其他问题-I

- 逻辑类型和逻辑判断
- 控制语句的扩充
- 函数的参数
- 变动对象的共享
- 自定义迭代器-generator
- 写好 Python 程序
- 怎样阅读 Python 手册

逻辑类型和逻辑判断

- 下面介绍几个前面没仔细讨论的问题

  这些问题都不是本质性的，但有时可能给编程带来方便

  在实际编程中使用很多

  阅读其他人的程序，也需要了解这些情况

- 逻辑数据也是一种数据
  - 逻辑类型 **bool** 只有两个值 **True** 和 **False**
  - 经常需要写各种逻辑表达式，值是逻辑值
  - 可以直接作为数据，例如前面蒙特卡罗模拟的例子
  - 常作为函数的返回值，存入变量，作为复合数据对象的元素
  - 主要用在 **if** 和 **while** 语句和条件表达式里，用做条件判断

逻辑类型和逻辑判断

- 为方便编程，**Python** 允许直接将其他内置类型的对象用于逻辑判断和逻辑运算。各种类型的下面值都表示假
  - **False, None**
  - 数值类型里的各种零：**0, 0.0, 0.0+0.0j**
  - 空序列：**"", [], ()**
  - 空字典：**{ }**

  要求逻辑值的位置，出现这些类型的其他值都作为真
- 例：

```
while lst: # 不必写 lst != []
    x = lst.pop()
    … x … #

if not lst or lst1: … # 将表用于逻辑运算
```

控制语句的扩充

- while 语句和 for 语句后面可以跟一个 else 段
  - 如果有 else 段，循环正常完成后执行这个段的语句组，而后整个 while 或 for 循环语句结束
  - 如果循环通过 break 结束，else 段的语句组不执行
- 一个简单例子

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # 循环做完没发现因子
        print(n, 'is a prime number')
```

注意循环语句的 else 段语句组与循环之后的语句之间的差别：

只对通过 break 结束循环的情况有影响

函数参数：形参的默认值

- 函数定义和调用中形参/实参的基本情况：
  - 在函数定义的形参表中列出各（形式）参数的名字
  - 函数调用中所给实参表达式，按位置做实参与形参的匹配
- 但前面已经看到，系统内置的 print 等函数，可以用关键字的形式描述实参，相应的形参可以有默认值
- 自定义函数也可以有带默认值的形参，调用时对应的实参可以缺省。要为这种形参提供实参值，可能需要使用关键字实参的形式
- 定义函数时为形参提供默认值的描述形式：

  形参名=表达式

  在函数定义时求值默认值表达式，记录形参名与表达式值的约束关系。如果调用该函数时没为这种形参提供实参，就用默认值。函数可有任意个带默认值形参，默认方式是实参顺序匹配

函数参数：形参的默认值

- 例：累计一个表里的数值：

```
def accum(lst, init=0):
    nsum = init
    for x in lst:
        nsum += x
    return nsum
```

合法调用：

```
a = accum([1, 2, 3])
b = accum([5, 7, 10], a)
```

- 例：累计一个表里前 num 项的数值，num 负值表示全累计：

```
def accum(lst, init=0, num=-1):
    nsum = init
    if num == -1:
        num = len(lst)
    for i in range(num):
        nsum += lst[i]
    return nsum
```

合法调用：

```
a = accum1(l1, 31)
b = accum1(l1, 10, 6)
c = accum1(l1, num=6)
```

函数参数：关键字实参

- 函数调用的关键字实参
  - 在函数调用中，可以指明调用式中某个实参约束到函数的哪个形参，这种实参描述形式称为关键字实参
  - 如果函数参数较多，关键字实参能帮助说明实参的约束关系
  - 当函数有多个默认值时，经常需要用关键字实参指定约束
- 例如，下面两个函数调用给出同样结果：

```
def f(x, y) :
    return x**y

print(f(2, 3))

print(f(y = 3, x = 2))
```

第一个调用的实际参数按位置匹配，第二个调用按参数名匹配

带星号形参

- 在函数的形参表里可以有一个前面加一个星号的形参
  - 函数调用时该形参约束到所有未得到匹配的普通实参构成的元组，默认情况是约束到空元组（如果没有未匹配实参）
  - 利用这种形参形式，可以定义允许任意多个实参的函数
- 下面定义了一个可对任意多个数值求和的函数：

```
def mysum (*args) :
    s = 0
    for x in args :
        s += x
    return s
```

调用 **f1(1,22,3,4,5)** 得到 **35**

调用 **f1()** 得到 **0**

双星号形参

- 在函数形参表最后，可以有一个前面加两个星号的形参
  - 函数调用时该形参约束到所有未匹配的关键字实参形成的字典，默认情况下约束到空字典
  - 这是一个自动建立的新字典，从各关键字映射到相应的值
  - 引入这种形参，是为了在一些复杂情况下编程方便
- 在一个函数形参表里，可以有任意多个普通形参（可以带默认值），至多一个带星号的形参，至多一个带双星号的形参
  - 为了意义清晰，Python 规定如果形参表里的某个普通形参带了默认值，位于其后所有普通形参都必须带默认值
  - 带星号形参需位于普通形参之后，前后都可有带默认值形参
  - 带双星号形参只能出现在形参数表的最后，否则是语法错

函数调用

- 函数调用时，先求出被调用的函数，而后从左到右求出各实参值
- 参数匹配方式：
  - 普通的实参按位置与形参一一匹配
  - 关键字实参按关键字与同名形参匹配，它们只能出现在按位置匹配的普通实参后面

    print(end=", ", 1, "abc") # 错误调用
  - 匹配剩下的普通实参做成一个元组约束到带星号形参
  - 匹配剩下的关键字实参做成一个字典约束到双信号形参
  - 如果没剩下相应种类的实参
    - 带星号实参约束到空元组
    - 带双星号形参约束到空字典

函数调用

- 函数调用的分拆实参（**unpacking argument**）
  - 在调用函数时可以将元组或表分拆（**unpacking**）打开，用其元素为函数提供若干个实际参数值
  - 分拆参数是一种特殊的实参形式，用在实参表达式前加一个星号表示。例如：

```
a = 1,2,3,4,5 # a 的值是元组
mysum(*a) # 得到 15

# 写 mysum(a) 将出错（元组不能与 0 相加）。再如

b = [0, 20, 3]
for i in range(*b): …  #  相当于 range(0, 20, 3)
```

  - 如果在一个函数调用式里出现了分拆实参，该实参之后不能再写普通实参，但可以写关键码实参

函数调用

- 实例：

```
def func(a, *b, c=0, d=1):
    print(a, b, c, d)
    return 0
func(1, 2, 3, d=10)

def func1(a, b=0, *c, d=11):
    print(a, b, c, d)
    return 0
func1(1, 2, 3, d=10)
```

- 与分拆函数类似，也可以用合适的字典为函数提供实参
  - 字典的关键码看作函数调用中关键码实参的关键码，对应的值作为实参的值（易见，字典的关键码应与函数形参匹配）
  - 写法：写在函数调用实参表里，前面加两个星号

函数调用

- 一般情况：调用式里首先是一些按位置的普通实参，随后是一些关键码实参，其中可以出现一个分拆实参，最后是一个字典分拆实参。几种实参都可以没有，如果有则应按顺序出现
- 对一个函数调用
  - 如果调用中有分拆实参，打开它应得到一串实参，把它们排在已有的普通按位置实参之后。首先按照位置顺序地把这些实参与函数形参表中的形参一一匹配
  - 按名字为关键码实参确定对应的形参匹配。如果有字典分拆实参，把打开它得到的关联作为附加的关键码实参。如果发现应该匹配的形参已有匹配，就报告**TypeError**错误
  - 剩下的所有普通实参按顺序做成一个元组，约束到被调用函数的带星号形参，剩下的所有关键码实参做成一个字典，约束到函数的双星号形参

函数调用

- 实参太多或太少的情况

  如果完成上述匹配后，还存在没有得到约束值的形参（假如实参太少），或者存在多余实参（在函数定义中没有带星号/双星号形参时可能出现这种情况），就报告**TypeError**错误

- 应该注意：
  - 由于存在分拆实参（分拆产生的实参个数在实际调用时才能确定），函数调用的实参错误只能在运行中检查和发现
  - **Python** 提供了丰富的形参定义和实参使用形式，是为了实际编程方便，各种机制都有有用的地方
  - 但不应该任意混合使用。太复杂的形参和实参会使程序中函数调用的意义变得很不清晰

变动性和共享

- 前面讨论了对象变动性和共享的关系
  - 例如，**list/set/dict**
  - 现在再看两个变动性造成程序问题的实例
- 表构造与变动性

```
list1 = [0] * 10
list10 = list1.copy()
list2 = [[]] * 10
list20 = list2.copy()
list3 = [[] for i in range(10)]
list30 = list3.copy()
list1[1] = 3
list2[1].append(1)
list3[1].append(1)
```

```
# 建立list1的拷贝，不变元素

# list2的拷贝，共享一个可变元素

# list3的拷贝，共享一组可变元素
# 换了一个元素
# 元素本身变动
# 元素本身变动
```

变动性和共享

- 函数的默认参数在函数定义时求值，如果默认值是可变对象，也可能引起共享问题。看个例子（希望把一些元素加入一个表）：

```
def insert(args, start=[]):
    for x in args:
        start.append(x)
    return start
```

这里默认值是可变对象，没提供 **start** 值时多次调用造成共享

- **Python**建议不用可变对象作为默认值，写法：

```
def insert(args, start=None):
    if start is None:
        start = []
    for x in args:
        start.append(x)
    return start
```

and/or 和短路求值

- 假设要处理表 L1 里从头开始的所有正的数值，遇到第一个非正数时就结束。我们可能写：

```
i = 0
while i < len(L1) and L1[i] > 0 : # 这段程序有问题吗？
    ... L1[i] ...
    i += 1
```

- 上面程序能正确工作的前提：and 在其第一个运算对象的值为假时，不再求值其第二个运算对象。否则就必须写

```
while i < len(L1):
    if L1[i] > 0 :
        ... L1[i] ...
        i += 1
    else:
        break
```

and/or 和短路求值

- and 和 or 的确切语义：

  ex_1 and ex_2 and ... and ex_n 顺序求值 ex_1, ex_2, ..., ex_n

  一旦某个ex_i 求出的值是假，整个 and 表达式的值就是假，其余 *ex* 不再求值

  ex_1 or ex_2 or ... or ex_n 顺序求值 ex_1, ex_2, ..., ex_n

  一旦某个ex_i 求出的值是真，整个 or 表达式的值是真，其余 *ex* 不求值

- and/or 表达式在确定了值后，剩下的子表达式不再求值
  - 就像是短路了表达式后面的部分
  - 这种求值方式称为短路求值

自定义迭代器－generator

- 现在介绍一种用户定义迭代器——generator（生成器）
  - 定义迭代器的最简单方法是用定义函数的形式定义一种生成器对象，这种对象可以作为迭代器使用
  - 函数定义里出现 yield 语句，定义的不是一个普通函数，而是一个构造生成器（对象）的函数
  - 生成器是可迭代对象
- 例：

```
def nat(limit):
    for n in range(limit):
        yield n

for i in nat(10):
    print(i)
```

定义生成器

- **yield** 语句
  - 形式：**yield** 表达式
  - 语义：使生成器对象返回表达式的值，但它不结束，而是停在这个位置，可以从这里继续下去（通过特殊调用形式）
- 生成器的一些情况：
  - 作为生成器对象，重要的是它 **yield** 的值，结束时的返回值不重要
  - 调用生成器函数，返回一个新建的生成器对象

    用同一个生成器函数可以创建多个生成器对象，相互无关
  - 把生成器（对象）作为 **for** 语句头部或表（元组/集合/字典）描述式里的迭代器，它将被反复执行取 **yield** 值，直到结束。遇到函数定义里的 **return** 语句，或执行完整个函数体时结束

generator 实例

- 生成器实例：
  - yield 和 return 的混合使用
  - 一个生成序列中所有非负数值的生成器
  - 一个生成斐波那契序列值的生成器
- 生成器不但可以用在需要可迭代对象的场合，也可以独立使用

  如果 g 的值是一个生成器对象

  每次调用 g.__next__() 得到该生成器 yield 的下一个值

generator

- 可以定义“有用的”无穷生成器，在这种生成器的活动期间，允许任意多次地调用其 __next__()
  - 如果在 for 等上下文中使用，无穷生成器将导致无穷循环
  - 自己通过 __next__() 使用生成器函数，怎么用完全由编写的程序控制，可使无穷生成器变成一种有用的工具
  - 例：生成斐波那契序列的生成器
  - 循环生成给定序列里的元素的生成器
- 内置函数 map, filter 也生成一个和生成器性质类似的对象，同样可以对其调用 __next__() 函数

写程序的一些问题

- 考虑写程序和写数学的类似与差异
- 类似：都是用符号形式写形式化描述，严格性
- 差异也很大：
  - 程序是一维顺序描述，数学表达式可能写成二维形式
    - 一维形式机器容易处理
    - 简短时容易阅读，很长时缺乏结构特征
  - 数学表达式通常较短，程序可能很长，成千上万行或更多
  - 机器对严格性的要求更强，形式错误（语法错误）是一类最基本错误。数学表达式可以依赖于人的理解和共识
  - 程序的意义是动态运行的效果，这种意义更隐晦，更难把握（再加上规模大等因素）

写程序的一些问题

- 大脑的特点限制了人对复杂事物的理解把握能力
  - 理解和把握程序这样复杂事物，非常困难，必须采用一些有效方法，从各个层面着手
  - **Python** 语言的设计中提供了模块分解、复杂功能的函数分解等很多机制，支持人们写出有层次结构的良好程序
  - 设法拉近程序的描述形式与程序语义的关系（如 **Python** 采用强制退格的程序形式等，引入一种二维结构）
  - 程序对象的命名，注释等基本措施
- **Python** 教程 **4.8. Intermezzo: Coding Style** 介绍了用 **Python** 编程的人们认为形式良好的 **Python** 程序应遵循的一些准则，供参考。更详细见 **PEP 8**。其中 **PEP, Python Enhancement Proposals** 译成中文 **"Python** 增强建议书**"**，是人们对 **Python** 语言发展的建议书，有很多建议已经实现

程序形式：对象的命名

- 程序里的名字用于指称事物
  - 如函数名，变量名
  - 数学领域习惯用单字母的名字：**x y z a b c f g h**（或带下标）

    作用比较局部，没有复杂的对象类型的问题
- 编程序的情况不同，倡导采用反映编程意图的有意义的名字
  - 对全局变量，函数名，或更高层结构，通常采用更完整的名字，以便在阅读时起到很好的提示作用
  - 作用范围有限局部变量，可采用简单的名字

    如 **for** 头部引进的循环变量

    描述式引进的局部变量，等等
  - 变量/函数/更高层的结构/常量等采用不同形式的名字

程序形式

- 空格：
  - 运算符左右应留空格，逗号之后应留空格
  - 函数名和括号之间不留空格，冒号之前不留空格
  - 层次之间统一退 **4** 个空格，不用 **tab**
- 换行和空行
  - 一行不要太长，过长的行应换行（必要是用续行符）
  - 函数定义之间，大的代码片段之间加入空行
  - 注释尽可能写为独立的注释行
- 等等

怎样阅读 Python 手册？

- 整体结构
- **Python 教程（tutorial）**

  中文 **3.4**：
  http://www.pythondoc.com/pythontutorial3/index.html

  http://python.usyiyi.cn/python_341/tutorial/index.html

  注意：**4.8. Intermezzo: Coding Style**
- **Python 安装和运行**
- **Python 语言手册**
- **Python 标准库手册**