# 1 基本配置及基础语法(1, 2 章)

1 Python.exe 的解释器 options:
  1.1 –d 提供调试输出
  1.2 –O 生成优化的字节码(生成.pyo 文件)
  1.3 –S 不导入 site 模块以在启动时查找 python 路径
  1.4 –v 冗余输出(导入语句详细追踪)
  1.5 –m mod 将一个模块以脚本形式运行
  1.6 –Q opt 除法选项(参阅文档)
  1.7 –c cmd 运行以命令行字符串心事提交的 python 脚本
  1.8 file 以给定的文件运行 python 脚本
2 _在解释器中表示最后一个表达式的值.
3 print 支持类 c 的 printf 格式化输出: print “%s is number %d!” % (“python”, 1)
4 print 的输入内容后面加逗号, 就会使其输入不换行
5 把输出重定向到日志文件:
logfile = open(“c:/1.log”, “a”); //打开文件 c:/1.log 使用 a 模式..即 add, 添加.
print >> logfile, “Fatal error: invalid input!”; >>为重定向..将 print 的结果重定向到 logfile, 输出内容是”Fatal error: invalid input!”…
logfile.close(); //关闭文件流…
6 程序输入: raw_input(“提示字符串”): user = raw_input(“请输入您的姓名”);
7 int(数值).....将数值字符串转换成整数值…
8 运算符:
  8.1 + - * / %是和其他语言相同的加减乘及取模运算.取余运算
  8.2 / 在浮点取模中得到的结果是完整的浮点数
  8.3 // 在浮点取模中得到的结果是经过舍去运算的结果.
  8.4 ** 是乘方
  8.5 >>和<<的移位运算也支持. 但不支持 java 中的>>> 和<<< 移位.
  8.6 < <= > >= ++ != <> 等比较运算符
  8.7 and or not 等逻辑运算符
9 变量和赋值: python 是弱类型语言..
10 list, tuple, map * 4 得到的结果是一个新的 list | tuple | map, 是原数据的 4 份
11 数字:
  11.1 int(有符号整数)
  11.2 long(长整数)
  11.3 bool(布尔值)
  11.4 float(浮点值)
  11.5 complex(复数)
  11.6 python2.3 开始, 如果结果从 int 溢出, 会自动转型为 long
  11.7 python2.4 开始支持 decimal 数字类型, 需要导入 decimal 模块..由于在二进制表示中会有一个无限循环片段, 普通的浮点 1.1 实际是不能被精确表示的, 被表示为 1.1000000000000001. 使用 print decimal.Decimal(‘1.1’);则可以得到精确的 1.1
12 字符串: 引号之间的字符集合, 支持使用成对的单引号和双引号, 三引号(三个连续单引

号或双引号)可以用来包含特殊字符. 使用索引运算符[]和切片运算符[：]可以得到子字符串…字符串中第一个字符的索引是 0, 最后一个字符的索引是-1;

13 列表和元组: 可以看作是普通的数组, 能保存任意数量任意类型的 python 对象…

13.1 列表元素用中括号包裹, 元素的个数及元素的值可以改变.

13.2 元组元素用小括号包裹, 不可以更改, 尽管他们的内容可以, 元组可以看成是只读的列表. 可以使用切片运算得到子集.

14 字典: 相当于其他语言中的 map, 使用{key: value}的方式表示. 取值的方式和其他语言的 map 一致. 也可以直接使用 map[key] = value 的方式为其赋值.

15 条件语句:

```
if expression:
    path 1
elif expression2:
    path2
else:
    path3
```

16 while 循环

```
while expression:
    process business
```

17 for 循环

```
for item in list|tuple|map:
    print item
```

17.1 range(len(list))得到一个 list 长度范围内的整数 list, 方便遍历过程中获取索引值.

17.2 python2.3 中增加了 enumerate(), 可以通过它遍历 list, 同时得到索引和值

```
for index, data in enumerate(list):
    print index, ":", data,
```

17.3 列表解析: sqdEvens = [x ** 2 for x in range(8) if not x % 2], 获取一个序列, 该序列是 0-8 的数字中所有 x%2 为 0(false)的 x 的平方

18 文件和内建函数: open(), file()

18.1 handle = open(file_name, access_mode = "r"), 只读方式打开文件, 得到的句柄是 handle..该方法如果没有提供 access_mode, 默认是 r

19 异常处理: raise 可以故意引发异常

```
try:
    # process
except IOError, e:
    # error process
```

20 函数: 如果函数中没有 return 语句, 自动返回 None 对象

```
def function_name([arguments]):
    "optional document string"
    function_suite
```

20.1 python 的函数调用中参数是引用传递

20.2 可以在定义函数的时候, 在参数列表中通过=设置参数的默认值.

21 类:

21.1 定义:

```
class class_name:
```

static_variable_name = value
def __init__(self, [arguments]):
//operation
//self in here is the reference for this class instance
def general_method_name(self, [arguments]):
//operation
//self is the class instance
//if you want to use class variable, please use like self.__class__.__name__

21.2 实例化: instance = class_name([arguments, …]);

22 模块: 不带.py 后缀名的文件名…一个模块创建之后, 可以使用 import 导入这个模块使用.

22.1 访问模块内的函数或变量: module_name.function() | module_name.variable | module_name.class_name

22.2 sys 模块概览

22.2.1 sys.stdout.write('Hello World!\n') //使用 sys 模块的标准输出

22.2.2 sys.platform //返回系统的标记

22.2.3 sys.version //返回系统的版本

23 PEP: 一个 PEP 就是一个 python 增强提案(python enhancement proposal), 是在新版 python 中增加新特性的方式…索引网址是: http://python.org/dev/peps

24 一些常用函数

24.1 dir([obj]) 显示对象的属性, 如果没有提供参数, 显示全局变量的名字

24.2 help([obj]) 显示对象的文档, 如果没有参数, 进入交互式帮助

24.3 int(obj) 将一个对象转换为整数

24.4 len(obj) 返回对象的长度

24.5 open(file_name, mode) 以 mode(r|w|a…)方式打开一个文件

24.6 range([[start, ]stop[, step]]) 返回一个整数列表…结束值是 stop-1, step 默认是 1

24.7 raw_input(str) 提示 str 等待用户输入

24.8 str(obj) 将一个对象转换为字符串

24.9 type(obj) 返回对象的类型…返回值本身是一个 type 对象

24.10 sum(iterable[, start=0]) 可以对纯数值的 list|tuple|map 进行求和操作..

24.11 dir([object]) 如果没有参数获得当前脚本 scope 内定义的对象, 如果有参数, 返回该对象内部定义的对象, 如果该对象有一个__dir__方法, 该方法将被调用, 并且必须返回属性的列表…这就允许通过自定义__getattr__()或__getattribute__()方法的方式实现 dir 的自定义显示属性列表….如果没有指定参数, 是根据该对象的__dict__内存字典的最佳聚合信息显示的..

24.12 type([object]) 参数为空显示<type 'type'>, 参数不为空显示该对象的类型

24.13 type(name, bases, dict) 通过名称, 基类, 内存字典动态创建一个类型

24.14 object__name.__doc__ 查看该对象的文档字符串

24.15 __doc__ 对象的文档字符串, 该文档字符串在定义对象时写在对象语句块中第一句, 使用单纯的字符串的方式表示

24.16 sys.exit() 退出 python 解释器

24.17 append(Object) 给 list 添加一个元素

24.18 os.linesep 返回的是系统换行符…不同的系统换行符是不同的, 使用 linesep 可以提高代码跨平台性

24.19 string_variable_name.strip([chars]) 脱离, 滤去字符串中的某些字符, 如果没有参数

返回原字符串
25 数值按进制分为:
25.1 二进制: 0b101010
25.2 八进制: 07167
25.3 十进制: 98767
25.4 十六进制: 0xf2134

# Python 基础(chapter3)

1 setence and syntax 语句和语法
1.1 #为注释符号
1.2 \n 是标准行分隔符, 通常一个语句一行
1.3 反斜线\表示下一行继续, 用来将单条语句放入多行…尽量使用括号代替
1.4 分号;表示将两个语句连接在一行中…不提倡
1.5 冒号:表示将代码块的头和体分开
1.6 语句(代码块)用缩进块方式体现: 同一个代码组的代码必须严格左对齐..由于不同的 editor 制表符宽度不一, 为了使代码跨平台, 推荐使用 4 个空格缩进
1.7 不同缩进深度分隔不同的代码块
1.8 python 文件以模块的形式组织: 模块以磁盘文件的形式存在, 不应该让某个模块充斥的太大
2 赋值
2.1 赋值语句没有返回值, 但可以使用链式赋值
2.2 python2.0 开始支持增量赋值(算符和等号连接赋值), 但是 python 不支持++, --
2.3 赋值操作中, 可变对象会被直接修改(引用位置值的修改), 不可变对象则被重新赋予新的对象(引用修改)
2.4 多元赋值: a, b, c = 1, 2, 'string', 建议使用 tuple 的方式进行多元赋值: (a, b, c) = (1, 2, 'string')
3 swap 操作: x, y = y, x
4 标识符
4.1 大小写敏感
4.2 python 的关键字列表和 iskeyword()函数在 keyword 模块, 方便查阅
4.3 内建: built-in 可以看作是系统保留字….对于一些内建函数需要覆盖(重定义, 替换)…built-in 是__builtins__模块的成员, 该模块由解释器自动导入
4.4 python 不支持重载
4.5 下划线: 作为变量前缀和后缀指定特殊变量
4.5.1 _xxx: 不用'from module import*'导入
4.5.2 __xxx__: 系统定义名字
4.5.3 _xxx: 类中的私有变量名
5 python 之禅
The Zen of Python, by Tim Peters
python 之禅. 作者 Tim Peters

Beautiful is better than ugly.

漂亮胜于丑陋
Explicit is better than implicit.
详尽胜于含蓄
Simple is better than complex.
简单胜于复杂
Complex is better than complicated.
组合胜于复杂(结构)
Flat is better than nested.
单一胜于嵌套
Sparse is better than dense.
稀少胜于繁杂
Readability counts.
可读性价值
Special cases aren't special enough to break the rules.
特例不足以违反规则
Although practicality beats purity.
实践胜于理论
Errors should never pass silently.
错误可能从不沉默
Unless explicitly silenced.
除非明白沉默
In the face of ambiguity, refuse the temptation to guess.
面对歧义，不被猜想诱惑
There should be one-- and preferably only one --obvious way to do it.
可能仅有一种更好的方法
Although that way may not be obvious at first unless you're Dutch.

Now is better than never.
现在胜于一切
Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.
难于解释的实现是不好的
If the implementation is easy to explain, it may be a good idea.
易于明白的实现可能是个好方案
Namespaces are one honking great idea -- let's do more of those!
名空间是一个好方案，让我们去超越这些
6 模块组织
# 起始行
# 模块文档
# 模块导入
# 变量定义
# 类定义
# 函数定义

# 主程序
7 __name__用于指示模块应该如何被加载，如果值是”__main__”说明是主模块，如果是模块名，说明是被导入的
8 主模块测试代码
```
def main():
    # business process
if(__name__ == '__main__')
    main()
```
9 内存管理
  9.1 基本
    9.1.1 弱类型 – 动态类型
    9.1.2 programmer 不关心内存管理
    9.1.3 变量并会被回收
    9.1.4 del 语句能够直接释放资源
  9.2 变量未初始化不允许使用
  9.3 引用计数：对于一个内存堆中的对象，有多少个指针引用它..引用计数就是多少，引用计数为 0 时，该对象可以被垃圾回收器回收
    9.3.1 增加
      9.3.1.1 对象创建
      9.3.1.2 别名创建(引用赋值)
      9.3.1.3 参数传递(引用传值)
      9.3.1.4 被容器引用
    9.3.2 减少
      9.3.2.1 引用生命周期结束
      9.3.2.2 对象别名被显示销毁 del y
      9.3.2.3 对象别名被赋予其他引用
      9.3.2.4 窗口对象本身被销毁
    9.3.3 del
      9.3.3.1 从现在的名字空间中删除一个别名
      9.3.3.2 对象的引用计数减一
  9.4 垃圾回收：有对象引用计数为 0，对象被显示告知需要销毁，有内存消耗大户存在导致系统压力较大时，垃圾回收机制运行，清理需要回收的内存区域…垃圾回收机制还有一个循环垃圾回收器，确保释放循环引用对象(a 引用 b, b 引用 a，导致其引用计数永远不为 0)
10 将引入的其他模块中常用的变量重新赋予一个本地别名(ls = os.linesep)不仅可以避免冗长的变量名，又可以提高加载速度(因为现在是本地变量了)
11 重要的其他模块
  11.1 debugger: pdb 允许设置断点，逐行调试，检查堆栈，还支持事后调试
  11.2 logger: logging 分紧急，错误，警告，信息，调试五级日志
  11.3 profilers：性能测试器
    11.3.1 profile: python 编写，测试函数执行时间，每次脚本执行总时间.
    11.3.2 hotshot：python2.2，c 编写，解决了性能测试过载问题，但需要更多时间生成结果，python2.5 修正了 hotshot 的时间计量 bug
    11.3.3 cProfile：python2.5，c 编写，需要较长时间从日志文件载入分析结果，不支持子函

数状态细节，某些结果不准

# python 对象(chapter4)

1 python 对象有三个特征：身份和类型是只读的，如果对象支持不支持更新操作，那么值也就是只读的.

1.1 身份：唯一的身份标识，可以使用内建函数 id()得到，可以看作是对象的内存地址…

1.2 类型：对象的类型决定该对象保存什么类型的值，可以进行什么操作，遵循什么规则.，可以使用内建函数 type()查看 python 对象的类型.

1.3 值：对象表示的数据项

2 对象属性：主要有属性，值，相关联的可执行代码(method)，一些 python 类型也有数据属性，含有数据属性的对象包括但不限于：类，类实例，模块，复数，文件.

3 基础数据类型：数字，整形，布尔型，长整型，浮点型，复数型，字符串，列表，元组，字典.

4 其他内建类型：类型, None，文件，集合/固定集合，函数/方法，模块，类

4.1 type(type(1)) 可以看出类型对象本身也是对象，它的类型是 type

4.2 None 的类型是 NoneType, NoneType 只有一个值，就是 None，不支持任何运算也没有任何内建方法，布尔值总返回 False

5 每个对象天生都有布尔值，以下对象的布尔值是 False

5.1 None

5.2 False(bool 类型)

5.3 所有值为 0 的数

5.4 ” 空字符串

5.5 [] | () | {} 空列表，空元组，空字典

5.6 用户创建的实例如果定义了 nonzero(__nonzero__())或 length(__len__())且值为 0，那么返回的布尔值也是 False

6 当要获取一个对象的布尔值的时候，首先会调用__nonzero__(必须返回布尔类型或者 int 类型.)方法，如果实现了该方法，就返回该方法返回的布尔值，如果没有实现该方法，继续调用__len__方法(该方法返回值必须是大于等于 0 的)，如果实现了__len__方法，就根据其返回值返回布尔值.

7 内部类型: python 内部机制

7.1 代码：编译过的 python 的源代码片段，是可执行对象. 通过调用内建函数 compile()可以得到代码对象. 代码对象可以被 exec 命令或 eval()内建函数来执行. 代码是用户自定义函数的核心，运行期获得上下文.. __code__是函数的一个属性，函数除了代码这个属性，还有函数名，文档字符串，默认参数，全局命名空间等必须的属性

7.2 帧对象：用于跟踪记录对象

7.3 跟踪记录：用于异常发生时，程序访问跟踪记录对象处理程序.

7.4 切片:

7.4.1 步进切片 sequence[::number] number 为负数，倒序显示字符串，正数则正序显示字符串，数值代表显示字符的 step 值.

7.4.2 多维切片 sequence[start1: end1, start2: end2]

7.4.3 省略切片 sequence[…, start1: end1]

7.4.4 切片对象 使用 slice([start, ]stop[, step])可以创建一个切片对象

7.5 省略对象：用于扩展切片语法，起记号作用..在切片语法中表示省略号，省略对象有一

个唯一的名字 Ellipsis，布尔值始终是 True

7.6 Xrange：调用 xrange()生成 Xrange 对象，类似内建函数 range，用于节省内存使用或 range 无法完成的超大数据集场合

8 标准类型运算符

8.1 对象值比较

8.1.1 数字根据大小比较

8.1.2 字符串根据字符先后顺序比较

8.1.3 list | tuple | dict 顺序按照其中元素(dict 按照键)比较

8.1.4 链式比较相当于多个比较使用 and 连接

8.1.5 自定义类型对象的比较是引用值比较，也就是 id(object_name)的比较

8.2 对象身份比较

8.2.1 is / is not 用来比较两个别名是否引用同一个对象

8.2.2 整数对象和字符串对象是不可变对象…

8.3 布尔类型运算符: and, or, not

9 标准类型内建函数

9.1 cmp(obj1, obj2) 比较 obj1 和 obj2，根据比较结果返回整数 i, i<0 → obj1<obj2, i>0 → obj1>obj2, i==0 → obj1 == obj2….自定义类型中通过定义方法__cmp__(target)来实现….使用比较运算符，或直接调用 cmp 时该方法被调用

9.2 type(obj) 得到一个对象的类型，返回相应的 type 对象

9.3 str(obj) 返回对象适合可读性好的字符串表示…在自定义类中使用__str__(self)返回一个字符串，调用 str(obj)时会被隐式调用

9.4 repr(obj) 返回一个对象的字符串表示, repr 返回的字符串通常可以被用于使用 eval 动态创建一个对象. 通常 obj == eval(repr(obj))是成立的

9.5 ``，反单引号, `obj`和 repr(obj)做的事情是一样的.

9.6 isinstance(obj, (type[, type, …])) 判断 obj 是不是第二个参数 tuple 中的列举的类型的实例

9.7 types 模块提供一些已知的类型

9.8 由于每一种类型都只有一个类型对象，所以，可以使用引用比较代替值比较以提升性能：用 if type(num) is types.IntType 替代 if type(num) == types.IntType

9.9 from-import：只引入某个模块的一部分属性：比如 from types import IntType 这样做可以有效的减少查询次数

9.10 python 的 operator 模块中有绝大多数运算符对应的同功能的函数可供使用.

10 类型工厂函数：int(), long(), float(), complex(), str(), Unicode(), basestring(), list(), tuple(), type(), dict(), bool(), set(), frozenset(), object(), classmethod(), staticmethod(), super(), property(), file()

11 标准类型的分类:

11.1 标准类型是”基本内建数据对象原始类型”

11.1.1 基本：是 python 的标准或核心

11.1.2 内建: python 默认提供

11.1.3 数据：用于一般数据存储

11.1.4 对象：对象是数据和功能的默认抽象

11.1.5 原始：这些类型提供的是最底层的粒度数据存储

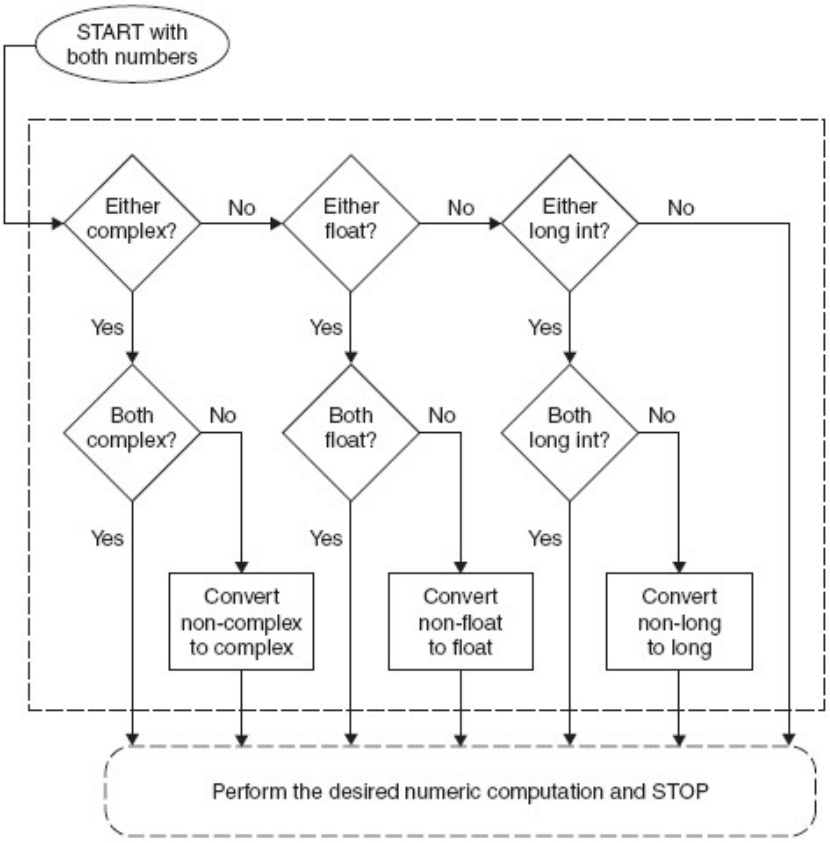
11.1.6 类型：本身就是数据类型

11.2 按存储模型进行分类:

11.2.1 标量/原子类型: 数值, 字符串等可以存储单一字面对象的类型.
11.2.2 容器类型: 列表, 元素, 字典等可以存储多个字面变量的类型, python 中的容器类型都可以存储不同类型的元素
11.3 按更新模型进行分类: 对象创建之后, 值不可以改变, 注意: 这里是对象, 而不是变量
11.3.1 可变类型: 列表, 字典
11.3.2 不可变类型: 数字, 字符串, 元组
11.4 按访问模型进行分类: 访问对象的方式
11.4.1 直接访问: 数值
11.4.2 顺序访问: 列表, 元组, 字符串等可以按照索引访问的类型
11.4.3 映射访问: 字典
12 不支持的类型: char, byte, pointer

# 数字(chapter5)

1 支持的数字类型: 整型, 长整型, 布尔型, 双精度浮点型, 十进制浮点型和复数
2 整型
2.1 布尔型 包含 True 和 False 两个值的整型
2.2 标准整数类型 0x 表示十六进制, 无前缀表示十进制, 0 表示八进制, 0b 表示二进制
2.3 长整型 整数后加 L, 表示范围与可用内存大小有关..
3 双精度浮点数: 一个符号位, 52 个底位, 11 个指数位. 使用后缀 e 表示指数
4 复数: 实数 + 序数 J 构成一个复数
4.1 python 中的复数概念
4.1.1 虚数不能单独存在, 总是和一个值为 0.0 的实数部分一起构成一个复数
4.1.2 复数由实数部分和虚数部分组成
4.1.3 表示虚数的语法: real + imagJ
4.1.4 实数部分和虚数部分都是浮点数
4.1.5 虚数部分必须有后缀 j 或 J
4.2 复数的内建属性
4.2.1 real 复数的实部
4.2.2 imag 复数的虚部
4.2.3 conjugate() 返回该复数的共轭复数
5 强制类型转换规则




6 python 提供 python 解释器的启动参数 Qdivision_style, -Qnew 执行新的除法行为, -Qold 执行传统除法行为, 默认是-Qold…-Qwarn 和-Qwarnall 可以用来度过过渡期

7 幂运算符** 比左侧单目运算符优先级高, 比右侧单目运算符优先级高

8 位运算符: ~, &, |, ^, <<, >>

8.1 负数会被当成正数的二进制补码处理

8.2 左移和右移 N 位等同与无溢出检查的 2 的 N 次幂运算

8.3 长整数的位运算使用一种经过修改的二进制补码形式, 使的符号位可以无限左移

8.4 ~是单目运算符

9 内建函数和工厂函数

9.1 数字类型函数:

9.1.1 转换工厂函数: int(), long(), float(), complex(), 接受参数第一个是要转换的值, 第二个是进制..如果要转换的值是 string 才能使用第二个函数指定进制.

9.1.1.1 python2.2 开始, 加入了 bool(), 用来将整数 1 和 0 转换成为标准布尔值(所有非 0 数都返回 True).

9.1.1.2 python2.3 的标准数据类型添加了 Boolean 类型, true 和 false 也有了常量值 True 和 False, 而不再是 1 和 0

9.1.1.3 bool(obj) 返回 obj 对象的布尔值, 也就是 obj.__nonzero__()

9.1.1.4 complex(real, imag=0.0)

9.1.2 功能函数

9.1.2.1 abs(number) 返回数字的绝对值, 如果是复数, 返回 math.sqrt(num.real2 + num.imag2)

9.1.2.2 coerce(number1, number2): 返回按照类型转换规则转换得到的两个数字组成的元组

9.1.2.3 divmod(number1, number2) 返回一个包含商和余数的元组，整数返回地板除和取余操作结果，浮点数返回的商部分是 math.floor(number1/number2)，复数的商部分是 ath.floor((number1/number2).real)

9.1.2.4 pow()和**功能相同

9.1.2.5 round(number[, base]) 对浮点数进行四舍五入运算, base 参数是小数位参数，如果不指定，返回与第一个参数最接近的整数的浮点形式

9.1.2.6 int() 直接去掉小数部分, floor()得到最接近但小于原数的整数, round()得到最接近原数的整数

9.1.3 整数的内建函数: hex(), oct(), bin()…ord("A") 接受一个 ascii 或 unicode 字符，返回相应的 unicode 值 ←→ char(65L)接受 unicode 码值，返回对应的 unicode 字符.

10 其他数字类型

10.1 布尔数:

10.1.1 布尔型是整型的子类，但是不能再被继承而生成它的子类.

10.1.2 没有__nonzero__()方法的对象默认值是 True

10.1.3 对于值为 0 的任何数字或空集(空的 list|tuple|dict)在 python 中值都是 False

10.1.4 数学运算中, True ==1, False == 0

10.2 十进制浮点数: from decimal import Decimal

11 数字科学计算的包

11.1 高级的 Third package: Numeric(NumPy)

11.2 python 自带的数字类型相关模块

11.2.1 decimal 十进制浮点运算类 Decimal

11.2.2 array 高效数值数组(字符，整数，浮点数)

11.2.3 match/cmatch 标准 c 库数学运算函数. 常规数学运算在 match，复数运算在 cmatch

11.2.4 operator 数字运算符的函数实现

11.2.5 random 伪随机数生成器

11.2.5.1 randint(start, end)随机生成 start, end 之间的一个整数

11.2.5.2 uniform(start, end)随机生成范围内的一个浮点数

11.2.5.3 randrange([start ,]stop[, step])随机生成 start, stop 内按 step 步增数字范围的一个整数

11.2.5.4 choice(sequence)随机返回给定的序列中的一个元素

# 序列：字符串，列表和元组(chapter6)

1 对所有序列类型都适用的操作符(优先级从高到低，不适用于复合类型的对象比较)

1.1 成员关系操作符: int, not in

1.2 连接操作符: +

1.3 重复操作符: *… sequence * int

1.4 切片操作符: (利用子序列方式结合三种操作方式，可以非常灵活的控制序列)

1.4.1 []，索引取值

1.4.2 [start : end]，索引范围取值

1.4.3 [::step]，切片的步长

1.4.4 切片操作符不会带来索引超出下标的异常

2 list.extend(iterable): 把另外一个序列追加到 list 中.

3 list.insert(index, object): 把一个元素 object 插入到 list 的 index 位置, 如果 index 是负数, 从后面向前数, 超过 list 下标后, 在末尾添加

4 类型转换

4.1 list(iter) 把可迭代对象转换为列表

4.2 str(obj) 把 obj 对象转换成字符串(对象的字符串表示法)

4.3 unicode(obj) 把对象转换成 Unicode 字符串(使用默认编码), 使用 u"汉字"可以得到其 unicode 编码

4.4 basestring() 抽象工厂函数, 不能被实例化, 不能被调用, 仅作为 str 和 unicode 的父类

4.5 tuple(iter) 把一个可迭代对象转换成元组对象

4.6 enumerate(iterable) 生成由 iterable 每个元素的 index 值和 item 值组成的元组, 可以使用 for key, value in enumerate 的方式进行迭代

4.7 max(iterable, key=None) | max(arg0, arg1, …, key=None) 返回 iterable 或 arg0…中的最大值, 如果要指定 key, 必须是一个可以传递给 sort()方法的回调函数…..要使用 key, 必须使用 key=method_name 的方式传参, key 指定的函数接收的参数是迭代的当前元素, 在该函数中, 对元素进行处理, 返回一个对象, python 会根据返回对象比较大小, 得到该结果最大的元素对应的 list 中的元素

4.8 min 同上面的 max 方法. 对于 max 和 min 方法, 如果是自定义类型, 又没有指定 key, 那么默认是按照 id(object)的结果计算的

4.9 reversed(sequence) 返回逆序访问的迭代器

4.10 sum(sequence, init=0) 返回 sequence 和可选参数 init 的总和, 等同于 reduce(operator.add, sequence, init)

4.11 sorted(iterable, func=None, key=None, reverse=False) 接受一个可迭代对象, 返回一个有序列表, 可选参数 func, key, reverse 含义和 list.sort 相同

4.12 zip([item0, item1, …, itemn]) 返回一个列表, [(item0, ), (item1, ), …, (itemn, )]

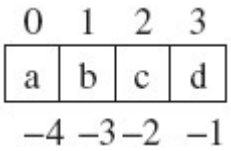
4.13 sort(cmp=None, key=None, reverse=False) 将序列进行排序, cmp 指定一个接受两个参数的回调函数, 该函数得到的两个参数是序列中的两个元素, 比较将按照 cmp 指定的回调函数进行, 返回数字类型的比较结果, , , key 指定一个接受一个参数的回调函数句柄, 该参数就是迭代到的元素, 在比较之前, 将会根据这个回调函数对要比较的元素进行一次处理, 实际参与比较的是经过该回调函数处理之后的返回值. reverse 指示是否对比较结果进行逆序

5 利用已有功能函数定义动态参数的函数: method = lambda x, y: cmp(x + 10, y - 10), 调用时还是使用句柄加参数的方式: method(x, y)

6 字符串: 如果先使用切片操作, 子串会被在内存中进行短时间的暂存, 可以通过 id()得到值

7 比较: 普通字符串按照 ascii 值比较, Unicode 字符串按照 unicode 值比较.

8 字符串切片, 下图是字符串切片的索引值, 如果开始或结束的索引值没有指定, 默认为 0, 或-1. [::step]用于指定步长




9 成员操作符: in, not in, 可以判断一个子串是否在字符串中存在…使用 find(), index(), rfind(), rindex()可以获得子串在源中的位置

10 string 模块有一些预定义的字符串: ascii_letters, ascii_lowercase, ascii_uppercase, digits

11 循环的改善: 如果循环的终止条件是不变的(一般都是不变的), 那么尽量不在循环终止条件中调用方法是一个好的习惯, 在大量数据的情况下(5 * 10^8 数据), 改善的循环效率提升了 4 秒左右, 平均计算得到改善后循环每次效率提升约为 7.154 * 10^-8s 也就是 71.54 ns

12 for-else 方式的循环, else 子句会在所有的元素都被循环完成之后执行, 如果 break, 就不执行

13 join 可以用来连接字符串, 这样的效率是更高的, 因为+连接必须为每个字符串创建内存

14 +连接字符串是运行时连接, “str1””str2”的方式则是编译时连接字符串

15 普通字符串和 unicode 字符串连接, 会把普通字符串转换成 unicode 字符串进行连接

16 %格式化字符串的参数:

16.1 %c 转换成字符(ascii 值, 或长度为一的字符串)

16.2 %r 优先使用 repr()函数进行字符串转换

16.3 %s 优先使用 str()函数进行字符串转换

16.4 %d / %i 转成有符号的十进制数

16.5 %u 转成无符号的十进制数 遇到负号添加-号

16.6 %o 转成无符号八进制数 遇到负号添加-号

16.7 %x / %X 转成无符号十六进制数(x|X 的大小写决定转换后得到十六进制数中的字母的大小写) 遇到负数, 则转换得到的结果中添加一个-号

16.8 %e / %E 转成科学计数法(e | E 的大小写决定转换后得到的 e 的大小写)

16.9 %f / %F 转成浮点数(小数部分自然截断)

16.10 %% 输出%号

17 其他格式化操作符辅助指令(位于%和格式化标识的中间)

17.1 * 定义宽度或小数点精度"adfas%*dfasdf" % (5, 2.000000888)

17.2 – 用于左对齐

17.3 + 在正数前加+号

17.4 (sp) 在正数前显示空格

17.5 # 在八进制前加 0, 十六进制前显示 0x 或 0X, 取决于用的是 x 或 X 比如: "integer:%#X!" % 1984

17.6 (var) 映射变量(字典参数)

17.7 m.n m 是显示的最小总宽度, n 是小数点后的位数

18 格式化字符串接收 dict 数据:”%(howmany)d days” % {“howmany”: 28}

19 字符串模板 s = Template(‘There are ${howmany} ${lang} Quotation Symbols’)

19.1 s.substitute([key = value, key = value…]) 这个函数必须提供所有的占位参数, 否则报错

19.2 s.safe_substitute([key = value, key = value…]) 和 substitute 功能是一样的, 都是转成得到一个字符串, 但是这个方法对参数没有要求, 如果没有提供对应的参数, 就直接输出.

20 原始字符串操作符(r/R), 紧贴字符串左引号, 不区分大小写. 用来使字符串描述各自原始的意义, 而不使用转义

21 使用正则查找空白字符: m = re.search(r‘\\[rtfvn]’, r’Hello World!\n’)…

22 可以使用 u | U ‘字符串’的方式创建 unicode 字符串, 该标识可以和 r/R 连用

23 python 参数有位置参数和关键字参数两种, 位置参数在定义时只有参数名, 关键字参数定义时是 key=value 的形式

24 python 也提供可变参, *为位置可变参, **为关键字可变参

25 如果使用* | **方式传递实参, * 可以将序列参数转变成每个元素作为单独参数, **则可以将 dict 转换成关键字参数

26 内建函数

26.1 cmp，使用字符的 ascii 值进行比较(Unicode 字符串按照 unicode 值比较)

26.2 max, min, len, enumerate, zip，其中 zip 可以接受多个参数，按照下面方式返回:

zip(“abcd”, “efg”, “hijk”, “lmn”) 返回: [('a', 'e', 'h', 'l'), ('b', 'f', 'i', 'm'), ('c', 'g', 'j', 'n')]

26.3 str 和 unicode 都是 basestring 的特化类，但是, Unicode 又包含(类之间仅仅是兄弟关系，元素范围上有包含关系)str 的表示范围

26.4 chr(number), unichr(number), ord(string), chr 和 unichr 分别用来把一个数字转换成一个对应类型的字符串，ord 则是将一个 string 类型的单字符对象转换成为对应的 ascii 码或 unicode 编码

27 string 模块的重要函数，所有这些函数，都可以省略第一个参数，使用 string.func 的方式调用

27.1 string.index()和 string.find()方法是一样的功能，但是，index 方法在查找不到字符串的时候，会报一个异常, string.rfind(), string.rindex()用法相同，是从右边开始查找

27.2 string.join(sequence[, str]) 如果只有一个参数，返回一个将所有元素插空一个空格的字符串(如果是序列，转换成为字符串)，如果有两个参数，把第二个参数向第一个参数的每个空位进行插空

27.3 string.ljust(string, width[, fillchar]) 填充 fillchar(只能是一个字符)width 个到 string 后面，使 string 左对齐，如果 fillchar 空，填充空格, rjust 为右对齐

27.4 string.lower(), string.upper(), string.swapcase()大小写转换

27.5 string.lstrip(), string.rstrip(), string.strip() 去除空格.

27.6 string.split(string, sub, count) 将 string 截取 sub，从左向右截取 count 个，返回 list

27.7 string.replace(string, old, new[, number=string.count(string, old)]) 将 string 的 old 子串替换成 new，最多替换不超过 number, number 默认是 old 在 string 的数量

27.8 string.translate

27.9 string.zfill(string, width) 用 width 个 0 填充 string 的左面使其右对齐

28 unicode.encode(CODE_STRING) 按照指定编码方式编码字符串，decode 反之，按照指定编码方式解码

29 除了 pickle 模块之外，其他模块都已经支持 unicode..

30 UnicodeError 异常在 exceptions 模块中定义，是 ValueError 的子类，所有关于 Unicode 编解码的异常都要继承自 UnicodeError

31 与字符串类型有关的模块:

31.1 string: 字符串相关操作函数和工具，比如 Template 类

31.2 re: 正则表达式

31.3 struct: 字符串和二进制之间的转换

31.4 c/StringIO 字符串缓冲对象，操作方法类似于 file 对象

31.5 base64 Base16, 32, 64 数据编解码

31.6 codecs 解码器注册和基类

31.7 crypt 进行单方面加密

31.8 difflib 找出序列间的不同

31.9 hashlib 多种不同安全哈希算法和信息摘要算法的 API

31.10 hma HMAC 信息鉴权算法的 python 实现

31.11 md5 RSA 的 MD5 信息摘要鉴权

31.12 rotor 提供多平台的加解密服务

31.13 sha NIAT 的安全哈希算法 SHA

31.14 stringprep 提供用于 IP 协议的 Unicode 字符串
31.15 textwrap 文本打包和填充
31.16 unicodedata Unicode 数据库
32 字符串的关键点:
32.1 不可分字符类型
32.2 类似 printf()的格式化字符串
32.3 三引号内可以接受特殊字符(What You See Is What You Get)
32.4 r | R 原始字符串
32.5 python 字符串不通过 NUL 或者\0 结束
33 list 的元素删除
33.1 del list[index] 根据 index 删除 list 中的元素
33.2 list.remove(value) 从 list 中移除值是 value 的第一个元素
33.3 list.pop() 类似栈的出栈操作, 弹出栈顶并返回
34 list 的成员关系操作还是使用 in | not in
35 列表解析: [var_name for var_name in list if expression]if 之后和 for 之前都可以使用 var_name 进行运算
36 列表的比较操作, 隐式调用 cmp 方法, 比较规则是逐个扫描元素, 进行比较, 如果可以比较, 比较, 如果相等扫描下一个元素, 如果不相等返回结果, 如果两个元素类型不可以比较, 就比较两个对象的 id()值.. 如果一直相等 ,直到一个列表扫描结束, 那么返回较长的列表较大
37 序列类型函数
37.1 len(), max(), min()
37.2 sorted() 和 reversed() 返回的是被排序或逆序之后的序列, 不会改变序列本身的引用, 而序列自己的成员方法则会改变自身
37.3 enumerate() 返回一个 key=>value 方式的 enumerate 对象
37.4 zip() 将多个列表压缩成为一个元组列表.....zip 返回的元组列表可以使用足够元组内元素数量的参数来迭代遍历每一个元素, 例如: for a, b, c, d in zip(alist, blist, clist, dlist)
37.5 使用 list()和 tuple()可以完成列表和元组之间的转换, 但是这种转换是值的转换, 所以他们是==的, 但是不是 is 的
37.6 extend(列表)方法接受另外一个列表追加到原列表之后
37.7 list.pop(index = -1) list 的 pop 可以弹出指定索引的值
38 处理一组对象的时候, 默认的是创建一个元组, 例如 a = 1, 2, 3, 4 实际上是创建了一个元组(1, 2, 3, 4)
39 单独使用 del 删除一个元组元素是不可行的, 只能通过重组
40 元组的可变性: 元组的某一个元素指向了一个对象, 该对象是可变的, 那么改变该对象就相当于改变了元组的内容, 然而, 真正的我们的元组确实是没有改变的. It's so wizardly. 为什么呢? 元组内部持有的是对方的引用, 那个对象无论怎么变都还是在那里, 所以, 元组内部的值(内存地址)是没有改变的.
41 函数可以返回多对象, 返回的实际也是一个元组
42 单元素元组使用括号创建时需要在后面显示的加上", ", 由于括号被重载作为一个分组操作符了, 在这里会优先使用分组功能, 所以, 返回的总是原始类型
43 相关模块:
43.1 数组 array 受限的可变序列类型, 要求所有元素都是相同的类型
43.2 operator 包含函数调用形式的序列操作符, operator.concat(m, n)相当于 m+n

43.3 re　perl 风格的正则查找

43.4 StringIO / cStringIO　长字符串作为文件来操作，比如 read()，seek()函数……c 版本的速度更快一些

43.5 Textwrap　包裹/填充文本的函数

43.6 types　包含 python 支持的所有类型

43.7 collections　高性能容器数据类型

43.8 UserList　包含了 list 对象的完全的类实现，允许用户获得类似 list 的类，用以派生新的类和功能

44 浅拷贝：拷贝原对象中的内容，但是新创建对象．比如一个 list 的浅拷贝就是把 list 中元素的引用值拷贝过去…　浅拷贝实例：完全切片操作，利用工厂函数，使用 copy 模块的 copy 函数

45 深拷贝：拷贝源对象中的内容，如果某个属性(或序列中的元素)是可变对象，把该可变对象的内容也进行拷贝

46 非容器类型对象没有拷贝一说

47 可变参：f(*args1, **args2)，如果关键字可变参不指定 key 值，会被作为位置参数和前面的参数放到一个元组中….位置可变参在倒数第二个位置，关键字可变参在倒数第一个位置．

48 map(function, iterable)，应用 function 在 iterable 的每一个元素，返回值作为新的序列的元素

# 映射和集合类型(chapter7)

1 字典使用 keys()获得键的列表，values()获得值的列表，items()获得包含 key->value 对的元组的列表

2 字典的创建和赋值

2.1 dict = {key: value}

2.2 dict = dict((key, value), (key, value))

2.3 dict.fromkeys(sequence_keys, default_value)　创建一个 key 是 sequence_keys 中元素的字典，所有的 value 都是 default_value，如果不指定 default_value，默认是 None

3 使用 dict.has_key()可以判断一个字典中是否有这个键，该方法在后期 python 可能弃用，推荐使用 in 和 not in

4 字典的键必须是可哈希的

5 print 中使用到字典的时候，使用字符串格式化方式是非常优雅的

6 dict1.update(dict2 | tuple_list[, **key=value])

6.1 将 dict2 字典更新到 dict1 中

6.2 如果参数是一个元组列表，将元组列表解析到 dict1 中(元组列表中每个元组必须有两个元素)

6.3 可以在参数后面跟 0—n 个关键字参数，以参数名: value 的方式更新到 dict1 中

7 元素的删除

7.1 del dict[“key”]　删除键是 key 的条目

7.2 dict.clear()　清空字典内的内容

7.3 dict.pop(“name”)　删除键是 key 的条目并返回

8 映射类型操作符

8.1 标准类型操作符：<, >等比较操作符可以使用，在比较过程中，还是调用了字典的 cmp

方法，但是，字典的 cmp 方法中指示，首先比较字典的长度，然后比较键的大小，最后比较值的大小

8.2 字典查找操作: []，成员关系操作: in, not in

9 dict 工厂函数

9.1 接受不定关键字参数: dict(a = 1, b = 2, c = 3)

9.2 接受字典或关键字参数: 将原有的字典拷贝出来成为一个新的字典(这里使用的浅拷贝，这里的浅拷贝得到的结果和使用 copy 函数得到的结果是一样的,但是, copy 函数的效率更高)

9.3 dict_instance.copy() 使用已有的字典拷贝一个字典(这里使用的也是浅拷贝)

10 系统内建函数

10.1 len(): 返回字典的 key-value 对的数目

10.2 hash(): 这个函数不是为字典设计的，但是它可以判断某个对象是否可以做一个字典的键，将一个对象作为参数传递给 hash()，会返回这个对象的哈希值. 如果对象是不可哈希的，会返回 TypeError，提示该对象是 unhashable 的

10.3 sorted(dict): 返回 dict 的所有 key 排序后的列表

11 dict 类型的内建方法

11.1 keys(): 返回字典的所有 key 的列表

11.2 values(): 返回字典的所有 value 的列表

11.3 items(): 返回字典的所有 key-value 的元组的列表

11.4 get(key, default = None): 获取字典内 key 对应的值，如果没有该 key，返回 default 指定的默认值

11.5 setdefault(key, default=None): 如果字典中不存在 key，由 dict[key]=default 为其赋值

11.6 iterkeys(), itervalues(), iteritems() 对应没有 iter 命名的方法，这里使用了惰性赋值的迭代器，节省内存

12 数字作为字典的键的时候，只要值相同，就代表相同的键，比如，1，和 1.0 代表的就是相同的键

13 键必须是可哈希的，所有的不可变对象都是可哈希的，可变对象如果在定义中定义了__hash__()方法，那么就可以作为键

14 如果元组中的值都是不可变类型的，那么元组也可以作为字典的键

15 集合 sets 有两种不同的类型，可变集合 set 和不可变集合 frozenset…可变集合也是不可哈希的…

16 集合中不能有重复的元素，如果有元素和已有元素重复，就不会被插入.集合是无序的，但是，可以使用排序函数为它排序

17 集合中可以使用的数学符号

| | | |
|---|---|---|
| ∈ | in | 是...的成员 |
| ∉ | not in | 不是...的成员 |
| = | == | 等于 |
| ≠ | != | 不等于 |
| ⊂ | < | 是...的(严格)子集 |
| ⊆ | <= | 是...的子集(包括非严格子集) |
| ⊃ | > | 是...的(严格)超集 |
| ⊇ | >= | 是...的超集(包括非严格超集) |
| ∩ | & | 交集 |
| ∪ | \| | 合集 |
| - or \ | - | 差补或相对补集 |
| Δ | ^ | 对称差分 |

18 集合只能使用工厂函数 set 和 frozenset 创建

19 访问集合中的数据使用循环，或者使用成员关系操作符 in, not in 判断元素是否属于集合

20 更新集合(只能是 set, frozenset 不能被更新)

20.1 add()　添加一个元素

20.2 s.update()　接受一个序列类型的参数，把该序列中有而集合中没有的元素添加到集合中

20.3 s.remove()　从集合中移除一个元素

21 集合可用的标准类型操作符

21.1 成员关系: in, not in

21.2 集合等价/不等价: ==, !=

21.3 子集/超集: <, <=, >, >=

22 集合类型操作符(所有的集合类型)　集合互相操作的时候，最后产生的集合是可变集合还是不可变集合取决于第一个参与操作的集合的类型

22.1 联合 |

22.2 交集 &

22.3 差补/相对补集　-　A-B 就返回属于 A 但不属于 B 的元素集合

22.4 对称差分　^　A^B = A-B + B-A

23 可变集合特有的操作符

23.1 |=　相当于 update 方法，并集赋值

23.2 &=　相当于 intersection_update()方法，交集赋值

23.3 -=　相当于 difference_update()方法，差集赋值

23.4 ^=　相当于 symmetric_difference_update()方法，对称差分更新

24 集合类型的内部方法:

24.1 s.issubset(t)　判断 s 是不是 t 的子集

24.2 s.issuperset(t)　判断 s 是不是 t 的超集

24.3 s.union(t)　返回一个新集合，该集合是 s 和 t 的并集

24.4 s.intersection(t)　返回一个新集合，该集合是 s 和 t 的交集

24.5 s.difference(t)　返回一个新集合，该集合是 s 的成员，但不是 t 的成员，即返回 s 不同于 t 的元素

24.6 s.symmetric_defference(t)　返回所有 s 和 t 独有的(非共同拥有)元素集合

24.7 s.copy()　返回一个 s 的浅拷贝，效率比工厂要好

25 可变集合特有的方法: add, remove, discard, pop, clear, 这些接受对象的方法，参数必须是可哈希的

26 那些和操作符提供相同功能的函数，有着更强的处理能力，因为运算符两边的操作数必须都是集合，然而函数可以接受任何的可迭代类型.

# 条件和循环(chapter8)

1 如果循环体，或条件语句体只有一句，可以和头写在同一行.

2 字典的搜索速度要比 for, if, while 等快得多

3 True and object，返回 object, True or object 返回 True，因此可以使用 expression and out1 or out2 模拟三元操作符

4 python 提供的三元操作: X if C else Y，如果 c 输出 x，否则输出 y

5 xrange()不会在内存中创建列表的完整拷贝，只被用在 for 循环中.

6 reversed()和 enumerate()内建函数返回的是一个迭代器

7 迭代器的优点

7.1 提供了可扩展的迭代器接口

7.2 对列表迭代带来了性能上的增强

7.3 在字典迭代中性能提升

7.4 创建真正的迭代接口,而不是原来的随机对象访问

7.5 与所有已经存在的用户定义的类以及扩展的模拟序列和映射的对象向后兼容

7.6 迭代非序列集合(例如映射和文件)时，可以创建更简洁的代码

8 迭代器工作原理: next()方法取出元素，元素全部取完后触发 StopIteration 异常，for 可以根据异常结束循环

9 any(), all()用来判断序列或迭代器中元素的真假, any 为或操作, all 为与操作

10 iter(iterable) 用来创建一个迭代器

11 文件的迭代器中调用 next 方法，默认会调用 readline 方法.　如果直接使用 for line in file 就可以创建一个默认的文件迭代器.

12 在迭代的时候，不要尝试改变可变对象

13 列表解析:

13.1 lambda，动态创建函数

13.2 map(function, sequence)，对 sequence 的每一个元素应用 function，返回一个新的列表

13.3 filter(function, sequence)，对 sequence 的每一个元素应用 function，某个元素应用该 function 后如果返回 false，该元素将被从最后返回的列表中过滤掉

13.4 列表解析中可以使用多个循环，来达到生成矩阵或多维序列的目的..例如【(x, y) for x in range(5) for y in range(3)】 越靠后的循环就相当于普通循环中的内层循环

14 生成器

14.1 生成器表达式和列表解析表达式是一样的， 但是生成器表达式没有中括号， 单独使用的时候， 使用圆括号， 作为参数的时候， 不需要圆括号

14.2 生成器表达式可以被用作迭代

14.3 生成器表达式是 lazy 的

14.4 示例: 获取文件的最长的行的长度: logest = max(len(line.strip()) for line in file)

# 文件和输入输出(chapter9)

1 文件只是连续的字节序列

2 file_object = open(file_name, access_mode = 'r', buffering = -1) mode 有 r, w, a, 分别表示读取, 写入, 追加. U 模式代表通用换行符支持

2.1 使用 r, U 打开的文件必须是已经存在的, 使用 w 打开的文件如果存在, 首先清空…使用 a 模式打开的文件是为追加数据做准备, 所有写入数据被追加到文件末尾. +代表可读可写, b 代表二进制访问… POSIX 兼容的 Unix 系统中, 'b'是可有可无的, 因为它们把所有的文件都当作二进制文件. 包括文本文件

2.2 如果在 mode 中要使用 b, b 不能作为第一个字符出现

2.3 buffering, 表示缓冲区大小, 0 表示不缓冲, 1 表示只缓冲一行数据, 其他大于 1 的值表示使用定值作为缓冲区大小. 不提供该参数或给定负值代表使用系统默认的缓冲机制

2.4 访问模式:

2.4.1 r 只读方式打开

2.4.2 rU 或 Ua 读方式打开, 同时提供通用换行符支持

2.4.3 w 写方式打开

2.4.4 a 追加模式打开, 必要时创建新文件

2.4.5 r+ 读写方式打开

2.4.6 w+ 读写方式打开

2.4.7 a+ 读写方式打开

2.4.8 rb 二进制读模式打开

2.4.9 wb 二进制写模式打开

2.4.10 ab 二进制追加模式打开

2.4.11 rb+ 二进制读写模式

2.4.12 wb+ 二进制读写模式

2.4.13 ab+ 二进制读写模式

3 file()内建函数和 open 的使用方式一样…但是推荐使用 open, 除非特定的要处理文件对象时, 使用 file()内建函数

4 通用换行符支持: 任何系统下的文件, 不管换行符是什么, 使用 U 模式打开时, 换行符都会被替换为 NEWLINE(\n)

5 python 默认打开 UNS, 如果不需要通用换行符支持, 在运行 configure 脚本时, 可以使用—without-universal-newlines 开关关闭.

6 文件方法:

6.1 输入

6.1.1 read(size=-1): 直接读取字节到字符串中, 最多读取给定数目个字节. 不指定 size 参数, 或 size 值为负, 文件将被读取到末尾.

6.1.2 readline(size = -1) size 为负数或不指定, 读取至行末, 如果指定了其他的 size 值, 读取 size 个字节 读到的换行符不会自动去掉, 而把这个操作留给程序员, 使用 strip 去掉

6.1.3 readlines(sizhint = -1) sizhint 和 size 功能相同, 如果指定, 会读取 sizhint 个字节, 实

际读取值可能比 sizhint 较大，因为需要填充缓冲区

6.1.4 file.xreadlines()和 xreadlines.xreadlines(file) 是一种更高校的文件读取，一次读取一块，而不是一次读取所有行，但是，由于 iter 取代了该功能，所以会被废弃.

6.2 输出

6.2.1 write() 把含有文本数据或二进制数据块的字符串写入到文件中

6.2.2 writelines() 接受一个字符串列表作为参数，将他们写如文件，行结束符不会自动加入，如果需要，必须在调用 writelines 之前手动在每行结尾加上换行符

6.2.3 seek(offset, whence) 随机文件访问，第一个参数 offset 表示偏移量，第二个参数表示位置，即第一个参数相对哪里去偏移

6.3 迭代: 直接似乎用 for 循环迭代

6.4 其他:

6.4.1 close() 关闭文件结束对它的访问. 垃圾回收机制也会在文件对象的引用计数降至 0 的时候自动关闭文件.

6.4.2 fileno() 返回打开文件的描述符…是一个整数，可以用在如 os 模块的 read 方法等一些底层操作上.

6.4.3 flush()方法直接把内部缓冲区的数据立刻写入文件，而不是被动的等待输出缓冲区写入.

6.4.4 tell()方法返回文件的当前位置，对应 seek 方法的 offset 参数

6.4.5 isatty() 是一个布尔内建函数，当文件是一个类 tty 设备时返回 true，否则返回 false.

6.4.6 truncate()方法将文件截取到当前文件指针位置或者到给定 size，以字节为单位

6.4.7 next() 返回文件的下一行

7 os 模块的重要属性

7.1 linesep 跨系统的分隔行的字符

7.2 sep 分割文件路径名的字符串( / \ 路径内的不同级分割)

7.3 pathsep 分割文件路径的字符串(; 多个路径分割)

7.4 curdir 当前工作目录的字符串名称(. )

7.5 pardir 当前工作目录的父目录的字符串名称(.. )

8 文件的内建属性:

8.1 name 文件名，打开文件或使用 file()工厂的时候传递的文件名参数

8.2 mode 打开模式

8.3 closed 是否已经关闭

8.4 encoding 编码方式, None 表示使用系统默认的编码

8.5 newlines 未读取到行分隔符时为 None，只有一个种行分割符时为一个字符串，当文件有多种类型的行结束符时，则为一个包含所有当前遇到的行结束符的列表

8.6 softspace 0 表示在输出一数据后，要加上一个空格符, 1 表示不加

9 标准文件, sys 模块下的 stdin, stdout, stderr 分别是标准输入文件(键盘)，标准输出文件(缓冲的屏幕输出)，标准错误(到屏幕的非缓冲输出) 这三个文件是预先打开的，可以直接使用

10 sys.argv 是命令行参数的列表，列表中第一个是当前 python 脚本的名称，后面分别是 python 脚本在命令行下被调用时传入的参数，类似于 java 的 main 方法接受的参数，len(sys.argv)是命令行参数的个数

11 其他提供命令行参数的辅助处理模块:

11.1 getopt 模块，简单的命令行参数处理

11.2 optparse 模块，复杂的命令行参数处理.

12 os 模块: 所有操作系统对外的门面，主要提供删除/重命名文件，遍历目录树，管理文件

访问权限等功能

12.1 文件处理

12.1.1 mkfifo()/mknod()　创建命名管道/创建文件系统节点

12.1.2 remove()/unlink()　删除文件，指定路径为参数进行删除，如果文件不存在会抛出异常

12.1.3 rename()/renames()　重命名文件，接受旧文件路径和新文件路径，进行名字的修改，这里可以通过指定新的路径达到文件移动的目的. 没有发现 renames 和 rename 的区别

12.1.4 stat()　返回文件信息，包含文件的 mode，编号，用户(创建者，所有者，组)，时间等信息

12.1.5 symlink()　创建符号链接　….已经废除???

12.1.6 utime()　更新时间戳

12.1.7 tmpfile()　创建并打开('w+b')一个新的临时文件　没有参数，直接获取一个临时文件

12.1.8 walk(top, [topdown=True[, onerror=None[, followlinks=False]]]) 生成一个目录树下的所有文件名.

12.1.8.1 该方法用于生成目录树下的所有子文件名，该方法返回一个生成器，遍历生成器可以得到一个三个元素的元组(dirpath, dirnames, filenames)

12.1.8.2 返回元素元组中的 dirpath 是 top 的目录名, dirnames 是 top 下所有子目录的文件名集合，不包含当前目录.和上级目录..，filenames 是 top 下的所有的非目录类型的文件的名字集合

12.1.8.3 walk 返回元组中的名字集合可以使用 os.path.join(dirpath, name)得到一个文件或目录的全路径.

12.1.8.4 walk 返回的生成器包含了 top 目录之下所有子孙目录的内部结构

12.1.8.5 walk 的第二个参数 topdown 指示了对该目录树的遍历是否采用自上而下的方式，默认是 True

12.1.8.6 walk 的第三个参数是指示错误发生时是否处理，接收的是一个回调函数，函数会得到一个参数 OSError 的实例

12.1.8.7 walk 的第四个参数是指示是否遍历 link 类型指向的目录，默认是 false，但是，如果你设置了 followlinks=True，并有一个 link 是指向它的父亲或更高辈分的目录，就会产生一个无穷递归.

12.1.8.8 注意:　如果使用 walk 的时候传递了一个相对路径，在重新开始 walk 之前，不要改变工作目录..　当前工作路径的改变会导致 walk 停止工作

12.2 目录/文件夹处理

12.2.1 文件的 fileno()返回该文件的文件描述符

12.2.2 chdir()/fchdir()　改变当前工作目录/通过一个文件描述符改变当前工作目录

12.2.3 chroot()　改变当前进程的根目录

12.2.4 listdir()　列出指定目录的文件，返回参数指定的路径下的所有文件和目录，不包括当前目录.和父亲..，也不进行递归

12.2.5 getcwd()/getcwdu()　返回当前工作目录/功能相同，但返回一个 unicode 对象

12.2.6 mkdir()/makedirs()　创建目录/创建多层目录，接受两个参数，一个表示路径，一个表示模式(权限???)　mkdir()只能用于创建一级目录，如果父目录不存在会抛出异常，而 makedirs()会向上递归的创建目录

12.2.7 rmdir()/removedirs()　删除目录/删除多层目录　rmdir 删除一个目录或文件，

removedirs()则递归删除一个目录树，removedirs()是一个难懂的函数，指定一个父亲无法删除父亲下的所有孩子，反而由孩子删除了自己的父亲和祖辈

12.3 访问/权限

12.3.1 access()　检验权限模式，两个参数，第一个是路径，第二个是模式，模式可以是 F_OK，用来检测路径是否存在，R_OK 用来检测是否可读，W_OK 用来检测是否可写，X_OK 用来检测是否可执行.　返回布尔值表示检验结果

12.3.2 chmod()　改变权限模式　接受两个参数，第一个是路径，第二个是模式，模式在 stat 模块下定义，是一些数字类型代表的模式，主要是针对 linux 下的权限控制，尽管 windows 下也支持这个方法，但是，只有 stat.S_IREAD, stat.S_IWRITE 才会起到真正的作用.

12.3.3 chown()/lchown()　改变 owner 和 group Id / 功能相同，但不会跟踪链接….或许这个方法 windows 平台不支持??? 接受三个参数，第一个是路径，第二个是用户 id，第三个是组 id

12.3.4 umask()　设置默认权限模式 接受一个数字类型的参数，设置新的默认权限模式，并返回之前的权限模式.

12.4 文件描述符操作

12.4.1 open()　底层的操作系统 open

12.4.2 read()/write()　根据文件描述符读取/写入数据

12.4.3 dup(file_descriptor)/dup2(file_descriptor, file_descriptor2)　dup 返回 file_descriptor 的复制品, dup2 则是把 file_descriptor 复制到 file_description2

12.4.3.1 每个进程在进程表中有一个记录项，每个记录项中有一张打开文件描述符表，可以看作是一个矢量，每个描述符占用一项，与每个文件描述符相关联的是:

12.4.3.1.1 文件描述符标志

12.4.3.1.2 指向一个文件表项的指针

12.4.3.2 内核为所有打开文件维持一张文件表，每个文件表项包含

12.4.3.2.1 文件状态标志(读，写，增写，同步，非阻塞等).

12.4.3.2.2 当前文件位移量

12.4.3.2.3 指向该文件 v 节点表项的指针

12.4.3.3 习惯上，标准输入(STDIN_FILENO)的文件描述符是 0，标准输出(STDIN_FILENO)的文件描述符是 1，标准错误(STDERR_FILENO)的文件描述符是 2

12.4.3.4 文件描述符是非负整数，打开现存文件或新建文件时，内核会返回一个文件描述符，读写文件也需要使用文件描述符来指定待读写的文件

12.4.3.5 文件描述符的有效范围是 0 ---- open_max，一般，每个进程最多打开 65 个文件，freebsd5.2.1, mac os x 10.3 和 solaris9 则支持每个进程打开最多文件数和内存, int 的大小，管理员设定有关, linux2.4.22 强制规定最多不超过 1048576 个.

12.4.3.6 fdopen(file_descriptor[, mode[, fuffer_size]])　通过指定文件描述符，模式，缓冲区大小打开一个文件.　由于缓冲区的存在，最后需要显示的 flush()一下以清空缓冲区

12.4.3.7 dup2(fd, fd2)　表示 fd 和 fd2 共享同一个文件表项，也就是说他们的文件表指针指向了同一个文件表项，因此他们也就是共享了相同的文件 i/o…第一个参数必须是已经存在的并且打开的合法的文件描述符，而第二个参数可以是任意的文件描述符

12.4.3.8 dup(fd)　返回一个和文件描述符 fd 同样的文件描述符，这里是为了备份文件描述符，当一个文件描述符使用 dup2 进行重定向之前，我们应该首先用 dup 拷贝一

份出来备份，需要恢复的时候进行再次重定向. 它的语义是: 返回一个新的文件操作符，和原来的文件操作符 fd 共享一个文件表项

12.5 设备号

12.5.1 makedev() 从 major 和 minor 设备号创建一个原始设备号

13 os.path 模块的路径名访问函数

13.1 分隔

13.1.1 basename() 去掉目录路径，返回文件名

13.1.2 dirname() 去掉文件名(自己的名字，如果是文件夹就是文件夹名)，返回目录路径

13.1.3 join() 将分离的各部分组合成一个路径名…可以接受不定数量个参数.

13.1.4 split() 返回一个元组，第一个元素是目录名，第二个元素是文件名(或最后一层的文件夹名)

13.1.5 splitdrive() 返回一个元组，第一个元素是该路径所属的设备号(比如 C 盘)，第二个元素是相对该设备的相对路径

13.1.6 splitext() 返回一个元组，第一个元素是该路径的全路径(不含文件后缀名，如果末级是文件夹，第二个元素为空)，第二个元素是文件的后缀名

13.2 信息

13.2.1 getatime() 返回最近访问时间 这里的返回时间的方法都是返回 int 型的秒数

13.2.2 getctime() 返回文件创建时间

13.2.3 getmtime() 返回最近文件修改时间

13.2.4 getsize() 返回文件大小

13.3 查询

13.3.1 exists() 判定文件/目录是否存在

13.3.2 isabs() 判定路径是否是绝对路径

13.3.3 isdir() 判定路径是否存在且为一个目录

13.3.4 isfile() 判定路径是否存在且为一个文件

13.3.5 islink() 判定路径是否存在且为一个符号链接

13.3.6 ismounts() 判定路径是否存在且为一个挂载点

13.3.7 samefile(path1, path2) 判定两个路径名是否指向同一个文件

14 永久存储模块

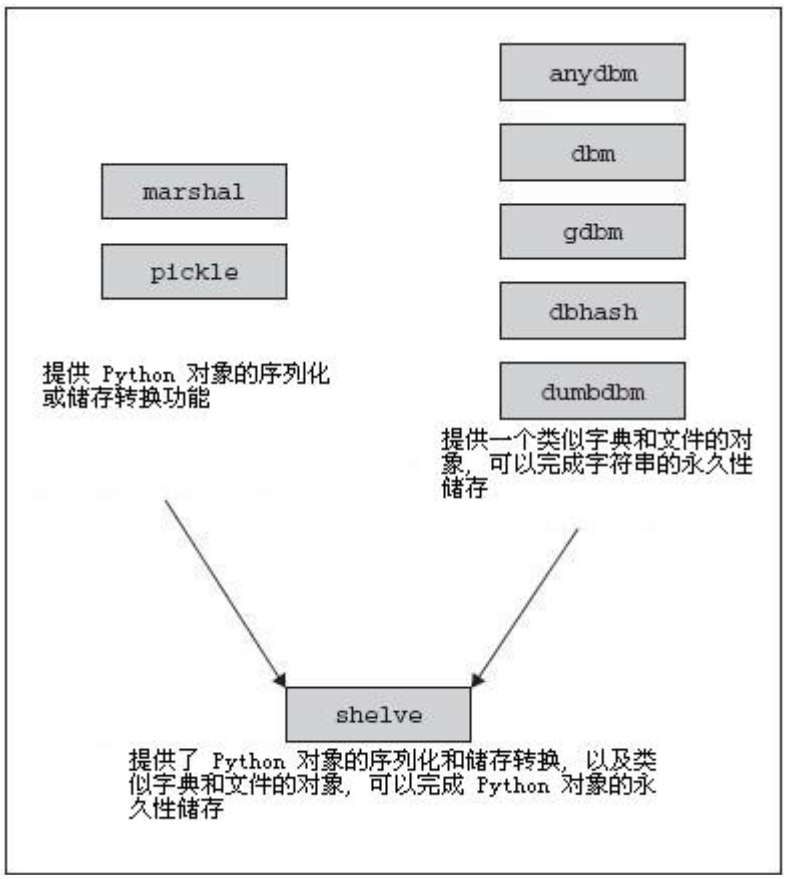
14.1 pickle 和 marshal: 一组最小化永久性存储模块(序列化). 可以用来转换并存储 python 对象. 是将 python 的对象转换成一个二进制数据集合保存起来，可以通过网络发送，然后重新把数据集合恢复成原来的对象格式，也就是 java 中的序列化.

14.2 marshal 和 pickle 模块的区别在于 marshal 只能处理简单的 python 对象(数字，序列，映射，代码对象)，而 pickle 还可以处理递归对象，被不同地方多次引用的对象，以及用户定义的类和实例...

14.3 pickle 模块还有一个增强的版本叫 cPickle

14.4 *db*系列的模块使用传统的 DBM 格式写入数据… python 提供了 dbhash/bsddb, dbm, gdbm, dumbdbm 等多个 DBM 实现. 如果不确定，使用 anydbm 模块，它会自动检测系统上已经安装的 DBM 兼容模块，选择'最好'的一个. dumbdbm 模块是功能最少的，没有其他模块可用时，anydbm 才会选择它….这些模块的不足指出在于他们只能存储字符串，不能对 python 对象进行序列化

14.5 shelve 模块使用 anydbm 模块寻找合适的 DBM 模块，然后使用 cPickle 完成对储存的转换过程. shelve 模块允许对数据库文件进行并发的读访问，但不允许共享读/写访问.




14.6 pickle 的使用

14.6.1 dump(obj, file, protocol = None): 接受一个数据对象和一个文件，把数据对象以特定的格式保存到给定的文件里.

14.6.2 load(file) 从文件中取出已经保存的对象.

15 相关模块:

15.1 base64 二进制字符串和文本字符串之间的编码/解码操作

15.2 binascii 二进制和 ascii 编码的二进制字符串间的编码/解码操作

15.3 bz2 访问 BZ2 格式的压缩文件

15.4 csv 访问 csv 文件(逗号分割文件)

15.5 filecmp 用于比较目录和文件

15.6 fileinput 提供多个文本文件的行迭代器

15.7 getopt/optparse 提供了命令行参数的解析/处理

15.8 glob/fnmatch 提供 Unix 样式的通配符匹配功能

15.9 gzip/zlib 读写 GNU zip(gzip)文件(压缩需要 zlib 模块)

15.10 shutil 提供高级文件访问能力

15.11 c/StringIO 对字符串对象提供类文件接口

15.12 tarfile 读写 TAR 归档文件，支持压缩文件

15.13 tempfile 创建一个临时文件(名)

15.14 uu 格式的编码和解码

15.15 zipfile 用于读取 ZIP 归档文件的工具

16 fileinput 模块遍历一组输入文件，每次读取它们内容的一行，类似 Perl 语言中的不带参数的<>操作.

# 错误和异常(chapter10)

1 常见异常:

1.1 NameError: 尝试访问一个未声明的变量

1.2 ZeroDivisionError: 除数为 0

1.3 StntaxError: python 解释器语法错误, 这个异常是唯一一个编译时错误.

1.4 IndexError: 请求的索引超出序列的范围

1.5 KeyError: 请求一个不存在的字典关键字

1.6 IOError: 输入/出处错误

1.7 AttributeError: 尝试访问一个未知的对象属性错误

1.8 ValueError: 值错误(参数值错误)

1.9 TypeError: 类型错误(参数类型错误)

2 异常的检测, 抓取处理:

2.1 try ---- except ---- finally

2.2 和 java 的异常处理类似, 但是这里不用 throws, 只要不显式的捕获, 就向上抛出

3 同一个 except 子句中可以处理多个异常: except (Exception1[, Exception2, …, Exceptionn])[, reson]: 需要注意的是, except 在处理多个异常的时候, 需要把多个异常放在一个元组中.

4 所有异常的基类是 Exception, 所以, 可以使用 Exception 捕获所有的异常…当然, 也可以使用裸 except 语句捕获多数的异常

5 异常发生时, 可以使用 sys.exc_info()或得到当前的系统执行信息, 得到的结果元组类似下面的格式: (<type 'exceptions.IndexError'>, IndexError('list index out of range',), <traceback object at 0x00C15EE0>) 第一个元素表明异常的类型, 第二个元素是异常, 第三个是堆栈信息

6 有些异常不是由于错误条件引起的, 比如 SystemExit, KeyboardInterupt…SystemExit 是由于当前 Python 应用程序需要退出引起的, KeyboardInterupt 则代表用户按下了 ctrl-c, 想要关闭 python

7 python2.5 之后的异常体系结构

- BaseException
|- KeyboardInterrupt
|- SystemExit
|- Exception
|- (all other current built-in exceptions) 所有当前内建异常

8 异常参数是可以被忽略的, 其中包含的是对导致异常的代码的诊断信息, 异常参数通常会自身组成一个元组, 并存储为类实例(异常类的实例)的属性.

9 带 else 的 try(如果 try 块中没有发生 except 的异常, 执行 else)

```
try:
    code_block
except (Exception …), reason:
    code_block
else:
    code_block
finally:
    code_block
```

10 with 语句用于上下文管理和资源分配

```
with context_expression [as var]:
    with_suite
```

上面的语句是 with 的标准语法，表示通过 context_expression 表达式创建一个上下文环境，并将其交给句柄 var，在 with_suite 中可以使用这个上下文环境，with 会自己去处理上下文环境的准备工作和收尾工作(无论是异常发生还是执行结束)

11 python2.6 开始支持 with 语法，支持的模块有:

11.1 file

11.2 decimal.Context

11.3 thread.LockType

11.4 threading.Lock

11.5 threading.RLock

11.6 threading.Condition

11.7 threading.Semaphore

11.8 threading.BoundedSemaphore

11.9 例如:

```
with open('c:\\1.txt', 'r') as f:
    #file operate
```

这样就会自动管理文件的打开关闭等操作和异常处理

12 上下文表达式和上下文管理器(可以通过自定义自己的类，实现系统方法__enter__()和__exit__()来定义自己的上下文对象/上下文管理器):

12.1 context_expression 用来获得一个上下文管理器…上下文管理器的职责是提供一个上下文对象. 通过__context__()方法实现，该方法返回一个上下文对象，用于在 with 语句块中处理细节. 上下文对象本身就可以是上下文管理器.. 所以 context_expression 既可以是一个真正的上下文管理器，也可以是一个自我管理的上下文对象，在后一种情况时，上下文对象茬苒有__context__()，返回其自身.

12.2 获得一个上下文对象之后，就会调用它的__enter__()方法，它将完成 with 语句块执行前的所有准备工作，with 语法中的 as 后面的句柄，就是通过__enter__()方法的返回值来赋值的. 如果没有 as 子句，就会丢弃返回值

12.3 执行 with 语句块,

12.4 with 语句块执行结束后，哪怕是由于异常结束，都会调用上下文对象的__exit__()方法… __exit__()方法接受三个参数，如果 with 语句块正常结束，三个参数全部都是 None，如果发生异常，得到的三个参数对应 sys.exc_info()得到的元组的三个元素(类型—异常类，值—异常实例，回朔—traceback)

12.5 contextlib 模块中是关于上下文管理的一些实现

13 触发异常使用 raise [SomeException[, args[, traceback]]]，第一项是异常类，第二项是参数(有多个参数使用元组)，最后一项是回朔信息(堆栈)

13.1 通常，错误的参数中指出错误的原因子串，错误编号，错误地址等. 也可以使用 raise [exception_instance]

13.2 如果给定的是一个异常实例，而发生的异常不是该异常类指示的异常，那么会根据异常类实例的参数，重新创建一个实际发生的异常的实例

13.3 raise 的语法:

13.3.1 raise exception_class: 触发一个异常，从 exception_class 生成一个实例，不包含任何异常参数

13.3.2 raise exception_class(): 同上

13.3.3 raise exception_class, args 同上，但是提供了异常参数，参数也可以是一个元组

13.3.4 raise exception_class(args): 同上

13.3.5 raise exception_class, args, traceback 同上，但提供一个追踪对象 traceback 供使用

13.3.6 raise exception_class, instance: 通过实例触发异常(通常是 exception_class 的实例): 如果实例是 exception_class 的子类实例，那么这个新异常的类型会是子类的类型. 如果实例既不是 exception_class 的实例也不是它的子类的实例，那么会复制此实例作为异常参数去生成一个新的 exception_class 的实例

13.3.7 raise instance: 通过实例触发异常，异常类型是实例的类型

13.3.8 raise string: 过时的，触发字符串异常

13.3.9 raise string, args: 同上，但触发伴随着 args

13.3.10 raise string, args, traceback: 同上，但提供了一个追踪对象供使用

13.3.11 raise(python1.5 新增): 重新触发前一个异常，如果之前没有发生异常，触发 TypeError

14 断言: assert expression[, arguments] 判断 expression 表达式，得到断言的真假，如果真不做任何操作，如果是假，触发 AssertionError, arguments 是在触发 AssertionError 时传递给异常的参数.

15 相关模块:

15.1 exceptions: 内建异常，不用导入这个模块

15.2 contextliba: 为使用 with 语句的上下文对象工具

15.3 sys 包含各种异常相关的对象和函数(sys.ex*)

# 函数和函数式编程(chapter11)

1 过程只处理一些操作，而没有返回值，函数则会经过一系列处理返回一些数据. python 的过程就是函数

2 关键字参数: 调用函数时按照函数定义时指定的形式参数名=实参句柄的方式指定实参值，这样的参数就叫关键字参数

3 函数属性不能在函数的声明中访问，因为函数体还没有被创建，但是当函数体创建完毕之后，就可以访问了.

4 python2.1 中引入了内部函数的支持机制，可以在一个函数体内直接定义另一个函数，内部函数的生命域在父亲方法内.

5 函数的装饰器: 和 java 中的注解语法一致，以@开头，接下来是装饰器函数的名字和可选的参数，接下来是其修饰的函数

@decorator(decorator_option_arguments)

def function2bedecoratedname(func_arguments):

5.1 没有装饰器之前，使用绑定的方式将方法注册成为类方法或静态方法:

5.1.1 object.function = staticmethod(function) 或在类内部直接 function = staticmethod(function)

5.1.2 object.function = classmethod(function) 或在类内部直接 function = classmethod(function)

5.2 使用装饰器的语法

5.2.1 静态方法

@staticmethod
def function():

5.2.2 类方法

@classmethod
def function():

5.2.3 带参数的装饰器

@deco1(deco_arg)
@deco2
def func():
就相当于做了
func = deco1(deco_arg)(deco2(func))

5.2.4 装饰器其实就是函数，装饰器接受函数对象，并对函数对象进行包装处理，就相当于 java 中的 aop，可以使用装饰器做一些日志，安全性检查等通用型的功能

```
def mydecorator(func):
    def wrappedFunc():
        print 'log is recorded'
        return func()
    return wrappedFunc

@mydecorator
def f():
    print 'Hello decorator'

print f()
```

5.2.5 装饰器的注意点:

5.2.5.1 必须定义一个内部函数名字是 wrappedFunc

5.2.5.2 装饰器函数返回的是内部函数 wrappedFunc 的引用(或别名)

5.2.5.3 wrappedFunc()函数包装器所接受的参数就是被装饰函数的参数列表

5.3 装饰器可以有多个，每行一个，按照顺序写就可以了，其实装饰器就相当于调用了对应的装饰器函数，对于多个装饰器，就类似数学上的函数嵌套.(g • f)(x) = g(f(x))

5.4 参数

5.4.1 位置参数，普通的参数

5.4.2 默认参数，有默认值的参数

5.4.3 关键字参数，针对调用时，使用参数名=值的方式调用的

6 使用*和**调用函数

6.1 *，将用星号指定的实参转换成一个元组，元组的每一个元素作为一个参数进行传递.例如 a = [1, 2, 3, 4] func(*a)则 func 接收到了 4 个参数.

6.2 **，用两个星号指定的实参必须是字典类型，而且字典的键必须是 str，这样传递的参数实际上是以字典中的数据得到了 N 个关键字参数

7 可变参

7.1 位置可变参使用*argument_name 定义，函数内部得到的将是一个元组.

7.2 关键字可变参使用**argument_name 定义，函数内部得到的将是一个字典.

8 匿名函数与 lambda：使用 lambda 参数列表：表达式的方式可以创建一个匿名函数，lambda 构建的函数可以使用可变参.　优点在于绕过了函数的栈分配，性能得到了提升

（测试了 10^9 数量级次数的空函数调用，性能提升了 1%左右）

9 内建函数

9.1 apply(func[, nkw][, kw])：用可选的参数来调用 func，nkw 是非关键字参数，kw 是关键字参数，返回值就是 func 调用之后的返回结果

9.2 filter(func, sequence)：调用一个布尔函数 func 迭代遍历序列中的每个元素，返回一个 func 返回 True 的元素的序列.

9.3 map(func, seq1[, seq2...])：将函数 func 作用于给定序列中的每个元素，并用一个列表来提供返回值，如果 func 为 None，func 就表现为身份函数(id)，返回一个含有每个元素中元素集合的 n 个元组的列表

9.3.1 提供了多少个 seq 参数，func 就接受多少个参数.

9.3.2 如果 func 存在，将各个列表的值按顺序传入到 func 中，返回的结果被组成一个新的列表作为 map 的返回.

9.3.3 如果 func 是 None，返回各个列表的类似压缩得到的元组的列表，这里的压缩不同于 zip，是类似 sql 中的外联接的模式.

9.4 reduce(func, seq[, init])：将二元函数作用于 seq 序列的元素，每次携带一对(先前的结果以及下一个元素)，连续的将现有的结果和下一个值作用在获得的随后的结果上，最后减少序列为一个单一的返回值，乳沟初始值 init 给定，第一个比较会是 init 和第一个元素而不是序列的前两个元素. 有一点类似递归调用

10 引入一个模块的时候可以起一个别名: from module_name import attribute_name as alias

11 偏函数: 使用 functools 模块中的 partial()函数创建偏函数，第一个参数是基函数的引用，后面跟随基函数的调用中要使用的一些默认值…这里还可以使用关键字参数调用的方式:

```
toDecimal = partial(int, base = 2)
```

调用的时候，只需要指定在 partial 中没有指定的参数，比如: toDecimal('101010')

由于在 partial 中指定的这些固定的参数是在参数列表的最左边，所以，如果不使用关键字参数，可能会导致参数顺序错误.

12 在函数体内使用 global 关键字可以使全局变量的变量名在函数体内获得全局变量的引用，在函数体内对全局变量的改变会影响全局变量，例如:

```
a = 100
def f():
    global a
    a += 300
    print a
```

得到的输出会是 400，并且，全局变量 a 也被修改为 400

注意: 使用 global 语法，必须是在该全局变量名第一次在函数体内被使用之前

13 嵌套函数的内部变量作用域为自己的函数体(包含自己的内部子孙函数). 在函数体内可以使用的变量域为自己的所有父辈中的变量.

14 闭包: 将内部函数自己的代码和作用于以外的外部函数的作用结合起来，在一个独立的作用域中，多用于安装计算，隐藏状态，函数对象和作用于中随意切换，回调.

14.1 自由变量: 定义在外部函数内(非全局变量)，但是由内部函数引用或者使用的变量.

14.2 闭包: 如果在一个函数内，对在外部作用域的变量(自由变量)进行引用，那么这个内部函数就被认为是闭包(closure)

14.3 闭包完成的功能很像类.

14.4 闭包将内部函数自己的代码和作用域以外的外部函数作用结合起来…闭包的词法变量不属于全局名字空间域也不属于局部的，而是属于其他的名字空间，带着"流浪"的作用

域. 但是, 这又不同于对象, 对象的变量存活在对象的名字空间, 而闭包变量存活在一个函数的名字空间和作用域.

14.5 闭包的语法:

14.5.1 定义

```
def out_function(arg1, arg2, arg3……):  //定义一个外部函数
    args = [arg1, arg2]  //将要作为自由变量的参数放到一个可变对象中.
    def inner_function(self_arg):  //定义一个内部函数, 这里可以接受自己的参数
        args[0] += args[1] + self_arg  //改变自由变量, 上面将自由变量放到一个可变对象中就是为了能够保存这种改变.
        return args[0]  //返回修改后的自由变量值
    return inner_function  //返回内部函数
```

14.5.2 调用

```
closure1 = out_function(1, 2, 3)  //得到一个闭包(调用外部函数, 返回了内部函数的引用, 实际上就形成了一个闭包, 这个闭包的作用域内持有了外部函数内定义的 args 自由变量和内部函数本身)
closure2 = out_function(4, 5, 6)  //得到一个闭包
closure1(100)  //调用闭包局部的实参是 100
closure2(200)  //调用闭包返回的
```

14.6 调用闭包返回的结果很类似调用对象的方法, 但是, 闭包是一种能够有自己的独立作用域的方法, 而对象则是类实例.

14.7 得到一个闭包的实例(实际上就是一个方法), 查看它的 func_closure 属性, 可以查看该闭包含有的自由变量.

14.8 一个闭包和装饰器结合的绝妙的例子见 python-15-function 下的 closureAndDecorator4Log.py

14.9 lambda 定义的函数作用域, 生命周期和普通函数是一致的, lambda 也可以不指定参数, 也可以使用全局或外部变量.

14.10 递归: 一个超级短的递归阶乘函数: f = lambda n: n == 1 and 1 or (n == 2 and 2 or n * a(n - 1))

15 生成器: 生成器使用生成器表达式创建, 和列表解析表达式语法相同, 使用圆括号包裹. 也是使用 next 方法, 完成遍历之后抛出 StopIteration 异常.

15.1 定义: 挂起返回出中间值并多次继续的协同程序成为生成器.

15.2 生成器语法:

```
def mygenerator(arr):
    for i in arr:
        yield i
```

15.3 生成器的创建使用了 yield 语法, 在调用 next 方法的时候, 返回下一个 yield 指示的值, 并让生成器在这个位置停留.

15.4 当然, yield 也可以指示做某一个动作之后停止. 甚至还允许 yield 指示的动作返回的是一个 None

15.5 生成器适用于较大的数据场合.

15.6 生成器的创建函数中不能有 return

15.7 生成器还可以接收外部传送回来的数据进行交互. 在外部调用生成器的 send 方法可以向生成器发送数据, send 方法接受一个参数, 这个参数的值就是 yield argument_name 词法的返回值. send 方法返回的是经过处理之后到下一次 yield 的值, 其实是和调用

next()方法的返回值是一样的，不过这次是经过了 send 进来的数据的处理方式.

```
def counter(start):
    count = start
    while True:
        input = yield count
        if input:
            count += input
        else:
            count += 1
```

# 模块(chapter12)

1 影响 python 的环境变量是 PYTHONPATH，这个是指定 python 的脚本路径的环境变量

2 运行时指定 python 的脚本路径通过访问修改 sys.path 来实现

3 sys.modules 可找到当前导入(import)了哪些模块和它们来自哪些地方，sys.modules 是一个字典，模块名是 key，物理地址是 value

4 名称空间：名称(标识符)到对象的映射

4.1 改变一个名字的绑定叫重新绑定

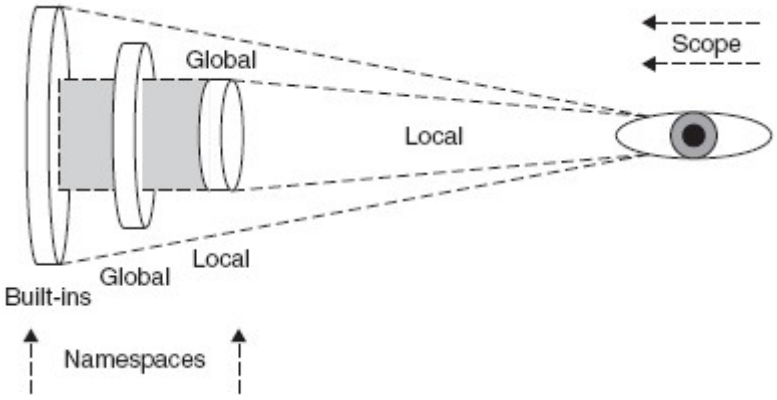
4.2 删除一个名字叫做解除绑定

4.3 执行期间有两个或三个活动的名称空间，全局名称空间和内建名称空间是固有的，局部名称空间是不断变化的.

4.4 从名称空间访问名字依赖于它们的加载顺序

4.5 python 解释器首先加载内建名称空间. 内建名称空间由__builtins__模块中的名字构成. 然后加载全局名称空间，在模块开始执行后变成活动名称空间…… 此时，我们就有了两个活动的名称空间. 如果在运行期间调用了一个函数，就创建出第三个名称空间，也就是局部名称空间. 也可以使用 globals()和 locals()内建函数判断出某一个名字属于那个名称空间

4.6 名称空间和变量作用域:




4.7 无限空间：指定一个名字之后，如果名字指向的是一个名字空间，都可以使用.语法为其添加属性

5 导入模块的顺序： 使用空行分割三种不同类型的导入

5.1 python 标准库

5.2 python 第三方库

5.3 python 应用程序自定义模块

6 from module import name1, name2…

7 from module import name1 as alias1, name2 as alias2

8 执行 import 语句进行导入的时候，被导入的模块的顶层代码将会被执行，并且，这种执行只有一次(该模块在整个 python 程序进程或 pythonVM 中第一次被导入时)

9 将已经安装的模块作为脚本执行，使用[$ foo.py]

10 from-import 导入方式并不是良好的编程风格，这会污染当前名称空间

11 from __future__ import new_feature 是用来导入 python 的新特性的，必须使用 from-import 的方式.

12 警告框架：Warning 直接从 Exception 继承，是所有警告的基类：UserWarning, DeprecationWarning, SyntaxWarning, RuntimeWarning

13 可以从指定 zip 文件名到 sys.path，从 zip 文件中直接导入模块

14 *新的导入钩子:

14.1 查找器：接受一个参数：模块或包的全名，查找器实例负责查找模块

14.2 载入器：如果查找器查找成功将会返回一个载入器对象. 载入器会将模块载入到内存.

14.3 查找器和载入器实例被加入到 sys.path_hooks

14.4 sys.path_importer_cache 用来保存这些实例.

14.5 sys.meta_path 用来保存一列需要在查询 sys.path 之前访问的实例

15 模块内建函数:

15.1 __import__(): python1.5 加入这个函数，是实际上的导入模块函数. __import__()函数的语法是: __import__(module_name[, globals[, locals[, fromlist]]]) 其中 module_name 是要导入的模块的名称，globals 是包含当前全局符号表的名字的字典，locals 是包含局部符号表的名字的字典，fromlist 是一个使用 from-import 语句所导入符号的列表…….globals, locals 和 fromlist 默认是 globals(), locals(), []

15.2 globals()和 locals()，返回调用者的全局和局部名称空间的字典.

15.3 在全局空间下使用 locals()和使用 globals()得到的结果是一样的，因为此时的局部名称空间就是全局名称空间.

15.4 reload(module_name)内建函数可以重新导入一个已经导入的模块(重新加载运行一次该模块) 使用 reload()重新加载的模块必须是之前被全部导入的模块，而不能是 from-import 导入的模块，也就是说在全局名称空间中必须要有这个模块

16 包

16.1 解决的问题

16.1.1 为平坦的名称空间加入了有层次的组织机构

16.1.2 允许程序员把有联系的模块组合到一起

16.1.3 允许分发者使用目录结构而不是一大堆混乱的文件

16.1.4 帮助解决有冲突的模块名称.

16.2 使用了包之后的导入语法和 modle 是一致的.

16.3 包内需要有一个__init__.py 文件，用来指示初始化模块. 使用 from-import 语句导入子包的时候需要用到它，如果没有用到，__init__.py 可以是空文件. 如果包内没有__init__.py，那么会导致一个 ImportWarning 信息…… 但是，除非给解释器传递了-Wd 选项，否则，该警告会被简单的忽略.

16.4 __init__.py 文件中通过指定__all__变量(模块名称字符串组成的列表)指定使用 from-import *方式导入的时候，导入包内的哪些模块

16.5 绝对导入: 现在的所有导入都被认为是绝对的, 也就是说这些名字必须通过 python 路径(sys.path 或环境变量 PYTHONPATH)来访问…. 绝对导入就是指直接使用 import 语句导入, 此时, 就必须指定可以通过 python 路径访问到的脚本

16.6 相对导入: from-import 导入, 相对包和模块而言.例如

16.6.1 from Phone.Mobile.Analog import dial

16.6.2 from .Analog import dial

16.6.3 from ..common_util import setup

16.6.4 from ..Fax import G3.dial

17 sys.modules 是一个列表, 包含了一个由当前载入(完整并且成功导入)到解释器的模块组成的字典. 模块名是键, 位置是值

18 *阻止属性导入: 如果不想让某个模块属性被 from module_name import *导入, 可以给不想导入的属性名称加上一个下划线. 也就是说_开头的模块是包内私有的, 不被外部导入. 但是, 如果使用显式的调用(例如 import module_name._private_attribute)导入, 那么这个隐藏就会失效.

19 加入 PYTHONCASEOK 的环境变量, 就会使 python 导入模块的时候不区分大小写.

20 python2.3 开始, 模块文件开始支持除 7 位 ASCII 之外的其他编码, 但是 ASCII 是默认的, 想要使模块文件支持其他编码, 必须在 python 模块头部加入一个额外的编码指示说明, 这样就可以让导入者使用指定的编码解析模块, 编码对应的 Unicode 字符串. 例如, UTF-8 的编码的文件可以这样指示:

# -*- coding: UTF-8 -*-

这样,

21 导入循环问题: A 模块加载需要导入 B 模块, B 模块加载需要导入 A 模块, 解决方案: 在循环的其中一个节点, 将依赖模块的导入分解到各个需要的方法中(也就是将导入局部化, 放到局部名称空间中.)

22 模块执行: 使用命令行, 或者在运行时调用 execfile(filename[, globals_dict[, locals_dict]])函数 如果不指定 globals_dict 和 locals_dict 默认是当前的 globals()和 locals()

23 相关模块

23.1 imp 提供一些底层的导入者功能

23.2 modulefinder 允许查找 python 脚本所使用的所有模块, 可以使用其中的 ModuleFinder 类或是把它作为一个脚本执行, 提供自己要分析的另一个 python 模块的文件名

23.3 pkgutil 提供了多种把 python 包打包成一个"包"文件进行分发的方法, 类似 site 模块, 使用*.pkg 文件帮助定义包的路径, 类似 site 模块的*.pth 文件

23.4 site 和*.pth 文件配合使用, 指定包加入 python 路径的顺序, 例如: sys.path, PYTHONPATH, 不需要显式的导入它, 因为 python 导入时默认已经使用该模块, 可以使用解释器参数-S 开关在 python 启动时关闭它.

23.5 zipiimport 可以使用该模块导入 zip 归档文件中的模块

23.6 distutils 提供了对建立, 安装, 分发 python 模块和包的支持, 可以帮助建立使用 C/C++完成 python 扩展.

23.7 importAs()函数提供了 from-import-as 的语法支持 python2.6 中已经被废弃???或者不再标准内建库中..

# 面向对象编程(chapter13)

1 新式类必须继承至少一个父类，类定义时的参数 bases 可以是一个(单继承)或多个(多重继承)用于继承的父类

2 object 是所有类的基类，如果定义类的时候没有定义 bases 参数，如果没有指定父类，那么创建的就是一个经典类.

3 类中的方法必须至少有一个参数，这个参数就是用来传递调用的实例的.

4 调用静态方法的时候，不会传入调用者自身，而调用类方法或者 unbound 的方法的时候就会自动的传入调用者自身，因此，类方法和 unbound 方法必须有至少一个参数.

5 __init__方法类似于类构造器，python 在创建实例后，在实例化过程中，调用__init__方法，

6 python 中的类继承结构是单层的，也就是说如果子类覆写了父类的构造方法，那么父类的构造方法将不会被自动调用，那么，其实你也可以手动的调用父类的构造方法，然而，这就跟调用普通的方法没有什么区别了，而不像是 java 中，每个子类实例其实都持有了父类实例的引用，这样就可以方便的关联操作.

7 命名规则，驼峰标记法+下划线方式.

8 OOD 是不要求实现语言的，也可以使用非 OO 语言实现 OOD 设计.

9 抽象：对现实世界问题和实体的本质表现，行为和特征的建模，建立一个相关的子集，可以用来描绘程序结构，从而实现这种模型. 抽象包括数据属性和数据接口.

10 封装/接口：描述了对数据/信息进行隐藏的观念，它对数据属性提供接口和访问函数. 但是，在 python 中，所有的类属性都是公开的.

11 合成：扩充了对类的描述，使得多个不同的类可以合成一个更大的类.

12 派生/继承/继承结构：派生描述了子类的创建，新类保留已存类类型中所有需要的数据和行为，但允许修改或者其他的自定义操作，都不会修改原类的定义.

13 多态：泛化的定义，得到特化的实现. 允许使用泛化的共同接口操作.

14 自省/反射：跟 java 的反射是一个道理???

15 python 中不支持抽象方法(C++中成为纯虚函数)，可以使用在方法中引发 NotImplementedError 异常获得类似的效果.

16 python 中的类属性就相当于静态属性.

17 python 类中的方法必须进行绑定，虽然很多时候，未绑定的方法也能够调用成功，但是，python 不保证未绑定方法一定调用成功. 而且，不论是否进行绑定，在类中定义了的方法都是所在类的固有属性.

18 globals()和 locals()中的名称都是在对象被装载的时候进行加载的，所以，只要在调用完成了装载，那么 locals()和 globals()中的名称就都是可以用的，除非对其进行卸载.

19 类的属性:

19.1 dir()返回对象的属性的名字列表.

19.2 __dict__返回对象的属性的 key-value 字典，但是，__dict__得到的属性不如 dir()得到的多，另外，使用 vars(class_name)内建函数同样可以得到__dict__属性中的内容，但是，这里 vars()函数返回的应该是__dict__的一个拷贝(深浅不知)，而不是源对象. __dict__中基本上只包含我们自定义的属性.

19.3 特殊的类属性

19.3.1 __name__ 类的名字

19.3.2 __doc__ 类的文档字符串….文档属性不能被派生类继承

19.3.3 __bases__ 类的所有基类组成的元组

19.3.4 __dict__ 类中的属性的 key-value 字典

19.3.5 __module__ 类定义所在的模块

19.3.6 __class__ 实例对应的类.(仅在新式类中)

20 经典类本质上是一个类**对象**，注意，这里是对象，而新式类是一种类型的定义，通过调用 repr(class_name)可以明确的发现，经典类返回的是<class module_name.class_name at 0x00B98CC0>，而新式类返回的是<class 'module_name.class_name'>，通过调用 type(class_name) 也可以看到，经典类返回的是<type 'classobj'>，而新式类返回的是<type 'type'>，也就是说，新式类将 class 和 type 进行了统一.

21 如果在一个类定义中两次使用了同一个名称，那么后面定义的名称会覆盖前面定义的名称.

22 经典类的实例用 type 函数得到的是<type 'instance'>而不会指定具体是哪一个类的实例

23 __init__，伪构造器

23.1 不用通过 new 创建实例, python 的类型没有构造器

23.2 __init__()方法只是在解释器创建一个实例**之后**调用的第一个方法.

24 __new__，更接近构造器的伪构造器

24.1 __new__是在 python2.2 时为了使用户可以对内建类型进行派生，而诞生的一种实例化不可变对象的途径.

24.2 __new__()接受的第一个参数就是类本身

24.3 __new__()是一个类方法，传入的参数是在类实例化操作之前生成的,

24.4 __new__()会调用父类的__new__()来创建对象(向上代理)

24.5 __new__()方法必须返回一个合法的实例

24.6 在__new__()方法中可以调用父类的__new__()方法创建对象…….

25 __new__和__init__对比

25.1 __new__接受的第一个参数是**类型**，而__init__接受的第一个参数是**实例**

25.2 __new__方法中，可以调用父类的__new__方法，第一个参数传递自己的__new__方法中接受的第一个参数. 这样，父类的__new__方法就可以构建出一个新的对象.

25.3 __new__方法中创造了实例对象(至少看起来是这样的，实际上是父类的__new__创建，归根结底, python 还是没有把这个工作交给我们)，并将它返回.

25.4 __new__返回的实例对象，就是__init__方法接受的第一个参数.

25.5 __new__和__init__方法虽然第一个参数的类型和意义不一样，但是，除了第一个参数，他们接受的参数都是必须一样的，就是调用构造的时候传递的参数.

26 __del__解构器:

26.1 解构器在一个实例的所有引用都被清除后才被调用的，比如，当引用计数已经减少到 0

26.2 一个对象的生命周期中，解构器只能被调用一次

26.3 解构器使用的注意点:

26.3.1 首先调用父类的解构器__del__()

26.3.2 调用 del x 不表示调用了 x.__del__()

26.3.3 如果有一个循环引用或其他原因，让一个实例的引用一直存在，那么__del__永远不被调用

26.3.4 __del__()未捕获的异常会被忽略.(因为__del__过程中可能有些变量已经被删除了)

26.3.5 如果不知道为什么，就不要实现__del__

27 可以通过结合__init__和__del__的方式，使用静态变量记录一个类具体有多少个实例等相关信息.

28 构造器中设置默认参数的时候，应该设置的是不可变对象，向列表，字典这样的可变对象可以扮演静态数据．然后在每个方法调用中维护它们的内容．

29 __init__方法不能返回 None 之外的任何类型．

30 特殊的实例属性:

30.1 __class__　实例的类型

30.2 __dict__　实例属性的字典集合

31 内建类型中不存在__dict__这个属性

32 无论在哪里使用类属性，都要加类名前缀

33 类属性的值只有在使用类调用的时候，才可以更新，如果使用实例名称访问类属性，视图修改的……给实例赋一个同名变量，就可以隐藏类中的变量．

34 方法就是类内部定义的函数．方法只有在其所属的类拥有实例时，才能被调用，当存在一个实例的时候，方法就被认为绑定到那个实例了．没有实例时的方法就是未绑定的．

35 方法纵览:

35.1 未绑定方法: 直接定义的方法，并且调用的时候使用类调用而不是实例调用

35.2 实例方法: 直接定义的方法，使用实例调用，在创建实例时，被自动绑定到了实例名称空间汇总

35.3 静态方法: 使用@staticmethod 定义或使用 staticmethod()内建函数绑定的方法，使用类和实例调用都不会传递默认参数

35.4 类方法: 使用@classmethod 定义或使用 classmethod()内建函数绑定的方法，不管使用类或实例调用，默认都会传递第一个参数，这个参数的值是当前调用的类本身，或实例所属的类

36 __bases__用来获取类的基类

37 python 中绕过多态可以以未绑定方式直接调用父类的方法，将子类实例作为第一个参数传入

38 内建函数 super(type, object | sub_type) 用来获取一个伪的父类引用．

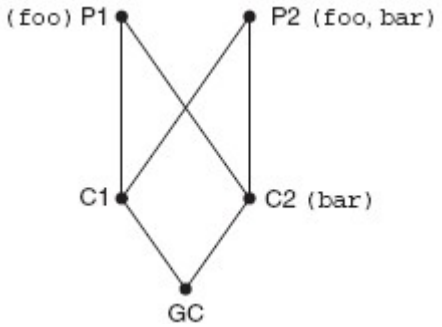
39 如果__init__()方法被子类重写，那么，在创建子类的实例的时候，就不会在自动的调用父类的__init__方法．　如果需要，就要手动的去调用

40 通过继承扩展标准类型(新式类)

40.1 不可变类型: 通过重写__new__方法实现．在__new__中可以 return super(class_name, first_argument).__new__(arguments)来返回一个实例

40.2 可变类型: 需要修改什么行为，就修改什么行为．

41 多重继承中的方法解析顺序



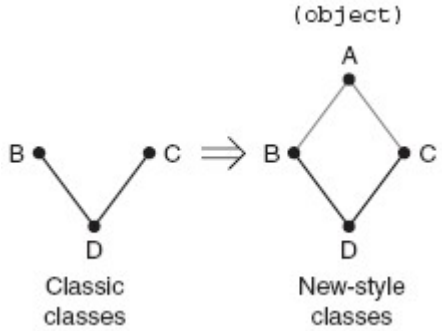

41.1 经典类采用了深度优先，从左到右的算法．　这种算法存在的问题是: 可能由于多重继承导致实际调用的方法不是最'亲'的祖先中定义的方法．　如上图，则GC中的实例调用 bar 方法实际会调用 p2 的 bar，而不是 c2．

41.2 新式类采用了广度优先，从左到右的算法．有效的解决了从继承树中得到的方法不是

最’亲’的祖先的方法的问题.

41.3 新式类中加入了新的属性__mro__，该属性是一个元组，__mro__就标识了新式类中继承树结构中的名称搜索顺序.

41.4 mro 和菱形继承结构：如下图，就发生了菱形继承结构，类 D 的名称查找过程可能会两次回朔到菱形的上顶点.




42 issubclass(sub_class, super_class | tuple_of_superclasses) 判断一个类是另一个类的子类或子孙类. python2.3 开始，第二个参数支持祖先类的元组方式.

43 isinstance(obj1, obj2) 判断类 obj1 是类 obj2 的一个实例或者是 obj2 的一个子孙类的实例，第二个参数应该是一个类.

44 hasattr(object, name), getattr(object, name[, default]), setattr(object, name, value), delattr(object, name)可以在各种对象下工作，用以操作对象的属性.

45 dir

45.1 作用在实例上：显示实例变量和所在类，以及所有基类中的方法和类属性

45.2 作用在类上：显示类以及它的所有基类的__dict__中的内容，但不会显示定义在元类中的类属性

45.3 作用在模块上，显示模块的__dict__内容

45.4 不带参数时，显示调用者的局部变量

46 super()内建函数返回的是 super 类的实例对象，主要被用来查找父类中的属性.

47 super(type, obj=None) 返回一个表示父类类型的代理对象；如果没有传入 obj，则返回的 super 对象是非绑定的，反之，如果 obj 是一个 type，issubclass(obj, type)就必然是 True，否则 isinstance(obj, type)就必然为 True

48 dir(obj=None) 返回 obj 的属性的一个列表，如果没有给定 obj，dir()返回的是当前局部名字空间中的属性，也就是 locals().keys()

49 vars(obj=None) 返回 obj 的属性及其值的一个字典，如果没有给出 obj，vars()显示局部名字空间字典(属性及其值)，也就是 locals()

50 用特殊方法定制类，python 中支持通过一些特殊的方法的重写，实现将类模拟成标准类型或对运算符进行重载，可以使用的特殊方法见 python 核心编程表 13.4

51 在 python 中，可以使用 assert condition, ‘msg’的方式断言.

52 如果要使__str__()和__repr__()做同样的输出，那么不妨实现其中的一个方法，而将另一个方法作为那个实现了的方法的别名，__repr__ = __str__

53 在类中，使用本类的构造器，直接使用 self.__class__()

54 使用__iter__方法和 next 可以重载使得对象支持迭代器，在__iter__方法中，我们只需要将 self 返回就可以了

55 私有化

55.1 双下划线：类中双下划线开始的属性，在运行时会被混淆，不允许直接访问，混淆的机制是采用了 self._classname__attributename 的方式，第一个是但下划线，第二个是双

下划线. 这样做的目的在于保护属性不至于由于和父类名字空间冲突导致父类的同名属性被隐藏.

55.2 但下划线: 在模块内部, 如果一个属性以单下划线开始, 那么, 该属性将不会被其他模块的 from-import *引用到, 但是, 对于 import 则会跳过这个隐藏.

56 包装和授权

56.1 包装是指对一个已经存在的对象进行包装, 不管它是数据类型还是一段代码. 可以是对一个已经存在的对象, 增加新的, 删除不要的, 或者修改其他已经存在的功能.

56.2 授权是包装的一个特性, 用于简化处理有关 dictating 功能, 采用已经存在的功能以达到最大限度的代码重用.

56.3 实现授权的关键在于覆盖__getattr__(self, attr_name)方法, 在代码中, 通过调用内建函数 getattr(object, attr_name)来获取对象的指定属性.

56.4 当引用一个属性的时候, python 解释器将首先在局部名称空间查找那个名字, 然后搜索类名称空间, 以防一个类属性被访问, 最后, 如果两类搜索都失败了, 搜索则对源对象开始授权请求, 此时, __getattr__()方法将会被调用

56.5 使用授权, 仅仅能对属性进行授权, 特殊的行为, 比如 list 的切片操作, 在包装后经过授权也是不能使用的. 原因是切片操作实际上是调用了__getitem__()方法, 且__getitem__()方法没有作为一个类实例方法进行定义.

57 __slots__类属性:

57.1 由于字典__dict__属性跟踪所有实例属性, 因此, 字典会占据大量内存, 当一个类有很多实例的时候, 就会带来内存的压力, 现在可以使用__slots__属性来代替__dict__

57.2 __slots__是一个类变量, 由一个序列型对象组成, 由所有合法标识构成的实例属性的集合来表示, 可以是一个列表, 元组或可迭代对象. 也可以是标识实例能拥有的唯一的属性的简单字符串. 任何在实例中试图创建__slots__中的名字的实例属性都将导致 AttributeError 异常.

57.3 __slots__的目的是节约内存, 但是, 却限制了用户自由的为实例添加动态属性.

57.4 带__slots__属性的类定义不会存在__dict__了, 除非在__slots__中增加__dict__元素

58 __getattribute__()被重写之后, 除非明确从__getattribute__()调用, 或__getattribute__()引发了 AttributeError 异常, 否则, __getattr__不会被调用.

59 __getattr__和__getattribute__两个系统方法比较

59.1 __getattr__方法只有在搜索完类名称空间, 实例名称空间没有发现变量的时候, 才会被调用

59.2 __getattribute__如果定义了, 则只使用__getattribute__方法进行搜索变量

59.3 getattr(object, attribute_name)内建函数实际上就是调用了 object 的__getattr__或__getattribute__方法(如果定义了__getattribute__就不会调用__getattr__), 所以, 在__getattr__或__getattribute__中使用 getattr 内建函数的时候需要注意, 当该方法没有返回合适的变量的时候, 就会带来无限循环的问题

60 描述符: 表示对象属性的一个代理

60.1 描述符就相当于一个类属性(对象)的一个代理, 这个代理实现了__get__(self, obj, type=None), __set__(self, obj, value)和__delete__(self, obj), 三个方法. 当试图访问, 修改或删除该对象指示的属性的时候, 实际上就会调用这三个方法进行处理

60.1.1 __get__(self, obj, type = None), 当使用描述符名称访问的对象的时候, 会调用该方法, self 代表描述符对象自己, obj 就是当前要访问该描述符的类实例

60.1.2 __set__(self, obj, value), 当使用描述符名称设置对象值的时候, 会调用该方法, self 代表描述符对象自己, obj 就是当前要持有描述符的类实例, value 就是要设置的值

60.1.3 __delete__(self, obj) 删除描述符指定的名称的时候，调用该方法，参数意义同上

60.2 代码描述:

```
'''
Created on 2009-9-29

@author: selfimpr
'''
class Window(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

class WindowDescriptor(object):
    def __init__(self, obj):
        self.data = obj
    '''
    @param self: instance of WindowDescriptor
    @param obj: instance of Main
    @param type: unknow
    '''
    def __get__(self, obj, type = None):
        print 'get method called'
        return 'width: %s, height: %s' % (self.data.width,
self.data.height)
    '''
    @param self: instance of WindowDescriptor
    @param obj: instance of Main
    @param value: value is 1024, it is the value after equal mark
    '''
    def __set__(self, obj, value):
        print 'set method called'
        self.data.width = value
        self.data.height = obj.height
    '''
    @param self: instance of WindowDescriptor
    @param obj: instance of Main
    '''
    def __delete__(self, obj):
        print 'delete method called'
        del self.data
        del self

class Main(object):
    def __init__(self, height):
```

```
        self.height = height
    '''
    This is a data descriptor
    '''
    window = WindowDescriptor(Window(width = 800, height = 600))


if __name__ == '__main__':
    m = Main(600)
    m.height = 768
    m.window = 1024 #call method name is __set__ in window data descriptor
    print m.window #call method name is __get__ in window data descriptor
    del m.window #call method name is __delete__ in window data descriptor
```

60.3 以上面代码为例, m.window 实际上就调用了 WindowDescriptor 的__get__()方法, 这其实就类似于 Main.__dict__['window'].__get__(obj, Main)

60.4 优先级别: 在一个类中, 所有的属性有以下集中, 使用属性名称访问的时候, 它们的优先级顺序如下.

60.4.1 类属性

60.4.2 数据描述符

60.4.3 实例属性

60.4.4 非数据描述符(方法)

60.4.5 默认为__getattr__方法

60.5 inspect(检查)模块提供了对于属性类型的检查: isdataDescriptor(obj)检查一个对象是否是一个数据描述符

60.6 任何函数(普通的函数, 静态方法, 类方法, 实例方法等)都是描述符, 函数自己的__get__方法用来处理调用对象, 并将调用对象返回给程序员.

61 属性和 property()内建函数: 对于一些数据属性, 我们可以采用预定义一些方法来提供其操作接口, 将这些预定义的方法设置成隐藏的, 数据本身是不可见的且只读的(使用双下划线定义为隐藏), 然后, 通过一个公共的接口, 使用 property(get_func = None, set_func = None, del_func = None, doc = None)公开其访问权限, 例如下面代码(下面代码中改造的是实例属性, 可以使用同样的方法改造类属性):

```
class a(object):
    def __init__(self, width):
        self.__width = ~width
    def getWidth(self):
        return ~self.__width
    def setWidth(self, width):
        self.__width = ~width
        print 'now, width is: %s' % ~self.__width
    def delWidth(self):
        self.__width = ~-1
        print 'now, width is: %s' % ~self.__width
    width = property(getWidth, setWidth, delWidth)
```

```
if __name__ == '__main__':
    aa = a(1000)
    print aa.width
    aa.width = 30
    del aa.width
```

62 对于上面的属性使用 property 的改进

62.1 ‘借用’一个函数的名字空间

62.2 编写一个用作内部函数的方法作为 property()的参数

62.3 (用 locals())返回一个包含所有的函数/方法名和对应对象的字典

62.4 把字典传入 property()

62.5 去掉临时的名字空间

63 使用装饰器的方式解决上面的问题: 该问题实际上是为了解决由于属性私有化之后需要提供多个操作接口导致的类名称空间庞杂的问题

```
class a(object):
    def __init__(self, width):
        self.__width = ~width
    @property
    def width(self):
        return ~self.__width
    @width.setter
    def width(self, width):
        self.__width = ~width
        print 'now, width is: %s' % ~self.__width
    @width.deleter
    def width(self):
        self.__width = ~-1
        print 'now, width is: %s' % ~self.__width

if __name__ == '__main__':
    aa = a(1000)
    print dir(a)
    print aa.width
    aa.width = 30
    del aa.width
```

64 元类(Metaclasses): 用来定义某些类是如何被创建的, 默认使用的元类是 type, 元类就是类的类

64.1 创建一个类的时候, 解释器需要知道这个类的正确的元类, 解释器首先寻找雷属性__metaclass__, 如果属性不存在, 继续查找继承树中类的__metaclass__属性, 如果还是没有发现该属性, 就会检查名字为__metaclass__的全局变量, 如果都没有查找到, 那么这个类就是一个传统类, 使用 types.ClassType 作为此类的元类.

64.2 如果在一个传统类中设置了__metaclass__=type 那么这个传统类就升级为新式类

64.3 元类的__init__方法接受四个参数, class 类, name 属性名, bases 基类元组, attrd 属性字典

64.4 元类使用示例:

```
class MyMetaClass(type):
    def __init__(cls, name, bases, attrd):
        print cls, name, bases, attrd
        super(MyMetaClass, cls).__init__(name, bases, attrd)
        if '__str__' not in attrd:
            raise TypeError, 'Must rewrite __str__ method'
class Class(object):
    __metaclass__ = MyMetaClass
    pass

if __name__ == '__main__':
    Class()
```

65 相关模块和文档:

65.1 inspect: 提供一些常用的判断, 比如判断一个名称是否是数据描述符等

65.2 types: python 类型库

65.3 operator: 标准操作符的函数版本库

65.4 UserList: 提供一个列表对象的封装类

65.5 UserDict: 提供一个字典对象的封装类

65.6 UserString: 提供一个字符串对象的封装类; 又包括一个 MutableString 子类

# 执行环境(chapter14)

1 Python 中的执行场景:

1.1 在当前脚本继续执行

1.2 创建和管理子进程

1.3 执行外部命令或程序

1.4 执行需要输入的命令

1.5 通过网络来调用命令

1.6 执行命令来创建需要处理的输出

1.7 执行其他的 python 脚本

1.8 执行一系列动态生成的 python 语句

1.9 导入 python 模块(同时执行它的顶层代码)

2 可调用对象: 任何能通过函数调用操作符"()"进行调用的对象.

2.1 apply(object[, args[, kargs]]): object 必须是一个可调用对象(实现了__call__()方法), 该方法是用来动态调用 object 这个可调用对象, 使用方法中传入的其他参数. 返回的就是 object 被调用之后的返回值.

2.2 filter(function or None, sequence): 对 sequence 的每个元素调用 function, 如果返回 True, 则将元素过滤掉, 返回的是过滤之后的 list

2.3 map(function, sequence[, sequence, ...])将所有被传入的所有序列遍历, 应用指定的 function 返回值作为新的 list 的元素, 生成的新的 list 是返回值.

2.4 reduce(function, sequence[, initial]): 将序列进行遍历, function 接受两个参数, 分别为遍历的当前元素和下一个元素, 进行处理, 返回的值作为新的 list 的元素, 如果指定 initial, 在遍历元素之前, 会有一个初始值. 生成的新的 list 会作为返回值返回.

3 python 有四种可调用对象: 函数, 方法, 类, 一些类的实例.

4 函数: python 有三种不同类型的函数对象: 内建函数, 用户自定义函数, 方法(方法又分为内建方法和用户自定义方法两种)

4.1 内建函数: 用 c/c++实现, 编译后放入 python 解释器, 作为内建名称空间的一部分加载进系统., 在_builtin_模块中, 在 python 中, 默认将该模块作为__builtins__模块导入.

4.1.1 内建函数属性:

4.1.1.1 __doc__ 文档字符串

4.1.1.2 __name__ 字符串类型的文档名字

4.1.1.3 __self__ 设置为 None, 保留给 built-in 方法(对比内建方法的__self__属性, 就可以看出不同之处)

4.1.1.4 __module__ 存放 bif 定义的模块名字

4.1.2 由于内建函数和内建方法的内部实现机制都是相同的, 所以, 对内建函数和内建方法调用 type 得到的结果相同, 都是: <type 'builtin_fuction_or_method'>, 内建的工厂函数调用 type 则会返回<type 'type'>

4.2 用户定义的函数: 通常是 python 写的, 定义在模块的最高级, 作为全局名字空间的一部分.

4.2.1 用户定义的函数的属性:

4.2.1.1 __doc__ 文档字符串, 也可以使用 func_doc 获取

4.2.1.2 __name__ 字符串类型的函数名称, 也可以使用 func_name 获取

4.2.1.3 func_code 字节编译的代码对象

4.2.1.4 func_defaults 默认的参数元组

4.2.1.5 func_dict 函数属性的名字空间

4.2.1.6 func_globals 全局名字空间, 和从函数中调用 globals 一样

4.2.1.7 func_closure 包含了自由变量的引用的单元对象元组.(闭包)

4.2.2 从实现上来看, 用户定义函数是'函数'类型的, 因此, type 调用的时候, 返回的是<type 'function'>

4.2.3 lambda 表达式创建的匿名函数的函数名是 lambda, 通过调用 func_name 或__name__可以获取.

4.3 内建方法: 内建类型中定义的方法.

4.3.1 内建方法的属性:

4.3.1.1 __doc__ 文档字串

4.3.1.2 __name__ 字符串类型的函数名字

4.3.1.3 __self__ 绑定的对象

4.3.2 内建方法和内建函数的不同在于他们的__self__属性指向不同, 内建函数的__self__指向 None, 而内建方法的__self__指向的是调用它的对象.

4.4 用户定义的方法: 包含在类定义中, 仅有定义它们的类的实例可以调用, 如果在子类中该方法没有被覆盖, 那么也可以被子类实例调用, 用户定义方法是和类对象关联的(非绑定方法), 但是只能通过类的实例进行调用(绑定方法).....无论用户定义方法是否绑定, 都是相同的类型"实例方法", 使用 type 得到的结果是<type 'instancemethod'>

4.4.1 用户定义方法的属性:

4.4.1.1 __doc__ 文档字符串

4.4.1.2 __name__ 字符串类型的方法名称, 与 im_func.__name__相同

4.4.1.3 __module__ 定义该方法的模块的名称

4.4.1.4 im_class 对于绑定的方法来说, 是方法相关联的类, 对于非绑定的方法, 是要

求用户定义方法的类

4.4.1.5 im_func 方法的函数对象

4.4.1.6 im_self 如果是绑定方法，就是相关联的实例，如果是非绑定方法，就是 None

5 可调用对象，实现__call__方法的类的实例可以作为可调用对象.

6 代码对象，代码对象返回的是该对象被编译成字节的代码，调用 eval(f.func_code)就相当于调用了 f()

7 可执行的对象声明和内建函数:

7.1 callable()，是一个布尔函数，确定一个对象是否可以通过函数操作符'()'进行调用

7.2 compile(source, filename, mode[, flags[, dont_inherit]])，允许程序员在运行时迅速生成代码对象，然后就可以使用 exec 语句或内建函数 eval()执行这些对象或对它们进行求值.

7.2.1 source: 要动态编译的代码字符串

7.2.2 filename: 存放代码对象的文件的名字

7.2.3 mode: 值为 eval(可求值的表达式，和 eval()一起使用) | single(单一可执行语句，和 exec 一起使用) | exec(可执行语句组，和 exec 一起使用)

7.3 eval(source[, globals[, locals]]): 对表达式求值，source 是字符串或预编译代码对象. globals 必须是字典，是 source 的全局名称空间. locals 可以是任意的映射对象(实现了__getitem__()方法的对象)，globals 和 locals 默认是 globals()和 locals()返回的值. 如果只传入了 globals 参数，那么该字典也会作为 locals 传入.

7.4 exec obj: 可以执行代码对象或者字符串形式的 python 代码，同样，使用 compile()预编译重复代码有助于改善性能. exec 也可以接受有效的 python 文件对象. 在执行文件对象的时候，只能执行一次有效，是因为文件读取完毕，指针就会指向文件末尾，可以使用 file.seek(0)方法把文件的指针重新指向文件头.

7.5 input(string): input 实际上就是 eval(raw_input(string))，获取一个字符串对该字符串执行动态求值.

8 execfile(filename[, globals = globals()[, locals = locals()]]) 使用传入的 globals 和 locals 名称空间执行一个 python 脚本文件. 如果不指定名称空间，使用当前的名称空间.

9 python –c -m 开关

10 执行其他程序(os 模块):

10.1 system(cmd): 执行命令行程序，接受字符串的参数. 等待程序结束，返回退出代码(windows 下始终返回 0)

10.1.1 约定使用返回状态 0 表示执行成功，非 0 表示其他类型的错误.

10.2 fork() 创建一个和父进程并行的子进程(通常来说和 exec*()一起使用)，返回两次……一次给父进程，一次给子进程…… 子进程得到的返回是 0，父进程得到的返回是父进程的 pid… 可以通过判定 fork()的返回值的方式，让两个进程执行不同的动作…两个进程是交互获得 CPU 时间片的. 可以通过下面的程序看出两个进程的交互执行的过程:

```
def f():
    r = os.fork()
    if r == 0:
        for i in range(100):
            print 'child:%d' % i
        print 'child process end'
    else:
        for i in range(100):
            print 'parent:%d' % i
```

print ‘parent process end’

10.3 execl(file, *args) 以 args 指定的参数模式运行可执行文件. 替换当前进程….

10.4 execv(file, args) 通过元素是字符串的列表或元组执行可执行文件, 并替换当前进程. 这里使用了参数向量列表

10.5 execle(file, arg0, arg1, …, env) 和 execl 相同, 但提供了环境变量字典 env

10.6 execve(file, arglist, env) arglist 是元素为字符串的列表或元组表示的向量参数列表., env 是要使用的环境变量.

10.7 execlp(cmd, arg0, arg1, …)和 execl()相同, 但是在用户的搜索路径下搜索完全的文件路径

10.8 execvp(cmd, arglist) 除了带有参数向量列表 和 execlp()相同.

10.9 execlpe(cmd, arg0, arg1, …, env) 执行外部程序, 和 execlpe 相同, 但是提供了环境变量字典 env

10.10 execvpe(cmd, arglist, env) 执行外部程序, 和 execvpe 相同, 但是提供了环境变量 env

10.11 spawn*(mode, file, args[, env]) 在一个新的进程中执行路径, args 作为参数, 环境变量 env 可有可无, 模式用于显示不同的操作模式. 这个系列的函数和 fork 结合 exec 系列函数功能相似, 因为 spawn 系列的函数在新的进程中执行命令. 但是不需要调用两个函数来创建进程, 并让这个进程执行命令; 另外有一点不同的是, 使用 spawn 系列的函数, 必须传入 spawn 的魔法模式参数 mode……在一些操作系统(比如嵌入式实时操作系统 RTOS), spawn 比 fork 要快很多…..其他情况的操作系统通常使用写实拷贝.

10.12 wait() 等待子进程完成(通常和 fork 和 exec*()联合使用) 子进程中的 exit()被调用意味着子进程的完成

10.13 waitpid(pid, options) 等待指定的子进程完成

10.14 popen(cmd, mode = ’r’, buffering = -1) 执行字符串 cmd, 返回一个类文件对象作为运行程序通信句柄, 默认为读取模式和默认系统缓冲. 实际上就是文件对象和 system()函数的结合.

10.15 startfile(path) 用关联的程序执行路径.

11 subprocess 模块: python2.3 开始, popen5 模块开始开发, 开始命名使用了 popen*()系列函数的传统, 最终被命名为 subprocess, 其中一个类叫 Popen, 在该类中实现了大部分的面向进程的函数.

11.1 call()函数可以取代 os.system():

```
import subprocess
res = subprocess.call((‘ls’, ‘/’))  #调用 call 执行 ls 命令查看/目录下文件
res  //res 是调用 call 的返回值.
```

11.2 Popen 实例取代 os.popen():

```
import subprocess
f = subprocess.Popen((‘ls’, ‘/’), stdout = subprocess.PIPE).stdout
data = f.readline()
f.close()
print data
```

12 相关函数:

12.1 os/popen2.popen2() 执行文件, 打开文件, 从新创建的运行程序读取 stdout, 或者向该程序写(stdin)

12.2 os/popen2.popen3() 和 popen2 功能相同, 但是可以从运行程序的 stder 中读取数据了

12.3 os/popen2.popen4()　和 popen3 功能相同，这个版本中 stdout 和 stderr 可以结合起来工作

12.4 commands.getoutput()　在子进程中执行文件，以字符串返回所有的输出.

12.5 subprocess.call()　创建 subprocess 的便捷函数.　Popen 等待命令完成，然后返回状态代码;　与 os.system 类型，但更灵活.

13 结束执行:　结束执行通常是程序执行完毕，另外一种情况是使用异常处理来结束程序的运行.　还有一种方法就是建造一个清扫器，将主要代码放在 if 语句中，在没有错误的情况下执行，因而可以让错误的情况正常的终结…　在有些时候，也需要在退出调用程序的时候，返回错误代码. 用来表明发生了什么事.

13.1 sys.exit()和 SystemExit

13.1.1 sys.exit([status]): 接受一个可有可无的参数 status，status 用来描述退出状态，也就是退出之后的返回消息.　status 默认为'0'

13.1.2 当调用 sys.exit()时，就会引发 SystemExit()异常，除非对异常进行监控(只有专门捕获该异常才会生效)，否则异常不会被捕捉或处理，解释器会用给定的状态参数退出. 如果没有给出状态参数，默认为 0…

13.1.3 退出之后，状态码是返回给该程序的调用者的.

13.2 sys.exitfunc(): 该函数默认是不可用的.　可以通过改写提供额外的功能.

13.2.1 当调用了 sys.exit()并在解释器退出之前，就会调用这个函数.　该函数不带任何参数，所以，创建的函数也要是无参的.

13.2.2 如果 sys.exitfunc 已经被先前定义的 exit 函数覆盖了，最好的方法是把这段代码作为自己的 exit()函数的一部分来执行.　一般而言，exit 函数用于执行某些类型的关闭活动，比如关闭文件和网络连接，最好用于完成维护任务，比如释放先前保留的系统资源.

13.2.3 设置一个 exit()函数:

```
import sys
prev_exit_func = getattr(sys, 'exitfunc', None)　#获取原来的 exitfunc，如果没有得到 None

def my_exit_func(old_exit = prev_exit_func):
    ############
    # execute clean work　执行清理工作
    ############
    if old_exit is not None and callable(old_exit):
        old_exit()　# 如果存在原来的 exitfunc，就调用它.

sys.exitfunc = my_exit_func　把新的 exit 函数注册到 sys.exitfunc
```

13.3 os._exit(): 之前的 sys.exit()和 sys.exitfunc()都会执行一些清理工作，而 os._exit()不会执行任何清理工作就直接退出.　并且，该函数是和平台相关的，适用于 Unix 类系统和 Win32 平台.　语法是 os._exit(status)

13.4 os.kill(pid, sign)　接受进程 id，模拟传统的 unix 函数来发送信号给进程，发送的典型的信号是: SIGINT, SIGQUIT, SIGKILL，使程序结束.

14 各种操作系统的接口见书 618 页列表.

14.1 用#!/usr/bin/env python 表示用系统默认的 python 运行

14.2 只有在修改文件的权限为可执行的时候才有区别

14.3 如果用 python file.py 系统就会忽略这一句

15 支持管道操作的 shell:

```
'''
Created on 2009-10-21

@author: selfimpr
@blog: http://blog.csdn.net/lgg201
@E-mail: lgg860911@yahoo.com.cn
'''

from sys import stdout
from subprocess import Popen, PIPE

def pipecmd(cmdstr):
    if isinstance(cmdstr, str): # estimate if the argument is string
        cmds = cmdstr.split('|') # split intact cmdstr to sigle command
        cmds = [cmd.strip() for cmd in cmds] # strip space character
        length = len(cmds)
        popens = []
        for index, cmd in enumerate(cmds): # each all the commands
            cmd_args = cmd.split(' ')
            cmd_args = [arg.strip() for arg in cmd_args]
            try:
                ##################
                # get all the instance of Popen
                ##################

popens.append(eval('Popen(cmd_args%(stdin)s%(stdout)s)' % \
                           {'stdin': '' if index == 0 else ', stdin=PIPE',
\
                            'stdout': ', stdout=stdout' if index == length
- 1 else ', stdout=PIPE'}))
            except OSError, e:
                print 'arises os error'
        ##################
        # process pipe
        ##################
        prev = None
        for index, popenobj in enumerate(popens):
            if not prev:
                prev = popenobj
                continue
            popenobj.stdin.write(prev.stdout.read())
            prev = popenobj
```

# 正则表达式(chapter15)

1 查找和匹配:
  1.1 搜索是在字符串任意部分中查找匹配的模式.
  1.2 匹配是判断一个字符串是否能从开始处全部或部分的匹配某个模式.
  1.3 查找和匹配通过 search, match 方法实现.
2 重要的概念:
  2.1 字符集合: 可以匹配的字符
  2.2 子组匹配: 提供用一个正则表达式提取多组值
  2.3 模式匹配重复次数: 一个字符或子组的重复出现次数
3 \A 匹配开始, \Z 匹配结束, 是因为有些键盘不支持^和$符号.
4 中括号中只有或的关系
5 中括号第一个字符指定为^表示不匹配中括号内任何字符.
6 问号跟在表示次数的符号后面表示使用非贪婪模式匹配.
7 ?:表明一组是非捕获组
8 核心函数和方法:
  8.1 compile(pattern, flags=0): 对正则表达式模式进行编译. 其中 flags 是一个标记, 对于同一个 pattern, flags 用来指示是否区分大小写, 是否多行匹配等问题.
  8.2 match(pattern, string, flags = 0) 尝试用正则表达式 pattern 匹配字符串 string…如果匹配成功, 返回一个匹配对象, 否则返回 None
  8.3 search(pattern, string, flags = 0) 在字符串 string 中查找正则表达式模式 pattern 的第一次出现, 匹配成功返回匹配对象, 否则返回 None
  8.4 findall(pattern, string[, flags]) 在字符串 string 中查找正则表达式模式 pattern 的所有非重复出现, 返回一个匹配对象的列表. 如果正则表达式没有分组, 直接返回查找得到的列表, 如果正则表达式有分组, 则返回所有的子组的匹配结果. 如果正则表达式有子组, 并且得到了多次匹配, 那么返回的结果是一个元组的列表, 每个元组分别表示一次匹配成功后得到的所有子组的匹配结果.
  8.5 finditer(pattern, string[, flags]) 和 findall 相同, 但是返回的是一个迭代器.
  8.6 split(pattern, string, max = 0) 根据正则表达式 pattern 中的分隔符把字符串 string 分隔成为一个列表. 返回成功匹配的列表, 最多分隔 max 次(默认分隔所有匹配的地方)
  8.7 sub(pattern, repl, string, max = 0) 把字符串 string 中的所有匹配正则表达式 pattern 的地方替换成字符串 repl, 如果 max 值没有给出, 则对所有匹配地方进行替换.
  8.8 subn, 和 sub 相同, 但是会返回替换次数. subn 返回一个元组, 元组第一个元素是替换后的字符串, 元组第二个元素是替换的次数.
  8.9 group(num = 0) 返回全部匹配对象, 或指定子组编号
  8.10 groups() 返回一个包含全部匹配的子组的元组. 如果没有成功匹配, 返回一个空的元组.
9 模块函数在对正则表达式进行预编译之后会进行缓存, 但是还是建议使用自己的预编译正则表达式, 因为即便解释器缓存了正则表达式, 那也需要查找. python1.5.2 中, 缓存区可以容纳 20 个已编译的正则表达式对象, 在 1.6 版本中, 由于添加了对 Unicode 的支持, 编译引擎速度变慢, 所以, 缓存区被扩展到可以容纳 100 个已编译的 regex 对象.
10 (?...) 子组内使用?开始表示该组有特殊含义
11 (?:…)非捕获组

12 (?P<name>…)有名称组

13 (?P=name)　匹配之前使用(?P<name>…)指定了名称的一个组相同的内容

14 (?#...)　正则表达式的注释文本

15 (?iLmsux): i　忽略大小写，L　依赖本地字符，m　多行匹配，　s　点号匹配所有，u Unicode 字符串，　x　贪婪匹配

16 (?=…)　向前断言是…　　表示匹配该元组内的元素，但是，不消耗字符串．　例如 re.match('abc(?=e).*', 'abcdef')　是匹配不成功的，因为匹配到 c 结束之后，使用子组匹配 e 匹配不到，所以匹配失败，修改为 re.match('abc(?=d).*', 'abcdef')匹配得到的结果则是 abcdef，因为匹配到 c 结束之后，用子组匹配 d，匹配成功，但是，字符串没有被消耗，所以，后面的.*继续从字符串的 d 开始匹配.

17 (?!...)　向前断言非…　　和(?=…)相反，不消耗字符串的前提下，匹配字符串要和子组内指定的模式不匹配才能够匹配成功．　例如 re.match('abc(?!e).*', 'abcdef')就能够匹配成功，因为 c 匹配结束之后，使用子组匹配 e，匹配失败了，所以可以继续向下.

18 (?<=…)　向后断言是…　和向前断言是相同

19 (?<!...)　向后断言非…　和向前断言非相同

20 (?(id/name)yes-pattern | no-pattern)　如果指定 id 或 name 的组匹配成功(存在匹配得到的字符串，而不能是?, + {0, n}等次数匹配得到的 0 次)，那么使用 yes-pattern 指定的规则匹配，如果指定的组匹配得到的是 0 次，那么使用 no-pattern 指定的规则匹配．　例如: (?**<starttag>**<)?html(?**(starttag)**>)，如果 starttag 组匹配<得到了 1 次(或以上)，那么(?(starttag)>)这个组使用>进行匹配，否则，使用|后的 no-pattern 匹配，由于这里没有指定管道|，所以，这一组就不匹配．　因此，上面的这个正则表达式可以匹配<html>也可以匹配 html

21 regular1|regular2　使用 regular1 匹配或使用 regular2 匹配

22 正则表达式选项: 使用(?iLmsux)指定，并且，这个开关的开启可以在正则表达式的任意位置，但在正则表达式全局有效．　i L m s u x 是区分大小写的

22.1 i　忽略大小写(不倾向于当前的本地环境)

22.2 L　使\w, \W, \b, \B, \s, \S 等特殊字符依赖本地环境表意

22.3 m　多行匹配，如果指定了^或$，则表示每一行的开始或结尾都要符合规则，并且，指定之后获取 match 对象，使用 group()接口只能获取到第一行的匹配结果，要想获得所有行的匹配结果需要使用 re.findall 接口，返回的列表中，每一行的匹配结果作为列表的一个元素，如果每一行有多个子组匹配，那么每一行的匹配返回的是一个元组，该行内每个子组的匹配结果是对应元组中的元素.

22.4 s　使.匹配包含换行符在内的所有字符．　不打开 s 开关的时候, .不能匹配换行符

22.5 u　使\w, \W, \b, \B, \d, \D, \s, \S 依赖于 unicode 字符串进行匹配

22.6 x　使用冗长的方式描述正则表达式. 这个状态下，普通的空格会被忽略，可以直接用#给正则表达式加注释，这样使用，可以使正则表达式更加易读．　如果要在这样的正则中匹配空格，就使用\转义的空格.

23 python 正则表达式在使用 flags 指定选项开关时，也使用了位操作进行授权

24 在正则表达式中使用原始字符串 r'', 之所以使用原始字符串，是因为正则表达式的一些特殊字符比如\b 和 ASCII 字符冲突, \b 在正则表达式中表示单词边界，但是 ASCII 字符中表示退格键．　使用\\b 可以解决这种冲突，让\\b 代表单词边界，但是，更好的方法是在创建正则表达式的时候使用原始字符串 r''，这样就不会存在这个问题了，而且不用为了一些冲突的字符用多一个\的方式搞乱正则表达式.

25 使用断言的时候，用 match 接口不能返回匹配结果…

# 网络编程(chapter16)

1 什么是客户/服务器架构:

1.1 服务器是一个软件或硬件, 用于提供客户需要的'服务', 服务器存在的唯一目的就是等待客户的请求, 给这些客户服务, 然后等待其他请求.

1.2 客户连上一个服务器, 提出自己的请求, 发送必要的数据, 然后就等待服务器的完成请求或说明失败原因的反馈. 服务器不停的处理外来的请求, 而客户一次只能提出一个服务的请求, 等待结果. 然后结束这个事务. 一个事务结束后, 客户再提出其他的请求会被看作是另一个事务.

2 NFS: Network File System, 是 Sun 公司的网络文件系统.

3 软件的 C/S 结构主要是程序的运行, 数据的发送与接收, 合并, 升级或其他的程序或数据的操作.

4 远程桌面连接实际上是 GUI 的远程数据发送.

5 架构: 服务端建立一个通讯端点, 监听服务端口(服务启动时可以向潜在客户端发送广播), 客户端创建一个通讯端点, 有目的的连接到服务器. 这样, 客户端就能够提出请求, 请求中, 可以包含必要的数据交互, 一旦请求处理完成, 客户受到了数据, 通讯就结束了.

6 套接字: 具有'通讯端点'概念的计算机网络数据结构. 网络化的应用程序在开始任何通讯之前都必须要创建套接字.

6.1 起源于 20 世纪 70 年代加利福尼亚大学伯克利分销版本的 Unix, 也就是 BSDUnix. 所以, 套接字也被称为伯克利套接字或 BSD 套接字.

6.2 最早套接字用于同一台 pc 上的多个应用程序之间的通讯, 也就是进程间通讯, 或 IPC.

6.3 套接字家族:

6.3.1 AF_UNIX: POSIX1.g 标准中也叫 AF_LOCAL, 表示'地址家族: UNIX'. 老一点的系统中, 地址家族被称为'域'或'协议家族', 使用缩写'PF'. 为了向后兼容, 很多系统中 AF_LOCAL_和 AF_UNIX 是等价的. AF_LOCAL 在 2000-2001 年被列为标准, 将会替代 AF_UNIX. 这个家族的套接字数基于文件系统的, 用于同一台 pc 上的不同进程进行通讯

6.3.2 AF_INET: 地址家族: Internet, 另外还有一种地址家族 AF_INET6 用于 IPv6 寻址上.

6.3.3 python2.5 中加入了一种 Linux 套接字的支持: AF_NETLINK(无连接)让用户代码与内核代码之间的 IPC 可以使用标准 BSD 套接字接口.

6.4 套接字有两种: 文件型和网络型.

6.5 套接字地址: 主机和端口就相当于电话的区号和电话号码.

6.6 合法的端口号是 0-65535, windows 中小于 1024 的端口号系统保留, Unix 操作系统, 通过/etc/services 文件获得系统保留端口号.

6.7 面向连接的套接字:

6.7.1 通讯之前, 一定要建立一套连接. 也被称为'虚电路'或'流套接字'

6.7.2 提供了顺序的, 可靠的, 不会重复的数据传输, 而且也不会被加上数据边界. 也就是说每一个要发送的信息, 可能会被拆分成多份. 每一份都会不多不少的正确的到达目的地, 然后被重新按顺序拼装起来, 传递给正在等待的应用程序.

6.7.3 TCP: 传输控制协议(Transfer Controll Protocol), 创建 TCP 套接字就得在创建的时候, 指定套接字类型为 SOCK_STREAM

6.7.4 TCP 套接字采用 SOCK_STREAM 作为名字, 表达了它作为流套接字的特点, 由于这些套接字使用 Internet 协议来查找网络中的主机, 这样形成了整个系统, 一般会由这连个

协议提及，即 TCP/IP

6.8 面向无连接的套接字:

6.8.1 数据报型的无连接套接字.

6.8.2 无需建立连接就能通讯，数据到达的顺序，可靠性和数据不重复性无法保证，数据报会保留数据边界，也就是说数据不会像面向连接的协议那样被拆分成小块.

6.8.3 UDP：用户数据报协议(User Datagram Protocol)．创建 UDP 套接字需要在创建的时候指定套接字类型为 SOCK_DGRAM.

6.8.4 UDP 也使用 Internet 协议查找网络中的主机，形成整个系统，因此，称为 UDP/IP.

7 Python 中的网络编程: socket 模块．其中的 socket()函数用来创建套接字.

7.1 socket()函数: socket(socket_family, socket_type, protocol=0): socket_family 是套接字家族(AF_UNIX 或 AF_INET), socket_type 可以是 SOCK_STREAM 或 SOCKET_DGRAM, protocol 一般不填，默认是 0.

7.2 创建 TCP/IP 套接字: tcpsocket = socket.socket(socket.AF_INET, socket.SOCKET_STREAM)

7.3 创建 UDP/IP 套接字: udpsocket = socket.socket(socket.AF_INET, socket.SOCKET_DGRAM)

8 套接字对象的内建方法:

8.1 服务端套接字函数:

8.1.1 s.bind(address)绑定地址(主机，端口号等)到套接字. address 的格式依赖于当前的 socket 的家族, python-library　page600 给出了家族 address 的相关信息，其中，使用 INET 就爱组的套接字时，对于 HOST 设置为''时，表示可以将套接字绑定到任何有效的地址上.

8.1.2 s.listen(backlog)　开始 TCP 监听. backlog 指定连接队列的最大数量，最小是 1，最大依赖于系统(一般是 5)

8.1.3 s.accept()　被动接受 TCP 客户的连接，阻塞式的等待连接的到来.

8.2 客户端套接字函数

8.2.1 s.connect()　主动初始化 TCP 服务连接

8.2.2 s.connect_ex()　connect()函数的扩展版本，出错是返回出错码，而不是抛出异常.

8.3 公共用途的套接字函数

8.3.1 s.recv(buffersize[, flag])　接受 TCP 数据, buffersize 指定缓冲区大小, flag 不知道意思，返回接受到的数据

8.3.2 s.send(data[, flag])　发送 TCP 数据, data 为 string 类型的数据, flag 不知道意思，返回发送数据的字节数

8.3.3 s.sendall()　完整发送 TCP 数据

8.3.4 s.recvfrom(buffersize[, flags])　接受 UDP 数据, buffersize 指定缓冲区大小，但是，接受到超过 buffersize 的数据的时候，会发生异常.

8.3.5 s.sendto(string[, flags], address)　发送 UDP 数据, string 是要发送的数据, flags 默认是 0, address 是要发送的地址，依赖于套接字的家族.

8.3.6 s.getpeername()　连接到当前套接字的远端的地址

8.3.7 s.getsocketname()　当前套接字的地址

8.3.8 s.getsocketopt()　返回当前套接字的参数

8.3.9 s.setsocketopt()　设置当前套接字的参数

8.3.10 s.close()　关闭套接字.

8.4 面向块的套接字方法:

8.4.1 s.setblocking()　设置套接字的阻塞与非阻塞模式

8.4.2 s.settimeout()　设置阻塞套接字操作的超时时间

8.4.3 s.gettimeout()　获取阻塞套接字操作的超时时间
8.5 面向文件的套接字函数
8.5.1 s.fileno()　套接字的文件描述符
8.5.2 s.makefile()　创建一个与套接字关联的文件.
9 创建 TCP 服务器:
9.1 创建服务器套接字 socket()
9.2 把地址绑定到套接字上 bind()
9.3 监听连接 listen()
9.4 无限循环开启接受请求
9.5 接受客户的连接 accpet()
9.6 开启通讯循环
9.7 处理客户的请求 recv()/send()
9.8 客户关闭 close()
9.9 服务器关闭
10 创建 TCP 客户端:
10.1 创建客户端套接字 socket()
10.2 尝试连接服务器 connect()
10.3 通讯循环
10.4 对话 send()/recv()
10.5 关闭套接字 close()
11 创建 UDP 服务器: UDP 是面向无连接的, 所以只要等待连接就可以了
11.1 创建一个服务器套接字 socket()
11.2 绑定服务器套接字 bind()
11.3 服务端循环
11.4 对话 recvfrom()/sendto()
11.5 关闭服务器套接字 close()
12 创建 UDP 客户端
12.1 创建客户端套接字 socket()
12.2 通讯循环
12.3 对话 sendto()/recvfrom()
12.4 关闭客户端套接字 close()
13 套接字模块的属性
13.1 数据属性
13.1.1 AF_UNIX, AF_INET, AF_INET6, Python 支持的套接字家族
13.1.2 SOCK_STREAM, SOCK_DGRAM　套接字类型
13.1.3 has_ipv6　是否支持 IPV6
13.2 异常
13.2.1 error　套接字相关错误
13.2.2 herror　主机和地址相关错误
13.2.3 gai　地址相关错误
13.2.4 timeout　超时
13.3 函数
13.3.1 socket()　创建套接字对象
13.3.2 socketpair()　创建套接字对象

13.3.3 fromfd()　用一个已经打开的文件描述符创建一个套接字对象

13.4 数据属性

13.4.1 ssl()　在套接字初始化一个安全套接字层，不做证书验证

13.4.2 getaddrinfo()　得到地址信息

13.4.3 getfqdn()　返回完整的域的名字

13.4.4 gethostname()　得到当前主机名

13.4.5 gethostbyname()　由主机名得到对应的 ip 地址

13.4.6 gethostbyname_ex()　gethostbyname()的扩展版本，返回主机名，主机所有别名和 IP 地址列表.

13.4.7 gethostbyaddr()　由 IP 地址得到 DNS 信息，返回一个类似 gethostbyname_ex()的 3 元组

13.4.8 getprotobyname()　由协议名得到对应的号码

13.4.9 getservbyname()　由服务名得到对应的端口号或相反

13.4.10 getservbyport　两个函数中，协议名都是可选的.

13.4.11 ntohl()/ntohs()　把一个整数由网络字节序转为主机字节序

13.4.12 htonl()/htons()　把一个整数由主机字节序转为网络字节序

13.4.13 inet_aton()/inet_ntoa()　把 IP 地址转为 32 位整型，以及反向函数(仅对 IPV4 有效)

13.4.14 inet_pton()/inet_pton()　把 IP 地址转为二进制格式以及反向函数(仅对 IPV4 有效)

13.4.15 getdefaulttimeout()/setdefaulttimeout()　获取/设置默认的套接字超时时间.(浮点数)

14 SocketServer 模块: 用于简化网络客户与服务器的实现.

14.1 BaseServer　包含服务器的核心功能与混合(mix-in)类的钩子功能，该类用于派生，不要直接生成

14.2 TCPServer/UDPServer　基本的网络同步 TCP/UDP 服务器

14.3 UnixStreamServer/UnixDatagramServer　基本的基于文件同步的 TCP/UDP 服务器

14.4 ForkingMixIn　实现了核心的进程化或线程化的功能，用于和服务器类进行混合，提供一些异步特性.

14.5 ThreadingMixIn　和 ForkingMixIn 相同，提供线程服务，不要直接生成这个类的实例

14.6 ForkingTCPServer/ForkingUDPServer　ForkingMixIn 和 TCPServer/UDPServer 的组合

14.7 ThreadingTCPServer/ThreadingUDPServer　ThreadingMixIn 和 TCPServer/UDPServer 的组合

14.8 BaseRequestHandler　包含处理服务请求的核心功能.　只用于派生新的类，不要直接生成该类对象，　可以考虑使用 StreamRequestHandler 或 DataRequestHandler

14.9 StreamRequestHandler/DatagramRequestHandler　TCP/UDP 服务器的请求处理类的一个实现.

15 使用 SocketServer 模块创建一个 TCP 服务器, SocketServer 的请求处理器的默认行为是接受连接，得到请求，然后就关闭连接，跟 UDP 的服务方式类似，为了保持连接，需要重写请求处理器中的其他方法.

15.1 实现一个继承请求处理器(StreamRequestHandler)的类，重写 handle 方法

15.2 实例化一个 TCPServer 的实例，传入自己的请求处理器类.

15.3 开启服务: tcpserver.serve_forever()

16 Twisted 框架(http://twistedmatrix.com)

16.1 Twiste 是一个完全事件驱动的网络框架.　允许使用和开发完全异步的网络应用程序和协议

16.2 包含: 网络协议, 线程, 安全和认证, 聊天/即使通讯, 数据库管理, gaunxi 数据库集成, 网页/互联网, 电力邮件, 命令行参数, 图形界面集成.

16.3 使用 Twisted Reactor TCP 服务器

16.3.1 服务端:

16.3.1.1 继承 twisted.internet.protocol.Protocol 实现一个服务协议

16.3.1.2 服务协议类中重写 connectionMake(self)方法: 使用 self.transport.getPeer().host 可以获取客户端信息

16.3.1.3 服务协议类中重写 dataReceived(self, data)方法, data 就是接收到的数据, 在这里可以使用 self.transport.write()向客户端发送数据

16.3.1.4 主程序中, 创建一个 twisted.internet.protocol.Factory()实例, 并设置 factory.protocol = 上面实现的服务协议类

16.3.1.5 调用 twisted.internet.reactor.listenTCP(PORT, factory)开始监听

16.3.1.6 调用 twisted.internet.reactor.run()开启服务

16.3.2 客户端

16.3.2.1 继承 twisted.internet.protocol.Protocol 实现一个客户端的协议类

16.3.2.2 协议类中重写 sendData(self)方法, 用于发送数据的控制

16.3.2.3 协议类中重写 connectionMade(self)方法, 用于创建连接, 由于是客户端, 这里只要调用 sendData 就可以了(也就是连接并发送数据)

16.3.2.4 协议类中重写 dataReceived(self, data)接收并处理数据, 并进行响应

16.3.2.5 继承 twisted.internet.protocol.ClientFactory 实现一个客户端的工厂类

16.3.2.6 客户端工厂类中定义 protocol 属性为上面实现的协议类

16.3.2.7 客户端工厂类中定义 clientConnectionLost = clientConnectionFailed = lambda self, connector, reason: reactor.stop() 指示连接失败处理方式

16.3.2.8 主程序中调用 twisted.internet.reactor.connectTCP(HOST, PORT, factory)连接服务器

16.3.2.9 reactor.run()开启客户端服务

17 相关模块:

17.1 select: 通常在底层套接字程序中与 socket 模块联合使用, 提供的 select()函数可以同时管理套接字对象, select()函数会阻塞, 直到至少有一个套接字准备好要进行通讯的时候才退出.

17.2 asyncore/asynchat 和 SocketServer 模块在创建服务器方面都提供了高层次的功能, 由于是基于 socket 和 select 模块, 封装了所有的底层代码, 所以开发简单. async*模块是唯一的异步开发支持库.

# 网络客户端编程(chapter17)

1 常见的文件传输协议:

1.1 FTP

1.2 UUCP: Unix-to-Unix 复制协议

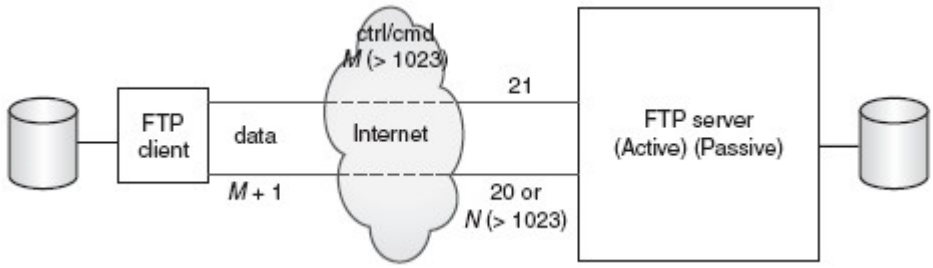
1.3 HTTP

1.4 rcp: Unix 下的远程文件复制指令

1.5 较 rcp 更灵活的 scp 和 rsync

2 FTP 是 C/S 中的一个特例, 客户端和服务器都使用两个套接字来通讯, 一个是控制和命令端

口(21), 另一个是数据端口(20)




3 FTP 的客户端和服务器使用指令和控制端口发送 FTP 协议, 数据通过数据端口传输.

4 FTP 有主动和被动两种模式

4.1 只有主动模式服务器才使用数据端口. 服务器将 20 端口设置为数据端口后, 它"主动"连接客户端的数据端口.

4.2 被动模式中, 服务器只是告诉客户端它的随机端口的号码, 客户端必须主动建立数据连接, 在这种模式下, FTP 服务器在建立数据连接时是被动的.

5 python 构建 FTP 客户端

5.1 使用 ftplib 模块

5.2 程序流程

5.2.1 连接到服务器, 创建一个 ftplib.FTP(HOST)实例

5.2.2 登录 ftp.login('anonymous', 'guess@who.org'), 调用 ftp 的登录接口

5.2.3 发出服务请求(有可能有返回信息)

5.2.4 退出 ftp.quit()

5.3 ftplib.FTP 类方法

5.3.1 login(user = 'anonymous', passwd = 'anonymous@', acct = '')登录到 FTP 服务器, acct 参数的意义是帐号的更详细的描述信息.

5.3.2 pwd() 得到当前工作目录

5.3.3 cwd(path) 改变当前工作目录

5.3.4 dir([path[, ...[, cb]]]) 显示 path 目录中的内容, 可选参数 cb 是一个回调函数, 会被传给 retrlines()方法. 回调函数可以接受一个参数, 每次读取的一行会被传入回调函数进行处理.

5.3.5 nlst([path[, ...]])与 dir()类似, 但返回一个文件名的列表, 而不是显示这些文件名

5.3.6 retrlines(cmd [, cb]) 个项 FTP 命令(如 RETR filename), 用于下载文本文件. 可选的回调函数 cb 用于处理文件的每一行

5.3.7 retrbinary(cmd, cb[, buffersize = 8192[, ra]]) 与 retrlines()类似, 只是这个指令处理二进制文件. 回调函数 cb 用于处理每一块(块大小默认为 8k)下载的数据

5.3.8 storlines(cmd, f) 给定 FTP 命令(如"STOR filename"), 以上传文本文件. 要给定一个文件对象 f

5.3.9 storbinary(cmd, f[, bs=8192]) 与 storlines()类型, 但是这个指令用来处理二进制而年间. 要给定一个文件对象 f 和一个上传块大小 bs, 默认 8kb

5.3.10 rename(old, new) 对远程文件进行改名

5.3.11 delete(path) 删除位于 path 的远程文件

5.3.12 mkd(directory) 创建远程目录

5.3.13 rmd(directory) 删除远程目录

5.3.14 quit() 关闭连接并退出

6 FTP 的处理细节及其他:

```
def storbinary(self, cmd, fp, blocksize=8192):
    '''Store a file in binary mode.'''
    self.voidcmd('TYPE I')
    conn = self.transfercmd(cmd)
    while 1:
        buf = fp.read(blocksize)
        if not buf: break
        conn.sendall(buf)
    conn.close()
    return self.voidresp()

def storlines(self, cmd, fp):
    '''Store a file in line mode.'''
    self.voidcmd('TYPE A')
    conn = self.transfercmd(cmd)
    while 1:
        buf = fp.readline()
        if not buf: break
        if buf[-2:] != CRLF:
            if buf[-1] in CRLF: buf = buf[:-1]
            buf = buf + CRLF
        conn.sendall(buf)
    conn.close()
    return self.voidresp()
```

6.1 上面是 ftplib.py 中的 storbinary 和 storlines 两个函数的源码，通过分析源码，可以看出，python 支持的 ftp 客户端程序编写，主要是基于 FTP 服务端提供的 FTP 命令，无论何种方式，最基础的接口是仍然是命令的发送，FTP 通过首先发送一个命令，获取到服务端的响应，响应正常，则返回一个 socket 连接，对方等待文件传输，如果响应失败，则返回相应的异常消息即可.

6.2 四种常见 FTP 客户端

6.2.1 命令行客户端，提供自己的命令接口供用户和服务端交互

6.2.2 GUI 客户端：图形用户界面

6.2.3 网页浏览器：浏览器一般都允许直接的 FTP 交互

6.2.4 定制程序：用于特殊目的的 FTP 文件传输程序，一般不允许用户与服务器接触.

7 Usenet 和网络新闻组:

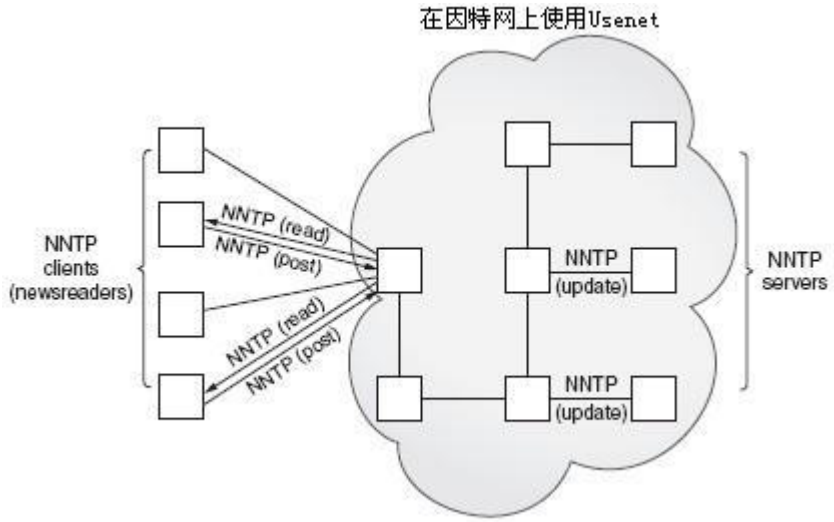
7.1 Usenet 新闻系统是一个全球存档的’电子公告板’.

7.2 Usenet 整个系统由大量计算机组成，计算机之间共享 Usenet 上的帖子. 用户发布的帖子会通过 Usenet 传播到其他相连的计算机上上.

7.3 每个系统都有一个它已经’订阅’的新闻组的列表，只接收它感兴趣的新闻组里的帖子.

7.4 系统可以设置权限控制(身份验证，操作权限控制等.)

8 网络新闻传输协议(NNTP)：供用户在新闻组中下载或发表帖子的协议.




8.1 加利福尼亚大学圣地亚哥分校和伯克利分销创建，记录在 RFC977. 1986 年 2 月公布，其后的更新记录在 RFC2980(2000 年 10 月公布)

8.2 NNTP 只使用 119 端口做通讯.

9 python 中的 NNTP(nntplib.NNTP)

9.1 流程

9.1.1 连接服务器　n = nntplib.NNTP('nntp_server')

9.1.2 登录(如果需要)　nntplib.NNTP(host[, port[, username[, password[, readermode][, usenetrc]]]])

9.1.3 发送请求: resp, count, first, last, name = n.group('comp.lang.python')

9.1.4 退出: n.quit()

9.2 nntplib.NNTP 类的方法

9.2.1 group(name)　选择一个组的名字.　返回一个元组(resp, count, first, last, name)，分别是: 服务器返回信息，文章数量，第一个和最后一个文章的号码，组名.　返回的所有数据都是字符串，返回的组名和传入的 name 应该相同

9.2.2 xhdr(header, string[, file])　发送一个 XHDR 命令, XHDR 命令没有在 RFC 中定义，但是是一个公共的扩展.header 是消息头的关键字(比如 subject), string 参数可以有一个表单'first-last'意思是第一个和最后一个文章号码去搜索，返回一个元组(response, list)　其中 list 是一个元组的列表(id, text)，其中, id 是文章的号码(string 类型), text 是文章内容.. 如果指定 file 参数, XHDR 命令的输出会存储到指定的文件中，如果 file 是一个 string，该方法会按照 file 指定的名称打开一个文件，写入并关闭.　如果 file 是一个文件对象，就直接调用 write 方法将 XHDR 的输出写入到文件.

9.2.3 body(id[, ofile])　给定文章的 id, id 可以是消息的 id(放到尖括号中)，或一个文章号(字符串)，返回一个元组(resp, anum, mid, data): 服务器返回消息，文章号，消息 id(放到尖括号中)，和文章所有行的列表或把数据输出到 ofile 指定的文件中

9.2.4 head(id)　和 body 类似，但是，返回的元组中那个行的列表中只包含了文章的标题

9.2.5 article(id)　和 body 类似，只是返回的元组中的那个行的列表中只包含了文章的标题和内容

9.2.6 stat(id)　让文章的指针指向 id，返回的还是和 body 返回一样的一个元组，但是不包含文章的数据.

9.2.7 next()　用法和 stat()类似，只是将文章的指针移到下一篇文章.

9.2.8 last()　和 next()一样，只不过是移到了最后一篇文章.

9.2.9 post(ufile)　上传 ufile 文件对象中的内容(使用 ufile.readline())　并在当前新闻组发表

9.2.10 quit()　关闭连接，然后退出

10 交互式 NNTP 的简单实现(无网络条件暂时未实现.)

11 电子邮件: RFC2822 中定义：消息由头域(消息头)以及后面可选的消息体组成.　邮件头是必要的

11.1 消息传输代理(MTA, Message Transfer Agency)：邮件交换主机上运行的一个服务器程序，负责邮件的路由，队列和发送工作.　也就是邮件从源主机到目的主机所要经过的跳板.

11.2 MTA 负责的事情:

11.2.1 如何找到消息应该去的下一台 MTA：由 DNS 来查找目的域名的 MX(Mail eXchange, 邮件交换)来完成，对于最后的收件人来说是非必须的，但对于其他的跳板来说，是必须的.

11.2.2 如何与另一台 MTA 通讯：依靠消息传输系统(MTS, MTA 之间通讯使用的协议, Message Transport System)，只有两个 MTA 都使用这个协议时，才可以通讯..　由于互联网络的复杂性, 1982 年提出了 SMTP(简单邮件传输协议, Simple Mail Transfer Protocol)

11.3 SMTP 是因特网上 MTA 之间用于消息交换的最常用的 MTS…被 MTA 用来把 email 从一台主机传送到另一台主机.　在发送 email 时，必须要连接到一个外部的 SMTP 服务器.　此时，邮件程序是一个 SMTP 客户端，对应的 SMTP 服务器就是消息的第一个跳板.

12 python 中的 SMTP: smtplib.SMTP

12.1 流程:

12.1.1 连接到服务器　n = smtplib.SMTP([host[, port[, local_hostname[, timeout]]]]):　如果给定了 host 和 port 参数，初始化的时候就会自动调用 connect 接口去连接指定的 SMTP 服务器.　如果没有正确的返回，抛出一个 SMTPConnectError, timeout 参数用于指定超时的秒数

12.1.2 登录(如果需要的话)　login(username, password)

12.1.3 发出服务请求　n.sendmail(from_addr, to_addr, msg[, mail_options, rcpt_options])

12.1.4 退出 n.quit()

12.2 消息在传输过程中, SMTP 不能以任何理由修改邮件消息

12.3 smtplib.SMTP 类方法：多数 e-mail 发送程序中, sendmail 和 quit 方法是必须的.

12.3.1 sendmail 的所有参数都要遵循 RFC2822, email 地址要有正确的格式，消息体要有正确的前导头，前导头后面是两个回车和换行(\r\n)对

12.3.2 RFC2822 中要求的必须的头信息只有发送日期和发送地址，也就是 Date:和 From:

12.3.3 sendmail(from_addr, to_addr, msg[, mail_options, rcpt_options])：将消息 msg 从 from_addr 发送到 to_addr(列表或元组)…… ESMTP 设置(mopts)和收件人设置(ropts)为可选

12.3.4 quit()　关闭连接，然后退出

12.3.5 login(username, password)　登录到 SMTP 服务器

13 交互式 SMTP:

smtp = smtplib.SMTP('smtp.qq.com')　连接服务器

smtp.login(username, password)　登录服务器

smtp.set_debuglevel(1)　　//设置为调试状态

smtp.sendmail('285821471@qq.com', 'lgg860911@yahoo.com.cn', '''From: 雷果国\r\nTo: 雷果国\r\nSubject: 你好.''')　发送邮件

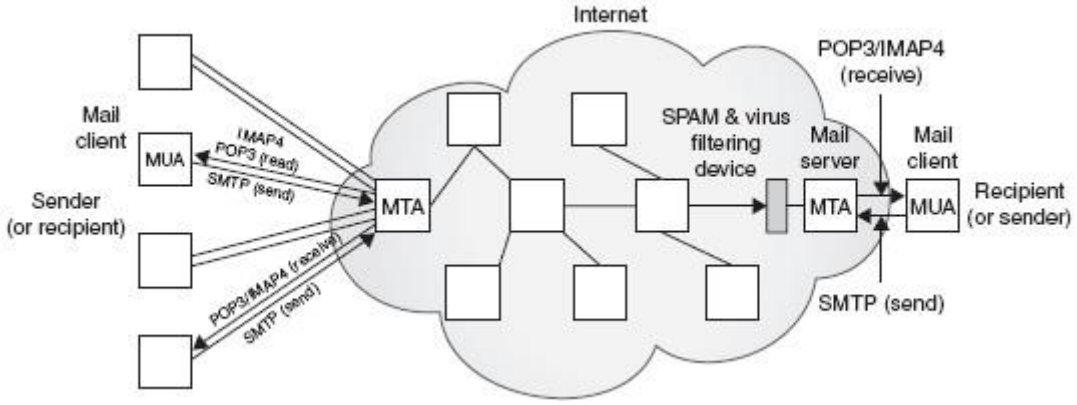
smtp.quit()　退出…

14 邮件接收:

14.1 用于下载邮件的第一个协议叫做邮局协议，目的是让用户的工作站可以访问邮箱服务器里的邮件.

14.2 POP3 是 POP 协议的最新版

14.3 IMAP: 交互式邮件访问协议，现在被使用的是 IMAP4rev1




15 python 中的 pop3: poplib.POP3

15.1 流程

15.1.1 连接服务器: p　= POP3('pop.qq.com')

15.1.2 登录: p.user(), p.pass_()

15.1.3 处理

15.1.4 退出: p.quit()

15.2 示例:

```
import poplib

yahoo = poplib.POP3('pop.mail.yahoo.com.cn')
yahoo.user('lgg860911')
yahoo.pass_('2a0d1q22f9')
yahoo.list()　　//得到邮件列表，返回一个元组的列表，元组中，第一个元素是服务器返回消息，第二个元素是消息的列表，第三个元素是返回信息的大小.
yahoo.retr(192)　//根据邮件编号获取邮件内容，并将其设置为'已读'标志.　返回一个和 list 一样的元组
```

15.3 poplib 中的方法

15.3.1 user(username)　发送用户名登录

15.3.2 pass_(password)　发送密码

15.3.3 stat()　返回邮件的状态，长度为 2 的元组(msg_ct, mbox_size)　是消息的数量和消息的总大小

15.3.4 list()

15.3.5 retr

15.3.6 dele(msgnum)根据邮件编号将其标记为删除.　大多数服务器在调用 quit()后执行删除操作

15.3.7 quit()　登出，保存修改.　解锁邮箱，结束连接，然后退出.

15.3.8 使用 list()方法处理的时候，参数 msg_list 的格式是: ['msgnum msgsize', …]其中，msgnum 和 msgsize 分别是每个消息的编号和消息的大小.

# 多线程编程(chapter18)

1 多线程编程的适合场景: 本质上就是异步的, 需要有多个并发事务, 各个事务的运行顺序可以是不确定的, 随机的, 不可预测的. 这种编程任务可以被分成多个执行流, 每个流都有一个要完成的目标, 根据应用的不同, 这些子任务可能都要计算一个中间结果, 用于合并得到最后的结果.

1.1 运算密集型的任务一般都比较容易分隔出多个子任务, 顺序或异步执行.

1.2 多个外部输入源的任务使用单线程方式很难解决(可能得需要使用计时器方式实现.)

1.3 顺序执行的程序必须使用非阻塞 I/O 或是带有计时器的阻塞 I/O, 因为用户的如入到达是不确定的.

2 使用多线程编程和一个共享的数据结构如 Queue, 这种程序任务可以用几个功能单一的线程来组织:

2.1 UserRequestThread: 负责读取客户的输入, 可能是一个 IO 信道. 程序可能创建多个线程, 每个客户一个, 请求会被放入到队列中

2.2 RequestProcessor: 一个负责从队列中获取并处理请求的线程, 会为下面那种线程提供输出

2.3 ReplyThread: 负责把给用户的输出取出来, 如果是网络应用程序就把结果发送出去, 否则就保存到本地文件系统或数据库中.

3 进程: 有时被称为重量级进程, 是程序的一次执行. 每个进程都有自己的地址空间, 内存, 数据栈和其他记录其运行轨迹的辅助数据. 操作系统管理在其上运行的所有进程, 并为这些进程公平的分配时间篇. 进程可以通过 fork 和 spawn 操作来完成其他任务. 进程之间的通讯使用 IPC. 而不能直接共享信息.

4 线程: 有时被成为轻量级进程. 所有的线程都是运行在同一个进程中, 共享相同的运行环境.

4.1 三种状态: 开始, 顺序执行, 结束

4.2 线程拥有自己的指令指针, 记录自己运行到什么地方. 线程的运行可能被抢占(中断), 或暂时的被挂起(睡眠), 让其他的线程运行, 这也叫做让步.

4.3 一个进程中的多个线程之间共享同一片数据空间, 所以, 线程之间的通讯比进程之间的通讯更加方便.

4.4 线程一般都是并发执行的, 并行和数据共享机制使得多任务合作成为现实

4.5 危险性: 多线程同时访问同一片数据的时候, 可能导致脏读等情况的发生. 这种情况叫做竞态条件, 这种危险的解决使用线程同步.

4.6 另一个危险来源于阻塞: 有的函数会在完成之前阻塞住, 在没有特别为多线程做修改的情况下, 这种”贪婪”的函数会让 CPU 时间分配有所倾斜, 导致多线程分配到的运行时间不尽相同, 不公平.

5 线程或进程的并行, 实际上都是线性的, CPU 分出时间片, 每个进程, 线程会根据优先级等 CPU 时间片分配策略得到相应的执行时间.