函数和环境

- 通用和专用的方法，实例
- 局部函数定义
- 变量和作用域（全局和局部）
  - 嵌套的作用域
  - 名字和变量（函数）的约束关系
  - **global** 和 **nonlocal** 声明，变量查找
- 函数执行中的环境变化

通用和专用的方法

- 解决计算问题，存在一些通用的方法，针对具体问题也可能开发出一些具体的专用方法
- 前面讨论过几个计算中常用的通用方法
  - 确定一组候选，然后从中找出所需的解（一个解）

    关键：保证解位于候选集中，选择不能漏掉解
  - 生成和筛选：生成一批候选，从中筛选出所需的对象

    通常筛选出的是一组对象

    关键：生成机制，合用的筛选函数（谓词）

  下面讨论中还会介绍一些方法
- 下面通过一个具体实例说明情况

例：求立方根

- 希望找到一个接近其立方根的数，例如，要求得到的根的立方与原数的差不超过 **0.001**
- 一种简单的通用方法是用一系列数值做试验，选出最接近的值
  - 如果试验的数值足够密集，就可能得到足够好的解
  - 首先考虑按照 **0.001** 步长做试验
- 函数代码和一些细节：
  - 将负数求根归结到正数统一处理
  - **math** 包里的 **fabs(x)** 求绝对值
- 试验发现：用一定步长检查，未必能保证对所有可能数值找到的根都满足要求（对大些的数都找不到）

  反思计算方法：步长（精度）与计算时间，步长与结果

例：求立方根

- 用固定步长检查，很费时，而且不能很好满足需要
- 现在考虑一种逼近方法（另一种通用方法）：
  - 取一个包含解（立方根）的区间
  - 逐步缩小区间范围，而且保证解包含在其中
  - 区间足够小时，以其中点作为解的近似值
- 问题：如何缩小？

  任何能保证不丢掉解的方法都可以考虑
- 一种方法：二分区间后转到合适的半区间。“一尺之棰，日取其半，万世不绝”，但区间可以变得任意短，得到任意精度的解

  程序代码
- 考虑初始区间设置的问题，取 [0, y] 合适吗？

例：求立方根

- 通用方法具有广泛的适用性，但解决问题的效率相对较低
- 研究给出的立方根逼近公式

$$x_{n+1} = \frac{1}{3}(2x_n + \frac{x}{x_n^2})$$

用公式求立方根，达到精度（相对误差）

$$\left|(\frac{x_{n+1} - x_n}{x_n})\right| < 10^{-6}$$

- 定义函数
  - 从 x 开始迭代
  - 收敛性由理论保证
  - 需要前后两个迭代值，以便判断

专用和通用方法：比较

- 适用性：
  - 通用方法可能用于解决许多不同的问题
  - 专用方法只能用于特定的问题
- 效率：一般而言，专用方法的效率较高

  可以分析和试验
- 如果要解决一个具体问题，但一时找不到专用的特殊的算法，也可以考虑通用的方法
  - 计算机长于反复操作，可以在很短时间里做很多尝试
  - 试探法是最基本的计算方法
- 下面考虑函数的结果组织

例：求立方根（函数抽象）

- 以求立方根的函数为例
  - 把计算下一逼近值和判断结束条件的操作抽象出来
  - 定义 **improve** 和 **accept** 函数
  - 换一种逼近方法或判断标准，只需要修改函数定义（示例）
- 问题：
  - 从使用者的角度，他们实际上只关注 **cbrt**
  - **cbrt** 里使用的（）为它定义的）**improve** 和 **accept** 函数很特殊，其他人一般不会用到它们
  - 但这两个辅助函数也出现在全局环境里，很不好：
    - 定义其他函数和变量时必须顾及它们的名字，可能误改
- 在一个局部使用的东西（函数/变量），应尽可能在局部定义

例：求立方根（嵌套函数）

- Python 允许把函数定义其他函数的内部
  - 这种函数只在其定义所在的函数内部可用
  - 用于建立局部概念，保护局部信息
  - 提高程序可读性，更容易修改维护
- 避免过多的全局变量相互干扰
  - 避免由于重名无意中造成的错误
  - 这种错误很难检查
- 编程原则：信息应该尽可能的局部化

  应利用 Python 的程序结构，尽可能做好信息局部化，包括辅助函数的局部化定义
- 整理修改后的 cbrt 函数代码

Python 程序结构

- 一个完整的 Python 程序就是一个模块
  - 模块里可以定义一些模块层的变量和函数
  - 还可以有其他定义，后面介绍
  - 这些都模块层面的定义称为全局定义
- 在函数里可以有局部定义
  - 函数的参数是局部的，只能在本函数的内部用
  - 函数体里定义的函数也是局部的，其名字与函数的约束只在本函数体里面有效
  - 函数还可以有局部变量
- Python 还存在另外的具有局部定义成分的结构，后面讨论

  引进复杂的结构有实际需要，但也带来一些需要弄清楚的问题

变量和作用域

- 程序里可能出现许多名字（标识符），表示程序变量
  - 需要弄清哪个名字说的是哪个变量
  - 特别是弄清在一个程序里，哪些名字出现代表同一个变量，哪些虽然名字相同，但表示的是不同的变量
  - 朴素方法：保证整个程序里的每个名字只有一个意义
  - 一般而言这件事很难做到。例如，使用他人开发的模块，其中可能用到一些名字，而使用者并不清楚
- 从一个函数的角度看，变量的一些不同情况
  - 本函数的局部变量（包括本函数的参数）
  - 全局变量（在当前模块里定义，但定义在所有函数之外）
  - 本函数的外围函数定义的变量（如果本函数嵌套在其中）

变量，作用域，名字空间

- 作用域：决定名字和变量之间约束关系的程序结构。在 **Python** 里，一个作用域确定了一个名字空间（**namespace**）
  - 全局作用域：
    - 一个模块是一个全局作用域，在这个作用域里定义的是全局变量和函数，这些定义在整个模块的范围内有效
    - 执行时建立全局名字空间，全局名字及其约束值都在其中
  - 函数作用域
    - 一个函数体是一个作用域，在这里可以定义局部函数和变量，局部函数/变量和函数参数在本函数的体内部有效
    - 执行时建立一个局部名字空间，局部名字在其中有约束
- 函数定义可以出现在全局作用域里，也可以出现在其定义所在的函数形成的外围作用域里，形成作用域嵌套

变量和名字空间

- 基本原则：
  - 在不同作用域里定义的名字，即使同名也相互无关

    例如：两个不同函数可以有同名的参数，相互无关
  - 当作用域出现嵌套时，内层作用域里定义的名字约束将遮蔽外围作用域里同名的定义

    例如：函数 **fun** 里有参数取名 **print**（或给 **print** 赋值），在 **fun** 里系统内置函数 **print** 就不能用了，在函数外可用
  - 在一个名字空间里，一个标识符只有最多一个定义

    例如，不能同时有一个全局变量 **x** 和一个全局函数 **x**。给全局名字 **print** 赋值，将导致其原有约束丢失
- 需要弄清程序里出现的每个名字（的每个使用），实际表示（指称）的是在哪个作用域里定义的变量

变量和作用域

- 用 **def** 定义的函数（显式定义，情况很清楚）

  def 以明确的方式，在其所在的作用域（对应名字空间）里定义（建立）给定的函数名，并使其约束于相应的函数对象

- 函数的形式参数（在函数头部显式描述，情况很清楚）

  在函数体的局部名字空间里定义具有这些参数名的局部变量

  形参也是本函数的局部变量，其初值来自实参，可重新赋值

- 有关变量的规定（**Python** 里的变量无须说明，隐式定义）

  Python 的默认规定：赋值即定义

  在一个作用域（全局或局部）里给一个变量赋值，就在相应名字空间里定义了这个变量（如果没有其他说明）

  for 的循环变量看作赋值（按默认规定），**for** 语句执行结束后该变量仍存在，其值为循环中的最后取值

变量和作用域：例

- 一个例子：

```
x = 1
def f () :
    y = x   # 将报“变量没定义错误”
    x = 2
    return x + y
```

- 原因：
  - 在一个作用域里，一个名字至多有一个定义（基本原则）

    在 f 的函数体里，x 只有一个定义
  - 赋值即定义，在这个作用域里 x 被赋值，因此 x 是局部变量
  - 在语句 y = x 执行时，局部变量 x 还没有定义，因此取它的值是非法的

修改全局或非局部变量

- 按照默认规定，在函数里无法修改非局部的变量
  - 给变量赋值就引进了一个局部变量，操作效果局限在函数内
  - 如确实需要修改非局部变量，必须在函数里使用声明语句
- 全局变量声明语句

  global 变量, …

  语义：声明本函数体里出现的（这些）变量是全局变量，到全局作用域里去找其定义（如无定义，给其赋值将建立全局定义）
- 非局部变量声明语句

  nonlocal 变量, …

  语义：声明在本函数体出现的（这些）变量不是局部变量，到本函数外围的非全局作用域里查找定义（无定义是错误）。如果外围有多层非局部作用域，从内向外逐层检查找最近的定义

变量查找

- 设在函数 f 的体里出现了 x，要确定其定义，顺序做：
  - 如果 f 有参数 x 或局部函数定义 x，则 x 的定义已确定
  - 如果 x 声明为 **global**，到全局名字空间去找 x 的定义
  - 如果 x 声明为 **nonlocal**，到 f 定义所在的作用域（f 的外围作用域）找 x 的定义（可能在更外围，非全局作用域）
  - 如果 x 在 f 的体里被赋值，x 就是 f 的局部变量
  - 否则（x 没声明也没赋值），x 非局部，到 f 定义所在的作用域找 x 的定义，如没有就到更外围的作用域去找，这种查找一直进行到全局作用域。如果查找到全局作用域仍不能找到所查名字的定义，就是变量未定义（错误）
- 注意，**global** 清晰指明了有关变量的定义所在的名字空间，而 **nonlocal** 只说明了从外围定义域出发去找定义。因此后者要求必须“已有定义”，前者的规定更宽松，没定义可以建立

变量和作用域

- 看一个例子（见代码文件）
- **Python** 允许任意嵌套的函数作用域，允许在复杂的作用域结构中任意使用同样的标识符（作为变量名或函数名）
  - 合法的结构都有意义，有明确的语义解释，能执行
  - 但实践中
    - 应该有节制的使用复杂结构
    - 避免不必要名字重复定义和相互遮蔽关系
  - 过度复杂的嵌套不是好的编程实践
    - 影响程序的可读性
    - 过分复杂，有可能隐藏着不易发现的错误
- 注意：外围作用域关系是静态的，基于它建立名字空间的关系

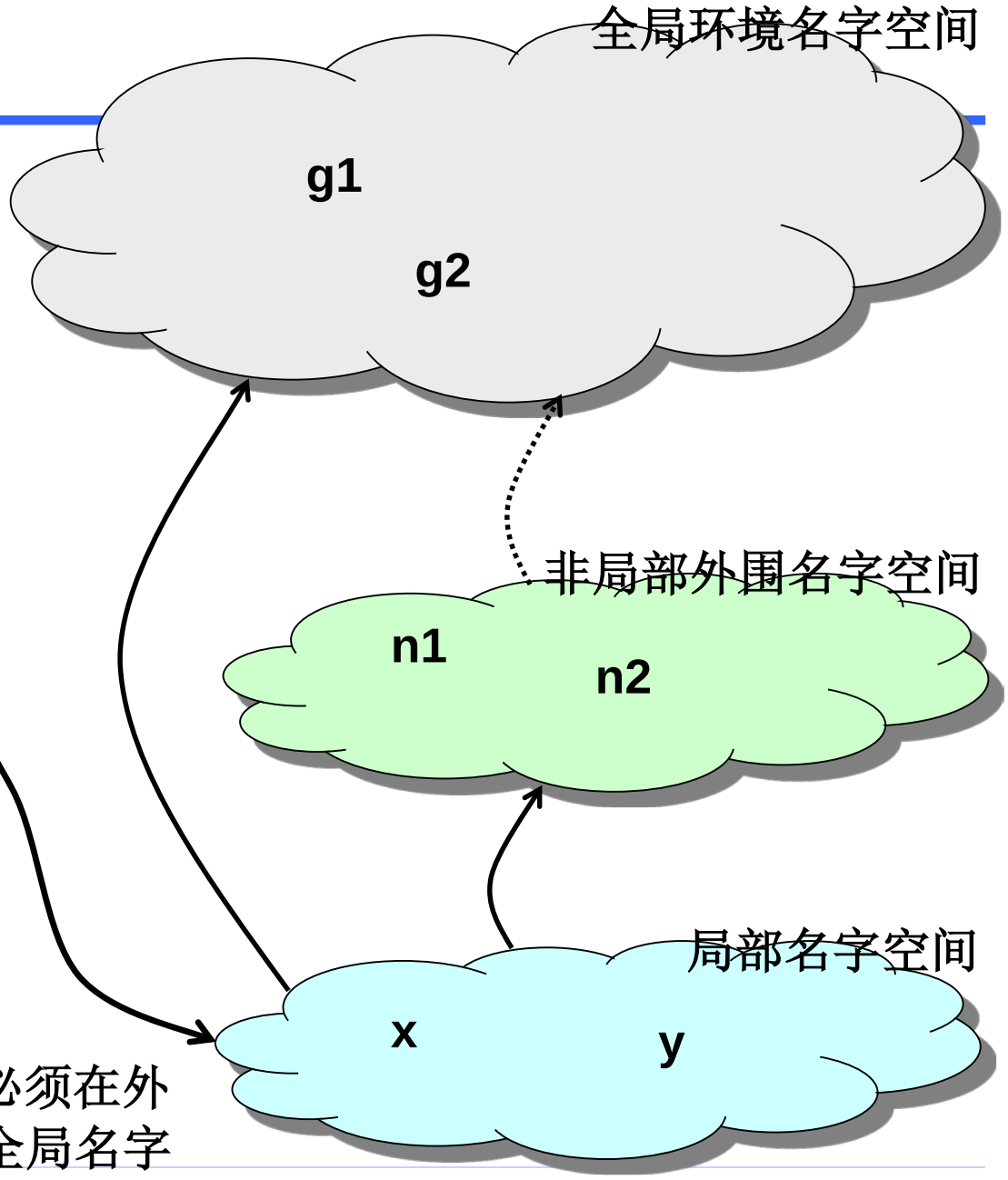
函数调用与环境

- 现在讨论程序执行中由于函数调用和返回导致的运行环境变化
- 函数定义中的代码出现在函数的局部作用域里，在代码里
  - 需要（也可以）使用用函数的参数和局部变量
  - 也可能需要使用全局的和其他非局部的变量或函数
- 函数调用时程序执行进入函数体，函数结束时执行返回到调用函数的位置（之后）。显然
  - 在一个函数被调用的执行期间和函数调用之前之后，两种情况下能看到的环境应该是不一样的
  - 在调用期间能看到函数的参数和局部变量等，而在函数调用之前和之后，都不应该看到它们
  - 函数调用结束后，应该回到函数调用前同样的环境

函数调用与环境

- 函数调用时的操作：
  - 根据函数名（按名字空间的查找规则）找到应该执行的函数（对象）。注意，调用函数只是使用，按前面规则，这个查找可能经过一系列名字空间，直至全局名字空间
  - 从左到右依次求值实参表达式，得到一组实参（对象）
  - 为函数建立一个局部名字空间后开始执行函数体代码
  - 代码执行完毕或者遇到 **return**，函数返回
- 进入函数：建立局部名字空间的初始状态后执行函数体
  - 各形式参数约束到对应的实参（默认为按位置一一约束）
  - 局部变量（根据作用域规定）加入局部环境，其值无定义
  - 开始执行函数体代码

函数调用与环境




```
def f (a b c) :
   global g1, g2
   nonlocal n1, n2
   x = ...a...u...v...
    y = ...b...g1...
   g1 = ...
   n1 = ...
   ... ...
```

u, v 必须在外围或全局名字空间有定义

函数调用与环境

- 函数执行结束（代码结束或遇到 return）
  - 计算出需要返回的值（如果有）
  - 撤销局部名字空间
  - 回到函数调用前的环境（原名字空间及其相关名字空间）
  - 函数值返回（可能赋值）
  - 从调用点之后继续
- 函数结束后，其执行中建立局部约束全部失效（丢掉）
  - 相关局部环境已撤销，其中的所有约束都不再存在
  - 再次调用同一个函数时，将建立一个新的局部名字空间，与前次这种函数执行时建立的局部名字空间无关

全局定义和局部定义

- Python 程序以模块为单位，一个 .py 文件是一个模块
- 模块层的定义是模块里的全局定义
  - 用 def 定义的函数
  - 在函数之外的赋值定义的变量
  - 在全局作用域的其他定义（后面会看到）
- 函数（等）内部的定义是局部定义
- 在一个模块运行时
  - 首先建立其全局定义的名字，执行中建立它们的约束
  - 一个函数被调用时，才建立其局部定义的名字空间
- 可用 import 把另一模块导入当前环境（通过模块名和 . 使用）

  或把另一模块里的全局定义（全部或部分）导入当前环境