MC-102 — Aula 5 Funções e listas I

Instituto de Computação – Unicamp

15 de Agosto de 2019

Funções

- Um ponto chave na resolução de um problema complexo é conseguir "quebrá-lo"em subproblemas menores.
- Ao criarmos um programa para resolver um problema, é crítico quebrar um código grande em partes menores, fáceis de serem entendidas e administradas.
- Isto é conhecido como modularizacão, e é empregado em qualquer projeto de engenharia envolvendo a construção de um sistema complexo.

Funções

- Funções são estruturas que agrupam um conjunto de comandos, que são executados quando a função é chamada/invocada.
- As funções normalmente retornam um valor ao final de sua execução.

Exemplo de função:

```
a = input()
```

Porque utilizar funções?

- Evitar que os blocos do programa fiquem grandes demais e, por consequência, mais difíceis de ler e entender.
- Separar o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Mais fácil para testar de forma iterativa
- Permitir o reaproveitamento de código já construído (por você ou por outros programadores).
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa, minimizando erros e facilitando alterações.

Definindo uma função

Uma função é definida da seguinte forma:

```
def nome(parâmetro1, ..., parâmetroN):
      comandos...
      return valor de retorno
```

- Os **parâmetros** são variáveis, que são inicializadas com valores indicados durante a invocação da função.
- O comando **return** devolve para o invocador da função o resultado da execução desta.
- os comandos dentro da função estão indentados

Definindo uma função: Exemplo 1

A função abaixo recebe como parâmetro dois valores inteiros. A função faz a soma destes valores, e devolve o resultado.

```
def soma(a, b):
    c = a + b
    return c
```

- Quando o comando **return** é executado, a função para de executar e retorna o valor indicado para quem fez a invocação (ou chamada) da função.

Definindo uma função: Exemplo 1

```
def soma (a, b):
    c = a + b
    return c
```

- Qualquer função pode invocar esta função, passando como parâmetro dois valores, que serão atribuídos para as variáveis **a** e **b** respectivamente.

```
r = soma(12, 90)

velocidade = 90
z = 27
x = soma(velocidade, z+4)
```

Definindo uma função: Exemplo 2

- A lista de parâmetros de uma função pode ser vazia.

```
def leNumeroInt():
    c = input("Digite um número inteiro: ")
    return int(c)
```

- O retorno será usado pelo invocador da função:

```
r = leNumeroInt()
print("Número digitado: ", r)
```

Definindo uma função: Exemplo 2

Programa completo:

```
def leNumeroInt():
    c = input("Digite um número inteiro: ")
    return int(c)


r = leNumeroInt()
print("Número digitado: ", r)
```

Definindo uma função: Exemplo 2

Atenção: a definição da função tem que vir antes do seu uso!

```
r = leNumeroInt()
print("Número digitado: ", r)

def leNumeroInt():
    c = input("Digite um número inteiro: ")
    return int(c)
```

```
% python a.py
Traceback (most recent call last):
  File "a.py", line 2, in <module>
    r = leNumeroInt()
NameError: name 'leNumeroInt' is not defined
```

Exemplo de função 3

```
def soma(a, b):
    c = a + b
    return c

x1 = 4
x2 = -10
res = soma(5, 6)
print("Primeira soma: ",res)
res = soma(x1, x2)
print("Segunda soma: ",res)
```

- Qualquer programa começa executando os comandos fora de qualquer função na ordem de sua ocorrência.
- Quando se encontra a chamada para uma função, o fluxo de execução passa para ela e se executa os comandos até que um **return** seja encontrado ou o fim da função seja alcançado.
- Depois disso o fluxo de execução volta para o ponto onde a chamada da função ocorreu.

Exemplo de função 4

- A expressão contida dentro do comando **return** é chamado de valor de retorno (é a resposta da função). Nada após ele será executado.

```
def soma(a, b):
   c = a + b
   return c

def leNumero():
   c = int(input("Digite um número: "))
   return c
   print("Bla bla bla!\n")


x1 = leNumero()
x2 = leNumero()
res = soma(x1, x2)
print("Soma é: ", res)
```

- Não será impresso *Bla bla bla!*

Exemplo de função

```
def somaEsquisita(x, y) :
  x = x + 1
  y = y + 1
  return x + y


a=10
b=5
print ("Soma de a e b:", a + b)
print ("Soma de x e y:", somaEsquisita(a, b))
print ("a:", a)
print ("b:", b)
```

Os valores de **a** e **b** não são alterados por operações feitas em **x** e **y**

Definindo funções depois do seu uso

Exemplo:

```
def main():
  x1 = leNumero()
  x2 = leNumero()
  res = soma(x1, x2)
  print("Soma é: ", res)


def soma(a, b):
  c = a + b
  return c

def leNumero():
  c = int(input("Digite um número: "))
  return c

main()
```

Agora a execução do programa ocorre sem problemas.

Programando em modo batch e testando no modo interativo

Uma forma interessante de trabalhar com o python e escrever um arquivo com as suas funções (inclusive a **main()**) e testar as funções em modo interativo.
Quando voce acha um erro, voce edita o arquivo com as funções, volta a carrega-lo e testa de novo.

- **python -i prog.py** carrega o **prog.py** e vai para o modo iterativo
- dento do modo interativo use **exec(open('prog.py').read())**
- use uma IDE como PyCharm, Spyder e outros que permite escrever o arquivo e mandar partes dele (funções) para o modo interativo

Exercício

- Escreva uma função que computa a potência a^b para valores a e b (assuma um inteiro) passados como parâmetros (não use o operador **).

Declaração de uma lista

Declara-se uma lista, colocando entre colchetes uma sequência de dados separados por vírgula:

$$\text{identificador} = [\text{dado}_1, \text{dado}_2, \ldots, \text{dado}_n]$$

Exemplo de listas

Exemplo de listas:

- Uma lista de inteiros.

```
x = [2, 45, 12, 9.78, -2]
```

- Listas podem conter dados de tipos diferentes.

```
x = [2, "qwerty" ,45.99087, 0, "a"]
```

- Listas podem conter outras listas.

```
x = [2, [4,5], [9]]
```

- Ou podem não conter nada. Neste caso **[]** indica a lista vazia.

```
x = []
```

Usando uma lista

- Pode-se acessar uma determinada posição da lista utilizando-se um índice de valor inteiro.
- Sendo n o tamanho da lista, os índices válidos para ela vão de 0 até $n - 1$.
  - A primeira posição da lista tem índice 0.
  - A última posição da lista tem índice $n - 1$.
- A sintaxe para acesso de uma determinada posição é:
  - identificador[**posição**];

Um elemento de uma lista em uma posição específica tem o mesmo comportamento que uma variável simples.

```
>>> notas= [4.5, 8.6, 9, 7.8, 7]
>>> notas[1]+2
10.6
>>> notas[3]=0.4
>>> notas
[4.5, 8.6, 9, 0.4, 7]
```

Listas - índices

- Índices negativos se referem a lista da direita para a esquerda:

```
>>> notas= [4.5, 8.6, 9, 7.8, 7]
>>> notas[-1]
7
```

- Ocorre um erro se tentarmos acessar uma posição da lista que não existe.

```
>>> notas[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>>> notas[-6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Listas - funções em listas

- A função **len(lista)** retorna o número de itens na lista.

```
>>> x=[16,5,4,4,7,9]
>>> len(x)
6
```

Listas - **append**

- Uma operação importante é acrescentar um item no final de uma lista. Isto é feito pela função **append**.

```
lista.append(item)
```

```
>>> x=[6,5]
>>> x.append(98)
>>> x
[6, 5, 98]
```

- Note o formato diferente da função **append**. A lista que será modificada aparece antes, seguida de um ponto, seguida do **append** com o item a ser incluído como argumento. Formalmente, este tipo de função é chamada de `método`.

Listas - funções em listas

- A operação de soma em listas gera uma nova lista que é o resultado de "grudar" **lista2** ao final da **lista1**. Isto é conhecido como **concatenação** de listas.

lista1 + lista2

```
>>> lista1=[1,2,4]
>>> lista2=[27,28,29,30,33]
>>> x=lista1+lista2
>>> x
[1, 2, 4, 27, 28, 29, 30, 33]
>>> lista1
[1, 2, 4]
>>> lista2
[27, 28, 29, 30, 33]
```

- Veja que a operação de concatenação não modifica as listas originais.

Exemplo: Produto Interno de dois vetores

- Uma função para computar o produto interno (produto escalar) de dois vetores (listas) de tamanho 3

```
#calculando o produto interno
def prodint(l1,l2):
   resultado = l1[0]*l2[0]+l1[1]*l2[1]+l1[2]*l2[2]
   return resultado
```

Listas - próximos passos

- O exemplo acima de uso de lista é muito simples. So funciona para listas de 3 elementos. O objetivo de apresentar listas agora é exatamente motivar o uso de comandos repetitivos, que virá na próxima aula
- Há muito mais funções e métodos que operam e modificam listas - veremos eles mais adiante, após os comandos repetitivos

String - primeira visão

- Stings são (quase) uma lista de caracteres.
- Acesso ([]), **len** e concatenação (++) funcionam em strings
- **append** não funciona. Strings nao podem ser modificados.

```
>>> x = 'asdfg'
>>> x[2]
'd'
>>> x + ' / '+ 'qwerty'
'asdfg / qwerty'
>>> x.append('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```