循环和程序

- 循环程序的问题
- 典型的循环案例
  - 重复
  - 累积
  - 递推
  - 输入控制的循环
- 循环的终止性
- 计算和环境

重复操作和循环程序

- 本次课首先集中讨论循环程序的的设计和实现
  - 直线型代码很容易写
  - 分支代码也比较简单
  - 开始学习编程，最重要问题之一是掌握写循环的技术
- 为什么写循环？有很多情况。典型如：
  - 有一批类似数据，需要按同样方式处理
  - 需要以同样方式计算出一批结果
  - 需要反复累积一些（按同样方式算出或获得的）数据
  - 需要按同样方式反复从已有数据推算下一个（下一批）数据
  - ……

重复操作和循环程序

- 需要循环的一些情况
  - 需要多次做类似操作而且次数较多，适合用循环
    - 用循环描述，通常能缩短程序
    - 用一段代码描述共用操作，容易检查/维护/修改
  - 需要重复操作，操作次数无法事先确定，结束条件由循环中数据变化的情况决定，必须用循环
  - 决定重复的因素来自函数参数或输入，也必须用循环描述
- 例如：
  - 生成华氏到摄氏温度的对照表，典型的第一种情况
  - 通过迭代改进求平方根，典型的第二种情况（函数定义）

循环程序

- 写循环的第一步是看到计算中需要重复操作，而且有规律可循
- 写循环需要考虑的一些问题，基础是需要做什么
  - 为完成循环计算需要引进哪些变量？怎样控制循环？
  - 循环开始前它们应该取什么值？
  - 循环体中（一次迭代计算中）它们的值应该如何变化？
  - 在什么条件下结束（或继续）循环？
  - 循环结束后怎么得到所需结果？
- 具体问题是选择哪种循环结构？
  - 循环的次数和方式清晰，有可能通过一个循环变量和一个迭代器（如 **range**）控制，用 **for** 语句更简单清晰
  - 不能确知循环的次数，循环方式复杂，就必须用 **while**

循环程序

- 每个循环都可能用多种不同方式描述
- 例：求 **13** 到 **26** 的整数之和
  - 应该采用最简单，最清晰的描述方式
  - 尽可能用 **for** 语句和向上循环（循环变量值递增）
- 浮点数与循环控制（计算有误差）
  - 尽可能不用浮点数控制循环
  - 绝不用浮点数作为控制循环范围的数值依据
- 注意 **Python** 里整数范围的描述和意义
  - 说“从 **m** 到 **n**”总指 **m, ..., n-1**，即整数区间 **[m, n)**
  - **range(n)**，**range(m, n)**，**range(m, n, d)** 都是
- 下面考虑几种典型的循环情况，仅供参考

重复工作

- 采用循环的最简单情况：需要重复做一批类似但相互无关的工作
  - 对一系列数据做相同的计算，分别得到结果
  - 反复输出一批数据，……
  - 例如前面的温度转换，或者作业里的生成表格
- 关键：
  - 计算或操作有统一模式，可以用一段代码描述
  - 不同计算之间从差异就是一个或几个变量的取值，而这些取值可以按一定的规律产生出来
- 重复工作
  - 识别和描述比较简单
  - 关键是总结共同计算模式，确定循环中变量取值的变化规律

累积

- 累积工作的特点，是在重复性工作中
  - 需要用一个或几个变量去累积循环中得到的数据
  - 每次迭代把一些数据的信息“记入”累积变量中
  - 它（它们）的值是循环的主要结果
- 累积常用的操作如 + 或 *，

  经常能用到扩展赋值运算符 +=，*= 等
- 累积程序的实例
  - 求整数 1 到 100 的立方和
  - 数项级数前 n 项的计算，如

$$\ln 2 = \sum_{n=1}^{\infty} (-1)^{n-1} \cdot \frac{1}{n}$$

累积（有条件）

- 循环中的累积也可能还有条件。一般情况是，
  - 通过某种统一方式枚举（生成、计算）出一些数据
  - 如果满足条件就将其“记入”累积数据
- 简单实例：
  - 求 100 以内 7 的倍数中不能整除 3 的整数之和
  - 输入的 10 个整数中的偶数之和
- 这是一种典型的计算模式，称为“生成和筛选”
  - 生成：产生出一批候选数据
  - 筛选：从生成数据中选出满足某些条件的合格数据
  - 有时难以直接生成所需要的数据（序列），可以考虑生成更多数据，从中选出所需数据

递推

- 递推是在循环的每次迭代中，基于某个（些）变量的当前值，按某种方式算出其下一个值。常见描述形式

  d = *f*(d, ...)

  d 的下个值需要通过 d 和其他数据，通过某种方法算出

- 求 x 平方根程序是典型的递推，从一个近似值求出下一近似值
  - 数学定义：x 的平方根就是满足 y*y = x 的非负数 y
  - 计算的说明：
    1) 任取 z
    2) 求出 y = x/z
    3) 如 y*y 足够接近 x，那么 y 就是 x 平方根的近似值
    4) 取 z = (z + y)/2，回到 2) 重复

例：求 sin 函数的值（通项计算和递推）

- 已知

$$\sin(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

考虑利用这个公式求 **sin(x)** 的近似值

用项的绝对值小于 **1e-6** 作为结束条件

- 一种朴素想法是

定义函数 **term(x, n)** 计算级数的项，再用一个主循环

```
while True :
    t = term(x, n)
    if fabs(t) < 1e-6 :  结束循环
    s += t
    n += 1
```

考虑 **term** 的定义

考虑**sin**的定义

也可以写成一个函数定义，其中用嵌套循环

例：求 sin 函数的值（通项计算和递推）

- 仔细观察不难看到

$$t_n = -\frac{x^2}{2n \cdot (2n+1)} \cdot t_{n-1}$$

  $t_0 = x$。利用这个递推公式，可以节省大量计算

  可以定义项递推函数 **nextTerm(x, n, t)**，函数参数 t 表示 t_{n-1}。采用与前面类似的主循环结构

  在代码文件里直接写出计算，没有另外定义函数

- 请分析这个函数的计算过程和进一步改进：
  - 大的参数值会带来很大的计算误差

    自己做试验，并想想为什么
  - 可以用 **fmod(x, pi)** 把参数归化到最小 **[0.0, π)**

输入和循环

- 一类常见循环是输入循环
  - 循环中从程序外部获得一批数据用于计算
  - 得到足够数据后结束循环，继续下步工作
- 控制循环的进行和结束？几种典型情况
  - 编程时知道需要的数据项数：程序内部控制，这是简单的“循环输入”
  - 由程序外部控制的输入循环（输入控制的循环）
    - 与程序用户（外部）约定循环结束的信号（特殊输入）
    - 在程序里检查输入情况，按约定结束循环
    - 具体约定要根据情况设计

输入控制的循环

- 例（第一种情况）：读入用户提供的 10 个实数，计算出它们的平均值
- 例：读入用户提供的一系列数，求出它们的平方和
- 新问题：
  - 程序用户可以输入任意多个数
  - 写程序时无法知道用户的输入结束意图
- 只能约定结束方式（建立与用户的协议），例如
  - 约定输入 0 就是结束
    - 这个约定导致“正式数据”里不能存在 0
  - 约定某个不是数的输入作为结束
  - 注意：这种约定决定了程序的写法和用法

终止性和循环程序

- 写程序时，需要考虑程序是否必定终止（对所有可能的输入都终止）。不终止的程序不能给出最后结果
- 直线型和分支程序必然终止（只要每个基本语句终止）

  循环程序则可能不终止（即使每个基本语句都终止）
- 有些程序可能需要运行很长时间

  例：调和级数部分和，计算这个级数的前多少项之和能超过某个给定的数
- 有关终止性的理论结果：程序终止性问题是不可判定的
  - 不存在判断任何程序对任何输入是否终止的有效方法
  - 直观说，就是无法写出一个软件，判断任意的一个程序对任意一个输入是否终止

循环程序和终止性

- 例（**Collatz conjecture**，猜想下面函数对每个 **n** 终止）：

```
def collatz(n):
    while n != 1:
        if n%2 == 0 : # n is even
            n = n//2
        else:             # n is odd
            n = n * 3 + 1
```

http://en.wikipedia.org/wiki/Collatz_conjecture

可检查它迭代多少次，或达到的最大数值等

- 递归函数（递归程序）存在类似的问题

  其终止性同样是不可判定的

Python 的若干问题

- 串联赋值
  - 可以用一个表达式的结果给多个变量赋值
  - 形式：

    变量 = ... = 变量 = 表达式
- 允许在一行里写几个语句
  - 形式：语句之间用分号分隔
  - 语义：按顺序逐个执行这些语句
  - 语用（情况和习惯用法）
    - 几个短语句可以写在一行，缩短代码的行数
    - **Python** 的习惯是一行只写一个语句
    - 课堂演示，如果程序很长，可能用这种形式，不是提倡

Python 的若干机制

- 检查类型
  - 前面用 **type(表达式) == 类型名**
  - 标准的做法是用 **isinstance(表达式, 类型名)**
- 函数的文档串
  - **Python** 的一种习惯是在函数定义的 **def** 子句之后写一个“文档串”，描述函数的使用方式和意义
  - 采用“三引号括号”形式的字符串
    - 文档串可以跨越若干行，具有较为可读的形式
    - 文档串对函数的执行没有任何影响

环境和状态

- 我们写程序，是为了指挥计算机运行，完成某种特定计算，得到所需的结果或产生所需的效果
- 前面讨论了（Python 的）一些基本编程机制

  为更好理解程序的行为，还需要理解一些概念

  下面介绍两个重要概念：环境和状态
- 变量是编程语言（Python）里表示存储（memory）的机制
  - 程序中可以给变量赋值，然后可以通过变量使用所赋的值
    - 赋值建立变量与计算出的对象的关联关系（约束）
    - 可以再次赋值，改变已有约束，建立新约束关系
  - 说明：所用系统有存储信息的能力

    系统需要记录这些变量，记录变量当前值（不记录历史）

环境和状态

- 在程序运行时，存在着一个环境，程序运行于该环境中
- 环境是理解程序行为（运行）的重要概念：
  - 环境记录着当前有意义的一组名字（变量）及其意义（变量的值），记录“名字→值”约束关系。值是对象
  - 任何时刻环境里记录的约束的全体形成当时的状态
  - 程序的表达式里可能包含变量，变量的值由环境获得
  - 语句的执行效果依赖于环境，还可能改变环境
- Python 系统启动时，建立起一个初始环境

  print, input 等及其意义在初始环境里有定义（称为内置的，build in），详情见 Python 标准库手册

  还有其他预定义“名字-对象”约束

环境/状态和计算

- 环境为程序的执行提供信息，（程序描述的）计算在环境里进行
  - 需要用一个变量的值时，到环境里去找

    找到了，取出相应的约束值使用

    找不到，报告变量无定义错误
  - 使用函数等，情况类似
- 执行语句有可能改变环境，例如
  - 给变量赋值可能加入新约束对（如果原来没有）
  - 特殊命令 **del**
    - **del(**变量**, …)** 从环境里删除（列出的）变量（及其约束）
    - **del** 还有其他用途，后面有些介绍
    - 细节可以参考 **Python** 材料

环境/状态和计算

- 状态：在程序运行中的一个具体时刻，环境中所有有定义变量及其当时约束的总和
  - 状态决定表达式的值和语句的效果
  - 语句的执行可能改变环境的状态
  - 一个程序，在一次执行中将经历一系列状态
  - 计算的“结果”来自当时的状态
- 例：
  - **x = y + 3** 可能改变状态
  - **print(x, y)** 改变状态吗？
  - 说法：“这个函数不改变环境”，“这个操作改变了环境”
- 在一次计算（程序的一次运行）中，变量可能经历一系列值

环境和状态

- 给 x 赋值
  - x 不存在时，把它和相应的值约束加入环境
  - x 存在就改变其值

- 执行命令：

  pi = 3.1416

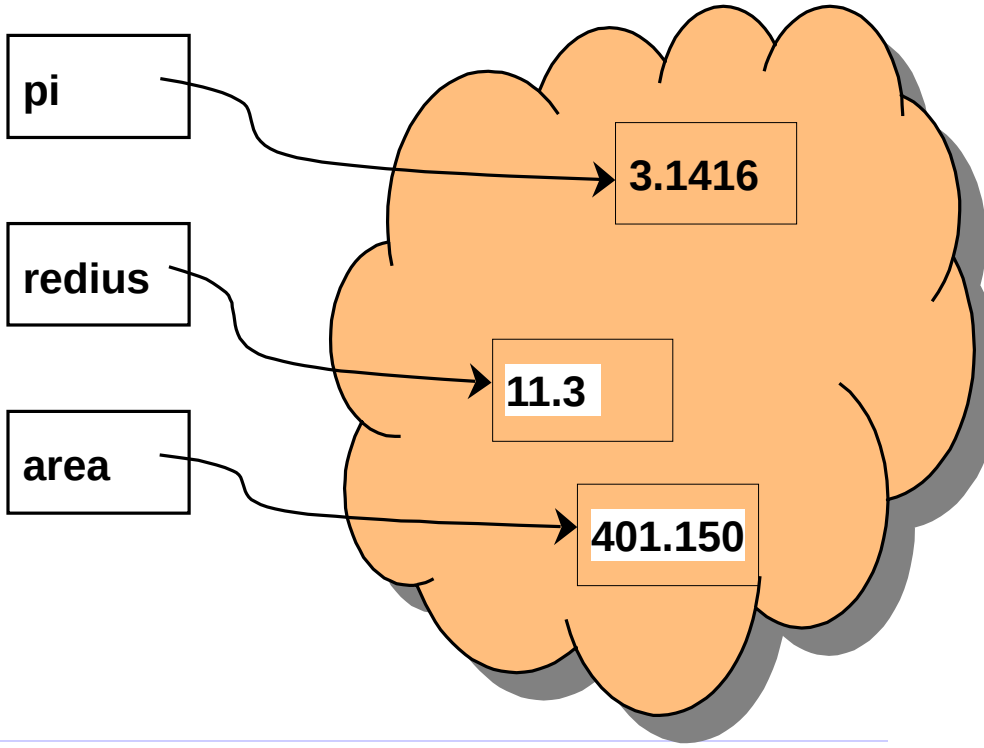
  redius = 2.58

  area = redius*redius*pi

- 再对变量赋值：

  redius = 11.3

  area = redius*redius*pi



环境/状态和计算

- 环境：
  - 当前有哪些可以用的（有意义的）名字？
  - 赋值语句和 **def** 等可能把新名字加入环境
  - 可以用 **del** 从环境中删除特定的变量
- 状态：
  - 在考虑变量的值时，考虑的是“状态”
  - 在环境中的变量没有增加或减少的情况下，程序执行中变量的值也可能变化
  - “当前状态”，指程序执行到某个位置时变量取值的全体
- 环境、状态是理解程序行为的基础，非常重要

执行模块，导入模块

- 运行一个模块（**run a module**）时，**Python**
  - 重新建立一个新的初始环境
  - 顺序处理模块中的一个个语句
  - 语句执行完毕时环境停在当时状态，回到交互方式
- 导入 模块（以 **math** 为例）
  - 执行“**import math**”将模块名 **math** 加入环境，载入该模块里定义的功能（函数等），将其约束在 **math** 模块里，可以通过 **math.xxx** “间接”使用（使用 **math** 模块里的 **xxx**）
  - 执行“**from math import ***”，把 **math** 模块里定义的所有功能（函数名，常量名如 **pi**，及其约束）直接加入环境
  - 执行“**from math import sin, cos, sqrt**”，把名字 **sin, cos, sqrt** 及其约束值（函数）加入在环境