**FACULDADE DE COMPUTAÇÃO E INFORMÁTICA**

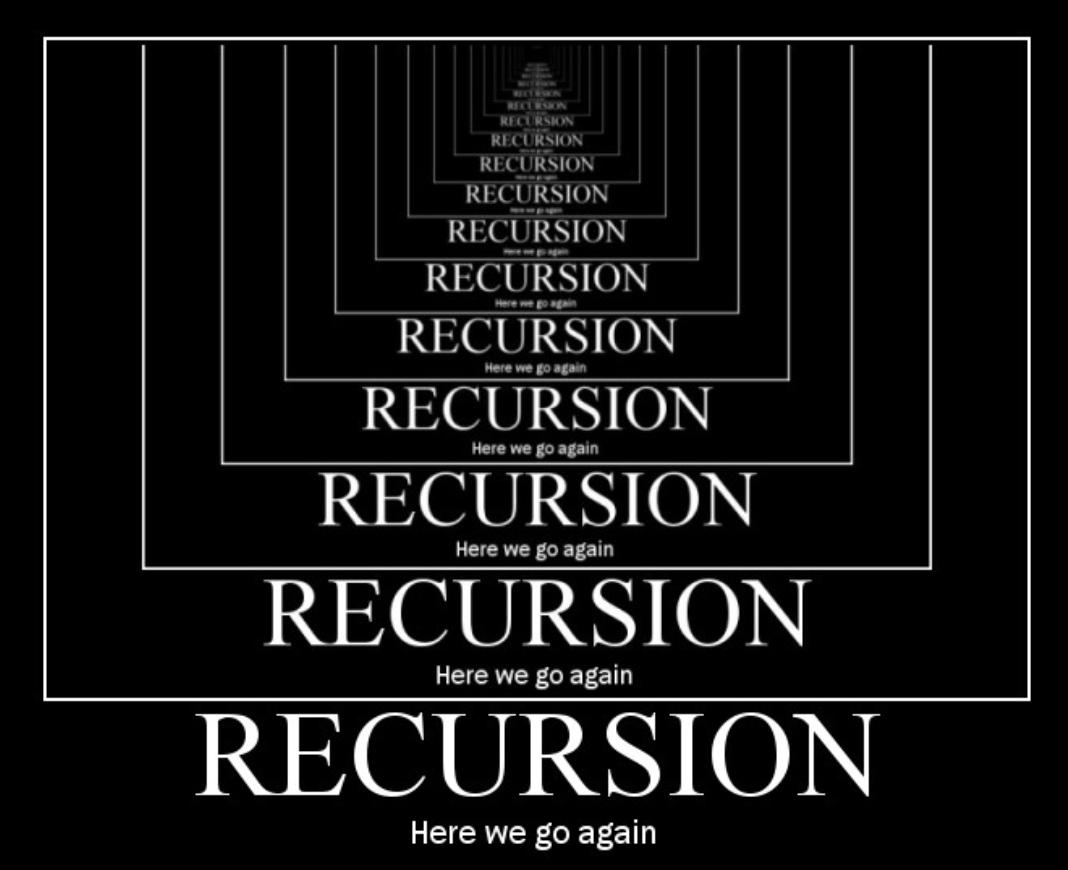
**CIÊNCIA DA COMPUTAÇÃO E SISTEMAS DE INFORMAÇÃO**

**Linguagem de Programação I**

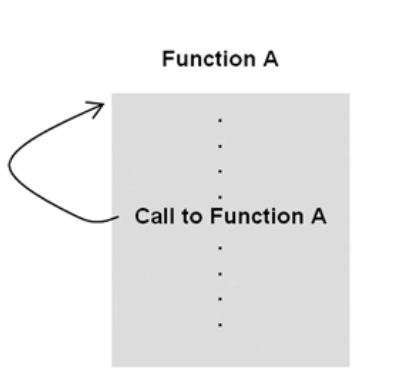
**AULA 03: Recursão em Python**

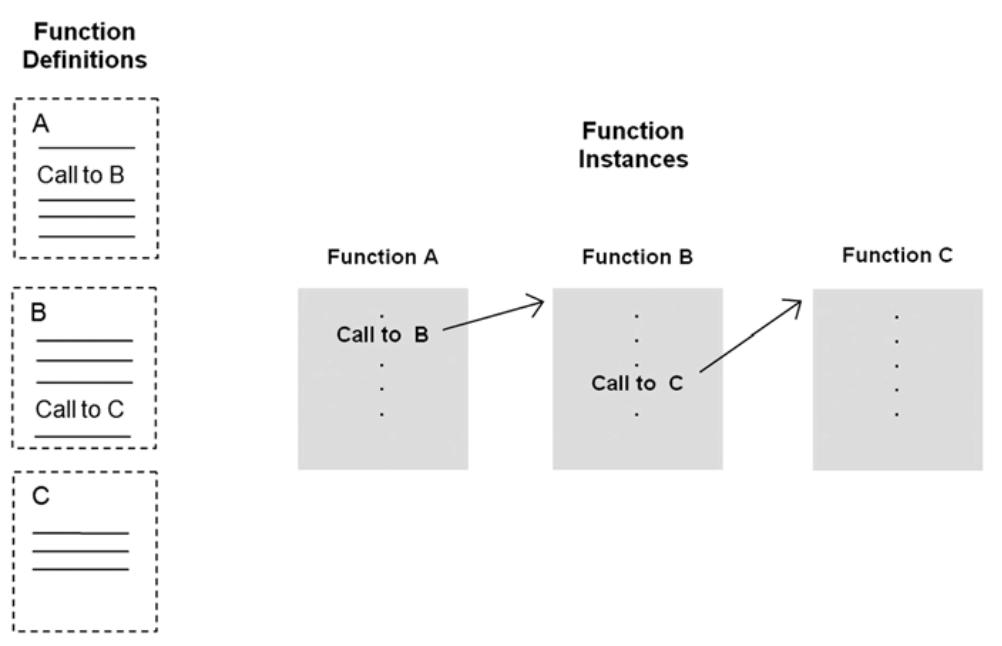
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | http://images.flatworldknowledge.com/ketchen/ketchen-fig05_x001.jpg | Nossos **objetivos** nesta aula são:   * Definir um função recursiva; * Explicar o efeito instâncias de execução e o overflow de pilha de execução; * Escrever funções recursivas; * Uso de funções recursivas versus funções iterativas; * Escrever programas em Python usando funções recursivas definidas pelo programador. | | Macintosh HD:Users:anacris:Desktop:Captura de Tela 2017-02-19 às 18.57.35.png | A referência para esta aula está no **Capítulo 11 (Recursive Functions)** do livro:  DIERBACH, C. *Introduction to Computer Science Using Python: A Computational Problem Solving Focus.* 1st Edition, New York: Wiley, 2012. | |
|  |

**Introdução**

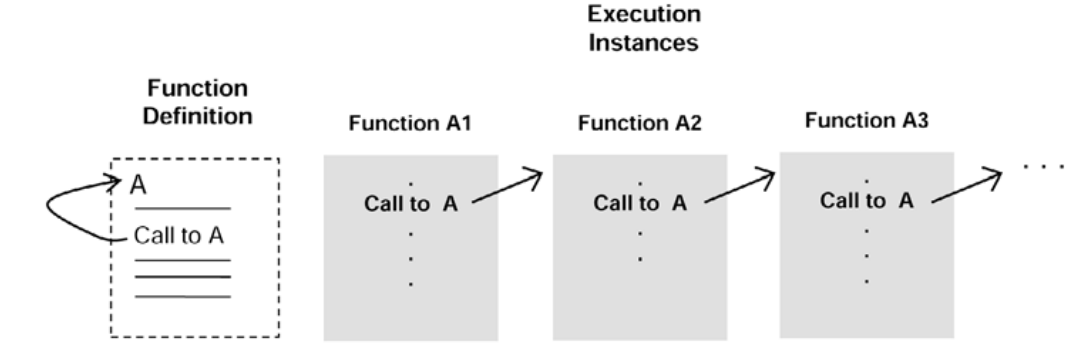
* A maioria dos problemas em computação envolve a repetição de passos. As declarações de controle iterativo, tais como for e while, é uma forma de controlar a execução repetida de instruções.
* Outro caminho para a solução desses problemas é por meio de recursão.
* Na solução de problemas recursivos, um problema é repetidamente quebrado em subproblemas até que o subproblema possa ser resolvido diretamente.

**Função Recursiva**

* Uma **função recursiva** é definida como uma função que condicionalmente chama a si mesmo.
* Na figura ao lado, temos uma função chamada A que é definida em algum ponto para chamar a função A (ela própria).
* Existe dois tipos de entidades relacionadas com qualquer função: a **definição da função** e quaisquer **instâncias de execução**. Uma função que chama a si mesmo é uma função instância de execução que chama outra instância de execução da mesma função.
* Nas chamadas de funções não-recursivas, não existe muita dúvida de como ocorre a sequência de chamadas. Na figura abaixo, primeiro uma instância de execução a partir da definição da função A é criada e inicia sua execução. Quando a chamada para a função B é encontrada, a instância de execução da função A é suspensa enquanto uma instância de execução da função B é criada e inicia sua execução. Quando a chamada da função C é encontrada, a função B suspense sua execução enquanto uma instância da função C é criada e inicia sua execução.



* Como a função C não chama qualquer outra função, ao término de sua execução o controle retorna para a função que a chamou , a função B. A função B continua sua execução e o controle retorna para quem a chamou, a função A. Finalmente, a função A completa sua execução e termina, retornando o controle para quem a chamou.
* Vamos analisar outra situação em que a função A seja recursiva. Cada instância de execução corrente da função A gerará uma nova instância de execução da função A.

****

* A execução de uma série de instâncias da função recursiva é similar a execução de uma série de instâncias não-recursivas, exceto que as instâncias são clones uma da outra. Assim, todas as instâncias são idênticas, a chamada da função ocorre exatamente no mesmo lugar.
* Por outro lado, toda função recursiva deve contemplar, no processo de diminuir sucessivamente o problema em um problema menor, uma chamada de instância de execução que permita resolver o problema de forma direta, sem recorrer a si mesmo.
* Quando isso ocorre, diz-se que o algoritmo atingiu uma **condição de parada ou o seu caso base**, a qual deve estar presente em pelo menos um local dentro da função. Sem esta condição o algoritmo não pára de chamar a si mesmo, até estourar a capacidade da pilha, o que geralmente causa efeitos colaterais e até mesmo o término indesejável do programa.

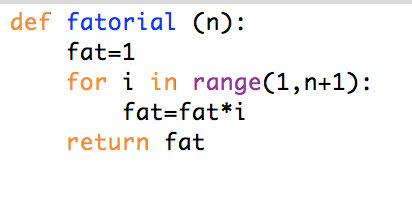
**EXERCÍCIO TUTORIADO 1**

Usando o Shell do Python, o que será exibido ao digitarmos as seguintes linhas de comando?

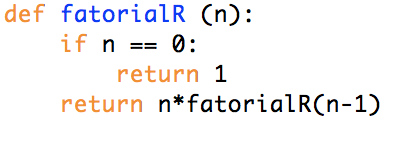
|  |  |
| --- | --- |
| >>> def rfunc(n):  print(n)  if n > 0:  rfunc(n-1)  (a)  >>> rfunc(4) | >>> def rfunc(n):  if n == 1:  return 1  else:  return n + rfunc(n-1)  (c)  >>> rfunc(1) |
| (b)  >>> rfunc(0) | (d)  >>> rfunc(3) |
| (c)  >>> rfunc(100) | (e)  >>> rfunc(100) |

**A Função Fatorial**

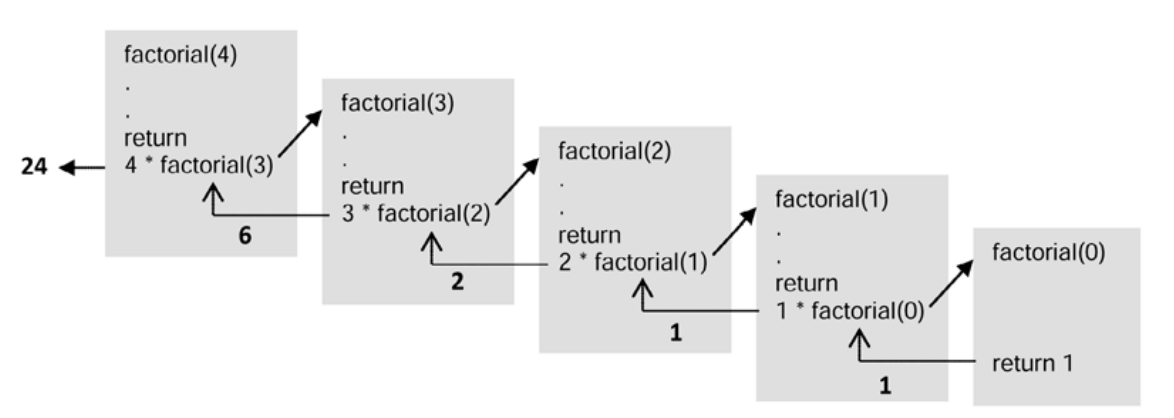
* Mecanismos de recursão são **adequados** para resolução de **problemas que já possuem uma estrutura recursiva**. Por exemplo, vamos considerar o problema de se calcular o fatorial de um número natural . Se considerarmos a definição de fatorial como:
* Não vislumbramos diretamente uma estrutura recursiva para o cálculo do fatorial. E, conforme estudado em Introdução a Programação, o fatorial como definido acima poderia ser implementado como a seguinte função:



* Este tipo de implementação é denominada **iterativa**, uma vez que ocorre uma repetição (iteração) pelo laço for.
* Porém, se considerarmos a definição de fatorial como:
* A estrutura recursiva do cálculo do fatorial fica evidente. A definição do fatorial é definida claramente em termos de si própria, referenciada como definição recursiva, a parte da definição para a n = 0 é o caso base.
* Assim, teremos a seguinte implementação recursiva:

****

* Se compararmos as implementações iterativa e recursiva do fatorial, não é difícil nos convencermos de que a **recursiva** é **mais simples**.
* E o que acontece numa chamada recursiva ? Abaixo, temos um esquema de uma chamada para calcular o fatorial do número 4:

****

* Cada instância de execução da função fatorial é suspensa enquanto avaliação da expressão n\*fatorial(n-1) é completada. Quando finalmente a chamada com o valor zero, existe 4 instâncias de execução e ficam suspensas até que a instância fatorial(0) complete.
* Quando fatorial(0) retorna o valor 1, a avaliação da expressão 1\*fatorial(0) pode ser completada retornando 1 como o valoro do fatorial\*(1) e assim sucessivamente, até 4\*fatorial(3) ser avaliado e 24 ser retornado para a chamada original.
* Embora a implementação seja simples, o custo de uma execução recursiva pode ser alto. Cada chamada recursiva irá empilhar dados na pilha de execução do programa (**Stack**) e, caso esta pilha não dimensionada corretamente, podemos ter um “estouro de pilha”. Além disto, o processo de empilhamento pode ter um impacto significativo no desempenho do programa. Veja a pilha de recursão para n = 4, no Python Tutor.
* Assim, a decisão entre usar uma implementação iterativa ou uma recursiva deve privilegiar, principalmente, quanto de memória temos para usar.

|  |  |
| --- | --- |
| **Macintosh HD:Users:anacris:Desktop:Captura de Tela 2017-02-19 às 21.58.11.png** | **Macintosh HD:Users:anacris:Desktop:Captura de Tela 2017-02-19 às 22.00.20.png** |

**EXERCÍCIO TUTORIADO 2**

Usando o Shell do Python, o que será exibido ao digitarmos as seguintes linhas de comando?

|  |  |
| --- | --- |
| >>>  **Macintosh HD:Users:anacris:Desktop:Captura de Tela 2017-02-19 às 22.16.13.png**  (a)  >>> fatorialR(4) | >>>  Macintosh HD:Users:anacris:Desktop:Captura de Tela 2017-02-19 às 21.45.19.png  (e)  >>> fatorial(4) |
| (b)  >>> fatorialR(0) | (f)  >>> fatorial(0) |
| (c)  >>> fatorialR(100) | (g)  >>> fatorial(100) |
| (d)  >>> fatorialR(1000) | (h)  >>> fatorial(1000) |

**RECURSÃO VERSUS ITERAÇÃO**

* Recursão é fundamentalmente um meio de executar repetidamente um conjunto de instruções. O conjunto de instruções são os de uma função, e a repetição vem do fato de que a função é repetidamente executada, uma vez para cada chamada de função recursiva. Assim, recursão e iteração são dois meios de realizar o mesmo resultado. O que pode ser calculado usando recursão também pode ser calculado usando iteração e vice-versa.
* Então, uma pergunta natural é "Quando devo usar recursão, e quando devo usar iteração?" Não há uma resposta clara a esta pergunta, mas existem algumas diretrizes.
* Geralmente, uma função recursiva leva mais tempo para executar do que uma abordagem iterativa equivalente. Isso ocorre porque as múltiplas chamadas de função demoram. Em contraste, while e para loops executar muito eficientemente. Assim, quando um problema pode ser resolvido de forma recursiva e iterativa com um esforço de programação semelhante, geralmente é melhor usar uma abordagem iterativa.
* Lembre-se que a função fatorial foi um exemplo de tal situação. A versão iterativa da computação fatorial é simples de implementar, e executa muito mais eficientemente.
* Por outro lado, alguns problemas são muito difíceis de resolver iterativamente, e quase trivial para resolver recursivamente.

**EXERCÍCIO COM DISCUSSÃO EM DUPLAS**

Os números de Fibonacci correspondem à seguinte sequência de números naturais:

1 1 2 3 5 8 13 ...

A regra de formação desta sequência é muito simples: exceto para os dois números iniciais (1 e 1), todos os outros números são a soma dos números imediatamente anteriores na sequencia.

(a) Proponha uma regra recursiva para definir o n-ésimo número de Fibonacci.

(b) Implemente uma função recursiva que calcule o n-ésimo número de Fibonacci e a teste dentro do programa.

(c) Faça um diagrama de execução das chamadas recursivas de sua implementação para calcular o 5-ésimo número de Fibonacci.

**EXERCÍCIOS DE LABORATÓRIO**

1. Implemente uma função recursiva para calcular a **potência** an, supondo que tanto a quanto n sejam números inteiros positivos.
2. Implemente uma função recursiva para calcular a **soma dos dígitos** de um número inteiro e positivo.
3. Implemente uma função recursiva para verificar se um determinado número natural, maior ou igual a 2, é **primo**.

**EXERCÍCIOS EXTRAS**

1. Qual o valor que a função recursiva seguinte retorna, dado um argumento n inteiro positivo qualquer?

def rfunc(n):

if n == 0:

return 0

else:

return rfunc(n - 1) + 2

1. Para cada uma das funções recursivas seguintes, indique qual dos requisitos é violado no projeto do algoritmo recursivo.

(a) def rfunc1(n):

return n 1 rfunc(n - 1)

(b) def rfunc2(n):

if n == 0:

return 1

return n + rfunc2(n + 1)

1. Implemente uma função recursiva para calcular o **máximo divisor comum** MDC(a,b) de dois números naturais.
2. Implemente uma função recursiva para calcular o **mínimo múltiplo comum** MMC (a,b) de dois números naturais.
3. Implemente uma função recursiva para calcular o **resto da divisão** inteira de X por Y, supondo X e Y números inteiros.
4. Implemente uma função recursiva para calcular o **quociente da divisão** inteira de X por Y, supondo X e Y números inteiros.
5. Implemente uma função iterativa e outra função recursiva que receba um número inteiro positivo na **base decimal** e o converta para a **base binária**.
6. Implemente uma função iterativa e outra função recursiva que receba um número inteiro positivo na **base binária** e o converta para a **base decimal**.
7. Faça a pilha de recursão para a seguinte função recursiva com N = 6:

**Função inteiro zzz (inteiro n)**

**Início**

SE ( n <= 2 )

RETORNA ( 1 )

FIM-SE

n ← n – 1

aux ← zzz( n )

n ← n – 1

RETORNA ( aux + zzz( n ) )

**Fim**

1. Faça a pilha de recursão para a função recursiva com N = 7:

**Função inteiro fusc(inteiro n)**

**Início**

SE ( n <= 1 )

RETORNA ( 1 )

FIM-SE

SE ( n % 2 = 0 )

RETORNA ( fusc( n//2 ) )

FIM-SE

RETORNA ( fusc((n-1)//2 ) + fusc((n+1)//2 ) )

**Fim**

1. Qual o valor de f(1, 10)? Como se poderia calcular f( x, y) de maneira mais simples ?

**Função real f ( real x, y)**

**Início**

SE ( x >= y )

RETORNA ( (x + y)//2 )

SENÃO

RETORNA ( f( f( x+2 , y – 1), f( x+1, y – 2)))

FIM SE

**Fim**