* 注释以#开头
* 当语句以冒号:结尾时，缩进的语句视为代码块。
* 约定俗成应该使用4个空格的缩进。
* 通常在文件开头写上这两行

#!/usr/bin/env python3

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

第一行注释告诉Linux/OS X系统这是一个Python可执行程序，Windows系统会忽略这个注释；

第二行注释告诉Python解释器，按照UTF-8编码读取源代码，否则中文输出可能会有乱码

* cmd输入pip install you-get

然后you-get http://www.bilibili.com/video/av33417359/

av33417359就被下载到当前目录里

# Python交互模式

* 在cmd敲命令python进入到Python交互模式，提示符是>>>。
* 在Python交互模式下输入exit()并回车，退出Python交互模式
* 也可以直接在开始菜单或桌面点击Python (command line)，直接进入Python交互模式，输入exit()后窗口会直接关闭，不会回到命令行模式

hello.py

print(100+200+3000)

命令行模式下进入.py目录，执行hello,py文件E:\tempDirectory>python hello.py

3300

# print格式化输出

print('feafea', 'nihao', 'shijie')

输出feafea nihao shijie

>>> s='hello'

>>> "The length of %s is %d" % (s,len(s))

The length of hello is 5

* 如果只有一个占位符则不需要括号
* 在格式化字符串的条件下（前边有%s之类的占位符，中间有%符号），字符串里需要一个普通的'%'，用'%%'转义字符表示。

>>> s='hello'

>>> 'nihao%s'

'nihao%s'

>>> 'nihao%s' % s

'nihaohello'

>>> 'nihao%%'

'nihao%%'

>>> 'nihao%%%s' % s

'nihao%hello'

'%.3f' % 6.22335 #6.223

* 精度值

'%f' '%e' '%g'中使用，表示出现在小数点后的位数。

'%s'中使用，表示字符串最大字段宽度

'%10.3f' % 6.22335

' 6.223'

* 输出最小宽度，不够用空格补齐。如果多了就无视这个数值

'%-10.3f' % 6.22335

'6.223 '

* 负号表示左对齐

'%-+10.3f' % 6.22335

'+6.223 '

* 正号表示在数字前加上正负号标志

'%+010.3f' % 6.22335

'+00006.223'

* 0表示如果位数不够就用0补齐

%d 带符号的十进制整数

%o 八进制

%x 十六进制

%u 不带符号的十进制

%f %F 十进制浮点数，默认保留小数点后六位

%e 科学计数法表示的浮点数，e小写。默认保留小数点后六位

%E 科学计数法表示的浮点数，E大写。默认保留小数点后六位

%g 如果数字位数(宽度)>6，用科学计数法表示；如果<6，则用小数

%C 单字符（接受整数或者单字符字符串）

%r 字符串（使用repr转换任意python对象)

%s 字符串（使用str转换任意python对象）

其实数字、Bool值也可以用%s

>>> 'Age: %s. Gender: %s' % (25, True)

'Age: 25. Gender: True'

不输出换行

print(pi, end='')

* format()方法

>>> 'Hello, {0}, 成绩提升了 {1:.1f}%'.format('小明', 17.125)

'Hello, 小明, 成绩提升了 17.1%'

# input()

>>> name = input()

之后命令行等待输入一个字符串。按enter后存储到name变量中

>>> name=input("what's your name? ")

在输入name之前有提示what's your name?

input()接收到的是string，即使输入的是1

# 条件判断&循环

if x:

<执行1>

elif <条件判断2>:

<执行2>

elif <条件判断3>:

<执行3>

else:

<执行4>

注意冒号

>>> names=['a', 'fafa', 123]

>>> for s in names:

... print(s)

a

fafa

123

for in遍历list或tuple中的每个元素。就是foreach循环

for n in list(range(5)):

sum=n+sum

相当于for(int i=0, i<5; i++)

list(range(5))生成一个0-4的整数list

range(1, 4)生成一个1-3的序列

亲测for n in range(5)也可以，但range(5)不是一个list。就是range类型

>>> for x, y in [(1, 1), (2, 4), (3, 9)]:

... print(x, y)

for循环可以同时引用两个变量

break跳出当前循环

continue跳过一次

n=0

while n<=100:

n=n+1

while循环

# 数据类型

## 基本数据类型

* 整数int

十六进制数字加0x前缀。0xa5b4c3d2

* 浮点数float

3.21 -5.3

对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x109就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5

* 字符串str

字符串双引号里有单引号，单引号不解析不转义，就是引号

貌似单引号里也可以有双引号

"I'm OK'"

就是I'm OK'

* 转义字符\' \" \n \t \b \\

r' '表示' '内部的字符串不转义

>>> print(r'\fa\fa\sv\vd\\\feaw\\f\\\\b\t\r\n')

\fa\fa\sv\vd\\\feaw\\f\\\\b\t\r\n

* 多行内容

print('''line1

line2

\tline3''')

line1

line2

line3

如果在命令行而不是文件中，>>>提示符将会变成…

* 布尔值bool

True False 开头大写

>>> **True** and False

False

可以用and or not进行逻辑运算

* 空值NoneType

**None**

* 函数function
* 字节bytes
* object

type(object)返回这个

## 变量

同一个变量可以反复赋值，而且可以是不同类型的变量

a = 'ABC'

b = a

a = 'XYZ'

print(b)

ABC

解释器创建字符串ABC和变量a🡪将a指向ABC🡪b指向ABC，而不是指向a🡪创建字符串XYZ🡪将a指向XYZ

>>> 9/3

3.0

/除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数

10//3

3

//地板除，运算保留整数

10%3

1

## 编码

* Unicode用两个字节或四个字节表示一个字符
* UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节
* Python 3中，字符串以Unicode编码，转化为UTF-8存储

ord()函数字符🡪整数，chr()函数整数编码🡪字符：

>>> ord('A')

65

>>> ord('中')

20013

>>> chr(66)

'B'

>>> chr(25991)

'文'

如果知道字符的Unicode整数编码，还可以用十六进制这么写：

>>> '\u4e2d\u6587'

'中文'

十进制25591=16进制6587

\x5a \x后接两个16进制字符。可以表示存储字符的字节

\uxxxx \u后接4个16进制字符，相当于用64位表示一个字符

\Uxxxxxxxx \U后接8个16进制字符，相当于用128位表示字符

>>> '\x51'

'Q'

>>> '\u51e1'

'凡'

bytes，带b前缀的单引号或双引号表示：

x = b'ABC'

bytes的每个字符都只占用一个字节

>>> 'ABC'.encode('ascii')

b'ABC'

>>> '中文'.encode('utf-8')

b'\xe4\xb8\xad\xe6\x96\x87'

以Unicode表示的str通过encode()方法可以编码为指定的bytes

‘中文’Unicode是'\u4e2d\u6587'，对应的UTF-8编码是\xe4\xb8\xad\xe6\x96\x87

上面b'\xe4\xb8\xad\xe6\x96\x87'，感觉上同b'e4b8ade69687'，都是用两个16进制字符表示单个字节嘛。

奇怪的是这俩并不相同。b'\xe4\xb8\xad'和'\xe4\xb8\xad'不同是肯定的，一个是字节流，一个是三个字符。但是b'\xe4\xb8\xad'和b'e4b8ad'的区别在哪至今没弄明白。

\x的作用是告诉解释器接下来两个字符应该解释为十六进制，并用相应的字节替换。而不带\x的str可能解释器不知道具体哪两个相邻字符是一个字节

从网络或磁盘上读取字节流，读到的数据就是bytes。decode()方法把bytes变为str

>>> b'ABC'.decode('ascii')

'ABC'

>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')

'中文'

'abc'的UTF-8编码就是'abc'，python中用b'abc'表示

len()计算str的字符数，或者bytes的字节数

>>> len("abcdefg")

7

>>> len(b"abcdefg")

7

>>> len("房间")

2

>>> len("房间".encode('utf-8'))

6

* 所以，1个中文字符经过UTF-8编码后通常会占用3个字节，1个英文字符只占用1字节

## list

列表，我感觉就是数组

>>> stu=['fe', 'zh', 'kf']

>>> len(stu) #len()获取list长度

3

>>> stu[0]

'fe'

>>> stu[-1]

'kf' #倒数第一个元素

* 追加元素到末尾：

>>> stu.append(12)

* 把元素插入到指定的位置，比如索引号为1的位置：

>>> stu=['Mike', 'Rog', 212]

>>> stu.insert(1, 'Hulk')

>>> stu

['Mike', 'Hulk', 'Rog', 212]

* 删除list末尾的元素

>>> stu.pop()

212

>>> stu

['Mike', 'Hulk', 'Rog']

* 删除指定位置的元素

>>> stu.pop(2)

'Rog'

>>> stu

['Mike', 'Hulk']

* 替换

stu[1]='bbb'

## tuple

元组，但是tuple一旦初始化就不能修改

>>> classmates = ('Michael', 'Bob', 'Tracy')

只有1个元素的tuple定义时必须加一个逗号,，来消除歧义：

>>> t = (1,)

>>> t

(1,)

tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的

>>> stu=('nihao', 'shijie', [1, 2])

>>> stu[1]

'shijie'

>>> stu[1]=3

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

不可以

>>> stu[2][1]=1

可以

>>>stu=1

可以

## dict

字典，就是map，感觉就是个对象啊。有引号，还是个Json对象

d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

另一种初始化方法

d = dict(name='Bob', age=20, score=88)

d['Michael']

取值

d['Adam'] = 67

增加键值对

删除

d.pop('Michael')

* 判断key是否存在

方法一

>>> 'Thomas' in d

False

方法二 get()方法，不存在返回None，或者自己指定的value；存在返回key对应的value

>>> d.get('Thomas')

>>> d.get('Thomas', -1)

-1

* dict内部存放的顺序和key放入的顺序没有关系
* key可以是整数

## set

集合

不重复，无序

s = set([1, 2, 3, 3]) #自动过滤重复元素

用list或tuple初始化set

s.add(42)  
s.remove(3)

可以进行交并操作

>>> s1 & s2

{2, 3}

>>> s1 | s2

{1, 2, 3, 4}

## 检查类型

print(**type**('fefa'))

<class 'str'>

print(type(type('fefefe')))

<class 'type'>

print(type(type))

<class 'type'>

type本身是一种type类型，不是函数

class Student(object):

pass

print(type(Student))

<class 'type'>

print(type(Student()))

<class '\_\_main\_\_.Student'>

print(type(Student)==type)

True

print(type(Student())==Student)

True

* type()返回一个type类型，这就和JavaScript不一样
* type(obj)会返回实例对应的class

int是整数类型，float浮点数，str字符串，types.FunctionType函数

>>> import types

>>> type(fn)==types.FunctionType #判断是不是函数

True

>>> type(abs)==types.BuiltinFunctionType

True

>>> type(lambda x: x)==types.LambdaType

True

>>> type((x for x in range(10)))==types.GeneratorType #判断是不是Generator

True

isinstance(x, (int, float))

* isinstance() 与 type() 区别：

type() 不会认为子类是一种父类类型，不考虑继承关系。

isinstance() 会认为子类是一种父类类型，考虑继承关系。

## 数据类型转换

str, float🡪int

int('23')

str, int🡪float

float('12.34')

int, float🡪str

str(123)

\*🡪bool

bool(1) #true

# 函数

* python自带函数
* abs(-20)
* max(3,4,60,2,1,32)
* type()
* isinstance(3, (int, float))
* 定义函数

def my\_abs(x):

if x >= 0:

return x

else:

return -x

def fun1(x, y):

if x >= 0:

return x, y

else:

return -x, -y

a, b=fun1(-1, 2)

python可以有多个返回值，实际上返回了一个tuple元组

多个变量接受一个tuple，按位置赋给对应的值

* 空函数 必须使用pass占位

def nop():

pass

pass也可以用在类、if语句中

* 如果没有return语句，函数执行完毕后也会返回None。return None可以简写为return

## 参数

### 默认参数

def fun1(x, y, z=10, c=39, e=10):

pass

调用

fun1(1, 2)

fun1(23, 1, c=56)

c的值设为56，其他使用默认值。但是这样子就得知道函数内部定义时的参数名

默认参数必须指向不变对象，比如None，int。不能把默认参数设置为list，如果是list，函数体内修改了list的值，下次调用默认的参数就不是原来设置的默认参数了

### 可变参数

可变参数允许传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple

def calc(\*numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

cal(3,4,5,6,7)

函数内部接收到一个tuple，3,4,5,6,7不用提前处理成一个list或tuple

如果参数已经是一个list，例如num=[1, 2, 3]

cal(num[0], num[1], num[2])

或

cal(\*num)

把list或tuple的元素变成可变参数传进去：

### 关键字参数

而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。

def person(name, age, \*\*kw):

* 调用方法1

传入任意个数的关键字参数

person('Adam', 45, gender='M', job='Engineer')

* 调用方法2

extra = {'city': 'Beijing', 'job': 'Engineer'}

person('Jack', 24, \*\*extra)

kw获得extra的一份拷贝，函数内改变kw不改变extra

print(kw)

{'city': 'Beijing', 'job: 'Engineer'}

函数内部参数组装为dict

### 命名关键字参数

关键字参数的基础上，限制关键字参数的名字

只接收city和job作为关键字参数

def f1(aaa, \*, bbb, ccc):

print(bbb, ccc)

\*后面的参数被视为命名关键字参数

f1(1, 2, 3)

报错

f1(1, bbb=2, ccc=3)

正确

f1(1, bbb=2)

报错，ccc是必选的参数。

关键字参数不是必需的，命名关键字参数是必需的

* 如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*了：

def person(name, age, \*args, city, job):

print(name, age, args, city, job)

* 命名关键字参数可以有缺省值

def f1(aaa, \*, bbb=4, ccc):

print(bbb, ccc)

由于命名关键字参数city具有默认值，调用时，可不传入city参数：

f1(2, ccc=3)

5种参数可以组合使用，定义的顺序必需是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数

def f2(a, b, c=0, \*args, d, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw, 'args=', args)

f2(1, 1, 123, 2, 3, 4, 5, d=333, nihao='nihao')

# a = 1 b = 1 c = 123 d = 333 kw = {'nihao': 'nihao'} args= (2, 3, 4, 5)

如果是以上这种声明方法，a，b赋值后，先给默认参数c赋值，而不是可变参数。如果默认参数和命名关键字参数之间还有参数，那么就赋值给可变参数。

而无论多复杂的参数形式，都可以通过一个tuple+一个dict通过func(\*args, \*\*kw)的形式调用

t=(1, 1, 123, 2, 3, 4, 5)

d={'d': 333, 'nihao': 'nihao'}

f2(\*t, \*\*d)

输出相同

\*args和\*\*kw是Python的习惯写法。

## 高阶函数Higher-order function

一个函数就接收另一个函数作为参数，这种函数就称之为高阶函数。

def add(x, y, f):

return f(x) + f(y)

print(add(-5, 6, abs))

### map

map()函数接收两个参数，一个是函数，一个是Iterable。

map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

>>> def f(x):

... return x \* x

>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> list(r)

[1, 4, 9, 16, 25, 36, 49, 64, 81]

由于结果r是一个Iterator，Iterator是惰性序列，因此通过list()函数让它把整个序列都计算出来并返回一个list。

注意，map是接收Iterable作为参数，而generator就是一种iterable。当接收generator作为参数时，虽然generator不存储序列的每个值，但是都要计算一次，然后将结果返回。后面的reduce和filter都是

### reduce

这个函数接收两个参数function和list，reduce把function的结果继续和list的下一个元素做累积计算，其效果就是：

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

所以这个函数有且只能有两个参数

>>> from functools import reduce

>>> def add(x, y):

... return x + y

>>> reduce(add, [1, 3, 5, 7, 9])

25

### filter

filter()把传入的函数依次作用于Iterable的每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

def is\_odd(n):

return n % 2 == 1

list(filter(is\_odd, [1, 2, 4, 5, 6, 9, 10, 15]))

### sorted

>>> sorted([36, 5, -12, 9, -21])

[-21, -12, 5, 9, 36]

逆序

>>> sorted([1, 3, 5, 2313, 321, 312, 555, 12], reverse=True)

[2313, 555, 321, 312, 12, 5, 3, 1]

用一个key函数把list的元素映射为其他的数据进行比较，最后显示还是原来的数据

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)

['about', 'bob', 'Credit', 'Zoo']

# 切片Slice

取一个子数组、子字符串

L=[0, 1, 2, 3, 4, 5]

>>> L[0:3]

[0, 1, 2]

>>> L[:3]

和L[0:3]效果一样

>>> L[1:3]

[1, 2]

>>> L[-2:]

取L[-2], L[-1]，即[4, 5]

>>> L[-2:-1]

[4]

前4个数，每两个取一个：

>>> L[:4:2]

[0, 2]

复制一个list：

L[:]

[0, 1, 2, 3, ..., 99]

tuple也可以进行切片操作

(0, 1, 2, 3, 4, 5)[:3]

(0, 1, 2)

字符串也可以用切片操作，操作结果仍是字符串：

>>> 'ABCDEFG'[:3]

'ABC'

Python没有针对字符串的截取函数，只需要切片操作

# 迭代/遍历

* Python的for循环抽象程度要高于C的for循环，因为Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。
* 很多数据类型没有下标，但是只要是可迭代对象，无论有无下标，都可以迭代。即使像list这样的有下标的数据类型也不通过下标迭代
* 判断是否是可迭代对象

>>> from collections import Iterable

>>> isinstance('abc', Iterable)

True

报警告……deprecated

* 遍历dict

for key in dic:

for value in d.values():

for k, v in d.items():

* 遍历字符串

for ch in 'abcdefg':

for index, ch in enumrate('abcdefg'):

* 遍历数组list

for value in [123, 34, 56]:

for index, value in enumerate(['A', 'B', 'C']):

enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。返回一个enumerate类型

# 列表生成式/列表推导式 List Comprehensions

用于生成一个list

>>>list(range(1, 11))

>>> [x \* x for x in range(1, 11)]

生成list[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

把要生成的元素x\*x写在前面，后面跟for循环

for循环后面还可以加上if判断

>>> [x \* x for x in range(1, 11) if x % 2 == 0]

[4, 16, 36, 64, 100]

还可以使用两层循环，可以生成全排列：

>>> [m + n for m in 'ABC' for n in 'XYZ']

['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

# 生成器generator

创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。所以，如果列表元素可以按照某种算法推算出来，循环的过程中不断推算出后续的元素，这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator

* 创建

g=(x\*x for x in range(10))

print(g)

返回<generator object <genexpr> at 0x0078CF70>

就是列表生成式的中括号变成小括号

创建以后这些值没有被计算出来也没有存储

* 通过函数创建

>>> def nihao():

... yield(1)

... yield(2)

... yield(3)

... return 'nihao'

函数中包含yield关键字，则此函数不再是一个普通函数，而是一个generator

* 访问

>>> a=nihao()

>>> next(a)

1

>>> next(a)

2

>>> next(a)

3

>>> next(a)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration: nihao

generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而generator在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行

a=nihao()先得到一个generator对象

next(a)不断获得下一个返回值

没有更多的返回值时，抛出StopIteration的错误。

for n in nihao():

print(n)

generator对象也是可迭代对象，可以用for循环迭代。

通过for循环访问也是最常用的方法，next()基本不用

而且for循环也不抛出错误

def fib(n):

a=0

b=1

temp=0

while(temp<n):

yield b

c=a

a=b

b=b+c

temp=temp+1

return 'over'

for x in fib(6):

print(x)

生成斐波那契数列

# 迭代器

* 可以直接作用于for循环的对象统称为可迭代对象：Iterable

比如list、tuple、dict、set、string、generator

* 可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator

比如生成器

可以使用isinstance()判断一个对象是否是Iterator对象：

>>> from collections import Iterator

>>> isinstance((x for x in range(10)), Iterator)

True

* 把list、dict、str等Iterable变成Iterator可以使用iter()函数：

>>> isinstance(iter([]), Iterator)

True

Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

# 模块

#!/usr/bin/env python3

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

' a test module '

\_\_author\_\_ = 'Ran Lin'

* 自建模块标准开头模板
* 第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释
* 第6行使用\_\_author\_\_变量把作者写进去
* 在模块文件中python把\_\_name\_\_变量设置为'\_\_main\_\_'

import sys

print(sys.argv)

引入其他模块

E:\temp\test python>python test.py fefefe

['test.py', 'fefefe']

sys.argv存储了命令行的参数，第一个就是文件名，后面就是是紧跟着传入的参数

引入其他的在同一目录下的自建模块

import test1

test1.test()

test1.py

def test():

print('fe')

## json

* json编码

>>> import json

>>> d = dict(name='Bob', age=20, score=88)

>>> json.dumps(d)

'{"age": 20, "score": 88, "name": "Bob"}'

json.dumps()把一个对象转化成json格式的str

* json解码

>>> json.loads('{"age": null, "score": 88, "name": "Bob"}')

{'age': None, 'score': 88, 'name': 'Bob'}

可以自动把null转换为python类型None

可以通过重写方法把自定义类转换为json

## datetime

from datetime import datetime

print(datetime.now())

2020-08-11 23:43:32.351407

导入datetime模块中的datetime类

print(type(datetime.now()))

<class 'datetime.datetime'>

dt = datetime(2015, 4, 19, 12, 20)

用指定日期时间创建datetime

>>> datetime.now().timestamp()

1597161017.740043

获取时间戳，从1970年1月1号0点到UTC+00:00时区的现在时刻的秒数。如果要算北京时间的时间戳，要加上28800秒

要把timestamp转换为datetime，使用datetime提供的fromtimestamp()方法：

>>> print(datetime.fromtimestamp(1597161017.74))

2020-08-11 23:50:17.740000

时间戳转为datetime。注意！datetime有时区，转化的时候自动转化成当前时区，加了8小时

>>> cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')

str转换为datetime

datetime.now().strftime('%a, %b %d %H:%M')

datetime转换为str

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用+和-运算符，不过需要

>>> from datetime import datetime, timedelta

>>> now=datetime.now()

>>> now

datetime.datetime(2020, 8, 12, 0, 6, 14, 201217)

>>> now+timedelta(days=8, hours=14)

datetime.datetime(2020, 8, 20, 14, 6, 14, 201217)

导入timedelta类,可以直接对datetime进行加减计算，获得一段时间之前或之后的时间

datetime类型有一个时区属性tzinfo，但是默认为None，所以无法区分这个datetime到底是哪个时区

可以强行给datetime设置一个时区：

from datetime import datetime, timedelta, timezone

utc8=timezone(timedelta(hours=8))

now=datetime.now()

dt=now.replace(tzinfo=utc8)

print(dt)

2020-11-22 21:49:48.711780+08:00

如果不replace时区信息，则print没有后边的+8:00

timezone(timedelta(hours=8)) 创建时区UTC+8:00

.replace(tzinfo=tz\_utc\_8) 设置时区信息

replace只是更换datetime的时区信息，时间的数值不变。比如replace前后都是21:49:48，但很明显0时区和+8时区的21:49:48已经不是一个时间了

from datetime import datetime, timedelta, timezone

utc\_dt = datetime.utcnow().replace(tzinfo=timezone.utc)

print(utc\_dt)

bj\_dt = utc\_dt.astimezone(timezone(timedelta(hours=8)))

print(bj\_dt)

2020-11-22 13:55:04.354269+00:00

2020-11-22 21:55:04.354269+08:00

datetime.utcnow()拿到UTC时间。.replace()将时区强制设为0

astimezone()转换到北京时区+8:00

astimezone前后时间数值发生变化，时区也发生变化

也可以通过bj\_dt.astimezone(timezone(timedelta(hours=9))将北京时间转为东京时间

存储datetime最佳方法是将其转换为timestamp，因为timestamp的值与时区完全无关

## collections

### namedtuple

>>> from collections import namedtuple

>>> Point = namedtuple('Point', ['x', 'y'])

>>> p = Point(1, 2)

>>> p.x

1

>>> p.y

2

namedtuple是一个函数，它用来创建一个自定义的tuple对象，并且规定了tuple元素的个数，并可以用属性而不是索引来引用tuple的某个元素。

这样一来，我们用namedtuple可以很方便地定义一种数据类型，它具备tuple的不变性，又可以根据属性来引用，使用十分方便。

可以验证创建的Point对象是tuple的一种子类：

>>> isinstance(p, Point)

True

>>> isinstance(p, tuple)

True

类似的，如果要用坐标和半径表示一个圆，也可以用namedtuple定义：

# namedtuple('名称', [属性list]):

Circle = namedtuple('Circle', ['x', 'y', 'r'])

# 类

* 声明类

class Student(object):

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

self.name = name

self.score = score

类名大写开头

(object)表示该类是从哪个类继承下来的，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类

\_\_init\_\_就相当于constructor了

self指向实例本身，永远是\_\_init\_\_的第一个参数，创建实例时不传入

* 创建实例

bart1=Student('Bjergsen', 44)

* 给实例绑定属性

bart.name = 'Bart Simpson'

可以绑定类定义时没有的属性

亲测可以

class Student(object):

def print\_score(self):

print(self.aaa)

s1.aaa='aaaaaaa'

s1.print\_score()

类里没有规定aaa属性，但是创建实例后绑定属性，类里原来规定的函数可以正常访问

如果s1.print\_score()在s1.aaa之前当然会报错

* 定义方法

class Student(object):

def printMsg(self):

print(self.name)

类里头定义的所有方法第一个参数都必需是self，而且实例调用时自动传入。

* 私有属性

把属性的名称前加上两个下划线\_\_，就变成了私有属性

class Student(object):

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

self.\_\_name = name

self.\_\_score = score

外部实例无法访问\_\_name和\_\_score

想访问和修改的话写get和set方法

def get\_name(self):

return self.\_\_name

def set\_name(self, name)

self.\_\_name=name

需要注意的是，在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量可以直接访问，不是private变量

属性名\_name，这样的实例变量外部是可以访问的，意思是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量也不是一定不能从外部访问。\_\_name被Python解释器对外修改成了\_Student\_\_name，仍然可以通过\_Student\_\_name来访问\_\_name变量

print(s.\_Student\_\_name)

## 继承

## 多态

class Animal(object):

def run(self):

print('the animal is running')

class Dog(Animal):

def run(self):

print('the dog is running')

d=Dog();

d.run(); # the dog is running

子类的run()覆盖了父类的run()。这就是多态。

def whoisrunning(animal):

animal.run()

不管是animal、dog、cat还是其他的类，调用此函数传入参数都调用自己类中的方法。也是多态

## 获取/修改/添加对象属性

>>> dir('ABC')

获得一个对象的所有属性名和方法名，返回一个包含字符串的list

返回['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

* 类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度

>>> len('ABC')

>>> 'ABC'.\_\_len\_\_()

效果一样

>>> class MyDog(object):

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

... return 100

...

>>> dog = MyDog()

>>> len(dog)

100

所以在自定义类中编写\_\_len\_\_方法，就可以使用len(obj)之类的形式

hasattr(obj, 'x')

检测obj有无x属性

setattr(obj, 'z', 19) # 设置一个属性'y'

getattr(obj, 'z') # 获取属性'y'

可以传入一个default参数，如果属性不存在，就返回默认值：

getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404

## 类属性

class Student(object):

name = 'Student'

这玩意还不是静态变量，通过实例改变以后，类变量值不变。不对，应该是实例没有权限修改类变量，obj.name='tea'效果是创建了一个新的名为name的实例属性，屏蔽掉了类属性

不过用Student.name='tea'修改以后还是起作用的。所以这玩意儿就是静态变量

## 动态绑定属性

python是动态语言，具有极大的灵活性，所以创建实例后还能给对象绑定新的方法

>>> 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

>>> s.set\_age(25) # 调用实例方法

给一个实例绑定的方法，对另一个实例是不起作用的

亲测s.set\_age=set\_age也能起到类似的作用，但第一个参数就不是self了，是一个普通参数

* 给类（所有实例）绑定方法

为了给所有实例都绑定方法，可以给class绑定方法：

>>> def set\_score(self, score):

... self.score = score

...

>>> Student.set\_score = set\_score

给class绑定方法后，所有实例均可调用：

通常情况下，上面的set\_score方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

* 限制属性绑定

class Student(object):

\_\_slots\_\_ = ('name', 'age')

>>> s = Student()

>>> s.age = 25

>>> s.score = 99

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Student' object has no attribute 'score'

如果想要限制实例的属性，比如只允许对Student实例添加name和age属性。不允许添加score或其他属性。则定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性

\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的；子类仍然可以随便定义

## @property

class Student(object):

@property

def score(self):

return self.\_score

@score.setter

def score(self, value):

if not isinstance(value, int):

raise ValueError('score must be an integer!')

if value < 0 or value > 100:

raise ValueError('score must between 0 ~ 100!')

self.\_score = value

两个score函数本来一个是get\_score，一个是set\_score

现在，可以看成类里没有get、sete方法，只有score属性。可以读取和赋值

普通的赋值不能检查score值是否符合规范，而用@property则实现了赋值前检查

从代码里也可以看出，这是把score绑定为另一个\_score属性实现的。在类外也可以通过s.\_score正常访问score。

如果直接设置s.\_score也可以，而且会影响到score的值，但强烈建议不要这样做。属性名前有\_的不要直接访问。

可以值设置@property，不设置setter，属性就变为只读属性

## 多重继承

class Bat(Mammal, Flyable):

pass

继承多个父类的所有功能

## \_\_str\_\_

class Student(object):

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

self.name=name

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

return 'Hello my name is '+self.name

print(Student('zwm'))

Hello my name is zwm

相当于重写toString(),调用print()的时候调用\_\_str\_\_

## \_\_repr\_\_

直接在命令行里Student('zwm')而不用print，打印出来的实例还是没有格式化

这时需要重写\_\_repr\_\_()

或者类声明最后一行

\_\_repr\_\_ = \_\_str\_\_

## \_\_iter\_\_

* 如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

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

现在，试试把Fib实例作用于for循环：

\_\_getitem\_\_() 让这个Fib可以f[3]，像这样一样用索引访问元素

## 枚举类

为枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例

from enum import Enum

Month = Enum('Months', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))

print(Month)

<enum 'Months'>

print(type(Month))

<class 'enum.EnumMeta'>

print(type(Month.Jan))

<enum 'Months'>

枚举所有成员

for n, m in Month.\_\_members\_\_.items():

print(n, m, m.value)

n是字符串类型的'Jan' 'Feb' 'Mar'

m是枚举类型的Months.Jan，Months.Feb，Months.Mar

value属性是自动赋给成员的int常量，默认从1开始计数。

定义枚举时，成员名称不允许重复

默认情况下，不同的成员值允许相同。但是两个相同值的成员，第二个成员的名称被视作第一个成员的别名

* 从Enum派生出自定义类，更精确地控制枚举类型

from enum import Enum, unique

@unique

class Weekday(Enum):

Sun = 0 # Sun的value被设定为0

Mon = 1

Tue = 2

Wed = 3

@unique装饰器保证没有重复值

查不到Month = Enum('Month',……)这样声明时第一个参数什么作用，所以用第二种方法声明吧

* 访问

>>> print(Weekday.Mon)

>>> print(Weekday['Mon'])

>>> print(Weekday.Tue.value)

2

>>> print(Weekday(1))

>>> for name, member in Weekday.\_\_members\_\_.items():

... print(name, '=>', member)

## 元类

# 错误、调试和测试

## 错误

### 捕获错误

try:

a=10/0

except ZeroDivisionError as e:

print('Except:', e)

except ValueError as e:

print('Except:', e)

else:

print('no error')

finally:

print('finally')

出现error时try代码块中后续代码不再运行，跳转到except错误处理语句块。但是还会继续执行try代码块外的语句

finally不管有错没错都会在最后执行，不是必须

可以有多个except

可以在except后加一个else，如果没有发生错误则执行

Python所有错误类型都继承自BaseException类

Python内置的logging模块可以非常容易地记录错误信息

import logging

except Exception as e:

logging.exception(e)

### 抛出错误

class MyBad(ValueError):

pass

def foo(n):

if(n==1):

raise MyBad('nihao')

return n

foo(1)

定义一个错误的class，需要继承一个错误类型

raise抛出一个错误

尽量使用python内置的

try:

foo('0')

except ValueError as e:

print('ValueError!')

raise

捕获了ValueError错误以后，如果当前函数不知道该如何处理，可以继续往上抛，让顶层调用者去处理

如果raise一个其他类型，则可以把一个错误类型转化为另一种错误类型。

如果不带参数，就把当前错误原样抛出

## 调试

### 断言assert

n=1

assert n==2, 'n is not 2'

如果n不等于2，则抛出AssertionError: n is not 2

python -O err.py

启动Python解释器时可以用-O参数来关闭assert

### logging

logging可以输出到控制台或者文件

import logging

logging.basicConfig(level=logging.INFO)

logging.debug('loggingdebug')

logging.info('loggingInfo')

logging.warning('loggingwarning')

logging.error('loggingerror')

四个等级，debug, info, warning, error

设置记录信息的特别为info，则debug信息不记录，不打印。同理可以设置为其他等级。默认warning

### pdb

python调试器

# IO

## 读

f=open('./test1.py', 'r')

print(f.read())

f.close()

read一次性读取全部内容

f.read(128)

每次最多读取128字节内容

f.readline()

读一行

f.readlines()

一次性读取所有内容，按行返回为一个list，每一行是一个str

str.strip()把每行末尾的\n去掉

with open('./test1.py', 'r') as f:

print(f.readlines())

由于open、read等方法经常报error，所以最好写在一个try/except中，最后在finally里写f.close()。用with可以简化写法，效果相当于try/except，而且自动调用close

open('./abc', 'r')

open('./abc', 'rb')

open('./abc', 'w')

open('./abc', 'wb')

分别为：读文本文件、读二进制文件、写文本文件、写二进制文件

f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')

读取非UTF-8编码文件

遇到有些编码不规范的文件，可能会遇到UnicodeDecodeError，因为在文本文件中可能夹杂了一些非法编码的字符。open()函数还接收一个errors参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')

while True:

s=f.readline()

if(s==''):

break

用readline读取文本文件

## 写

>>> f = open('./test.txt', 'w')

>>> f.write('Hello, world!')

>>> f.close()

调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件

test.txt可以不存在，open函数自动创建。如果存在，则删除原来的内容

写文件时往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用close()方法时，操作系统才保证把没有写入的数据全部写入磁盘。所以f1.write('nihao')以后立马f2=open('t.txt', 'r').read()读出来是空的

最好使用用with语句。就不需要写f.close()

with open('/Users/michael/test.txt', 'w') as f:

f.write('Hello, world!')

## StringIO

from io import StringIO

f=StringIO()

f.write('fefefe')

f.write(' ')

f.write('fefe')

print(f.getvalue())

fefefe fefe

StringIO在内存中读写str

可以用一个str初始化StringIO，然后像读文件一样读取

>>> from io import StringIO

>>> f = StringIO('Hello!\nHi!\nGoodbye!')

>>> while True:

... s = f.readline()

... if s == '':

... break

... print(s.strip())

## BytesIO

>>> from io import BytesIO

>>> f = BytesIO()

>>> f.write('中文'.encode('utf-8'))

6

>>> print(f.getvalue())

在内存中读写bytes

# 操作系统

## os模块

>>> import os

>>> os.name

nt

表示windows系统

文件分隔符

os.sep

环境变量

os.environ

获取某个环境变量的值可以调用os.environ.get('key')：

os.environ.get('PATH')

os.path.abspath('.')

'E:\\programs\\Python\\Python38'

返回路径 path 的绝对路径

'.'表示当前路径

Linux中使用/作为分隔符，Windows中使用\（字符串中用\\转义）

os.path.split('E:\\temp\\test python')

('E:\\temp', 'test python')

拆分路径

>>> os.path.basename('E:\\temp\\test python')

'test python'

相当于split返回的元组的第二个元素

>>> os.path.dirname('E:\\temp\\test python')

'E:\\temp'

相当于split返回的元组的第1个元素

os.path.splitext('E:\\temp\\test python\\t.py')

('E:\\temp\\test python\\t', '.py')

得到文件扩展名

os.path.join('E:\\temp\\test', 'test python')

'E:\\temp\\test\\test python'

os.path.join()函数拼接路径

如果join的几个参数中有一个从根路径，即以'/'开头或以盘符开头，则丢弃先前所有的部分重新拼接

os.path.join('E:', 'temp')

'E:temp'

需要注意的是在windows系统中第一个参数直接是驱动器名，则后面无法正常拼接。必需加上\\才能得到预期结果

os.path.join('E:\\', 'temp')

'E:\\temp'

## shutil模块

# 序列化

## 序列化pickling

import pickle

d={'name': 'nihao', 'gender': 'male', 'age':12}

print(pickle.dumps(d))

b'\x80\x04\x95,\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x04name\x94\x8c\x05nihao\x94\x8c\x06gender\x94\x8c\x04male\x94\x8c\x03age\x94K\x0cu.'

pickle.dumps()方法把任意对象序列化成一个bytes

f=open('t.txt', 'wb')

pickle.dump(d, f)

f.close()

pickle.dump()把对象序列化后写入一个文件

## 反序列化unpickling

f1=open('t.txt', 'rb')

c=f1.read()

c=pickle.loads(c)

print(c)

f1.close()

{'name': 'nihao', 'gender': 'male', 'age': 12}

先把文件里的二进制数据读到一个对象，用pickle.loads()函数反序列化出原来的dict

f1=open('t.txt', 'rb')

c=pickle.load(f1)

或者用pickle.load()函数直接读文件

# 进程/线程

## 进程

### fork

import os

print('Process (%s) started' % os.getpid())

pid = os.fork()

if pid == 0:

print('This is Child process %s. Parent process is %s.' % (os.getpid(), os.getppid()))

else:

print('This is parent process %s. Created child process %s.' % (os.getpid(), pid))

显示

Process (1628) started

This is parent process 1628. Created child process 1629.

This is Child process 1629. Parent process is 1628.

linux上可以运行以上代码。Windows上没有fork

fork把当前进程复制一份，调用一次返回两次。父进程返回新建的子进程的ID

子进程返回0

### multiprocessing

multiprocessing模块是跨平台版本的多进程模块。

multiprocessing提供了一个Process类来代表一个进程对象

from multiprocessing import Process

import os

def run\_proc(name):

print('Run child process %s (%s)' % (name, os.getpid()))

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

p = Process(target=run\_proc, args=('test',))

p.start()

p.join()

print('Child process end.')

执行结果如下：

Run child process test (8004)

Child process end.

通过传入执行函数和函数的参数，创建一个Process实例，start()方法启动，来创建子进程

join()方法等待子进程结束后再继续往下运行，通常用于进程间的同步。相当于close了

windows中可以运行上述代码

### 进程池

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

from multiprocessing import Pool

import os

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

p = Pool(4)

for i in range(5):

p.apply\_async(run\_proc, args=(i,))

p.close()

p.join()

对象调用join()方法会等待所有子进程执行完毕

调用join()之前必须先调用close()，调用close()之后就不能继续添加新的Process了。

Pool(4)最多同时执行4个进程

Pool的默认大小是CPU的核数

apply\_async同时创建4个进程并开始运行

### subprocess

fork只能复制出自身进程，很多时候，子进程并不是自身，而是一个外部进程。

创建子进程后，还需要控制子进程的输入和输出。subprocess模块可以解决

import subprocess

r = subprocess.call(['nslookup', 'www.python.org'])

print('Exit code:', r)

r = subprocess.call(['nslookup', 'www.python.org'])相当于控制台直接运行

nslookup www.python.org

nslookup是一个域名查询IP地址的命令，具体返回信息没太看懂

如果子进程还需要输入，则可以通过communicate()方法输入：

import subprocess

p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

output, err = p.communicate(b'set q=mx\npython.org\nexit\n')

print(output.decode('GBK'))

print('Exit code:', p.returncode)

上面的代码相当于在命令行执行命令nslookup，然后手动输入：

set q=mx

python.org

exit

运行结果依旧看不懂

也不太清楚stdin=subprocess.PIPE等等参数的意思

### Queue

进程间通信

multiprocessing模块提供了Queue、Pipes等多种方式用于进程间交换数据。

Queue相当于一个队列

from multiprocessing import Process, Queue

import os, time, random

def write(q):

for value in ['A', 'B', 'C']:

print('Put %s to queue' % value)

q.put(value)

time.sleep(random.random())

def read(q):

while True:

value = q.get(True)

print('Get %s from queue.' % value)

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

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

pw.start()

pr.start()

pw.join()

pr.terminate()

父进程创建Queue，并传给各个子进程

q.put(value) 子进程1写入数据到q中

q.get(True) 子进程2读取数据

pr进程里是死循环，无法等待其结束，只能pr.terminate()强行终止:

## 线程

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

import time, threading

def loop():

print('thread %s is running' % threading.current\_thread().name)

print('thread %s ended.' % threading.current\_thread().name)

t = threading.Thread(target=loop, name='LoopThread')

t.start()

t.join()

Python的标准库提供了两个模块：\_thread和threading，\_thread是低级模块，threading是高级模块，对\_thread进行了封装。绝大多数情况下使用threading

任何进程默认就会启动一个主线程MainThread，我们把该线程称为主线程

current\_thread()函数返回当前线程的实例

和process很像啊

创建线程时需要指定线程名

### Lock

多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响

多线程中，所有变量都由所有线程共享

因此，线程之间共享数据最大的危险在于多个线程同时改一个变量

x=0

for n in range(100000):

x=x+1

x=x-1

正常运行顺序：

temp1=x+1

x=temp1

temp2=x-1

x=temp2

100000次以后，x=0

多线程同时运行这段程序，有可能

temp1=x+1

temp2=x-1

x=temp2

x=temp1

100000次以后，x=100000

因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算x=x+1也分两步：

计算x+1，存入临时变量中；

将临时变量的值赋给x。

或者，一个进程修改x为x+1, 一个进程x=x\*2，只要同时运行，结果就是不确定的。当然这种情况下，加锁后结果也是不确定的

如果要保证运行结果的确定性，需要在线程里修改变量之前给修改变量的动作加锁。

其他线程想要执行这个动作，就得等待锁释放

无论多少线程，同一时刻最多只有一个线程持有这个锁

lock = threading.Lock()

def run\_thread(n):

for i in range(100000):

lock.acquire()

try:

change\_value()

finally:

lock.release()

threading.Lock()创建锁

lock.acquire获得锁

lock.release()释放锁。线程用完后一定要释放锁

一般用try...finally来确保锁一定会被释放。

锁也有坏处，可能死锁

一个线程使用全局变量必需加锁

### GIL锁

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：

Global Interpreter Lock

任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

### ThreadLocal

import threading

globalValueSet= threading.local()

def process\_value():

print('可以修改线程%s的变量%s' %(threading.current\_thread().name, globalValueSet.value))

def single\_thread(x):

globalValueSet.value=x

process\_value()

t1=threading.Thread(target=single\_thread, args=(123,), name="ThreadA")

t2=threading.Thread(target=single\_thread, args=('fff',), name="ThreadB")

t1.start()

t2.start()

t1.join()

t2.join()

执行结果：

可以修改线程ThreadA的变量123

可以修改线程ThreadB的变量fff

threading.local()创建ThreadLocal对象

globalValueSet.value=x给当前线程绑定一个变量

ThreadLocal需要是一个全局变量，这样每个Thread都访问到它，和它里边的属性，而且互不影响

ThreadLocal的每个属性（本例中为value）相当于是特定线程的局部变量

思来想去也不知道这样做和什么都不做的区别。就上边这个例子，和全程使用single\_thread()的参数x也没什么区别

# 正则

在正则表达式中

\d 匹配一个数字

\w 匹配一个字母或数字

\s 匹配一个空格

. 匹配任意单个字符

\* 匹配任意个字符（包括0个）

+ 至少一个字符

? 0个或1个字符

{n} n个字符

{n,m} n-m个字符

^表示行的开头，^\d表示必须以数字开头。

$表示行的结束，\d$表示必须以数字结束。

[0-9a-zA-Z\\_] 匹配一个数字、字母或者下划线；

[a-zA-Z\\_][0-9a-zA-Z\\_]\* 匹配Python合法的变量；

(P|p)ython 匹配'Python'或者'python'。

r'\d{3}\-\d{3,8}'匹配'010-12345'

'-'是特殊字符，在正则表达式中，要用'\'转义，

py也可以匹配'python'

^py$整行匹配，只能匹配'py'了

py不能匹配'apy'。也就是正则从第一个字符开始检查

正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符

强烈建议使用r前缀，表示这是一个正则表达式

>>> import re

>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

match()方法判断是否匹配，如果匹配成功，返回一个Match对象，否则返回None。

re.match(正则, 待检测字符串)

## 切分字符串

>>> 'a b c'.split(' ')

['a', 'b', '', '', 'c']

无法识别连续的空格

>>> re.split(r'\s+', 'a b c')

['a', 'b', 'c']

无论多少个空格都可以正常分割。

>>> re.split(r'[\s\,]+', 'a,b, c d')

['a', 'b', 'c', 'd']

>>> re.split(r'[\s\,\;]+', 'a,b;; c d')

['a', 'b', 'c', 'd']

## 分组

正则表达式还有提取子串的强大功能

用()表示的就是要提取的分组（Group）

^(\d{3})-(\d{3,8})$分别定义了两个组，可以直接从电话号中提取出区号和本地号码：

>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')

>>> m.group(0)

'010-12345'

>>> m.group(1)

'010'

>>> m.group(2)

'12345'

如果正则表达式中定义了组，就可以在Match对象上用group()方法提取出子串来。

group(0)是原始字符串，group(1)表示第一组

(())()测试得这样定义组，group(1)提取外层括号，group(2)提取内层括号，group(3)提取右边括号里的。所以group可以嵌套