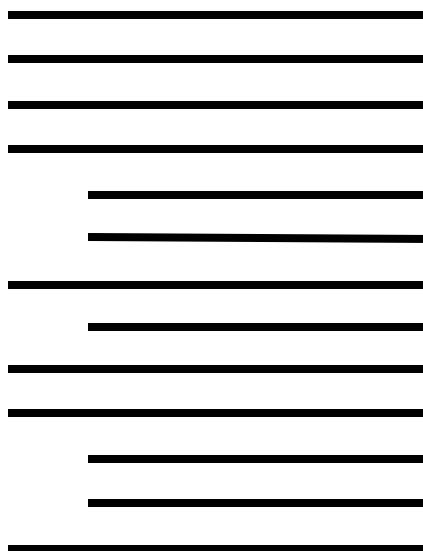


Subprogramação



O que vimos até agora

- Programas que usam **apenas sequência, repetição e decisão**
- Capacidade de resolver diversos problemas, mas **difícil de resolver problemas grandes**
 - Em algumas situações, é necessário **repetir o mesmo trecho de código** em diversos pontos do programa



Exemplo 1

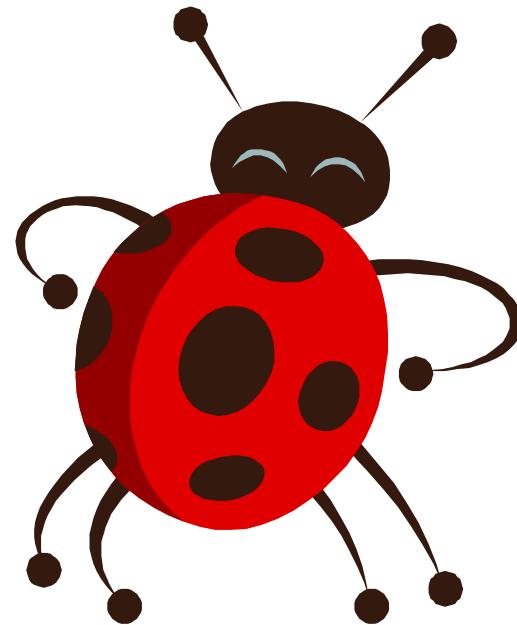
```
a = [1, 2, 3, 4, 5]
soma = 0
for e in a:
    soma += e
media = soma/len(a)
print(media)
```

```
b = [10, 20, 30, 40]
soma = 0
for e in b:
    soma += e
media = soma/len(b)
print(media)
```

Trecho se
repete 2
vezes

Problemas desta “repetição”

- Programa muito grande, porque tem várias **partes repetidas**
- **Defeitos ficam difíceis de corrigir** (e se eu esquecer de corrigir o defeito em uma das N repetições daquele trecho de código?)



Solução: subprogramação

- Definir o trecho de código que se repete como uma **função** que é chamada no programa
- A função é definida uma única vez, e chamada várias vezes dentro do programa
- Antes: um programa gigante



- Depois: vários programas menores



Voltando ao exemplo 1

```
def calcula_media(v)
    soma = 0
    for e in v:
        soma += e
    media = soma/len(v)
    return media
```

```
a = [1, 2, 3, 4, 5]
print(calcula_media(a))
b = [10, 20, 30, 40]
print(calcula_media(b))
```

Definição da função

Chamada da função

Chamada da função

Vantagens

- Economia de código
 - Quanto mais repetição, mais economia
- Facilidade na correção de defeitos
 - Corrigir o defeito em um único local
- Legibilidade do código
 - Podemos dar nomes mais intuitivos a blocos de código
 - É como se criássemos nossos próprios comandos
- Melhor tratamento de complexidade
 - Estratégia de “dividir para conquistar” nos permite lidar melhor com a complexidade de programas grandes
 - Abordagem *top-down* ajuda a pensar!

Fluxo de execução

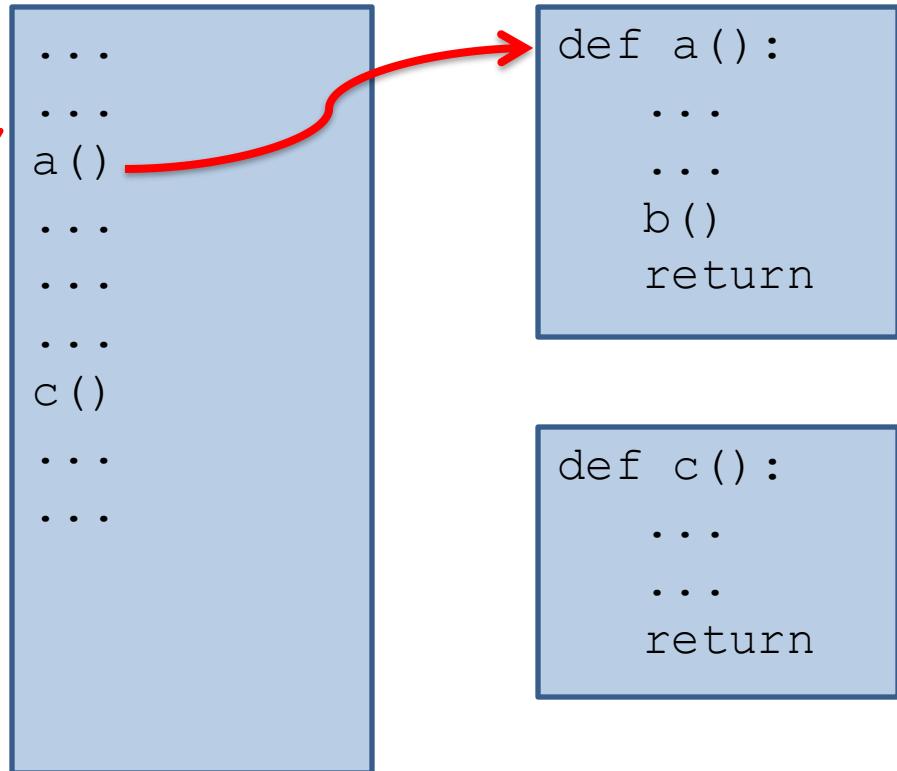
```
...  
...  
a()  
...  
...  
...  
c()
```

```
def a():  
    ...  
    ...  
    b()  
    return
```

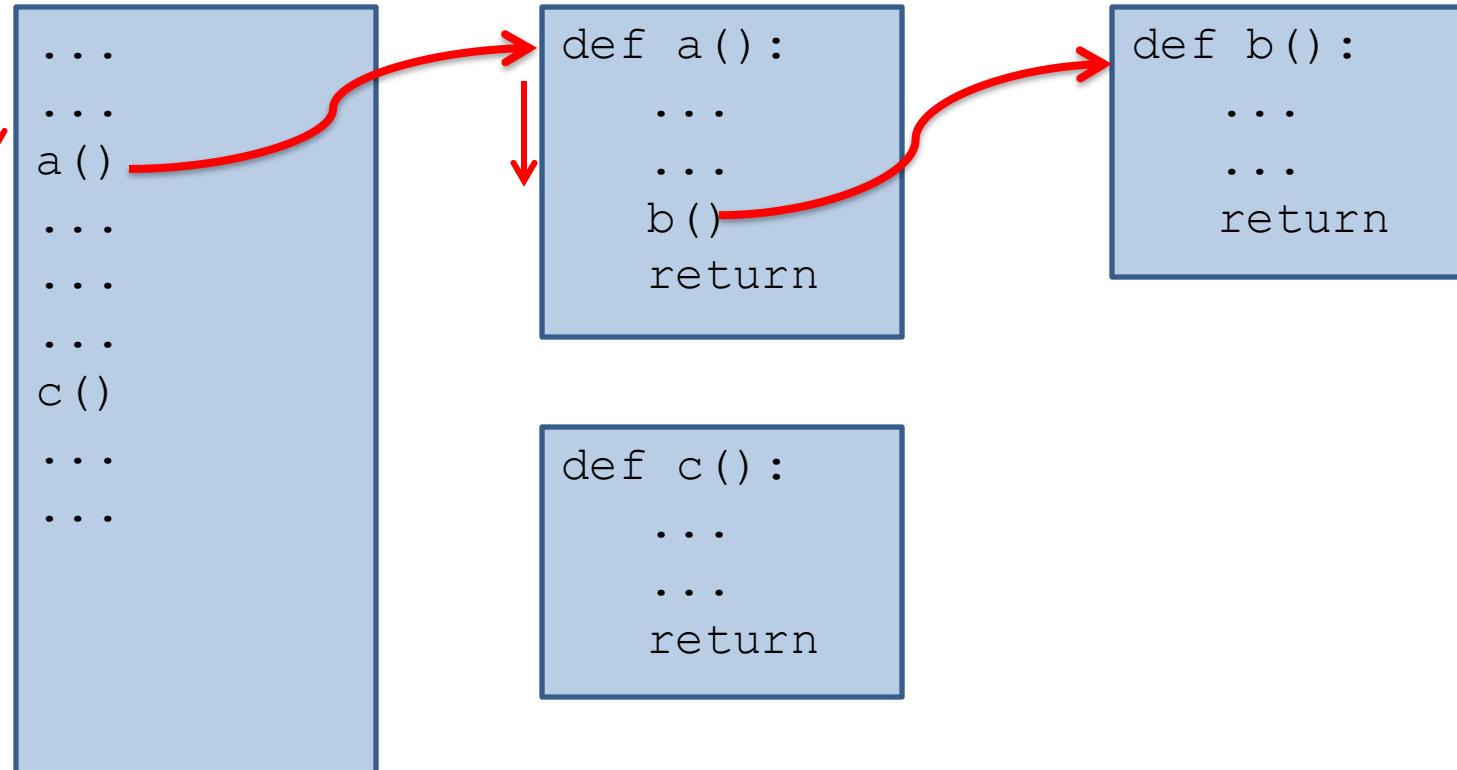
```
def b():  
    ...  
    ...  
    return
```

```
def c():  
    ...  
    ...  
    return
```

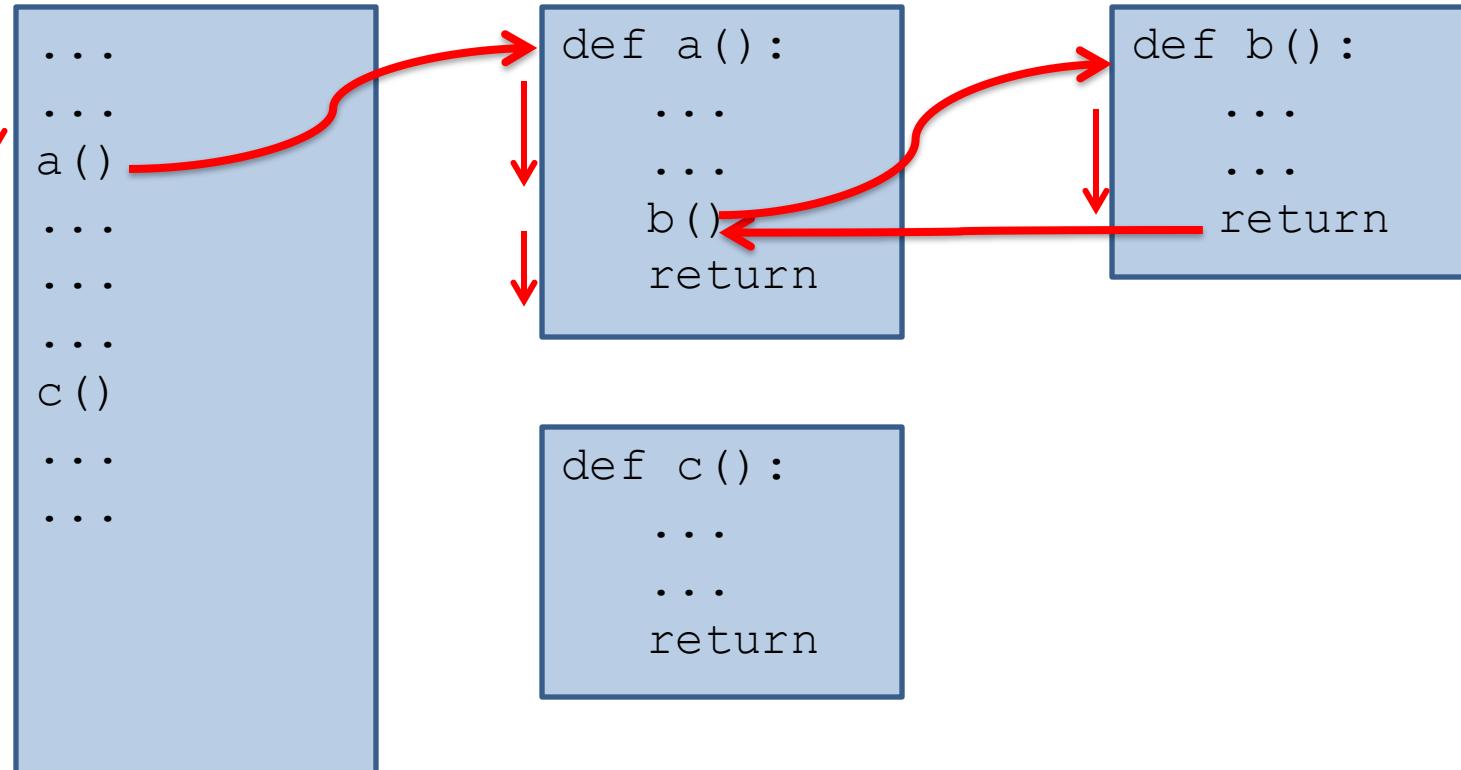
Fluxo de execução



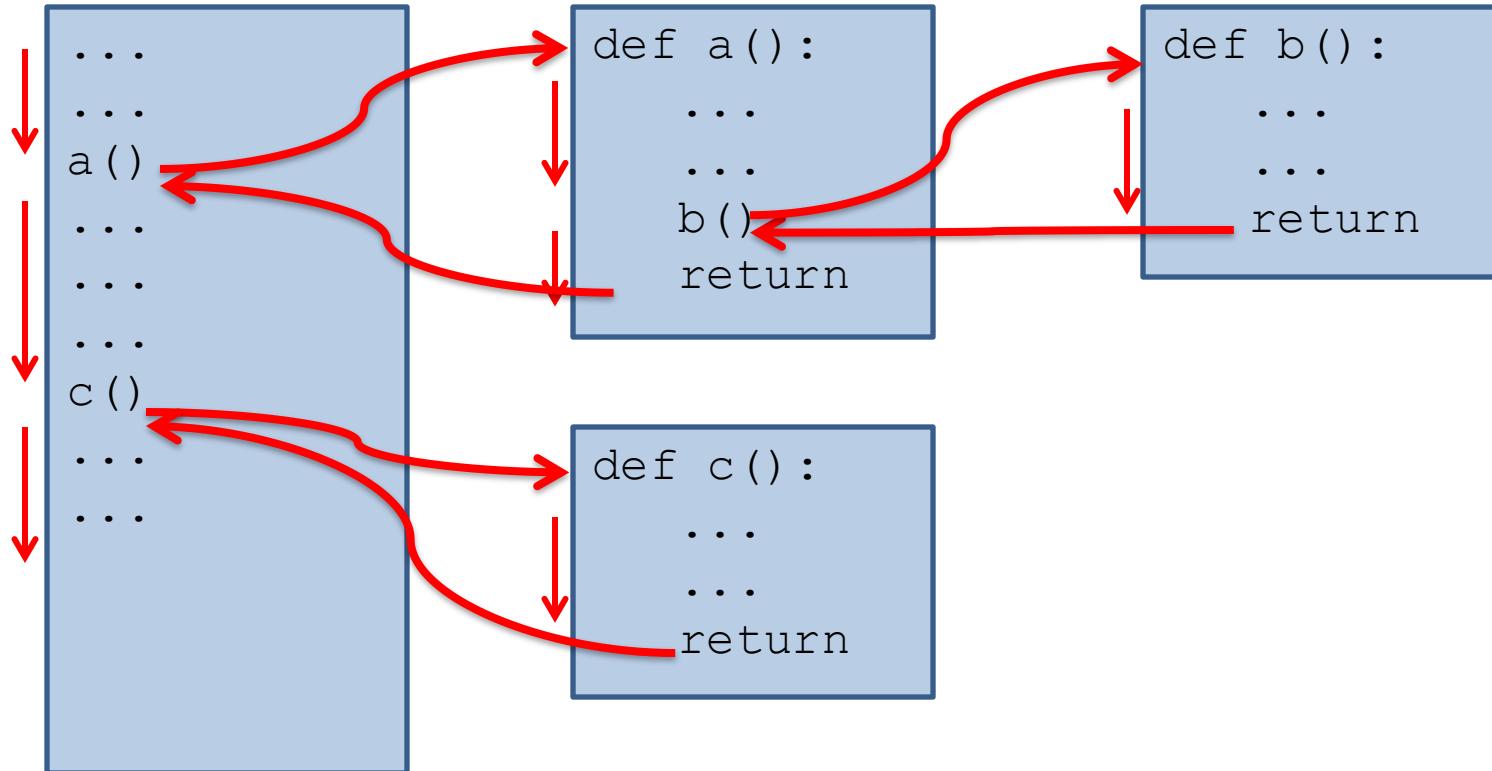
Fluxo de execução



Fluxo de execução



Fluxo de execução



Fluxo de execução

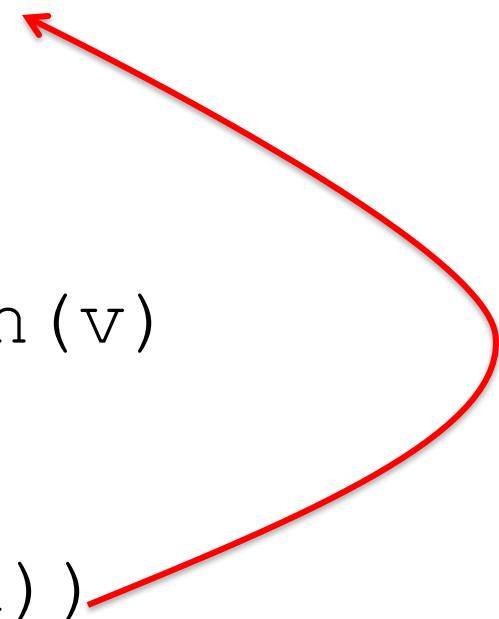
```
def calcula_media(v):  
    soma = 0  
    for e in v:  
        soma += e  
    media = soma/len(v)  
    return media
```

```
a = [1, 2, 3, 4, 5]  
print(calcula_media(a))  
b = [10, 20, 30, 40]  
print(calcula_media(b))
```

Execução começa no primeiro comando que está **fora de uma função**

Fluxo de execução

```
def calcula_media(v):  
    soma = 0  
    for e in v:  
        soma += e  
    media = soma/len(v)  
    return media  
  
a = [1, 2, 3, 4, 5]  
print(calcula_media(a))  
b = [10, 20, 30, 40]  
print(calcula_media(b))
```



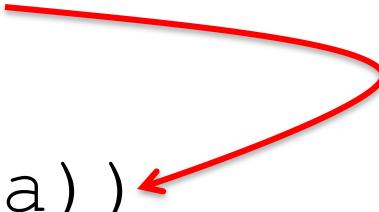
Fluxo de execução

```
def calcula_media(v):
    soma = 0
    for e in v:
        soma += e
    media = soma/len(v)
    return media

a = [1, 2, 3, 4, 5]
print(calcula_media(a))
b = [10, 20, 30, 40]
print(calcula_media(b))
```

Fluxo de execução

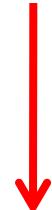
```
def calcula_media(v):
    soma = 0
    for e in v:
        soma += e
    media = soma/len(v)
    return media
a = [1, 2, 3, 4, 5]
print(calcula_media(a))
b = [10, 20, 30, 40]
print(calcula_media(b))
```



Fluxo de execução

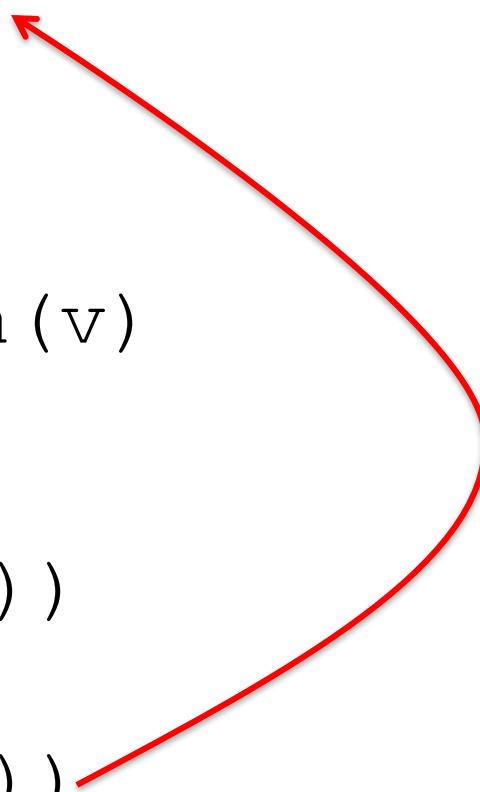
```
def calcula_media(v):
    soma = 0
    for e in v:
        soma += e
    media = soma/len(v)
    return media

a = [1, 2, 3, 4, 5]
print(calcula_media(a))
b = [10, 20, 30, 40]
print(calcula_media(b))
```



Fluxo de execução

```
def calcula_media(v):  
    soma = 0  
    for e in v:  
        soma += e  
    media = soma/len(v)  
    return media  
  
a = [1, 2, 3, 4, 5]  
print(calcula_media(a))  
b = [10, 20, 30, 40]  
print(calcula_media(b))
```



Fluxo de execução

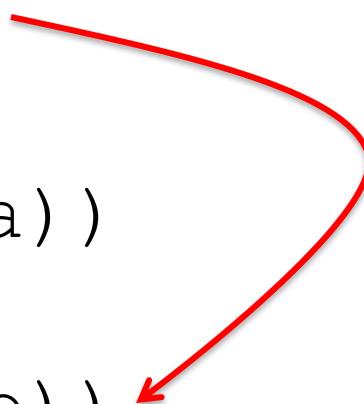
```
def calcula_media(v):
    soma = 0
    for e in v:
        soma += e
    media = soma/len(v)
    return media

a = [1, 2, 3, 4, 5]
print(calcula_media(a))
b = [10, 20, 30, 40]
print(calcula_media(b))
```

Fluxo de execução

```
def calcula_media(v):
    soma = 0
    for e in v:
        soma += e
    media = soma/len(v)
    return media

a = [1, 2, 3, 4, 5]
print(calcula_media(a))
b = [10, 20, 30, 40]
print(calcula_media(b))
```



Declaração de função

```
def nome_funcao(<parâmetros>):
    <comandos>
    [return <expressão>]
```

Exemplo:

```
def calcula_media(v):
    soma = 0
    for e in v:
        soma += e
    media = soma/len(v)
    return media
```

Exemplo

```
def calcula_tempo(velocidade, distancia):  
    tempo = distancia/velocidade  
    return tempo  
  
def calcula_distancia(velocidade, tempo):  
    distancia = velocidade * tempo  
    return distancia  
  
t = calcula_tempo(10, 5)  
print(t) → 0.5  
d = calcula_distancia(5, 4)  
print(d) → 20
```

Importante lembrar

- Um programa Python pode ter **zero ou mais** definições de função
- Uma função pode ter **zero ou mais** parâmetros
- Uma função pode ser chamada **zero ou mais** vezes
- Uma função só é **executada** quando é **chamada**
- Duas chamadas de uma mesma função podem produzir **resultados diferentes**

Importante lembrar

- Uma função que **retorna** um valor deve usar `return`
 - Assim que o comando `return` é executado, a função termina
- Uma função pode **não retornar** nenhum valor
 - Nesse caso a função termina quando sua última linha de código for executada

Escopo de variáveis

- Variáveis podem ser locais ou globais
- **Variáveis locais**
 - Declaradas dentro de uma função
 - São visíveis somente dentro da função onde foram declaradas
 - São destruídas ao término da execução da função
- **Variáveis globais**
 - Declaradas fora de todas as funções
 - São visíveis por TODAS as funções do programa

Exemplo: variáveis locais

```
def calcula_tempo(velocidade, distancia):  
    tempo = distancia/velocidade  
    return tempo  
  
def calcula_distancia(velocidade, tempo):  
    distancia = velocidade * tempo  
    return distancia  
  
t = calcula_tempo(10, 5)  
print(t)  
d = calcula_distancia(5, 4)  
print(d)
```

Exemplo: parâmetros também se comportam como variáveis locais

```
def calcula_tempo(velocidade, distancia):  
    tempo = distancia / velocidade  
    return tempo  
  
def calcula_distancia(velocidade, tempo):  
    distancia = velocidade * tempo  
    return distancia  
  
t = calcula_tempo(10, 5)  
print(t)  
d = calcula_distancia(5, 4)  
print(d)
```

Exemplo: variáveis globais

```
def calcula_tempo(velocidade, distancia):  
    tempo = distancia/velocidade  
    return tempo  
  
def calcula_distancia(velocidade, tempo):  
    distancia = velocidade * tempo  
    return distancia  
  
t = calcula_tempo(10, 5)  
print(t)  
d = calcula_distancia(5, 4)  
print(d)
```

Uso de variáveis globais x variáveis locais

- Cuidado com o uso de variáveis globais dentro das funções
 - Dificultam o entendimento do programa
 - Dificultam a correção de defeitos no programa
 - Se a variável pode ser usada por qualquer função do programa, encontrar um defeito envolvendo o valor desta variável pode ser muito complexo
- Recomendação
 - Sempre que possível, usar variáveis **locais** e passar os valores necessários para a função como parâmetro

Uso de variáveis globais

- Variáveis globais podem ser acessadas dentro de uma função
- Se for necessário altera-las, é necessário declarar essa intenção escrevendo, no início da função, o comando **global <nome da variável>**

Exemplo: variáveis globais acessadas na função

```
def maior():
    if a > b:
        return a
    else:
        return b

a = 1
b = 2
m = maior()
print(m)
```

Péssima prática
de programação!

Exemplo: variável global modificada na função

```
def maior():
    global m
    if a > b:
        m = a
    else:
        m = b

m = 0
a = 1
b = 2
maior()
print(m)
```

Péssima,
péssima, péssima
prática de
programação!

Sem uso de variáveis globais: muito mais elegante!

```
def maior(a, b):  
    if a > b:  
        maior = a  
    else:  
        maior = b  
    return maior
```

```
a = 1  
b = 2  
m = maior(a, b)  
print(m)
```

Vejam que agora **a** e
b são parâmetros.

Sem uso de variáveis globais: muito mais elegante!

```
def maior(x, y):  
    if x > y:  
        maior = x  
    else:  
        maior = y  
    return maior
```

```
a = 1  
b = 2  
m = maior(a, b)  
print(m)
```

Os parâmetros
também poderiam
ter **outros nomes**

Passagem de parâmetro

- Quando uma função é chamada, é necessário fornecer um valor para cada um de seus parâmetros
- Isso por ser feito informando o valor diretamente

```
t = calcula_tempo(1, 2)
```

- Ou usando o valor de uma variável

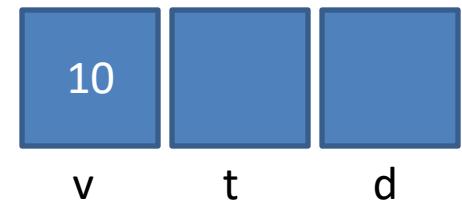
```
t = calcula_tempo(v, d)
```

Passagem de parâmetro

```
def calcula_tempo(velocidade, distancia):  
    tempo = distancia/velocidade  
return tempo
```

```
def calcula_distancia(velocidade, tempo):  
    distancia = velocidade * tempo  
return distancia
```

```
v = 10  
t = calcula_tempo(v, 5)  
print(t)  
d = calcula_distancia(v, t)  
print(d)
```



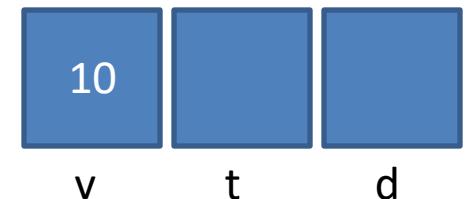
Passagem de parâmetro

```
def calcula_tempo(velocidade, distancia):
    tempo = distancia/velocidade
    return tempo
```



```
def calcula_distancia(velocidade, tempo):
    distancia = velocidade * tempo
    return distancia
```

```
v = 10
t = calcula_tempo(v, 5)
print(t)
d = calcula_distancia(v, t)
print(d)
```



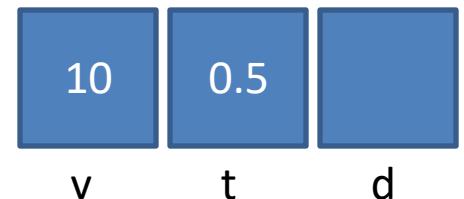
Passagem de parâmetro

```
def calcula_tempo(velocidade, distancia):
    tempo = distancia/velocidade
    return tempo
```



```
def calcula_distancia(velocidade, tempo):
    distancia = velocidade * tempo
    return distancia
```

```
v = 10
t = calcula_tempo(v, 5)
print(t)
d = calcula_distancia(v, t)
print(d)
```

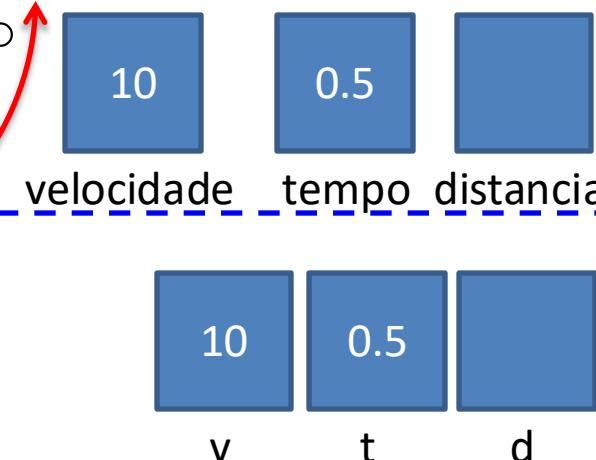


Passagem de parâmetro

```
def calcula_tempo(velocidade, distancia):
    tempo = distancia/velocidade
    return tempo
```

```
def calcula_distancia(velocidade, tempo):
    distancia = velocidade * tempo
    return distancia
```

```
v = 10
t = calcula_tempo(v, 5)
print(t)
d = calcula_distancia(v, t)
print(d)
```

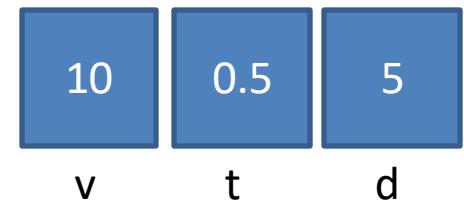


Passagem de parâmetro

```
def calcula_tempo(velocidade, distancia):
    tempo = distancia/velocidade
    return tempo
```

```
def calcula_distancia(velocidade, tempo):
    distancia = velocidade * tempo
    return distancia
```

```
v = 10
t = calcula_tempo(v, 5)
print(t)
d = calcula_distancia(v, t)
print(d)
```



Tipos de passagem de parâmetro

- **Por valor:** o valor da variável na chamada é copiado para a variável da função
 - Alterações não são refletidas na variável original

- **Por referência:** o endereço de memória é copiado para a variável da função
 - Alterações são refletidas na variável original

Passagem de parâmetro por valor

- Python usa **passagem de parâmetro por valor**
 - Faz **cópia do valor** da variável original para o parâmetro da função
 - **Variável original fica preservada** das alterações feitas dentro da função

Passagem de parâmetro por valor

- Como as variáveis do tipo **lista** guardam na verdade um endereço de memória (reveja o final da aula de manipulação de listas), o efeito é diferente
 - Se atribuir uma nova lista à variável, a **atribuição não será notada fora da função**
 - Se alterar elementos da lista, as **alterações serão notadas fora da função**

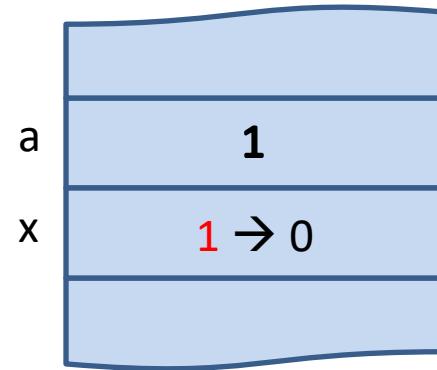
Exemplo

Variável primitiva

```
def zera(x):  
    x = 0
```

```
a = 1  
zera(a)  
print(a) → 1
```

- Na Memória

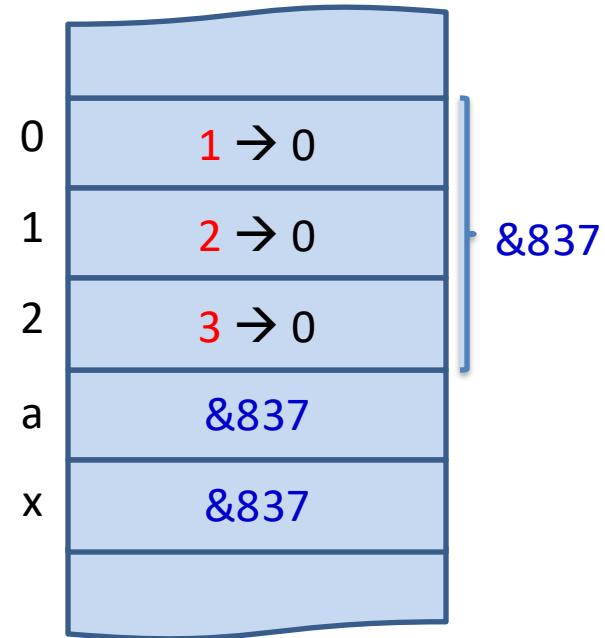


Exemplo

Alteração de lista

```
def zera(x):  
    for i in range(len(x)):  
        x[i] = 0  
  
a = [1, 2, 3]  
zera(a)  
print(a) → [0, 0, 0]
```

■ Na Memória

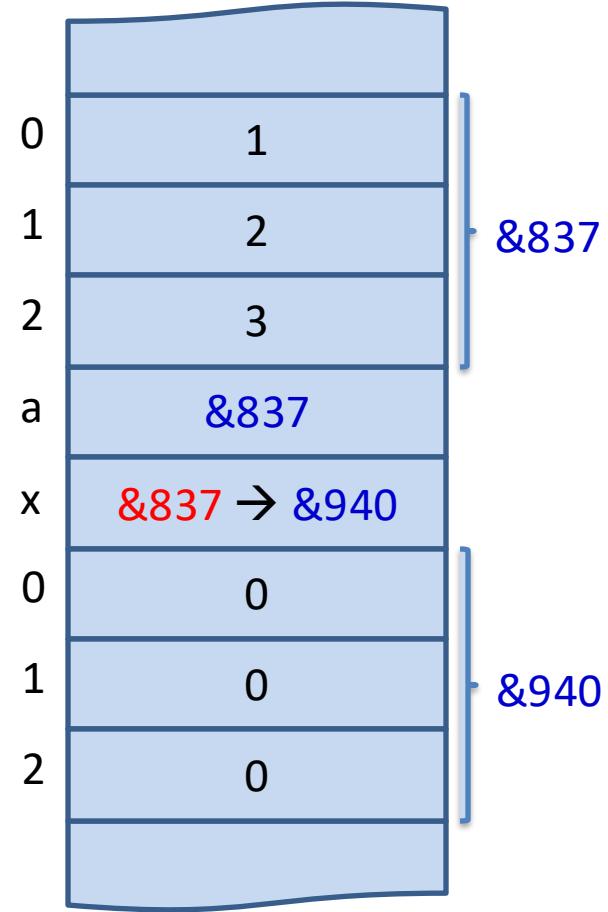


Exemplo

Atribuição em lista

```
def zera(x):  
    x = [0] * len(x)  
  
a = [1, 2, 3]  
zera(a)  
print(a) → [1, 2, 3]
```

■ Na Memória



Exemplo de função sem retorno

```
def imprime_linha(tamanho):  
    print('-' * tamanho)
```

```
texto = 'PROGRAMAR É LEGAL'  
imprime_linha(len(texto))  
print(texto)  
imprime_linha(len(texto))
```

Chamada de função

- Se a função retorna um valor, pode-se atribuir seu resultado a uma variável

```
m = maior(v)
```

- Se a função não retorna um valor (não tem **return**), não se deve atribuir seu resultado a uma variável (se for feito, variável ficará com valor **None**)

```
imprime_asterisco(3)
```

Função sem parâmetro

- Nem toda função precisa ter parâmetro
- Nesse caso, ao definir a função, deve-se abrir e fechar parênteses, sem informar nenhum parâmetro
- O mesmo deve acontecer na chamada da função

Exemplo

```
def menu():
    print('*****')
    print('1 - Somar')
    print('2 - Subtrair')
    print('3 - Multiplicar')
    print('4 - Dividir')
    print('*****')

menu()
opcao = int(input('Digite a opção desejada: '))
```

Parâmetros *default*

- Em alguns casos, pode-se definir um valor padrão (*default*) para um parâmetro. Caso ele não seja passado na chamada, o valor *default* será assumido.
- Exemplo: uma função para calcular a gorjeta de uma conta tem como parâmetros o **valor da conta** e o **percentual da gorjeta**. No entanto, na grande maioria dos restaurantes, a gorjeta é sempre 10%. Podemos então colocar 10% como valor default para o parâmetro “percentual”

Exemplo da gorjeta

```
def calcular_gorjeta(valor, percentual=10):  
    return valor * percentual/100
```

```
gorjeta = calcular_gorjeta(400)
```

```
print('O valor da gorjeta de 10% de uma conta de R$ 400  
é', gorjeta)
```

```
gorjeta = calcular_gorjeta(400, 5)
```

```
print('O valor da gorjeta de 5% de uma conta de R$ 400  
é', gorjeta)
```

Quando a **gorjeta** não é informada na chamada da função, o valor do parâmetro **gorjeta** fica sendo 10

Colocar funções em arquivo separado

- Em alguns casos, pode ser necessário colocar todas as funções em um módulo separado (ex: `funcoes`)
- Nesse caso, basta definir todas as funções num arquivo `.py` (ex.: `funcoes.py`).
- Quando precisar usar as funções em um determinado programa, basta fazer **import <nome do módulo que contém as funções>**
- Ao chamar a função, colocar o nome do módulo na frente

Exemplo

Arquivo util.py

```
def soma(v):  
    soma = 0  
    for e in v:  
        soma += e  
    return soma  
  
def media(v):  
    return soma(v)/len(v)
```

Arquivo teste.py

```
import util  
  
v = [1, 3, 5, 7, 9]  
print(util.soma(v))  
print(util.media(v))
```

OU

```
from util import soma, media  
  
v = [1, 3, 5, 7, 9]  
print(soma(v))  
print(media(v))
```

Exercícios

1. O professor deseja dividir uma turma com N alunos em dois grupos: um com M alunos e outro com (N-M) alunos. Faça o programa que lê o valor de N e M e informa o número de combinações possíveis

- Número de combinações é igual a $N!/(M! * (N-M)!)$
- Use funções para evitar repetição de código

2. Faça uma função que informe o status do aluno a partir da sua média de acordo com a tabela a seguir:

- Nota acima de 6 → “Aprovado”
- Nota entre 4 e 6 → “Verificação Suplementar”
- Nota abaixo de 4 → “Reprovado”

Exercícios

3. Faça uma calculadora que forneça as seguintes opções para o usuário, usando funções sempre que necessário. Cada opção deve usar como operando um número lido do teclado e o valor atual da memória. Por exemplo, se o estado atual da memória é 5, e o usuário escolhe somar, ele deve informar um novo número (por exemplo, 3). Após a conclusão da soma, o novo estado da memória passa a ser 8.

Estado da memória: 0

Opções:

- (1) Somar
- (2) Subtrair
- (3) Multiplicar
- (4) Dividir
- (5) Limpar memória
- (6) Sair do programa

Qual opção você deseja?

Exercícios

4. Refaça o programa anterior para adicionar uma opção para escrever um número por extenso

- Aceite números de até 9 dígitos
- Use vetores para armazenar as traduções
- Use funções para evitar código redundante

Exercícios

5. Faça um programa que, dado uma figura geométrica que pode ser uma circunferência, triângulo ou retângulo, calcule a área e o perímetro da figura

- O programa deve primeiro perguntar qual o tipo da figura:
 - (1) circunferência
 - (2) triângulo
 - (3) retângulo
- Dependendo do tipo de figura, ler o (1) tamanho do raio da circunferência; (2) tamanho de cada um dos lados do triângulo; (3) tamanho dos dois lados retângulo
- Usar funções sempre que possível

Referências

- Slides feitos em conjunto com Aline Paes e Vanessa Braganholo

Subprogramação

