MC-102 — Aula 12
Funções I

Eduardo C. Xavier

Instituto de Computação – Unicamp

12 de Abril de 2018

Roteiro

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

- Um conceito muito comum de modularização em linguagens de programação são funções:

Função

Estrutura da linguagem que agrupa um conjunto de comandos que realizam uma tarefa específica. Uma função recebe argumentos como entrada, realiza uma computação e em geral devolve uma resposta para o invocador da função.

- Vocês já usaram algumas funções como **input** e **print**.
- Algumas funções podem devolver algum valor ao final de sua execução como **input** e **math.sqrt**:

```
n = float(input())
x = math.sqrt(n)
```

- Vamos aprender como criar nossas próprias funções.

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 (estes valores são os argumentos 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
```

- Pode-se invocar esta função passando como parâmetro dois valores inteiros, que serão associados com as variáveis **a** e **b** respectivamente.

```
r = soma(12, 10)
r = soma (-9, 11)
```

Exemplo de função 1

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

r = soma(12,10)
print("r = ", r)
r = soma(-9, 11)
print("r = ", r)
```

- O programa faz a definição da função **soma** e em seguida executa os demais comandos.
- Quando se encontra a chamada para uma função, o fluxo de execução passa para ela e é executado 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 1

- 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
  print("Bla bla bla!")

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

- Não será impresso *Bla bla bla!*

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)


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

Funções que retornam nada

- Faz sentido para uma função não retornar nada. Em particular, funções que apenas imprimem algo normalmente não precisam retornar nada.
- **None** é um objeto em Python que representa o "nada".
- Há dois modos de criar funções que não retornam nada:
  - Use o comando **return None**.
  - Não use o comando **return** na função. Por padrão será devolvido **None** no fim da execução da função.

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

Definindo uma função: Exemplo 3, return None

```
def imprime_com_caixa(n):
    tam = len(str(n))
    for i in range(2+tam):
        print('+',end='')
    print()
    print('|'+str(n)+'|')

    for i in range(2+tam):
        print('+',end='')
    print()


imprime_com_caixa(5)
imprime_com_caixa(12345)
```

Definindo uma função: Exemplo 3, return None

```
def imprime_com_caixa(n):
    tam = len(str(n))
    for i in range(2+tam):
        print('+',end='')
    print()
    print('|'+str(n)+'|')

    for i in range(2+tam):
        print('+',end='')
    print()


imprime_com_caixa(5)
imprime_com_caixa(12345)
a = imprime_com_caixa(4)
print(a)
```

O que será impresso neste exemplo?

Invocando uma função

- Na chamada da função, para cada um dos parâmetros desta, devemos fornecer um argumento 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 associados à variáveis:

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

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

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. Isto significa que objetos deste 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). Isto significa que objetos deste tipo podem ter seus valores alterados.

```
>>> a = []
>>> id(a)
4331278472
>>> a.append(1)
>>> b = a
>>> b[0] = 10
>>> a
[10]
>>> id(a)
4331278472
>>> id(b)
4331278472
```

- No exemplo acima a lista cujo **id** é 4331278472, é alterada inicialmente de **[]** para **[1]** permanecendo com o mesmo **id**.
- Depois esta mesma lista é alterada para **[10]** mesmo que tenhamos usado a variável **b**, pois esta variável está associada com a mesma lista de **a**.

Funções: Objetos mutáveis e imutáveis

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

Parâmetros

- O parâmetro é uma variável da função que só existe durante a execução da função e é inicializada com o **identificador do objeto** do argumento passado na invocação da função.
- Por isso a variável parâmetro fica associada com o mesmo objeto que foi passado por parâmetro.
  - Os valores dos objetos passados como parâmetro para a função podem ser alterados ou não dentro da função dependendo se estes são objetos **mutáveis** ou **imutáveis**.

Funções: Objetos mutáveis e imutáveis

- 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
```

Funções: Objetos mutáveis e imutáveis

```
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.

Funções: Objetos mutáveis e imutáveis

- Considere este outro exemplo:

```
def addTwo(b):
    print("ID antes da inserção:", id(b))
    b.append(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]
```

Funções: Objetos mutáveis e imutáveis

```
def addTwo(b):
    print("ID antes da inserção:", id(b))
    b.append(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**.

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 um 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.

Funções Podem Invocar Funções

- Qualquer função pode invocar outra função.
- Veja o exemplo no próximo slide.

Funções Podem Invocar Funções

- Note que **fun1** invoca **fun2**, e isto é perfeitamente legal.
- O que será impresso?

```
def main():
  c = 5
  c = fun1(c)
  print("c =", c)

def fun1(a):
  a = a + 1
  a = fun2(a)
  return a

def fun2(b):
  b = 2*b
  return b

main()
```

Exemplo Utilizando Funções

- Em uma das aulas anteriores vimos como testar se um número candidato em **cand** é primo:

```
eprimo = True
for div in range(2,cand//2 + 1):
    if cand % div == 0:
        eprimo = False
        break

if eprimo:
    print(cand)
```

Exemplo Utilizando Funções

- Depois usamos este código para imprimir os n primeiros números primos:
- Veja no próximo slide.

Exemplo Utilizando Funções

```
n = int(input('Quantidade de primos:'))

primos_impressos = 0
cand = 2
while primos_impressos < n:
    eprimo = True
    for div in range(2, cand//2 + 1):
        if cand % div == 0:
            eprimo = False
            break

    if eprimo:
        print(cand)
        primos_impressos = primos_impressos + 1
    cand = cand + 1
```

Exemplo Utilizando Funções

- Podemos criar uma função que testa se um número é primo ou não (note que isto é exatamente um bloco que realiza uma computação bem definida).
- Depois fazemos chamadas para esta função.

Exemplo Utilizando Funções

```
def ePrimo(cand):
    for div in range(2, cand//2 + 1):
        if cand % div == 0:
            return False
    #se terminou o laço necessariamente é primo
    return True
```

Exemplo Utilizando Funções

```
def ePrimo(cand):
    for div in range(2, cand//2 + 1):
        if cand % div == 0:
            return False
    #se terminou o laço necessariamente é primo
    return True

def main():
    n = int(input('Quantidade de primos:'))
    primos_impressos = 0
    cand = 2
    while(primos_impressos < n):
        if ePrimo(cand):
            print(cand)
            primos_impressos = primos_impressos + 1
        cand = cand + 1

main()
```

Exercício

- Escreva uma função que computa o fatorial de um número n passado por parâmetro. Sua função deve ter o seguinte protótipo:

  def fat(n): 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 $n = 1, \ldots, 20$.

Exercícios

- Escreva uma função que recebe uma matriz de inteiros como parâmetro e devolve uma lista contendo os elementos de menor e maior frequência de ocorrência na matriz. Assuma que os valores da matriz são inteiros entre 0 e 100.

Exercícios

- Escreva uma função que recebe duas matrizes de floats como parâmetro e devolve uma terceira matriz que é o resultado da soma das duas matrizes passados por parâmetro.

Exercícios

- Escreva uma função que recebe duas matrizes de floats como parâmetro e devolve uma terceira matriz que é o resultado da multiplicação das duas matrizes passados por parâmetro.