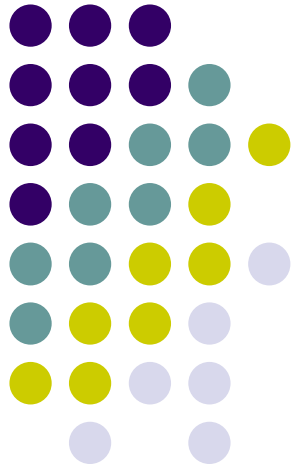
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

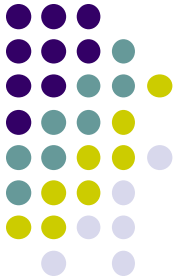
第3章数据处理的流程控制
Process Control

申丽萍

lpshen@sjtu.edu.cn

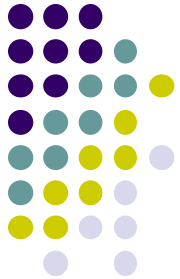


第3章 数据处理的流程控制



- 顺序控制结构
- 分支控制结构
- 异常处理
- 循环控制结构
- 结构化程序设计

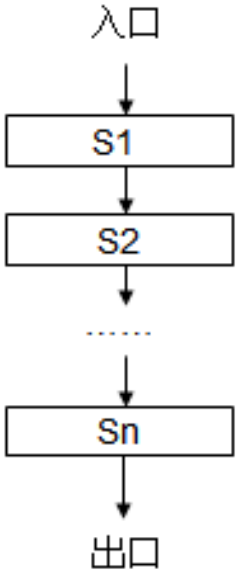
流程控制



- 数据是被加工处理的原材料,而处理过程要用流程控制结构来描述
  - 类比:烹调=食材+烹制过程
    - 烹制过程:先炒再煮;如果淡了则加盐;反复翻炒5分钟;...
- 常见的流程控制结构
  - 顺序,跳转,分支,循环,子程序等
- 好的流程:结构清晰,易理解,易验证,易维护

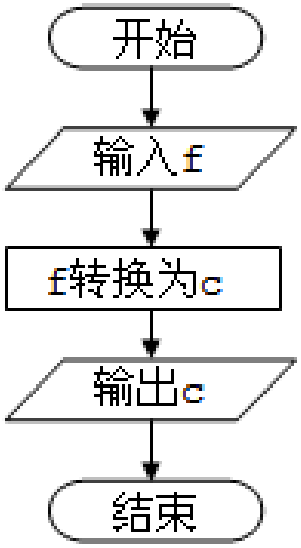
顺序控制结构

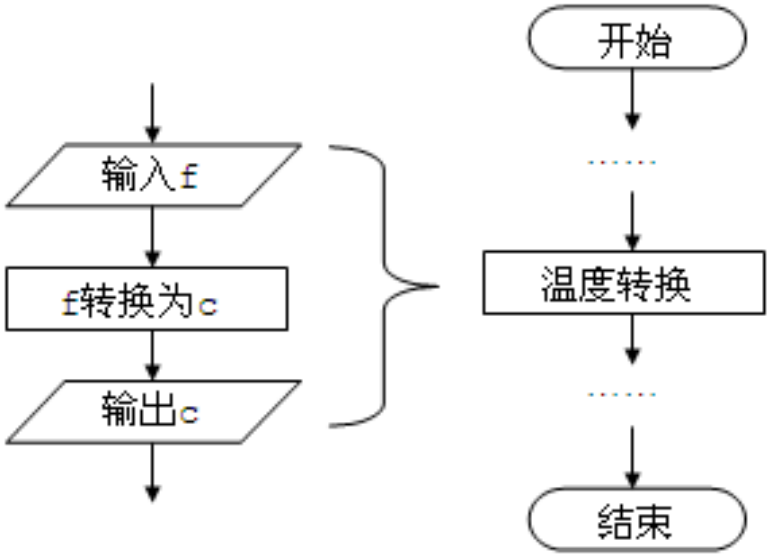
- 按语句的自然先后顺序执行



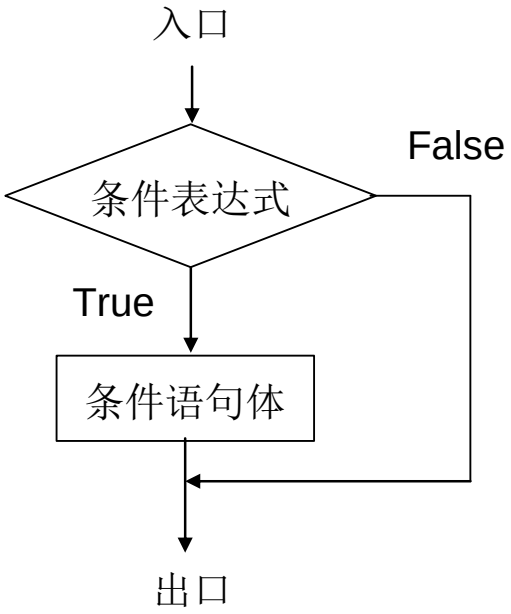
编程实例

- 温度转换程序eg3_1.py :华氏转换成摄氏
- 流程图:用标准化的图形符号来表示程序步骤
  - 流程图中的步骤可以是不同抽象级的





分支控制结构(1)

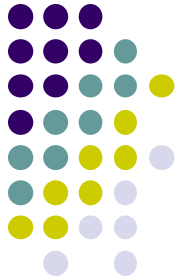


- 可以选择不同的执行路径
- 单分支结构

```
if <条件>:
        <语句体>
```

  - <条件>: 布尔表达式
  - <语句体>: 语句序列.
    - 左边需要"缩进"一些空格.
- 语义:计算<条件>的真假.若为真,则执行<语句体>,并把控制转向下一条语句;若为假,则直接把控制转向下一条语句.

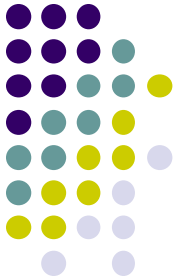
布尔表达式(1)



- <条件>是一个布尔表达式.
  - 结果为布尔值True或False
- 简单布尔表达式:

  <表达式1> <关系运算符> <表达式2>
  - 关系运算: <, <=, ==, >=, >, !=
    - 数值比较
    - 字符串比较: 按字典序.
      - 字符序由编码(ASCII等)决定. 如:大写字母在小写字母前.
    - 列表,元组的比较

例:一局乒乓球比赛的结束



- 双方任何人先得11分

```
a == 11 or b == 11
```

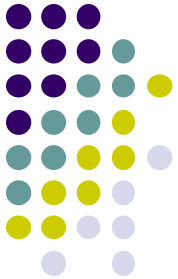
- 更准确的:一方至少要多2分才胜

```
(a >= 11 and a - b >= 2) or
(b >= 11 and b - a >= 2)
```

或者写成

```
(a >= 11 or b >= 11) and abs(a - b) >= 2
```

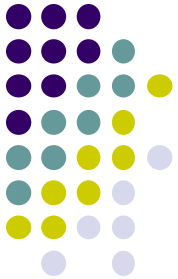
编程实例



- 温度转换程序:eg3_2.py
  - 增加热浪告警功能

```
f = input("Temperature in degrees Farenheit: ")
c = (f - 32) * 5.0 / 9
print "Temperature in degrees Celsius:",c
if c > 35:
    print "Warning: Heat Wave!"
```

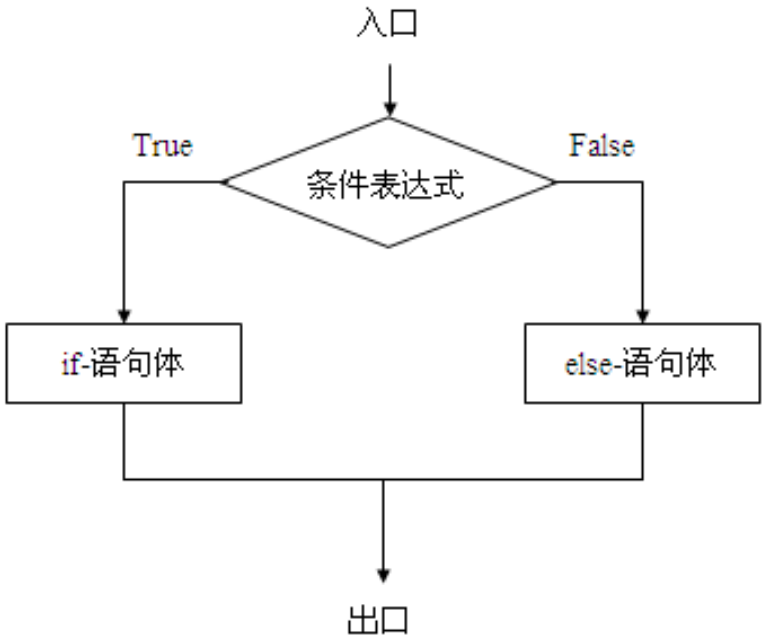
编程实例



- 温度转换程序:eg3_3.py
  - 增加热浪和寒潮告警功能

```
f = input("Temperature in degrees Farenheit: ")
c = (f - 32) * 5.0 / 9
print "Temperature in degrees Celsius:",c
if c >= 35:
    print "Warning: Heat Wave!"
if c <= -6:
    print "Warning: Cold Wave!"
```

两路分支结构



- 语法

  if <条件>:

  <if-语句体>

  else:

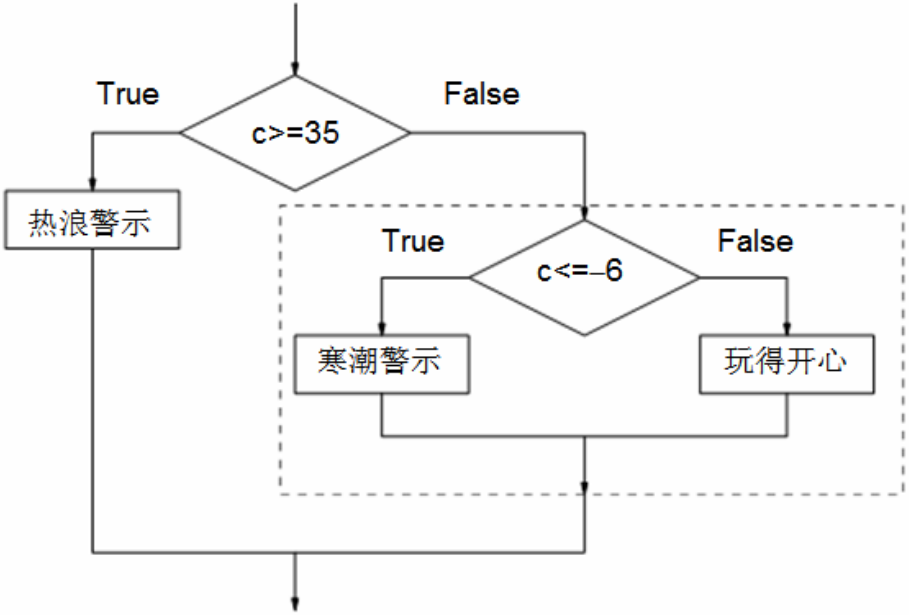
  <else-语句体>

- **if和else是非此即彼的关系.**
  - 下列语句对吗?

```
if c >= 35:
    print "Warning: Heat Wave!"
else:
    print "Warning: Cold Wave!"
```

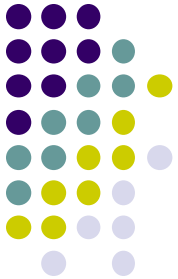
多路分支:嵌套if-else

- if语句可以嵌套
  - 多重嵌套不好
    - 难读
    - 代码松散




```
if c >= 35:
    print "Warning: Heat Wave!"
else:
    if c <= -6:
        print "Warning: Cold Wave!"
    else:
        print "Have fun!"
```

多路分支:if-elif-else结构

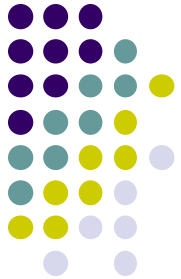


- 语法

```
if <条件1>:
    <情形1语句体>
elif <条件2>:
    <情形2语句体>
...
elif <条件n>
    <情形n语句体>
else <其他情形语句体>
```

- 语义:找到第一个为真的条件并执行对应语句序列,控制转向下一条语句;若无,则执行else下的语句序列,控制转向下一条语句.

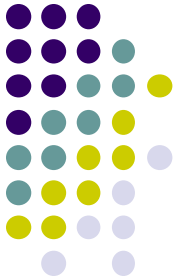
编程实例



- 温度转换程序**eg3_4.py**

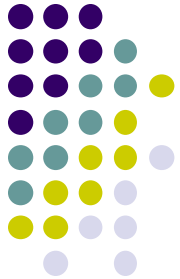
```
f = input("Temperature in degrees Farenheit: ")
c = (f - 32) * 5.0 / 9
print "Temperature in degrees Celsius:", c
if c >= 35:
    print "Warning: Heat Wave!"
elif c <= -6:
    print "Warning: Cold Wave!"
else:
    print "Have fun!"
```

第3章 数据处理的流程控制



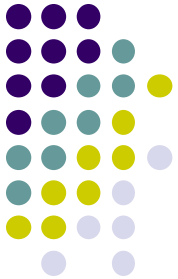
- 顺序控制结构
- 分支控制结构
- 异常处理
- 循环控制结构
- 结构化程序设计

程序运行错误的处理



- 程序编译正确,但在运行时发生错误.
  - 例如:a/b语法没错,但运行时万一b=0,就会出错
  - 又如:输入数据的类型和个数不对,列表索引越界,等等
- 编程时如果没有考虑运行错误,程序就很容易运行崩溃,非正常结束.
- 好的程序应该是健壮的.

编程实例

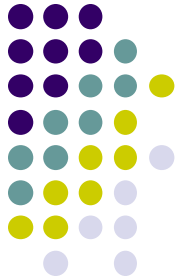


- 求一元二次方程根:eg3_5.py

```
import math
a, b, c = input("Enter (a, b, c): ")
discRoot = math.sqrt(b*b - 4*a*c)
root1 = (-b + discRoot) / (2*a)
root2 = (-b - discRoot) / (2*a)
print "The solutions are:", root1, root2
```

  - 运行程序,输入1,2,3
  - 程序崩溃!

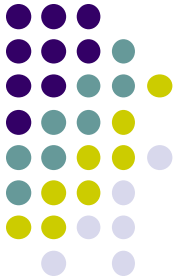
提高健壮性:使用错误检测代码



- 错误检测代码:利用if判断是否发生了某种运行错误.

```
do_sth()
if some-error:
    do_sth_else()
```

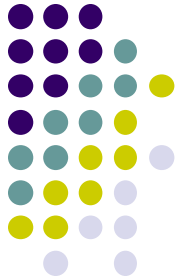
编程实例



- 解方程程序的改进:**eg3_6.py**

```
import math
a, b, c = input("Enter (a, b, c): ")
discrim =  b * b - 4 * a * c
if discrim >= 0:
    discRoot = math.sqrt(discrim)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)
    print "The solutions are:", root1, root2
else:
    print "The equation has no real roots!"
```

提高健壮性:利用函数返回码



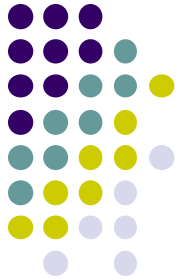
- 函数中有检测代码,执行正常与否可利用返回值作为标志码.
- 调用者无条件调用函数,并检测返回值.
  - 例如,为了解决sqrt函数的问题,设计robustSqrt():

```
def robustSqrt(x):
    if x < 0:
        return -1
    else:
        return math.sqrt(x)
```

  - 则程序中可以这样检测

```
if robustSqrt(b*b-4*a*c) < 0:...
```

异常处理



- 错误检测代码的缺点:当程序中大量充斥着错误检测代码时,解决问题的算法反而不明显了.

```
x = doOneThing()
if x == ERROR:
    异常处理代码
......
```

或写成:

```
if doOneThing() == ERROR:
    异常处理代码
......
```

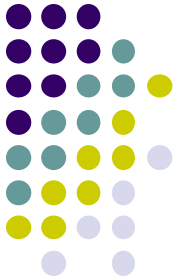
算法清晰但不健壮:

```
doStep1()
doStep2()
doStep3()
```

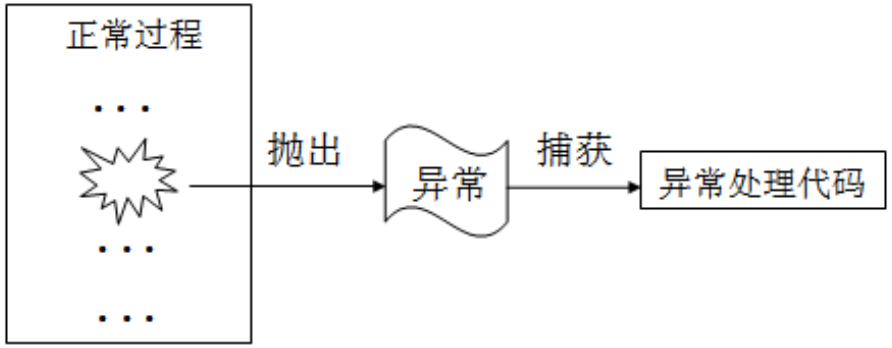
健壮但算法不清晰:

```
if doStep1() == ERROR:
    错误处理代码1
elif doStep2() == ERROR:
    错误处理代码2
elif doStep3() == ERROR:
    错误处理代码3
```

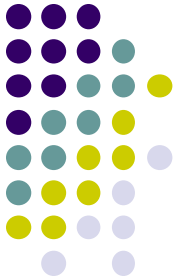
异常处理



- 能否既健壮,又不破坏原来算法的清晰?
- 异常处理机制
  - 程序运行时如果出错则"抛出"一个"异常";
  - 程序员能编写代码"捕获"并处理异常;
  - 可使程序不因运行错误而崩溃,尽量使用户不受意外结果的困扰.



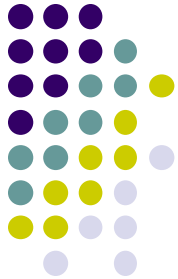
Python的缺省异常处理



- 程序运行出错时,抛出的异常被Python系统自动处理—基本上就是中止程序的执行并显示一些错误信息.

```
>>> a = "Hello"
>>> print a[5]
Traceback (most recent call last):  File
  "<stdin>", line 1, in <module>
IndexError: string index out of range
```

程序员自定义异常处理

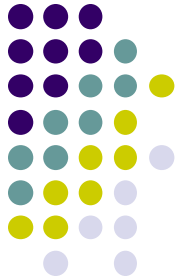


- **Python提供try-except语句,可用来自定义异常处理代码.**

```
>>> a = "Hello"
>>> try:
        print a[5]
    except IndexError:
        print "Index wrong!"

Index wrong!
```

异常处理机制的优点



- 既保持核心算法的清晰,又能提高程序的健壮性.

算法清晰但不健壮:

```
doStep1()
doStep2()
doStep3()
```

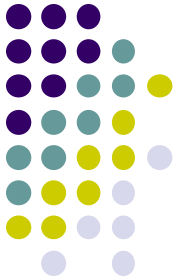
健壮但算法不清晰:

```
if doStep1() == ERROR：
    错误处理代码1
elif doStep2() == ERROR:
    错误处理代码2
elif doStep3() == ERROR:
    错误处理代码3
```

算法清晰且健壮:

```
try:
    doStep1()
    doStep2()
    doStep3()
except ERROR:
    错误处理代码
```

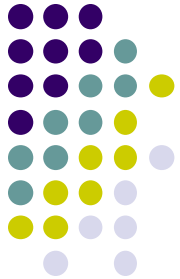
异常处理语句



- 可以对不同类型的错误分别指定处理代码

```
try:
        <正常程序体>
except <错误类型1>:
        <异常处理程序1>
...
except <错误类型n>:
        <异常处理程序n>
except:
        <其他异常的处理程序>
```

编程实例

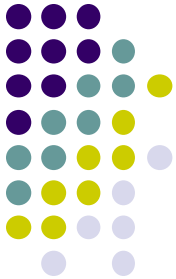


- 解方程程序的改进:用异常处理语句来捕获**math.sqrt**的溢出错误.**(eg3_7.py)**

```
import math
try:
    a, b, c = input("Enter (a, b, c): ")
    discRoot = math.sqrt( b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)
    print "The solutions are:", root1, root2
except ValueError:
    print "The equation has no real roots!"
```

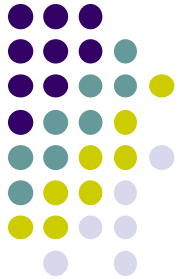
  - 更完善的版本:**eg3_8.py**

第3章 数据处理的流程控制



- 顺序控制结构
- 分支控制结构
- 异常处理
- 循环控制结构
- 结构化程序设计

为什么需要循环?

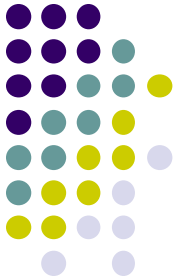


- 有时需要重复做相同或相似的事情,程序中如何表达?
  - 例如:在屏幕上显示1~5

```
print 1
print 2
print 3
print 4
print 5
```

  - 繁琐且不具有扩展性(显示1~10000怎么办?)
- 循环:用很少的语句表达重复执行的很多语句.

for循环



- 语法

  for <循环控制变量> in <序列>:

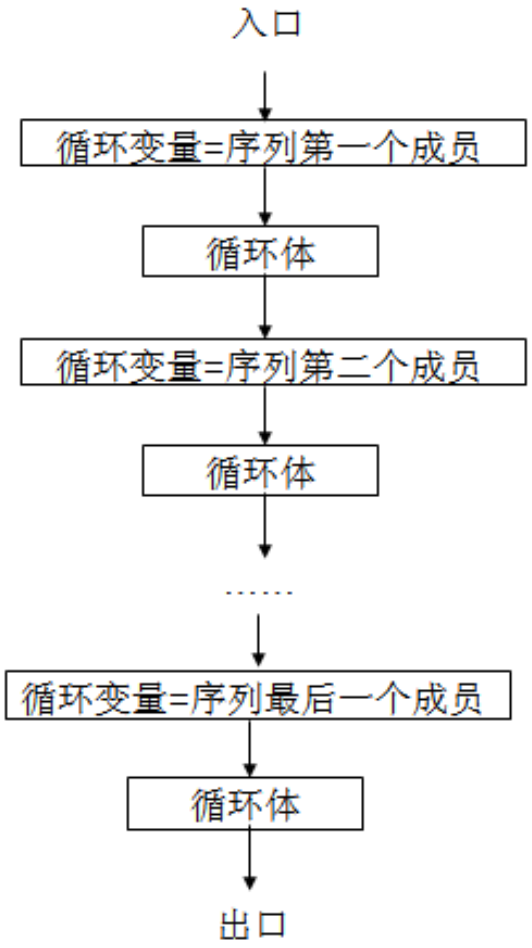
  <循环体>

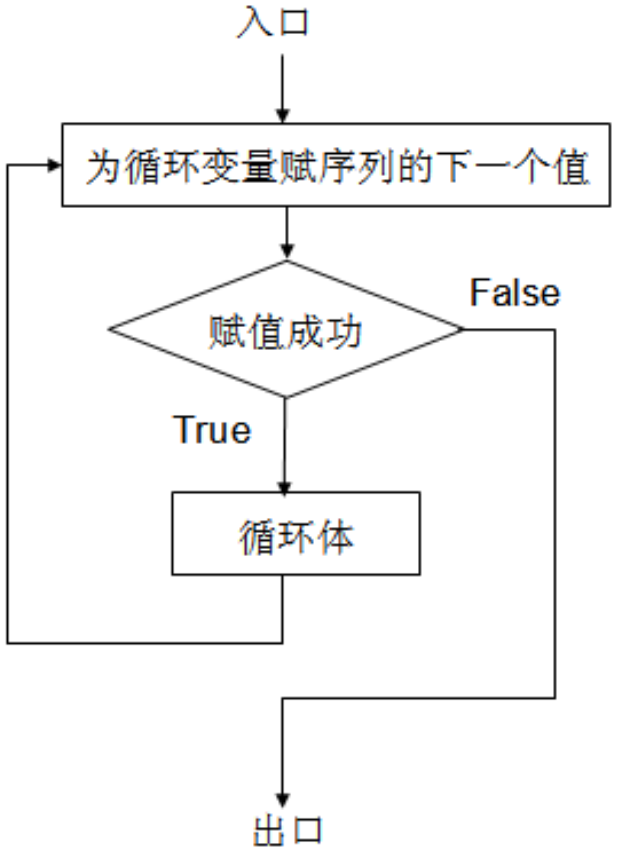
- 语义:令<循环控制变量>取遍<序列>中的每个值,并对变量所取的每个值执行一遍循环体.
- 例如:

```
for i in [1,2,3,4,5]:
    print i
```

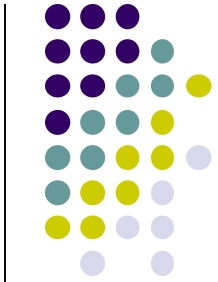
  - 显示1~10000:用[1,2,...,10000]显然不合适,可以用range()

for循环的流程图


入口
循环变量=序列第一个成员
循环体
循环变量=序列第二个成员
循环体
……
循环变量=序列最后一个成员
循环体
出口


入口
为循环变量赋序列的下一个值
赋值成功
False
True
循环体
出口

for语句中序列的作用



- 计数器:序列只是用来控制循环的次数.

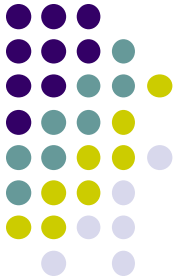
```
for i in range(10):
    print "烦"
```

  - 循环体不引用循环变量.
- 数据:序列本身是循环体处理的数据.

```
for i in range(10):
    print i*i
```

  - 循环体引用循环变量.
  - 两种遍历方式(见下一片)

用for处理序列数据



- 直接遍历序列

```
>>> data = ['Born on:','July',2,2005]
>>> for d in data:
        print d,
```

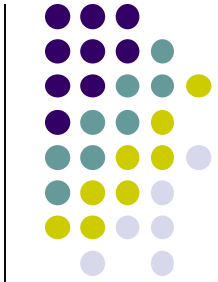
- 通过索引遍历序列

```
>>> data = ['Born on:','July',2,2005]
>>> for i in range(len(data)):
        print data[i],
```

  - 可以更灵活地处理序列数据,如

```
>>> for i in range(0,len(data),3): ...
```

用for处理各种序列数据



- 字符串

```
>>> for c in "hello world":
        print c
```

- 元组

```
>>> for i in (1,2,3):
        print i
```

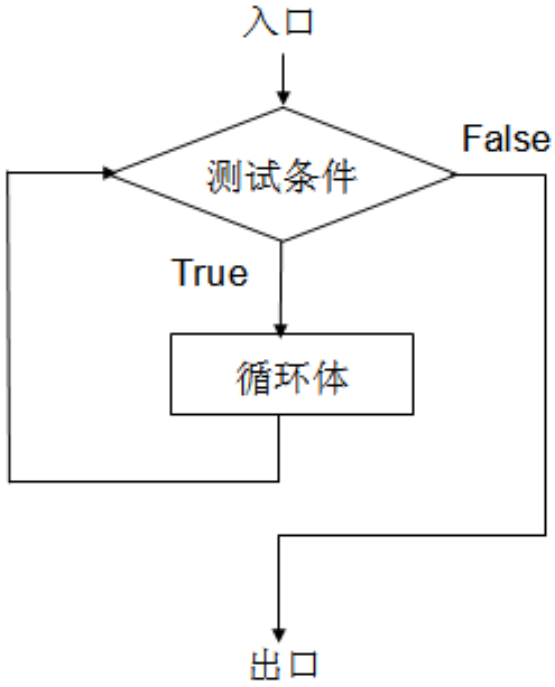
- 嵌套序列:如元组的列表

```
>>> for t in [(1,2),(3,4),(5,6)]:
        print t,t[0],t[1]
```

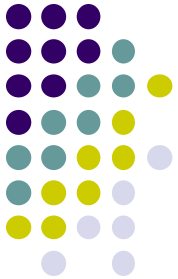
while循环

- for主要用于确定次数的循环
- 不确定次数的循环:while

```
while <布尔表达式>:
        <循环体>
```



while循环的特点

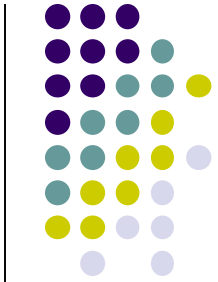


- 循环前测试条件
  - 若不满足,则循环体一次都不执行
- 循环体必须影响下一次条件测试!
  - 否则导致无穷循环
  - 例如:for循环改写成while循环

```
i = 0
while i < 10:
    print i
    i = i + 1
```

  - 若忘了最后一条语句会怎样?

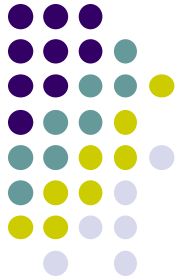
常用循环模式:交互循环



- 根据用户交互来决定是否循环下去
- 例:求和**(eg3_10.py)**

```
sum = 0
moredata = "yes"
while moredata[0] == "y":
    x = input("Input a number: ")
    sum = sum + x
    moredata = raw_input("More?(yes/no)")
print "The sum is", sum
```

常用循环模式:哨兵循环



- 交互循环不断要用户输入moredata,很烦人.
- 改进:设置一个特殊数据值(称为哨兵)作为终止循环的信号.
  - 对哨兵唯一的要求就是能与普通数据区分
- 算法模式:

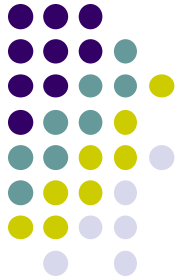
  前导输入

  while 该数据不是哨兵:

  处理该数据

  循环尾输入(下一个数据)

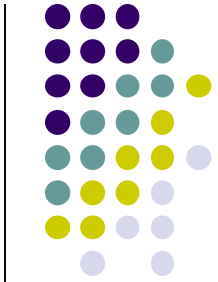
哨兵循环例(1)



- 正常数据是非负数,则可以-1作为哨兵: **eg3_11.py**

```
sum = 0
x = input("Input a number (-1 to quit): ")
while x >= 0:
    sum = sum + x
    x = input("Input a number (-1 to quit): ")
print "The sum is", sum
```

哨兵循环例(2)



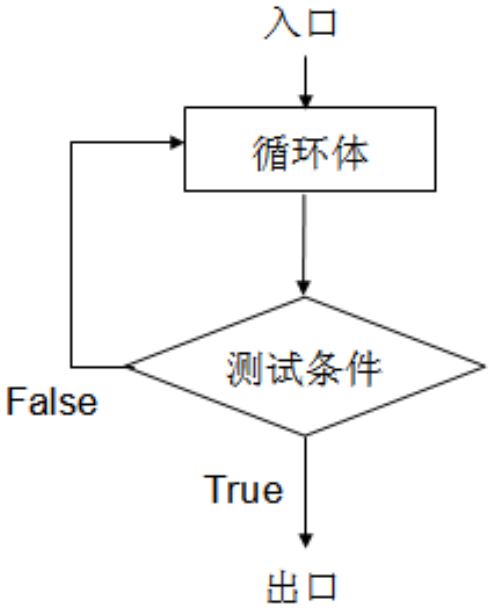
- 正常数据是任何实数,则可以空串作为哨兵:
  eg3_12.py

```
sum = 0
x = raw_input("Input a number (<Enter> to quit): ")
while x != "":
    sum = sum + eval(x)
    x = raw_input("Input a number (<Enter> to quit): ")
print "The sum is", sum
```

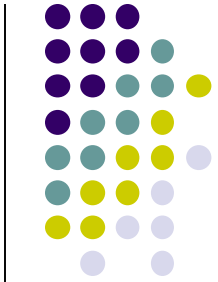
常用循环模式:后测试循环

- 输入验证问题:
  - 检查用户输入是否符合要求,不符合就要求用户重新输入,直至符合为止.
- 这是一种后测试循环:
  - 执行循环体后才测试条件
  - 循环体至少执行一次
  - 直至条件成立才退出循环
  - 有些语言提供repeat...until语句
- Python可用while实现
  - 只需确保首次进入while时条件成立

```
x = -1
while x < 0: ...
```



常用循环模式:while计数器循环



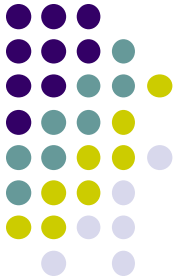
- 用while实现计数器循环

```
计数器count置为0
while count < n:
    处理代码
    count = count + 1
```

```
i = 0
while i<10:
    print i
    i = i+1
```

```
i = 10
while i>0:
    print i
    i = i-1
```

循环非正常中断:break



- 中止本轮循环,结束break所处循环语句.
  - 常与while True形式的无穷循环配合使用
- 例1:输入合法性检查

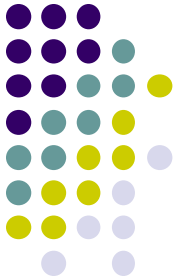
```
while True:
    x = input("请输入非负数:")
    if x >= 0: break
```

- 例2:break也可以跳出for循环

```
for i in range(10):
    print "烦"
    if i > 4: break
```

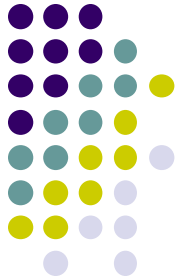
- 慎用break!尤其是一个循环体中有多个break.

循环非正常中断:continue



- 中止本轮循环,控制转移到所处循环语句的开头"继续"下一轮循环.
- 例:对列表中的奇数求和

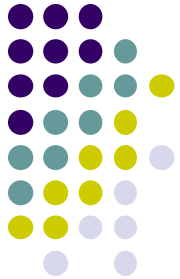
```
a = [23,28,39,44,50,67,99]
sum = 0
for i in a:
    if a % 2 == 0:
        continue
    sum = sum + i
print sum
```



常用循环模式:嵌套循环

- 一个循环的循环体中有另一个循环.
- 如果序列的成员本身又是序列,就需要嵌套循环来处理.
  - 数学中向量是一维序列,矩阵是二维序列
- 用嵌套循环遍历矩阵元素:

```
a = [[11,12,13,14],[21,22,23,24],[31,32,33,34]]
sum = 0
for i in a:
    for j in i:
        sum = sum + j
print sum
```



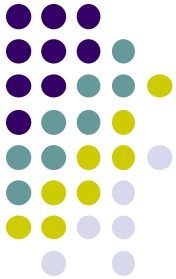
嵌套循环例

- 打印乘法口诀表
  - 关键是输出的排列

```
>>> for i in range(1,10):
        for j in range(1,i+1):
            print "%dx%d=%-2d" % (j,i,j*i),
        print
```

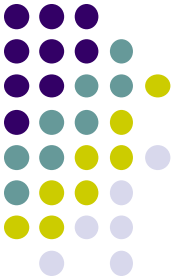
```
1x1=1
1x2=2  2x2=4
1x3=3  2x3=6  3x3=9
1x4=4  2x4=8  3x4=12 4x4=16
1x5=5  2x5=10 3x5=15 4x5=20 5x5=25
1x6=6  2x6=12 3x6=18 4x6=24 5x6=30 6x6=36
1x7=7  2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49
1x8=8  2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64
1x9=9  2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

第3章 数据处理的流程控制



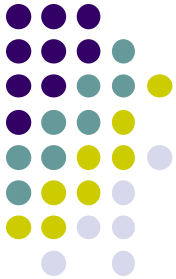
- 顺序控制结构
- 分支控制结构
- 异常处理
- 循环控制结构
- 结构化程序设计

程序设计的发展



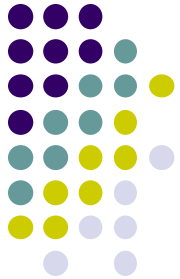
- 早期:手工作坊式
  - 程序规模小,功能简单
  - 要在有限内存中尽快完成计算
  - 凭借程序员的个人编程技巧
- 后来:作为工程来开发
  - 程序规模大,功能复杂
  - 内存和速度不是问题,软件正确性和开发效率变得突出
  - 依靠系统化的开发方法和工具

程序开发周期



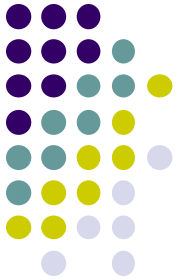
- 明确需求:问题是什么?用户要求是什么?
- 制定程序规格:描述“做什么”.
- 设计程序逻辑:描述“怎么做”.
- 实现:用编程语言编写代码.
- 测试与排错:用样本数据执行程序,测试结果是否与预期吻合.有错则排错.
- 维护:根据用户需求持续改进程序.

什么是好的程序



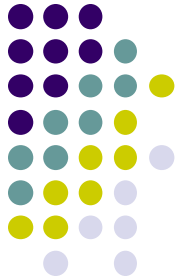
- 解决同一个问题,可以设计出多种处理过程,即编制多种程序.
- 即使各种程序都正确,仍然有好坏之分.
- 除了正确性,好的程序应该是:
  - 效率高
  - 易理解
  - 易维护
  - 可扩展

如何得到好的程序



- 手工作坊阶段靠的是个人编程技巧
- 如今则依靠程序设计方法和工具
  - 方法:结构化方法,模块化方法,面向对象方法等
  - 工具:建模工具,集成开发环境,项目管理工具等

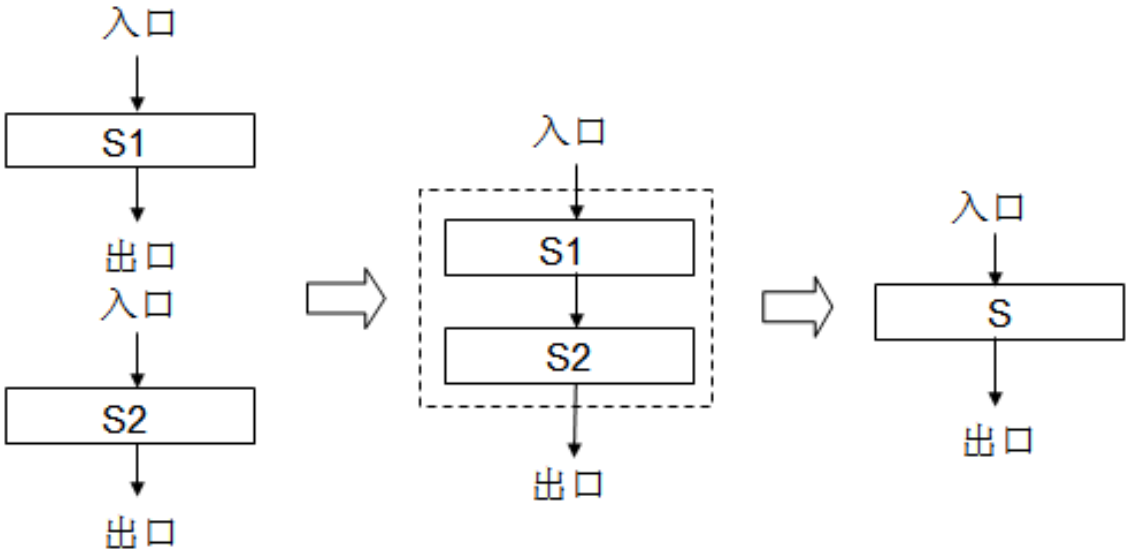
结构化程序设计(1)



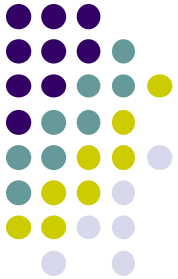
- 确保程序具有良好的文本结构, 使程序易理解, 易验证,易维护.
- 基本原则
  - 只使用顺序,分支,循环三种基本控制结构
  - goto有害
    - 好在goto不是必须的
    - break和continue有点类似goto,因此要慎用

结构化程序设计(2)

- 单入口单出口的程序块
  - 多条语句可以组合成程序块,只要是单入口单出口,仍然可当作一条语句.



编程案例:求最大值(1)



- 先考虑求三个数x1,x2,x3的最大值的问题
- 策略1:每个数都与其他数比较大小

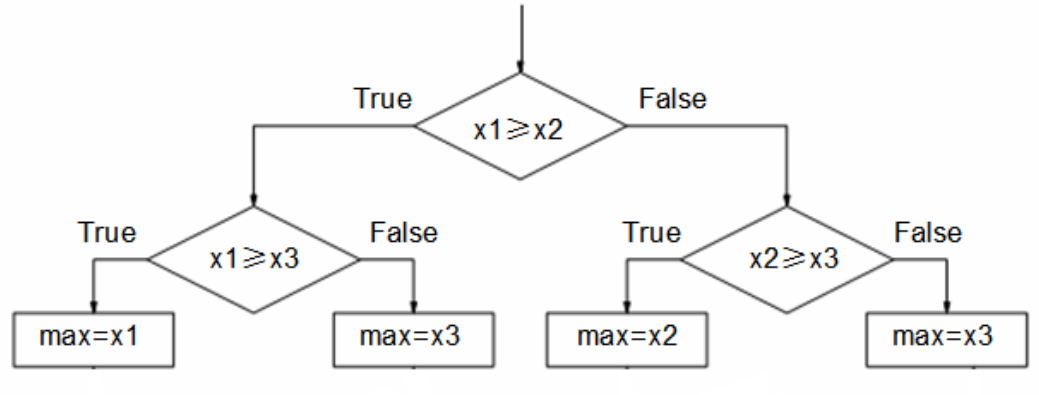
```
if x1 >= x2 and x1 >= x3:
    max = x1
elif x2 >= x1 and x2 >= x3:
    max = x2
else:
    max = x3
```

  - 此算法中各分支彼此独立.但实际上一个分支的信息对其他分支是有用的!
  - 此算法不适合较大n的情况

编程案例:求最大值(2)

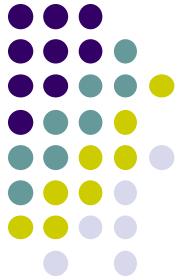
- 策略2:判定树

```
if x1 >= x2:
    if x1 >= x3:
        max = x1
    else:
        max = x3
else:
    if x2 >= x3:
        max = x2
else:
    max = x3
```



好处:只需两次比较,效率高.
坏处:结构复杂, 复杂度随n爆炸式增长.

编程案例:求最大值(3)



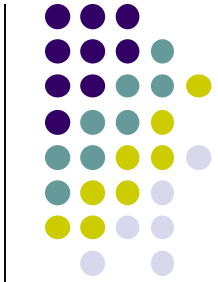
- 策略3:顺序处理,记录当前最大值.

```
max = x1
if x2 >= max:
    max = x2
if x3 >= max:
    max = x3
```

  - 效率高
  - 易读易理解
  - 可扩展到n

```
n = input("How many numbers? ")
max = input("Input a number: ")
for i in range(n-1):
    x = input("Input a number: ")
    if x > max:
        max = x
print "max =", max
```

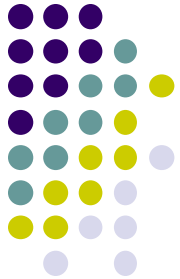
编程案例:求最大值(4)



- 策略4:利用现成代码.
  - Python提供内建函数max(x_1, x_2, ... , x_n)

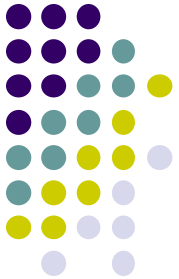
```
>>> x1,x2,x3 = input("enter 3 numbers:")
>>> print "max =", max(x1,x2,x3)
```

小结

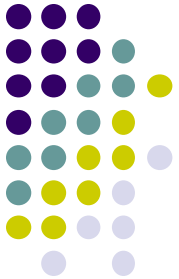


- 给定问题,有多种解决方法
  - 不要只凭第一感匆忙编程,要多考虑是否有更好算法;
  - 首先要正确,其次要结构清晰,高效,可扩展,漂亮.
- 问自己会如何解决问题
- 追求一般性
- 借鉴,重用现成算法

assign4-



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



End