Algoritmos e Lógica de Programação

80 horas // 4 h/semana

Recursão e Módulos em Python

Aula 13

Prof. Piva

Para começar...

Nós vimos na última aula, o processo de criação de funções.

A área de programação funcional tem muitos outros detalhes e aplicações.

Uma dessas vertentes é a recursão.

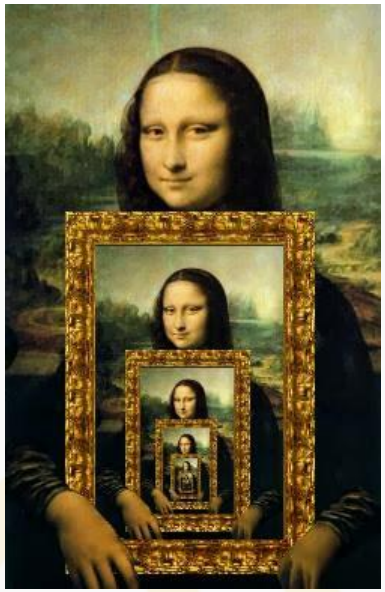
Trataremos disso e também exploraremos com mais detalhes o processo de trabalhar com módulos ou pacotes em Python.

O que é recursão...

Recursividade...

Seja um Profissional Python !

- Propriedade de uma função chamar a si mesma.
- Necessariamente, toda função recursiva deve ter uma ou mais condições de parada.



Exemplo clássico:
  - Cálculo FATORIAL

Fatorial de 5
5 * 4 * 3 * 2 * 1 = 120

Fatorial de 0 = 1

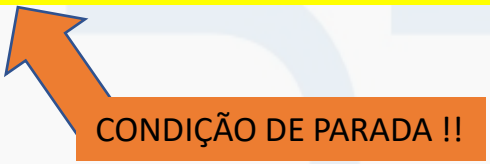
Fatorial de 1 = 1

Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```



Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

N = 5

N *

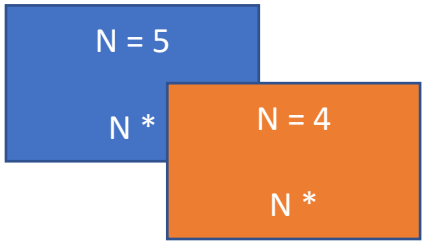
```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!

Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**



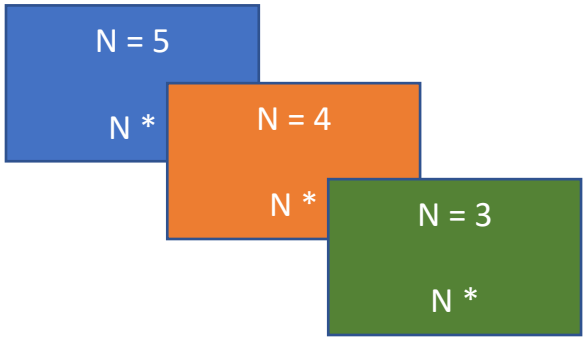

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!

Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**




```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!

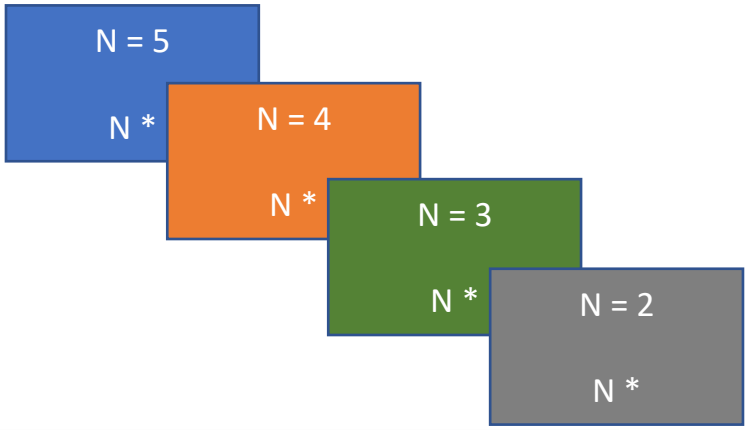
Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!



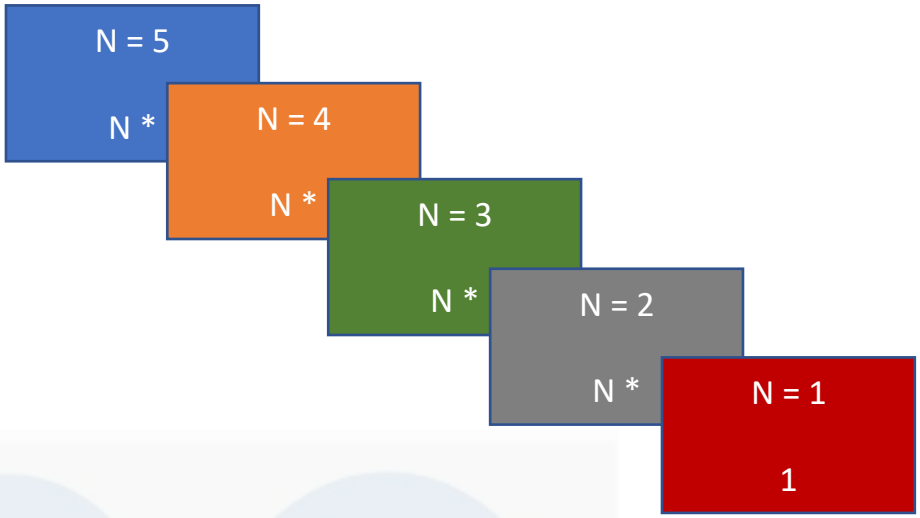
Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!



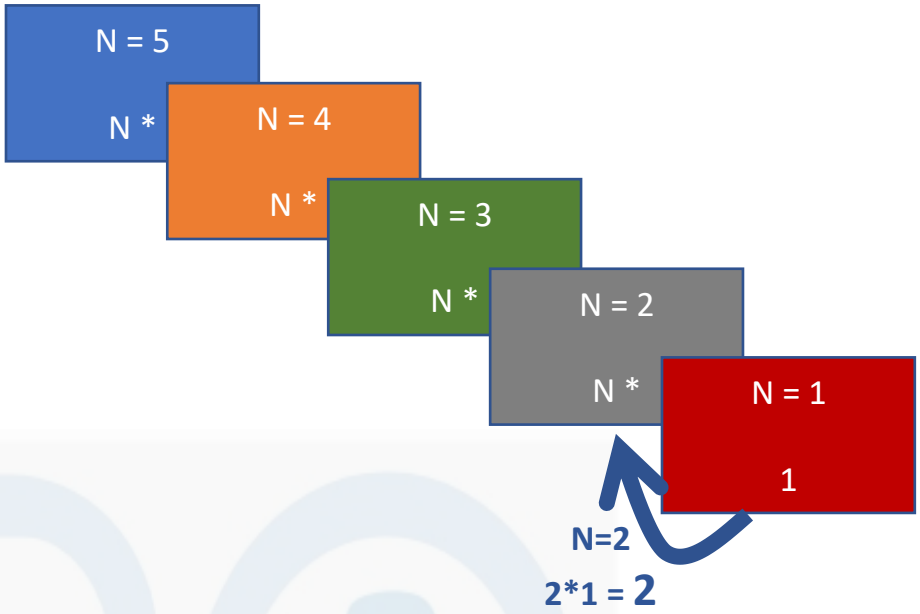
Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!



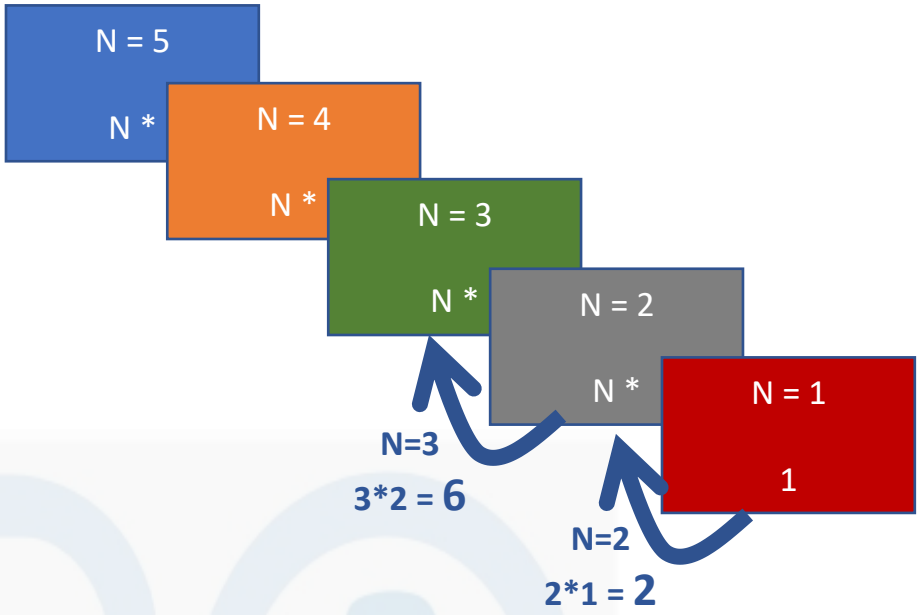
Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!



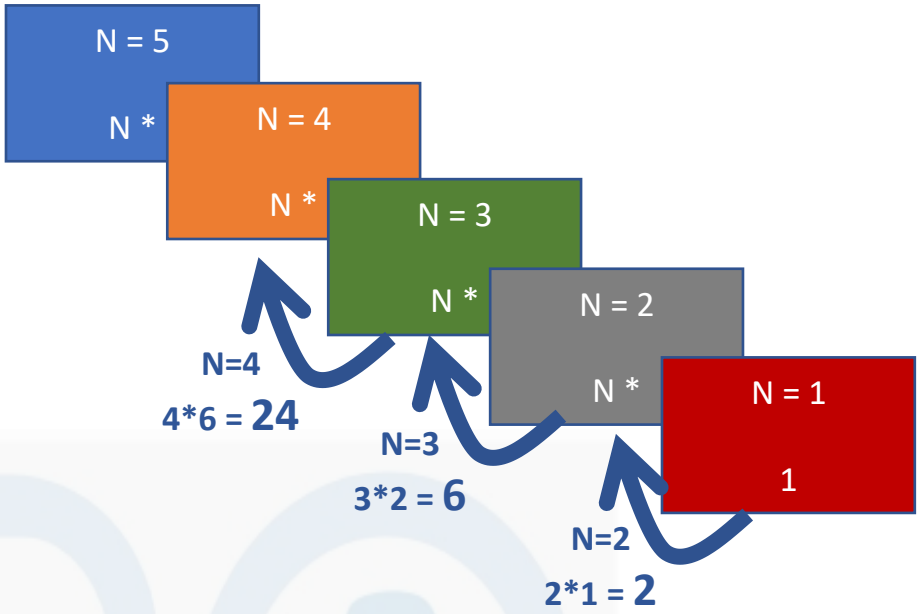
Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!



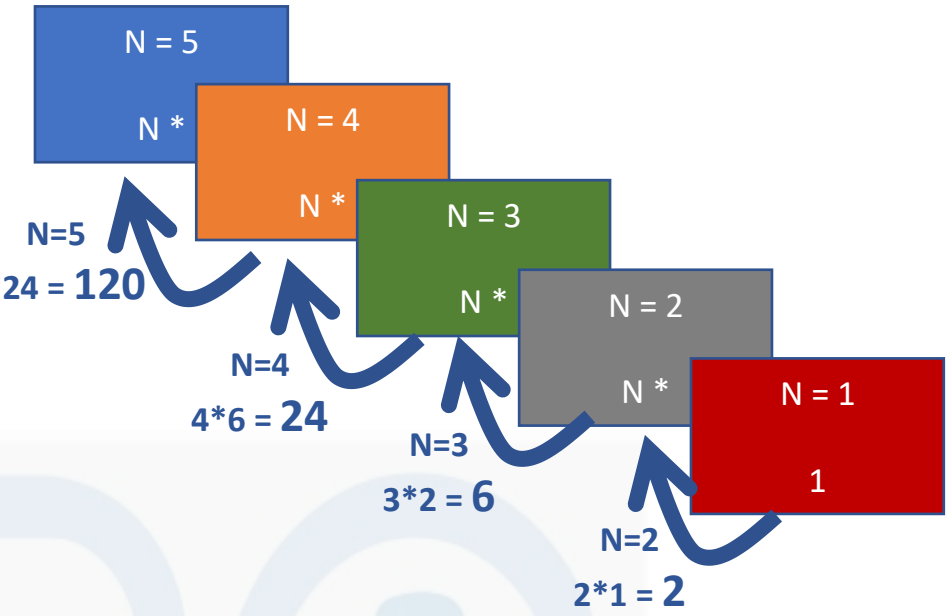
Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

CONDIÇÃO DE PARADA !!



Recursividade

Seja um Profissional Python !

- **Cálculo Fatorial**

```
def fatorial(n):
    if n > 1:
        return n * fatorial(n-1)
    else:
        return 1
```

```
def fatorial(n):
    return (n * fatorial(n-1)) if n > 1 else 1
```

VAMOS PARA A PRÁTICA ?!!!

Python: Recursão

Recursão

- É um princípio muito poderoso para construção de algoritmos
- A solução de um problema é dividido em
  - Casos simples:
    - São aqueles que podem ser resolvidos trivialmente
  - Casos gerais:
    - São aqueles que podem ser resolvidos compondo soluções de casos mais simples
- Semelhante à prova de teoremas por indução
  - Casos simples: O teorema é verdadeiro trivialmente
  - Casos genéricos: são provados assumindo-se que todos os casos mais simples também são verdadeiros

Função recursiva

- Implementa um algoritmos recursivo onde a solução dos casos genéricos requerem chamadas à própria função
- Uma função recursiva é a maneira mais direta (mas não necessariamente a melhor) de se resolver problemas de natureza recursiva ou para implementar estruturas de dados recursivas
- Considere, por exemplo, a definição da sequência de Fibonacci:
  - O primeiro e o segundo termo valem 0 e 1, respectivamente
  - O *i*-ésimo termo é a soma do (*i*-1)-ésimo e o (*i*-2)-ésimo termo

```
>>> def fib(i):
        if i==1: return
        elif i==2: return
        else: return fib(i-1)+fib(i-2)
>>> for i in range(1,11):
        print (fib(i), end= " ")
0 1 1 2 3 5 8 13 21 34
```

Exemplo: Busca binária

- Um exemplo clássico de recursão é o algoritmo conhecido como busca binária que é usado para pesquisar um *valor* em uma *lista* ordenada
- Chamemos de imin e imax os índices mínimo e máximo da lista onde a busca será feita
  - Inicialmente, imin = 0 e imax = len(*lista*)-1
- O caso base corresponde a imin == imax
  - Então, ou o *valor* é igual a *lista* [imin] ou não está na lista
- Senão, podemos dividir o intervalo de busca em dois
  - Seja meio = (imin+imax)/2
  - Se o *valor* é maior que *lista* [meio] , então ele se encontra em algum dos índices entre meio+1 e imax
  - Caso contrário, deve se encontrar em algum dos índices entre imin e meio

Busca binária: implementação

```
def testa(lista,valor):
    def busca_binaria(imin,imax):
        if imin==imax: return imin
        else:
            meio=(imax+imin)/2
            if valor>lista[meio]:
                return busca_binaria(meio+1,imax)
            else:
                return busca_binaria(imin,meio)
    i = busca_binaria(0,len(lista)-1)
    if lista[i]==valor:
        print (valor,"encontrado na posicao",i)
    else:
        print (valor,"nao encontrado”)

>>> testa([1,2,5,6,9,12],3)
3 nao encontrado
>>> testa([1,2,5,6,9,12],5)
5 encontrado na posicao 2
```

Recursão infinita

- Assim como nos casos dos laços de repetição, é preciso cuidado para não escrever funções infinitamente recursivas
  - Ex.:

```
def recursiva(x):
    if f(x): return True
    else: return recursiva(x)
```

- Uma função recursiva tem que
  - Tratar *todos* os casos básicos
  - Usar recursão apenas para tratar casos *garantidamente mais simples* do que o caso corrente
  - Ex.:

```
def recursiva(x):
    if f(x): return True
    elif x==0: return False
    else: return recursiva(x-1)
```

Eficiência de funções recursivas

- Quando uma função é chamada, um pouco de memória é usado para guardar o ponto de retorno, os argumentos e variáveis locais
- Assim, soluções iterativas são normalmente mais eficientes do que soluções recursivas equivalentes
- Isto não quer dizer que soluções iterativas sempre sejam preferíveis a soluções recursivas
- Se o problema é recursivo por natureza, uma solução recursiva é mais clara, mais fácil de programar e, frequentemente, mais eficiente

Pensando recursivamente

- Ao invés de pensar construtivamente para para obter uma solução, às vezes é mais simples pensar em termos de uma prova indutiva
- Considere o problema de testar se uma lista *a* é uma permutação da lista *b*
  - Caso básico: *a* é uma lista vazia
    - Então *a* é permutação de *b* se *b* também é uma lista vazia
  - Caso básico: *a*[0] não aparece em *b*
    - Então *a* não é uma permutação de *b*
  - Caso genérico: *a*[0] aparece em *b* na posição *i*
    - Então *a* é permutação de *b* se *a*[1:] é uma permutação de *b* do qual foi removido o elemento na posição *i*

Exemplo: Testa permutações

```
def e_permutacao(a,b):
    """
    Retorna True sse a lista a é uma
    permutação da lista b
    """
    if len(a) == 0 : return len(b)==0
    if a[0] in b:
        i = b.index(a[0])
        return e_permutacao(a[1:],b[0:i]+b[i+1:])
    return False
```

```
>>> e_permutacao([1,2,3],[3,2,1])
True
>>> e_permutacao([1,2,3],[3,3,1])
False
>>> e_permutacao([1,2,3],[1,1,2,3])
False
>>> e_permutacao([1,1,2,3],[1,2,3])
False
```

Estruturas de dados recursivas

- Há estruturas de dados que são inerentemente recursivas, já que sua própria definição é recursiva
- Por exemplo, uma lista pode ser definida recursivamente:
  - [] é uma lista (vazia)
  - Se *A* é uma lista e *x* é um valor, então *A*+[*x*] é uma lista com *x* como seu último elemento
- Esta é uma definição construtiva, que pode ser usada para escrever funções que criam listas
- Uma outra definição que pode ser usada para analisar listas é:
  - Se *L* é uma lista, então:
    - *L* == [] , ou seja, *L* é uma lista vazia, ou
    - *x* = *L*.pop() torna *L* uma lista sem seu último elemento *x*
  - Esta definição não é tão útil em Python já que o comando for permite iterar facilmente sobre os elementos da lista

Exemplo: Subsequência

```
def e_subseq(a,b):
    """ Retorna True sse a é subsequência de b,
    isto é, se todos os elementos a[0..n-1] de a
    aparecem em b[j(0)], b[j(1)]... b[j(n-1)]
        onde j(i)<j(i+1) """
    if a == []:
        # Lista vazia é subsequência de qq lista
        return True
    if a[0] not in b:
        return False
    return e_subseq (a[1:], b[b.index(a[0])+1:])
```

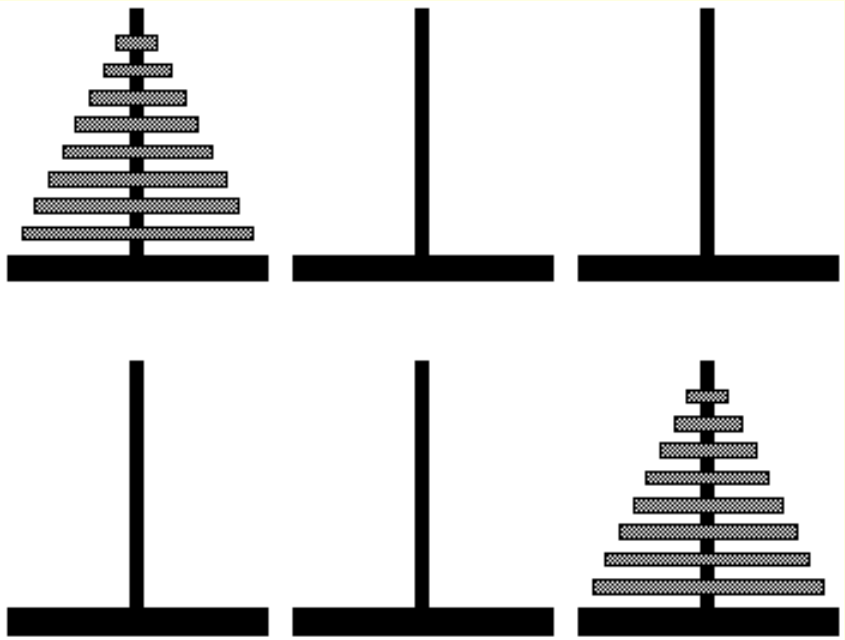
Encontrando a recorrência

- Alguns problemas não se apresentam naturalmente como recursivos, mas pensar recursivamente provê a solução
- Tome o problema de computar todas as permutações de uma lista
  - Assumamos que sabemos computar todas as permutações de uma lista sem seu primeiro elemento *x*
    - Seja perm uma dessas permutações
    - Então, a solução do global contém todas as listas obtidas inserindo x em todas as possíveis posições de perm

Exemplo: computar todas as permutações de uma lista

```
def permutacoes(lista):
    """ Dada uma lista, retorna uma lista de listas, onde cada elemento é uma
      permutação da lista original """
    if len(lista) == 1:  # Caso base
        return [lista]
    primeiro = lista[0]
    resto = lista [1:]
    resultado = []
    for perm in permutacoes(resto):
        for i in range(len(perm)+1):
            resultado += \
                [perm[:i]+[primeiro]+perm[i:]]
    return resultado
```

Torres de Hanói



- Jogo que é um exemplo clássico de problema recursivo
- Consiste de um tabuleiro com 3 pinos no qual são encaixados discos de tamanho decrescente
- A ideia é mover os discos de um pino para outro sendo que:
  - Só um disco é movimentado por vez
  - Um disco maior nunca pode ser posto sobre um menor

Torres de Hanói: Algoritmo

- A solução é simples se supusermos existir um algoritmo capaz de mover todos os discos menos um do pino de origem para o pino sobressalente
- O algoritmo completo para mover *n* discos do pino de origem *A* para o pino de destino *B* usando o pino sobressalente *C* é
  - Se *n* é 1, então a solução é trivial
  - Caso contrário,
    - Usa-se o algoritmo para mover *n*-1 discos de *A* para *C* usando *B* como sobressalente
    - Move-se o disco restante de *A* para *B*
    - Usa-se o algoritmo para mover *n*-1 discos de *C* para *B* usando *A* como sobressalente

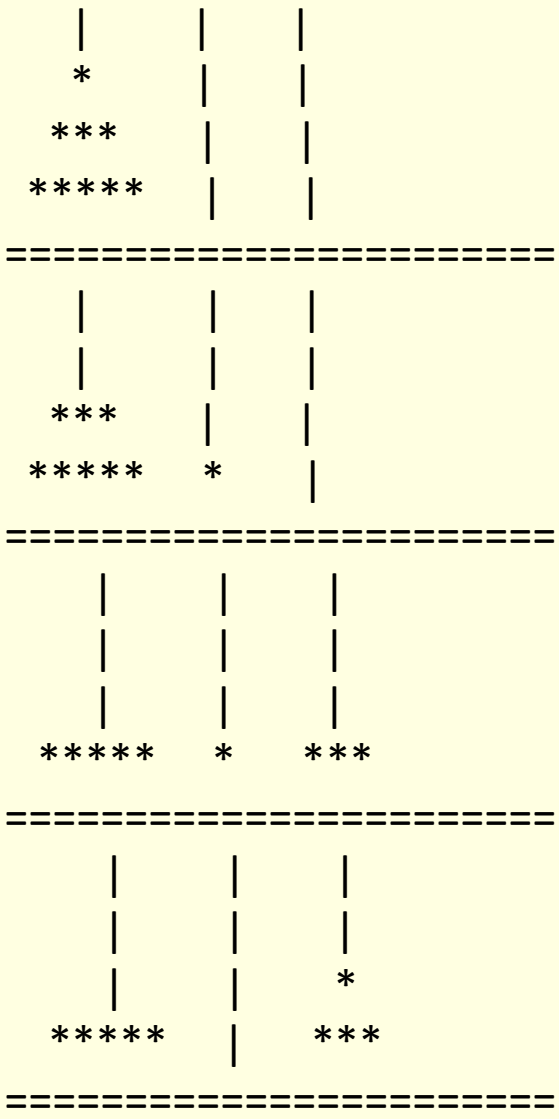
Torres de Hanói: Implementação

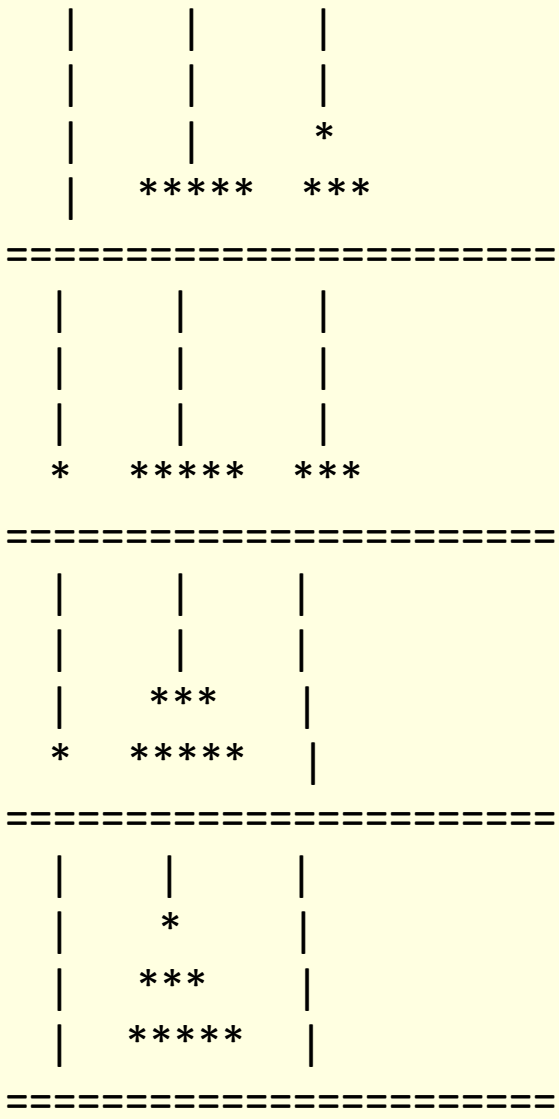
```
def hanoi(n,origem,destino,temp):
    if n>1: hanoi(n-1,origem,temp,destino)
    mover(origem,destino)
    if n>1: hanoi(n-1,temp,destino,origem)

def mover(origem,destino):
     print (“Mover de“, origem, “para”, “destino”)
```

- Com um pouco mais de trabalho, podemos redefinir a função mover para que ela nos dê uma representação “gráfica” do movimento dos discos

Torres de Hanói: Exemplo





Python:
Módulos

Módulos

- Módulos são programas feitos para serem reaproveitados em outros programas
- Eles tipicamente contêm funções, variáveis, classes e objetos que provêm alguma funcionalidade comum
- Por exemplo, já vimos que o módulo math contém funções matemáticas como sin, exp, etc, além da constante pi
- Toda a biblioteca padrão do Python é dividida em módulos e *pacotes* (veremos mais tarde)
- Alguns dos mais comuns são: sys, os, time, random, re, shelve

Escrevendo módulos

- Na verdade, qualquer programa que você escreva e salve num arquivo pode ser importado como um módulo
- Por exemplo, se você salva um programa com o nome prog.py, ele pode ser importado usando o comando import prog
  - Entretanto, a “importação” só ocorre uma vez
  - Python assume que variáveis e funções não são mudados e que o código do módulo serve meramente para inicializar esses elementos

Escrevendo módulos

- Após a importação de um módulo, este é compilado, gerando um arquivo .pyc correspondente
  - No exemplo, um arquivo prog.pyc será criado
  - Python só recompila um programa se o arquivo .py for mais recente que o arquivo .pyc

Exemplo (em Unix)

```
$ cat teste.py
def f():
   print "alo"
f()
$ python
...
>>> import teste
alo
>>> import teste
>>> teste.f()
alo
>>>
$ dir teste*
teste.py  teste.pyc
```

Tornando módulos disponíveis

- Em que diretório os módulos são buscados durante a importação?
  - No diretório corrente
  - Nos diretórios da lista sys.path
- Se for desejável especificar o local onde os módulos residem, há essencialmente duas opções
  - Alterar diretamente a variável sys.path
  - Alterar a *variável de ambiente* PYTHONPATH
    - É o método recomendável pois não requer que o programa que importará o módulo seja alterado

Exemplo

```
$ mkdir python
$ mv teste.py python/
$ cat python/teste.py
def f():
    print "alo"
f()
$ export PYTHONPATH=~/python
$ python
Python 2.4.2 (#2, Sep 30 2005, 21:19:01)
...
>>> import teste
alo
```

A variável __name__

- Se um programa pode ser executado por si só ou importado dentro de outro, como distinguir as duas situações?
  - A variável __name__ é definida para cada programa:
    - Se é um módulo, retorna o nome do módulo
    - Se é um programa sendo executado, retorna '__main__'
- Para saber se o código está sendo executado como módulo, basta testar:
  - If __name__ == '__main__': *código*
- Isto é útil em diversas circunstâncias
  - Por exemplo, para colocar código de teste, código para instalação do módulo ou exemplos de utilização

Exemplo

```
$ cat teste.py
def f():
    print "alo"
if __name__ == '__main__':
    f()
$ python teste.py
alo
$ python
Python 2.4.2 (#2, Sep 30 2005, 21:19:01)
...
>>> import teste
>>> print __name__
__main__
>>> print teste.__name__
teste
```

Pacotes

- São hierarquias de módulos
- Um pacote é um *diretório* que contém um arquivo chamado __init__.py
  - O pacote deve estar em um dos diretórios nos quais o Python busca por módulos
  - Para importar o pacote, use o nome do diretório
    - O programa correspondente ao pacote é __init__.py

Pacotes

- Os demais arquivos e diretórios dentro do pacote são encarados recursivamente como módulos
  - Por exemplo, se um pacote se chama p e contém um arquivo chamado m.py, então podemos importar
    - p (arquivo p/__init__.py)
    - p.m (arquivo p/m.py)
  - Semelhantemente, p poderia ter um outro pacote sob a forma de outro diretório contendo um arquivo __init__.py

Exemplo

```
$ dir python/
pacote  teste.py
$ dir python/pacote/
__init__.py  teste2.py
$ cat python/teste.py
print "teste"
$ cat python/pacote/__init__.py
print "pacote"
$ cat python/pacote/teste2.py
print "teste2"
$ python
...
>>> import teste
teste
>>> import pacote
pacote
>>> import pacote.teste2
teste2
```

EXERCÍCIOS

Recursão e Pacotes em Python...

EXERCÍCIO 1

Faça a implementação de uma função recursiva que calcule a sequência de Fibonacci, recebendo como parâmetro o número de elementos da sequência.

EXERCÍCIO 2

ARQUIVO: funcoes.py:
Importe o pacote "Random" e prepare uma função função que irá retornar aleatoriamente um valor de um dos lados de um dado (valores variando de 1 a 6).
ARQUIVO:jogo.py
Importe o arquivo funções.py
Implemente um programa de jogo de dados entre você e o computador, usando essa função.