## 指定编码

# \_\*\_ coding: utf-8 \_\*\_

## 描述

"""类型和运算----类型和运算----类型和运算----类型和运算----类型和运算----类型和运算----类型和运算----类型和运算----类型和运算----类型和运算----类型和运算"""

## 寻求帮助

dir(obj) # 简单的列出对象obj所包含的方法名称，返回一个字符串列表

help(obj.func) # 查询obj.func的具体介绍和用法

## 测试类型的三种方法，推荐第三种

if type(L) == type([]): print("L is list")

if type(L) == list: print("L is list")

if isinstance(L, list): print("L is list")

## Python数据类型：哈希类型、不可哈希类型

# 哈希类型，即在原地不能改变的变量类型，不可变类型。可利用hash函数查看其hash值，也可以作为字典的key

"数字类型：int, float, decimal.Decimal, fractions.Fraction, complex"

"字符串类型：str, bytes"

"元组：tuple"

"冻结集合：frozenset"

"布尔类型：True, False"

"None"

# 不可hash类型：原地可变类型：list、dict和set。它们不可以作为字典的key。

## 数字常量

1234, -1234, 0, 999999999 # 整数

1.23, 1., 3.14e-10, 4E210, 4.0e+210 # 浮点数

0o177, 0x9ff, 0X9FF, 0b101010 # 八进制、十六进制、二进制数字

3+4j, 3.0+4.0j, 3J # 复数常量，也可以用complex(real, image)来创建

hex(I), oct(I), bin(I) # 将十进制数转化为十六进制、八进制、二进制表示的“字符串”

int(string, base) # 将字符串转化为整数，base为进制数

# 2.x中，有两种整数类型：一般整数（32位）和长整数（无穷精度）。可以用l或L结尾，迫使一般整数成为长整数

float('inf'), float('-inf'), float('nan') # 无穷大, 无穷小, 非数

## 数字的表达式操作符

yield x # 生成器函数发送协议

lambda args: expression # 生成匿名函数

x if y else z # 三元选择表达式

x and y, x or y, not x # 逻辑与、逻辑或、逻辑非

x in y, x not in y # 成员对象测试

x is y, x is not y # 对象实体测试

x<y, x<=y, x>y, x>=y, x==y, x!=y # 大小比较，集合子集或超集值相等性操作符

1 < a < 3 # Python中允许连续比较

x|y, x&y, x^y # 位或、位与、位异或

x<<y, x>>y # 位操作：x左移、右移y位

+, -, \*, /, //, %, \*\* # 真除法、floor除法：返回不大于真除法结果的整数值、取余、幂运算

-x, +x, ~x # 一元减法、识别、按位求补（取反）

x[i], x[i:j:k] # 索引、分片、调用

int(3.14), float(3) # 强制类型转换

## 整数可以利用bit\_length函数测试所占的位数

a = 1; a.bit\_length() # 1

a = 1024; a.bit\_length() # 11

## repr和str显示格式的区别

"""

repr格式：默认的交互模式回显，产生的结果看起来它们就像是代码。

str格式：打印语句，转化成一种对用户更加友好的格式。

"""

## 数字相关的模块

# math模块

# Decimal模块：小数模块

import decimal

from decimal import Decimal

Decimal("0.01") + Decimal("0.02") # 返回Decimal("0.03")

decimal.getcontext().prec = 4 # 设置全局精度为4 即小数点后边4位

# Fraction模块：分数模块

from fractions import Fraction

x = Fraction(4, 6) # 分数类型 4/6

x = Fraction("0.25") # 分数类型 1/4 接收字符串类型的参数

## 集合set

"""

set是一个无序不重复元素集, 基本功能包括关系测试和消除重复元素。

set支持union(联合), intersection(交), difference(差)和sysmmetric difference(对称差集)等数学运算。

set支持x in set, len(set), for x in set。

set不记录元素位置或者插入点, 因此不支持indexing, slicing, 或其它类序列的操作

"""

s = set([3,5,9,10]) # 创建一个数值集合，返回{3, 5, 9, 10}

t = set("Hello") # 创建一个唯一字符的集合返回{}

a = t | s; t.union(s) # t 和 s的并集

b = t & s; t.intersection(s) # t 和 s的交集

c = t - s; t.difference(s) # 求差集（项在t中, 但不在s中）

d = t ^ s; t.symmetric\_difference(s) # 对称差集（项在t或s中, 但不会同时出现在二者中）

t.add('x'); t.remove('H') # 增加/删除一个item

t.update([10,37,42]) # 利用[......]更新s集合

x in s, x not in s # 集合中是否存在某个值

s.issubset(t); s.issuperset(t); s.copy(); s.discard(x); s.clear()

{x\*\*2 for x in [1, 2, 3, 4]} # 集合解析，结果：{16, 1, 4, 9}

{x for x in 'spam'} # 集合解析，结果：{'a', 'p', 's', 'm'}

## 集合frozenset，不可变对象

"""

set是可变对象，即不存在hash值，不能作为字典的键值。同样的还有list、tuple等.

frozenset是不可变对象，即存在hash值，可作为字典的键值

frozenset对象没有add、remove等方法，但有union/intersection/difference等方法

"""

a = set([1, 2, 3])

b = set()

b.add(a) # error: set是不可哈希类型

b.add(frozenset(a)) # ok，将set变为frozenset，可哈希

## 布尔类型bool

type(True) # 返回<class 'bool'>

isinstance(False, int) # bool类型属于整形，所以返回True

True == 1; True is 1 # 输出(True, False)

## 动态类型简介

"""

变量名通过引用，指向对象。

Python中的“类型”属于对象，而不是变量，每个对象都包含有头部信息，比如"类型标示符" "引用计数器"等

"""

#共享引用及在原处修改：对于可变对象，要注意尽量不要共享引用！

#共享引用和相等测试：

L = [1], M = [1], L is M # 返回False

L = M = [1, 2, 3], L is M # 返回True，共享引用

#增强赋值和共享引用：普通+号会生成新的对象，而增强赋值+=会在原处修改

L = M = [1, 2]

L = L + [3, 4] # L = [1, 2, 3, 4], M = [1, 2]

L += [3, 4] # L = [1, 2, 3, 4], M = [1, 2, 3, 4]

#-- 常见字符串常量和表达式

S = '' # 空字符串

S = "spam’s" # 双引号和单引号相同

S = "s\np\ta\x00m" # 转义字符

S = """spam""" # 三重引号字符串，一般用于函数说明

S = r'\temp' # Raw字符串，不会进行转义，抑制转义

S = b'Spam' # Python3中的字节字符串

S = u'spam' # Python2.6中的Unicode字符串

s1+s2, s1\*3, s[i], s[i:j], len(s) # 字符串操作

'a %s parrot' % 'kind' # 字符串格式化表达式

'a {0} parrot'.format('kind') # 字符串格式化方法

for x in s: print(x) # 字符串迭代，成员关系

[x\*2 for x in s] # 字符串列表解析

','.join(['a', 'b', 'c']) # 字符串输出，结果：a,b,c

#-- 内置str处理函数：

str1 = "stringobject"

str1.upper(); str1.lower(); str1.swapcase(); str1.capitalize(); str1.title() # 全部大写，全部小写、大小写转换，首字母大写，每个单词的首字母都大写

str1.ljust(width) # 获取固定长度，左对齐，右边不够用空格补齐

str1.rjust(width) # 获取固定长度，右对齐，左边不够用空格补齐

str1.center(width) # 获取固定长度，中间对齐，两边不够用空格补齐

str1.zfill(width) # 获取固定长度，右对齐，左边不足用0补齐

str1.find('t',start,end) # 查找字符串，可以指定起始及结束位置搜索

str1.rfind('t') # 从右边开始查找字符串

str1.count('t') # 查找字符串出现的次数

#上面所有方法都可用index代替，不同的是使用index查找不到会抛异常，而find返回-1

str1.replace('old','new') # 替换函数，替换old为new，参数中可以指定maxReplaceTimes，即替换指定次数的old为new

str1.strip(); str1.lstrip(); str1.rstrip(); str1.strip('d'); str1.lstrip('d'); str1.rstrip('d')

str1.startswith('start') # 是否以start开头

str1.endswith('end') # 是否以end结尾

str1.isalnum(); str1.isalpha(); str1.isdigit(); str1.islower(); str1.isupper() # 判断字符串是否全为字符、数字、大写、小写

#-- 三重引号编写多行字符串块，并且在代码折行处嵌入换行字符\n

mantra = """hello world

hello python

hello my friend"""

# mantra为"""hello world \n hello python \n hello my friend"""

#-- 索引和分片：

S[0], S[len(S)–1], S[-1] # 索引

S[1:3], S[1:], S[:-1], S[1:10:2] # 分片，第三个参数指定步长

#-- 字符串转换工具：

int('42'), str(42) # 返回(42, '42')

float('4.13'), str(4.13) # 返回(4.13, '4.13')

ord('s'), chr(115) # 返回(115, 's')

int('1001', 2) # 将字符串作为二进制数字，转化为数字，返回9

bin(13), oct(13), hex(13) # 将整数转化为二进制/八进制/十六进制字符串，返回('1001', '0o15', '0xd')

#-- 另类字符串连接

name = "wang" "hong" # 单行，name = "wanghong"

name = "wang" \

"hong" # 多行，name = "wanghong"

#-- Python中的字符串格式化实现1--字符串格式化表达式

"""

基于C语言的'print'模型，并且在大多数的现有的语言中使用。

通用结构：%[(name)][flag][width].[precision]typecode

"""

"this is %d %s bird" % (1, 'dead') # 一般的格式化表达式

"%s---%s---%s" % (42, 3.14, [1, 2, 3]) # 字符串输出：'42---3.14---[1, 2, 3]'

"%d...%6d...%-6d...%06d" % (1234, 1234, 1234, 1234) # 对齐方式及填充："1234... 1234...1234 ...001234"

x = 1.23456789

"%e | %f | %g" % (x, x, x) # 对齐方式："1.234568e+00 | 1.234568 | 1.23457"

"%6.2f\*%-6.2f\*%06.2f\*%+6.2f" % (x, x, x, x) # 对齐方式：' 1.23\*1.23 \*001.23\* +1.23'

"%(name1)d---%(name2)s" % {"name1":23, "name2":"value2"} # 基于字典的格式化表达式

"%(name)s is %(age)d" % vars() # vars()函数调用返回一个字典，包含了所有本函数调用时存在的变量

#-- Python中的字符串格式化实现2--字符串格式化调用方法

# 普通调用

"{0}, {1} and {2}".format('spam', 'ham', 'eggs') # 基于位置的调用

"{motto} and {pork}".format(motto = 'spam', pork = 'ham') # 基于Key的调用

"{motto} and {0}".format('ham', motto = 'spam') # 混合调用

# 添加键 属性 偏移量 (import sys)

"my {1[spam]} runs {0.platform}".format(sys, {'spam':'laptop'}) # 基于位置的键和属性

"{config[spam]} {sys.platform}".format(sys = sys, config = {'spam':'laptop'}) # 基于Key的键和属性

"first = {0[0]}, second = {0[1]}".format(['A', 'B', 'C']) # 基于位置的偏移量

# 具体格式化

"{0:e}, {1:.3e}, {2:g}".format(3.14159, 3.14159, 3.14159) # 输出'3.141590e+00, 3.142e+00, 3.14159'

"{fieldname:format\_spec}".format(......)

# 说明:

"""

fieldname是指定参数的一个数字或关键字, 后边可跟可选的".name"或"[index]"成分引用

format\_spec ::= [[fill]align][sign][#][0][width][,][.precision][type]

fill ::= <any character> #填充字符

align ::= "<" | ">" | "=" | "^" #对齐方式

sign ::= "+" | "-" | " " #符号说明

width ::= integer #字符串宽度

precision ::= integer #浮点数精度

type ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

"""

# 例子:

'={0:10} = {1:10}'.format('spam', 123.456) # 输出'=spam = 123.456'

'={0:>10}='.format('test') # 输出'= test='

'={0:<10}='.format('test') # 输出'=test ='

'={0:^10}='.format('test') # 输出'= test ='

'{0:X}, {1:o}, {2:b}'.format(255, 255, 255) # 输出'FF, 377, 11111111'

'My name is {0:{1}}.'.format('Fred', 8) # 输出'My name is Fred .' 动态指定参数

#-- 常用列表常量和操作

L = [[1, 2], 'string', {}] # 嵌套列表

L = list('spam') # 列表初始化

L = list(range(0, 4)) # 列表初始化

list(map(ord, 'spam')) # 列表解析

len(L) # 求列表长度

L.count(value) # 求列表中某个值的个数

L.append(obj) # 向列表的尾部添加数据，比如append(2)，添加元素2

L.insert(index, obj) # 向列表的指定index位置添加数据，index及其之后的数据后移

L.extend(interable) # 通过添加iterable中的元素来扩展列表，比如extend([2])，添加元素2，注意和append的区别

L.index(value, [start, [stop]]) # 返回列表中值value的第一个索引

L.pop([index]) # 删除并返回index处的元素，默认为删除并返回最后一个元素

L.remove(value) # 删除列表中的value值，只删除第一次出现的value的值

L.reverse() # 反转列表

L.sort(cmp=None, key=None, reverse=False) # 排序列表

a = [1, 2, 3], b = a[10:] # 注意，这里不会引发IndexError异常，只会返回一个空的列表[]

a = [], a += [1] # 这里实在原有列表的基础上进行操作，即列表的id没有改变

a = [], a = a + [1] # 这里最后的a要构建一个新的列表，即a的id发生了变化

#-- 用切片来删除序列的某一段

a = [1, 2, 3, 4, 5, 6, 7]

a[1:4] = [] # a = [1, 5, 6, 7]

a = [0, 1, 2, 3, 4, 5, 6, 7]

del a[::2] # 去除偶数项(偶数索引的)，a = [1, 3, 5, 7]

#-- 常用字典常量和操作

D = {}

D = {'spam':2, 'tol':{'ham':1}} # 嵌套字典

D = dict.fromkeys(['s', 'd'], 8) # {'d': 8, 's': 8}

D = dict(name = 'tom', age = 12) # {'age': 12, 'name': 'tom'}

D = dict([('name', 'tom'), ('age', 12)]) # {'age': 12, 'name': 'tom'}

D = dict(zip(['name', 'age'], ['tom', 12])) # {'age': 12, 'name': 'tom'}

D.keys(); D.values(); D.items() # 字典键、值以及键值对

D.get(key, default) # get函数

D.update(D\_other) # 合并字典，如果存在相同的键值，D\_other的数据会覆盖掉D的数据

D.pop(key, [D]) # 删除字典中键值为key的项，返回键值为key的值，如果不存在，返回默认值D，否则异常

D.popitem() # pop字典中的一项（一个键值对）

D.setdefault(k[, d]) # 设置D中某一项的默认值。如果k存在，则返回D[k]，否则设置D[k]=d，同时返回D[k]。

del D # 删除字典

del D['key'] # 删除字典的某一项

if key in D: if key not in D: # 测试字典键是否存在

# 字典注意事项：（1）对新索引赋值会添加一项（2）字典键不一定非得是字符串，也可以为任何的不可变对象

#-- 字典解析

D = {k:8 for k in ['s', 'd']} # {'d': 8, 's': 8}

D = {k:v for (k, v) in zip(['name', 'age'], ['tom', 12])}

#-- 字典的特殊方法\_\_missing\_\_：当查找找不到key时，会执行该方法

class Dict(dict):

def \_\_missing\_\_(self, key):

self[key] = []

return self[key]

dct = Dict()

dct["foo"].append(1) # 这有点类似于collections.defalutdict

dct["foo"] # [1]

#-- 元组和列表的唯一区别在于元组是不可变对象，列表时可变对象

a = [1, 2, 3] # a[1] = 0, OK

a = (1, 2, 3) # a[1] = 0, Error

a = ([1, 2]) # a[0][1] = 0, OK

a = [(1, 2)] # a[0][1] = 0, Error

#-- 元组的特殊语法: 逗号和圆括号

D = (12) # 此时D为一个整数 即D = 12

D = (12, ) # 此时D为一个元组 即D = (12, )

#-- 文件基本操作

output = open(r'C:\spam', 'w') # 打开输出文件，用于写

input = open('data', 'r') # 打开输入文件，用于读。打开的方式可以为'w', 'r', 'a', 'wb', 'rb', 'ab'等

fp.read([size]) # size为读取的长度，以byte为单位

fp.readline([size]) # 读一行，如果定义了size，有可能返回的只是一行的一部分

fp.readlines([size]) # 把文件每一行作为一个list的一个成员，并返回这个list。其实它的内部是通过循环调用readline()来实现的。如果提供size参数，size是表示读取内容的总长。

fp.readable() # 是否可读

fp.write(str) # 把str写到文件中，write()并不会在str后加上一个换行符

fp.writelines(seq) # 把seq的内容全部写到文件中(多行一次性写入)

fp.writeable() # 是否可写

fp.close() # 关闭文件。

fp.flush() # 把缓冲区的内容写入硬盘

fp.fileno() # 返回一个长整型的”文件标签“

fp.isatty() # 文件是否是一个终端设备文件（unix系统中的）

fp.tell() # 返回文件操作标记的当前位置，以文件的开头为原点

fp.next() # 返回下一行，并将文件操作标记位移到下一行。把一个file用于for … in file这样的语句时，就是调用next()函数来实现遍历的。

fp.seek(offset[,whence]) # 将文件打操作标记移到offset的位置。whence可以为0表示从头开始计算，1表示以当前位置为原点计算。2表示以文件末尾为原点进行计算。

fp.seekable() # 是否可以seek

fp.truncate([size]) # 把文件裁成规定的大小，默认的是裁到当前文件操作标记的位置。

for line in open('data'):

print(line) # 使用for语句，比较适用于打开比较大的文件

open('f.txt', encoding = 'latin-1') # Python3.x Unicode文本文件

open('f.bin', 'rb') # Python3.x 二进制bytes文件

# 文件对象还有相应的属性：buffer closed encoding errors line\_buffering name newlines等

#-- 其他

# Python中的真假值含义：1. 数字如果非零，则为真，0为假。 2. 其他对象如果非空，则为真

# 通常意义下的类型分类：1. 数字、序列、映射。 2. 可变类型和不可变类型

"""语法和语句----语法和语句----语法和语句----语法和语句----语法和语句----语法和语句----语法和语句----语法和语句----语法和语句----语法和语句----语法和语句"""

#-- 赋值语句的形式

spam = 'spam' # 基本形式

spam, ham = 'spam', 'ham' # 元组赋值形式

[spam, ham] = ['s', 'h'] # 列表赋值形式

a, b, c, d = 'abcd' # 序列赋值形式

a, \*b, c = 'spam' # 序列解包形式（Python3.x中才有）

spam = ham = 'no' # 多目标赋值运算，涉及到共享引用

spam += 42 # 增强赋值，涉及到共享引用

#-- 序列赋值 序列解包

[a, b, c] = (1, 2, 3) # a = 1, b = 2, c = 3

a, b, c, d = "spam" # a = 's', b = 'p'

a, b, c = range(3) # a = 0, b = 1

a, \*b = [1, 2, 3, 4] # a = 1, b = [2, 3, 4]

\*a, b = [1, 2, 3, 4] # a = [1, 2, 3], b = 4

a, \*b, c = [1, 2, 3, 4] # a = 1, b = [2, 3], c = 4

# 带有\*时 会优先匹配\*之外的变量 如

a, \*b, c = [1, 2] # a = 1, c = 2, b = []

#-- print函数原型

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

# 流的重定向

print('hello world') # 等于sys.stdout.write('hello world')

temp = sys.stdout # 原有流的保存

sys.stdout = open('log.log', 'a') # 流的重定向

print('hello world') # 写入到文件log.log

sys.stdout.close()

sys.stdout = temp # 原有流的复原

#-- Python中and或or总是返回对象(左边的对象或右边的对象) 且具有短路求值的特性

1 or 2 or 3 # 返回 1

1 and 2 and 3 # 返回 3

#-- if/else三元表达符（if语句在行内）

A = 1 if X else 2

A = 1 if X else (2 if Y else 3)

# 也可以使用and-or语句（一条语句实现多个if-else）

result = (a > 20 and "big than 20" or a > 10 and "big than 10" or a > 5 and "big than 5")

#-- Python的while语句或者for语句可以带else语句 当然也可以带continue/break/pass语句

while a > 1:

anything

else:

anything

# else语句会在循环结束后执行，除非在循环中执行了break，同样的还有for语句

for i in range(5):

anything

else:

anything

#-- for循环的元组赋值

for (a, b) in [(1, 2), (3, 4)]: # 最简单的赋值

for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: # 自动解包赋值

for ((a, b), c) in [((1, 2), 3), ("XY", 6)]: # 自动解包 a = X, b = Y, c = 6

for (a, \*b) in [(1, 2, 3), (4, 5, 6)]: # 自动解包赋值

#-- 列表解析语法

M = [[1,2,3], [4,5,6], [7,8,9]]

res = [sum(row) for row in M] # G = [6, 15, 24] 一般的列表解析 生成一个列表

res = [c \* 2 for c in 'spam'] # ['ss', 'pp', 'aa', 'mm']

res = [a \* b for a in [1, 2] for b in [4, 5]] # 多解析过程 返回[4, 5, 8, 10]

res = [a for a in [1, 2, 3] if a < 2] # 带判断条件的解析过程

res = [a if a > 0 else 0 for a in [-1, 0, 1]] # 带判断条件的高级解析过程

# 两个列表同时解析：使用zip函数

for teama, teamb in zip(["Packers", "49ers"], ["Ravens", "Patriots"]):

print(teama + " vs. " + teamb)

# 带索引的列表解析：使用enumerate函数

for index, team in enumerate(["Packers", "49ers", "Ravens", "Patriots"]):

print(index, team) # 输出0, Packers \n 1, 49ers \n ......

#-- 生成器表达式

G = (sum(row) for row in M) # 使用小括号可以创建所需结果的生成器generator object

next(G), next(G), next(G) # 输出(6, 15, 24)

G = {sum(row) for row in M} # G = {6, 15, 24} 解析语法还可以生成集合和字典

G = {i:sum(M[i]) for i in range(3)} # G = {0: 6, 1: 15, 2: 24}

#-- 文档字符串:出现在Module的开端以及其中函数或类的开端 使用三重引号字符串

"""

module document

"""

def func():

"""

function document

"""

print()

class Employee:

"""

class document

"""

print()

print(func.\_\_doc\_\_) # 输出函数文档字符串

print(Employee.\_\_doc\_\_) # 输出类的文档字符串

#-- 命名惯例:

"""

以单一下划线开头的变量名(\_X)不会被from module import\*等语句导入

前后有两个下划线的变量名(\_\_X\_\_)是系统定义的变量名，对解释器有特殊意义

以两个下划线开头但不以下划线结尾的变量名(\_\_X)是类的本地(私有)变量

"""

#-- 列表解析 in成员关系测试 map sorted zip enumerate内置函数等都使用了迭代协议

'first line' in open('test.txt') # in测试 返回True或False

list(map(str.upper, open('t'))) # map内置函数

sorted(iter([2, 5, 8, 3, 1])) # sorted内置函数

list(zip([1, 2], [3, 4])) # zip内置函数 [(1, 3), (2, 4)]

#-- del语句: 手动删除某个变量

del X

#-- 获取列表的子表的方法:

x = [1,2,3,4,5,6]

x[:3] # 前3个[1,2,3]

x[1:5] # 中间4个[2,3,4,5]

x[-3:] # 最后3个[4,5,6]

x[::2] # 奇数项[1,3,5]

x[1::2] # 偶数项[2,4,6]

#-- 手动迭代：iter和next

L = [1, 2]

I = iter(L) # I为L的迭代器

I.next() # 返回1

I.next() # 返回2

I.next() # Error:StopIteration

#-- Python中的可迭代对象

"""

1.range迭代器

2.map、zip和filter迭代器

3.字典视图迭代器：D.keys()), D.items()等

4.文件类型

"""

"""函数语法规则----函数语法规则----函数语法规则----函数语法规则----函数语法规则----函数语法规则----函数语法规则----函数语法规则----函数语法规则----函数语法规则"""

#-- 函数相关的语句和表达式

myfunc('spam') # 函数调用

def myfunc(): # 函数定义

return None # 函数返回值

global a # 全局变量

nonlocal x # 在函数或其他作用域中使用外层（非全局）变量

yield x # 生成器函数返回

lambda # 匿名函数

#-- Python函数变量名解析:LEGB原则，即:

"""

local(functin) --> encloseing function locals --> global(module) --> build-in(python)

说明:以下边的函数maker为例 则相对于action而言 X为Local N为Encloseing

"""

#-- 嵌套函数举例:工厂函数

def maker(N):

def action(X):

return X \*\* N

return action

f = maker(2) # pass 2 to N

f(3) # 9, pass 3 to X

#-- 嵌套函数举例:lambda实例

def maker(N):

action = (lambda X: X\*\*N)

return action

f = maker(2) # pass 2 to N

f(3) # 9, pass 3 to X

#-- nonlocal和global语句的区别

# nonlocal应用于一个嵌套的函数的作用域中的一个名称 例如:

start = 100

def tester(start):

def nested(label):

nonlocal start # 指定start为tester函数内的local变量 而不是global变量start

print(label, start)

start += 3

return nested

# global为全局的变量 即def之外的变量

def tester(start):

def nested(label):

global start # 指定start为global变量start

print(label, start)

start += 3

return nested

#-- 函数参数，不可变参数通过“值”传递，可变参数通过“引用”传递

def f(a, b, c): print(a, b, c)

f(1, 2, 3) # 参数位置匹配

f(1, c = 3, b = 2) # 参数关键字匹配

def f(a, b=1, c=2): print(a, b, c)

f(1) # 默认参数匹配

f(1, 2) # 默认参数匹配

f(a = 1, c = 3) # 关键字参数和默认参数的混合

# Keyword-Only参数:出现在\*args之后 必须用关键字进行匹配

def keyOnly(a, \*b, c): print('') # c就为keyword-only匹配 必须使用关键字c = value匹配

def keyOnly(a, \*, b, c): ...... # b c为keyword-only匹配 必须使用关键字匹配

def keyOnly(a, \*, b = 1): ...... # b有默认值 或者省略 或者使用关键字参数b = value

#-- 可变参数匹配: \* 和 \*\*

def f(\*args): print(args) # 在元组中收集不匹配的位置参数

f(1, 2, 3) # 输出(1, 2, 3)

def f(\*\*args): print(args) # 在字典中收集不匹配的关键字参数

f(a = 1, b = 2) # 输出{'a':1, 'b':2}

def f(a, \*b, \*\*c): print(a, b, c) # 两者混合使用

f(1, 2, 3, x=4, y=5) # 输出1, (2, 3), {'x':4, 'y':5}

#-- 函数调用时的参数解包: \* 和 \*\* 分别解包元组和字典

func(1, \*(2, 3)) <==> func(1, 2, 3)

func(1, \*\*{'c':3, 'b':2}) <==> func(1, b = 2, c = 3)

func(1, \*(2, 3), \*\*{'c':3, 'b':2}) <==> func(1, 2, 3, b = 2, c = 3)

#-- 函数属性:(自己定义的)函数可以添加属性

def func():.....

func.count = 1 # 自定义函数添加属性

print.count = 1 # Error 内置函数不可以添加属性

#-- 函数注解: 编写在def头部行 主要用于说明参数范围、参数类型、返回值类型等

def func(a:'spam', b:(1, 10), c:float) -> int :

print(a, b, c)

func.\_\_annotations\_\_ # {'c':<class 'float'>, 'b':(1, 10), 'a':'spam', 'return':<class 'int'>}

# 编写注解的同时 还是可以使用函数默认值 并且注解的位置位于=号的前边

def func(a:'spam'='a', b:(1, 10)=2, c:float=3) -> int :

print(a, b, c)

#-- 匿名函数:lambda

f = lambda x, y, z : x + y + z # 普通匿名函数，使用方法f(1, 2, 3)

f = lambda x = 1, y = 1: x + y # 带默认参数的lambda函数

def action(x): # 嵌套lambda函数

return (lambda y : x + y)

f = lambda: a if xxx() else b # 无参数的lambda函数，使用方法f()

#-- lambda函数与map filter reduce函数的结合

list(map((lambda x: x + 1), [1, 2, 3])) # [2, 3, 4]

list(filter((lambda x: x > 0), range(-4, 5))) # [1, 2, 3, 4]

functools.reduce((lambda x, y: x + y), [1, 2, 3]) # 6

functools.reduce((lambda x, y: x \* y), [2, 3, 4]) # 24

#-- 生成器函数:yield VS return

def gensquare(N):

for i in range(N):

yield i\*\* 2 # 状态挂起 可以恢复到此时的状态

for i in gensquare(5): # 使用方法

print(i, end = ' ') # [0, 1, 4, 9, 16]

x = gensquare(2) # x是一个生成对象

next(x) # 等同于x.\_\_next\_\_() 返回0

next(x) # 等同于x.\_\_next\_\_() 返回1

next(x) # 等同于x.\_\_next\_\_() 抛出异常StopIteration

#-- 生成器表达式:小括号进行列表解析

G = (x \*\* 2 for x in range(3)) # 使用小括号可以创建所需结果的生成器generator object

next(G), next(G), next(G) # 和上述中的生成器函数的返回值一致

#（1）生成器(生成器函数/生成器表达式)是单个迭代对象

G = (x \*\* 2 for x in range(4))

I1 = iter(G) # 这里实际上iter(G) = G

next(I1) # 输出0

next(G) # 输出1

next(I1) # 输出4

#（2）生成器不保留迭代后的结果

gen = (i for i in range(4))

2 in gen # 返回True

3 in gen # 返回True

1 in gen # 返回False，其实检测2的时候，1已经就不在生成器中了，即1已经被迭代过了，同理2、3也不在了

#-- 本地变量是静态检测的

X = 22 # 全局变量X的声明和定义

def test():

print(X) # 如果没有下一语句 则该句合法 打印全局变量X

X = 88 # 这一语句使得上一语句非法 因为它使得X变成了本地变量 上一句变成了打印一个未定义的本地变量(局部变量)

if False: # 即使这样的语句 也会把print语句视为非法语句 因为:

X = 88 # Python会无视if语句而仍然声明了局部变量X

def test(): # 改进

global X # 声明变量X为全局变量

print(X) # 打印全局变量X

X = 88 # 改变全局变量X

#-- 函数的默认值是在函数定义的时候实例化的 而不是在调用的时候 例子:

def foo(numbers=[]): # 这里的[]是可变的

numbers.append(9)

print(numbers)

foo() # first time, like before, [9]

foo() # second time, not like before, [9, 9]

foo() # third time, not like before too, [9, 9, 9]

# 改进:

def foo(numbers=None):

if numbers is None: numbers = []

numbers.append(9)

print(numbers)

# 另外一个例子 参数的默认值为不可变的:

def foo(count=0): # 这里的0是数字, 是不可变的

count += 1

print(count)

foo() # 输出1

foo() # 还是输出1

foo(3) # 输出4

foo() # 还是输出1

"""函数例子----函数例子----函数例子----函数例子----函数例子----函数例子----函数例子----函数例子----函数例子----函数例子----函数例子----函数例子----函数例子"""

"""数学运算类"""

abs(x) # 求绝对值，参数可以是整型，也可以是复数，若参数是复数，则返回复数的模

complex([real[, imag]]) # 创建一个复数

divmod(a, b) # 分别取商和余数，注意：整型、浮点型都可以

float([x]) # 将一个字符串或数转换为浮点数。如果无参数将返回0.0

int([x[, base]]) # 将一个字符串或浮点数转换为int类型，base表示进制

long([x[, base]]) # 将一个字符串或浮点数转换为long类型

pow(x, y) # 返回x的y次幂

range([start], stop[, step]) # 产生一个序列，默认从0开始

round(x[, n]) # 四舍五入

sum(iterable[, start]) # 对集合求和

oct(x) # 将一个数字转化为8进制字符串

hex(x) # 将一个数字转换为16进制字符串

chr(i) # 返回给定int类型对应的ASCII字符

unichr(i) # 返回给定int类型的unicode

ord(c) # 返回ASCII字符对应的整数

bin(x) # 将整数x转换为二进制字符串

bool([x]) # 将x转换为Boolean类型

"""集合类操作"""

basestring() # str和unicode的超类，不能直接调用，可以用作isinstance判断

format(value [, format\_spec]) # 格式化输出字符串，格式化的参数顺序从0开始，如“I am {0},I like {1}”

enumerate(sequence[, start=0]) # 返回一个可枚举的对象，注意它有第二个参数

iter(obj[, sentinel]) # 生成一个对象的迭代器，第二个参数表示分隔符

max(iterable[, args...][key]) # 返回集合中的最大值

min(iterable[, args...][key]) # 返回集合中的最小值

dict([arg]) # 创建数据字典

list([iterable]) # 将一个集合类转换为另外一个集合类

set() # set对象实例化

frozenset([iterable]) # 产生一个不可变的set

tuple([iterable]) # 生成一个tuple类型

str([object]) # 转换为string类型

sorted(iterable[, cmp[, key[, reverse]]]) # 集合排序

L = [('b',2),('a',1),('c',3),('d',4)]

sorted(L, key=lambda x: x[1]), reverse=True) # 使用Key参数和reverse参数

sorted(L, key=lambda x: (x[0], x[1])) # 使用key参数进行多条件排序，即如果x[0]相同，则比较x[1]

"""逻辑判断"""

all(iterable) # 集合中的元素都为真的时候为真，特别的，若为空串返回为True

any(iterable) # 集合中的元素有一个为真的时候为真，特别的，若为空串返回为False

cmp(x, y) # 如果x < y ,返回负数；x == y, 返回0；x > y,返回正数

"""IO操作"""

file(filename [, mode [, bufsize]]) # file类型的构造函数。

input([prompt]) # 获取用户输入，推荐使用raw\_input，因为该函数将不会捕获用户的错误输入

raw\_input([prompt]) # 设置输入，输入都是作为字符串处理

open(name[, mode[, buffering]]) # 打开文件，与file有什么不同？推荐使用open

"""其他"""

callable(object) # 检查对象object是否可调用

classmethod(func) # 用来说明这个func是个类方法

staticmethod(func) # 用来说明这个func为静态方法

dir([object]) # 不带参数时，返回当前范围内的变量、方法和定义的类型列表；带参数时，返回参数的属性、方法列表。

help(obj) # 返回obj的帮助信息

eval(expression) # 计算表达式expression的值，并返回

exec(str) # 将str作为Python语句执行

execfile(filename) # 用法类似exec()，不同的是execfile的参数filename为文件名，而exec的参数为字符串。

filter(function, iterable) # 构造一个序列，等价于[item for item in iterable if function(item)]，function返回值为True或False的函数

list(filter(bool, range(-3, 4)))# 返回[-3, -2, -1, 1, 2, 3], 没有0

hasattr(object, name) # 判断对象object是否包含名为name的特性

getattr(object, name [, defalut]) # 获取一个类的属性

setattr(object, name, value) # 设置属性值

delattr(object, name) # 删除object对象名为name的属性

globals() # 返回一个描述当前全局符号表的字典

hash(object) # 如果对象object为哈希表类型，返回对象object的哈希值

id(object) # 返回对象的唯一标识，一串数字

isinstance(object, classinfo) # 判断object是否是class的实例

isinstance(1, int) # 判断是不是int类型

isinstance(1, (int, float)) # isinstance的第二个参数接受一个元组类型

issubclass(class, classinfo) # 判断class是否为classinfo的子类

locals() # 返回当前的变量列表

map(function, iterable, ...) # 遍历每个元素，执行function操作

list(map(abs, range(-3, 4))) # 返回[3, 2, 1, 0, 1, 2, 3]

next(iterator[, default]) # 类似于iterator.next()

property([fget[, fset[, fdel[, doc]]]]) # 属性访问的包装类，设置后可以通过c.x=value等来访问setter和getter

reduce(function, iterable[, initializer]) # 合并操作，从第一个开始是前两个参数，然后是前两个的结果与第三个合并进行处理，以此类推

def add(x,y):return x + y

reduce(add, range(1, 11)) # 返回55 (注:1+2+3+4+5+6+7+8+9+10 = 55)

reduce(add, range(1, 11), 20) # 返回75

reload(module) # 重新加载模块

repr(object) # 将一个对象变幻为可打印的格式

slice(start, stop[, step]) # 产生分片对象

type(object) # 返回该object的类型

vars([object]) # 返回对象的变量名、变量值得字典

a = Class(); # Class为一个空类

a.name = 'qi', a.age = 9

vars(a) # {'name':'qi', 'age':9}

zip([iterable, ...]) # 返回对应数组

list(zip([1, 2, 3], [4, 5, 6])) # [(1, 4), (2, 5), (3, 6)]

a = [1, 2, 3], b = ["a", "b", "c"]

z = zip(a, b) # 压缩：[(1, "a"), (2, "b"), (3, "c")]

zip(\*z) # 解压缩：[(1, 2, 3), ("a", "b", "c")]

unicode(string, encoding, errors) # 将字符串string转化为unicode形式，string为encoded string。

"""模块Moudle----模块Moudle----模块Moudle----模块Moudle----模块Moudle----模块Moudle----模块Moudle----模块Moudle----模块Moudle----模块Moudle----模块Moudle"""

#-- Python模块搜索路径:

"""

(1)程序的主目录 (2)PYTHONPATH目录 (3)标准链接库目录 (4)任何.pth文件的内容

"""

#-- 查看全部的模块搜索路径

import sys

sys.path

#-- 模块的使用代码

import module1, module2 # 导入module1 使用module1.printer()

from module1 import printer # 导入module1中的printer变量 使用printer()

from module1 imoprt \* # 导入module1中的全部变量 使用不必添加module1前缀

#-- 重载模块reload: 这是一个内置函数 而不是一条语句

from imp import reload

reload(module)

#-- 模块的包导入:使用点号(.)而不是路径(dir1\dir2)进行导入

import dir1.dir2.mod # d导入包(目录)dir1中的包dir2中的mod模块 此时dir1必须在Python可搜索路径中

from dir1.dir2.mod import \* # from语法的包导入

#-- \_\_init\_\_.py包文件:每个导入的包中都应该包含这么一个文件

"""

该文件可以为空

首次进行包导入时 该文件会自动执行

高级功能:在该文件中使用\_\_all\_\_列表来定义包(目录)以from\*的形式导入时 需要导入什么

"""

#-- 包相对导入:使用点号(.) 只能使用from语句

from . import spam # 导入当前目录下的spam模块（错误: 当前目录下的模块, 直接导入即可）

from .spam import name # 导入当前目录下的spam模块的name属性（错误: 当前目录下的模块, 直接导入即可，不用加.）

from .. import spam # 导入当前目录的父目录下的spam模块

#-- 包相对导入与普通导入的区别

from string import \* # 这里导入的string模块为sys.path路径上的 而不是本目录下的string模块(如果存在也不是)

from .string import \* # 这里导入的string模块为本目录下的(不存在则导入失败) 而不是sys.path路径上的

#-- 模块数据隐藏:最小化from\*的破坏

\_X # 变量名前加下划线可以防止from\*导入时该变量名被复制出去

\_\_all\_\_ = ['x', 'x1', 'x2'] # 使用\_\_all\_\_列表指定from\*时复制出去的变量名(变量名在列表中为字符串形式)

#-- 可以使用\_\_name\_\_进行模块的单元测试:当模块为顶层执行文件时值为'\_\_main\_\_' 当模块被导入时为模块名

if \_\_name\_\_ == '\_\_main\_\_':

doSomething

# 模块属性中还有其他属性，例如：

\_\_doc\_\_ # 模块的说明文档

\_\_file\_\_ # 模块文件的文件名，包括全路径

\_\_name\_\_ # 主文件或者被导入文件

\_\_package\_\_ # 模块所在的包

#-- import语句from语句的as扩展

import modulename as name

from modulename import attrname as name

#-- 得到模块属性的几种方法 假设为了得到name属性的值

M.name

M.\_\_dict\_\_['name']

sys.modules['M'].name

getattr(M, 'name')

"""类与面向对象----类与面向对象----类与面向对象----类与面向对象----类与面向对象----类与面向对象----类与面向对象----类与面向对象----类与面向对象----类与面向对象"""

#-- 最普通的类

class C1(C2, C3):

spam = 42 # 数据属性

def \_\_init\_\_(self, name): # 函数属性:构造函数

self.name = name

def \_\_del\_\_(self): # 函数属性:析构函数

print("goodbey ", self.name)

I1 = C1('bob')

#-- Python的类没有基于参数的函数重载

class FirstClass:

def test(self, string):

print(string)

def test(self): # 此时类中只有一个test函数 即后者test(self) 它覆盖掉前者带参数的test函数

print("hello world")

#-- 子类扩展超类: 尽量调用超类的方法

class Manager(Person):

def giveRaise(self, percent, bonus = .10):

self.pay = int(self.pay\*(1 + percent + bonus)) # 不好的方式 复制粘贴超类代码

Person.giveRaise(self, percent + bonus) # 好的方式 尽量调用超类方法

#-- 类内省工具

bob = Person('bob')

bob.\_\_class\_\_ # <class 'Person'>

bob.\_\_class\_\_.\_\_name\_\_ # 'Person'

bob.\_\_dict\_\_ # {'pay':0, 'name':'bob', 'job':'Manager'}

#-- 返回1中 数据属性spam是属于类 而不是对象

I1 = C1('bob'); I2 = C2('tom') # 此时I1和I2的spam都为42 但是都是返回的C1的spam属性

C1.spam = 24 # 此时I1和I2的spam都为24

I1.spam = 3 # 此时I1新增自有属性spam 值为2 I2和C1的spam还都为24

#-- 类方法调用的两种方式

instance.method(arg...)

class.method(instance, arg...)

#-- 抽象超类的实现方法

# (1)某个函数中调用未定义的函数 子类中定义该函数

def delegate(self):

self.action() # 本类中不定义action函数 所以使用delegate函数时就会出错

# (2)定义action函数 但是返回异常

def action(self):

raise NotImplementedError("action must be defined")

# (3)上述的两种方法还都可以定义实例对象 实际上可以利用@装饰器语法生成不能定义的抽象超类

from abc import ABCMeta, abstractmethod

class Super(metaclass = ABCMeta):

@abstractmethod

def action(self): pass

x = Super() # 返回 TypeError: Can't instantiate abstract class Super with abstract methods action

#-- # OOP和继承: "is-a"的关系

class A(B):

pass

a = A()

isinstance(a, B) # 返回True, A是B的子类 a也是B的一种

# OOP和组合: "has-a"的关系

pass

# OOP和委托: "包装"对象 在Python中委托通常是以"\_\_getattr\_\_"钩子方法实现的, 这个方法会拦截对不存在属性的读取

# 包装类(或者称为代理类)可以使用\_\_getattr\_\_把任意读取转发给被包装的对象

class wrapper:

def \_\_init\_\_(self, object):

self.wrapped = object

def \_\_getattr(self, attrname):

print('Trace: ', attrname)

return getattr(self.wrapped, attrname)

# 注:这里使用getattr(X, N)内置函数以变量名字符串N从包装对象X中取出属性 类似于X.\_\_dict\_\_[N]

x = wrapper([1, 2, 3])

x.append(4) # 返回 "Trace: append" [1, 2, 3, 4]

x = wrapper({'a':1, 'b':2})

list(x.keys()) # 返回 "Trace: keys" ['a', 'b']

#-- 类的伪私有属性:使用\_\_attr

class C1:

def \_\_init\_\_(self, name):

self.\_\_name = name # 此时类的\_\_name属性为伪私有属性 原理 它会自动变成self.\_C1\_\_name = name

def \_\_str\_\_(self):

return 'self.name = %s' % self.\_\_name

I = C1('tom')

print(I) # 返回 self.name = tom

I.\_\_name = 'jeey' # 这里无法访问 \_\_name为伪私有属性

I.\_C1\_\_name = 'jeey' # 这里可以修改成功 self.name = jeey

#-- 类方法是对象:无绑定类方法对象 / 绑定实例方法对象

class Spam:

def doit(self, message):

print(message)

def selfless(message)

print(message)

obj = Spam()

x = obj.doit # 类的绑定方法对象 实例 + 函数

x('hello world')

x = Spam.doit # 类的无绑定方法对象 类名 + 函数

x(obj, 'hello world')

x = Spam.selfless # 类的无绑定方法是函数 在3.0之前无效

x('hello world')

#-- 获取对象信息: 属性和方法

a = MyObject()

dir(a) # 使用dir函数

hasattr(a, 'x') # 测试是否有x属性或方法 即a.x是否已经存在

setattr(a, 'y', 19) # 设置属性或方法 等同于a.y = 19

getattr(a, 'z', 0) # 获取属性或方法 如果属性不存在 则返回默认值0

#这里有个小技巧，setattr可以设置一个不能访问到的属性，即只能用getattr获取

setattr(a, "can't touch", 100) # 这里的属性名带有空格，不能直接访问

getattr(a, "can't touch", 0) # 但是可以用getattr获取

#-- 为类动态绑定属性或方法: MethodType方法

# 一般创建了一个class的实例后, 可以给该实例绑定任何属性和方法, 这就是动态语言的灵活性

class Student(object):

pass

s = Student()

s.name = 'Michael' # 动态给实例绑定一个属性

def set\_age(self, age): # 定义一个函数作为实例方法

self.age = age

from types import MethodType

s.set\_age = MethodType(set\_age, s) # 给实例绑定一个方法 类的其他实例不受此影响

s.set\_age(25) # 调用实例方法

Student.set\_age = MethodType(set\_age, Student) # 为类绑定一个方法 类的所有实例都拥有该方法

"""类的高级话题----类的高级话题----类的高级话题----类的高级话题----类的高级话题----类的高级话题----类的高级话题----类的高级话题----类的高级话题----类的高级话题"""

#-- 多重继承: "混合类", 搜索方式"从下到上 从左到右 广度优先"

class A(B, C):

pass

#-- 类的继承和子类的初始化

# 1.子类定义了\_\_init\_\_方法时，若未显示调用基类\_\_init\_\_方法，python不会帮你调用。

# 2.子类未定义\_\_init\_\_方法时，python会自动帮你调用首个基类的\_\_init\_\_方法，注意是首个。

# 3.子类显示调用基类的初始化函数：

class FooParent(object):

def \_\_init\_\_(self, a):

self.parent = 'I\'m the Parent.'

print('Parent:a=' + str(a))

def bar(self, message):

print(message + ' from Parent')

class FooChild(FooParent):

def \_\_init\_\_(self, a):

FooParent.\_\_init\_\_(self, a)

print('Child:a=' + str(a))

def bar(self, message):

FooParent.bar(self, message)

print(message + ' from Child')

fooChild = FooChild(10)

fooChild.bar('HelloWorld')

#-- #实例方法 / 静态方法 / 类方法

class Methods:

def imeth(self, x): print(self, x) # 实例方法：传入的是实例和数据，操作的是实例的属性

def smeth(x): print(x) # 静态方法：只传入数据 不传入实例，操作的是类的属性而不是实例的属性

def cmeth(cls, x): print(cls, x) # 类方法：传入的是类对象和数据

smeth = staticmethod(smeth) # 调用内置函数，也可以使用@staticmethod

cmeth = classmethod(cmeth) # 调用内置函数，也可以使用@classmethod

obj = Methods()

obj.imeth(1) # 实例方法调用 <\_\_main\_\_.Methods object...> 1

Methods.imeth(obj, 2) # <\_\_main\_\_.Methods object...> 2

Methods.smeth(3) # 静态方法调用 3

obj.smeth(4) # 这里可以使用实例进行调用

Methods.cmeth(5) # 类方法调用 <class '\_\_main\_\_.Methods'> 5

obj.cmeth(6) # <class '\_\_main\_\_.Methods'> 6

#-- 函数装饰器:是它后边的函数的运行时的声明 由@符号以及后边紧跟的"元函数"(metafunction)组成

@staticmethod

def smeth(x): print(x)

# 等同于:

def smeth(x): print(x)

smeth = staticmethod(smeth)

# 同理

@classmethod

def cmeth(cls, x): print(x)

# 等同于

def cmeth(cls, x): print(x)

cmeth = classmethod(cmeth)

#-- 类修饰器:是它后边的类的运行时的声明 由@符号以及后边紧跟的"元函数"(metafunction)组成

def decorator(aClass):.....

@decorator

class C:....

# 等同于:

class C:....

C = decorator(C)

#-- 限制class属性: \_\_slots\_\_属性

class Student:

\_\_slots\_\_ = ('name', 'age') # 限制Student及其实例只能拥有name和age属性

# \_\_slots\_\_属性只对当前类起作用, 对其子类不起作用

# \_\_slots\_\_属性能够节省内存

# \_\_slots\_\_属性可以为列表list，或者元组tuple

#-- 类属性高级话题: @property

# 假设定义了一个类:C，该类必须继承自object类，有一私有变量\_x

class C(object):

def \_\_init\_\_(self):

self.\_\_x = None

# 第一种使用属性的方法

def getx(self):

return self.\_\_x

def setx(self, value):

self.\_\_x = value

def delx(self):

del self.\_\_x

x = property(getx, setx, delx, '')

# property函数原型为property(fget=None,fset=None,fdel=None,doc=None)

# 使用

c = C()

c.x = 100 # 自动调用setx方法

y = c.x # 自动调用getx方法

del c.x # 自动调用delx方法

# 第二种方法使用属性的方法

@property

def x(self):

return self.\_\_x

@x.setter

def x(self, value):

self.\_\_x = value

@x.deleter

def x(self):

del self.\_\_x

# 使用

c = C()

c.x = 100 # 自动调用setter方法

y = c.x # 自动调用x方法

del c.x # 自动调用deleter方法

#-- 定制类: 重写类的方法

# (1)\_\_str\_\_方法、\_\_repr\_\_方法: 定制类的输出字符串

# (2)\_\_iter\_\_方法、next方法: 定制类的可迭代性

class Fib(object):

def \_\_init\_\_(self):

self.a, self.b = 0, 1 # 初始化两个计数器a，b

def \_\_iter\_\_(self):

return self # 实例本身就是迭代对象，故返回自己

def next(self):

self.a, self.b = self.b, self.a + self.b

if self.a > 100000: # 退出循环的条件

raise StopIteration()

return self.a # 返回下一个值

for n in Fib():

print(n) # 使用

# (3)\_\_getitem\_\_方法、\_\_setitem\_\_方法: 定制类的下标操作[] 或者切片操作slice

class Indexer(object):

def \_\_init\_\_(self):

self.data = {}

def \_\_getitem\_\_(self, n): # 定义getitem方法

print('getitem:', n)

return self.data[n]

def \_\_setitem\_\_(self, key, value): # 定义setitem方法

print('setitem:key = {0}, value = {1}'.format(key, value))

self.data[key] = value

test = Indexer()

test[0] = 1; test[3] = '3' # 调用setitem方法

print(test[0]) # 调用getitem方法

# (4)\_\_getattr\_\_方法: 定制类的属性操作

class Student(object):

def \_\_getattr\_\_(self, attr): # 定义当获取类的属性时的返回值

if attr=='age':

return 25 # 当获取age属性时返回25

raise AttributeError('object has no attribute: %s' % attr)

# 注意: 只有当属性不存在时 才会调用该方法 且该方法默认返回None 需要在函数最后引发异常

s = Student()

s.age # s中age属性不存在 故调用\_\_getattr\_\_方法 返回25

# (5)\_\_call\_\_方法: 定制类的'可调用'性

class Student(object):

def \_\_call\_\_(self): # 也可以带参数

print('Calling......')

s = Student()

s() # s变成了可调用的 也可以带参数

callable(s) # 测试s的可调用性 返回True

# (6)\_\_len\_\_方法：求类的长度

def \_\_len\_\_(self):

return len(self.data)

#-- 动态创建类type()

# 一般创建类 需要在代码中提前定义

class Hello(object):

def hello(self, name='world'):

print('Hello, %s.' % name)

h = Hello()

h.hello() # Hello, world

type(Hello) # Hello是一个type类型 返回<class 'type'>

type(h) # h是一个Hello类型 返回<class 'Hello'>

# 动态类型语言中 类可以动态创建 type函数可用于创建新类型

def fn(self, name='world'): # 先定义函数

print('Hello, %s.' % name)

Hello = type('Hello', (object,), dict(hello=fn))

# 创建Hello类 type原型: type(name, bases, dict)

h = Hello() # 此时的h和上边的h一致

"""异常相关----异常相关----异常相关----异常相关----异常相关----异常相关----异常相关----异常相关----异常相关----异常相关----异常相关----异常相关----异常相关"""

#-- #捕获异常:

try:

except: # 捕获所有的异常 等同于except Exception:

except name: # 捕获指定的异常

except name, value: # 捕获指定的异常和额外的数据(实例)

except (name1, name2):

except (name1, name2), value:

except name4 as X:

else: # 如果没有发生异常

finally: # 总会执行的部分

# 引发异常: raise子句(raise IndexError)

raise <instance> # raise instance of a class, raise IndexError()

raise <class> # make and raise instance of a class, raise IndexError

raise # reraise the most recent exception

#-- Python3.x中的异常链: raise exception from otherException

except Exception as X:

raise IndexError('Bad') from X

#-- assert子句: assert <test>, <data>

assert x < 0, 'x must be negative'

#-- with/as环境管理器:作为常见的try/finally用法模式的替代方案

with expression [as variable], expression [as variable]:

# 例子:

with open('test.txt') as myfile:

for line in myfile: print(line)

# 等同于:

myfile = open('test.txt')

try:

for line in myfile: print(line)

finally:

myfile.close()

#-- 用户自定义异常: class Bad(Exception):.....

"""

Exception超类 / except基类即可捕获到其所有子类

Exception超类有默认的打印消息和状态 当然也可以定制打印显示:

"""

class MyBad(Exception):

def \_\_str\_\_(self):

return '定制的打印消息'

try:

MyBad()

except MyBad as x:

print(x)

#-- 用户定制异常数据

class FormatError(Exception):

def \_\_init\_\_(self, line ,file):

self.line = line

self.file = file

try:

raise FormatError(42, 'test.py')

except FormatError as X:

print('Error at ', X.file, X.line)

# 用户定制异常行为(方法):以记录日志为例

class FormatError(Exception):

logfile = 'formaterror.txt'

def \_\_init\_\_(self, line ,file):

self.line = line

self.file = file

def logger(self):

open(self.logfile, 'a').write('Error at ', self.file, self.line)

try:

raise FormatError(42, 'test.py')

except FormatError as X:

X.logger()

#-- 关于sys.exc\_info:允许一个异常处理器获取对最近引发的异常的访问

try:

......

except:

# 此时sys.exc\_info()返回一个元组(type, value, traceback)

# type:正在处理的异常的异常类型

# value:引发的异常的实例

# traceback:堆栈信息

#-- 异常层次

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

+-- StopIteration

+-- ArithmeticError

+-- AssertionError

+-- AttributeError

+-- BufferError

+-- EOFError

+-- ImportError

+-- LookupError

+-- MemoryError

+-- NameError

+-- OSError

+-- ReferenceError

+-- RuntimeError

+-- SyntaxError

+-- SystemError

+-- TypeError

+-- ValueError

+-- Warning

"""Unicode和字节字符串---Unicode和字节字符串----Unicode和字节字符串----Unicode和字节字符串----Unicode和字节字符串----Unicode和字节字符串----Unicode和字节字符串"""

#-- Python的字符串类型

"""Python2.x"""

# 1.str表示8位文本和二进制数据

# 2.unicode表示宽字符Unicode文本

"""Python3.x"""

# 1.str表示Unicode文本（8位或者更宽）

# 2.bytes表示不可变的二进制数据

# 3.bytearray是一种可变的bytes类型

#-- 字符编码方法

"""ASCII""" # 一个字节，只包含英文字符，0到127，共128个字符，利用函数可以进行字符和数字的相互转换

ord('a') # 字符a的ASCII码为97，所以这里返回97

chr(97) # 和上边的过程相反，返回字符'a'

"""Latin-1""" # 一个字节，包含特殊字符，0到255，共256个字符，相当于对ASCII码的扩展

chr(196) # 返回一个特殊字符：Ä

"""Unicode""" # 宽字符，一个字符包含多个字节，一般用于亚洲的字符集，比如中文有好几万字

"""UTF-8""" # 可变字节数，小于128的字符表示为单个字节，128到0X7FF之间的代码转换为两个字节，0X7FF以上的代码转换为3或4个字节

# 注意：可以看出来，ASCII码是Latin-1和UTF-8的一个子集

# 注意：utf-8是unicode的一种实现方式，unicode、gbk、gb2312是编码字符集

#-- 查看Python中的字符串编码名称，查看系统的编码

import encodings

help(encoding)

import sys

sys.platform # 'win64'

sys.getdefaultencoding() # 'utf-8'

sys.getdefaultencoding() # 返回当前系统平台的编码类型

sys.getsizeof(object) # 返回object占有的bytes的大小

#-- 源文件字符集编码声明: 添加注释来指定想要的编码形式 从而改变默认值 注释必须出现在脚本的第一行或者第二行

"""说明：其实这里只会检查#和coding:utf-8，其余的字符都是为了美观加上的"""

# \_\*\_ coding: utf-8 \_\*\_

# coding = utf-8

#-- #编码: 字符串 --> 原始字节 #解码: 原始字节 --> 字符串

#-- Python3.x中的字符串应用

s = '...' # 构建一个str对象，不可变对象

b = b'...' # 构建一个bytes对象，不可变对象

s[0], b[0] # 返回('.', 113)

s[1:], b[1:] # 返回('..', b'..')

B = B"""

xxxx

yyyy

"""

# B = b'\nxxxx\nyyyy\n'

# 编码，将str字符串转化为其raw bytes形式：

str.encode(encoding = 'utf-8', errors = 'strict')

bytes(str, encoding)

# 编码例子：

S = 'egg'

S.encode() # b'egg'

bytes(S, encoding = 'ascii') # b'egg'

# 解码，将raw bytes字符串转化为str形式：

bytes.decode(encoding = 'utf-8', errors = 'strict')

str(bytes\_or\_buffer[, encoding[, errors]])

# 解码例子：

B = b'spam'

B.decode() # 'spam'

str(B) # "b'spam'"，不带编码的str调用，结果为打印该bytes对象

str(B, encoding = 'ascii')# 'spam'，带编码的str调用，结果为转化该bytes对象

#-- Python2.x的编码问题

u = u'汉'

print repr(u) # u'\xba\xba'

s = u.encode('UTF-8')

print repr(s) # '\xc2\xba\xc2\xba'

u2 = s.decode('UTF-8')

print repr(u2) # u'\xba\xba'

# 对unicode进行解码是错误的

s2 = u.decode('UTF-8') # UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not in range(128)

# 同样，对str进行编码也是错误的

u2 = s.encode('UTF-8') # UnicodeDecodeError: 'ascii' codec can't decode byte 0xc2 in position 0: ordinal not in range(128)

#-- bytes对象

B = b'abc'

B = bytes('abc', 'ascii')

B = bytes([97, 98, 99])

B = 'abc'.encode()

# bytes对象的方法调用基本和str类型一致 但:B[0]返回的是ASCII码值97, 而不是b'a'

#-- #文本文件: 根据Unicode编码来解释文件内容，要么是平台的默认编码，要么是指定的编码类型

# 二进制文件：表示字节值的整数的一个序列 open('bin.txt', 'rb')

#-- Unicode文件

s = 'A\xc4B\xe8C' # s = 'A?BèC' len(s) = 5

#手动编码

l = s.encode('latin-1') # l = b'A\xc4B\xe8C' len(l) = 5

u = s.encode('utf-8') # u = b'A\xc3\x84B\xc3\xa8C' len(u) = 7

#文件输出编码

open('latindata', 'w', encoding = 'latin-1').write(s)

l = open('latindata', 'rb').read() # l = b'A\xc4B\xe8C' len(l) = 5

open('uft8data', 'w', encoding = 'utf-8').write(s)

u = open('uft8data', 'rb').read() # u = b'A\xc3\x84B\xc3\xa8C' len(u) = 7

#文件输入编码

s = open('latindata', 'r', encoding = 'latin-1').read() # s = 'A?BèC' len(s) = 5

s = open('latindata', 'rb').read().decode('latin-1') # s = 'A?BèC' len(s) = 5

s = open('utf8data', 'r', encoding = 'utf-8').read() # s = 'A?BèC' len(s) = 5

s = open('utf8data', 'rb').read().decode('utf-8') # s = 'A?BèC' len(s) = 5

"""其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他----其他"""

#-- Python实现任意深度的赋值 例如a[0] = 'value1'; a[1][2] = 'value2'; a[3][4][5] = 'value3'

class MyDict(dict):

def \_\_setitem\_\_(self, key, value): # 该函数不做任何改动 这里只是为了输出

print('setitem:', key, value, self)

super().\_\_setitem\_\_(key, value)

def \_\_getitem\_\_(self, item): # 主要技巧在该函数

print('getitem:', item, self) # 输出信息

# 基本思路: a[1][2]赋值时 需要先取出a[1] 然后给a[1]的[2]赋值

if item not in self: # 如果a[1]不存在 则需要新建一个dict 并使得a[1] = dict

temp = MyDict() # 新建的dict: temp

super().\_\_setitem\_\_(item, temp) # 赋值a[1] = temp

return temp # 返回temp 使得temp[2] = value有效

return super().\_\_getitem\_\_(item) # 如果a[1]存在 则直接返回a[1]

# 例子:

test = MyDict()

test[0] = 'test'

print(test[0])

test[1][2] = 'test1'

print(test[1][2])

test[1][3] = 'test2'

print(test[1][3])

#-- Python中的多维数组

lists = [0] \* 3 # 扩展list，结果为[0, 0, 0]

lists = [[]] \* 3 # 多维数组，结果为[[], [], []]，但有问题，往下看

lists[0].append(3) # 期望看到的结果[[3], [], []]，实际结果[[3], [3], [3]]，原因：list\*n操作，是浅拷贝，如何避免？往下看

lists = [[] for i in range(3)] # 多维数组，结果为[[], [], []]

lists[0].append(3) # 结果为[[3], [], []]

lists[1].append(6) # 结果为[[3], [6], []]

lists[2].append(9) # 结果为[[3], [6], [9]]

lists = [[[] for j in range(4)] for i in range(3)] # 3行4列，且每一个元素为[]