Atribuições e funções com listas

Aula 11

Diego Padilha Rubert

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Algoritmos e Programação

Conteúdo da aula

Introdução

- até agora vimos apenas atribuições e funções com variáveis contendo valores simples (não compostos)
- o comportamento de atribuições e parâmetros de funções é diferente quando estamos trabalhando com listas

Introdução

- até agora vimos apenas atribuições e funções com variáveis contendo valores simples (não compostos)
- o comportamento de atribuições e parâmetros de funções é diferente quando estamos trabalhando com listas

Atribuições com valores simples

- Observe o trecho de código abaixo:

```
x = 2
y = x
y += 1
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena o valor 2
- Na segunda instrução, criamos uma variável `y` que recebe uma cópia do valor armazenado no momento em `x`, ou seja, 2
- Na terceira instrução, aumentamos em 1 o valor armazenado na variável `y`

Atribuições com valores simples

- Observe o trecho de código abaixo:

```
x = 2
y = x
y += 1
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena o valor 2
- Na segunda instrução, criamos uma variável `y` que recebe uma cópia do valor armazenado no momento em `x`, ou seja, 2
- Na terceira instrução, aumentamos em 1 o valor armazenado na variável `y`

Atribuições com valores simples

- Observe o trecho de código abaixo:

```
x = 2
y = x
y += 1
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena o valor 2
- Na segunda instrução, criamos uma variável `y` que recebe uma cópia do valor armazenado no momento em `x`, ou seja, 2
- Na terceira instrução, aumentamos em 1 o valor armazenado na variável `y`

Atribuições com valores simples

- Observe o trecho de código abaixo:

```
x = 2
y = x
y += 1
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena o valor 2
- Na segunda instrução, criamos uma variável `y` que recebe uma cópia do valor armazenado no momento em `x`, ou seja, 2
- Na terceira instrução, aumentamos em 1 o valor armazenado na variável `y`

Atribuições com listas

- Observe agora um trecho de código similar ao anterior, onde as variáveis agora armazenam listas e não valores simples:

```
x = [6, 2, 7]
y = x
y.append(100)
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena a lista `[6, 2, 7]`
- Na segunda instrução, criamos uma variável `y` que recebe `x`, contudo, quando trabalhamos com listas, atribuições fazem com que as variáveis envolvidas passem a **referenciar** a mesma lista ao invés de criar uma cópia, como ocorre com valores simples
- Na terceira instrução, adicionamos ao final da lista referenciada por `y` o valor 100, contudo, `y` é apenas uma segunda referência para a lista também referenciada por `x`

Atribuições com listas

- Observe agora um trecho de código similar ao anterior, onde as variáveis agora armazenam listas e não valores simples:

```
x = [6, 2, 7]
y = x
y.append(100)
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena a lista `[6, 2, 7]`
- Na segunda instrução, criamos uma variável `y` que recebe `x`, contudo, quando trabalhamos com listas, atribuições fazem com que as variáveis envolvidas passem a **referenciar** a mesma lista ao invés de criar uma cópia, como ocorre com valores simples
- Na terceira instrução, adicionamos ao final da lista referenciada por `y` o valor 100, contudo, `y` é apenas uma segunda referência para a lista também referenciada por `x`

Atribuições com listas

- Observe agora um trecho de código similar ao anterior, onde as variáveis agora armazenam listas e não valores simples:

```
x = [6, 2, 7]
y = x
y.append(100)
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena a lista `[6, 2, 7]`
- Na segunda instrução, criamos uma variável `y` que recebe `x`, contudo, quando trabalhamos com listas, atribuições fazem com que as variáveis envolvidas passem a **referenciar** a mesma lista ao invés de criar uma cópia, como ocorre com valores simples
- Na terceira instrução, adicionamos ao final da lista referenciada por `y` o valor 100, contudo, `y` é apenas uma segunda referência para a lista também referenciada por `x`

Atribuições com listas

- Observe agora um trecho de código similar ao anterior, onde as variáveis agora armazenam listas e não valores simples:

```
x = [6, 2, 7]
y = x
y.append(100)
print (x, y)
```

- Na primeira instrução, criamos uma variável `x` que armazena a lista `[6, 2, 7]`
- Na segunda instrução, criamos uma variável `y` que recebe `x`, contudo, quando trabalhamos com listas, atribuições fazem com que as variáveis envolvidas passem a **referenciar** a mesma lista ao invés de criar uma cópia, como ocorre com valores simples
- Na terceira instrução, adicionamos ao final da lista referenciada por `y` o valor 100, contudo, `y` é apenas uma segunda referência para a lista também referenciada por `x`

Listas como parâmetros de funções

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def remove_minimo(lista):
    min = lista[0]
    for v in lista:
        if v < min:
            min = v
    lista.remove(min)

# Lê vários inteiros em uma única linha e os armazena em uma lista
l = list(map(int,input().split()))
remove_minimo(l)
print(l)
exit(0)
```

A função recebe e trabalha com a lista original!

Listas como parâmetros de funções

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def remove_minimo(lista):
    min = lista[0]
    for v in lista:
        if v < min:
            min = v
    lista.remove(min)

# Lê vários inteiros em uma única linha e os armazena em uma lista
l = list(map(int,input().split()))
remove_minimo(l)
print(l)
exit(0)
```

A função recebe e trabalha com a lista original!

Listas como parâmetros de funções

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def dobro(lista):
    for i in range(len(lista)):
        lista[i] *= 2

# Lê vários inteiros em uma única linha e os armazena em uma lista
l = [ int(v) for v in input().split() ]
dobro(l)
print(l)
exit(0)
```

Exercícios

1. (a) Escreva uma função com a seguinte interface:

```
def subconjunto(A, B):
```

que receba duas listas A e B, ambas representando conjuntos, devolvendo `True` se A está contido em B ($A \subset B$) e `False` caso contrário.

(b) Escreva um programa que leia dois conjuntos com números inteiros e diga se os conjuntos são iguais (obs: os elementos podem estar repetidos e em ordens diferentes). Use a função do item (a).

Exercícios

2. (a) Escreva uma função com a seguinte interface:

```
def remrep(lista):
```

que receba uma lista e remova elementos repetidos.

(b) Escreva um programa que leia um conjunto de números inteiros e escreva o conjunto lido sem repetições de valores. Use a função do item (a).

Exercícios

3. (a) Escreva uma função com a seguinte interface:

```
def intersec(A, B):
```

que dadas duas listas com números inteiros A e B que representam conjuntos, construa e devolva uma nova lista contendo a intersecção dos conjuntos formados pelos elementos de A e B ($A \cap B$).

(b) Escreva um programa que leia dois conjuntos com números inteiros e escreva a intersecção dos dois conjuntos. Use a função do item (a).

Exercícios

4. Uma sequência de $n > 0$ números inteiros é chamada de **cheia** se os valores absolutos das diferenças entre os elementos consecutivos representam todos os possíveis valores entre 1 e $n - 1$.
   Exemplo: $1\ \ 4\ \ 2\ \ 3$ é uma sequência cheia, porque os valores absolutos das diferenças entre seus elementos consecutivos são 3, 2 e 1, respectivamente.
   Observe que esta definição implica que qualquer sequência contendo exatamente um número inteiro é uma sequência cheia.

Exercícios

4. (*continuação*)
   (a) Escreva uma função com a seguinte interface:

```
def uns(contadores):
```

   que receba uma lista de números inteiros chamada *contadores*, devolvendo `True` se todos elementos da lista são iguais a 1 ou `False` caso contrário.
   (b) Escreva um programa que leia uma sequência de números inteiros e diga se a sequência é cheia ou não. Use a função do item (a).

Exercícios

5. O **crivo de Eratóstenes** é um método para encontrar números primos até um certo valor limite. Segundo a tradição, foi criado pelo matemático grego Eratóstenes, o terceiro bibliotecário-chefe da Biblioteca de Alexandria. A ideia do método é começar com todos os inteiros no intervalo $[2, n]$ e eliminar, em cada iteração, os múltiplos próprios de um número primo considerado. O primeiro número primo a ser considerado é 2 e o último é $\lfloor \sqrt{n} \rfloor$. Dados dois números inteiros a e b, dizemos que a é **múltiplo próprio** de b se a é múltiplo de b e $a > b$.

Exercícios

5. (*continuação*)
   Exemplo:
   Se $n = 25$, consideramos a lista C com os $n - 1$ elementos a seguir

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25.$$

   Eliminando os múltiplos próprios de 2 restam

$$2,\ 3,\ 5,\ 7,\ 9,\ 11,\ 13,\ 15,\ 17,\ 19,\ 21,\ 23,\ 25\ .$$

   Eliminando os múltiplos próprios de 3 restam

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 25\ .$$

   Eliminando os múltiplos próprios de 5 restam

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23\ .$$

   Não há necessidade de verificar os múltiplos próprios de $7, 11, 13, 17, 19$ e 23, já que $\lfloor \sqrt{25} \rfloor = 5$.

Exercícios

5. (*continuação*)
   (a) Escreva uma função com a seguinte interface:

```
def elimina(C, i):
```

   que receba uma lista C com números inteiros e um índice i, e então elimine desta lista os múltiplos próprios do valor em $C[i]$ a partir da posição seguinte a esse valor, ou seja, a partir da posição $i + 1$ de C. Lembre-se que para remover o elemento na posição j de uma lista C podemos utilizar `del C[j]`.
   (b) Escreva um programa que receba um número inteiro n, com $2 \leqslant n \leqslant 1000$, e imprima os números primos de 2 a n. Utilize a função do item (a).