Python递归函数

讲师：Wayne

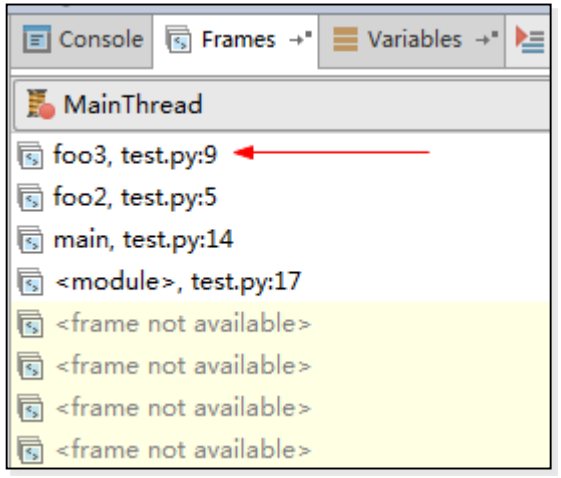
从业十余载，漫漫求知路

函数执行流程

http://pythontutor.com/visualize.html#mode=edit

```
def foo1(b, b1=3):
    print("foo1 called", b, b1)

def foo2(c):
    foo3(c)
    print("foo2 called", c)

def foo3(d):
    print("foo3 called", d)

def main():
    print("main called")
    foo1(100, 101)
    foo2(200)
    print("main ending")

main()
```

```
main called
foo1 called 100 101
foo3 called 200
foo2 called 200
main ending
```



- 全局帧中生成foo1、foo2、foo3、main函数对象
- main函数调用
- main中查找内建函数print压栈，将常量字符串压栈，调用函数，弹出栈顶
- main中全局查找函数foo1压栈，将常量100、101压栈，调用函数foo1，创建栈帧。print函数压栈，字符串和变量b、b1压栈，调用函数，弹出栈顶，返回值。
- main中全局查找foo2函数压栈，将常量200压栈，调用foo2，创建栈帧。foo3函数压栈，变量c引用压栈，调用foo3，创建栈帧。foo3完成print函数调用后返回。foo2恢复调用，执行print后，返回值。main中foo2调用结束弹出栈顶。main继续执行print函数调用，弹出栈顶。main函数返回

函数执行流程

http://pythontutor.com/visualize.html#mode=edit

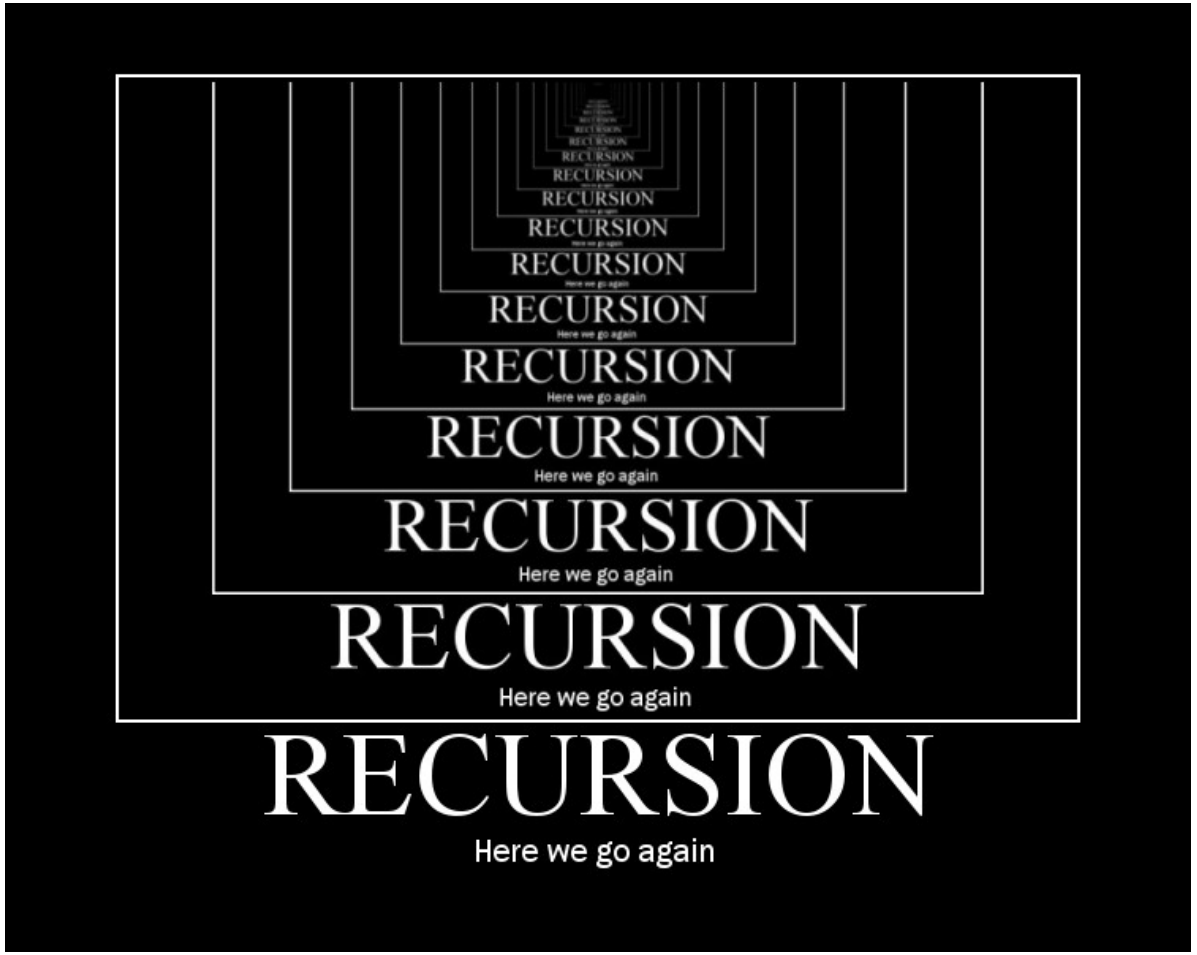
```
def foo1(b, b1=3):
    print("foo1 called", b, b1)
    foo2(2)

def foo2(a):
    pass
```

foo1函数的字节码

```
4           0 LOAD_GLOBAL              0 (print)
            3 LOAD_CONST               1 ('foo1 called')
            6 LOAD_FAST                0 (b)
            9 LOAD_FAST                1 (b1)
           12 CALL_FUNCTION            3 (3 positional, 0 keyword pair)
           # CALL_FUNCTION是函数调用，调用完成后，弹出所有函数参数，函数本身关闭堆栈，并推送返回值
           15 POP_TOP # 删除顶部（TOS）项目

5          16 LOAD_GLOBAL              1 (foo2)
           19 LOAD_CONST               2 (2)
           22 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
           25 POP_TOP
           26 LOAD_CONST               0 (None)
           29 RETURN_VALUE
```

递归Recursion

- 函数直接或者间接调用自身就是 递归
- 递归需要有边界条件、递归前进段、递归返回段
- 递归一定要有边界条件
- 当边界条件不满足的时候，递归前进
- 当边界条件满足的时候，递归返回



递归Recursion

- 斐波那契数列Fibonacci number：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- 如果设F(n）为该数列的第n项（n∈N*），那么这句话可以写成如下形式：:F(n)=F(n-1)+F(n-2)
- F(0)=0，F(1)=1, F(n)=F(n-1)+F(n-2)

```
a = 0
b = 1
n = 10 # 55
# 循环实现
for i in range(n - 1):
    a, b = b, a + b
else:
    print(b)
```

递归Recursion

- F(0)=0，F(1)=1, F(n)=F(n-1)+F(n-2)

```
def fib(n):
    return 1 if n < 2 else fib(n-1) + fib(n-2) # 条件为 n < 3更好
```

fib(4)解析：

fib(3) + fib(2)

fib(3)调用fib(3)、fib(2)、fib(1)

fib(2)调用fib(2)、fib(1)

fib(1)是边界return 1，所有函数调用返回

递归Recursion

- 递归要求
  - 递归一定要有退出条件，递归调用一定要执行到这个退出条件。没有退出条件的递归调用，就是无限调用
  - 递归调用的深度不宜过深
    - Python对递归调用的深度做了限制，以保护解释器
    - 超过递归深度限制，抛出RecursionError： maxinum recursion depth exceeded 超出最大深度
    - sys.getrecursionlimit()

递归的性能 fib35项比较

- for 循环

```
import datetime
start = datetime.datetime.now()
a = 0
b = 1
n = 35
# 循环实现
for i in range(n - 1):
    a, b = b, a + b
else:
    pass #print(b) 9227465

delta = (datetime.datetime.now() -
start).total_seconds()
print(delta)
```

- 递归

```
import datetime
n = 35
def fib(n):
    return 1 if n < 3 else fib(n-1) + fib(n-2)
start = datetime.datetime.now()
fib(35)

delta = (datetime.datetime.now() -
start).total_seconds()
print(delta)
```

递归的性能

- 循环稍微复杂一些，但是只要不是死循环，可以多次迭代直至算出结果
- fib函数代码极简易懂，但是只能获取到最外层的函数调用，内部递归结果都是中间结果。而且给定一个n都要进行近2n次递归，深度越深，效率越低。为了获取斐波那契数列需要外面在套一个n次的循环，效率就更低了
- 递归还有深度限制，如果递归复杂，函数反复压栈，栈内存很快就溢出了

- 思考：这个极简的递归代码能否提高性能呢？

递归的性能

- 斐波那契数列的改进

```
def fib(n, a=0, b=1):
    a, b = b, a+b
    if n == 0:
        return a
    return fib(n-1, a, b)

print(fib(4))
```

- 改进
  - 左边的fib函数和循环的思想类似
  - 参数n是边界条件，用n来计数
  - 上一次的计算结果直接作为函数的实参
  - 效率很高
  - 和循环比较，性能相近。所以并不是说递归一定效率低下。但是递归有深度限制

- 对比一下三个fib函数的性能

递归

- 间接递归

```
def foo1():
    foo2()

def foo2():
    foo1()

foo1()
```

间接递归，是通过别的函数调用了函数自身

但是，如果构成了循环递归调用是非常危险的，但是往往这种情况在代码复杂的情况下，还是可能发生这种调用。要用代码的规范来避免这种递归调用的发生

递归总结

- 递归是一种很自然的表达，符合逻辑思维
- 递归相对运行效率低，每一次调用函数都要开辟栈帧
- 递归有深度限制，如果递归层次太深，函数反复压栈，栈内存很快就溢出了
- 如果是有限次数的递归，可以使用递归调用，或者使用循环代替，循环代码稍微复杂一些，但是只要不是死循环，可以多次迭代直至算出结果
- 绝大多数递归，都可以使用循环实现
- 即使递归代码很简洁，但是能不用则不用递归

递归练习

- 求n的阶乘
- 将一个数逆序放入列表中，例如1234 => [4,3,2,1]
- 解决猴子吃桃问题
  - 猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个。第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第10天早上想吃时，只剩下一个桃子了。求第一天共摘多少个桃子。

谢谢

咨询热线 400-080-6560