MC-102 — Aula 11
Objetos Mutáveis e Imutáveis
Funções I

Prof. Luiz F. Bittencourt

Turmas QR

Instituto de Computação – Unicamp

2019

Conteúdo adaptado de slides fornecidos pelo Prof. Eduardo Xavier.

Roteiro

Objetos mutáveis e imutáveis

- Cada objeto criado em Python (um **int**, **float**, **list**, etc) é classificado como mutável ou imutável.
- Os objetos do tipo **int**, **float**, **string**, e **bool** são imutáveis. Isso significa que objetos desse tipo não podem ter seus valores alterados.
- Cada objeto criado está em uma posição de memória e possui um identificador único que pode ser obtido com a função **id()**

```
>>> a = 94
>>> id(a)
4297373664
>>> id(94)
4297373664
>>>
```

- A variável **a** está associada com o objeto **int** de valor 94, que possui o identificador 4297373664.

Objetos mutáveis e imutáveis

- Como um **int** é imutável, quando fazemos o incremento da variável **a**, o que ocorre na verdade é a criação de um novo objeto do tipo **int** que será associado com **a**.

```
>>> a = 94
>>> id(a)
4297373664
>>> id(94)
4297373664
>>> a = a + 1
>>> id(a)
4297373696
>>> id(95)
4297373696
>>>
```

Objetos mutáveis e imutáveis

- Objetos do tipo **list** são mutáveis (veremos outros tipos mutáveis posteriormente no curso). Isso significa que objetos desse tipo podem ter seus valores alterados.

```
>>> a=[]
>>> id(a)
4328743752
>>> a.append(1)
>>> a
[1]
>>> id(a)
4328743752
>>> a += [2]
>>> a
[1, 2]
>>> id(a)
4328743752
```

- No exemplo acima a lista cujo **id** é 4328743752, é alterada inicialmente de **[]** para **[1,2]**.

Objetos mutáveis e imutáveis

```
>>> a=[]
>>> id(a)
4328743752
>>> a.append(1)
>>> a
[1]
>>> id(a)
4328743752
>>> a += [2]
>>> a
[1, 2]
>>> id(a)
4328743752
>>> a = [1,2]   #última atribuição
>>> id(a)
4328777800
```

- Note que a variável **a** fica associada com a mesma lista de identificador 4328743752, exceto na última atribuição.
- Na última atribuição é criada uma nova lista e esta é associada com **a**.

Objetos mutáveis e imutáveis

- Objetos mutáveis e imutáveis possuem comportamentos distintos quando usados em funções como veremos adiante.

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, sendo 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 podem retornar 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 conseqüência, mais difíceis de ler e entender.
- Separar o programa em partes que possam ser logicamente compreendidas de forma isolada.
- 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.

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)
r = soma (-9, 45)
```

Definindo uma função: Exemplo 1

Programa completo:

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

r = soma(12,90)
print("r = ", r)
r = soma(-9, 45)
print("r = ", r)
```

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)
```

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!*

Invocando uma função

- Uma forma clássica de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor a uma variável:

```
x = soma(4, 2)
```

- Na verdade, o resultado da chamada de uma função é uma expressão e pode ser usada em qualquer lugar que aceite uma expressão:

Exemplo

```
print("Soma de a e b:", soma(a, b))
```

Invocando uma função

- Na chamada da função, para cada um dos parâmetros devemos fornecer um valor que pode ser uma variável ou uma constante.
- Neste exemplo a função possui dois parâmetros e na sua invocação são passados dois valores constantes inteiros:

```
def quadradoDaSoma(a, b):
    a = (a+b)*(a+b)
    return a

r = quadradoDaSoma(2, 2)
print(r)  #imprime 16
```

- Neste outro exemplo são passados dois valores de variáveis:

```
def quadradoDaSoma(a, b):
    a = (a+b)*(a+b)
    return a

a = 2
c = 3
r = quadradoDaSoma(a, c)
print(r)  #imprime 25
```

Invocando uma função

- O parâmetro é uma variável da função que só existe durante a execução da função e é inicializada com o **identificador do objeto** correspondente na invocação da função.
  - Os valores das variáveis na invocação da função podem ser alterados ou não dentro da função dependendo se estes estão associadas com objetos **mutáveis** ou **imutáveis**.

Invocando uma função

- Considere o exemplo:

```
def quadrado(a):
    print("ID antes da multiplicação:", id(a))
    a = a*a
    print("ID depois da multiplicação:", id(a))
    return a

a = 2
print("ID original:", id(a))
r = quadrado(a)
print("ID depois da função:", id(a))
print(r)
print(a)
```

- A saída é:

```
ID original: 4297370720
ID antes da multiplicação: 4297370720
ID depois da multiplicação: 4297370784
ID depois da função: 4297370720
4
2
```

Invocando uma função

```
def quadrado(a):
    print("ID antes da multiplicação:", id(a))
    a = a*a
    print("ID depois da multiplicação:", id(a))
    return a

a = 2
print("ID original:", id(a))
r = quadrado(a)
print("ID depois da função:", id(a))
print(r)
print(a)
```

- Note que o valor da variável **a** de fora da função permanece com o valor 2, pois a variável **a** de dentro da função tem seu identificador alterado para o novo objeto de valor 4.

Invocando uma função

- Considere este outro exemplo:

```
def addTwo(b):
    print("ID antes da inserção:", id(b))
    b += [2]
    print("ID depois da inserção:", id(b))
    return b

a = [5]
print("ID original:", id(a))
r = addTwo(a)
print("ID depois da função:", id(a))
print("ID de r:", id(r))
print(a)
```

- A saída será:

```
ID original: 4320355272
ID antes da inserção: 4320355272
ID depois da inserção: 4320355272
ID depois da função: 4320355272
ID de r: 4320355272
[5, 2]
```

Invocando uma função

```
def addTwo(b):
    print("ID antes da inserção:", id(b))
    b += [2]
    print("ID depois da inserção:", id(b))
    return b

a = [5]
print("ID original:", id(a))
r = addTwo(a)
print("ID depois da função:", id(a))
print("ID de r:", id(r))
print(a)
```

- Neste outro exemplo **b** permanece com o mesmo identificador de **a**, mesmo após a inserção de um novo valor no fim da lista, pois uma lista é mutável.
- Por isso alterações feitas dentro da função em **b** são observadas depois fora da função em **a**.
- Note também que **r** possui o mesmo identificador de **a**.

Funções que não retornam nada

- Em alguns casos faz sentido para uma função não retornar nada. Em particular, funções que apenas imprimem algo normalmente não precisam retornar nada.
- Há dois modos de criar funções que não retornam nada:
  - Não use o comando **return** na função.
  - Use o **return None**.
- **None** é um valor que representa o "nada".

```
def imprime(num):
  print("Número: ", num)
```

Sem return

```
def imprimeCaixa (numero):
  tamanho=len(str(numero))
  for i in range(12+tamanho):
    print('+',end='',sep='')
  print()
  print('| Número:',numero,'|')
  for i in range(12+tamanho):
    print('+',end='',sep='')
  print()

imprimeCaixa(10)
imprimeCaixa(23456)
```

Com return None

```
def imprimeCaixa (numero):
  tamanho=len(str(numero))
  for i in range(12+tamanho):
    print('+',end='',sep='')
  print()
  print('| Número:',numero,'|')
  for i in range(12+tamanho):
    print('+',end='',sep='')
  print()
  return None

imprimeCaixa(10)
imprimeCaixa(23456)
```

Em ambos os casos, a chamada da função é um comando por si só.

Definindo funções depois do seu uso

- Até o momento, aprendemos que devemos definir as funções antes do seu uso. O que ocorreria se declarássemos depois?

```
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
```

- Ocorre um erro ao executarmos o programa!

```
Traceback (most recent call last):
  File "t2.py", line 2, in <module>
    x1 = leNumero()
NameError: name 'leNumero' is not defined
```

Definindo funções depois do seu uso

- É comum criarmos uma função **main()** que executa os comandos iniciais do programa.
- O seu programa conterá então várias funções (incluindo a **main()**) e um único comando no final do arquivo que é a chamada da função **main()**.
- O programa será organizado da seguinte forma:

```
import bibliotecas

def main():
  Comandos Iniciais

def fun1(Parâmetros):
  Comandos

def fun2(Parâmetros):
  Comandos

...
...
main()
```

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.

Exercício

- Escreva uma função que computa a potência a^b para valores a e b (assuma um inteiro) passados por parâmetro (não use o operador **).
- Use a função anterior e crie um programa que imprima todas as potências:

$$2^0, 2^1, \ldots, 2^{10}, 3^0, \ldots, 3^{10}, \ldots, 10^{10}.$$

Exercício

- Escreva uma função que computa o fatorial de um número n passado por parâmetro. OBS: Caso $n \leq 0$ seu programa deve retornar 1.
- Use a função anterior e crie um programa que imprima os valores de $n!$ para $n = 1, \ldots, 20$.

Definindo parâmetros com valor default

- Até agora, na chamada de uma função era preciso colocar tantos argumentos quantos os parâmetros definidos para a função.
- Mas é possível definir uma função onde alguns parâmetros vão ter um valor default, e se não houver na invocação o argumento correspondente, este valor default é usado como valor do parametro.

```
def fx (a,b=9):
  return a+b
```

```
>>> fx(3)
12
>>> fx(3,4)
7
```

Invocando funções com argumentos nomeados

- Os argumentos de uma função podem ser passados por nome em vez de por posição.

```
def fx2(a,b=9,c=0):
  return 100*a+10*b+c
```

```
>>> fx2(3)
390
>>> fx2(3,4,5)
345
>>> fx2(b=8,a=5,c=7)
587
```

- Usualmente parâmetros com valor default são nomeados na chamada da função (mas isso não é obrigatório - veja que o parâmetro **a** também foi chamado nomeado).

A função **print**

- A função **print** tem 2 parâmetros default, que devem ser passados nomeados: o **sep** (que é a string que separa na impressão um argumento do outro) e o **end** (o que é impresso ao final do **print**).
- O valor default para o **sep** é **' '** (um branco) e para o **end** é **'\n'**.

```
>>> print(3,4,5,end='=  ',sep=' + ')
3 + 4 + 5=  >>>
```

- note que o **print** imprimiu o + com 2 brancos como separador dos números, e no fim o sinal = sem mudar de linha. O prompt do modo interativo veio logo depois, na mesma linha.
- O **print** tem outra característica: ele pode receber um número qualquer de argumentos. Mas não veremos neste curso como fazer isso.