# Python程序设计

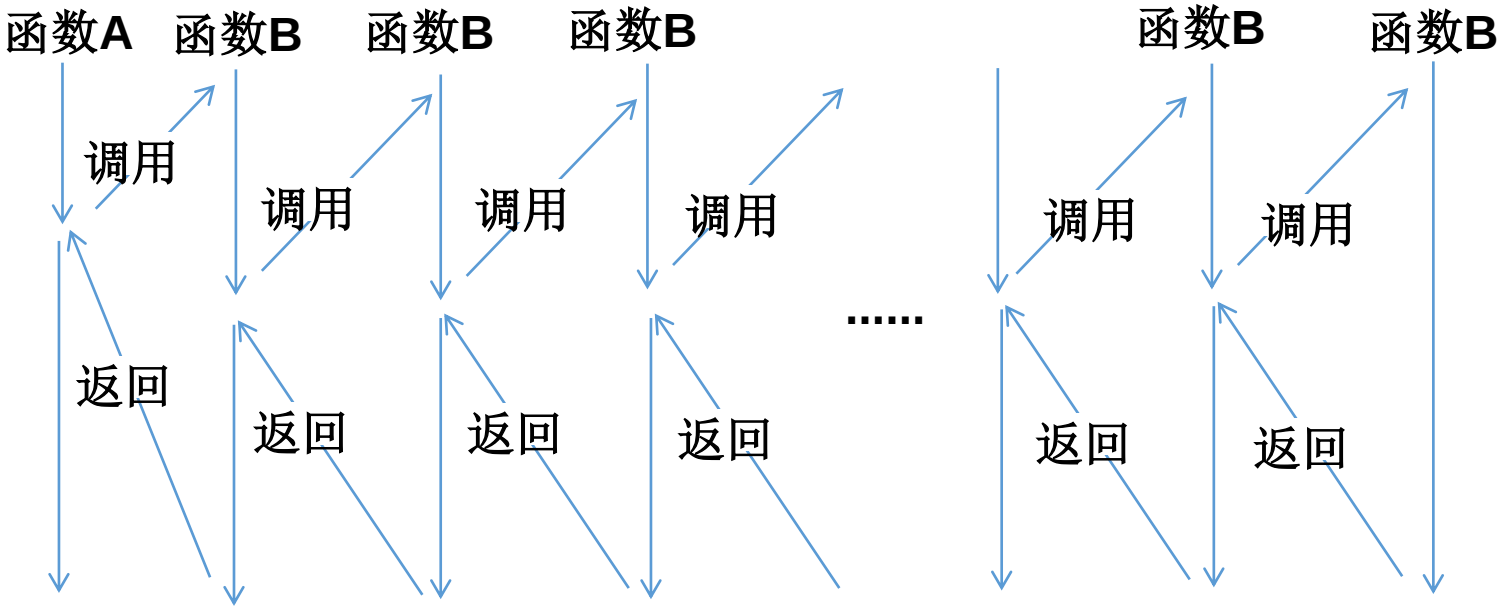
## 第六讲 函数与模块化

## 递归函数

张 华
WHU

# 递归函数

- 直接或间接调用自己的函数被称为递归函数。
  - 函数的递归调用是函数调用的一种特殊情况，函数调用自己，自己再调用自己，自己再调用自己，...，当某个条件得到满足的时候就不再调用了，然后再一层一层地返回直到该函数第一次调用的位置。



# 递归函数举例

## 问题：用递归方法计算n!。

### 分析

- 5! = 5 * 4 * 3 * 2 * 1
- 5! = 5 * 4!
- 4! = 4 * 3!
- ...
- 1! = 1

用迭代法实现

```
s,n=1,1
while n<=5:
    s = n * s
    n += 1
```

n! = n * (n-1)!

### 递归公式

$$n! = \begin{cases} 1 & (n=0或1) \\ n*(n-1)! & (n>1) \end{cases}$$

基本情形

简化问题

# 递归函数举例

## 案例：用递归方法计算n!。

### 程序代码

```
def fact(n):
    result = 1
    if n==1:
        result = 1
    else:
        result = n*fact(n-1)

    print("{0}!={1}".format(n, result))
    return result

num = int(input('input an integer(>=1):'))
fact(num)
```

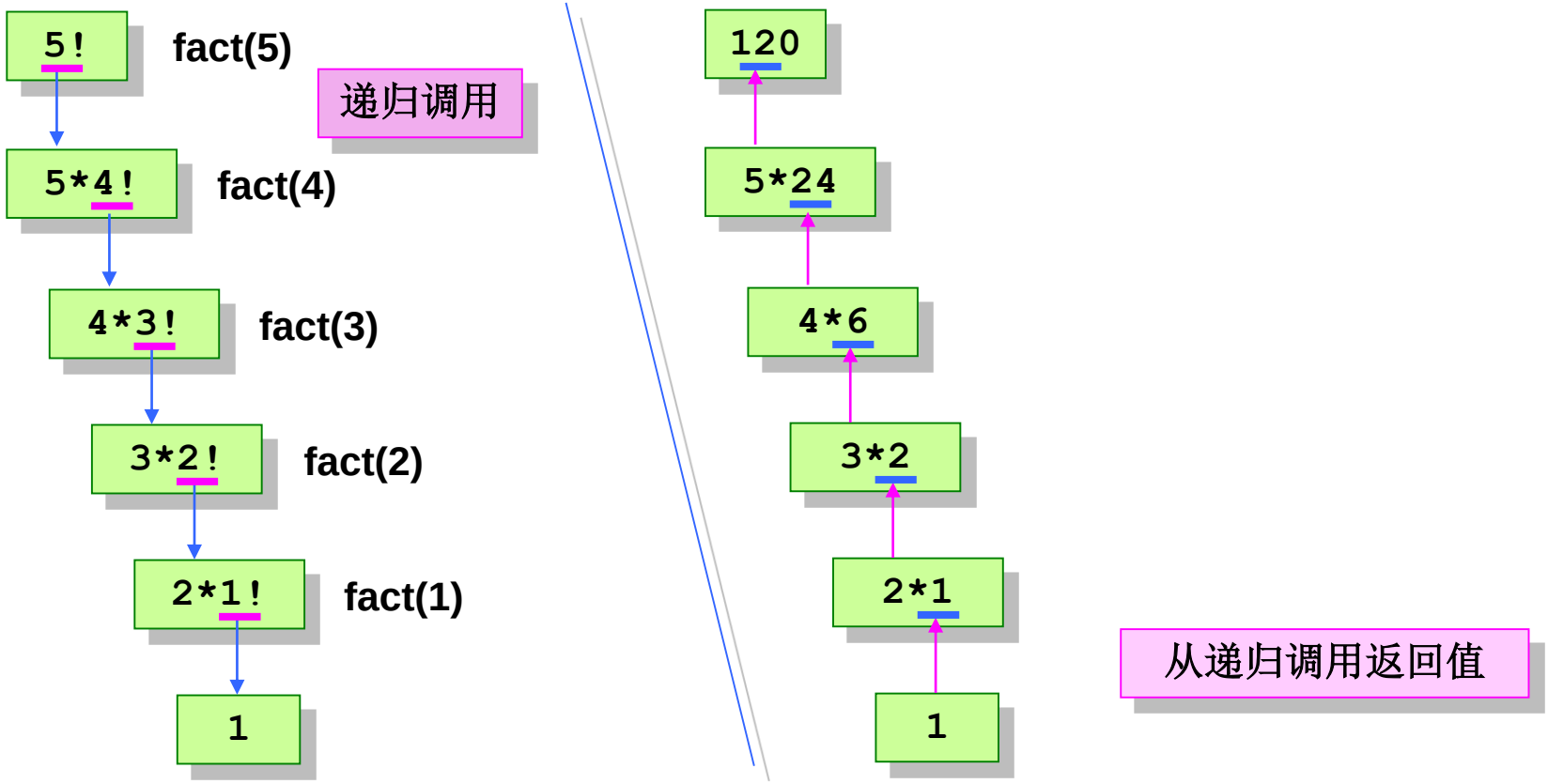
```
1!=1
2!=2
3!=6
4!=24
5!=120
```

递归调用

# 递归函数举例

## 案例：用递归方法计算n!。

### 程序执行的过程



# 递归方法与递归函数

- 用递归（函数）方法解决问题
  - 递归函数只知道如何去解最简单的情形（基本情形）
    - 例如，简单的返回一个值
  - 把复杂的问题分成两部分：
    - 可以直接求解的部分
    - 不能直接求解的部分
      - 这部分的问题与原问题相似，且复杂度降低
      - 函数可以通过调用自己来解决这个问题（递归调用）
  - 最终遇到基本情形
    - 函数识别出基本情形，将结果返回给前一个情形
    - 一系列的结果按顺序返回
    - 直到把最终结果返回给原始的调用者

# 递归方法举例

- 问题：使用递归方法计算斐波拉契（Fibonacci）数列。

  1 1 2 3 5 8 …

  - 分析
    - 第一、二个数是确定的。
    - 从第三个数开始，每个数是前两个数的和。
  - 递归公式
    - `fibo(1) = 1`
    - `fibo(2) = 1`
    - `fibo(n) = fibo(n-1) + fibo(n-2)`

# 递归法与迭代法

## 递归与迭代的比较

- 循环
  - 迭代：明确使用了循环结构
  - 递归：重复调用递归函数
- 终止条件
  - 迭代：循环条件不满足
  - 递归：遇到基本情形
- 都有可能出现无限循环
- 如何选择
  - 迭代：性能好
  - 递归：可读性好

# 递归函数小结

## 定义递归函数注意事项

- 必须设置终止条件
  - 缺少终止条件的递归函数，将会导致无限递归函数调用，其最终结果是系统会耗尽内存
- 必须保证收敛
  - 否则，也会导致无限递归函数调用
- 必须保证内存和运算消耗控制在一定范围

# 递归函数案例1

- 问题：用递归方法实现字符串反转。
  - 分析与设计
    - 将一个字符串视为递归对象。
      - 大的字符串由较小的对象组成，这些对象也是字符串。
    - 事实上，分割任何序列有一个非常方便的方法，即将它看成第一个数据项和后面跟随的另一个序列。
      - 对于字符串，可以将它划分为第一个字符和“所有其他字符”。
      - 如果我们反转字符串的剩下部分，然后将第一个字符放在最后一个字符之后，就反转了整个字符串。

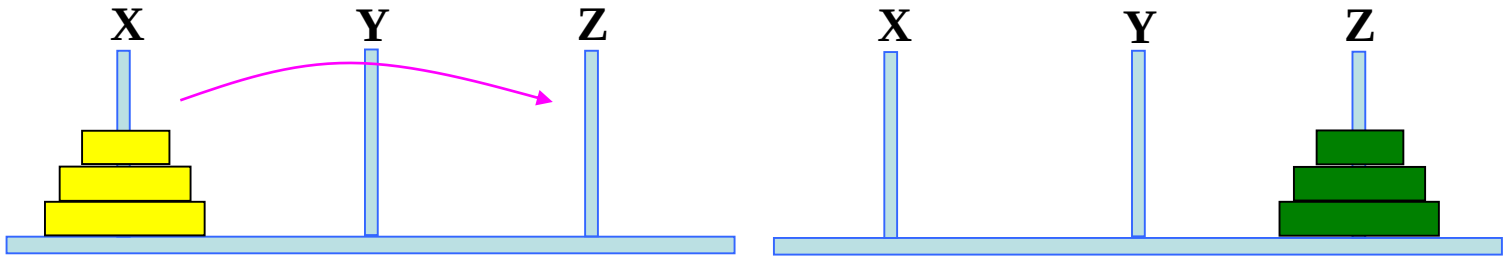
```
def reversestr(s):
    if s=="":
        return s
    else:
        return reversestr(s[1:])+s[0]

print(reversestr("Hello"))
```

# 递归函数案例2：汉诺塔问题

## 汉诺塔问题

### 问题

- 假设有三个分别命名为X，Y和Z的塔座，在塔座X上按从小到大的顺序放了n个大小不相同的圆盘，并依次编号为1，2，…，n。现在，要求将X上的n个圆盘移到塔座Z上，并按同样的顺序叠放。
- 移动时必须遵循以下规则：
  - 每次只能移动一个圆盘；
  - 圆盘可以插在X，Y和Z中的任一塔座上；
  - 任何时候都不能将一个较大的圆盘放在较小的圆盘上面。

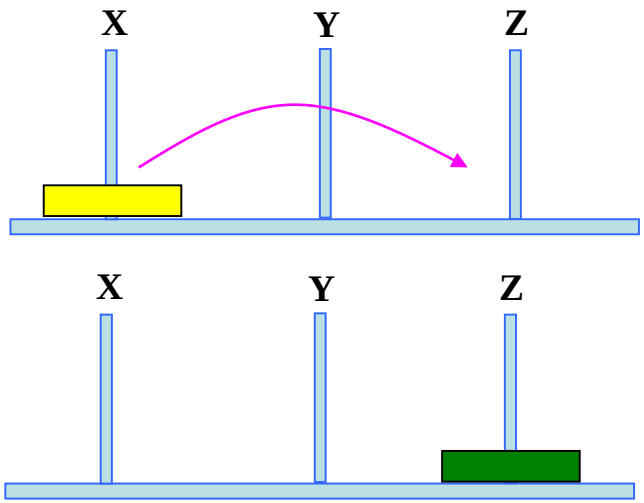


# 递归函数案例2：汉诺塔问题

## 汉诺塔问题

### 分析

- n=1时
  - 将圆盘1从塔座X移到塔座Z。




基本情形

# 递归函数案例2：汉诺塔问题

## 汉诺塔问题

### 分析

- n>1时

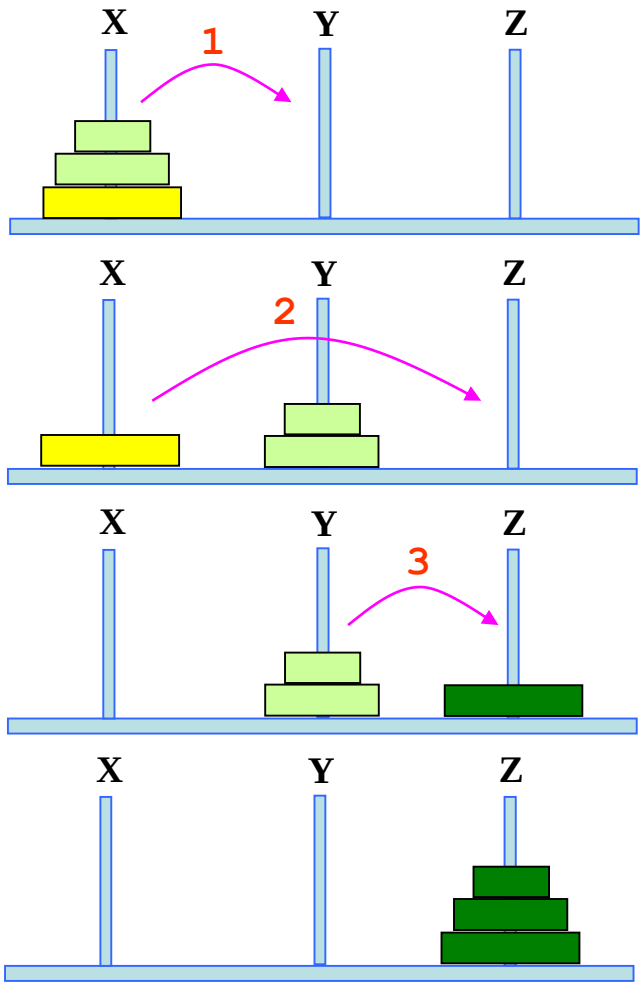
1．利用塔座z为辅助塔座，将压在圆盘n之上的n-1个盘从塔座x移到塔座y；

（—与原问题类似）

2．将圆盘n从塔座x移到塔座z；

3．利用塔座x为辅助塔座，将塔座y上的n-1个圆盘移动到塔座z。

（—与原问题类似）



# 递归函数案例2：汉诺塔问题

## 汉诺塔问题

### 设计

- **move 函数：移动一个盘**
  - 把编号为 n的盘 从 s 塔移到 d 塔

```
move(n, s, d);
```

- **hanoi 函数：移动n个盘的汉诺塔问题**
  - 把 n 个盘从 x 塔移到 z 塔，y 塔作为辅助塔

```
hanoi(n, x, y, z);
```

# 递归函数案例2：汉诺塔问题

## 汉诺塔问题

### 实现

```
def move(n, s, d):
    print("{0}\t{1}-->{2}".format(n, s, d))

def hanoi(n, x, y, z):
    if n==1:
        move(n, x, z)
    else:
        hanoi(n-1, x, z, y)
        move(n, x, z)
        hanoi(n-1, y, x, z)

hanoi(3, 'X', 'Y', 'Z')
```

```
1       X-->Z
2       X-->Y
1       Z-->Y
3       X-->Z
1       Y-->X
2       Y-->Z
1       X-->Z
```

# 递归函数案例3

- 问题：使用递归法对整数进行因数分解。

```
from random import randint

def factors(num, fac=[]):
    #每次都从2开始查找因数
    for i in range(2, int(num**0.5)+1):
        #找到一个因数
        if num%i == 0:
            fac.append(i)
            #对商继续分解，重复这个过程
            factors(num//i, fac)
            #注意，这个break非常重要
            break
    else:
        #不可分解了，自身也是个因数
        fac.append(num)
```

```
facs = []
n = randint(2, 10**8)
factors(n, facs)
result = '*'.join(map(str, facs))
if n==eval(result):
    print(n, '= '+result)
```