文件处理实例 1：文件内容统计

- 问题：定义函数统计一个文件里各类字符出现的次数
  - 根据文件名 (字符串参数) 打开一个文件
  - 依次处理文件里的每个字符
    - 判别每个字符的类别 (字母、数字、空白和其他字符等)，并进行统计
  - 关闭文件，并输出统计结果
- 细节：文件里的字符可能超出基本的 ASCII 编码字符集 的范围
  - 打开文件时，可能需要通过函数 open 的 encoding 参数说明被读文本文件的编码方式 (编解码器 codec, encoder/decoder)
  - 例如，当读入文件里有中文，可以给 open 增加关键字实参 encoding="utf_8" 或 encoding="gbk"
    - 需要根据文件的实际编码方式
    - 'utf8', 'utf-8', 'U8' 等是 'utf_8' 的别名

演示

文件处理实例 2：词频统计

- 需求：实现函数统计一个文本文件 (如一部小说或其他文献) 中各个单词出现的频率，并把统计结果输出至文件
  - 事先不知道文件包含哪些单词，遇到任意单词都需要记录
  - 单词可能重复出现，需要找到已经记录的单词更新统计
- 用字典记录统计结果
  - 关键字 → 单词，关联值 → 相应计数值
- 读文本过程中遇到每一个单词时均计数
  - 如果单词不在字典里，就加入字典，计数值设定为 1
  - 如果已经在字典里，计数值加 1
- 文本形式：文本是空白字符分隔的一系列单词
  - 一个单词就是非空白字符的一段连续字符序列 (非常粗略地处理)
- (演示)

文件处理实例 3：提取文件中的数值数据 (1)

- 问题：设一个正文文件中保存一批由空白字符 (空格、换行和制表符) 分隔的数值数据，要求定义函数提取该文件中的各项数据
- 思路一：一次性读入文件里的所有信息，之后再分解并转换数据 (演示)
- 思路二：定义生成器函数 (演示)
- 思路三：定义提取数据的函数，每调用一次从指定文件对象中读取下一个数据 (惰性方式，避免建立大型数据对象)
  - 函数 **open_floats**：负责打开数据文件、建立读文件对象；其参数为文件名字符串 (只被调用一次)
  - 函数 **next_float**：采用“缓冲式”工作方式，用一个‘数值字符串’表保存文件里的“当前一行”；每调用一次则生成并给出下一项数据；如表内的现有数据被处理完，则从文件对象读入新的一行并分解；其参数为读文件对象
  - 问题：两个函数之间如何传递信息？ **next_floats** 的多次调用之间，如何传递作为缓冲区的表，及当前行中当前数据的位置？(实现略)

文件处理实例 3：提取文件中的数值数据 (2)

- 实现四：考虑如下定义函数 **read_floats**
  - 该函数负责数据文件的打开操作
  - 在该函数内部，保存文件读入过程中各方面信息 (➔ 信息的局部化)
    - 读文件对象
    - 作为缓冲区的字符串表
    - 对于当前行的数据表，已提取数据的项数
  - 该函数返回“从文件中逐个读取数据”的计算功能 ➔ 返回一个函数对象
    - 每一次调用所返回的函数对象，得到文件中的下一项数据
    - 所返回的函数对象：需要使用记录文件读入过程的各方面信息 (文件对象、缓存表等，都是 **read_floats** 函数的局部变量)，应被定义在 **read_floats** 局部

```
def read_floats(fname):
    infile = open(fname) # 局部变量，记录读文件对象
    nlist = []       # 局部变量，记录当前一行中的数值字符串(表) — 数据缓冲区
    crt = 0          # 局部变量，记录表中当前数据项的下标

    def next_float():                       # 局部函数定义
        nonlocal nlist, crt                 # 非局部变量声明

        if crt == len(nlist):               # 当前缓冲行已用完
            line = infile.readline()        # 读入新的一行
            if not line:                    # 文件已处理完
                infile.close()              # 关闭文件
                return None                 # 返回 None 表示读完文件
            nlist = line.split()            # 分解正文行成数值字符串表
            crt = 0                         # 重置当前项下标 crt

        crt += 1
        return float(nlist[crt - 1])        # 返回当前项对应的数值

    return next_float                       # 返回局部函数对象
```

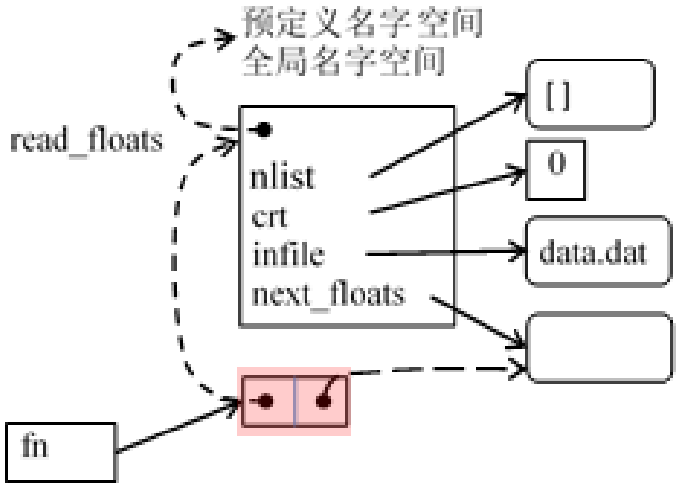
闭包的基本原理 (1)

- 执行语句 **fn = read_floats("data.dat")**
  - 开始执行 **read_floats** 函数时，创建该函数的局部名字空间
    - 其中，建立各局部变量与其值对象的关联
  - **read_floats** 执行结束时，变量 **fn** 接收其返回值，其中包括两部分信息
    1. **next_float** 所定义的局部函数对象
    2. **read_floats** 本次调用所建立的名字空间

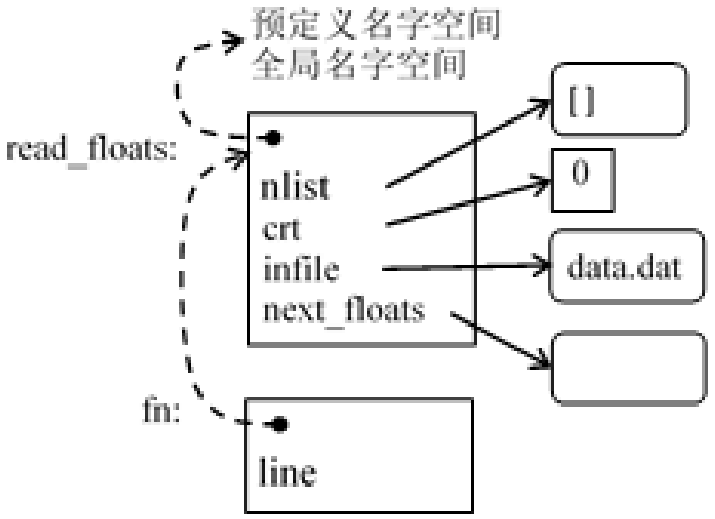
    } 闭包二元组
  - **read_floats** 执行结束后，执行时所建立的名字空间仍被变量 **fn** (通过闭包二元组) 间接引用
    - 该名字空间不会被回收，以保证该名字空间及其内部的变量继续可用

闭包的基本原理 (2)

- 执行语句 **x = fn()**
  - **fn** 所关联的函数对象被执行，解释器将为其建立名字空间
    - 其外围是闭包二元组所引用的名字空间
  - 执行期间，外围名字空间的变量 **nlist, crt** 的关联值将被修改
    - 即，局部函数对象 **next_float** 访问了其定义体之外的非全局变量 (作用域的延伸)
  - 执行结束时，返回所提取的一个浮点数，且闭包中的外围名字空间的状态也被更新



read_floats("data.dat") 执行后的环境和状态



x = fn() 执行时的环境和状态

闭包函数

- 在函数 f 中定义了局部函数 g，如果 g 使用了非局部状态 (如 f 中的局部变量)，且 f 的返回值是局部函数 g，则函数 f 是闭包函数 (工厂函数)
  - 当用实参调用 f 时，将建立一个与实参相关联的新闭包，其中包括
    1. 局部函数 g 定义的函数对象
    2. f 本次执行的名字空间 (即闭包的内部状态)
  - 闭包中的局部函数 g 可以访问和修改 f 执行时所建的名字空间，称之为闭包的接口函数 (演示)
  - **Note**：多次调用 f 时，将产生不同的、各自独立的闭包
    - 其中的局部函数对象，虽然共享同一段函数定义代码，但所引用的外围名字空间不同，因此执行状态不同且无关

```
>>> fn1 = read_floats("data1.dat")
>>> fn2 = read_floats("data2.dat")
```

闭包和生成器函数

- 闭包和生成器函数都是能定义数据抽象/封装的机制
- 生成器函数构造生成器对象
  - 基本功能是用 next 取值
  - 可用于 for 语句头部、各种推导式的迭代描述、生成序列 (表、元组) 等
- 闭包技术的功能更强
  - 如接口函数是无参函数，闭包的功能类似于生成器
  - 通过接口函数引入参数则能实现复杂功能的闭包
  - 可用于构造装饰器

```
#### 采用闭包技术生成斐波那契序列
def fibs_closure(limit):
    f1, f2 = 0, 1
    i = 0                          } 闭包的内部状态

    def generator():
        nonlocal i, f1, f2
        if i == limit:
            return None
        tmp = f1
        f1, f2 = f2, f1 + f2
        i += 1
        return tmp

    return generator
```

```
if __name__ == "__main__":
    fibs = fibs_closure(15)
    for i in range(8):
        print(str(i) + ": ", fibs())

    while True:
        x = fibs()
        if x is None:
            break
        print(x, end =" ")
```

带接口的闭包

- 实例：利用闭包技术实现一个计数器对象，要求该对象
  - 能记录一个计数值
  - 能支持增减、检查当前值等简单计数器操作

```
def counter(init=0):
    count = init                              # 局部变量，保存计数值

    def interface(command="value"): # 接口函数，参数为操作命令
        '''解释和分发由参数获得的操作命令，即是命令分发器'''
        nonlocal count
        if command == "value": return count
        elif command == "inc": count += 1
        elif command == "dec": count -= 1
        else: return "Not understood."

    return interface                          # 返回接口函数
```

```
count1, count2 = counter(), counter(10) # 建立两个计数器对象（闭包）

for i in range(4): count1("inc")

count2("inc")
for i in range(3): count2("dec")

print("counter1 =", count1())
print("counter2 =", count2("value"))
```

简单的装饰器

- 装饰器 (decorator) 是一个函数 (可调用对象)，简单情况下：
  - 参数是一个函数对象 func (即，需要被装饰的函数)
  - 返回值是另一个函数对象 (即，装饰后的函数)
  - 功能：不修改被装饰函数 func 的定义，在完成 func 工作的基础上，附加上一些额外的功能
    - 装饰后的函数是被装饰函数 func 的特定扩充，通常二者接受相同的参数，且返回相同的值
- 在实现通用的装饰器 (工厂函数) 时，一般会用到以下机制
  - 函数参数
  - 闭包
  - 单星号形参、双星号形参：被装饰的函数可能有不同的参数
- (演示)

Python 模块和执行

- 一个定义好的 Python 模块 (py 文件) 有两种使用方式
  - 作为程序的主模块，直接启动运行
  - 作为辅助模块，被其他模块导入 (import) 和调用
- 一个大些的程序通常由一个主模块和若干辅助模块组成

  程序从主模块启动，主模块导入其他模块并使用其中的功能
- 对于辅助模块，有时也需要独立考虑和检查其功能，例如在开发和修改测试阶段。利用 Python 的模块命名机制可以方便地处理这两种情况
  - 程序运行时，总有一个全局变量 __name__，在程序执行中的每个时刻，这个变量都具有一个自动设置的字符串值
  - 通过导入的方式执行一个模块，__name__ 将被设置为该模块的 py 文件的名字；当一个 py 文件被作为主模块启动时， __name__ 的值设置为特殊名字 "__main__"
  - 检查 __name__ 的值，则可以判断本模块是以什么方式运行

执行 Python 程序

- IDLE 是 PSF 主导开发 CPython 系统所附带的编程环境
  - 可用于编写、修改程序，运行、测试和调试程序
  - 也可用其他编辑器或专门开发环境，如 PyCharm, Spyder…
  - 当程序开发完成之后，Python 程序可以独立地运行
- 正常安装 Python 后，安装程序应该已经设置好程序路径，把 Python 解释器的相关目录加入可执行程序的查找路径，可以直接启动执行 (也可以手动设置)
- 在命令窗口 (如 Windows 下的 cmd, PowerShell 等) 通过命令行方式

```
python abc.py      # 启动 python 解释器运行程序 abc.py
python /?          # 查看启动 python 系统时可以使用的参数
```

  - 启动时可通过各种启动参数来指定程序的运行方式等

命令行参数

- 在以命令行方式启动 **Python** 程序时，还可以为被执行的程序提供“参数”，供程序在运行中使用
  - 以命令行方式启动 **prog.py** 时，在命令行中 **prog.py** 的后面，可以写任意多个空格分隔的“参数”，称为**命令行参数**
  - 对这些参数，程序 **prog.py** 需要通过标准库模块 **sys** 取得和使用
- 程序 **prog.py** 里取得命令行参数的方法
  - 在 **prog.py** 里导入标准包 **sys** 后使用 **sys.argv** (或者，从 **sys** 导入 **argv** 后直接使用 **argv**)，其值是一个**字符串的列表**
  - 设启动程序的命令行为 ***python prog.py arg1 arg2 …***：在该程序运行时，表 **argv** 里的每个字符串对应一个命令行参数
    - **argv[0]** 是脚本程序的名字 **"prog.py"**，**argv[1]** 是字符串 **"arg1"**，**argv[2]** 是字符串 **"arg2"**，依此类推

两个演示