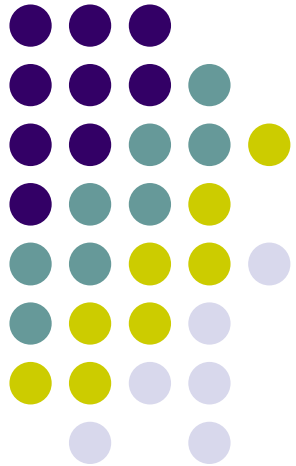
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

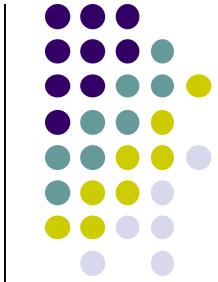
第4章模块化编程
Modular Programming



申丽萍

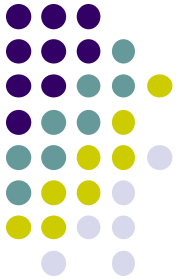
lpshen@sjtu.edu.cn

第4章 模块化编程



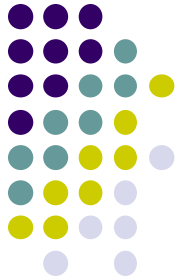
- 模块化程序设计
- 函数
- 自顶向下设计
- Python模块

模块化设计和建造



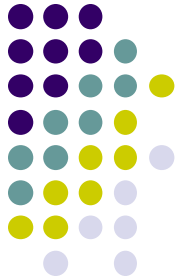
- 在对产品进行功能分析的基础上,将产品分解成若干个功能模块,预制好的模块再进行组装,形成最终产品.
- 模块:提供特定功能的相对独立的单元.
  - 标准化:标准尺寸和标准接口
  - 可组装:多个模块可以方便灵活地组合
  - 可替换:改变系统的局部功能
  - 可维护:对模块进行局部修改或设置

模块化编程



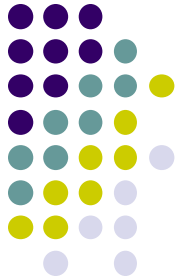
- 将程序分解为独立的、可替换的、具有预定功能的模块,每个模块实现一个功能.各模块组合在一起形成最终程序.
- 好处:
  - 易设计:复杂问题化成简单问题
  - 易实现:可以团队开发
  - 易测试:可各自测试
  - 易维护:修改或增加模块
  - 可重用:一个模块可参与组合不同程序

分离关注点原则



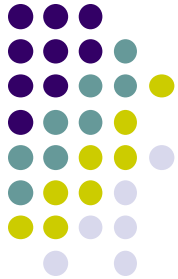
- 关注点:是指设计者关心的某个系统特性或行为
- 分离关注点(SoC):将系统分解为互不重叠的若干部分,每个部分对应于一个关注点.
- 模块化编程是SoC的具体体现,以程序的各个功能作为关注点,模块划分就是分离关注点的结果.

编程语言中的模块化构造



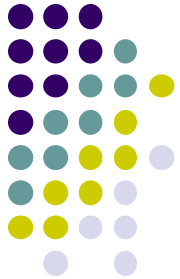
- 汇编语言:子例程,宏
- 高级语言:函数,过程
  - 有的语言不加区分,统称为函数.
- 包,模块,函数库,类库
  - 如数学库math和字符串库string

第4章 模块化编程



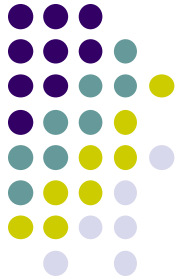
- 模块化程序设计
- 函数
  - 函数定义
  - 参数传递
  - 变量作用域
  - 返回值
- 自顶向下设计
- Python模块

什么是函数?



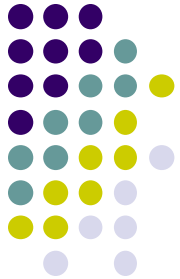
- 函数是一种程序构件,是构成大程序的小功能部件(子程序)
  - *function*一词本身就有"功能"的含义
  - 我们已经熟悉的函数:
    - 自己编的函数,如常用的`main()`
    - Python内建函数,如`abs(), type(),int(),eval(),sorted()`
    - Python库函数,如`math.sqrt(),string.count()`
    - 对象的方法,如`win.close()`和`p.draw()`
- 与数学函数的异同
  - 同:函数名,参变量,函数值,定义与使用
  - 异:算法过程定义;参量传递;副作用

为什么需要函数?



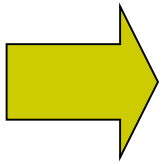
- 编程更容易把握
  - 大程序分解成小功能部件
- 代码重用,避免重复相同/相似代码
  - 提高开发效率
  - 更易维护
- 程序更可读,更易理解
- 代码简洁美观

函数用途:减少重复代码



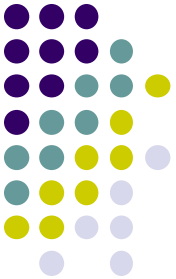
- 编程实例:画一棵树

```
print "    *"
print "   ***"
print "  *****"
print " *******"
print "    *"
print "   ***"
print "  *****"
print " *******"
print "    #"
print "    #"
print "    #"
```



```
def treetop():
    print "    *"
    print "   ***"
    print "  *****"
    print " *******"

def tree():
    treetop()
    treetop()
    print "    #"
    print "    #"
    print "    #"
tree()
```

重复代码的弊端

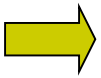


- 程序不必要地冗长
- 代码一致性维护麻烦:若修改一处代码,则所有重复的地方都要一致地修改
- 程序看上去累赘不美观
- 输入大量重复代码很单调乏味

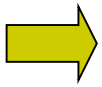
函数用途:改善程序结构

- 模块化:将程序分解成多个较小的相对独立的函数,可使程序结构清晰,容易理解.

```
def treetop():
    print "   *"
    print "  ***"
    print " *****"
    print "********"

def tree():
    treetop()
    treetop()
    print "   #"
    print "   #"
    print "   #"
tree()
```



```
def treetop():
    print "   *"
    print "  ***"
    print " *****"
    print "********"
def treetrunk():
    print "   #"
    print "   #"
    print "   #"
def main():
    treetop()
    treetop()
    treetrunk()
main()
```

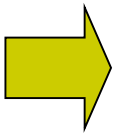


```
def treetop1():
    print "   *"
    print "  ***"
    print " *****"
    print "********"
def treetop():
    treetop1()
    treetop1()
def treetrunk():
    print "   #"
    print "   #"
    print "   #"
def main():
    treetop()
    treetrunk()
main()
```

函数用途:提高程序通用性

- 换用^字符来画树,以便比较美观度.

```
def treetop1():
    print "    *"
    print "   ***"
    print "  *****"
    print " *******"
def treetop2():
    print "    ^"
    print "   ^^^"
    print "  ^^^^^"
    print " ^^^^^^^"
def star_treetop():
    treetop1()
    treetop1()
def caret_treetop():
    treetop2()
    treetop2()
......
```

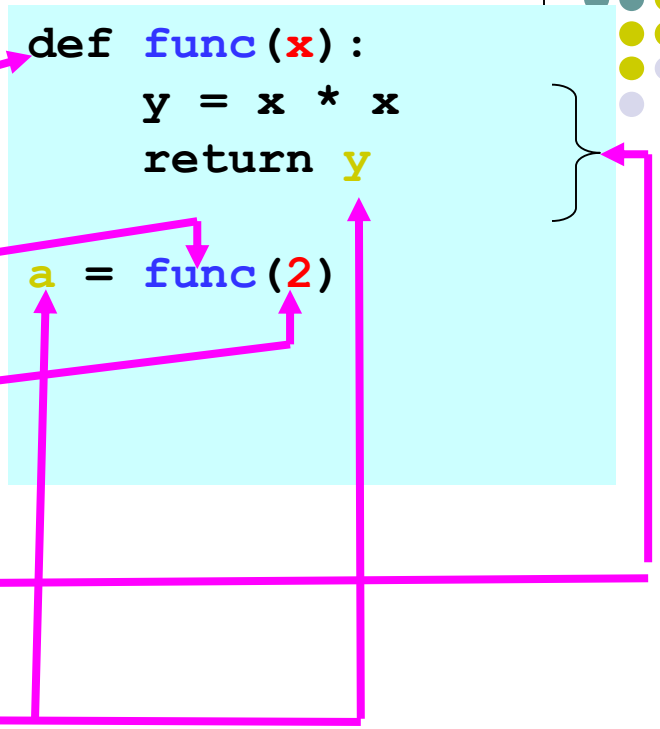


利用函数参数
提高通用性

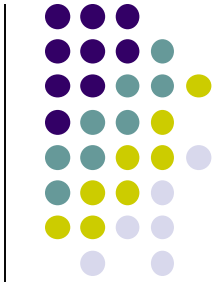
```
def treetop(ch):
    print "    %s" % (ch)
    print "   %s" % (3*ch)
    print "  %s" % (5*ch)
    print " %s" % (7*ch)
def star_treetop():
    treetop('*')
    treetop('*')
def caret_treetop():
    treetop('^')
    treetop('^')
......
```

函数的定义和调用

- 先定义(*define*)
- 再通过函数名调用
- 调用时传递参数
- 调用执行的是函数体(语句序列)
- 调用产生返回值
- 函数定义可置于程序中任何地方,但必须在调用之前



函数的定义与调用过程



- 函数定义

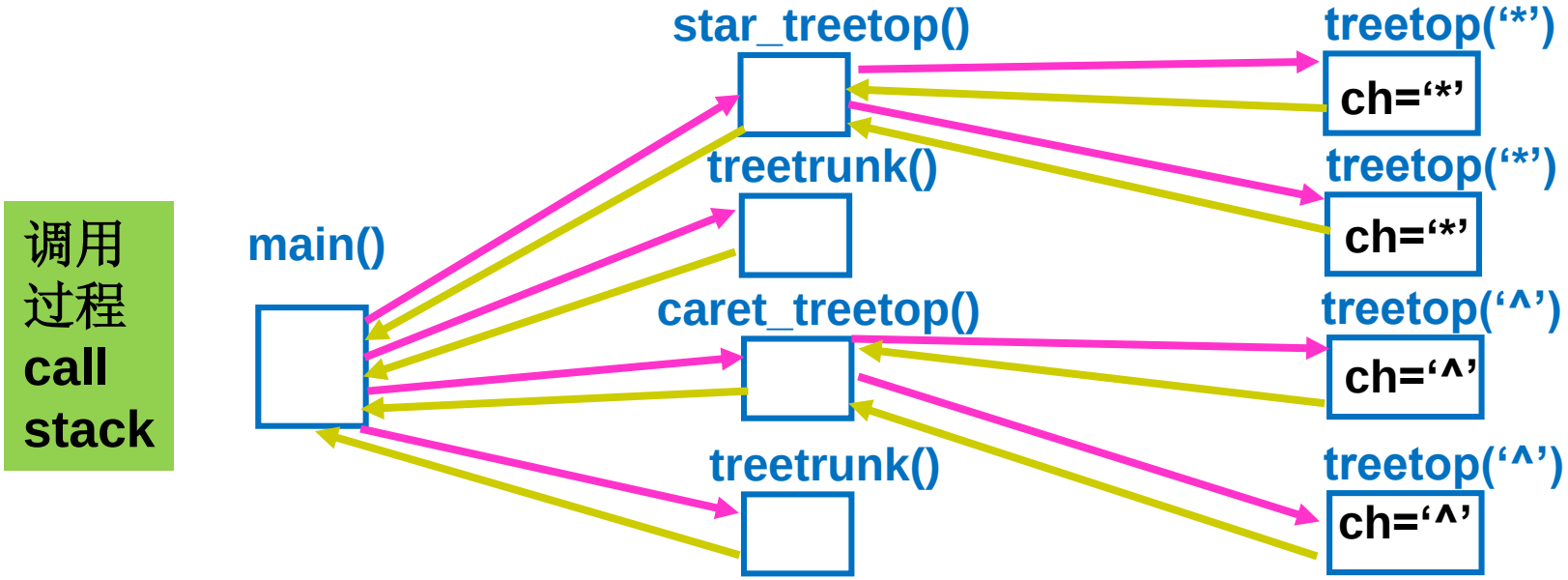
  def <函数名>(<形参列表>):

  <函数体>
- 函数调用

  <函数名>(<实参列表>)
  - 调用者被挂起
  - 函数形参被赋值为实参
  - 执行函数体
  - 控制返回调用者(调用点的下一条语句)

函数调用过程图解

```
def main():
    star_treetop()
    treetrunk()
    print
    caret_treetop()
    treetrunk()

def star_treetop():
    treetop("*")
    treetop("*")
def caret_treetop():
    treetop("^")
    treetop("^")
def treetrunk():
    print "    #"
    print "    #"
    print "    #"

def treetop(ch):
    print "    %s" % (ch)
    print "  %s" % (3*ch)
    print " %s"% (5*ch)
    print "%s" % (7*ch)
```



调用过程 **call stack**

参数传递

- 对于函数定义 `def f(x,y,z): ...`
  - 按位置传递

```
f(1,2,3)
f(1,a,b)
```

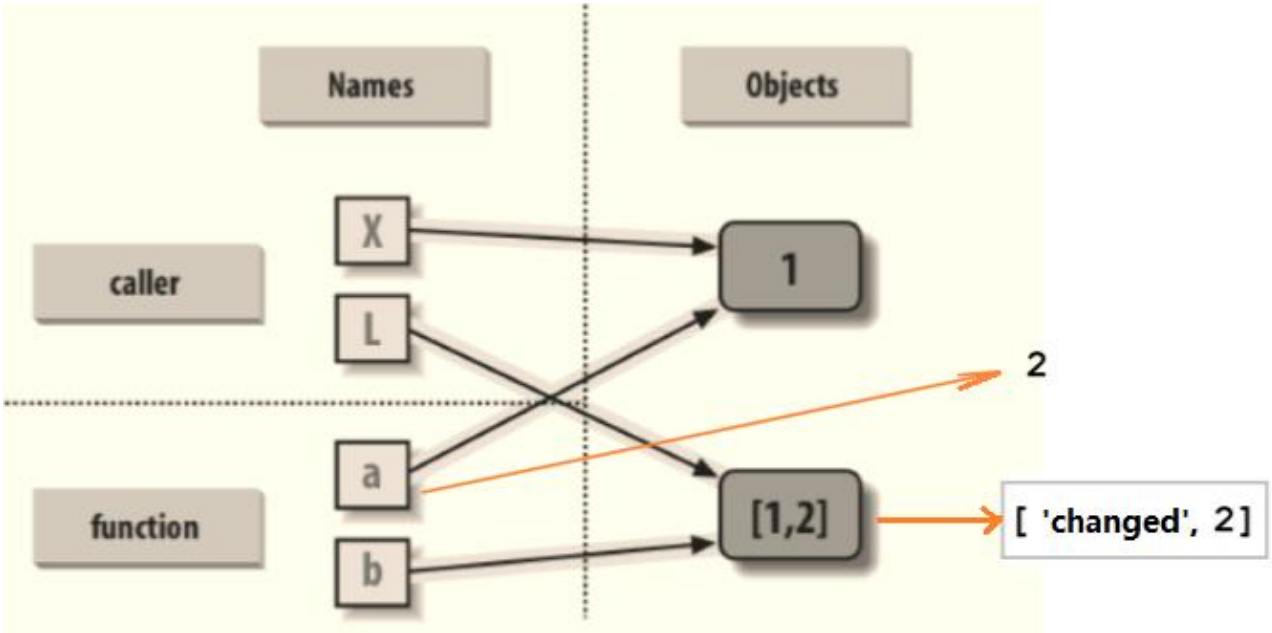
  - 按名传递:形参=实参

```
f(z=3, x=1,y=2)
```

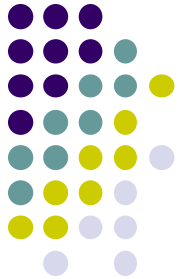
- 为函数参数指定默认值 `def f(x,y=2,z=3): ...`
  - `f(1)` x=1,y=2,z=3
  - `f(1,7)` x=1,y=7,z=3
  - `f(x=1,z=8)` x=1,y=2,z=8

参数通过引用传递，会改变实参

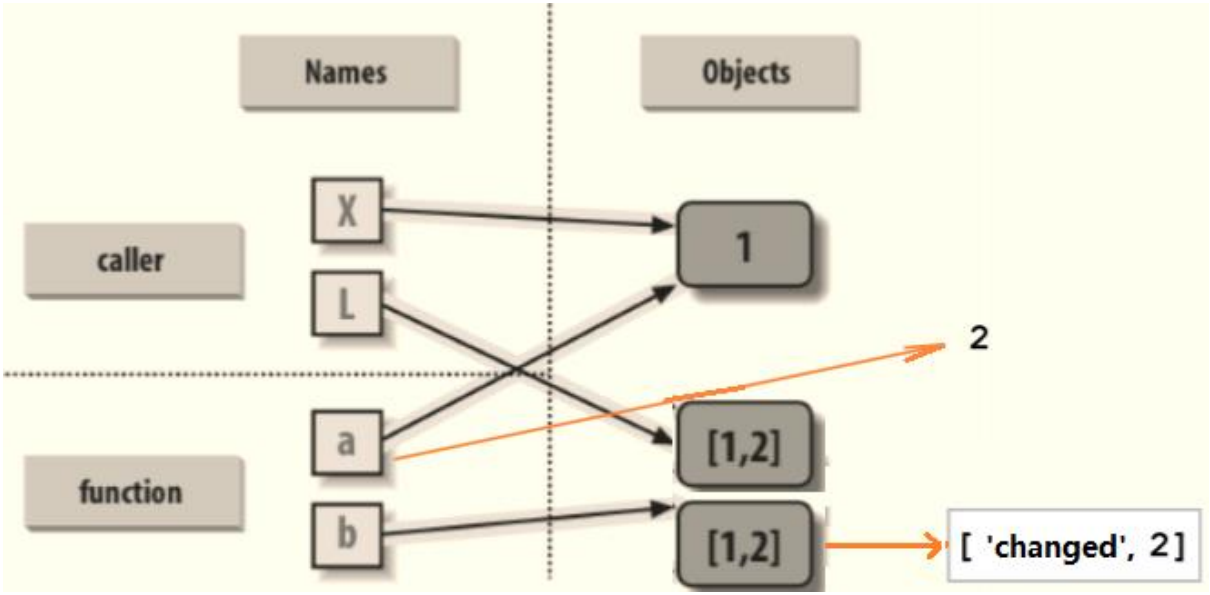
```
def changer(a,b):
    a=2                 # changes local name's value only
    b[0]='changed'      # changes shared object in-place

def main():
    X=1
    L=[1,2]
    changer(X,L)        # pass immutable and mutable objects
    print X,L           # X unchanged, and L changed
```



避免改变实参：传递不可改变实参

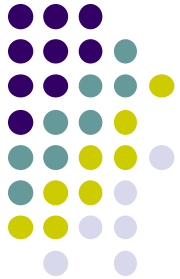


```
def changer(a,b):
    a=2                 # changes local name's value only
    b[0]='changed'      # changes shared object in-place

def main():
    X=1
    L=[1,2]
    changer(X,L[:]) # avoid passing mutable objects
    print X,L       # X and L both unchanged
```




```
>>> list1=[1,2,3]
>>> list2=[1,2,3]
>>> list3=list2
>>> list2[1]=9
>>> list3
[1, 9, 3]
>>> list4=list1[:]
>>> list1[1]=0
>>> list4
[1, 2, 3]
```

变量的作用域

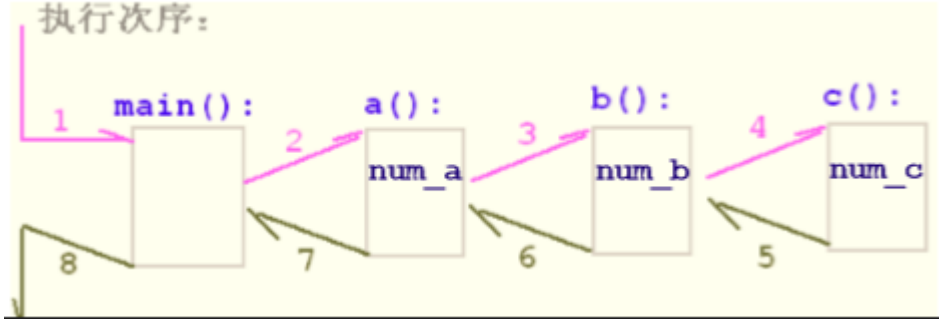


- 变量具有作用域(*scope*):即可以引用该变量的程序区域.
  - 不同作用域中的变量,即使同名,也是不同的变量!
  - 作用域可以是局部,也可以是全局
- 函数中定义的变量是局部的:即作用域是函数体.
  - 函数的形参可视为局部变量,只不过是在调用时才赋值.

```
x,y = 0,0
def f(x):
    y = 1
    print x,y
f(x)
print x,y
```

变量的作用域

```
def a():
    print 'Hello from a()'
    num_a=10
    b()
def b():
    print 'Hello from b()'
    num_b=20
    c()
def c():
    print 'Hello from c()'
    num_c=30
    print num_a
    print num_b
    print num_c

def main():
    a()
```



Debug Control

Go | Step | Over | Out | Quit | Stack | Source | Locals | Globals

demo-function-scope.py:14: c()

NameError: global name 'num_a' is not defined

'bdb'.run(), line 400: exec cmd in globals, locals
'__main__'.<module>(), line 21: main()
'__main__'.main(), line 19: a()
'__main__'.a(), line 6: b()
'__main__'.b(), line 10: c()
> '__main__'.c(), line 14: print num_a

Locals

num_c 30

Globals

__builtins__	<module '__builtin__' (built-in)>
__doc__	None
__file__	'D:/courses/Python/Demo/demo-function-scope.py'
__name__	'__main__'
__package__	None
a	<function a at 0x00000000028DAEB8>
b	<function b at 0x0000000002FF9BA8>
c	<function c at 0x0000000002FF9C18>
main	<function main at 0x0000000002FF9588>

```
Hello from a()
Hello from b()
Hello from c()

Traceback (most recent call last):

  File "D:/courses/Python/Demo/demo-function-scope.py
    print num_a
NameError: global name 'num_a' is not defined
```

变量的作用域

```
def a():
    print 'Hello from a()'
    num_a=10
    b()
def b():
    print 'Hello from b()'
    num_b=20
    def c():
        print 'Hello from c()'
        num_c=30
        print num_a
        print num_b
        print num_c
    c()

def main():
    a()

num_a=8
main()
```

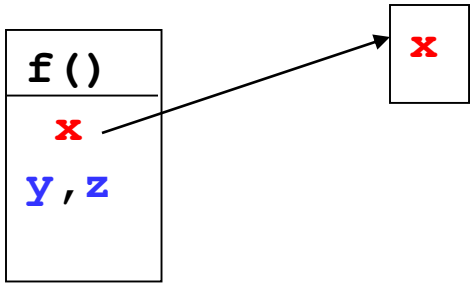
```
Hello from a()
Hello from b()
Hello from c()
8
20
30
```

请画出各函数执行顺序、call stack及变量作用域

函数如何使用外部数据

- Python中,函数体可直接引用外部的变量.

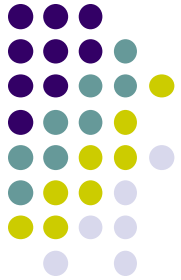
```
x = 0
def f(y):
    z = 1
    print x,y,z
f(10)
```



  - 但这用法很不好!不符合模块化要求.
- 应当通过参数向函数传递数据

```
x = 0
def f(p,y):
    z = 1
    print p,y,z
f(x,10)
```

全局变量



- 函数若需引用并修改外部变量,可声明全局变量
  - 全局变量声明： global x
  - 函数内的全局变量声明前不可以赋值（可读），否则视为局部变量。

```
>>> x = 1
>>> def h():
        global x
        print x
        x = 2      #此赋值是针对全局变量的
        print x
>>> def f():
        print x    #未定义变量视为全局变量
        x = 2      #赋值引入局部变量,则前行出错!
```

```
>>> h()
1
2
```

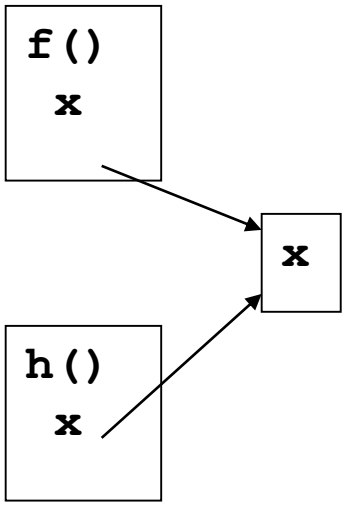
```
>>> f()

Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    f()
  File "<pyshell#35>", line 2, in f
    print x
UnboundLocalError: local variable 'x' referenced before assignment
```

```
>>> def foo():
        c = 8
        global c
        c = 3

SyntaxError: name 'c' is assigned t
(<pyshell#18>, line 1)
```

全局变量的用途 （注意：不提倡使用全局变量）

- 多个函数处理共享数据

```
def f():
    global x
    x = x + 1
    print x
def g():
    global x
    x = x - 1
    print x
x = 0
f()
g()
```



函数的返回值

- 函数与调用者之间的信息交互:
  - 通过形参从调用者输入值
  - 通过返回值向调用者输出值
- 定义

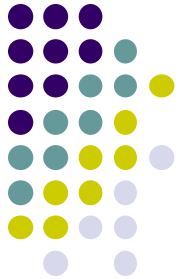
  def <函数名>(<形参>):

  ……

  return <表达式1>，…，<表达式n>
  - return计算各表达式,将结果返回调用者,退出函数
- 如果函数定义中没有return, Python仍会返回一个特殊值:None.

```
>>> def f(x):
    if x <= 0:
        print "请输入正数."
        return
    y = x ** 3
    return y

>>> print 3+f(-3)
请输入正数.

Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    print 3+f(-3)
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

函数返回值的使用



- 如果没有或者用不上函数返回值,则函数调用可以直接当成一条语句。如f(3)
  - 相当于某些语言中的"过程调用"
- 如果想使用函数返回值,则有两种用法
  - 用变量接收返回值,如

    x = f(3)

    print 2 + x * 4
  - 直接用在表达式中,如

    print 2 + f(3) * 4
- 忘记接收函数返回值是Python初学者的常见错误

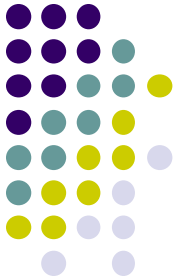
例：函数返回值使用

- 求两点距离的函数

```
from math import sqrt
def sq(x):
    return x * x

def dist(u,v):
    d = sqrt(sq(v[0]-u[0])+sq(v[1]-u[1]))
    return d
```

- 用一个函数辅助定义另一个函数，这是化繁为简的常用做法.

函数多个返回值的接收



```
>>> def headtail(list):
        return list[0],list[len(list)-1]
```

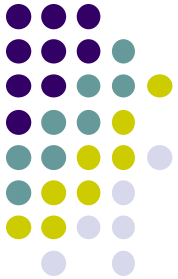
- 用多个变量

```
>>> h,t = headtail([1,2,3,4,5])
>>> print h,t
1 5
```

- 用一个变量,接受的值是元组.

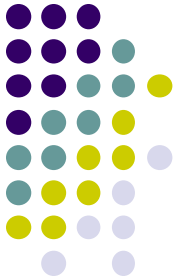
```
>>> v = headtail([1,2,3,4,5])
>>> v
(1, 5)
```

第4章 模块化编程



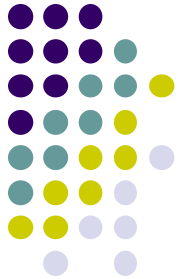
- 模块化程序设计
- 函数
- **自顶向下设计**
- Python模块

自顶向下设计



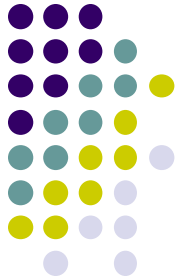
- 对复杂问题常采用逐层分解的设计方法,也称为逐步求精.
  - 首先对整个系统进行顶层子系统的设计.在此并不给出各个子系统的细节.
    - 只定义子系统的规格（输入输出），即接口定义。
  - 其次对每个子系统重复这个设计过程,即再分解为下一层子系统.
  - 直至每个子系统的功能足够简单,可以直接编码实现.

编程案例:打印年历



- 程序规格
  - 程序:calendar
  - 输入:公元年份year(1900以后)
  - 输出:year年年历
  - 输入与输出的关系是:根据year可算出相对于1900年1月1日(星期一)总共过去了多少天,按7天循环即可得知year年1月1日是星期几,从而可推算出全年年历.

顶层设计



- 基本算法

  输入year

  计算year年1月1日是星期几

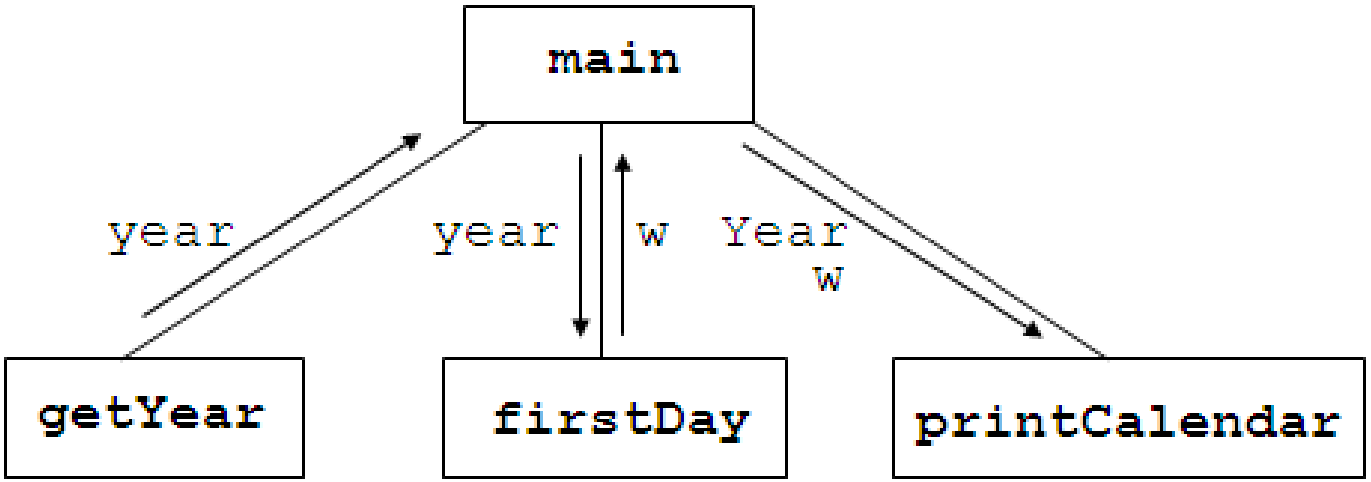
  输出年历基本程序

- 主模块

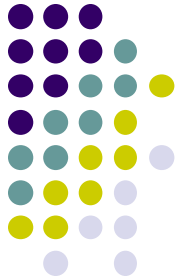
```
def main():
    year = getYear()
    w = firstDay(year)
    printCalendar(year,w)
```

结构图

- 用结构图(或称模块层次图)表示:

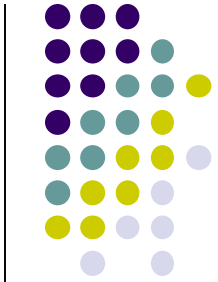


函数的抽象



- 抽象:确定某事物的重要特性并忽略其他细节的过程.
- 在自顶向下设计的每一层,用接口指明需要下一层的哪些细节;其他可暂时忽略.
  - 例如,main()需要getYear返回year,但如何获得这个信息则暂时忽略;
  - 又如,main()将year传给firstDay(),并期望该模块返回元旦的星期信息w.如何得到w也暂时忽略.

第二层设计(1)



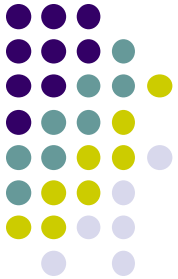
- getYear足够简单,可以直接编码:

```
def getYear():
    print "本程序打印年历."
    year = input("请输入年份(1900后):")
    return year
```

- firstDay还需要闰年的信息

```
def firstDay(year):
    k = leapyears(year)
    n = (year-1900) * 365 + k
    return (n + 1) % 7
```

第二层设计(2)

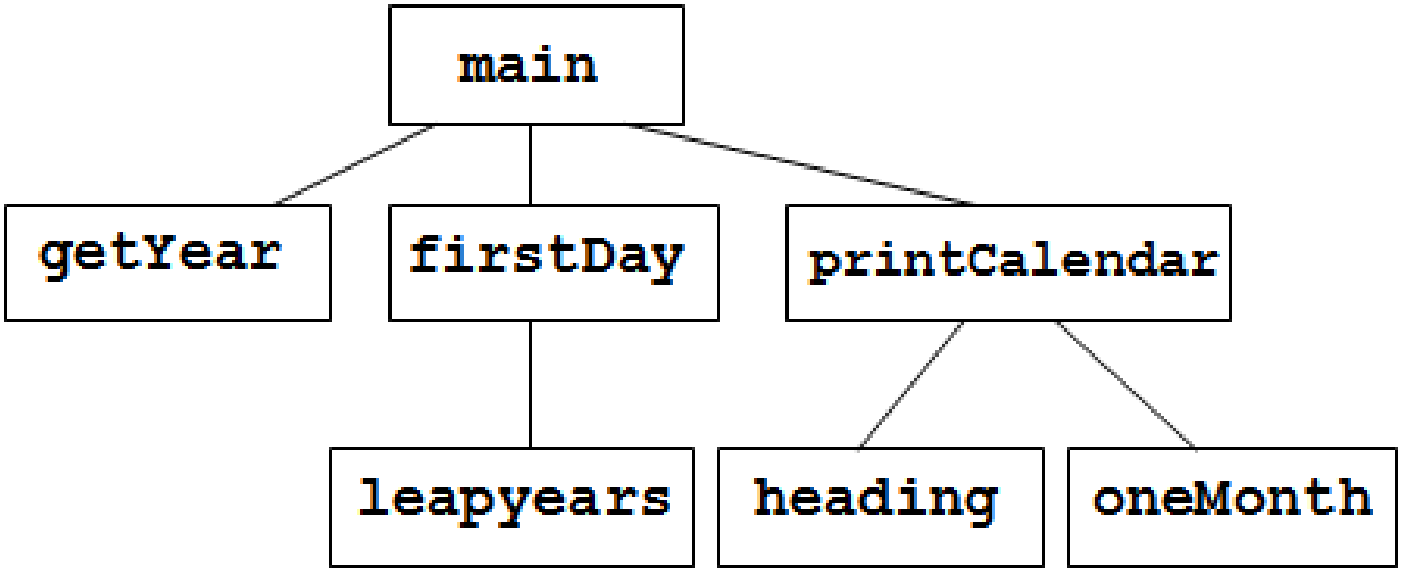


- printCalendar():打印年历需要打印每个月,每个月应该有标题.

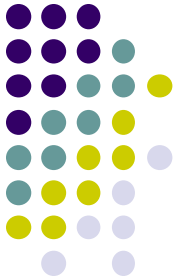
```
def printCalendar(year,w):
    print
    print "===== " + str(year) + " ====="
    first = w
    for month in range(12):
        heading(month)
        first = oneMonth(year,month,first)
```

第二层设计(3)

- 结构图


main
getYear
firstDay
printCalendar
leapyears
heading
oneMonth

第三层设计(1)



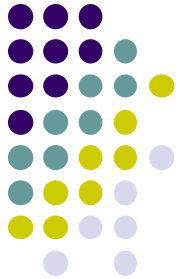
- leapyears():1900~year间的闰年数

```
def leapyears(year):
    count = 0
    for y in range(1900,year):
        if y%4 == 0 and (y%100 != 0 or y%400 == 0):
            count = count + 1
    return count
```

- heading():月历标题栏

```
def heading(m):
    months = ["Jan","Feb", ... ,"Dec"]
    print "        %s        " % (months[m])
    print "Mon Tue Wed Thu Fri Sat Sun"
```

第三层设计(2)

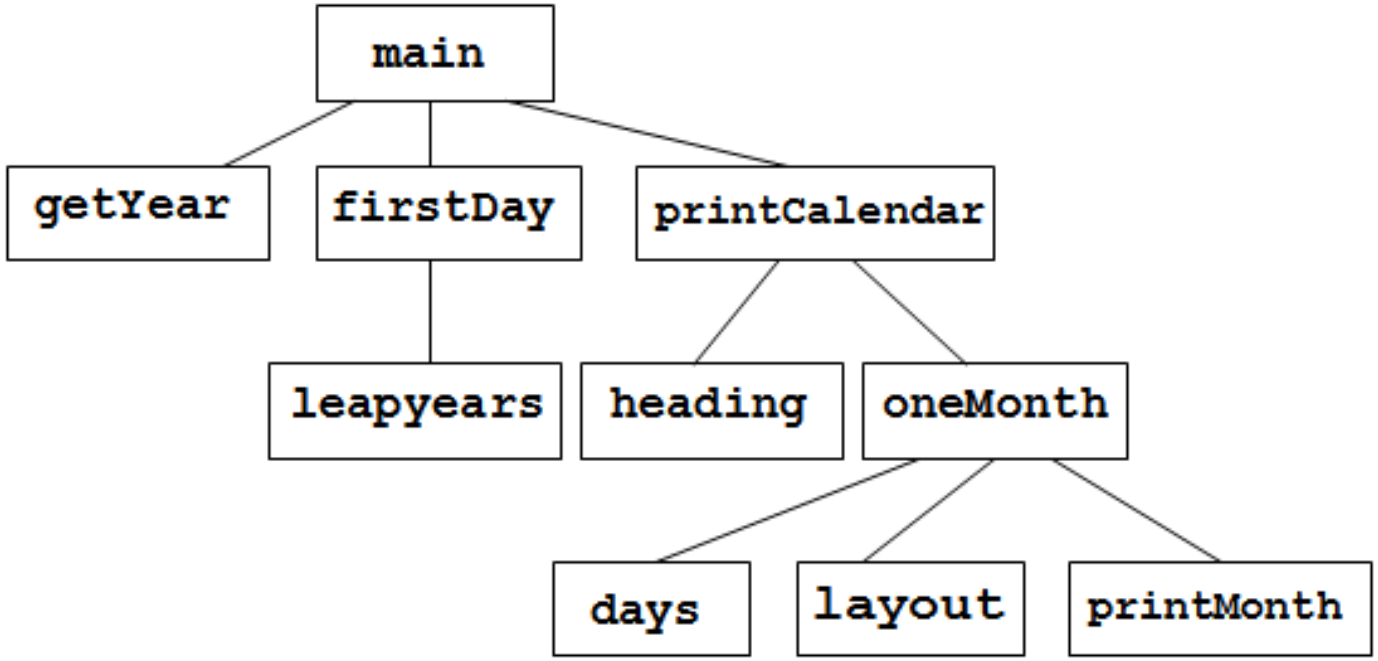


- oneMonth():打印一个月的月历
  - 需要知道该月1日是星期几,该月有多少天,输出布局是怎样的. 返回该月1日是星期几作为下一个月打印用。

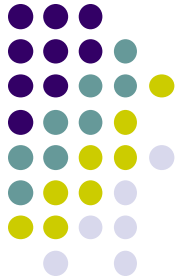
```
def oneMonth(year,month,first):
    d = days(year,month)
    frame = layout(first,d)
    printMonth(frame)

    return (first + d) % 7
```

第三层设计(3)

- 结构图


main
getYear
firstDay
printCalendar
leapyears
heading
oneMonth
days
layout
printMonth

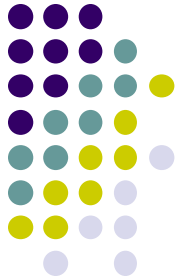
第四层设计(1)



- 函数days()

```
def days(y,m):
    month_days = [31,28,31,30,31,30,31,31,30,31,30,31]
    d = month_days[m]
    if (m == 1) and (y%4 == 0 and \
       (y%100 != 0 or y%400 == 0)):
        d = d + 1
    return d
```

第四层设计(2)

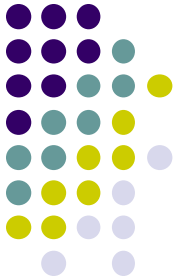


- 函数layout()

Mon	Tue	Wed	Thu	Fri	Sat	Sun

```
def layout(first,d):
    frame = 42 * [""]
    if first == 0:
        first = 7
    j = first - 1
    for i in range(1,d+1):
        frame[j] = i
        j = j + 1
    return frame
```

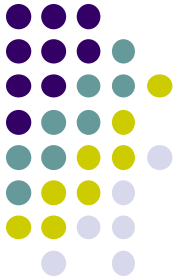
第四层设计(3)



- 函数printMonth()

```
def printMonth(frame):
    for i in range(42):
        print "%3s" % (frame[i]),
        if (i+1)%7 == 0:
            print
```

单元测试

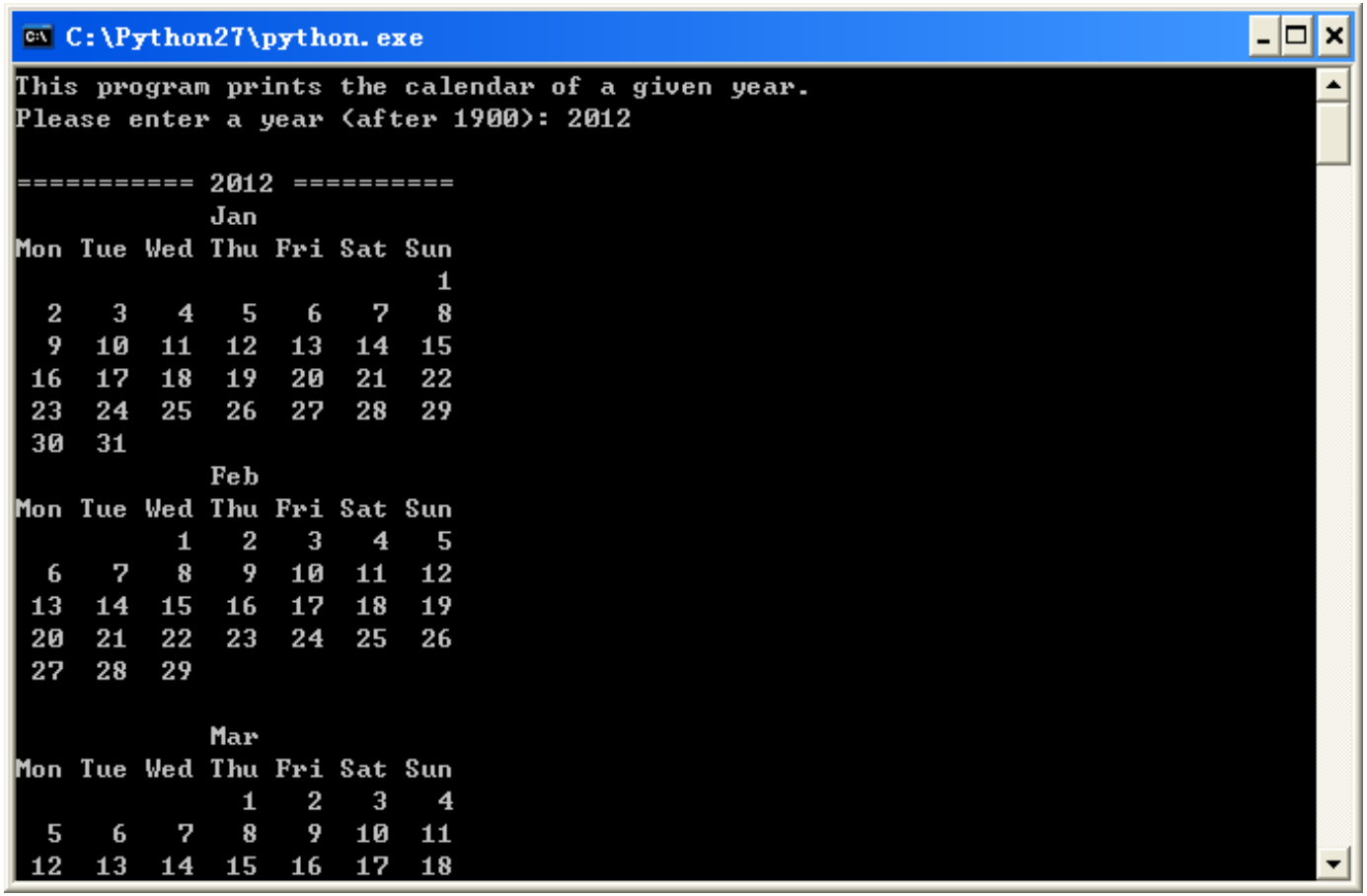


- 在模块化编程中,适合采用单元测试技术,即先分别测试每一个小模块,然后再逐步测试较大的模块,直至最后测试完整程序.
- 例如:测试days(y,m)——y年m+1月的天数.
  - **将days(y,m)存入模块moduletest.py**

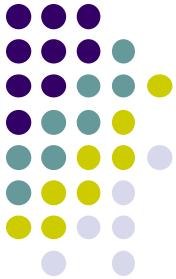
```
>>> from moduletest import days
>>> days(1900,0)
31
>>> days(1900,1)
28
......
```

完整程序

- 完整程序:calendar.py

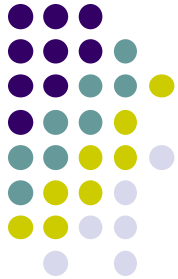


设计与实现过程小结



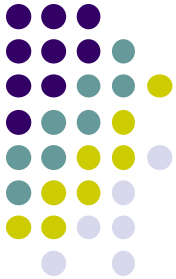
- 自顶向下设计,逐步求精
  - 将问题分解为若干子问题
  - 为每个子问题设计一个(函数)接口
  - 将原问题的算法用各子问题对应的函数接口来表达
  - 对每个子问题重复上述过程
- 自底向上实现
- 单元测试

第4章 模块化编程



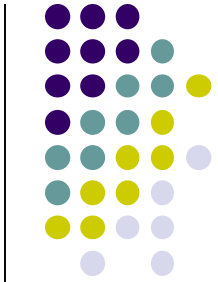
- 模块化程序设计
- 函数
- 自顶向下设计
- Python模块

Python模块



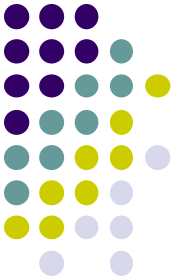
- 模块是指相对独立的程序单元。
- Python模块是最高层结构单元，对应于一个Python程序文件。
- Python模块必须用import或from语句导入后才能使用。
- 模块的作用：
  - 实现代码重用：模块中的代码可以被多次加载运行，也可以被多个程序加载使用。
  - 提供独立的名字空间：模块中定义的所有名字（函数名、类名、变量名等）都局限于本模块，与模块外不会发生冲突。
  - 实现共享：一个模块中定义一个全局变量，然后使用者可以导入该模块，从而共享该全局变量。

Python标准库中的模块

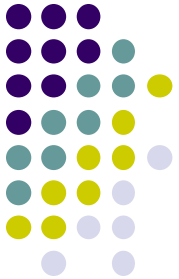


- string
- math
- sys
  - byteorder, long_info, path, modules, getdefaultencoding(),
- time
  - gmtime(), localtime(),
  - strftime("%a, %d %b %y %h:%m:%s +0000", localtime())
  - strptime("30 nov 00", "%d %b %y")
- random
  - random(), uniform(a,b), randint(a,b), getrandbits(n)
  - choice(items), shuffle(items) , sample(items, 3)

assign5



- 注意从ftp://public.sjtu.edu.cn/ct/assignments 下载详细作业5文档
- 上机时间：11月19日 8：00~9：40
- 上机地点：电院4号楼313机房
- 截止日期：11月19日 24：00



End