Universidade Federal de Pernambuco
Centro de Informática

Subrotinas e Recursão

Prof. Roberto Souto Maior de Barros
roberto@cin.ufpe.br

Subrotinas (subprogramas)

- São estruturas que permitem agrupar comandos em blocos, que recebem um *nome,* e podem ser executados (ativados, chamados) em outras partes dos programas.
- Objetivos:
  - Evitar a repetição de blocos de comandos que teriam que ser escritos em mais de uma parte do programa.
  - Estruturar melhor programas muito grandes visando facilitar o seu entendimento.

Subrotinas – parâmetros

- Parâmetros, semelhantes aos das funções matemáticas, são suportados pela subrotinas na maioria das linguagens de programação.
- Os parâmetros servem para:
  - Tornar as subrotinas *mais genéricas*, aumentando a possibilidade de reutilização de código.
  - Tornar possível escrever o código de subrotinas que sejam *independentes* do código dos programas onde serão utilizadas.
  - Lembrar que em Python (assim como em Java) a passagem de parâmetros é sempre por cópia.
    - Mas se parâmetro for objeto, é passível de alteração...

Subrotinas – parâmetros

- Em Python:
  - Os parâmetros:
    - são delimitados por *parênteses* – (e).
      - Mesmo que *não haja* parâmetros *é necessário* escrever os parênteses, ou seja, ( ).
    - são separados por *vírgulas* (se houver mais de um).
    - não exigem a escrita do *tipo* de cada um deles, como é mais comum na maioria das linguagens.
  - É possível especificar um valor padrão a ser usado em cada parâmetro caso *não* seja fornecido um valor na hora da ativação da subrotina.
    - Pode-se usar apenas um subconjunto deles na ativação...

Subrotinas – tipos

- Conceitualmente, as subrotinas podem ser:
  - Funções
    - Retornam resultados explicitamente.
    - São semelhantes às *funções matemáticas*.
  - Procedimentos
    - *Não* retornam resultados explicitamente.
    - São semelhantes a *programas*, podendo executar comandos quaisquer – partes de um programa.
- *Sintaticamente*, Python só tem funções (o tipo é *function*, e a estrutura é *def*), mas elas também servem para definir procedimentos.

Subrotinas – observações

- O *local do retorno* e a escolha do *valor retornado* são feitos com o comando return...
  - Em Python ele é *opcional*, como em outras linguagens. Se omitido, o retorno se dá ao final do seu código e o valor retornado será *None*.
  - Um comando *return* sem especificar um valor define o local do retorno e será o mesmo que *return None*.
- Em Python, *sintaticamente, não* se escreve o *tipo de retorno* da subrotina, como é mais comum na maioria das linguagens de programação...
  - ele será definido pelo *tipo do valor retornado*.

Subrotinas – exemplos

```
import  math
def  Primo (num) :      # Definição de função, num é o parâmetro…
     raiz = int (math.sqrt (num))
     i = 2
     while  (i <= raiz)  and  ((num % i) != 0) :
          i = i + 1
     if  i > raiz : i = 0
     return  i       # Retorno do resultado da função…

numero = input ("Digite um número inteiro: ")
res = Primo (numero)            # Chamada ou ativação da subrotina…
if  res == 0 : print (numero, ‘é primo.‘)
else:
     print (numero, 'não é primo:’, res, ‘é um divisor.')
```

Subrotinas – exemplos

```
# Definição de procedimento usando valores padrão nos parâmetros.
def  repMens  (m = 'Hello!',  qtd = 1) :
     m = m * qtd     # Valor de  m  só é alterado dentro da subrotina…
     print (m)
     return      # return é opcional: não retorna resultado de verdade...
repMens ('Olá!', 3)     # Imprime  'Olá!Olá!Olá!'
repMens ('Olá!')        # Imprime  'Olá!'
repMens ( )             # Imprime  'Hello!'
repMens (qtd=3)         # Imprime  'Hello!Hello!Hello!'
res = repMens ('Olá!', 3)    # Imprime  'Olá!Olá!Olá!'  e  res = None
repMens (4, 3)          # Imprime  '12'!!!!

mens = 'Mensagem'
repMens (mens, 3)    # mens  permanece com  'Mensagem'...
```

Exercício revisitado

- Fazer um programa para:
  - Ler uma tabela com N profissões, onde
    - O valor de N é informado antes pelo usuário.
    - Cada profissão é formada por um código (*número positivo*), um nome (*String*) e uma área (*String*).
    - Leitura da tabela deve ser feita em subrotina.
  - Depois o usuário fornecerá uma lista de códigos para que o programa informe o nome de cada profissão.
  - Se o código da profissão não existir na tabela, mostrar a mensagem “Profissão Inexistente” e continuar.
  - O programa pára com a digitação de um código inválido (negativo ou zero).

Resolução 1 – vista anteriormente


```
# Profissões - V1 - Tabela = Dicionário com chaves e resto dos registros.

n = int(input ('Digite o tamanho da tabela de profissões: '))
while (n < 1) :
    n = int(input ('Tamanho deve ser inteiro positivo. Tente novamente: '))

tab = { }       # Criação do dicionário...

for  i  in  range (n) :
    codP = int(input ('Digite o código de uma profissão: '))
    while (codP < 1) :
        codP = int(input ('Código deve ser inteiro positivo. Tente novamente: '))
    nomeP = input ('Digite o nome da profissão %d:\n'  %  (codP))
    areaP = input ('Digite a área da profissão %d:\n'  %  (codP))
    tab [codP] = (nomeP, areaP)       # Inserção no dicionário...
```

Resolução 1 – vista anteriormente

```
...
print  ('Tabela com %d profissões foi lida corretamente.'  %  (n))
print  ('Tabela ->', tab)

codP = int(input ('Digite um código de profissão para busca (<=0 para parar): '))

while  codP > 0 :
    if  codP  in  tab :       # Verifica se a profissão existe na tabela...
        nomeP, areaP  =  tab[codP]      # Recupera os outros dados...
        print ('Profissão %d é %s e sua área é %s.'  %  (codP, nomeP, areaP))
    else:
        print ('Profissão %d não existe na tabela.'  %  (codP))

    codP = int(input ('Digite outro código para busca (<=0 para parar): '))

print  ('Fim de Programa‘)
```

Resolução 2 – com subrotina S1


```
# Profissões - S1 - Tabela = Dicionário e função com resultado.

def  preencheTab ( ) :
     n = int(input ('Digite o tamanho da tabela de profissões: '))
     while (n < 1) :
          n = int(input ('Tamanho deve ser inteiro positivo. Tente novamente: '))

     tabela = { }    # Cria a tabela na subrotina para retorná-la no final.
     for  i  in  range (n) :
          codP = int(input ('Digite o código de uma profissão: '))
          while (codP < 1) :
               codP = int(input ('Código deve ser inteiro positivo. Tente novam.: '))
          nomeP = input ('Digite o nome da profissão %d:\n' % (codP))
          areaP = input ('Digite a área da profissão %d:\n' % (codP))
          tabela [codP] = (nomeP, areaP)       # Inserção no dicionário...
     return  tabela    # O resultado retornado é o endereço do objeto  tabela...
```

Resolução 2 – com subrotina S1

```
# Programa principal...
tab = preencheTab ( )    # variável  tab  receberá o endereço da tabela...
print  ('Tabela com %d profissões foi lida corretamente.' % (len(tab)))
print  ('Tabela ->', tab)

codP = int(input ('Digite um código de profissão para busca (<=0 para parar): '))

while  codP > 0 :
    if  codP  in  tab :       # Verifica se a profissão existe na tabela...
        nomeP, areaP  =  tab[codP]       # Recupera os outros dados...
        print ('Profissão %d é %s e sua área é %s.'  %  (codP, nomeP, areaP))
    else:
        print ('Profissão %d não existe na tabela.'  %  (codP))

    codP = int(input ('Digite outro código para busca (<=0 para parar): '))

print  ('Fim de Programa‘)
```

Resolução 3 – com subrotina S2


```
# Profissões – S2 - Tabela = Dicionário e procedimento com parâmetro.

def  preencheTab (tabela) :   # Recebe objeto  tabela  já criado para preencher...
     n = int(input ('Digite o tamanho da tabela de profissões: '))
     while (n < 1) :
          n = int(input ('Tamanho deve ser inteiro positivo. Tente novamente: '))
     for  i  in  range (n) :
          codP = int(input ('Digite o código de uma profissão: '))
          while (codP < 1) :
               codP = int(input ('Código deve ser inteiro positivo. Tente novam.: '))
          nomeP = input ('Digite o nome da profissão %d:\n' % (codP))
          areaP = input ('Digite a área da profissão %d:\n' % (codP))
          tabela [codP] = (nomeP, areaP)        # Inserção no dicionário...
     return    # Este return pode ser omitido.
```

Resolução 3 – com subrotina S2

```
# Programa principal...
tab  =  { }    # Cria a tabela e passa o endereço para a subrotina preencher...
preencheTab (tab)
print  ('Tabela com %d profissões foi lida corretamente.' % (len(tab)))
print  ('Tabela ->', tab)

codP = int(input ('Digite um código de profissão para busca (<=0 para parar): '))

while  codP > 0 :
    if  codP  in  tab :      # Verifica se a profissão existe na tabela...
        nomeP, areaP  =  tab[codP]      # Recupera os outros dados...
        print  ('Profissão %d é %s e sua área é %s.'  %  (codP, nomeP, areaP))
    else:
        print  ('Profissão %d não existe na tabela.'  %  (codP))

    codP = int(input ('Digite outro código para busca (<=0 para parar): '))
print  ('Fim de Programa‘)
```

Recursão

- Um subprograma (subrotina) é *recursivo* se ele é definido em termos de si mesmo. Na prática isto significa que:
  - ele possui dentro de seu corpo de comandos uma ou mais *chamadas a si mesmo*.
  - Obviamente que com *parâmetros diferentes*, senão ele ficaria em *loop infinito*.
- Obs: É relativamente fácil escrever um método recursivo, o mais difícil é reconhecer as situações em que é apropriado fazer isto.

Recursão

- Em muitos casos, a idéia é semelhante à prova de propriedades por indução finita na matemática:
  - Condição básica – caso inicial.
  - Redução do problema – prova em função do anterior.
- No caso da recursão em subprogramas:
  - A condição básica será a condição de parada das chamadas recursivas – o resultado é conhecido.
  - A redução do problema será feita escrevendo o cálculo atual em função do resultado do valor anterior, desde que *não seja a condição básica*.

Recursão

- Observações importantes
  - Já foi provado que tudo que se faz usando recursão também se faz *sem recursão*, usando comandos de repetição.
  - Nos casos de *recursão linear* (em um único caminho), as duas soluções são “equivalentes” em termos de eficiência ou complexidade.
  - Quando a recursão se dá em *mais de um caminho*, os subprogramas recursivos tendem a ser *muito menores* e *mais intuitivos* do que os não recursivos.
    - Caso por exemplo dos algoritmos para implementação de árvores.

Recursão

- Exemplo:
  - Cálculo do fatorial – subprograma usando repetição:

```
def  fatorial  (num) :
        f = 1
        for  i  in  range (2, num + 1) :
            f = f * i
        return  f

...
fat = fatorial (5)
```

Recursão

- Exemplo:
  - Cálculo do fatorial – subprograma recursivo:

```
def  fatorial  (num) :
        if  num < 2 :
              f = 1
        else :
              f  =  num * fatorial (num – 1)
        return  f
…
fat = fatorial (5)
```

Recursão

- Problemas de cálculo de séries matemáticas em geral também podem ser resolvidos facilmente usando subrotinas recursivas
  - O número de termos pode ser usado como base para controlar a recursão.
  - Parâmetros auxiliares podem ser utilizados para levar resultados ou valores para as chamadas recursivas.
    - Neste caso, pode ser interessante usar um outro nome para esconder estes parâmetros dos usuários.

Recursão

- Exemplo: S = 1 + 3/2 + 5/3 + 7/4 + ... (n termos)

```
def  serieR (n, nu = 1, de = 1.0) :
    if  n <= 1 :
        res = nu / de
    else :
        res =  nu / de + serieR (n – 1, nu + 2, de + 1)
    return res

num = int(input('Número de termos: '))
res = serieR(num)
print(res)
```

Recursão

- Solução alternativa:

```
def  serieR (n, nu = 1, de = 1.0) :
    res = nu / de
    if  n > 1 :
        res =  res + serieR (n – 1, nu + 2, de + 1)
    return res
```

- Se quiser esconder os parâmetros adicionais…

```
def  serie (n) :
    return serieR (n, 1, 1.0)
```