程序初步 - 2

- 循环的需求和问题
- 从问题到程序
- 递归
- 程序的终止性
- 函数的定义与调用

写循环程序需要考虑的问题

- 首先是要发现计算中需要重复执行且有规律可循的操作
- 循环控制涉及的一些具体问题：
  - 为完成循环计算需要引进哪些变量？如何控制循环？
  - 循环开始前，变量应该取什么值？(初值问题)
  - 循环体中 (一次迭代计算中) 变量的值应该如何变化？
  - 在什么条件下结束 (或继续) 循环？
  - 循环结束后，如何得到所需结果？
- 循环结构的选择
  - 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**，字符串等) 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，则必须用 **while**

常见循环形式 1：简单重复

- 采用循环的最简单情况：需要重复做一批类似但相互独立的工作
  - 对一系列数据中的每一项做完全相同的计算，分别得到结果
  - 反复输出一批数据，……
- 特点
  - 计算或操作有统一模式，可以用同一段代码描述
  - 不同计算之间的差异只存在于一个或几个变量的取值，而这些取值可以按某种的规律产生，或者按同样的方式获得
- 用循环实现重复工作
  - 识别和描述比较简单
  - 关键是总结共同计算模式，确定循环控制的方式以及循环中变量取值的变化规律

常见循环形式 2：累积

- 特点：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信息 (数据)
  - 每次迭代把一些信息以相同的方式“记入”累积变量中
  - 累积变量的最终值即是循环的主要结果
- 累积程序中常用 + 或 * 操作，适合用扩展赋值运算符 +=，*= 等
  - 对于特殊的累积方式，可以定义专门的累积函数
- 累积程序的实例
  - 阶乘 (代码见前)
  - 计算如下数项级数的前 n 项 (请自己写代码实现)

也属于作累积

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

带条件的累积

- 循环中的累积也可能依据条件，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

➡ 一种典型的计算模式：枚举与筛选

  - 枚举：生成一批候选数据
  - 筛选：从候选数据中选出满足某些条件的合格数据

带条件的累积

```
##### 求用户输入的前 10 个偶数之和
sum, n = 0, 0                          # 两个累积变量

while True:
    x = int(input("Next integer: "))
    if x%2 == 0:                       # 累积的条件
        sum += x
        n += 1
        if n == 10:
            break

print("The sum of ten even number is :", sum)

# 问题：这里可否用 for 语句？
# 嵌套在条件结构中的 break 语句控制循环的退出
```

常见循环形式 3：递推

- 递推：在循环的每次迭代中，基于某个 (或某些) 变量的当前值，按照某种特定方式计算得到其下一个 (或下一组) 值
  - 常见程序描述形式

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

    d 的更新值依赖于 **d** 的当前值和其他数据
- 实现递推循环，需要
  - 选择循环中使用的递推变量
  - 确定各个递推变量的初值
  - 确定如何从已知信息 (包括递推变量的当前值) 计算出各个递推变量新值的方法
  - 确定递推循环结束的条件

例：求 sin 函数的值 (通项计算和递推)

- 已知正弦函数的幂级数展开式 $\sin(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$

  在 $(-\infty, +\infty)$ 上处处成立. 现考虑利用该公式求 **sin(x)** 的近似值

  - 循环结束条件：当前项的绝对值小于 **1e-6**

- 简单思路：先定义函数 **term(x, n)** 计算级数的第 **n** 项，再写一个主循环求部分和

```
def term(x, n):                 # 计算级数的第 n 项
    t = (-1)**n * x**(2 * n + 1)
    for i in range(2, 2 * n + 2): t /= i
    return t
```

```
def mysin(x):                   # 自定义的 sin 函数
    sin_x, n = 0.0, 0
    while True:
        t = term(x, n)
        if abs(t) < 1e-6: return sin_x
        sin_x, n = sin_x + t, n + 1
```

例：求 sin 函数的值 (通项计算和递推)

- 改进：发现递推关系

$$t_0 = x$$

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

  - 利用递推关系可以在级数项的计算中避免大量重复性计算 (具体实现参加演示脚本)

- 测试并分析函数的计算过程：
  - 对程序的测试应该足够充分，且使用各种情况的实例
  - 选择测试实例时，应该选择容易判断结果正误且反映测试需要的实例
  - 浮点数计算中，误差的积累可能会很严重 (参见代码演示)

常见循环形式 4：输入循环

- 输入循环的基本情况
  - 循环中程序从外部不断获得一系列数据用于计算
  - 得到足够多数据之后结束循环，继续后续工作
- 控制循环的进行和结束？两种典型情况
  - 编程时已知需要的数据项数：程序内部控制，属于简单“循环输入”
  - 由程序外部控制的输入循环 (由实际的输入控制循环)
    - 与程序用户 (外部) 约定循环结束的信号 (特殊输入)
    - 在程序里不断检查输入情况，按约定结束循环
    - 具体约定要根据情况设计

```
##### 例1：简单输入循环，已确定输入数据的项数
print("Calculating average of 10 numbers")
ss, num = 0.0, 10
for i in range(num):
    x = float(input("Next number: "))
    ss += x
print("The average of these 10 numbers is", ss/num)
```

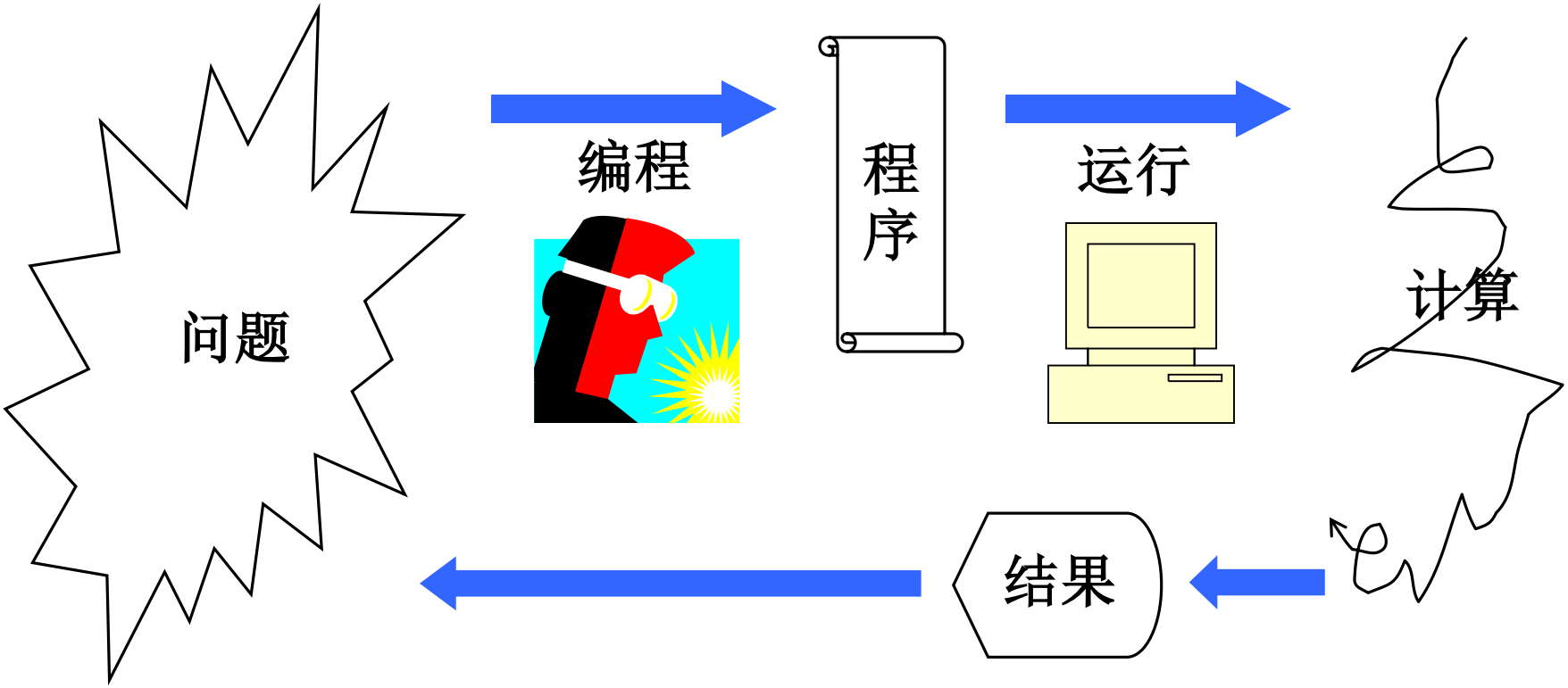
```
##### 例2：输入控制的循环，用约定的特殊输入控制循环
print("Calculating average.")
ss, num = 0.0, 0

while True:
    input_num = input("Next number ('end' to stop): ")
    if input_num == "end":
        break
    ss, num = ss + float(input_num), num + 1

if num != 0:
    print("The average of these " + str(num) + " numbers is", ss/num)
else:
    print("There is no input number!")
```

```
###### 例3：循环输入人名
name = ""
while name != "no more":
    name = input("Hi, friend! What is your name? ")
    print("Hello, " + name + "!\n")

# "no more" 表示输入结束，不能作为人名

name = ""
while True :
    name = input("Hi, friend! What is your name? ")
    if name == "no more":
        break
    print("Hello, " + name + "!\n")
```

```
##### 分开处理实际数据和控制循环的输入
while True:
    name = input("Hi, friend! What is your name? ")  # 实际数据的输入
    print("Hello, " + name + "!\n")
    yesno = input("Next? (Yes/No): ")                # 控制循环的输入
    if yesno == "No":
        break
```

问题和程序

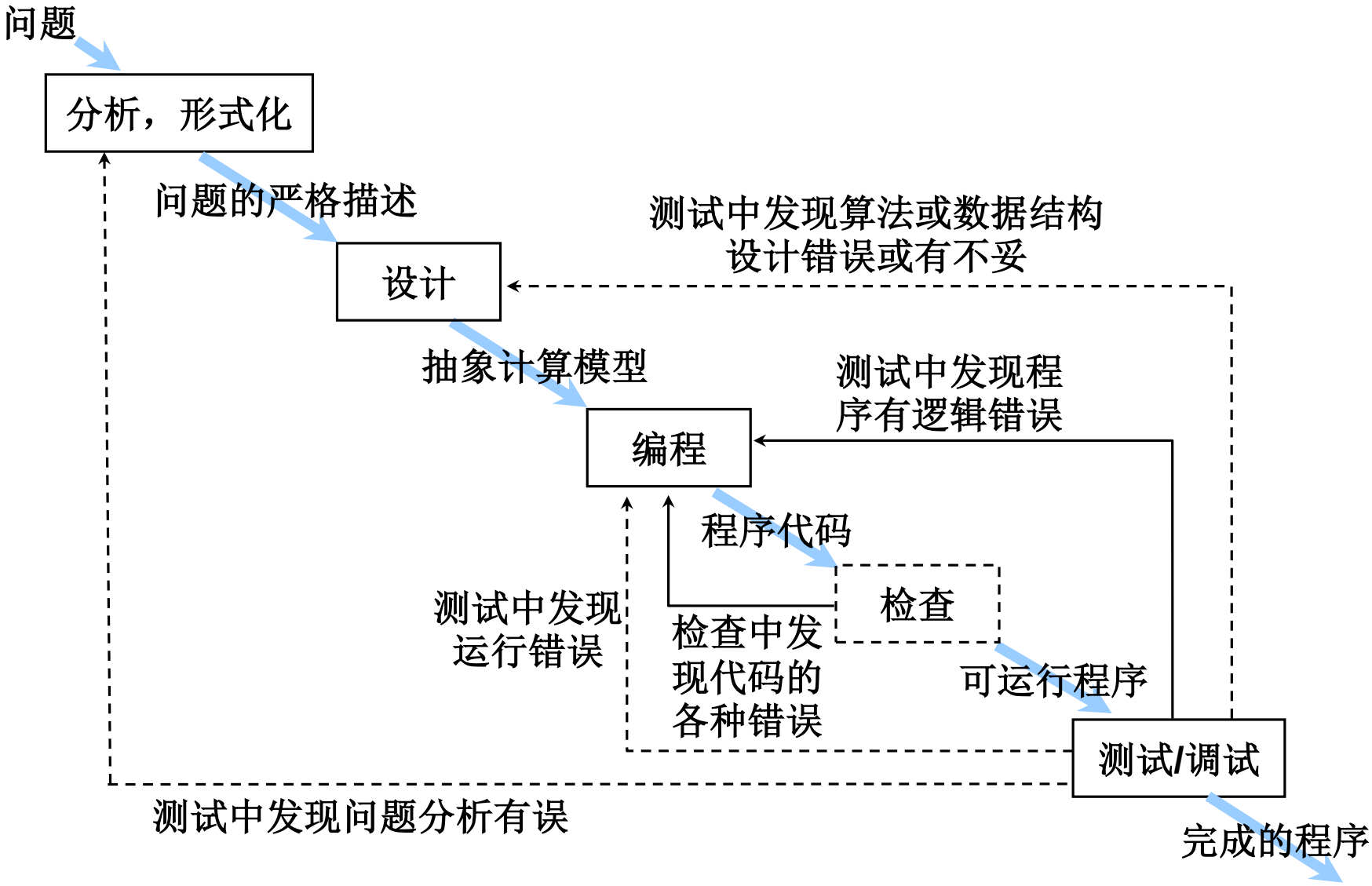
问题、程序和计算



从问题到程序

- 一般地，“问题”和“程序”之间存在距离
  - 问题的自然描述通常采用自然语言，常以“是什么”的方式表达；不精确，描述性，非形式化
  - 程序用精确的编程语言描述，“怎么做”
- 用计算机解决问题的一般过程
  - 设法得到 (写出) 问题的描述 (非形式的)
  - 将问题严格化 (仍然为“是什么”，但更严格准确)
  - 设计问题的解决过程 (怎样做？)，直至得到“算法”
  - 用编程语言写出程序
  - 设法确认写出的程序解决了问题 (可能反复)

从问题到程序 (程序开发过程)



简单问题：生成并输出 1 到 200 的完全平方数

- 方法一：检查从 1 到 200 的整数，遇到完全平方就输出
  - 用循环变量遍历 1 到 200 的所有整数 (适合用 `for` 语句)
  - “判断一个整数是否完全平方”定义为辅助函数
    - 谓词 (的判断函数

```
### 检查一个整数是否完全平方（谓词）
def is_square(n):
    m = 0
    while m * m < n:
        m += 1
        if m * m == n:
            return True
    return False


### 输出 1 到 200 的完全平方数
# squares in [1, 200], method #1
for k in range(1, 201):
    if is_square(k):
        print(k)
```

简单问题：生成并输出 1 到 200 的完全平方数

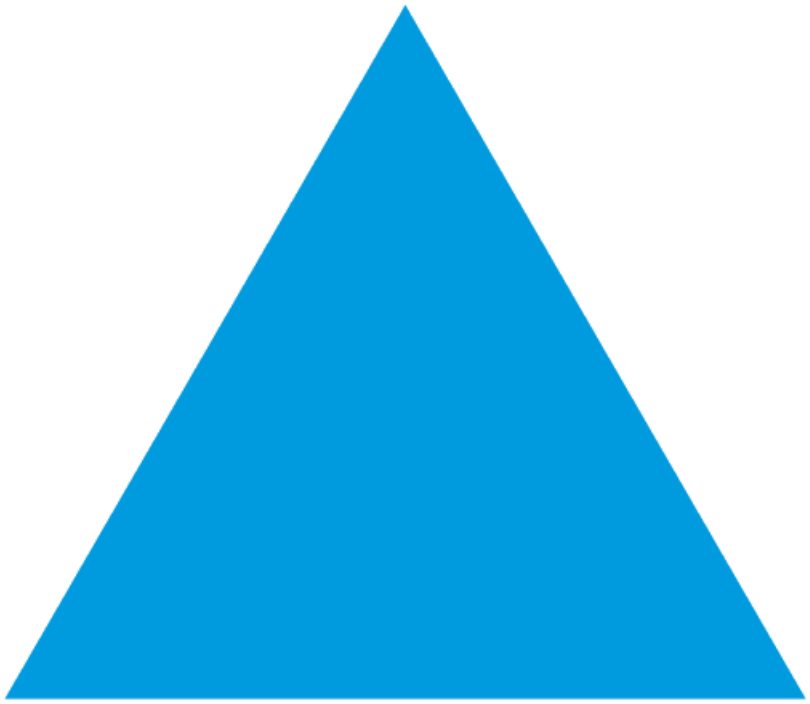
- 方法二：直接输出从 1 到某个数的平方 (要求所输出的平方不超过 200)

```
k = 1
while k * k <= 200:
    print(k * k)
    k += 1
```

- 方法三：根据公式 $(n+1)^2 = n^2 + 2n + 1$
  - 从 1 开始，输出当前的平方数，并递推出下一个平方数，直至当前平方数超过 200
  - 循环中，需要记录当前的 n，且所输出的总是 n 的平方 (循环不变关系)

```
n = 1  # 记录当前的数
s = 1  # 记录当前数 n 的平方
while s <= 200:
    print(s)
    s += 2 * n + 1
    n += 1
```

递归机制



分形图：Sierpinski triangle

自然界中的分形

俄罗斯套娃

函数的递归定义

- 递归函数：在定义时，在函数体里调用自身
  - 利用自己完成自己的一部分工作
- 计算阶乘函数

$$n! = \begin{cases} n \times (n-1)! & n > 0 \\ 1 & n = 0 \end{cases}$$

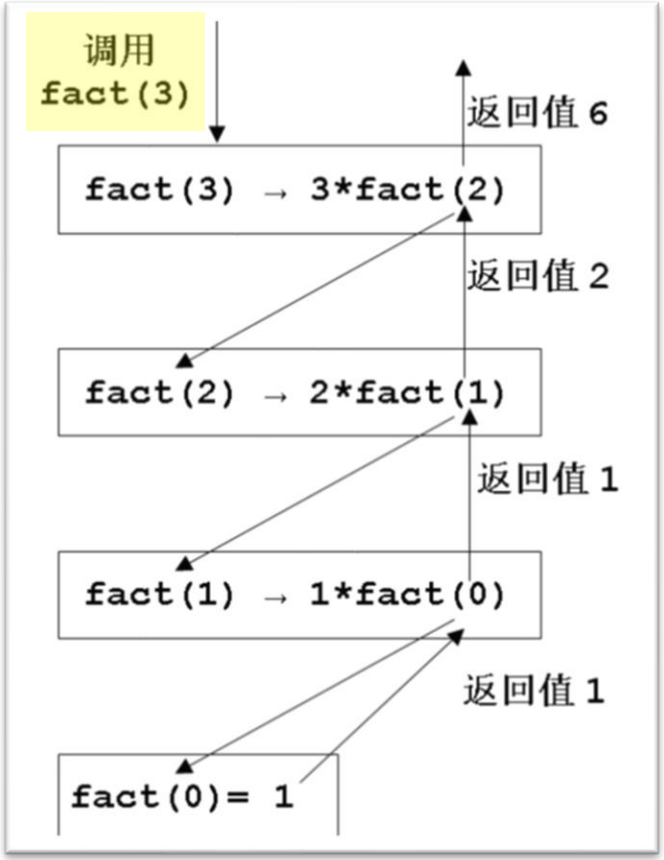
  - 阶乘的数学定义：n 的阶乘基于 n-1 的阶乘定义
  - Python 允许在函数定义的函数体里调用被定义的函数 (允许以递归的方式定义函数)
    - 因此，上述数学定义可以直接翻译为 Python 函数定义

递归函数

```
##### 阶乘函数的递归定义
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

递归结束的条件
— 递归的出口

```
# fact 函数也可利用条件表达式实现
def fact(n):
    return 1 if n == 0 else n * fact(n - 1)
```



线性递归

递归函数

- 递归函数定义的特点：
  - 必须有 (一个或多个) 基础情况：可以直接得到结果 (不需要调用自身)，用来终止递归 — 递归出口
  - (一种或多种) 一般情况：把较复杂参数的计算归结到同一问题的更简单参数的计算

  ➔ 递归函数一般都有参数，且递归调用时让参数朝着终止条件演变
- Python 限制了程序中函数 (递归/嵌套) 调用深度的上限，以防止发生无穷递归

```
def f(n):                # 错误定义的递归函数
    return f(n-1)        # 对 f 的任意调用，都报 RecursionError
```

  - sys 标准库包中函数 getrecursionlimit() 和 setrecursionlimit(limit) 可检查和设置调用深度的上限
- 追踪递归函数的调用 (实例)