# 条件和循环

## If语句

1. if语句的语法：

If expr:

if代码块

* 1. if代码块只有在expr结果为True时才执行，否则执行下一条语句

1. 多重条件表达式：
   1. if表达式可以通过布尔操作符and,or,not实现多重判断条件
2. 单一语句的代码块：
   1. 如果if代码块只有一行，可以并在一行，不过因为可读性不好，不推荐：
      1. if true: print(“hello”)

## else语句

1. else语句语法：

if expr:

if代码块

else:

else代码块

* 1. if代码块只有在expr结果为True时才执行，否则执行else代码块

1. 避免悬挂else：
   1. Python的缩进使用强制使代码正确对齐，让程序员决定else属于哪个if，避免了悬挂else，减少了不确定性

## elif语句

1. elif语法：

if expr1:

if代码块

elif expr2:

elif代码块1

elif expr3:

elif代码块2

else:

else代码块

1. elif和else都是可选的，elif可以有任意多个，而else最多有一个

## 条件表达式（三元操作符）

1. 类似于与java和c的三元操作符a<b?c:d
2. 在python中可以写为c if a<b else d

## while语句

1. while循环的语法：

while expr:

while代码块

* 1. 如果表达式为真，会一直执行while代码块，直到表达式为假

1. 计数循环：
   1. 为了执行有限次数，可以加一个计数作为表达式判断，并在while代码块中对计数进行更改，以避免死循环
2. 无限循环：
   1. 如果判断表达式一直为真，就会无限循环下去
   2. 许多通讯服务器的客户端/服务器系统就是通过无限循环执行等待的

## for语句

1. 简介：
   1. 可以遍历序列成员，会自动地调用迭代器的next()方法
   2. 和C、java等的for语句不同，它更像是foreach循环
2. 一般语法：

for var in iter:

for代码块

* 1. 从头遍历iter可迭代对象，直到结束
  2. 每次循环，var被设置为当前元素，提供给for代码块使用

1. 用于序列类型：
   1. 迭代字符串：for char in “hello”
      1. 迭代字符串时，迭代变量char只包含一个字符，并不常用
   2. 迭代列表：for var in [‘a’,’b’,’c’]
   3. 通过序列的索引迭代：for index in range(len(list))
      1. 之后通过list[index]访问序列元素
   4. 通过项和索引迭代：for index, var in enumerate([‘a’,’b’,’c’])
      1. 这样既可获得索引，也可获得元素值
      2. 索引依然是从0开始
2. 用于迭代器类型：
   1. 迭代器对象有一个next()方法：返回下个条目
   2. 当所有条目迭代完后，迭代器引发StopIteration异常，for语句捕获异常结束循环
3. range()内建函数：
   1. 通过range()，可以把for循环变成熟悉的JAVA形式的for循环
   2. range(start=0,end,step=1)：
      1. 实质是start和end的左闭右开区间，包含start不包含end
      2. start和step都可以省略，但**有step时必须有start**
4. xrange()内建函数：
   1. 类似于range()，但更适合范围很大的列表
   2. 它和range()不同的地方在于，不会在内存里创建列表的完整拷贝，性能更高
   3. 它只用于for循环中，for循环外没有意义
5. 与序列相关的内建函数：
   1. sorted()、zip()返回序列
   2. reverse()、enumerate()返回迭代器

## Break语句

1. 类似于java和c，python中的break也用于结束当前循环然后跳转到下一条语句

## Continue语句

1、类似于java和c，python中的continue也用于停止本次循环，并忽略剩余的语句，判断是否进入下一次循环

## Pass语句

1. 因为python没有大括号，而是用缩进对齐的方式。因此如果代码块里没有代码，解释器会视为语法错误。因此需要用到pass，对应java中的{}，即什么都不执行的空代码块

## 循环语句的else

1. 在其他语言中，else只会出现在条件判断里，但在python中else可以配合while和for使用
2. 用于循环后处理，只要循环是正常结束的（不是通过break等），else字句就会执行。

## 迭代器和iter()函数

1. 什么是迭代器：
   1. Python的迭代无缝支持序列对象，也允许迭代非序列类型，包括自定义对象
   2. Python让迭代器可以像序列一样操作，用计数的方法从头到尾遍历
2. 为什么要用迭代器：
   1. 对列表、字典迭代带来了性能上的增强
   2. 创建了真正的迭代接口，而不像原来那样随即对象访问
   3. 迭代非序列集合时，可以是代码更简洁易读
3. 如何迭代：
   1. 迭代器实际就是一个有next()方法的对象，而不是通过索引计数
   2. next()方法用于获取下一个条目
   3. 条目全部取出后，会引发StopIterator异常
      1. 这个异常不表示错误，只是为了告知迭代完成
   4. 迭代器的限制：
      1. 无法回到开始
      2. 无法向开始移动
      3. 不能复制一个迭代器，如果要再次迭代同一个对象，只能创建新的迭代器
   5. 迭代器方法：
      1. reversed()：返回一个反序访问的迭代器
      2. enumerate()：返回迭代器
      3. any()：如果迭代器某个条目为真，则返回真
      4. all()：如果迭代器所有条目都为真，则返回真
4. 使用迭代器：
   1. 序列：
      1. for循环会自动调用迭代器的next()方法，监视StopIteration异常
      2. while循环则：

itera = iter([‘a’,’b’,’c’])

while True:

try:

i = itera.next()

except StopIteration:

break

* 1. 字典：
     1. 字典的迭代器会遍历它的键
     2. for key in dict.keys()可以缩写为for key in dict
     3. 字典的呢间方法也可以定义迭代：dict.iterkeys()、dict.itervalues()、dict.iteritems()
  2. 文件：
     1. 文件对象生成的迭代器会自动调用readline()
     2. 因此可以用for line in file 替换 for line in file.readline()

file = open(‘filename.txt’)

for line in file:

print(line)

1. 可变对象和迭代器
   1. 对迭代元素改变时，更新会立即反映。
   2. 因此和其他语言的迭代器一样，在迭代可变对象时最好不要修改它们，以免造成不可预知的错误。
2. 如何创建迭代器：
   1. iter(obj)：创建对象的迭代器
   2. iter(func,sentinel)：重复调用func，直到next()返回的值为sentinel

## 列表解析

1. 列表解析来自于函数式编程语言haskell
2. 列表解析语法：

[expr for var in iter]

1. 列表解析的核心是for循环，expr应用于序列的每一个成员，最后的结果值是该表达式产生的列表
2. 结合if语句的列表解析扩展语法：

[expr for var in iter if expr2]

1. 样例：
   1. 三行五列矩阵：[(x+1,y+1) for x in range(3) for y in range(5)]
   2. 计算非空白字符数目：len([word for line in file for word in line.split()])

## 生成器表达式

1. 生成器表示式是列表解析的扩展，为了解决列表解析的一个不足：
   1. 列表解析必须生成所有的数据，用以创建整个列表
2. 生成器是特定的函数，允许返回一个值，然后暂停代码的执行，稍后恢复。
3. 生成器和列表解析语法基本相同，但并不真正创建数字列表，而是返回一个生成器
   1. 使用了延迟计算的方式
   2. 在内存上更有效
4. 生成器表达式语法：

(expr for var in iter if expr2)

1. 生成器不会取代列表解析，它的目的只是让内存更友好

# 文件和输入输出

## 文件对象

1. 文件对象不仅可以用来访问普通的磁盘文件，也可以访问任何其他类型抽象层面的文件
2. 文件只是连续的字节序列。
3. 数据的传输经常用到字节流，不管是单个字节还是大块数据

## 文件内建函数

1. open()和file()提供了初始化输入/输出的通用接口
2. open()的基本语法：file = open(file\_name,access\_mode=’r’,buffering=-1)
   1. 成功打开文件后会返回一个文件对象，否则引发一个错误
   2. 文件名是一个字符串，可以是绝对路径，也可以是相对路径
   3. 文件模式默认值是r，即只读：
      1. r：以读的方式打开
      2. rU或Ua：以读方式打开，同时提供通用换行符支持
      3. w：以写的方式打开（必要时清空）
      4. a：以追加模式打开（从EOF开始追加，必要时创建新文件）
      5. r+：以读写模式打开
      6. w+：以读写模式打开（参见w）
      7. a+：以读写模式打开（参见a）
      8. rb：以二进制读模式打开
      9. wb：以二进制写模式打开（参见w）
      10. ab：以二进制写模式打开（参见w）
      11. rb+：以二进制读写模式打开（参见r+）
      12. wb+：以二进制读写模式打开（参见w+）
      13. ab+：以二进制读写模式打开（参见a+）
      14. 使用’r’或’U’模式打开文件必须已经存在
      15. 使用’w’模式打开的文件如果存在则先清空，再重新创建
          1. 如果文件不存在，将自动创建
      16. 使用’a’模式打开的文件都从末尾开始写，即使seek()到其它地方
          1. 如果文件不存在，将自动创建
   4. buffering用于指示访问文件采用的缓冲方式
      1. 0表示不缓冲
      2. 1表示只缓冲一行
      3. 任何大于1的值代表缓冲区的大小
      4. 不提供或提供负值代表使用系统默认缓冲机制，一般情况用系统默认缓冲即可
3. 工厂函数file()：
   1. file()和open()功能相同，可以任意替换
   2. 建议用open()读写文件，用file()说明在处理文件对象
4. 通用换行符支持（UNS）
   1. 不同平台表示行结束的符号是不同的
   2. 使用’U’模式打开文件时，所有的换行符都会被替换为\n
   3. 文件对象的newlines属性会记录原本的换行符
      1. 如果文件刚被打开，还没遇到换行符，则newlines为None
      2. 如果遇到了多种换行符，newlines会被设置成元组
   4. UNS只用于读取文本文件，没有对应的处理文件输出的方法

## 文件内建方法

1. 所有对文件的操作，都将通过open()返回的句柄完成
2. 输入：
   1. read(size=-1)：直接读取size个字节到字符串中，包括行结束符
      1. 如果为默认参数，或者size值为负，文件将被读取直至末尾
   2. readline(size=-1)：读取文件的一行，包含结束符，一起作为字符串返回
      1. 如果提供了size，那么在读取size个字节后会返回不完整的行
   3. readlines(size=-1)：读取所有剩余的行，把它们作为一个字符串列表返回
      1. 如果提供了size，那么返回的所有行大约有size个字节
         1. 为了凑齐缓冲区可能比size稍大
   4. next()：返回文件的下一行，没有行时引发StopIteration异常
3. 输出：
   1. write()：把含有文本数据或二进制数据块的字符串写入文件
      1. 行结束符不会自动加入，如果需要，必须手动加入
   2. writelines(str\_list)：接受一个字符串列表作为参数，将它们写入文件
      1. 行结束符不会自动加入，如果需要，必须手动加入
4. 文件内移动：
   1. seek(size,offset=0)：可以在文件中移动文件指针到不同的位置
      1. offset为0表示从头算起
      2. offset为1表示从当前位置算起
      3. offset为2表示从文件末尾算起
   2. tell()：返回当前文件指针在文件中的位置
      1. 从文件开始算起，单位为字节
5. 文件迭代：
   1. 一行一行访问文件：for line in file
      1. 其中line代表文件的一行（包括行结束符），是个字符串
   2. 老方法是for line in file.readline()，已经不用了，文件迭代更为高效
6. 其他：
   1. close()：通过关闭文件结束对它的访问
      1. 如果不显式关闭文件，可能就会丢失输出缓冲区的数据
      2. Python垃圾收集机制也会在文件对象的引用计数为0时自动关闭文件
   2. flush()：直接把内部缓冲区的数据立刻写入文件，而不是被动等待
   3. truncate()：将文件截取到当前文件指针位置或给定size，size以字节为单位
7. 文件方法杂项：
   1. Python的os模块有几个比较有用的属性：
      1. linesep：用于在文件中分隔行的字符串，即换行符
      2. sep：用于分隔文件路径名的字符串
      3. pathsep：用于分隔文件路径的字符串
      4. curdir：当前工作目录的字符串名称
      5. pardir：当前工作目录父目录字符串名称
   2. print()默认在输出内容末尾加一个换行符

## 文件内建属性

1. 文件对象有一些数据属性，保存了文件对象相关的附加数据：
   1. file.closed：True表示文件已经被关闭，否则为False
   2. file.encoding：文件使用的编码，为None表示系统默认编码
   3. file.mode：文件打开时使用的访问模式
   4. file.name：文件名
   5. file.newlines：
      1. 未读取到行分隔符时为None
      2. 只读取过一种行分隔符时为一个字符串
      3. 读取过多种行分隔符时为一个字符串列表
   6. file.softspace：
      1. 为0时表示在输出一行数据后，要加上一个空格
      2. 为1时表示不加

## 标准文件

1. 一般来说，只要python程序一执行，就可以访问三个标准文件：
   1. 标准输入：一般为键盘
   2. 标准输出：到显示器的缓冲输出
   3. 标准错误：到屏幕的非缓冲输出
2. 在程序执行后，这三个文件就已经被预先打开了，只要知道它们的文件句柄就可以随时访问。
3. Python可以通过sys模块访问这些文件句柄：sys.stdin,sts.stdout,sys.stderr
   1. print通常是输出到sys.stdout
   2. input()通常是从sys.stdin接受输入

## 命令行参数

1. 命令行参数是调用某个程序时，除了程序名以外的其他参数
2. sys模块通过sys.argv属性提供了对命令行参数的访问
3. sys.argv实名行行参数的列表，sys.argv[0]永远是程序的名称
4. len(sys.argv)是命令行参数的个数，相当于argc
5. 命令行参数使程序员可以在启动一个程序的时候对程序行为作出选择。
6. 还有两个模块辅助处理命令行参数：
   1. getopt：比较简单，但不是很精细
   2. optparse：比较强大，且面向对象

## 文件系统

1. 对文件系统的访问大多通过python的os模块实现，它是python访问操作系统功能的主要接口。
2. 实际上真正与操作系统交互的是posix、nt、mac、dos等接口，但不需要手动导入，只要导入了os模块，python会自动选择正确地模块，不需要人为考虑底层工作
3. sys.path.append(path)可以将某个系统路径加入当前
   1. 这样导入模块时就可以直接导入
4. os模块除了负责处理大部分文件系统操作外，还对进程和进程环境进行管理。
   1. 常用文件处理函数：
      1. remove()、unlink()：删除文件
      2. tmpfile()：创建并打开（w+b的方式）一个新的临时文件
      3. stat()：返回文件信息
      4. walk()：生成一个目录树下的所有文件名
   2. 常用目录函数
      1. chdir()、fchdir()：改变当前工作目录
      2. chroot()：改变当前进程的根目录
      3. listdir()：列出指定目录的文件
      4. mkdir()、makedirs()：创建（多层）目录
      5. rmdir()、removedirs()：删除（多层）目录
   3. 常用访问权限函数：
      1. access()：检验权限模式
      2. chmod()：改变权限模式
      3. umask()：设置默认权限模式
   4. 文件描述符操作
      1. open()：底层的操作系统打开
      2. read()、write()：根据文件描述符读写数据
5. os.path模块可以完成一些针对路径名的操作
   1. 分隔：
      1. basename()：去掉目录路径，返回文件名
      2. dirname()：去掉文件名，返回目录路径
      3. join()：将分离的目录路径和文件名组成一个完整的路径名
   2. 信息：
      1. getatime()：返回最近访问时间
      2. getctime()：返回文件创建时间
      3. getmtime()：返回最近修改时间
      4. getsize()：返回文件大小
   3. 查询：
      1. exists()：指定目录或文件是否存在
      2. isabs()：指定路径是否为绝对路径
      3. isdir()：指定路径是否存在且是否为一个目录
      4. isfile()：指定路径是否存在且是否为一个文件
      5. islink()：指定路径是否存在且是否为一个链接
      6. ismount()：指定路径是否存在且是否为一个挂载点
      7. samefile()：两个路径名是否指向同一个文件

## 文件执行、永久存储模块、相关模块

跳过

# 错误和异常

## 什么是异常

1. 错误：语法错误和逻辑错误
   1. 语法错误：代码结构有误或拼写有误，导致不能被解释器解释或编译器无法编译
      1. 这些错误必须在程序执行前纠正
   2. 逻辑错误：不完整或不合法的输入；生成、计算、输出结果需要的过程无法执行
2. 异常：
   1. 所有的标准异常都是使用类实现的。
   2. 异常：因为程序出现了错误而在正常的控制流以外采取的行为
   3. 分为两个阶段：
      1. 引起异常发生的错误
      2. 检查阶段，采取可能的措施
   4. Python的异常也可以是程序员自己引发的。
   5. 异常发生时当前流被打断，采取相应的操作补救

## Python中的异常

1. 所有错误，无论是语义上还是逻辑上的，都是由于与python解释器不相容导致的，后果就是引发异常。
2. 几种异常：
   1. NameError：尝试访问一个未申明的变量
   2. ZeroDivisionError：除数为零
   3. SyntaxError：python解释器语法错误
      1. 这是唯一不在运行时发生的异常
   4. IndexError：请求的索引超出序列范围
   5. KeyError：请求一个不存在的字典关键字
   6. IOError：输入/输出错误
   7. AttributeError：尝试访问未知的对象属性

## 检测和处理异常

1. try-except语句：
   1. 语法：

try:

异常发生代码块

except 异常,[原因]:

处理异常代码块

* 1. 在程序运行时，解释器尝试执行try里的代码块
     1. 如果代码块完成后没有异常发生，就会忽略except语句继续执行
     2. 如果有异常发生，就会直接跳转到except语句执行相应处理代码
  2. try代码块中，如果异常发生了，那么之后的语句都不会再执行了，而是转到except
  3. 如果没有合适的处理代码，异常就会一层一层向上移交。如果移交到最顶层还没有对应的处理器，python解释器就会显示跟踪返回信息，然后退出

1. 带有多个except的try语句：
   1. try-except语句中，可以连接多个except，捕获多种异常
   2. 语法：

try:

异常发生代码块

except 异常1,[原因1]:

处理异常代码块1

except 异常2,[原因2]:

处理异常代码块2

1. 处理多个异常的except语句：
   1. 在同一个except里面可以捕获多个异常，这时多个异常要放在一个元组里
   2. 语法：

try:

异常发生代码块

except (异常1,异常2,...),[原因]:

处理异常代码块

1. 捕获所有异常：
   1. Exception是最顶层异常，任何异常都可以是Exception异常
   2. Except可以不指定异常，那么它将捕获所有异常，效果和捕获Exception一样

try:

异常发生代码块

except:

处理异常代码块

* 1. Except可以不指定异常，那么它将捕获所有异常，效果和捕获Exception一样
  2. 不推荐直接处理Exception，这样会忽略掉很多错误。
     1. 最好还是能精确地捕获某个异常

1. 异常原因：
   1. except子句的原因这个参数是可选的，它是一个包含诊断信息的**类实例**
      1. reason实际就是异常类实例
   2. 标准内建异常都含有异常原因这一个参数
   3. 如果try语句引发了异常，并且except提供了原因参数，那么相应的诊断信息会被自动封装入原因类实例汇总
2. else子句：
   1. 语法：

try:

异常发生代码块

except:

处理异常代码块

else：

else代码块

* 1. 如果try代码块全部运行成功，没有引发任何异常，才会执行else代码块

1. finally子句：
   1. finally代码块是无论异常是否发生都会执行的一段代码块
   2. 可以是try-finally，也可以是try-except-finally
   3. 常用用来做一些扫尾工作，比如关闭文件，关闭连接
2. try-finally语句：
   1. try单独和finally在一起，而不捕获异常
   2. 常常用来维持一致的行为，无论异常是否发生
   3. 语法：

try:

异常发生代码块

finally:

无论如何都执行的代码块

* 1. 如果产生一个异常，直接跳到finally代码块，否则执行完才跳

## 上下文管理

1. with语句：
   1. with语句是用来简化代码的
   2. 目的在于把try\except\finally关键字和资源分配释放的相关代码统统去掉
   3. 语法为：

with expr [as var]:

with代码块

* 1. as var就是把一个表达式赋值给一个变量
  2. with语句会完成所有准备工作和清理工作，都不用人工维护了
     1. 像关闭文件，或者异常处理等
  3. 支持with的对象：
     1. 基本记住file即可
     2. 其他省略

1. 上下文管理协议：
   1. 一般用于自定义与with工作的类
   2. 跳过

## 触发异常

1. raise语句：
   1. 由程序员主动地触发异常
   2. 语法：

raise [exception[,args[,traceback]]]

* + 1. excption可以是异常类、异常名字符串、或者异常实例
       1. 推荐使用异常类
    2. args：可以是元组，也可以是单独的对象。但最终都作为一个元组传入异常
       1. 一般是错误字符串，错误编号，错误地址等
    3. traceback：追踪对象，用于重新引发异常
  1. 当直接使用raise时，则触发前一个异常。如果之前没有异常，触发TypeError

## 断言

1. 如果断言为真，不采取任何措施，否则触发AssertionError异常
2. 断言语法：

assert expr[,args]

1. 在Python中断言是在运行时构建
2. 断言引发的异常同样可以用try-except捕捉
3. 可以把断言的工作原理看做以下：

def assert(expr,args=None):

if \_\_debug\_\_ and not expr:

raise AssertionError,args

## 标准异常

1、异常的结构：

- BaseException

|- KeyboardInterrupt

|- SystemExit

|- Exception

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

## 异常和sys模块

1. 通过sys.exc\_info()可以获取异常信息，可以获取三元组：
   1. exc\_type：异常类
   2. exc\_value：异常类实例
   3. exc\_traceback：追踪对象

## 相关模块

1. exceptions：内建异常
   1. 永远不需要导入这个模块
2. contextlib：为使用with语句的上下文对象工具
3. sys：包含各种异常相关的对象和函数

# 函数和函数式编程

## 什么是函数

1. 函数是对程序逻辑进行结构化或过程化的一种编程方法
2. 能够将整块代码巧妙地隔离成易于管理的小块
3. 能够把重复代码放到函数中而不是大量拷贝
4. 创建函数：

def func():

pass

1. 引用函数：

func

1. 调用函数：

func()

1. 返回值与函数类型：
   1. 函数会向调用者返回一个值
      1. 实际编程中大多不返回任何东西，在python中也就是返回None
   2. Python可以返回一个值或一个对象，包括元组、列表等
      1. 因为元组语法上可以省略括号，所以return a,b,c看着像是返回多个对象

## 调用函数

1. 函数操作符：
   1. 同大多数语言一样，用一对圆括号调用函数，任何输入的参数都放在括号中
2. 关键字参数：
   1. 可以通过关键字匹配参数的值，以此可以参数确实或不按顺序
      1. 参数缺失时仍然必须是有默认参数的情况
   2. 例如def func(a,b=1,c) func(c=3,a=2)
3. 默认参数：
   1. 声明了默认值的参数，在调用时不给该参数传入值也是允许的

## 创建函数

1. 函数使用def语句创建的，语法：

def func(args):

“文档注释”

函数代码块

1. Python将函数的声明和定义视为一体
2. 前向引用：
   1. Python不允许在函数未声明之前对它引用或调用
   2. 但是在一个函数里调用之后要声明的函数是可以的
   3. 也就是说，要直接调用一个函数，需要函数里所有函数都声明完才可以
3. 函数属性：
   1. 当两个函数都有同一个名字的属性时，命名空间和句点属性标识可以解决冲突
4. 内嵌函数：
   1. 可以在函数体内创建另外一个函数
   2. 内部函数在外部函数的作用域之内，如果外部没有对内部函数的引用，则无法调用内部函数，唯一的办法就是调用外部函数
5. 函数与装饰器：
   1. 语法为：

@dec1([args1])

@dec2([args2])

def func([args3]):

* 1. 以上等价于func = dec1(args1)(dec2(args2)(func))
  2. 装饰器：
     1. 可以知道，装饰器本质上就是函数，它们接受函数对象
     2. 装饰器是在函数调用之上的修饰
     3. 装饰器的目的是：在函数执行前执行些预备代码，函数执行后执行写清理工作
     4. 参考JAVA的AOP，可以用装饰器：
        1. 引入日志
        2. 给函数加入事务的能力
        3. 增加计时逻辑检测性能

## 传递函数

1. 函数是可以被引用的（以其他变量作为别名），相当于给函数区别名
   1. 比如def func() foo = func foo()
2. 也可以作为参数传入函数，这样可以在另一个函数中调用被传入的函数
3. 也可以作为列表和字典等容器的元素

## 形式参数

1. 位置参数：
   1. 必须以在被调用函数中定义的精确顺序来传递
   2. 如果没有默认参数，传入函数的参数数目必须与声明时参数数目一致
2. 默认参数：
   1. 在调用函数时如果没有提供参数，则可以使用默认参数提供的默认值
   2. 所有位置参数必须出现在任何一个默认参数之前
   3. 默认参数的好处：
      1. 预先选择一个最佳参数
      2. 提升程序的健壮性

## 可变长度的参数

1. 非关键字可变长参数：
   1. 在函数被调用时，所有形参（位置参数和默认参数）都被赋值给函数声明中对应的局部变量。
   2. 通常情况下，如果在调用时多给了参数会报错
   3. 也可以通过在函数声明增加非关键字可变长参数，允许多余的参数
   4. 语法为：

def func([formal\_args,] \*tuple)

* 1. 在调用时给的多余的非关键字参数就会被顺序填入tuple元组中

1. 关键字变量参数（字典）：
   1. 在函数在函数声明增加关键字变量参数，允许多余的参数
   2. 语法为：

def func([[formal\_args,] \*tuple,] \*\*dict)

* 1. 变量名加入字典的键，变量值加入字典的值
  2. \*和\*\*是为了区别非关键字参数和关键字参数
  3. 在调用时给的多余的关键字参数就会被顺序填入dict字典中

## 函数式编程

1. Python不是也不大可能称为函数式编程语言，但支持许多有价值的函数式编程语言构建
2. 匿名函数与lambda：
   1. Python允许用lambda关键字创造匿名函数。
   2. 语法为：

lambda [args1[,args2,...argsN]]:expression

* 1. lambda表达式的定义体必须和声明放在同一行，参数是可选的
  2. lambda表达式返回可调用的函数对象
     1. 它们可被传入给其他函数，用额外的引用别名化，作为容器对象以及作为可调用的对象被调用（如果需要的话，可以带参数）。当被调用的时候，如果给定相同的参数，这些对象会生成一个和相同表达式等价的结果。它们和那些返回等价表达式计算值相同的函数是不能区分的。
  3. Python中单行函数可以和标题写在同一行：def true():return True
     1. 等价于lambda表达式lambda:True
  4. 如果lambda表达式创建后没有引用保存，那么引用计数为0，会被垃圾回收
     1. 可以true = lambda:True，然后true()
  5. 例子：
     1. def add(x,y):return x+y -> lambda x,y:x+y
     2. def add2(x,y=2):return x+y -> lambda x,y=2:x+y
     3. def show\_tuple(\*z):return z -> lambda \*z:z

1. 内建函数apply()、filter()、map()、reduce()：
   1. 跳过
2. 偏函数应用：
   1. 跳过

## 变量作用域：

1、全局变量与局部变量：

* 1. 定义在函数内的变量有局部作用域，在模块中最高级别的变量有全局作用域
  2. 全局变量的一个特征是除非被删除掉，否则它们能存活到脚本运行结束。对于所有函数来说，它们的值都是可以被访问的。
  3. 局部变量仅仅依赖于定义它们的函数现阶段是否处于活动。当一个函数调用出现时，其局部变量就进入声明他们的作用域。一旦函数完成，变量离开作用域
  4. 搜索标识符：
     1. 当搜索一个标识符时，先搜索局部作用域，再搜索全局域，如果都没有则抛出NameError异常
     2. 因此局部变量如果和全局变量重名，会隐藏或者说覆盖全局变量

1. global语句：
   1. 为了明确地在函数体内引用一个已命名的全局变量，必须使用global语句
   2. 这样，函数体内就无需创建一个新的局部变量，而是使用全局变量即可。
   3. 语法为：

global var1[,var2,...]

1. 作用域的数字：
   1. Python从语法上支持多个函数嵌套级别，但在函数内都为同一个局部作用域
2. 闭包：
   1. 跳过
3. 作用域和lambda：
   1. Python的lambda匿名函数遵循和标准函数一样的作用域规则
4. 变量作用域和名字空间：
   1. 跳过

## 递**归**

1. 如果函数包含了对其自身的调用，该函数就是递归
2. 这个不多说，比如汉诺塔，阶乘，dfs等

## 生成器

1. 协同程序：可以运行的独立函数的调用，可以**暂停或者挂起，并从程序离开的地方继续或重新开始**。
   1. 当协同程序暂停时，我们能从其中获得一个中间的返回值
   2. 当调用回到程序中时，能够传入额外或者改变了的参数，但仍能够从上一次离开的地方继续，并且所有状态完整。
2. 挂起返回出中间值，并多次继续的协同程序被称为**生成器**。
   1. 生成器语法：

yield expr

* 1. 当生成器的next()方法被调用时，会准确从暂停的地方开始执行语句，返回yield后面的表达式，然后挂起。继续执行next()就执行下一条yield语句，直到没有yield

1. 简单地生成器特性：
   1. 当调用next()时，返回下一个值
   2. 如果函数结束没有更多值返回，一个StopIteration异常就会抛出
2. 加强的生成器特性：
   1. send()：将值回送给生成器，并从生成器返回值
   2. close()：退出生成器

# 模块

## 什么是模块：

1. 模块支持从逻辑上组织python代码。
2. 导入就是把其他模块的属性附加到当前使用的模块。

## 模块和文件

1. 模块是按照逻辑组织python代码的方法，而文件是物理层上组织模块的方法。
   1. 因此，一个文件可以看做一个独立模块，一个模块也可以看做一个文件。
   2. 模块的文件名就是模块的名字+扩展名.py
2. 模块名称空间：
   1. 一个名称空间就是一个从名称到对象的关系映射集合。
   2. 所有模块都有自己唯一的名称空间。不同模块间不会有出现名称交叉的现象
      1. 即使属性之间有名称冲突，通过点属性标识也防止了名称冲突
3. 搜索路径和路径搜索：
   1. 模块的导入需要路径搜索：在文件系统“预定义区域”查找模块.py文件
      1. 只要预定义区域包含这个文件，模块就会被正确导入
   2. 这些预定义区域就是python搜索路径的集合
   3. sys.path包含了模块的搜索路径的列表，可以通过append()增加路径
      1. 有需要时也可以通过insert()方法加入
   4. sys.modules是一个字典，键为模块名，值为对应物理地址
   5. 在有很多模块拷贝的情况下，解释器会沿搜索路径取找到的第一个模块

## 名称空间

1. 概念：
   1. 名称空间是名称（标识符）到对象的映射
   2. 执行期间有两个或三个活动的名称空间：全局、局部、内建
      1. 局部名称空间在执行期间是不断变化的
   3. Python解释器首先加载内建名称空间，随后加载全局名称空间
      1. 内建名称空间由\_\_builtins\_\_模块中的名字构成
   4. \_\_builtins\_\_模块和\_\_builtin\_\_模块：
      1. \_\_builtins\_\_模块包含内建名称空间中内建名字的集合
      2. \_\_builtin\_\_包含内建函数、异常，以及其他属性
      3. 在标准python执行环境下，\_\_builtins\_\_包含\_\_builtin\_\_的所有名字
2. 名称空间与变量作用域的比较：
   1. 名称空间是名字和对象间的映射关系
   2. 作用域指出了从用户代码的哪些物理地址可以访问这些名字
   3. 所有局部名称空间的名称都在局部作用范围内，局部作用范围以外的所有名称都在全局作用范围内。
   4. 局部名称空间和作用域会随函数的调用不断变化，全局名称空间是不变的
3. 名称查找，确定作用域，覆盖：
   1. 作用域的规则通过名称查询联系到名称空间
   2. 访问一个属性时，先查找局部名称空间，没有则继续查找全局名称空间，没有则继续查找内建名称空间，还没有则抛出NameError异常
   3. 如果局部名称空间的名称和全局名称空间的名称相同，则局部覆盖全局
4. 无限制的名称空间：
   1. 可以在任何需要放置数据的地方获得一个名称空间
   2. 可以在任何时候给函数添加属性

## 导入模块

1. Import语句
   1. 语法：

import module1[,module2,...]

* 1. 可以一行导入多个模块，也可以多行一个一个导入
     1. 一般推荐多行一个一个导入，这样可读性好
  2. 推荐所有在模块导入语句都写在python模块的开头部分
  3. 模块导入最好是这样的顺序：
     1. Python标准库模块
     2. Python第三方模块
     3. 应用程序自定义模块
     4. 使用一个空行分割这三类模块的导入语句
  4. 如果在一个模块的顶层导入，那么它的作用域是全局的；如果在函数中导入模块，那么它的作用域是局部的

1. from-import语句：
   1. 可以在模块里导入指定的模块属性，也就是把指定名称导入到当前作用域
   2. 语法为：

form module import name1[,name2,...]

1. 多行导入：
   1. 在一个模块导入多个属性时，import行会很长：

from module import name1,name2,name3,...

* 1. 可以用多行的from-import语句替代：

from module import name1

from module import name2

1. 扩展的import语句（as）：
   1. 可以使用自己想要的名字替换模块原有的名字：
   2. 语法为：

import module as xx

from module import attr as xxx

## 模块导入的特性

1. 载入时执行模块：
   1. 加载模块会导致这个模块被执行
      1. 被导入模块的顶层代码将直接被执行
      2. 包括设定全局变量以及类和函数的声明
      3. 如果有检查\_\_name\_\_的操作，那么也会被执行
         1. 但这可能并不是我们想要的，因此尽可能把代码封装到函数
         2. 只把函数和模块定义放入模块的顶层是良好的模块编程习惯
2. 导入import和加载load：
   1. 一个模块只被加载一次，无论它被导入多少次
   2. 这可以阻止多重导入时，代码被多次执行
3. 导入到当前名称空间的名称：
   1. 调用from-import可以把名字导入当前的名称空间
      1. 这意味着不需要使用句点属性表示来访问模块的标识符
   2. 限制使用from module import \*：
      1. 它会污染当前名称空间，可能会覆盖当前名称空间现有的名字
      2. 只有两个场合建议：
         1. 目标模块的属性非常多，反复键入模块名很不方便
         2. 交互解释器，可以减少输入次数
4. 关于\_\_future\_\_：
   1. 可以通过导入该模块尝试新特性或特性变化：
   2. 语法为：from \_\_future\_\_ import new\_feature
   3. 单纯使用import \_\_futrue\_\_不会有任何变化
5. 警告框架：
   1. 警告过滤器：
      1. 有多种级别的警告，警告的数量和类型是可控制的
      2. 可用于收集关于警告的信息（行号、警告原因等）
      3. 可用于控制是否忽略警告
      4. 可控制是否显示自定义格式，或者转换为错误
   2. 命令行可以控制警告过滤器，在启动python解释器时用-W选项
   3. 警告会有一个默认的输出显示到sys.stderr
6. 从ZIP文件导入模块：
   1. 如果搜索路径存在一个包含python模块的zip文件，导入时会把ZIP文件当做目录处理，在里面搜索模块
   2. Python通过ZIP导入时，不会添加对应的.pyc文件，因此导入速度会相对慢一些
7. 新的导入钩子
   1. 跳过

## 模块内建函数

1. \_\_import\_\_()：
   1. 它是实际上导入模块的函数，import语句就是调用该函数完成工作的
   2. 语法为：

\_\_import\_\_(module\_name[,globals[,locals[,fromlist]]])

* 1. module\_name是要导入模块的名称
  2. globals是包含当前全局符号表的名字的字典
  3. locals是包含局部符号表的名字的字典
  4. fromlist是一个使用from-import语句所导入符号的列表

1. globals()和locals()：
   1. globals()返回调用者全局名称空间的字典
   2. locals()返回调用者局部名称空间的字典
   3. 在全局名称空间下，locals()和globals()返回相同
      1. 因为这是局部名称空间就是全局空间
2. reload()：
   1. reload(module)：可以重新导入一个已经导入的模块
   2. 模块必须是全部导入，而不是from-import
   3. 参数是模块本身，而不是模块名的字符串

## 包

1. 包是一个有层次的文件目录结构，定义了一个由模块和子包组成的python执行环境
2. 包用来帮助解决一些问题：
   1. 使命名空间有层次，有目录结构
   2. 能够把有联系的模块组织到一起
   3. 可以帮助有冲突的模块名称
3. 包也是用句点属性标识访问元素的。
4. 目录结构：
   1. 包目录需要加入\_\_init\_\_.py初始化模块，可以是空文件。
      1. 如果没有，会导致importWarning信息
   2. 可以一直沿着子包的树状结构导入：
      1. 比如form father.son.sub.sub2.sub3 import attr
5. 使用from-import导入包：
   1. 包支持from-import all语句：
      1. from package.module import \*
   2. 在\_\_init\_\_.py中加入\_\_all\_\_变量，改变量包含执行这样的语句时应该导入的模块的名字，它由一个模块名字字符串列表组成
6. 绝对导入：
   1. import语句总是绝对导入的
   2. 所有导入都必须通过Python路径来访问
   3. 否则导入子包会导致和真正的标准库模块发生冲突
   4. 也就是说，如果有father.son和son两个绝对路径，如果不用绝对导入，import son就会导致冲突
7. 相对导入：
   1. 相对导入只应用于from-imort语句
   2. 通过句点指示一个相对的导入操作，之后附加句点表示当前查找位置后一个级别
   3. 比如from ..sub2 import attr，就是在从上往下的第三个级别

## 模块的其他特性

1. 自动载入的模块：
   1. 当python解释器在标准模式下启动，会自动导入一些模块用于系统相关操作
   2. sys.modules变量是由当前完整并成功导入到解释器的模块组成的字典
      1. 键为模块名
      2. 值为模块位置
2. 阻止属性导入：
   1. 如果不想当前模块某个属性被别的模块导入，即属性私有，可在属性名称前面加\_
      1. 比如\_att
   2. 在显式地导入该属性或直接导入整个模块的时候失效
      1. import module.\_att时失效
      2. import module时失效
      3. 一般就用于from module import \*
3. 不区分大小的导入：
   1. Python会先不区分大小写匹配模块名
   2. 如果匹配不到会区分大小写再次匹配？
4. 源代码编码：
   1. Python的模块支持ascii外的其他编码，不过ascii是默认编码
   2. 可以在python文件头部指定编码，语法为：

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

* 1. 如果文件包含了非ascii字符串而没有在文件头部说明，会导致语法错误

1. 导入循环：
   1. 比如a模块导入b模块，b模块又导入a模块，这样会导致导入失败
   2. 解决的办法有两种：
      1. 把其中一个导入语句移到模块底部
      2. 把其中一个导入语句移到函数内部

## 相关模块

跳过

# 面向对象编程

## 介绍

跳过

## 面向对象编程

1. pyhton中，类和oop都不是日常编程必须的。Python不限定必须使用OOP
2. 常用术语：
   1. 抽象/实现：
      1. 抽象指对现实世界问题和实体的本质表现
      2. 抽象不仅包括行为特征模型和数据属性，还定义了数据的接口
      3. 对抽象的实现就是对数据和相关接口的现实化
   2. 封装/接口：
      1. 封装描述了对数据/信息进行隐藏的观念，对数据属性提供接口和访问函数
         1. 直接对数据访问而无视接口，违背了封装性
   3. 合成：
      1. 多个不同的类合成一个大的类，可以解决一些现实的问题
      2. 要么组件联合，即允许对子组件的访问
      3. 要么聚合，封装的组件仅能通过定义好的接口访问
   4. 派生/继承/继承结构：
      1. 派生描述了子类的创建，子类保留已有的类类型和数据和行为
      2. 允许修改和其他自定义操作，但都不会修改原类的定义
   5. 泛化/特化：
      1. 泛化表示所有子类与父类及祖先有一样的特点
      2. 特化描述所有子类自定义的特点
   6. 多态：
      1. 多态表明动态（运行时）绑定的存在，允许重载及运行时类型确定和验证
   7. 自省/反射：
      1. 对象可以在运行期通过反射取得自身信息
      2. dir()和type()内建函数都有自省功能

## 类

1. 类是一种数据结构，我们可以用它定义对象，把数据值和行为特性融合在一起。
2. 创建类：
   1. 语法：

class class\_name(father\_class=object):

“文本字符串”

类代码块

* 1. 如果一个类没有父类，则默认是object的子类
  2. 父类可以有一个或多个
  3. 类代码块声明与、类成员定义、数据属性、函数组成
  4. 类通常在一个模块的顶层定义，以便实例能在之后任何地方创建

1. 声明与定义：
   1. 对python来说，声明与定义类没什么区别，因为它们是同时进行的。
   2. Python不支持虚类或抽象类

## 类属性

1. 概述：
   1. 属性就是**属于另一个对象**的数据或函数元素，可以通过句点属性标识法来访问
   2. 像复数的实部和虚部叫数据属性，像列表和字典的方法叫函数属性
   3. 当访问一个属性时，它同时也是一个对象，拥有自己的属性，因此导致了属性链
2. 类的数据属性：
   1. 数据属性仅仅是所定义的类的变量，可以在类创建后被使用
   2. 要么是由类中的方法更新，要么是在主程序其他地方被更新
   3. 类似于java的static静态变量，与类绑定，不依赖于任何实例
   4. 静态成员仅仅用来追踪与类相关的值。大多数情况还是用实例属性
3. 方法：
   1. 任何类中定义的方法仅应用于类，无法脱离类调用
   2. Python中，没有实例，方法是不能调用的，这种限制叫做绑定
4. 决定类的属性：
   1. dir(cla)：以列表的形式输出类的属性
   2. cla.\_\_dirc\_\_：作为类中的属性，是一个字典
      1. 键为类中的属性名
      2. 值为属性名对应的数据值
5. 特殊的类属性：
   1. cla.\_\_name\_\_：类的名字，类型为字符串
   2. cla.\_\_doc\_\_：类的文档字符串
   3. cla.\_\_bases\_\_：类的所有父类，类型为元组
   4. cla.\_\_dict\_\_：类的属性，类型为字典
   5. cla.\_\_module\_\_：类的定义所在的模块
   6. cla.\_\_class\_\_：实例对应的类

## 实例

1. 如果说类是一种数据结构，那么实例则是变量
2. 初始化：通过调用类对象来创建实例
   1. Python的实例化，不需要new，直接用函数操作符即可
   2. 语法：

cla = MyClass()

* 1. 和函数调用非常相像，只不过返回的是类的实例
  2. 当定义一个类后，就相当于已经创建了一个新的类型

1. \_\_init\_\_()构造器方法：
   1. 调用类时（即创建实例），传进的任何参数都交给了\_\_inti\_\_()
   2. 和java不同，python可以给\_\_init\_\_()传self参数。
   3. \_\_init\_\_()是创建实例后调用的第一个方法，用于一些准备工作，即初始化
   4. \_\_init\_\_()是一个实例方法，什么都不返回
2. \_\_new\_\_()构造器方法：
   1. \_\_new\_\_()是一个静态方法，可以返回一个合法的实例
   2. 和\_\_init\_\_()一样，传入的参数是在类实例化是生成的
   3. \_\_new\_\_()调用父类的\_\_new\_\_()创建对象（向上代理）
   4. 主要用于实例化不可变对象，只有在class返回一个实例时，后面的\_\_init\_\_()才能被调用
3. \_\_del\_\_()析构器方法：
   1. Python具有垃圾回收机制，因此这个函数会知道对象的所有引用被销毁时才执行
   2. 主要用于实例释放前的一些后处理
   3. 一般不实现这个方法

## 实例属性

1. 实例仅拥有数据属性，这些值独立与其他实例或类。
   1. 当一个实例被释放后，它的属性也被清除
2. 实例化实例属性：
   1. Python能够在运行时创建实例属性
   2. 经常通过\_\_init\_\_()的默认参数对实例属性进行设置
3. 查看实例属性：
   1. dir()和\_\_dir\_\_既可以查看类属性，也可以查看实例属性
4. 内建类型属性：
   1. 内建类型，比如复数的image和real，可以对它们使用dir()，但无法用\_\_dict\_\_
5. 实例属性与类属性：
   1. 类是类属性的名字空间，实例是实例属性的名字空间
   2. 访问类属性：
      1. 可以通过--类.属性或者--实例.属性访问
      2. 类属性只能通过类引用修改
         1. 如果通过实例引用修改，实际是创建了一个实例属性
         2. 也就是说如果类属性和实例属性同名，实例属性在实例层面覆盖类属性
      3. 因此类属性如果是可变对象，则能够被实例更改
      4. 类属性的修改会影响所有实例

## 绑定和方法调用

1. 方法：
   1. 方法仅仅是类内部定义的函数，是类属性而不是实例属性。
   2. 当有一个实例时，方法才被认定绑定到实例。没有实例时，方法就是未绑定的
   3. 任何一个方法定义的第一个参数都是变量self，表示调用此方法的实例对象
      1. 方法声明上是必须手动加上self的，但是可以不用
      2. 如果方法没有用到self，可以考虑创建常规函数（非类函数）
      3. self相当于java的this
2. 调用绑定方法：
   1. 调用绑定方法可以不再传递self参数
   2. 调用非绑定方法必须传递self参数
3. 调用非绑定方法：
   1. 不经常用到，需要用到的场景是：
      1. 在\_\_init\_\_()方法中调用父类的\_\_init\_\_()方法，同时传入了self。避免了把父类\_\_init\_\_()代码复制粘贴过来。
      2. 类似于java的super()

## 静态方法和类方法

1. 静态方法：
   1. 语法：

def func():

xxxx

foo = staticmethod(func)

* 1. 和其他语言的静态方法类似，静态方法使类中的函数，不需要实例化即可调用

1. 类方法：
   1. 类方法需要类作为第一个参数，这个参数通过解释器传给方法
      1. 类参数不需要特别的命名，不过一般用cls做变量名
   2. 语法：

def func(cls):

xxxx

foo = classmethod(func)

1. 使用函数修饰符：
   1. 可以对类方法和静态方法使用装饰器：
   2. 静态方法装饰器@staticmethod
   3. 类方法装饰器@classmethod

## 组合

1. 一个类被定义后，目标就是要把它当成一个模块来使用
2. 组合就是让不同类混合并加入到其他类，来增加功能和代码重用性

## 子类和派生

1. 概念：
   1. 当类之间有显著的不同，并且较小类是较大类所需要的组件时，用组合更好
   2. 当想设计相同的类但有一些不同的功能时，派生更好
2. 创建子类：
   1. 创建子类语法和普通类区别不大，参数中跟着需要派生的父类
   2. 和java不同，python可以继承多个父类
   3. 语法为：

class SubClassName(Parent1[,Parent2,...]):

“文档注释”

子类代码块

## 继承

1. 概念：
   1. 继承描述了基类的属性如何遗传给派生类。
   2. 一个子类可以继承它的基类的任何属性，不管是数据属性还是方法
   3. 和java不同，子类会继承父类的\_\_init\_\_()方法
   4. 但需要注意，文档注释是唯一的，所以\_\_doc\_\_不会被继承
2. \_\_bases\_\_类属性：
   1. \_\_bases\_\_是一个包含父类集合的
   2. 这里指的是直接父类，不向上追溯
3. 覆盖：
   1. 如果子类定义了和父类名字+参数列表相同的方法，则覆盖父类方法
   2. 在子类实例化的情况，即使覆盖了亦可以调用父类的方法：
      1. 通过父类未绑定的方法，传入子类实例作为参数
   3. 如果在子类覆盖了\_\_init\_\_()方法，需要在子类方法中非绑定调用父类的\_\_init\_\_()方法，否则父类的\_\_init\_\_()方法不会被自动调用
   4. super(当前类,self/cls)：相当于返回父类的self
      1. 这样就可以直接通过绑定的方式调用父类方法
4. 从标准类型派生：
   1. 不可变类型：
      1. 借助不可变类型\_\_new\_\_(cls[,args1,...])，返回不可变类实例
5. 多重继承：
   1. Python允许子类继承多个基类
   2. 跳过

## 类、实例和其他对象的内建函数

1. issubclass(sub,sup)：
   1. 布尔判断sub是否是sup的子类或子孙类
   2. sup可以是一个元组，那么只要符合元组中任意一个元素则返回True
2. isinstance(obj1,obj2)：
   1. 布尔判断obj1是否是obj2类的一个实例，或者是obj2子类的一个实例
   2. obj2必须是类，否则返回TypeError
   3. obj2可以是一个元组，那么只要符合元组中任意一个元素则返回True
3. \*attr(obj,’attr1’[,attr2,...])：
   1. obj是一个对象，不限于类和实例
   2. attr是属性的字符串名字
   3. hasattr()：
      1. 布尔判断对象是否有某个属性
   4. getattr():
      1. 取得对象某个属性的值
      2. 如果取得对象不存在的属性，会引发AttributeError异常
   5. setattr()：
      1. 给对象的一个属性赋值
      2. 可以是加入一个新的属性，也可以是更改一个已存在的属性
   6. delattr()：
      1. 从对象中删除属性
4. dir(obj=None)：
   1. 作用在实例时，显示实例变量、实例所在类和基类定义的方法和类属性
   2. 作用在类时，显示类和所有基类\_\_dict\_\_中的内容
      1. 但不会显示定义在元类的类属性
   3. 作用在模块时，显示模块的\_\_dict\_\_de neirong
   4. 不带参数时，显示调用者的局部变量
5. super(type[,obj])：
   1. 目的是帮助找出相应的父类，然后方便地调用相关属性
   2. 如果希望父类被绑定，可以传入obj参数；否则父类不会被绑定
      1. obj必须是type类型的，或者type的子类
   3. super()实际上是一个工厂函数
6. vars(obj=None)：
   1. 与dir()类似，给定的对象参数都必须有\_\_dict\_\_属性
   2. 返回一个字典，包含了对象存储于\_\_dict\_\_中的属性和值
   3. 如果没有提供参数，它和locals()功能相同，包含本地名字空间的属性和值

## 用特殊方法定制类

1. 基本定制型：
   1. C.\_\_init\_\_(self[,args1,...])：构造器
   2. C.\_\_new\_\_(self[,args1,...])：构造器，通常用于设置不可变类的子类
   3. C.\_\_del\_\_(self)：解构器
   4. C.\_\_str\_\_(self)：可打印的字符输出；内建str()和print()
      1. 使print(C)能够打印出想要的语句，类似于JAVA的toString()
   5. C.\_\_repr\_\_(self)：运行时的字符串输出；内建repr()和’’操作符
      1. 使C能够打印出想要的语句
   6. C.\_\_unicode\_\_(self)：Unicode字符串输出；内建unicode()
   7. C.\_\_call\_\_(self,\*args)：可调用的实例
   8. C.\_\_nonzero\_\_(self)：为object定义False值；内建bool()
   9. C.\_\_len\_\_(self)：长度；内建len()
2. 对象（值）比较：
   1. C.\_\_cmp\_\_(self, obj) ：对象比较；内建cmp()
   2. C.\_\_lt\_\_(self, obj) and ：小于/小于或等于；对.<及<=操作符
   3. C.\_\_gt\_\_(self, obj) and ：大于/大于或等于；对.>及>=操作符
   4. C.\_\_eq\_\_(self, obj) and ：等于/不等于；对.==,!=及<>操作符
3. 属性：
   1. C.\_\_getattr\_\_(self, attr) ：获取属性；内建getattr()；仅属性没有找到时用
   2. C.\_\_setattr\_\_(self, attr, val) ：设置属性
   3. C.\_\_delattr\_\_(self, attr) ：删除属性
   4. C.\_\_getattribute\_\_(self, attr)：获取属性；内建
   5. C.\_\_get\_\_(self, attr) ： （描述符）获取属性
   6. C.\_\_set\_\_(self, attr, val) ：（描述符）设置属性
   7. C.\_\_delete\_\_(self, attr) ：（描述符）删除属性
4. 数值类型--二进制操作符：
   1. i代替\*号表示左结合，r代替\*号表示右结合
   2. 重载\_\_i\*\_\_()方法必须返回self
   3. C.\_\_\*add\_\_(self, obj) ：加；+操作符
   4. C.\_\_\*sub\_\_(self, obj) ：减；-操作符
   5. C.\_\_\*mul\_\_(self, obj) ：乘；\*操作符
   6. C.\_\_\*div\_\_(self, obj) ：除；/操作符
   7. C.\_\_\*truediv\_\_(self, obj)： True 除；/操作符
   8. C.\_\_\*floordiv\_\_(self, obj) ： Floor 除；//操作符
   9. C.\_\_\*mod\_\_(self, obj) ：取模/取余；%操作符
   10. C.\_\_\*divmod\_\_(self, obj) ：除和取模；内建divmod()
   11. C.\_\_\*pow\_\_(self, obj[, mod]) ：乘幂；
   12. C.\_\_\*lshift\_\_(self, obj) ：左移位；<<操作符
   13. C.\_\_\*rshift\_\_(self, obj) ：右移；>>操作符
   14. C.\_\_\*and\_\_(self, obj) ：按位与；&操作符
   15. C.\_\_\*or\_\_(self, obj) ：按位或；|操作符
   16. C.\_\_\*xor\_\_(self, obj) ：按位与或；^操作符
5. 一元操作符：
   1. C.\_\_neg\_\_(self) ：一元负
   2. C.\_\_pos\_\_(self)： 一元正
   3. C.\_\_abs\_\_(self)： 绝对值；内建abs()
   4. C.\_\_invert\_\_(self) ：按位求反；~操作符
6. 数值转换：
   1. C.\_\_complex\_\_(self, com)： 转为complex(复数;内建complex()
   2. C.\_\_int\_\_(self)： 转为int;内建int()
   3. C.\_\_long\_\_(self)： 转为long；内建long()
   4. C.\_\_float\_\_(self)： 转为float；内建float()
7. 基本表示法：
   1. C.\_\_oct\_\_(self) ：八进制表示；内建oct()
   2. C.\_\_hex\_\_(self)： 十六进制表示；内建hex()
8. 数值压缩：
   1. C.\_\_coerce\_\_(self, num)： 压缩成同样的数值类型；内建coerce()
   2. C.\_\_index\_\_(self)：在有必要时压缩可选的数值类型为整型
9. 序列类型：
   1. C.\_\_len\_\_(self) ：序列中项的数目
   2. C.\_\_getitem\_\_(self, ind)： 得到单个序列元素
   3. C.\_\_setitem\_\_(self, ind,val)： 设置单个序列元素
   4. C.\_\_delitem\_\_(self, ind)： 删除单个序列元素
   5. C.\_\_getslice\_\_(self, ind1,ind2)： 得到序列片断
   6. C.\_\_setslice\_\_(self, i1, i2,val)： 设置序列片断
   7. C.\_\_delslice\_\_(self, ind1,ind2) ：删除序列片断
   8. C.\_\_contains\_\_(self, val) ： 测试序列成员；内建in 关.字
   9. C.\_\_\*add\_\_(self,obj) ：串连；+操作符
   10. C.\_\_\*mul\_\_(self,obj) ：重复；\*操作符
   11. C.\_\_iter\_\_(self) ： 创建迭代类；内建iter()
10. 映射类型
    1. C.\_\_len\_\_(self) ：mapping 中的项的数目
    2. C.\_\_hash\_\_(self)： 散列(hash)函数值
    3. C.\_\_getitem\_\_(self,key)： 得到给定键(key)的值
    4. C.\_\_setitem\_\_(self,key,val)： 设置给定键(key)的值
    5. C.\_\_delitem\_\_(self,key)： 删除给定键(key)的值
    6. C.\_\_missing\_\_(self,key)： 给定键如果不存在字典中，则提供一个默认值
11. 相关代码实例跳过

## 私有化

1. 默认情况下，python中的属性都是public
2. 单下划线（\_）：
   1. 简单地模块级私有化，只需要在属性名前加一个\_
   2. 防止模块的属性被 from module import \* 加载
   3. 严格基于作用域，同样适用于函数
3. 双下划线(\_\_)：
   1. 在属性名前加\_\_，在运行时会被混淆，会在属性名前加上”\_类名”
      1. 比如xxx.\_\_attr会变为xxx.\_cla\_\_attr
   2. 防止了在祖先类或子孙类的同名冲突

## 授权

1. 包装：
   1. 包装就是对一个已存在的对象进行包装，不管是数据类型，还是一段代码，或是已存在的对象。增加新的，删除不要的，或者修改已存在的功能
2. 授权：
   1. 跳过

## 新式类的高级特性

1. 新式类的通用特性：
   1. 跳过
2. \_\_slots\_\_类属性：
   1. 字典会占据大量的内存。为内存考虑，可以使用\_\_slots\_\_代替\_\_dict\_\_
   2. 带\_\_slots\_\_属性的类定义不会存在\_\_dict\_\_（除非在\_\_slots\_\_增加\_\_dict\_\_）
   3. slots可以是列表、元组、字符串
   4. 它包含了类的实例的所有属性
3. 特殊方法\_\_getattribute\_\_()：
   1. 当属性被访问时，就会被调用
   2. 当\_\_getattribute\_\_()和\_\_getattr\_\_()同时存在时，只有明确调用\_\_getattr\_\_()或者\_\_getattribute\_\_()引发AttributeError异常时，\_\_getattr\_\_()才会被调用
   3. 要小心在\_\_getattribute\_\_()中再次调用\_\_getattribute\_\_()造成无穷递归
4. 描述符：
   1. 可以认为描述符是表示对象属性的一个代理，当需要属性时，可以通过句点属性标识法，也可以通过描述符访问属性
   2. 同时实现了\_\_get\_\_()和\_\_set\_\_()的类叫做数据描述符
   3. 优先级从高到底排序：
      1. 类属性
      2. 数据描述符
      3. 实例属性
      4. 非数据描述符
      5. 默认为\_\_getattr\_\_()
   4. 后面跳过
5. Metaclasses和\_\_metaclass\_\_:
   1. 元类是什么：
      1. 元类让程序员定义类是如何创建的，也就是赋予了如何创建类的控制权
      2. 可以想象元类是一个类中类，它的实例就是一个类
   2. 什么时候使用元类：
      1. 元类一般用于创建类，也就是定义类的时候，元类代码就被执行
      2. 在执行类定义的时候，解释器需要知道这个类的元类。
      3. 先查找类属性\_\_metaclass\_\_，如果存在，作为该类元类；如果不存在，向上查找父类的\_\_metaclass\_\_；如果没有父类，会从对象或类型中继承；如果还没有，则会检查全局变量的\_\_metaclass\_\_；如果还没有，这个类就是传统类，用types.ClassType作为此类的元类
      4. 如果定义了传统类，并设置为\_\_metaclass\_\_=type，其实是升级为新风格类
   3. 谁在用元类：
      1. 元类的最终使用者是程序员
   4. 元类何时被创建：
      1. 创建一个新风格的类或传统类一般直接用系统提供的元类的默认方式
   5. 实例代码跳过

## 相关模块和文档

1. UserList：提供一个列表对象的封装类
2. UserDict：提供一个字典对象的封装类
3. UserString：提供一个字符串对象的封装类；又包括一个MutableString子类
4. types：定义所有python对象的类型在标准Python解释器中的名字
5. operator：标准操作符的函数接口
   1. 导入后可以使用add(a,b)、sub(a,b)等函数

# 执行环境

## 可调用对象

1. 可调用对象即是可以通过函数操作符“()”来调用的对象
2. 函数：
   1. 内建函数（BIFS）：
      1. BIF使用C或C++写的，编译后放入python解释器，然后作为内建名字空间的一部分加载进系统
      2. 这些函数在\_builtin\_模块，并作为\_\_builtins\_\_模块导入解释器
      3. 内建函数属性：
         1. bif.\_\_doc\_\_：文档字符串或None
         2. bif.\_\_name\_\_：文档名字的字符串
         3. bif.\_\_self\_\_：设置为None
         4. bif.\_\_module\_\_：内建函数所属的模块或None
   2. 用户定义函数(UDF)：
      1. 通常是用python写的，定义在模块的最高级，会作为全局名字空间的一部分
      2. 属性跳过
      3. 通过lambda表达式创建函数的对象享有和UDF相同的属性
3. 方法：跳过
4. 类：跳过
5. 类的实例：跳过

## 代码对象

1. 如果要执行python代码，那么该代码必须先转成字节码，这就是代码对象
2. 但是代码对象不包含任何执行环境的信息，因此通过可调用物包装一个代码对象提供额外的信息
3. 代码对象可以作为函数或方法调用的一部分执行，也可以用exec语句或eval()函数执行

## 可执行的对象声明和内建函数

1. callable(obj)：
   1. 布尔判断一个对象是否可以通过函数操作符（()）来调用
2. compile(string,file,type)：
   1. 从type类型中创建代码对象，type可以是：
      1. ‘eval’：那么字符串就是可求值的字符串表达式，代码对象将和eval()一起用
      2. ‘single’：那么字符串就是单一可执行语句，代码对象将和exec一起用
      3. ‘exec’：那么字符粗就是可执行语句组，代码对象将和exec一起用
   2. file是代码存放的地方，通常设为””
   3. string代表了要编译的python代码：
      1. 可以是可求职的
   4. 允许程序员在运行时迅速生成代码对象
   5. compile()提供了一次性字节代码预编译，以后每次调用都不要再编译了
3. eval(obj,globals=globals(),locals=locals())：
   1. 对obj进行求值，obj是已编译为代码对象的表达式，或是一个字符串表达式
      1. globals必须是字典
      2. locals可以是任何映射对象
   2. 可以给出全局或局部命名空间
4. exec obj语句：
   1. 执行代码对象、文件对象、单一的python语句或者语句块
5. input(prompt=’’)：
   1. 参数代表了给用户的字符串提示
   2. 会对输入的字符串中的表达式求值，而不是原样
6. 代码示例：
   1. 跳过

## 执行其他程序

1. 导入：
   1. 当导入模块后，会先执行该模块，运行该模块最高级的代码
2. execfile(filename,globals=globals(),locals=locals())：
   1. 用于执行一个文件，但尽可以在现有的执行环境下执行，有自己的名字空间
   2. 不保证不会修改局部名字空间
3. 将模块作为脚本执行：
   1. 如shell中，通过Python xxx.py执行python文件

## 执行非python程序

1. 在python可执行非python的二进制可执行文件、其他的shell脚本的，要求的就是一个有效的执行环境
2. oss.system(cmd)：
   1. 执行命令cmd（字符串），等待程序结束，返回退出代码（windows始终返回0）
   2. 当执行命令的时候，python的运行是挂起的
   3. 保留了现有的标准输出，但通常不会和产生输出的命令一起使用
3. os.popen(cmd,mode=’r’,buffering=-1)：
   1. 执行命令cmd（字符串），返回一个类文件对象作为通信句柄
      1. 默认为读取模式和默认系统缓冲
   2. 建立了一个指向cmd程序的单向连接
4. os.fork()：
   1. 创建一个和父进程并行的子进程
   2. 返回两次：
      1. 一次给父进程，值为子进程的进程标识符
      2. 一次给子进程，返回值永远为0
   3. 可以进行父子继承分流，语句：

ret = os.fork()

if ret == 0:

子进程代码

else:

父进程代码

* 1. 普遍做法是让子进程做脏活，父进程耐心等待或继续执行稍后检查

1. os.wait()、os.exec\*()、subprocess模块、相关函数：
   1. 跳过

## 结束执行

1. sys.exit(status=0)：
   1. 当调用该函数时，引发SystemExit异常
   2. SystemExit是唯一不看做错误的异常，除非被try-except，否则不会被捕获和处理
2. sys.exitfunc()：
   1. 跳过
3. os.\_exit(status)：
   1. 只适用于特定的平台，比如基于Unix的，以及win32平台
   2. 根本不执行任何清理就立即退出python
4. os.kill(PID,signal)：
   1. 模拟unix函数来发送信号给进程
   2. 典型型号是SIGINT，SIGQUIT，SIGKILL等，使进程总结
5. 各种操作系统接口：
   1. 跳过

# 正则表达式

## 介绍

1. 正则表达式为高级文本模式匹配，为搜索\替代功能提供了基础。是一些由字符和特殊符号组成的字符串，描述了字符串的特征
2. Python通过标准库的re模块支持正则表达式
3. 搜索指的是在字符串字符串任意部分查找匹配的模式，通过search()实现
4. 匹配指的是判断一个字符串能否从起始处全部或部分匹配摸个模式，通过match9)实现
5. 最基本的正则表达式：
   1. 仅由一个字符串定义一种模式，该模式仅匹配这个字符串本身

## 正则表达式使用的特殊符号和字符

1. 表格：

|  |  |  |
| --- | --- | --- |
| 记号 | 说明 | 举例 |
| letter | 匹配字符串本身 | abc |
| re1|re2 | 匹配正则表达式re1或re2 | Abc|abc |
| . | 匹配任何字符（除换行符） | .bc |
| ^ | 匹配字符串的开始 | ^abc |
| $ | 匹配字符串的结尾 | abc$ |
| \* | 匹配前面的字符串出现0次或多次 | a\* |
| + | 匹配前面的字符串出现1次或多次 | a+ |
| ? | 匹配前面的字符串出现0次或1次 | a?c |
| {n} | 匹配前面的字符串出现n次 | a{n} |
| {m,n} | 匹配前面的字符串出现m到n次 | a{m,n} |
| [...] | 匹配字符串组里出现的任意一个字符 | [abcdefg] |
| [..x-y...] | 匹配从x-y出现的任一个字符 | [0-9],[A-Za-z] |
| [^..x-y..]或[^xxx] | 不匹配范围里出现的任意字符 | [^abcdefg],[^a-c] |
| {...} |  |  |
| \d | 匹配任何数组，和[0-9]一样 | \d |
| \D | 匹配任何非数字，和[^0-9]一样 | \D |
| \w | 匹配任何数字和字母，和[A-Za-z0-9]一样 | \w |
| \W | 匹配任何非数字字母，和[^A-Za-z0-9]一样 | \W |
| \s | 匹配任何空白符，和[\n\t\r\v\f]一样 | \s |
| \S | 匹配任何非空白符，和[^\n\t\r\v\f]一样 | \S |
| \b | 匹配单词边界 | \b |
| \B | 匹配非单词边界 | \B |
| \ | 按字面匹配特殊字符 | \.，\\，\\*，\? |
| \A | 匹配字符串的起始，和^一样 | \Aabc |
| \Z | 匹配字符串的结束，和$一样 | abc\Z |

1. 用管道符号（|）匹配多个正则表达式：
   1. 相当于或or
   2. 可匹配不只两个，比如[a-z]|[0-9]|[A-Z]，表示匹配所有数字字母
2. 匹配任意单个字符（.）：
   1. 可匹配除了换行符\n外的任意可打印和不可打印字符
   2. 比如a.c，匹配aac、a?c、a&c等
   3. 如果要匹配句点（.），则在前面加斜杠，即/.
3. 从字符串开头、结尾、单词边界开始匹配：
   1. 这些符号指定的是位置
   2. 要匹配^和$本身，则同样是在前面加斜杠
   3. ^abc匹配以abc开始的字符串，abc$匹配以abc结尾的字符串
   4. \babc\b仅匹配单词abc，\Babc匹配任意不以abc开头的单词
4. 创建字符类（[]）:
   1. [ab][cd][ef][gh]，可以是aceg、adfh、bdfh、bceg等
5. 指定范围（-）和否定（^）：
   1. [0-9]：所有数字，[a-c]：a或b或c
   2. [^aeiou]：非元音字符
6. 用圆括号（()）组建组：
   1. 用于对正则表达式进行分组或匹配子组：
   2. 比如\d+(\.\d\*)?表示浮点数，0.04、0.等
   3. 类似于数学表达式的括号的作用

## 正则表达式和python语言

1. re模块：核心函数和方法
   1. re模块常见正则表达式函数与方法:
      1. compile(pattern,flags=0)：对模式进行编译，返回regex对象
      2. match(pattern,string,flags=0)：用模式匹配字符串
         1. 从起始处匹配，如果起始处都不匹配就匹配失败
         2. 匹配成功返回对象，否则返None
      3. search(pattern,string,flags=0)：在字符串查找模式的第一次程序
         1. 不一定要从起始处匹配
         2. 匹配成功返回对象，否则返回None
      4. findall(pattern,string[,flags=0])：在字符串中查找素有与模式匹配的出现
         1. 返回匹配对象的列表
      5. finditer(pattern,string[,flags=0])：在字符串中查找素有与模式匹配的出现
         1. 返回迭代器；对每个匹配，迭代器返回一个匹配对象
      6. split(pattern,string,max=0)：以模式为分隔符把字符串分割为一个列表
         1. 最多分割max次，默认分割所有
      7. sub(pattern,repl,string,max=0)：把string中与模式匹配的部分全部替换为repl
         1. 最多替换max次，默认替换所有
      8. subn(pattern,repl,string,max=0)：几乎和sub一样
         1. 还返回一个替换次数
      9. group(num=0)：返回全部匹配对象或者根据要求返回某个特定子组
         1. match返回的对象再调用group可以输出对应字符串或None
      10. groups()：返回一个包含全部匹配子组的元组
   2. Re编译：
      1. 在模式匹配之前，正则表达式模式需要先通过compile()被编译成regex对象
      2. 尽管建议预编译，但它不是必须的。也就是传入的pattern既可以是regex对象，也可以是字符串
      3. 模块函数会对已编译对象进行缓存，可以容纳100个，对Unicode支持
2. 实例：
   1. 的

```python

m = re.match(‘(\w\w\w)-(\d\d\d)’,’abc-123’) #通过括号()进行分组，分别提取字母和数字部分

m.group() #所有匹配部分，返回’abc-123’

m.group(1) #匹配子组1，返回abc

m.group(2) #匹配子组2，返回123

m.groups() #所有匹配子组，返回(‘abc’,’123’)

```

1. Python原始字符串的用法：
   1. 原始字符串就是因为正则表达式而产生的，因为ascii字符和正则表达式之间的冲突。
      1. 比如\b在在ascii表示退格，在正则表达式表示单词边界
         1. 为了让RE编译器把它当成想要的单词边界，可以写成\\b
   2. 使用原始字符串就不用担心这个问题了。
   3. 原始字符串的语法就是在字符串前面加r，比如r”hello”

## 示例

1. 跳过

# Time模块

1. 导入time模块
   1. import time
2. time.time()：
   1. 以毫秒的方式返回当前时间戳，从1970-1-1到现在的毫秒数
3. time.ctime()：
   1. 以----星期 月 日 时:分:秒 年的方式返回当前时间
      1. 星期和月是三位英文缩写
4. Time.localtime()：
   1. 当前时间的structtime形式

time.struct\_time(tm\_year=年, tm\_mon=月, tm\_mday=日（相对月）, tm\_hour=时（24制）, tm\_min=分,tm\_sec=22秒, tm\_wday=一周的第几天（周日为0开始算起）, tm\_yday=一年的第几天, tm\_isdst=是否夏时令)

1. time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())：
   1. 以某种格式定义时间，arg1为格式，arg2为时间
   2. 年-月-日 时:分:秒，注意大小写