Aula 4

Professores:

Dante Corbucci Filho
Leandro A. F. Fernandes

Conteúdo:

- Subprogramação:
  - Funções
  - Passagem de Parâmetros
  - Recursividade

Subprogramação: Funções

- A utilização de funções permite que:
  - Diferentes partes do programa possam ser desenvolvidas e testadas separadamente;
  - Partes do código possam ser reutilizadas em diferentes pontos do programa;
  - Programas complexos possam ser montados a partir de unidades menores já desenvolvidas e testadas.

Subprogramação: Funções

- A utilização de funções permite que:
  - Diferentes partes do programa possam ser desenvolvidas e testadas separadamente;
  - Partes do código possam ser reutilizadas em diferentes pontos do programa;
  - Programas complexos possam ser montados a partir de unidades menores já desenvolvidas e testadas.
- Trata-se de um grupo de sentenças (suite), comando(s) e/ou estrutura(s) de controle (e funções internas), ao qual é atribuído um nome, que após a sua execução retorna um valor.

Subprogramação: Funções

- A utilização de funções permite que:
  - Diferentes partes do programa possam ser desenvolvidas e testadas separadamente;
  - Partes do código possam ser reutilizadas em diferentes pontos do programa;
  - Programas complexos possam ser montados a partir de unidades menores já desenvolvidas e testadas.
- Trata-se de um grupo de sentenças (suite), comando(s) e/ou estrutura(s) de controle (e funções internas), ao qual é atribuído um nome, que após a sua execução retorna um valor.
- Sua ativação se dá através de seu nome ou de expressões que o contenha.

Declaração de Uma Função em Python

def nomeEscolhido(*lista de parâmetros*):

suite do corpo da função

A *lista de parâmetros* pode ter zero ou mais parâmetros, separados por vírgulas.

Declaração de Uma Função em Python

def nomeEscolhido(*lista de parâmetros*):

suite do corpo da função

A *lista de parâmetros* pode ter zero ou mais parâmetros, separados por vírgulas.

A *suite do corpo da função* deve possuir zero ou mais retornos de valores, expressos por

return *valor apropriado*

Caso nenhum valor seja retornado, corresponde a

return ou **return None**

Esboço de Um Programa Contendo Funções

Sempre que possível, as funções devem ser declaradas antes de serem utilizadas. Portanto, devem ficar acima do programa principal ou de outras funções que as utilize.

```
# Programa Completo
# Subprogramas
def nomeFun1(<listaParametros1>):
    <corpo nomeFun1>
...
def nomeFunN(<listaParametrosN>):
    <corpo nomeFunN>
```

Esboço de Um Programa Contendo Funções

Sempre que possível, as funções devem ser declaradas antes de serem utilizadas. Portanto, devem ficar acima do programa principal ou de outras funções que as utilize.

```
# Programa Completo
# Subprogramas
def nomeFun1(<listaParametros1>):
    <corpo nomeFun1>
...
def nomeFunN(<listaParametrosN>):
    <corpo nomeFunN>
# Programa Principal
utiliza nomeFun1(...) ou/e nomeFunN(...)
utiliza nomeFunN(...) ou/e nomeFunN(...)
```

Esboço de Um Programa Contendo Funções

Sempre que possível, as funções devem ser declaradas antes de serem utilizadas. Portanto, devem ficar acima do programa principal ou de outras funções que as utilize.

```
# Programa Completo
# Subprogramas
def nomeFun1(<listaParametros1>):
    <corpo nomeFun1>
...
def nomeFunN(<listaParametrosN>):
    <corpo nomeFunN>
# Programa Principal
utiliza nomeFun1(...) ou/e nomeFunN(...)
utiliza nomeFunN(...) ou/e nomeFunN(...)
```

Portanto, um **programa** passa a ser composto de um **programa principal** e vários outros **subprogramas**, definidos em seu início.

Programa com Variáveis Globais e Funções

Um programa pode conter variáveis globais. No entanto, o uso de variáveis globais não é considerada uma boa prática de programação, por dificultar a legibilidade e compreensão dos códigos. Desta forma, não mais utilizaremos este conceito, após esta aula.

```
# Programa Completo com Variáveis Globais
global var1
...
global varM
# Subprogramas
def nomeFun1(<listaParametros1>):
    <corpo nomeFun1>
...
def nomeFunN(<listaParametrosN>):
    <corpo nomeFunN>
# Programa Principal
utiliza var1 e/ou varM e/ou nomeFun1(...) e/ou nomeFunN(...)
utiliza var1 e/ou varM e/ou nomeFun1(...) e/ou nomeFunN(...)
```

Variáveis Globais vs. Passagem de Parâmetros

Veremos dois programas, um utilizando variáveis globais e outro utilizando passagem de parâmetros, que acessam funções que calculam a multiplicação de dois números através de somas sucessivas.

- A função **mult** retorna o resultado da multiplicação das variáveis **x** e **y**, através do cálculo de sucessivas somas.

- As chamadas **mult()** ou **mult(x, y)** representarão, além da ativação da função, o valor a ser retornado.

Variáveis Globais (Escopo)

```
# Programa Completo com Variáveis Globais
global x
global y
# Subprogramas
def mult():
    z = 0
    for u in range(x):
        z = z + y
    return z
# Programa Principal
x = 20
y = 200
w = mult() + 10
# Neste ponto, w vale 4010
print(w)
```

Variáveis Globais (Escopo)

```
# Programa Completo com Variáveis Globais
global x
global y
# Subprogramas
def mult():
    z = 0
    for u in range(x):
        z = z + y
    return z
# Programa Principal
x = 20
y = 200
w = mult() + 10
# Neste ponto, w vale 4010
print(w)
```

Variáveis Globais (Escopo)

```
# Programa Completo com Variáveis Globais
global x
global y
# Subprogramas
def mult():
    z = 0
    for u in range(x):
        z = z + y
    return z
# Programa Principal
x = 20
y = 200
w = mult() + 10
# Neste ponto, w vale 4010
print(w)
```

Variáveis Globais (Escopo)

```
# Programa Completo com Variáveis Globais
global x
global y
# Subprogramas
def mult():
    z = 0
    for u in range(x):
        z = z + y
    return z
# Programa Principal
x = 20
y = 200
w = mult() + 10
# Neste ponto, w vale 4010
print(w)
```

A função **mult** é ativada nesta atribuição e retorna o valor 4000

Passagem de Parâmetros

Caso uma função deva ser aplicada a diferentes operandos, ou seja, a diferentes valores de entrada, a definição desta função deve conter parâmetros.

Passagem de Parâmetros

```
# Programa Completo sem Variáveis Globais

# Subprogramas
def mult(x,y):
    z = 0
    for u in range(x):
        z = z + y
    return z

# Programa Principal
w = mult(20, 200) + 10
# Neste ponto, w vale 4010
print(w)
w = mult(10, 100) + 20
# Neste ponto, w vale 1020
print(w)
```

Passagem de Parâmetros

```
# Programa Completo sem Variáveis Globais

# Subprogramas
def mult(x,y):
    z = 0
    for u in range(x):
        z = z + y
    return z

# Programa Principal
w = mult(20, 200) + 10
# Neste ponto, w vale 4010
print(w)
w = mult(10, 100) + 20
# Neste ponto, w vale 1020
print(w)
```

Observe que as variáveis **x** e **y**, que antes eram variáveis globais, agora são parâmetros da função **mult** e só são utilizadas dentro desta.

Passagem de Parâmetros

```
# Programa Completo sem Variáveis Globais

# Subprogramas
def mult(x,y):
    z = 0
    for u in range(x):
        z = z + y
    return z

# Programa Principal
w = mult(20, 200) + 10
# Neste ponto, w vale 4010
print(w)
w = mult(10, 100) + 20
# Neste ponto, w vale 1020
print(w)
```

Observe que as variáveis **x** e **y**, que antes eram variáveis globais, agora são parâmetros da função **mult** e só são utilizadas dentro desta.

Na **primeira ativação** da função **mult**, os parâmetros **x** e **y** recebem os valores 20 e 200

Passagem de Parâmetros

```
# Programa Completo sem Variáveis Globais

# Subprogramas
def mult(x,y):
    z = 0
    for u in range(x):
        z = z + y
    return z

# Programa Principal
w = mult(20, 200) + 10
# Neste ponto, w vale 4010
print(w)
w = mult(10, 100) + 20
# Neste ponto, w vale 1020
print(w)
```

Observe que as variáveis **x** e **y**, que antes eram variáveis globais, agora são parâmetros da função **mult** e só são utilizadas dentro desta.

Na **primeira ativação** da função **mult**, os parâmetros **x** e **y** recebem os valores 20 e 200

Na **segunda ativação** da função **mult**, os parâmetros **x** e **y** recebem os valores 10 e 100

Passagem de Parâmetros em Python

- Passagem de parâmetro por valor:
  - No início da função, os parâmetros sempre são inicializados com a cópia das referências (ponteiros) para os valores passados na ativação da função
    - Valores de tipos básicos são imutáveis
    - Valores de tipos estruturados (a serem vistos) são mutáveis

Passagem de Parâmetros em Python

- Passagem de parâmetro por valor:
  - No início da função, os parâmetros sempre são inicializados com a cópia das referências (ponteiros) para os valores passados na ativação da função
    - Valores de tipos básicos são imutáveis
    - Valores de tipos estruturados (a serem vistos) são mutáveis
  - Os valores podem vir de
    - Constantes
    - Variáveis
    - Resultados de funções
    - Ou seja, resultados de expressões

Passagem de Parâmetros em Python

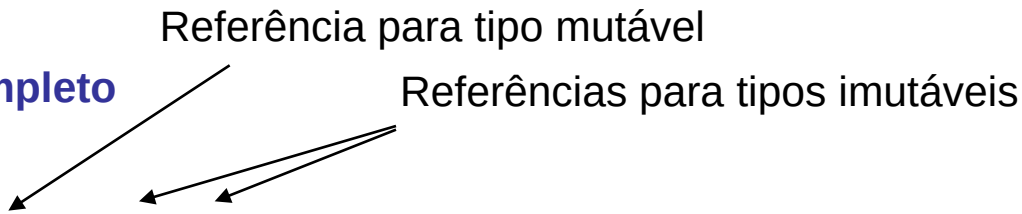
- Passagem de parâmetro por valor:
  - No início da função, os parâmetros sempre são inicializados com a cópia das referências (ponteiros) para os valores passados na ativação da função
    - Valores de tipos básicos são imutáveis
    - Valores de tipos estruturados (a serem vistos) são mutáveis
  - Os valores podem vir de
    - Constantes
    - Variáveis
    - Resultados de funções
    - Ou seja, resultados de expressões
  - Observe estas diferentes ativações da função **mult**:

w = **mult**(20, 200) + 10

w = **mult**(a, b) + 10

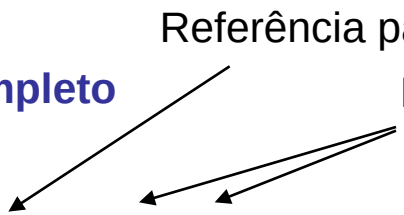
w = 13 + **mult**(a+b, **mult**(c,d))

Tipos Mutáveis e Imutáveis como Parâmetro




```
# Programa Completo

# Subprograma
def trocar(valores, pos1, pos2):        # se possível, modifica o conteúdo de duas células
    if 0<=pos1<len(valores) and 0<=pos2<len(valores):
        temp = valores[pos1]
        valores[pos1] = valores[pos2]
        valores[pos2] = temp
    return None

# Programa Principal
amigas = [“Maria”, “Regina”, “Eliana”, “Angelica”]   # vetor com 4 strings – próximas aulas
trocar(amigas, 3, 1)
# Neste ponto, amigas = [“Maria”, “Angelica”, “Eliana”, “Regina”]

trocar(amigas, 0, 2)
# Neste ponto, amigas = [“Eliana”, “Angelica”, “Maria”, “Regina”]
```

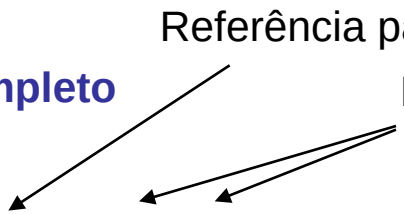
Tipos Mutáveis e Imutáveis como Parâmetro




```
# Programa Completo

# Subprograma
def trocar(valores, pos1, pos2):        # se possível, modifica o conteúdo de duas células
    if 0<=pos1<len(valores) and 0<=pos2<len(valores):
        temp = valores[pos1]
        valores[pos1] = valores[pos2]
        valores[pos2] = temp
    return None

# Programa Principal
amigas = [“Maria”, “Regina”, “Eliana”, “Angelica”]   # vetor com 4 strings – próximas aulas
trocar(amigas, 3, 1)
# Neste ponto, amigas = [“Maria”, “Angelica”, “Eliana”, “Regina”]

trocar(amigas, 0, 2)
# Neste ponto, amigas = [“Eliana”, “Angelica”, “Maria”, “Regina”]
```

Esta passagem de parâmetro ocorre como se o primeiro argumento (a variável **amigas** – também chamado “parâmetro real”) substituísse o parâmetro dentro do escopo do função (o “parâmetro formal” **valores**).

Tipos Mutáveis e Imutáveis como Parâmetro




```
# Programa Completo

# Subprograma
def trocar(valores, pos1, pos2):      # se possível, modifica o conteúdo de duas células
    if 0<=pos1<len(valores) and 0<=pos2<len(valores):
        temp = valores[pos1]
        valores[pos1] = valores[pos2]
        valores[pos2] = temp
    return None

# Programa Principal
amigas = [“Maria”, “Regina”, “Eliana”, “Angelica”]   # vetor com 4 strings – próximas aulas
trocar(amigas, 3, 1)
# Neste ponto, amigas = [“Maria”, “Angelica”, “Eliana”, “Regina”]

trocar(amigas, 0, 2)
# Neste ponto, amigas = [“Eliana”, “Angelica”, “Maria”, “Regina”]
```

Esta passagem de parâmetro ocorre como se o primeiro argumento (a variável **amigas** – também chamado “parâmetro real”) substituísse o parâmetro dentro do escopo do função (o “parâmetro formal” **valores**).

Esta é a forma de se modificar, dentro de uma função, conteúdos de variáveis de um programa principal.

Funções Como Parâmetros

```
# Programa Completo
# Subprogramas
def fatorial(num):
    if num==0:
        return 1
    else:
        return num*fatorial(num-1)

def fib(num):
    if 1<=num<=2:
        return 1
    else:
        return fib(num-1)+fib(num-2)


def soma(f, n):      # Esta função soma os n primeiros valores de uma dada função f

    parcial = 0

    for ind in range(1, n+1):

        parcial = parcial + f(ind)

    return parcial


# Programa Principal
total =  soma(fatorial,10) + soma(fib,10)
print(total)
```

Funções Como Parâmetros

```
# Programa Completo
# Subprogramas
def fatorial(num):
    if num==0:
        return 1
    else:
        return num*fatorial(num-1)

def fib(num):
    if 1<=num<=2:
        return 1
    else:
        return fib(num-1)+fib(num-2)

def soma(f, n):      # Esta função soma os n primeiros valores de uma dada função f
    parcial = 0
    for ind in range(1, n+1):
        parcial = parcial + f(ind)
    return parcial

# Programa Principal
total =  soma(fatorial,10) + soma(fib,10)
print(total)
```

Funções Como Parâmetros

```
# Programa Completo
# Subprogramas
def fatorial(num):
    if num==0:
        return 1
    else:
        return num*fatorial(num-1)

def fib(num):
    if 1<=num<=2:
        return 1
    else:
        return fib(num-1)+fib(num-2)

def soma(f, n):      # Esta função soma os n primeiros valores de uma dada função f
    parcial = 0
    for ind in range(1, n+1):
        parcial = parcial + f(ind)
    return parcial

# Programa Principal
total =  soma(fatorial,10) + soma(fib,10)
print(total)
```

Funções Como Parâmetros

```
# Programa Completo
# Subprogramas
def fatorial(num):
    if num==0:
        return 1
    else:
        return num*fatorial(num-1)

def fib(num):
    if 1<=num<=2:
        return 1
    else:
        return fib(num-1)+fib(num-2)

def soma(f, n):      # Esta função soma os n primeiros valores de uma dada função f
    parcial = 0
    for ind in range(1, n+1):
        parcial = parcial + f(ind)
    return parcial

# Programa Principal
total =  soma(fatorial,10) + soma(fib,10)
print(total)
```

Funções Como Parâmetros

```
# Programa Completo
# Subprogramas
def fatorial(num):
    if num==0:
        return 1
    else:
        return num*fatorial(num-1)

def fib(num):
    if 1<=num<=2:
        return 1
    else:
        return fib(num-1)+fib(num-2)

def soma(f, n):      # Esta função soma os n primeiros valores de uma dada função f
    parcial = 0
    for ind in range(1, n+1):
        parcial = parcial + f(ind)
    return parcial

# Programa Principal
total = soma(fatorial,10) + soma(fib,10)
print(total)
```

Funções Como Parâmetros

```
# Programa Completo
# Subprogramas
def fatorial(num):
    if num==0:
        return 1
    else:
        return num*fatorial(num-1)

def fib(num):
    if 1<=num<=2:
        return 1
    else:
        return fib(num-1)+fib(num-2)

def soma(f, n):      # Esta função soma os n primeiros valores de uma dada função f
    parcial = 0
    for ind in range(1, n+1):
        parcial = parcial + f(ind)
    return parcial

# Programa Principal
total = soma(fatorial,10) + soma(fib,10)
print(total)
```

Escopo de um Identificador em Python

Os parâmetros, as variáveis locais e as funções declaradas internamente a uma função definem identificadores que são **locais** a esta função, isto é: têm **escopo local**.

Estes identificadores não podem ser utilizados fora da respectiva função, isto é, não são **visíveis** em outra parte do programa.

Escopo de um Identificador em Python

- A região de validade de um identificador é chamada de escopo do identificador.

Escopo de um Identificador em Python

- A região de validade de um identificador é chamada de escopo do identificador.

- Um identificador é chamado de global se o seu escopo é todo o programa.

  – Isto é: o programa principal e os seus subprogramas.

Escopo de um Identificador em Python

- A região de validade de um identificador é chamada de escopo do identificador.

- Um identificador é chamado de global se o seu escopo é todo o programa.

  – Isto é: o programa principal e os seus subprogramas.

- O escopo de um identificador é dito local se ele é válido apenas na função que é definido.

Escopo de um Identificador em Python

```
# Programa Completo
global x                        # x escopo global

# Subprogramas
def p(n):                       # n escopo local a p
    y = 13                      # y escopo local a p
                                # Neste ponto, x, y e n são conhecidos
    return x+y+n

def q(m):                       # m escopo local a q
    z = 3                       # z escopo local a q
                                # Neste ponto, x, z e m são conhecidos
    return m**z - x*m

# Programa Principal
x = 26
# Neste ponto, apenas x, p e q são conhecidos
print(p(x), q(2*x+p(3)))
```

Escopo de um Identificador em Python

```
# Programa Completo
global x                        # x escopo global

# Subprogramas
def p(n):                       # n escopo local a p
    y = 13                      # y escopo local a p
                                # Neste ponto, x, y e n são conhecidos
    return x+y+n

def q(m):                       # m escopo local a q
    z = 3                       # z escopo local a q
                                # Neste ponto, x, z e m são conhecidos
    return m**z - x*m

# Programa Principal
x = 26
# Neste ponto, apenas x, p e q são conhecidos
print(p(x), q(2*x+p(3)))
```

Escopo de um Identificador em Python

```
# Programa Completo
global x                        # x escopo global

# Subprogramas
def p(n):                       # n escopo local a p
    y = 13                      # y escopo local a p
                                # Neste ponto, x, y e n são conhecidos
    return x+y+n

def q(m):                       # m escopo local a q
    z = 3                       # z escopo local a q
                                # Neste ponto, x, z e m são conhecidos
    return m**z - x*m

# Programa Principal
x = 26
# Neste ponto, apenas x, p e q são conhecidos
print(p(x), q(2*x+p(3)))
```

Escopo de um Identificador em Python

```
# Programa Completo
global x                        # x escopo global

# Subprogramas
def p(n):                       # n escopo local a p
    y = 13                      # y escopo local a p
                                # Neste ponto, x, y e n são conhecidos
    return x+y+n

def q(m):                       # m escopo local a q
    z = 3                       # z escopo local a q
                                # Neste ponto, x, z e m são conhecidos
    return m**z - x*m

# Programa Principal
x = 26
# Neste ponto, apenas x, p e q são conhecidos
print(p(x), q(2*x+p(3)))
```

Escopo de um Identificador em Python

```
# Programa Completo
global x
```

Variável Global **x**

```
# Subprograma
def colisao(m):
    x = 8
    z = 13
    # Neste ponto, as variáveis x e z locais a q
    # e o parâmetro m são conhecidos.
    x = x + 1
    # A variável local x foi alterada.
    print(x, z, m*2) # escreve 9, 13 e 2000
    return None

# Programa Principal
# Neste ponto, apenas variável global x é conhecida,  além do nome da função.
x = 57
colisao(1000)
print(x) # escreve 57
```

No caso de existirem dois identificadores, definidos em escopos diferentes, com o mesmo nome **x**, a ocorrência do nome **x** estará referenciando aquele com o escopo mais local.

Escopo de um Identificador em Python

```
# Programa Completo
global x
```

Variável Global **x**

```
# Subprograma
def colisao(m):
    x = 8
    z = 13
    # Neste ponto, as variáveis x e z locais a q
    # e o parâmetro m são conhecidos.
    x = x + 1
    # A variável local x foi alterada.
    print(x, z, m*2) # escreve 9, 13 e 2000
    return None
```

No caso de existirem dois identificadores, definidos em escopos diferentes, com o mesmo nome **x**, a ocorrência do nome **x** estará referenciando aquele com o escopo mais local.

```
# Programa Principal
# Neste ponto, apenas variável global x é conhecida,  além do nome da função.
x = 57
colisao(1000)
print(x) # escreve 57
```

Escopo de um Identificador em Python

```
# Programa Completo
global x
```

Variável Global **x**

```
# Subprograma
def colisao(m):
    x = 8
    z = 13
    # Neste ponto, as variáveis x e z locais a q
    # e o parâmetro m são conhecidos.
    x = x + 1
    # A variável local x foi alterada.
    print(x, z, m*2) # escreve 9, 13 e 2000
    return None
```

Variável Local **x**

No caso de existirem dois identificadores, definidos em escopos diferentes, com o mesmo nome **x**, a ocorrência do nome **x** estará referenciando aquele com o escopo mais local.

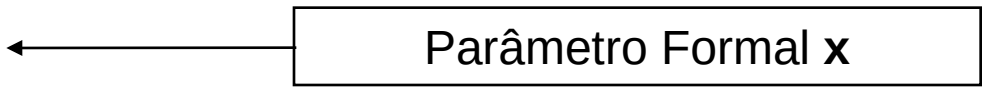
```
# Programa Principal
# Neste ponto, apenas variável global x é conhecida,  além do nome da função.
x = 57
colisao(1000)
print(x) # escreve 57
```

Escopo de um Identificador em Python

```
# Programa Completo
# Subprograma
def calcula(x,y):
    # Função Interna
    def cubo(z):
        return z**3
    #
    return cubo(x) - y
# Programa Principal
x  = calcula(10,20)
# Neste ponto, x vale (10*10*10)-20.
print(x) # escreve 980
```

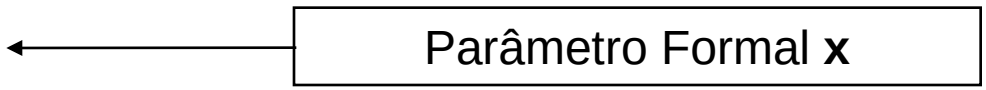
Escopo de um Identificador em Python

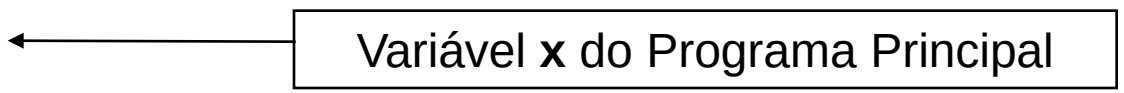
```
# Programa Completo
# Subprograma
def calcula(x,y):
    # Função Interna
    def cubo(z):
        return z**3
    #
    return cubo(x) - y
# Programa Principal
x  = calcula(10,20)
# Neste ponto, x vale (10*10*10)-20.
print(x) # escreve 980
```



Escopo de um Identificador em Python

```
# Programa Completo
# Subprograma
def calcula(x,y):
    # Função Interna
    def cubo(z):
        return z**3
    #
    return cubo(x) - y
# Programa Principal
x  = calcula(10,20)
# Neste ponto, x vale (10*10*10)-20.
print(x) # escreve 980
```





Ativação de Funções

Quando uma função é chamada, duas tarefas são executadas:

a) Criação de um espaço de memória para as variáveis locais e parâmetros; e

b) Passagem efetiva de parâmetros.

Ativação de Funções

- Durante a execução de um programa:
  - Uma área especial de memória, organizada em forma de pilha (“stack”), é utilizada para armazenar os valores
    - das variáveis locais e
    - parâmetros das funções;
  - Outra área é organizada para manter as variáveis globais do programa principal.

- Quando o programa principal é iniciado, um espaço é criado para manter as variáveis globais.

- Sempre que uma função é ativada, um espaço contendo os valores das variáveis locais e dos parâmetros é reservado no topo da pilha.

- Ao fim de uma função, seu espaço é automaticamente eliminado do topo da pilha, podendo ser reutilizado.

Ativação de Funções

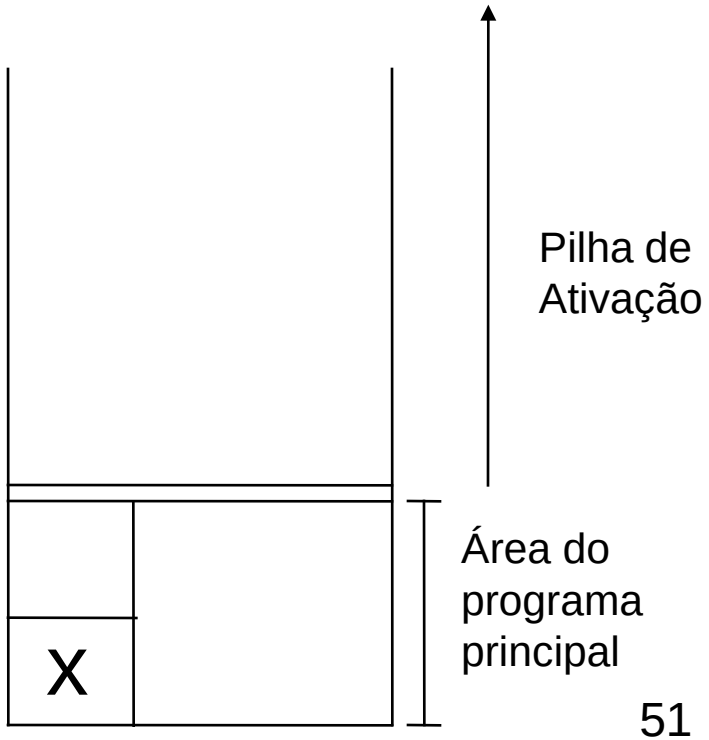
```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```

Pilha de Ativação e Registro de Ativação de Funções

```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```

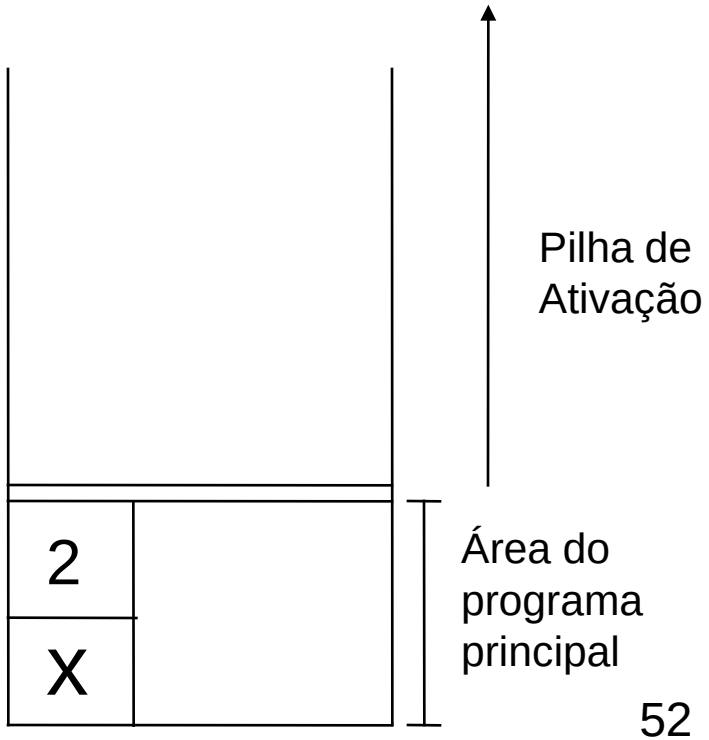
Pilha de Ativação e Registro de Ativação de Funções

```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```



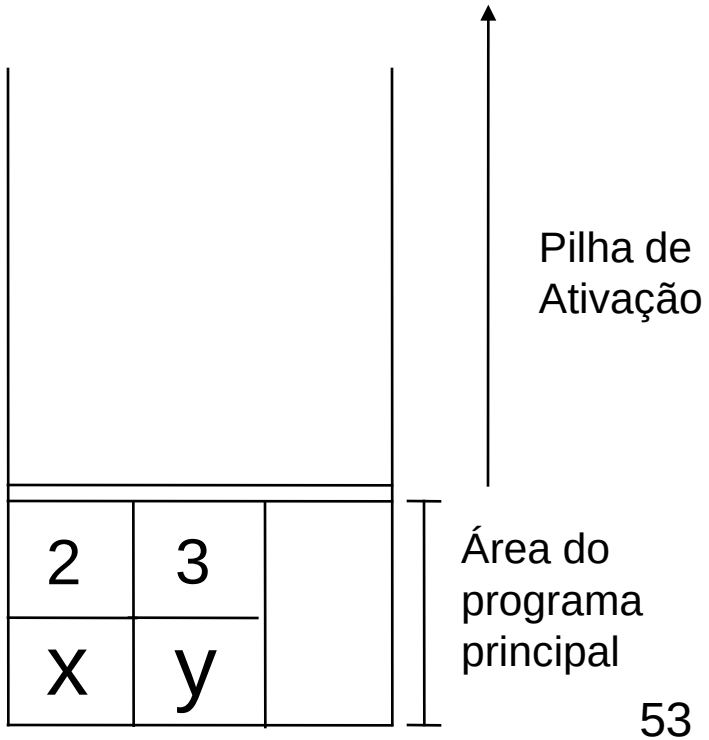
Pilha de Ativação e Registro de Ativação de Funções

```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```



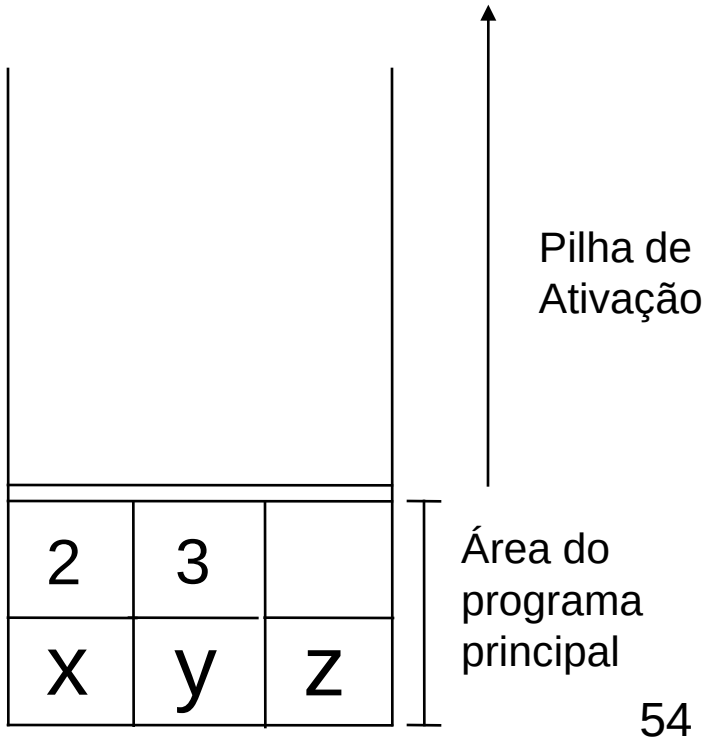
Pilha de Ativação e Registro de Ativação de Funções

```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```



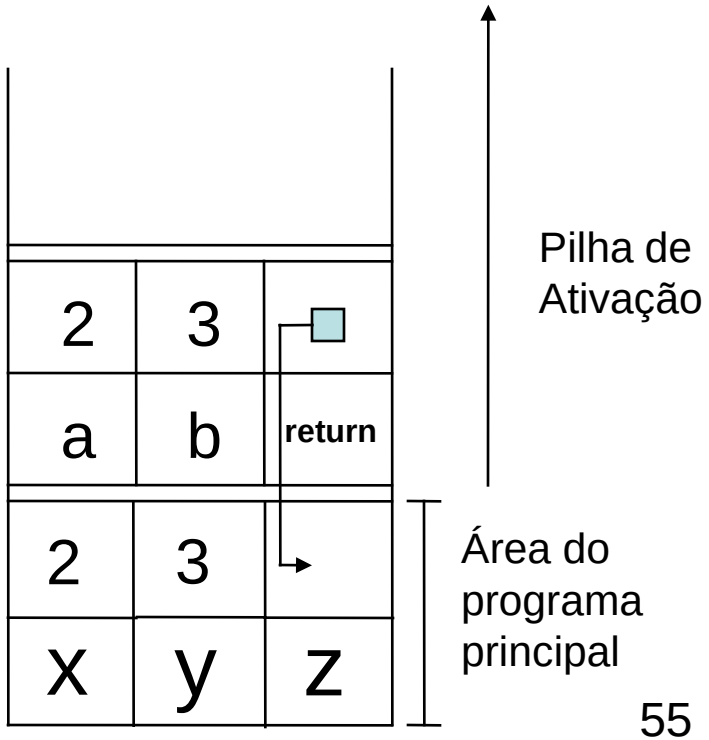
Pilha de Ativação e Registro de Ativação de Funções

```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```



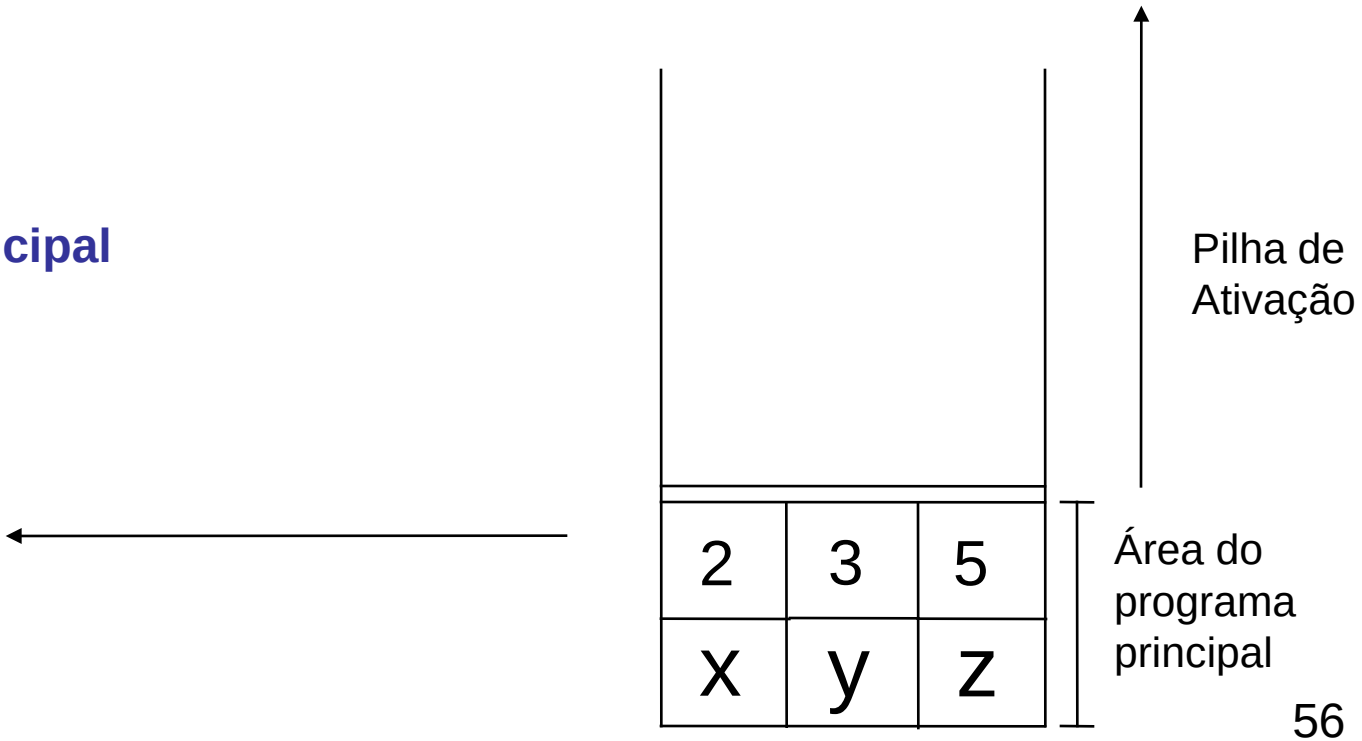
Pilha de Ativação e Registro de Ativação de Funções

```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```



Pilha de Ativação e Registro de Ativação de Funções

```
# Programa Completo

# Subprograma
def soma(a,b):
    return a + b

# Programa Principal
x = 2
y = 3
z = soma(x,y)
print(z)
```



Recursividade

- Uma função é chamada recursiva quando possui no seu corpo uma chamada a ela própria.

- Um exemplo comum de utilização de recursividade é o cálculo da função fatorial de um número natural.

$$n! = \begin{cases} 1 & , \text{se } n = 0; \quad \{\text{caso base}\} \\ n * (n-1)! & , \text{se } n > 0. \quad \{\text{expressão de recorrência}\} \end{cases}$$

$$3! = 3*2! = 3*2*1! = 3*2*1*0! = 3*2*1*1 = 6$$

Recursividade

$$n! = \begin{cases} 1 & , \text{se } n = 0; \quad \{\text{caso base}\} \\ n * (n-1)! & , \text{se } n > 0. \quad \{\text{expressão de recorrência}\} \end{cases}$$

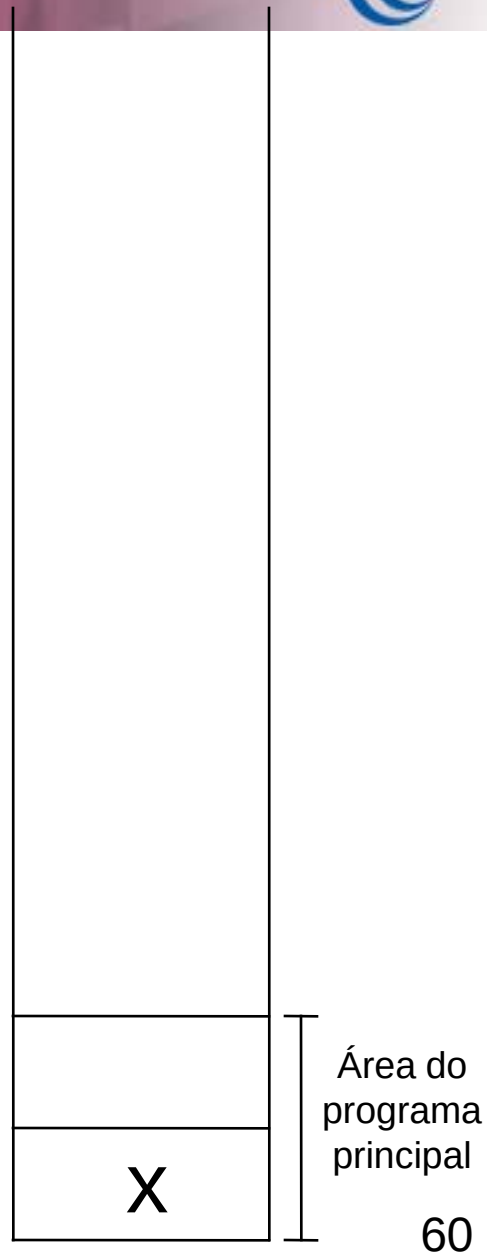
```
def fat(n):
    if n == 0:                      # condição de parada
        return 1
    else:
        return n*fat(n-1)           # chamada recursiva
```

Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```

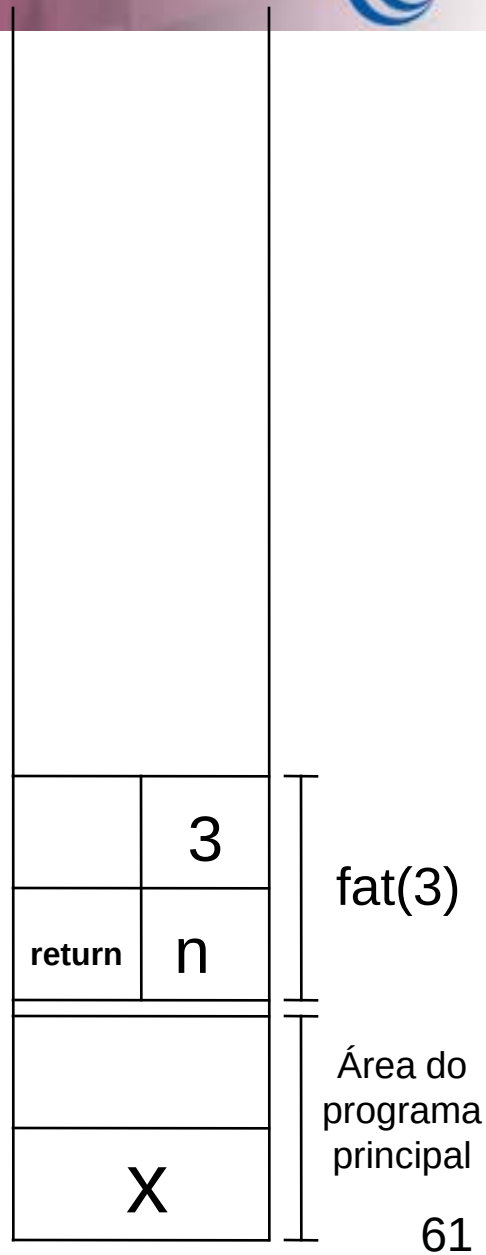
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



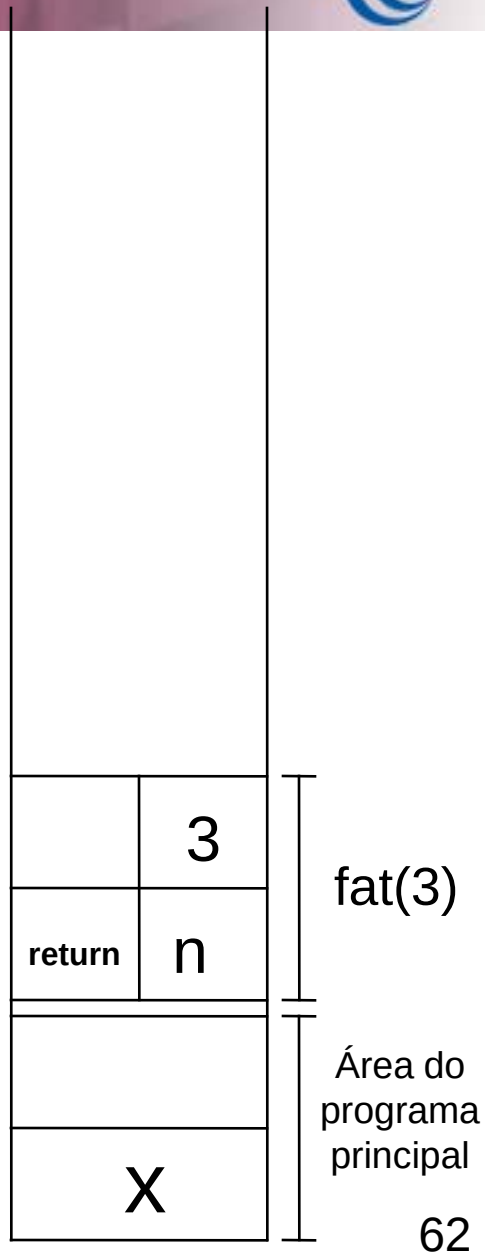
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



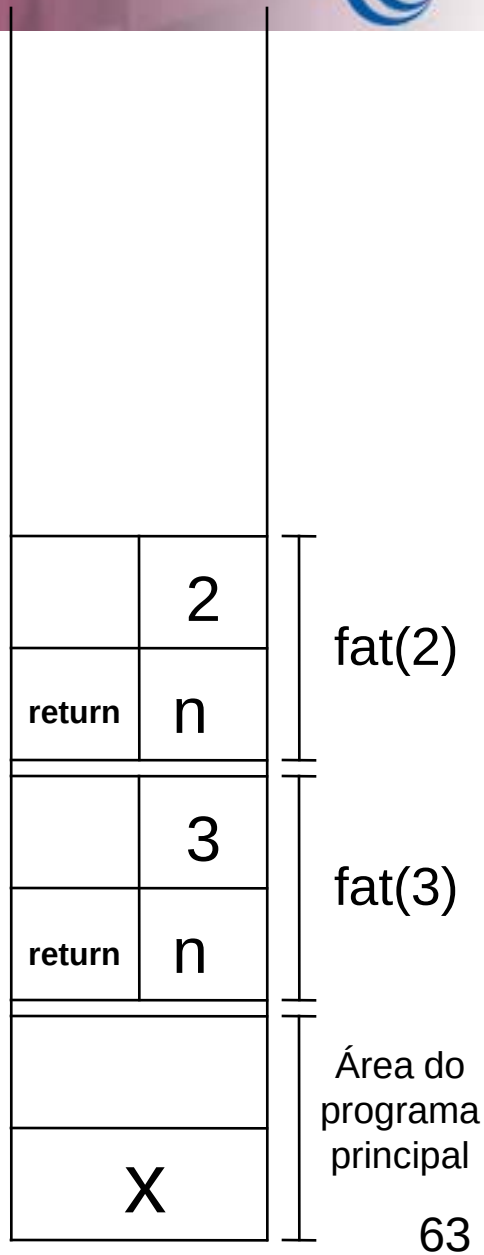
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



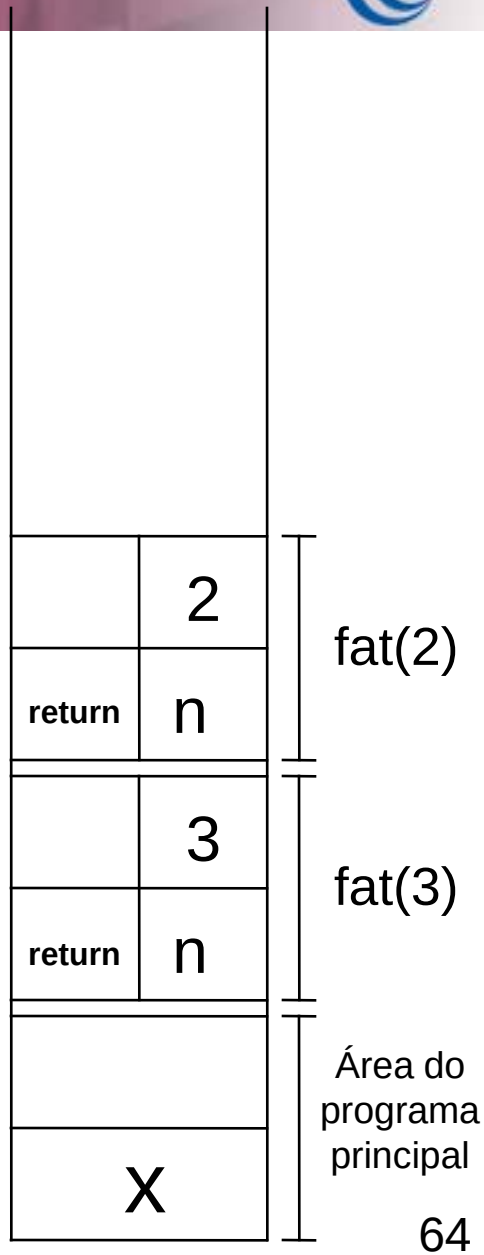
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



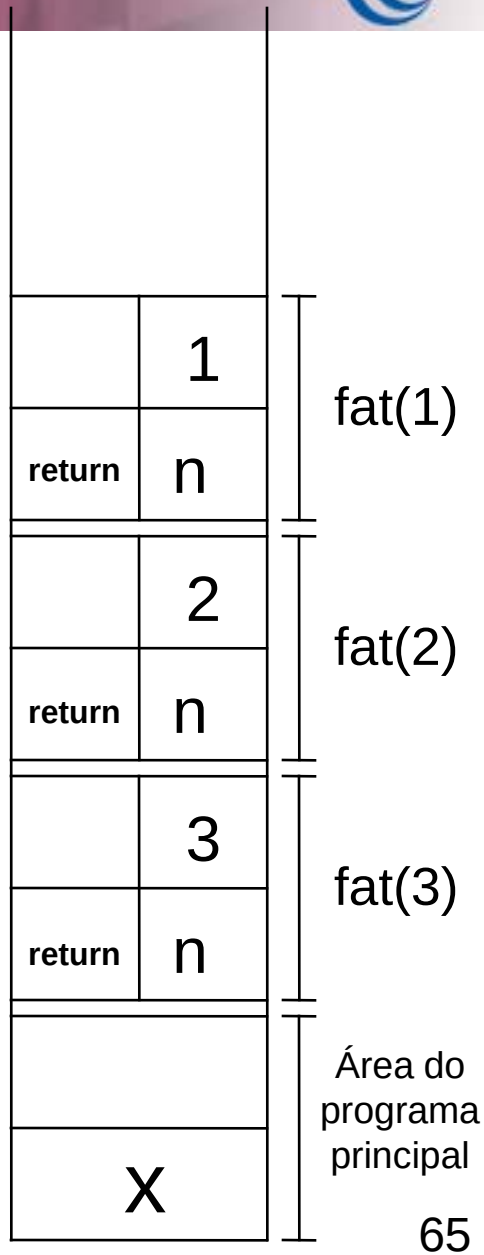
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



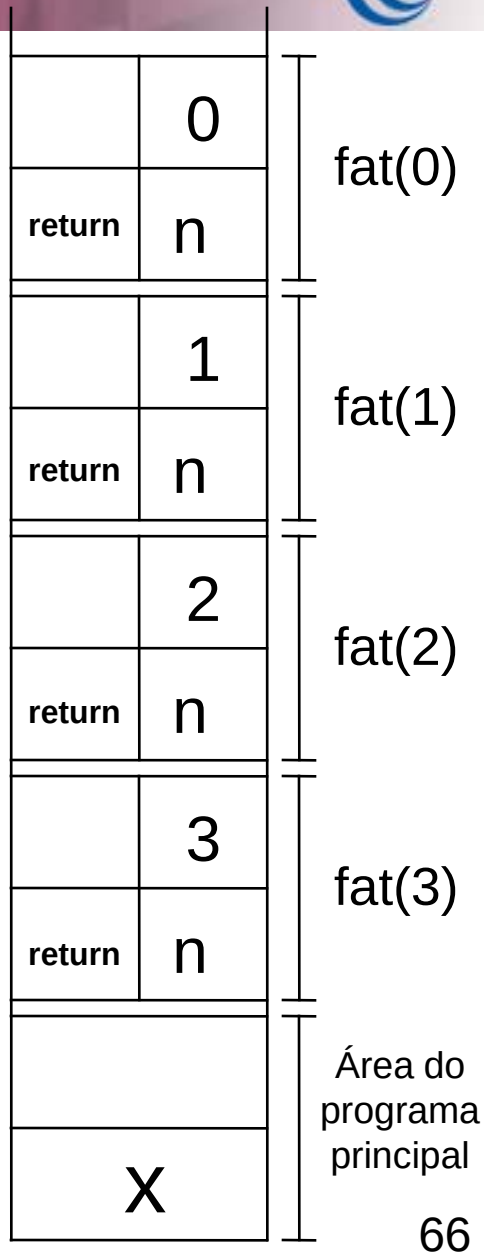
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



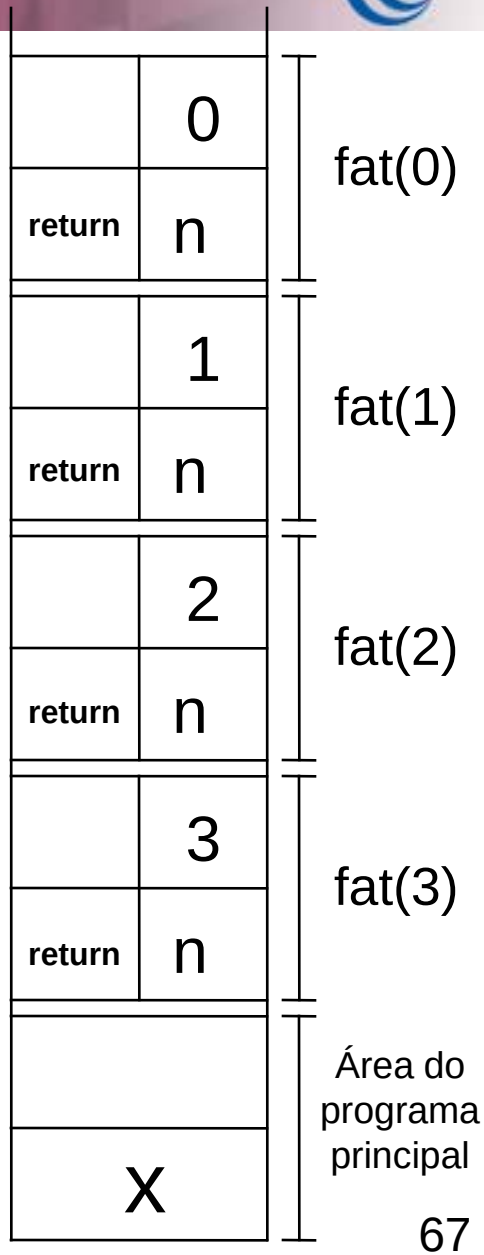
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



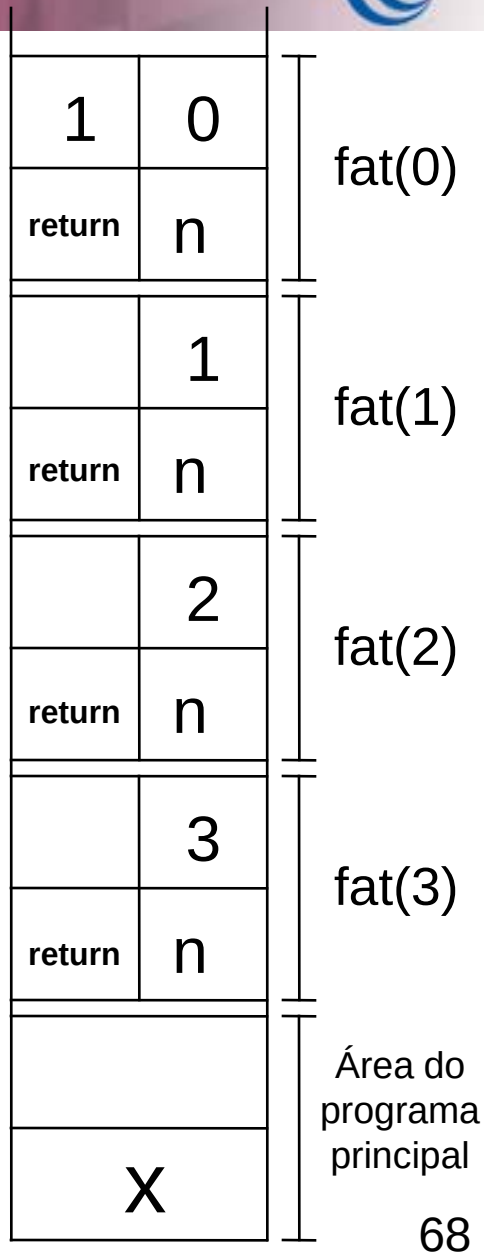
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



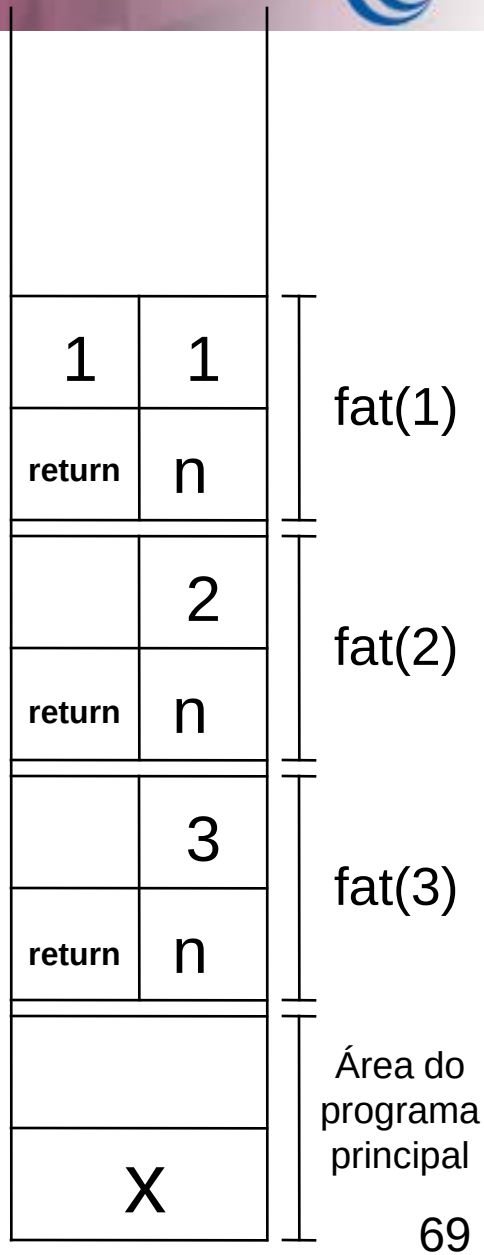
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



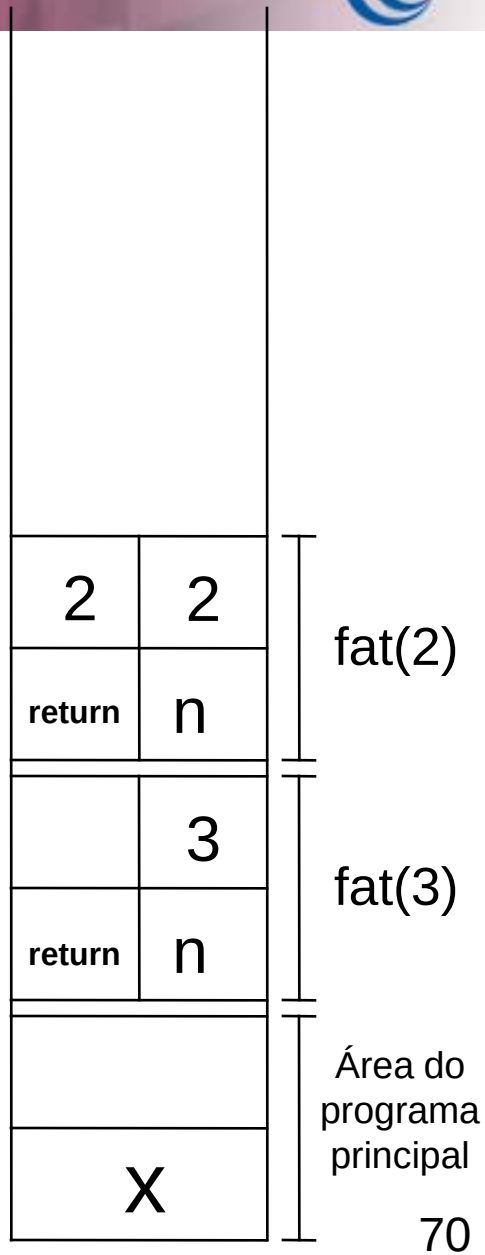
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



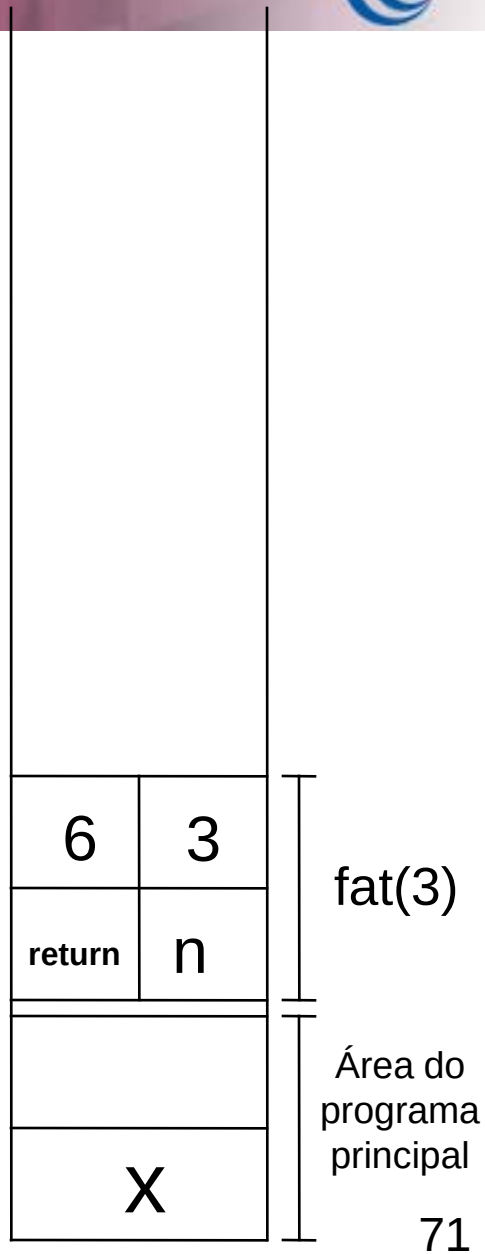
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



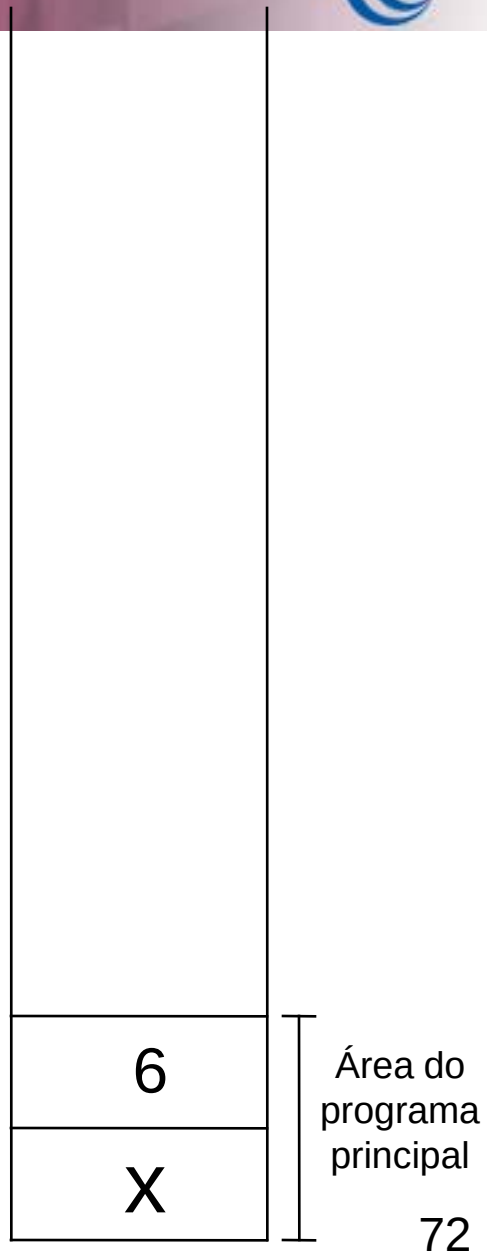
Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



Recursividade

```
# Programa Completo

# Subprograma

def fat(n):
    if n == 0:
        return 1
    else:
        return n*fat(n-1)

# Programa Principal
x = fat(3)
print(x)
```



Recursividade

- Trata-se, também, de uma forma de repetição da execução de um determinado trecho de código.

- Apesar de nenhum comando explícito de repetição ter sido utilizado, na prática o código anterior executa um produto de *N* termos.

- A função fatorial poderia também ser definida por:

$$n! = \begin{cases} 1 & , \text{se } n = 0; \\ 1 * 2 * \ldots * n & , \text{se } n > 0. \end{cases}$$

Implementação Iterativa para o Fatorial

$$n! = \begin{cases} 1 & , \text{se } n = 0; \\ 1 * 2 * \ldots * n & , \text{se } n > 0. \end{cases}$$

```
def fat(n):
    p = 1
    for ind in range(1,n+1):
        p = p * ind
    return p
```

Recursividade Mútua

- Em alguns casos, pode ser necessário que dois subprogramas se chamem reciprocamente (recursividade mútua).

```
# Programa Completo

# Subprogramas

def flip(n):

    print("Flip")

    if n>0:

        flop(n-1)
def flop(n):
    print("Flop")
    if n>0:
        flip(n-1)
# Programa Principal
flip(5)
```

Recursividade Mútua

- Em alguns casos, pode ser necessário que dois subprogramas se chamem reciprocamente (recursividade mútua).

```
# Programa Completo

# Subprogramas

def flip(n):

    print("Flip")

    if n>0:

        flop(n-1)
def flop(n):
    print("Flop")
    if n>0:
        flip(n-1)
# Programa Principal
flip(5)
```

Saída:

Flip

Flop

Flip

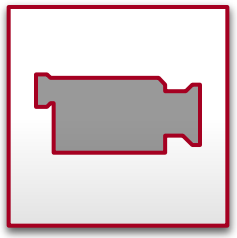
Flop

Flip

Flop

Exemplos de Aplicação dos Conteúdos Vistos

Clique no botão para assistir ao tutorial:



Faça os Exercícios Relacionados a essa Aula

Clique no botão para visualizar os enunciados:

Aula 4

Professores:

Dante Corbucci Filho
Leandro A. F. Fernandes

Conteúdo Apresentado:

- Subprogramação:
  - Funções
  - Passagem de Parâmetros
    Passagem por Valor,
    Passagem de Função
  - Recursividade