forma mais geral, temos a função e seu driver abaixo:

```
def explo_funcao(*args, **kwargs):
    print(args)
    print("Número de argumentos nomeados:", len(kwargs))
    print("Chaves encontradas:", end = '')
    for chave in kwargs.keys():
        print(chave, end = ' ')
    print()
    sbr = kwargs.get('sobrenome')
    if sbr !=None:
        print("Sobrenome do usuário: ",sbr)
    try:
        peso=kwargs['peso']
    except KeyError:
        pass

#Driver:
lista = [1,2,3] #lista
lista2 = (10,20,30) #tupla
num = 78
explo_funcao(lista,lista2,num,nome="Caio",sobrenome="Martins",idade=19)
```

Observe como os elementos de `*args` se tornam elementos de uma tupla e que todos os métodos de dicionário passam a serem válidos para `**kwargs`.

Argumentos mistos

Por último, podemos utilizar argumentos posicionais, argumentos `*args` e `**kwargs`. Observe:

```
def conta_aluno(nome, peso, *args,**kwargs):  
    print("Nome: {0}, peso = {1}".format(nome, peso))  
    print("args: ", args)  
    print("kwargs: ", kwargs)  
    for arg in args:  
        print(arg)  
    for kwarg in kwargs.items():  
        print(kwarg)  
  
#Driver:  
conta_aluno("Marcio", "80kg",5,3,2,a=1, b=2, c=9)
```

que gera a saída:

```
Nome: Marcio, peso = 80kg  
args: (5, 3, 2)  
kwargs: {'a': 1, 'b': 2, 'c': 9}  
5  
3  
2  
( 'a', 1)  
( 'b', 2)  
( 'c', 9)
```

Argumentos mistos

O script abaixo também funciona:

```
def conta_aluno(nome, peso='80kg', *args,**kwargs):  
    print("Nome: {0}, peso = {1}".format(nome, peso))  
    print("args: ", args)  
    print("kwargs: ", kwargs)  
    for arg in args:  
        print(arg)  
    for karg in kwargs.items():  
        print(karg)  
  
#Driver:  
conta_aluno("Marcio", 70,5,3,2,a=1, b=2, c=9)
```

que gera a saída:

```
Nome: Marcio, peso = 70  
args: (5, 3, 2)  
kwargs: {'a': 1, 'b': 2, 'c': 9}  
5  
3  
2  
( 'a', 1)  
( 'b', 2)  
( 'c', 9)
```

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Aninhamento de funções

É possível em python aninhar funções, colocar funções dentro de funções.

```
def deposito_dinheiro(moeda, valor):  
    def deposito_reais(valor):  
        print(f"Depositado {valor} em reais")  
    def deposito_euros(valor):  
        print(f"Depositado {valor} em em euros")  
    if moeda == "RBr":  
        deposito_reais(valor)  
    elif moeda == "EU":  
        deposito_euros(valor)  
    deposito_dinheiro('RBr', 1000)
```

O problema desta construção é que as funções internas, `deposito_euros(valor)` e `deposito_reais(valor)` não podem ser acessadas de fora da função `deposito_dinheiro(moeda, valor)`.

Não há como usar, por exemplo:

`deposito_dinheiro('RBr', 1000).deposito_reais(valor)`. Isto não funciona.

Aninhamento de funções

É possível em python aninhar funções, colocar funções dentro de funções.

```
def deposito_dinheiro(moeda, valor):  
    def deposito_reais(valor):  
        print(f"Depositado {valor} em reais")  
    def deposito_euros(valor):  
        print(f"Depositado {valor} em em euros")  
    if moeda == "RBr":  
        deposito_reais(valor)  
    elif moeda == "EU":  
        deposito_euros(valor)  
    deposito_dinheiro('RBr', 1000)
```

O problema desta construção é que as funções internas, `deposito_euros(valor)` e `deposito_reais(valor)` não podem ser acessadas de fora da função `deposito_dinheiro(moeda, valor)`.

Não há como usar, por exemplo:

`deposito_dinheiro('RBr', 1000).deposito_reais(valor)`. Isto não funciona.

Aninhamento de funções

É possível em python aninhar funções, colocar funções dentro de funções.

```
def deposito_dinheiro(moeda, valor):  
    def deposito_reais(valor):  
        print(f"Depositado {valor} em reais")  
    def deposito_euros(valor):  
        print(f"Depositado {valor} em em euros")  
    if moeda == "RBr":  
        deposito_reais(valor)  
    elif moeda == "EU":  
        deposito_euros(valor)  
    deposito_dinheiro('RBr', 1000)
```

O problema desta construção é que as funções internas, `deposito_euros(valor)` e `deposito_reais(valor)` não podem ser acessadas de fora da função `deposito_dinheiro(moeda, valor)`.

Não há como usar, por exemplo:

`deposito_dinheiro('RBr', 1000).deposito_reais(valor)`. Isto não funciona.

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- **Retorno de referência de funções**
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Retorno de referência de funções

Observe a sequência de código:

```
1 def pai(numero):
2     def filho_1():
3         print("Chamou filho 1")
4     def filho_2():
5         print("Chamou filho 2")
6     if numero==1:
7         return filho_1
8     else:
9         return filho_2
10 resultado = pai(1)
11 resultado()
```

Como resposta temos: Chamou filho 1

Na linha 7 foi retornada uma referência da função `filho_1`, veja que não há parênteses após a função, `filho_1()`, logo ela não é chamada, somente referenciada.

Na linha 10 a variável `resultado` é retornada da função, e recebe uma referência da função `filho_1`, então ela torna-se a função `filho_1()` e a partir de então pode ser executada como função.

Veja na linha abaixo a chamada: `resultado()`.

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Decoração de funções

Vamos supor que queiramos chamar 3 funções em sequência:

- Ação antes.
- Dar parabéns.
- Ação depois.

A forma mais ingênua seria fazer:

```
def antes():  
    print("Antes da função")  
def depois():  
    print("Depois da função")  
def parabenizar():  
    print("Parabens")  
print("Primeira forma:\nReferência direta às funções:")  
antes()  
parabenizar()  
depois()
```

Decoração de funções

Vamos supor que queiramos chamar 3 funções em sequência:

- Ação antes.
- Dar parabéns.
- Ação depois.

A forma mais ingênua seria fazer:

```
def antes():  
    print("Antes da função")  
def depois():  
    print("Depois da função")  
def parabenizar():  
    print("Parabens")  
print("Primeira forma:\nReferência direta às funções:")  
antes()  
parabenizar()  
depois()
```

Decoração de funções

Vamos supor que queiramos chamar 3 funções em sequência:

- Ação antes.
- Dar parabéns.
- Ação depois.

A forma mais ingênua seria fazer:

```
def antes():  
    print("Antes da função")  
def depois():  
    print("Depois da função")  
def parabenizar():  
    print("Parabens")  
print("Primeira forma:\nReferência direta às funções:")  
antes()  
parabenizar()  
depois()
```

Decoração de funções

Vamos supor que queiramos chamar 3 funções em sequência:

- Ação antes.
- Dar parabéns.
- Ação depois.

A forma mais ingênua seria fazer:

```
def antes():  
    print("Antes da função")  
def depois():  
    print("Depois da função")  
def parabenizar():  
    print("Parabens")  
print("Primeira forma:\nReferência direta às funções:")  
antes()  
parabenizar()  
depois()
```


Decoração de funções

Vamos supor que queiramos chamar 3 funções em sequência:

- Ação antes.
- Dar parabéns.
- Ação depois.

A forma mais ingênua seria fazer:

```
def antes():  
    print("Antes da função")  
def depois():  
    print("Depois da função")  
def parabenizar():  
    print("Parabens")  
print("Primeira forma:\nReferência direta às funções:")  
antes()  
parabenizar()  
depois()
```

Decoração de funções

A segunda forma seria usar referência à função.

```
def antes():  
    print("Antes da função")  
def depois():  
    print("Depois da função")  
def parabenizar():  
    print("Parabens")  
def meu_decorador(funcao): #Envia a referencia da função  
    def envelope():  
        antes()  
        funcao() #Agora não é referencia, executa a função  
        depois()  
    return envelope #Retorna somente a referencia da função.  
resultado=meu_decorador(parabenizar)  
resultado()
```

Observe que a função `meu_decorador` recebe uma referencia de função genérica e devolve a referência da função envelopadora.

Existe um atalho para esta operação, tornando-a muito mais limpa, o **decorador de função**.

Decoração de funções

A segunda forma seria usar referência à função.

```
def antes():  
    print("Antes da função")  
def depois():  
    print("Depois da função")  
def parabenizar():  
    print("Parabens")  
def meu_decorador(funcao): #Envia a referencia da função  
    def envelope():  
        antes()  
        funcao() #Agora não é referencia, executa a função  
        depois()  
    return envelope #Retorna somente a referencia da função.  
resultado=meu_decorador(parabenizar)  
resultado()
```

Observe que a função `meu_decorador` recebe uma referencia de função genérica e devolve a referência da função envelopadora.

Existe um atalho para esta operação, tornando-a muito mais limpa, o **decorador de função**.

Decoração de funções

A segunda forma seria usar referência à função.

```
def antes():
    print("Antes da função")
def depois():
    print("Depois da função")
def parabenizar():
    print("Parabens")
def meu_decorador(funcao): #Envia a referencia da função
    def envelope():
        antes()
        funcao() #Agora não é referencia, executa a função
        depois()
    return envelope #Retorna somente a referencia da função.
resultado=meu_decorador(parabenizar)
resultado()
```

Observe que a função `meu_decorador` recebe uma referencia de função genérica e devolve a referência da função envelopadora.

Existe um atalho para esta operação, tornando-a muito mais limpa, o **decorador de função**.

Decoração de funções

Construção do decorador:

```
1 def meu_decorator(funcao):
2     def envelope():
3         antes()
4         funcao()
5         depois()
6     return envelope
7
8 @meu_decorator
9 def dar_parabens():
10     print("Parabens 2.")
11
12 dar_parabens()
```

A linha 8 substitui a linha

`resultado=meu_decorator(dar_parabens)` como foi utilizado na forma anterior. Então antes de definir a função (na linha 8) adiciona-se a ela a funcionalidade (linhas 3 e 5), no caso aplicação das funções `antes()` e `depois()`.

Chamamos isto de envelopar a função

Geralmente isto é usado para ações que precisam ser repetidas várias vezes em diferentes funções, como contagem de tempo para execução de uma função. Coloca-se uma função que inicia a contagem de tempo, chama a função que se quer medir, e por fim coloca-se um terminador de contagem de tempo. Então sabemos o tempo que a função custou para ser executada.

Exemplo

```
import time

def MedeTempo(func):
    def envelope(*args, **kwargs):
        tempo_inicial = time.perf_counter()
        resultado = func(*args, **kwargs)
        tempo_final = time.perf_counter()
        DeltaTempo = tempo_final-tempo_inicial
        strg = "Tempo de {0}: {1:.5f}".format(func.__name__, DeltaTempo)
        print(strg)
        return resultado
    return envelope
```

```
@MedeTempo
def tempo_laco(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(key, value)
    for _ in range(int(1e06)):
        pass
    print("Fim da função tempo_laco")
tempo_laco(2,4,nome = "masr")
```

Indiretamente o decorador calcula o custou de tempo da função `tempo_laco()`.

Se não quisermos esta funcionalidade basta comentar a linha `@MedeTempo`.

Inclua na função `tempo_laco` a linha `time.sleep(3)`, que faz com o que o programa interrompa sua execução por 3 segundos, e veja a variação temporal.

Exemplo

```
import time

def MedeTempo(func):
    def envelope(*args, **kwargs):
        tempo_inicial = time.perf_counter()
        resultado = func(*args, **kwargs)
        tempo_final = time.perf_counter()
        DeltaTempo = tempo_final-tempo_inicial
        strg = "Tempo de {0}: {1:.5f}".format(func.__name__, DeltaTempo)
        print(strg)
        return resultado
    return envelope
```

```
@MedeTempo
def tempo_laco(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(key, value)
    for _ in range(int(1e06)):
        pass
    print("Fim da função tempo_laco")
tempo_laco(2,4,nome = "masr")
```

Indiretamente o decorador calcula o custou de tempo da função `tempo_laco()`.

Se não quisermos esta funcionalidade basta comentar a linha `@MedeTempo`.

Inclua na função `tempo_laco` a linha `time.sleep(3)`, que faz com o que o programa interrompa sua execução por 3 segundos, e veja a variação temporal.

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

Introdução às dicas de tipo

Dicas de tipo são um modo de incluir anotações sobre os tipos esperados de **variáveis**, **parâmetros de função** e tipos de **retorno de funções**.

Dicas de tipo servem para o programador deixar registrado qual o tipo esperado em determinada parte do programa escrito em Python.

Python é uma linguagem não tipada, não exige do programador especifique o tipo de variável. A única maneira de se descobrir o tipo de uma variável é por meio de uma inspeção, tipo `print(type(nome_da_variável))`.

Para programas pequenos é até vantajoso que não se trabalhe com uma linguagem tipada, mas para programas grandes pode ser um problema, pois pode-se ter uma variável que seja usada para referenciar vários tipos diferentes, por vezes até de forma acidental, gerando problemas difíceis de resolver.

Introdução às dicas de tipo

Dicas de tipos são os tipos que o programador espera que a variável referencie. Pelo menos até a versão 3.7 são opcionais, e não afetam o resultado em tempo de execução. Se o programador esperar uma variável `int` na dica de tipo e o resultado for um `float` o programa vai trabalhar com um `float`, independentemente da dica de tipo.

Tenha em mente que:

Pode-se usar dicas de tipo onde quiser, mas elas não são obrigatórias.

1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- **Aparência das Dicas de Tipo**
- Verificador de tipo

Aparência das dicas de tipo

As dicas de tipo são acrescentadas em uma linha de código na qual uma variável ou função é declarada.

Símbolos usados:

- : (dois pontos) sinalizam o início de uma dica de tipo para uma variável ou parâmetro de função.
- > (seta) sinaliza o início de uma dica de tipo para o tipo de retorno de uma função.

Exemplo:

Função sem dica: `def repetir(item, vezes):`

Função com dica: `def repetir(item: any, vezes:int)->list[any]:`

Um exemplo do que poderia ser a função `repetir`:

```
def repetir(item: any, vezes:int)->list[any]:  
    return [item for i in range(vezes)]  
print(repetir("maçã", 3))
```

Seu retorno é uma lista tipo: `['maçã', 'maçã', 'maçã']`

Aparência das dicas de tipo

As dicas de tipo são acrescentadas em uma linha de código na qual uma variável ou função é declarada.

Símbolos usados:

- : (dois pontos) sinalizam o início de uma dica de tipo para uma variável ou parâmetro de função.
- > (seta) sinaliza o início de uma dica de tipo para o tipo de retorno de uma função.

Exemplo:

Função sem dica: `def repetir(item, vezes):`

Função com dica: `def repetir(item: any, vezes:int)->list[any]:`

Um exemplo do que poderia ser a função `repetir`:

```
def repetir(item: any, vezes:int)->list[any]:  
    return [item for i in range(vezes)]  
print(repetir("maçã", 3))
```

Seu retorno é uma lista tipo: `['maçã', 'maçã', 'maçã']`

Aparência das dicas de tipo

As dicas de tipo são acrescentadas em uma linha de código na qual uma variável ou função é declarada.

Símbolos usados:

- : (dois pontos) sinalizam o início de uma dica de tipo para uma variável ou parâmetro de função.
- > (seta) sinaliza o início de uma dica de tipo para o tipo de retorno de uma função.

Exemplo:

Função sem dica: `def repetir(item, vezes):`

Função com dica: `def repetir(item: any, vezes:int)->list[any]:`

Um exemplo do que poderia ser a função `repetir`:

```
def repetir(item: any, vezes:int)->list[any]:  
    return [item for i in range(vezes)]  
print(repetir("maçã", 3))
```

Seu retorno é uma lista tipo: `['maçã', 'maçã', 'maçã']`

Aparência das dicas de tipo

As dicas de tipo são acrescentadas em uma linha de código na qual uma variável ou função é declarada.

Símbolos usados:

- : (dois pontos) sinalizam o início de uma dica de tipo para uma variável ou parâmetro de função.
- > (seta) sinaliza o início de uma dica de tipo para o tipo de retorno de uma função.

Exemplo:

Função sem dica: `def repetir(item, vezes):`

Função com dica: `def repetir(item: any, vezes:int)->list[any]:`

Um exemplo do que poderia ser a função `repetir`:

```
def repetir(item: any, vezes:int)->list[any]:  
    return [item for i in range(vezes)]  
print(repetir("maçã", 3))
```

Seu retorno é uma lista tipo: `['maçã', 'maçã', 'maçã']`

Aparência das dicas de tipo

As dicas de tipo são acrescentadas em uma linha de código na qual uma variável ou função é declarada.

Símbolos usados:

- : (dois pontos) sinalizam o início de uma dica de tipo para uma variável ou parâmetro de função.
- > (seta) sinaliza o início de uma dica de tipo para o tipo de retorno de uma função.

Exemplo:

Função sem dica: `def repetir(item, vezes):`

Função com dica: `def repetir(item: any, vezes:int)->list[any]:`

Um exemplo do que poderia ser a função `repetir`:

```
def repetir(item: any, vezes:int)->list[any]:  
    return [item for i in range(vezes)]  
print(repetir("maçã", 3))
```

Seu retorno é uma lista tipo: `['maçã', 'maçã', 'maçã']`

Aparência das dicas de tipo

As dicas de tipo são acrescentadas em uma linha de código na qual uma variável ou função é declarada.

Símbolos usados:

- : (dois pontos) sinalizam o início de uma dica de tipo para uma variável ou parâmetro de função.
- > (seta) sinaliza o início de uma dica de tipo para o tipo de retorno de uma função.

Exemplo:

Função sem dica: `def repetir(item, vezes):`

Função com dica: `def repetir(item: any, vezes:int)->list[any]:`

Um exemplo do que poderia ser a função `repetir`:

```
def repetir(item: any, vezes:int)->list[any]:  
    return [item for i in range(vezes)]  
print(repetir("maçã", 3))
```

Seu retorno é uma lista tipo: `['maçã', 'maçã', 'maçã']`

Aparência das dicas de tipo

As dicas de tipo são acrescentadas em uma linha de código na qual uma variável ou função é declarada.

Símbolos usados:

- : (dois pontos) sinalizam o início de uma dica de tipo para uma variável ou parâmetro de função.
- > (seta) sinaliza o início de uma dica de tipo para o tipo de retorno de uma função.

Exemplo:

Função sem dica: `def repetir(item, vezes):`

Função com dica: `def repetir(item: any, vezes:int)->list[any]:`

Um exemplo do que poderia ser a função `repetir`:

```
def repetir(item: any, vezes:int)->list[any]:  
    return [item for i in range(vezes)]  
print(repetir("maçã", 3))
```

Seu retorno é uma lista tipo: `['maçã', 'maçã', 'maçã']`

Aparência das dicas de tipo

Podemos usar dicas de tipo também para variáveis. Suponha que a função `repetir` seja chamada num arquivo sendo que sua definição esteja num módulo separado.

Podemos fazer:

Variável sem dica: `lista = repetir("maçã", 3)`

Variável com dica: `lista2: list[str] = repetir("maçã", 3)`

Da mesma forma que em funções, a variável é definida (no caso `lista2`) e usa-se o comando `:` para indicar ao lado o tipo que se espera obter.

No caso a variável retornada é `lista2`, com o tipo esperado uma lista de strings. Vale salientar que funciona normalmente o código se fizermos:

```
lista2: list[float] = repetir("maçã", 3)
```

Fica sem sentido a dica de tipo no caso acima, pois a lista é de strings, mas o Python não faz a verificação de tipo. A dica de tipo é como se fosse um comentário, mas que não pode ser escrito errado.

O uso, todavia, mais comum para dicas de tipo em variável é:

```
nome:str="Clésio"
```

```
salario:float=3000.00
```

Aparência das dicas de tipo

Podemos usar dicas de tipo também para variáveis. Suponha que a função `repetir` seja chamada num arquivo sendo que sua definição esteja num módulo separado.

Podemos fazer:

Variável sem dica: `lista = repetir("maçã", 3)`

Variável com dica: `lista2: list[str] = repetir("maçã", 3)`

Da mesma forma que em funções, a variável é definida (no caso `lista2`) e usa-se o comando `:` para indicar ao lado o tipo que se espera obter.

No caso a variável retornada é `lista2`, com o tipo esperado uma lista de strings. Vale salientar que funciona normalmente o código se fizermos:

```
lista2: list[float] = repetir("maçã", 3)
```

Fica sem sentido a dica de tipo no caso acima, pois a lista é de strings, mas o Python não faz a verificação de tipo. A dica de tipo é como se fosse um comentário, mas que não pode ser escrito errado.

O uso, todavia, mais comum para dicas de tipo em variável é:

```
nome:str="Clésio"
```

```
salario:float=3000.00
```

Aparência das dicas de tipo

Podemos usar dicas de tipo também para variáveis. Suponha que a função `repetir` seja chamada num arquivo sendo que sua definição esteja num módulo separado.

Podemos fazer:

Variável sem dica: `lista = repetir("maçã", 3)`

Variável com dica: `lista2: list[str] = repetir("maçã", 3)`

Da mesma forma que em funções, a variável é definida (no caso `lista2`) e usa-se o comando `:` para indicar ao lado o tipo que se espera obter.

No caso a variável retornada é `lista2`, com o tipo esperado uma lista de strings. Vale salientar que funciona normalmente o código se fizermos:

```
lista2: list[float] = repetir("maçã", 3)
```

Fica sem sentido a dica de tipo no caso acima, pois a lista é de strings, mas o Python não faz a verificação de tipo. A dica de tipo é como se fosse um comentário, mas que não pode ser escrito errado.

O uso, todavia, mais comum para dicas de tipo em variável é:

```
nome:str="Clésio"
```

```
salario:float=3000.00
```

Aparência das dicas de tipo

Podemos usar dicas de tipo também para variáveis. Suponha que a função `repetir` seja chamada num arquivo sendo que sua definição esteja num módulo separado.

Podemos fazer:

Variável sem dica: `lista = repetir("maçã", 3)`

Variável com dica: `lista2: list[str] = repetir("maçã", 3)`

Da mesma forma que em funções, a variável é definida (no caso `lista2`) e usa-se o comando `:` para indicar ao lado o tipo que se espera obter.

No caso a variável retornada é `lista2`, com o tipo esperado uma lista de strings. Vale salientar que funciona normalmente o código se fizermos:

```
lista2: list[float] = repetir("maçã", 3)
```

Fica sem sentido a dica de tipo no caso acima, pois a lista é de strings, mas o Python não faz a verificação de tipo. A dica de tipo é como se fosse um comentário, mas que não pode ser escrito errado.

O uso, todavia, mais comum para dicas de tipo em variável é:

```
nome:str="Clésio"
```

```
salario:float=3000.00
```

Aparência das dicas de tipo

Podemos usar dicas de tipo também para variáveis. Suponha que a função `repetir` seja chamada num arquivo sendo que sua definição esteja num módulo separado.

Podemos fazer:

Variável sem dica: `lista = repetir("maçã", 3)`

Variável com dica: `lista2: list[str] = repetir("maçã", 3)`

Da mesma forma que em funções, a variável é definida (no caso `lista2`) e usa-se o comando `:` para indicar ao lado o tipo que se espera obter.

No caso a variável retornada é `lista2`, com o tipo esperado uma lista de strings. Vale salientar que funciona normalmente o código se fizermos:

```
lista2: list[float] = repetir("maçã", 3)
```

Fica sem sentido a dica de tipo no caso acima, pois a lista é de strings, mas o Python não faz a verificação de tipo. A dica de tipo é como se fosse um comentário, mas que não pode ser escrito errado.

O uso, todavia, mais comum para dicas de tipo em variável é:

```
nome:str="Clésio"
```

```
salario:float=3000.00
```

Aparência das dicas de tipo

Podemos usar dicas de tipo também para variáveis. Suponha que a função `repetir` seja chamada num arquivo sendo que sua definição esteja num módulo separado.

Podemos fazer:

Variável sem dica: `lista = repetir("maçã", 3)`

Variável com dica: `lista2: list[str] = repetir("maçã", 3)`

Da mesma forma que em funções, a variável é definida (no caso `lista2`) e usa-se o comando `:` para indicar ao lado o tipo que se espera obter.

No caso a variável retornada é `lista2`, com o tipo esperado uma lista de strings. Vale salientar que funciona normalmente o código se fizermos:

```
lista2: list[float] = repetir("maçã", 3)
```

Fica sem sentido a dica de tipo no caso acima, pois a lista é de strings, mas o Python não faz a verificação de tipo. A dica de tipo é como se fosse um comentário, mas que não pode ser escrito errado.

O uso, todavia, mais comum para dicas de tipo em variável é:

```
nome:str="Clésio"  
salario:float=3000.00
```


1 Funções - revisão e novidades

- Revisão dos princípios básicos das funções
- Funções recursivas
- Funções polimórficas
- Argumentos posicionais e nomeados
- Empacotamento e desempacotamento de vetores

2 Funções com argumentos dinâmicos

- Argumentos posicionais dinâmicos (*args)
- Argumentos nomeados dinâmicos (**kwargs)

3 Decoradores de funções

- Aninhamento de funções e referências
- Retorno de referência de funções
- Decoradorando uma função

4 Dicas de tipo - *Type hints*

- O que são dicas de tipo
- Aparência das Dicas de Tipo
- Verificador de tipo

O verificador mypy

O `mypy` é um verificador de tipos do python, que deve ser instalado (*no ubuntu com o comando* `sudo apt install mypy`) e serve para verificar se todos os tipos determinados nas dicas de tipo estão corretos.

No terminal executa-se `mypy exemplo.py` ao invés de `python3 exemplo.py`. O `mypy` não executa o programa e gera uma resposta do programa, ao contrário vai simplesmente verificar se todos os tipos estão consistentes com as dicas e gerar uma saída indicando `success` se tudo estiver correto. Então se uma dica de tipo indicar que será criada uma lista com strings e for retornada uma lista com floats o `mypy` exibirá falha, porém o python vai executar normalmente o código, ignorando as dicas de tipo.