补充

- repr()函数
  - 返回对象的可打印表示形式
  - printable representation
  - 代码测试时常被使用（打印对象的原始“面貌”）
  - print(5)与print('5')有何区别？
  - repr(5)与repr('5')有何区别？
  - a='10'
  - b=eval(a) #b=10
  - b=eval(repr(a))#b='10'
  - b == a ??

补充

- 字典推导（dictionary comprehension）
  - `even_dict={x : x**2 for x in range(11) if x % 2 ==0}`
- 集合推导（set comprehension）
  - `odd_set={x**2 for x in range(11) if x % 2 ==1}`

补充

- 字典更新时可使用get()方法
  - freq={}#统计词频
  - if term in freq:
    - freq[term]+=1
  - else:
    - freq[term]=1
  - #更简单的方案
  - count=freq.get(term,0)
  - freq[term]=count+1

School of Economics ang Management, Beihang University

现代程序设计技术

赵吉昌

本周内容

- Python基础
  - 控制流
  - 函数

控制流

- 条件控制
  - if condition_1:
  - statement_block_1
  - elif condition_2:
  - statement_block_2
  - else:
  - statement_block_3

控制流

- 条件控制
  - 每个条件后面要使用冒号：，表示接下来是满足条件后要执行的语句块
  - 使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块
  - 没有`switch - case`语句
  - 如果只有一条语句，可以写在一行
    - `if test<10: print(test)`
  - 在嵌套 if 语句中，可以
    - `if...elif...else` 结构放在另外一个 `if...elif...else` 结构中

控制流

- while循环
  - n = 100
  - sum = 0
  - counter = 1
  - while counter <= n:
  - sum = sum + counter
  - counter += 1

控制流

- while循环
  - 没有 do...while 循环
  - 无限循环
    - while True:
    - 使用 CTRL+C 可退出当前的无限循环
  - while 循环使用 else 语句
    - while … else 在条件语句为 False 时执行 else 的语句块

控制流

- for循环
  - for <variable> in <sequence>:
- <statements>
  - else:
- <statements>
  - 经常与range()函数配合使用
    - range()函数的使用要熟练(range class)
    - range(start, stop, step)
    - range(10)
    - range(1,11)
    - range(1,10,2)

控制流

- break和continue
  - break 语句可以跳出 for 和 while 的循环体
    - 对应循环的else 块不执行
  - continue语句跳过当前循环块中的剩余语句，然后继续进行下一轮循环

补充

- while和for中的else
  - 当循环条件为假时执行
  - 在某些情况下可能有必要
    - 如需要知道迭代变量在循环结束时的值
  - 如果循环没有从头到尾执行（如通过break提前终止），则else块不会执行
  - 不要在for与while后面写else块

函数

- 函数定义
  - 以 `def` 关键词开头，后接函数标识符名称和圆括号 `()`
    - 圆括号之间可以用于定义参数
  - 函数内容以冒号起始，并且换行缩进
  - 函数第一行语句应用文档字符串进行函数说明
    - 第一行关于函数用途的简介，这一行应该以大写字母开头，以句号结尾
    - 如果文档字符串有多行，第二行应该空白，与其后的详细描述明确分隔
    - 详细描述应有一或多段以描述对象的调用约定、边界效应等

函数

- 文档字符串
  - def fdoc():
    - '''
    - 这是一个函数，用来演示函数文档的说明
    - 这里是详细的功能的说明，调用的说明，边界的约定
    - '''
    - pass
  - print(fdoc.__doc__)

函数

- 参数传递
  - 不可变类型传值，如数、字符串和元组
    - 如fun(a)，传递的只是a的值，没有影响a对象本身，在 fun(a)内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身

```
def fa(a):
    a=100
    print(hex(id(a)))
    print(hex(id(100)))
a=10
print(hex(id(a)))
print(hex(id(10)))
fa(a)
print(hex(id(a)))
print(hex(id(10)))
```

函数

- 参数传递
  - 可变类型传引用，如列表，字典，集合
    - 如 fun(L)修改后fun外部的L也会受影响

```
def fc(a):
    a.append(100)
    print(hex(id(a)))
l=[1,2,3]
print(l)
print(hex(id(l)))
fc(l)
print(l)
```

函数

- 参数类型
  - 必需参数
    - 位置参数
  - 关键字参数
  - 默认参数
  - 不定长参数
    - 可变参数

函数

- 必需参数
  - 必须以正确的顺序传入函数
  - 调用时数量必须和声明时一样

函数

- 关键字参数
  - 使用关键字参数来确定传入的参数值
  - 关键字的参数应跟随在位置参数后
  - 允许函数调用时参数的顺序与声明时不一致
    - 解释器能够根据参数名匹配参数值

  - `def fun(name,key):`
    - `pass`
  - `fun(key='lambda x:x[1]',name='test')`

函数

- 默认参数
  - 调用函数时，如果没有传递参数，则会使用默认值
    - `def fun(name='zjc',key):`
      - `pass`
    - `fun(key='lambda x:x[1]')`
  - 默认值只被赋值一次，这使得当默认值是可变对象时会有所不同，比如列表、字典或者大多数类的实例，也即默认值在后续调用中会累积
    - 联系到C语言的静态变量

函数

- 默认参数
  - def f(a, L=[]):
  - print(hex(id(L)))
  - L.append(a)
  - return L
  - print(f(1))
  - print(f(2))
  - print(f(3))
  - 如何避免累计？
  - def f(a, L=None):
  - if L is None:
  - L = [] #每次调用重新进行初始化
  - print(hex(id(L)))
  - L.append(a)
  - return L

函数

- 不定长参数
  - 需要一个函数能够处理比声明时更多的参数
  - 且声明时不需要命名
  - 加*的参数会以元组(tuple)的形式导入，存放所有未命名的参数变量

```
def ptest( arg1, *vartuple ):
    print (arg1,end=' , ')
    print (vartuple)
ptest( 70, 'test', 50 )
```

函数

- 不定长参数
  - 加**的参数会以字典的形式导入
  - def ptest2(arg1, **vardict):
  - print (arg1)
  - print (vardict)
  - ptest2(10,a=2,b=4)
    - 会输出什么结果？

函数

- 不定长参数
  - 通常这些可变参数是参数列表中的最后一个
  - 任何出现在不定长参数的后面的参数只能是关键字参数，不能是位置有关参数
  - `def concat(*args, sep="/"):`
- `    pass`
  - 补充
    - 新版本中引入了/等来对位置参数和关键字参数进行指定

函数

- 参数
  - 声明函数时，参数中星号 * 可以单独出现
  - * 后的参数必须用关键字传入

```
def f(a,b,*,c):
    return a+b+c
f(1,2,3)#错误，c必须通过关键词传入
f(1,2,c=3)
```

函数

- 参数列表的分拆
  - 要传递的参数已经是一个数据结构如列表等，但要调用的函数却只接受分成一个一个的参数值

```
- args = [3, 6]
- list(range(*args)) #列表分拆(*)

- d={'name':'zjc',’age’:36,'job':'prof.'}
- def printInfo(name,age,job):
-     print("Name:{0}\tAge:{1}\tJob:{2}".format\
•           (name,age,job))
-     pass
- printInfo(**d) #字典分拆(**)
```

函数

- 匿名函数
  - 不再使用def语句标准的形式定义函数
  - 使用lambda来创建匿名函数
    - lambda只是一个表达式，函数体比def简单很多
    - lambda的主体是一个表达式，而不是一个代码块
    - 仅仅能在lambda表达式中封装有限的逻辑
      - 简单形式下只能使用内部变量
    - 普通函数定义中的一个语法技巧
  - lambda [arg1 [,arg2,.....argn]]:expression
  - sum = lambda arg1, arg2: arg1 + arg2
  - sum(1, 2)

函数

- return语句
  - return [表达式]
  - 用于退出函数，选择性地向调用方返回一个表达式
  - 不带参数的return语句返回None
  - 没有return语句自动返回None
  - 返回多个值时可以通过“拆箱”来接收不需要捕获的值
    - 但一般建议不要把返回值分拆至超过三个以上的变量中
    - Demo: rem.py

函数

- 嵌套函数
  - def func():
  - [statements]
  - def func_inner():
  - [statements]
- lambda 如何从外部作用域引用变量？
  - def make_incrementor(n):
  - return lambda x: x + n
  - f=make_incrementor(1)
  - f(0)
    - make_incrementor(1)(0)

函数

- 闭包
  - 在一个外函数中定义了一个内函数，内函数里运用了外函数的变量，并且外函数的返回值是对内函数的引用
  - 闭包变量实际上只有一份，每次开启内函数时都在使用同一份闭包变量
    - “惰性”求值
    - “懒”加载
  - 装饰器
    - 后面会专门讲

函数

- 闭包
  - def outer(x):
  - b=[x] #python 2.x
  - def inner(y):
  - nonlocal x #python 3.x
  - x+=y
  - b[0]+=y
  - return x #注意返回
  - return inner #注意返回

  - f1=outer(10)
  - print(f1(1)) #11
  - print(f1(2)) #?
  - print(outer(10)(1))
  - print(outer(10)(2))

函数

- 函数注解(不建议滥用)
  - Function annotations, both for parameters and return values, are *completely* optional
  - 函数注解以字典形式存储在函数的 __annotations__ 属性
  - 参数注解（Parameter annotations）定义在参数名称的冒号后，紧随着一个用来表示注解的值表达式
  - 返回注释（Return annotations）是定义在一个 -> 后面，紧随着一个表达式，在冒号与 -> 之间
    - `def f(ham: 42, eggs: str = 'spam') -> "Nothing to see here":`

函数

- 编程风格
  - PEP(Python Enhancement Proposals)8: https://www.python.org/dev/peps/pep-0008/
  - 使用 4 空格缩进，而非 TAB
    - 在小缩进（可以嵌套更深）和大缩进（更易读）之间，4空格是一个很好的折中。TAB 引发了一些混乱，最好弃用
  - 折行以确保其不会超过 79 个字符
    - 这有助于小显示器用户阅读，也可以让大显示器能并排显示几个代码文件

函数

- 编程风格
  - 使用空行分隔函数以及函数中的大块代码
  - 可能的话，注释独占一行
  - 使用文档字符串
  - 把空格放到操作符两边，以及逗号后面，但是括号里侧不加空格
    - `a = f(1, 2) + g(3, 4)`
  - 统一函数和类命名
    - 类名用驼峰命名
    - 函数和方法名用小写和下划线(unix式)
  - 不要使用花哨的编码

本周作业

- 情绪理解是文本处理里最常见任务之一。现提供一个五类情绪字典（由情绪词组成，5个文件，人工标注），实现一个情绪分析工具，并利用该工具对10000条新浪微博进行测试和分析（一行一条微博）。微博数据见课程资料中提供的weibo.txt（200万条，包括地理位置，文本和发布时间），字典数据见公开数据中的emotion lexicon（https://doi.org/10.6084/m9.figshare.12163569.v2）。请按要求用函数进行功能封装，并在main中调用测试，鼓励尝试不同方式的可视化。
  - 1. 实现一个函数，对微博数据进行清洗，去除噪声（如url等），过滤停用词。注意分词的时候应该将情绪词典加入Jieba或pyltp的自定义词典，以提高这些情绪词的识别能力。
  - 2. 实现两个函数，实现一条微博的情绪分析，返其情绪向量或情绪值。目前有两种方法，一是认为一条微博的情绪是混合的，即一共有n个情绪词，如果joy有n1个，则joy的比例是n1/n；二是认为一条微博的情绪是唯一的，即n个情绪词里，anger的情绪词最多，则该微博的情绪应该为angry。注意，这里要求用闭包实现，尤其是要利用闭包实现一次加载情绪词典且局部变量持久化的特点。同时，也要注意考虑一些特别的情况，如无情绪词出现，不同情绪的情绪词出现数目一样等，并予以处理（如定义为无情绪，便于在后面的分析中去除）。
  - 3. 微博中包含时间，可以讨论不同时间情绪比例的变化趋势，实现一个函数，可以通过参数来控制并返回对应情绪的时间模式，如joy的小时模式，sadness的周模式等。
  - 4. 微博中包含空间，可以讨论情绪的空间分布，实现一个函数，可以通过参数来控制并返回对应情绪的空间分布，即围绕某个中心点，随着半径增加该情绪所占比例的变化，中心点可默认值可以是城市的中心位置。
  - 5. （附加）讨论字典方法进行情绪理解的优缺点，有无可能进一步扩充字典来提高情绪识别的准确率？如何扩充，有无自动或半自动的扩充思路？
  - 6. （附加）可否对情绪的时间和空间分布进行可视化？（如通过matplotlib绘制曲线，或者用pyecharts(注意版本的兼容性)在地图上标注不同的情绪）
  - 7. （附加）思考情绪时空模式的管理意义，如营销等。
  - 注意：如果规模太大处理不了，可以酌情处理一部分。