# **python**

while 循环

代码示例：

while condition:

pass1

else: #else是condition条件为false时执行的代码块

pass2

count = 1

while count <=10:

count ++

print(count)

else:

print('EOF')

while的使用场景，设定condition为条件判断，condition条件判断为true时就一直执行语句块，条件判断为false时就执行else:后面的语句块

while 多用在递归

for 循环，主要用来遍历/循环 序列或集合或字典

for target\_list in expression\_list:

pass

示例：  
a = [['a','b','c'], (1,2,3)]

for x in a:

for y in x:

print(y,end=',')

print(y,end=',') #使打印出来的元素横向排列用“,”间隔

for else：  
a = [['a','b','c'],(1,2,3)]

for x in a

print(x)

else:

print('iterate done')

# else语句块在for循环结束后执行, 但如果for语句块内循环是被break中断的情况则else：语句块不执行

终止循环

a = [1,2,3]

for x in a:

if (x == 2):

break

print(x, end=',')

# x==2时跳出循环，整个过程只会打印出1

跳过循环

a = [1,2,3]

for x in a:

if(x==2):

continue

print(x,end = ',')

#x ==2 时跳过本次循环，整个打印结果是 1，3

for 循环的范围限定使用range()函数，类似JS for

for(let i=0;i<=a.length;i++){

console.log(i)

}

**range(num1,num2, length) 范围不包含num2 是指偏移量, length是指步长，步长可以为负数表示递减**

for x in range(0,10):

print(x)

#打印出0，1，2，3，4，5，6，7，8，9 共10个数字，偏移量是10

for x in range(0,10,2):  
print(x)

#打印出0，2，4，6，8 步长是2，每间隔两个数字打印一次

for x in range(10,2,-2):

print(x)

#打印出10，8，6，4，2，0

列表切片也能使用步长

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

b = a[0:len(a):2]

print(b)

#结果打印出 1，3，5，7

关于break,

break会中断for循环遍历，使for循环结束后的else不执行，但在循环嵌套的场景下，break只会中断所在层级的循环，对外层循环不会中断

python 工程组织结构，**包，模块，类**

python项目的组织结构，最顶级的层级包层级（类似文件夹），然后是模块层级（类似文件）、类。包的文件夹结构是可以嵌套的。

文件夹下如果有**\_\_init\_\_.py**文件则代表该文件夹是一个包。如果一个文件夹下没有\_\_init\_\_.py则python认为该文件夹是普通文件夹。\_\_init\_\_.py本质上就是一个模块文件，\_\_init\_\_.py文件可以写代码，也可以不写只是用来标注一个包。\_\_init\_\_.py模块的名称就是包的名称，比如seven目录下有\_\_init\_\_.py，则\_\_init\_\_.py模块的名称就是seven。\_\_init\_\_.py就是包的入口文件。

一个 .py 文件可以称为模块，包含了 \_\_init\_\_.py 文件的称为包。

包、模块的名称

命名空间，同名的模块用命名空间来区分

seven.c4, six.c4

seven,six是包名称，c4分别是包seven、six下面的模块名称

导入模块 import, from import

[1] import 方法

import module\_name

示例：  
import module\_name #导入同级目录下的模块文件

import t.module\_name #导入**当前目录下**t文件目录下面的module\_name模块文件

python的import无法像js那样导入js模块文件中的某个变量，只能导入整个模块文件，然后通过命名空间来访问模块下的变量。

import t.b.c7 **as m** # 导入目录t/b 下面的c7模块，并用m表示，调用c7模块下的变量b时使用m.b调用

导入其他目录的模块，使用绝对路径

import os  
path1=os.path.abspath('.') # 表示当前所处的文件夹的绝对路径  
print(path1)  
path2=os.path.abspath('..') # 表示当前所处的文件夹上一级文件夹的绝对路径  
print(path2)

import os,sys

parentdir = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

sys.path.insert(0,parentdir)

【2】 from import方法

from module import a,def

from t.c7 import a # 导入t.c7模块下的a变量，调用a变量时可直接调用

from t import c7# 导入t模块下的c7模块，调用a变量时使用t.a调用

from t.c7 import a, b, c #导入t.c7模块下的a,b,c三个变量

from t.c7 from (a, b

c) #导入t.c7模块下的a,b,c三个变量,c变量可以在括号内换行

import方法导入的是模块，from import 导入的可以是某个变量也可以是某个模块

from t.c7 import \* #导入t.c7模块下的所有变量

\_\_init\_\_.py的用法

导入一个包时，\_\_init\_\_.py文件会**自动执行**，

\_\_init\_\_.py文件的使用场景

【1】 模块重复批量导入

在包t下的\_\_init\_\_.py模块文件中集中编写需要多次导入的模块

import sys

import datetime

import io

在其他模块文件中导入包t即可导入包t文件下\_\_init.py模块文件中导入的变量/库

import t

print(t.sys.path)

1.包和模块时不会被重复导入的，多次引用同一个包或模块，但其导入过程只会执行一次。

2.当导入某个模块时，那个模块的文件就会自动执行一次

3.避免循环导入

// require  
const module = require('module')

// exports  
export.fs = fs  
module.exports = fs

// import  
import fs from 'fs';  
import { newFs as fs } from 'fs'; // ES6语法, 将fs重命名为newFs, 命名冲突时常用  
import { part } from fs;  
import fs, { part } from fs;

// export  
export default fs;  
export const fs;  
export function part;  
export { part1, part2 };  
export \* from 'fs';

函数参数

1. 必须参数。调用时传入的实际参数必须和形式参数个数一致且顺序一致

def add(x,y):

return x + y

pass1

1. 关键字参数，调用的时候可指定实参匹配哪个形参，**不用考虑传参的顺序**。前提是要知道函数定义时内部的形式参数名称。

js?

c = add(x=2, y=3)

1. 默认参数

def add(x=1. y):

return x + y

调用时 add(2) 传入的是，第一个参数**没传的时候**默认使用定义函数时的默认参数x=1，传入的是y=2

调用时 add(3,4) 传入的参数在函数内部是x=3 y=4

1. 可变参数，收敛传入的参数，转成tuple格式

def add(\*param):

print(param)

调用时 add(1,2,3,4,5), 打印param是(1,2,3,4,5) 会**自动将参数列表转换成成tuple，函数内部定义的可变参数param会自动收敛传入参数，并转成tuple**

调用时 add(**\*(1,2,3,4,5)**) 使用\*调用将传入的参数tuple平铺逐一匹配到函数内部的形式参数位置

js?

1. 关键字可变参数，收敛传入的关键字参数，转成dict格式

def add(\*\*param):

print(param)

调用函数时 add(x='1', y ='2', z='3') #**指定关键字参数调用时**，函数内部定义的可变参数param会自动收敛传入参数，并转成dict {'x':1,'y':2,'z':3}

或者以传入字典的方式调用

add(**\*\*{'x':1, 'y':2，z:'3' }**) #传入字典就需要用\*\*平铺

js? 解构赋值

for key, value in target.items():

注意：**声明函数时，必须参数靠前**，有默认值的默认参数尽量放后面。

**变量作用域**

函数内部定义的变量（局部变量）作用域在函数内部，函数外部无法引用函数内部变量（闭包除外，还可以使用global关键字变量在函数内部定义全局变量）。但函数内部可以引用全局变量即函数外部定义的变量，或者嵌套函数，其内部变量可以引用外层变量。

c=10

demo():

print(c) #打印出10

注意：python只有全局作用域、函数作用域的概念，python没有代码块级的块作用域的概念(for循环，while循环，if..else等非函数不能形成独立的块作用域)，所以python for循环中声明的变量，在for循环外部是可以访问的，不同于js

global 在函数内部定义全局变量

def demo():

global c

c = 2

执行demo() 函数后在其内部会执行声明全局变量

print(c) #在函数外部就可以访问到函数内部使用global定义的全局变量

**某个模块中函数内部global关键字定义的全局变量也是可以被其他模块导入使用的**

python的作用域链，

python面向对象

**类 class 首字母大写，驼峰命名**

定义类

class Student():

name=''

aget=0

def print\_fn(**self**): #类似js的this，需要显式的传入

print(**self.**name)

stu1 = Student() # 实例化类，**不需要加new**关键字

stu1.print\_fn() #调用类的方法

导入类

from c1 import Student()

student = Student()

student.print\_file()

类和对象的关系

类是现实世界或思维世界中的实体在计算机中的一种抽象反映，将抽象出的实体，其**属性**和对属性的操作即**方法**封装在一起。简单理解，类抽象出**实体的特征与行为，类基本可以理解为模板**

对象，即通过**类实例化出的一个实力**。

class Studen():

name= ‘default value’

age= 0

def \_\_init\_\_(**self, name, age**): #python类的构造函数，js的constructor函数

self.name=name #通过构造函数的执行来初始化对象的属性

self.age=age #通过构造函数的执行来初始化对象的属性

实例化一个实例时，类的构造函数会自动执行。**构造函数或实例方法内的self始终指向实例**。

类变量、实例变量

类变量即**类的静态属性**，实例变量即**类实例化后的属性**，实例化后生成对象的属性。

class Student():

name='' **#类变量**

age=0 **#类变量**

def \_\_init\_\_(**self**, name, age): #**self只是占位符，可以用this替代**

self.name=name **#实例变量**

self.age=age **#实例变量**

def fn1(**self**): # **定义函数的时候参数传入self，表示该函数是实例的方法**

print(self.name)

stu1 = Student()

stu1**.\_\_dict\_\_** #对象（包含类或者实例）的\_\_dict\_\_对象中保存了当前对象的所有变量，类似js .prototype

注意：类的静态属性（类变量）和实例属性（实例变量）的查找顺序，首先在实例属性中查找，如果找不到就去类的惊天属性中查找，如果类中找不到就会去类的父类中查找。**类似JS的原型对象，JS中的类有静态属性吗？**

在实例方法中访问实例变量与类变量

实例方法中访问实例变量，**self.val**

实例方法中访问类变量：**ClassName.val 或者 self.\_\_class\_\_.val**

类方法即类的静态方法，作用

**@classmethod #定义类的静态方法时增加@classmethod装饰器**

def fn(**cls**): #传入的参数cls表示class，可以使用其他变量代替

cls.val #访问类的属性，和实例方法中self.\_\_class\_\_.val 等价

类的方法中访问类的属性，cls.val 等价与self.\_\_class\_\_.val 等价与 ClassName.val

调用类方法ClassName.fn()

通过类的实例调用类的方法(不建议这么调用没意义)：

ins1 = ClassName()

ins1.\_\_class\_\_.fn()

js的静态方法 static

成员的可见性

对类的变量即类的静态属性的操作，放到类的静态方法，或者实例方法中去操作。不建议在外部直接操作类的属性比如 ClassName.val = 1 这种操作不建议

类的私有方法，在一个类中定义的方法(类方法或者实例方法)名称前面加**\_\_** 表示该方法为类的私有方法，私有方法在类的外部无法调用，比如ClassName.fn()。

私有变量，在变量名前加\_\_ 该变量就成为私有变量，私有变量只能在对象内部访问，无法在外部访问比如obj.a

python继承

#c6文件定义Human类

class Human():

sum = 0

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

self.name=name

self.age=age

def get\_name(self):

print(self.name)

from c6 import **Human**

class Student(**Human**): **#类的参数用来传入父类**

def \_\_init\_\_(self, school, **name, age**):

self.school = school #子类定义的实例属性

**super(Student, self).\_\_init\_\_(name, age)**

**#Human.\_\_init\_\_**(**self,** **name, age**) **#子类中传入父类定义的参数执行父类的构造函数**

self.\_\_score = 0

self.\_\_class\_\_.sum += 1

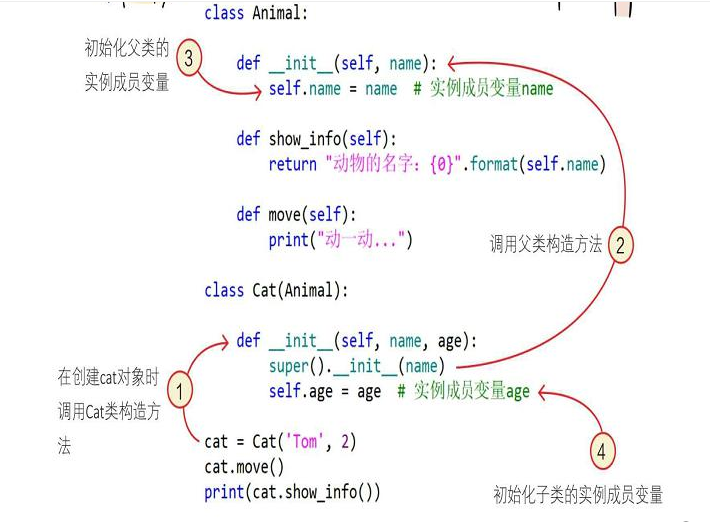
def do\_homework(self):

**super(Student,self).get\_name()** #spuer不仅可以执行父类的构造函数，还能在子类中执行父类的实例方法

print('do homework')

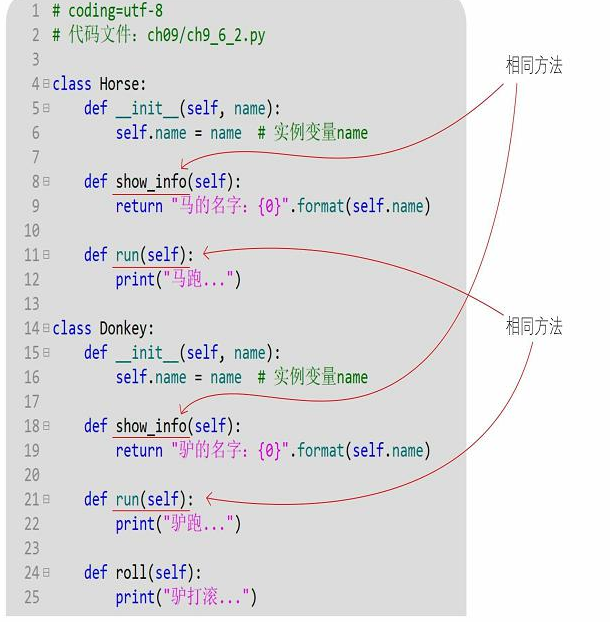
为了让项目结构更清晰，建议一个模块只写一个类

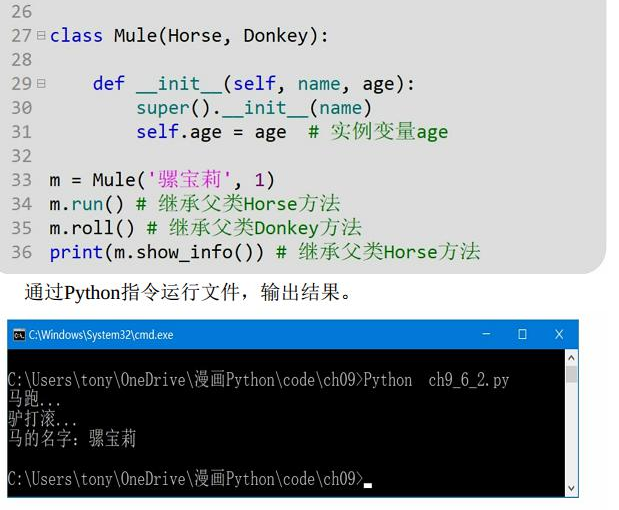
子类继承父类的过程



多继承

子类继承多个父类时，如果多个父类有相同的成员方法或成员变量，则子类优先继承左边的父类的成员方法或变量，继承时传入的父类参数从左到右其继承级从高到低。当子类方法和父类方法同名时，子类方法会重写overwrite父类方法





异常处理 try...except...finally

出现异常后捕获异常，加以处理，不让程序终止并退出。

常见的异常有：Exception, Error, warning，**error类型继承自Exception类**

try:

pass #可能会出现错误的代码

except **异常类型 as e**: #**e是异常变量**

pass #出现异常后的处理代码

finally:

pass **#try、except代码块执行完之后，最终都会执行的代码块**

except语句可指定异常类型，如果不指定则except语句捕获try语句中的所有异常，当指定具体异常时则except只捕获try中特定的异常。多个异常类型的捕获

finally代码块中常用来释放资源，比如打开的文件、网络连接、打开数据库的连接、及数据结果集都会占用计算机资源，再finally中释放资源

try:

pass #可能会出现错误的代码

except 异常类型1 as e:

pass

except 异常类型2 as e:

pass

except (异常类型3, 异常类型4) as e **#多个异常类型相似可以合并**

except: **#没指定异常类型，默认捕获上面未捕获到的异常**

pass

finally:

pass #释放资源

try...except 可以嵌套，但实际开发中尽量不要嵌套，应梳理好程序执行流程再考虑是否需要try...except嵌套

自定义异常，继承exception类

class customException(**Exception**): 继承Exception类生成新的customeException异常类

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

super().\_\_init\_\_(message)

触发自定义异常

raise customException('这是一个自定义的异常信息')

正则匹配规则re模块

import re

re.findall('正则规则'，目标字符串) #查找出与匹配规则的结果，返回结果是数组

【1】元字符

re.findall('**\d**', a) #查找0-9的数字

\d、[0-9] #匹配0-9数字

‘\D’ 、[^0-9] #匹配0-9非数字

'\w'、'[A-Za-z0-9\_]' #匹配单词字符，即数字和字母和下划线

''\W' #匹配非单词字符符号，包含空白字符符号比如空格，回车，制表符等

'\s' #匹配空白字符符号比如空格，回车，制表符

'\S' #匹配非空白字符

'[\s\S]\*?' #匹配除所有字符

'.' #匹配除换行符\n之外的其他所有字符

‘a[**cf**]c’ #查找a开头，c结尾，中间是**c或f**的结果

‘a[**^cf**]c’ #查找a开头，c结尾，中间不是**c或f**的结果

‘a[**c-f**]c’ #查找a开头，c结尾，中间是**c到f之间**的结果

【2】数量词

'[a-z]{3,6}' #匹配a-z之间的字符，最少匹配3位，最多匹配6位，贪婪模式匹配

'[a-z]{3,6}**?**' #匹配a-z之间的字符，最少匹配3位，最多匹配6位，**非贪婪模式**匹配3位即可

'pytho**\***' #匹配pytho**0次**或**无限多次**

'pytho**+**' #匹配pytho**1次**或**无限多次**

'pytho**?**' #匹配pytho**0次**或**1次**

【3】边界匹配

'^\d{3,8}$' #^即字符串开始位置，$即字符串结束位置。数字开头，数字结尾，且长度在3-8位之间

【4】组

r = re.findall('(Python){3}', s)

(Python){3} #将Python作为完整的一个分组，判断python是否重复3次

Python{3} #只判断Python中的单一字符n是否重复3次

小括号**(abc)**分组后，括号中的字符是且关系，中括号**[abc]**中的字符是或关系

【5】匹配模式参数，findall的第三个参数

r = re.findall('正则规则', 目标字符串, re.I | re.S) # re.I即不区分大小写，re.S改变元字符**.**的匹配规则。多个模式用管道符连接，是且关系

r = re.findall('c#', 'PythonC#JavaPHP', re.I)

r = re.findall('c#**.{1}**', 'PythonC#**\n**JavaPHP', re.I | re.S) #目标字符串里面有换行，元字符.是匹配除换行外所有字符，模式参数加上了re.S 就是改变元字符.的匹配特性，匹配包括换行符在内的所有字符。

【6】正则替换，返回结果是字符串

r = re.sub('C#', 'GO', 'PythonC#JavaPHP',0) #第二个参数是要替换成的字符串，第4个参数count设置为0是不限制替换次数

第2个参数可以定义成函数，实现更复杂的需求，**比如第一个参数定义的匹配规则匹配出的是变量的时候**

def convert(value):

mached = value.group() #value**是匹配到正则结果的match对象**

return 'AA' + mached + 'AA'

r = re.sub('C#', convert, 'PythonC#JavaPHP', 0) #第1个参数匹配到之后会传入到convert函数中（convert函数中定义的value参数就是匹配上规则的一组match对象），返回函数执行的内容

示例：

def fn(value):

matched = value**.group()**

if(int(matched)>=6):

return '9' #这里必须返回字符串，因为正则只能操作字符串

else:

return '0'

re.sub('\d', fn, 'A8C3721D86', 0)

函数作为参数传入另一个函数，就是高阶函数

【7】re模块中除了findall之外的其他两个函数search, match，，只匹配一次，**返回结果是match对象**

r = re.match('\d', 'A83C7') #返回结果为None，因为match函数会从目标字符串的**首字母**开始匹配，如果首字母没匹配到就返回None

r = re.search('\d', 'A83C7') #search函数会搜索整个字符串，直到找到第一个匹配的结果就会**返回match对象**

【8】group方法,，match对象需要使用group()方法来获取具体的结果

r=re.search('life**(.\*)**python**(.\*)**python', 'life is short, i use python, i love python')

r.group(0) #获取所有分组匹配内容，即返回完整的匹配结果 'life is short, i use pyhon, i love python'

r.group(1) #获取第一个分组(.\*)匹配的内容，即'is short, i use '

r.group(2) #获取第二个分组(.\*)匹配的内容，即', i love'

r.group(0,1,2) #获取指定分组已元组形式返回 ('life is short, i use pyhon, i love python', 'is short, i use ',', i love')

groups()方法

m = re.match(r"(\d+)\.(\d+)", "24.1632")

m.groups() #返回('24','1632')

JSON是轻量级的数据交换格式

JSON是数据格式，与之对应的是XML数据格式

字符串是JSON的表现形式，符合JSON格式的字符串就是JSON

python中JSON和字典格式一样，在JS中JSON和对象格式一样，在每种语言中都有特定的数据格式与JSON对应从而实现转换。

1. JSON反序列化，转换json格式字符串为JSON必须是标准格式的JSON字符串'{"name":"abc"}'

json转字典

import json

ditc1 = json.loads(jsonStr) #JS中JSON反序列化为对象。JSON.parse(jsonString)

JSON 对象数组也能反序列化为python list

list1 = json.loads('[{"name":"a1"},{"name2":"a2"}]') #JS中JSON反序列化为数组，JSON.parse(jsonString)

**JSON数据格式中的boolean值是小写**

|  |  |
| --- | --- |
| **json** | **python** |
| object | dict |
| array | list |
| string | str |
| number | int |
| number | float |
| true | True |
| false | False |
| null | None |

1. JSON序列化，转换JSON数据格式为JSON格式字符串

import json

jsonStr = json.dumps(jsonObject)

JSON：一种通用的数据交换格式，支持各种语言。

JSON字符串：符合标准JSON语法格式的字符串

JSON对象：只在JS语言中存在JSON对象，在python中是dict，或list dict

枚举

枚举其实是一个类，枚举不可实例化。枚举就是列举出有穷集合的所有元素，枚举的成员**不可修改**，且枚举类型中**没有重复成员名称name，但成员名称对应的值可以相同，后面相同值对应成员名称name相当于是别名**

from enum import Enum

class VIP(Enum): #继承Enum父类生成一个子类VIP，这里不同于普通的类，普通类的属性是可修改的

YELLOW=1

GREEN=2

abc =2 #因为枚举值相等，所以后面的abc相当于是前面GREEN的别名

VIP.YELLOW =2 #会提示报错，因为枚举类型不能修改

VIP.YELLOW #返回VIP.YELLOW 获取的是一个**枚举类型**即<enum 'VIP'>，不是具体的值

VIP.YELLOW.value #获取枚举类型中标签对应的**枚举值**

VIP.YELLOW.name #获取枚举类型中**枚举名称**

**遍历枚举**

for v in VIP:

print(v) #返回VIP类中的所有的枚举类型

枚举的比较运算，**枚举类型之间**只能判断是否相等，是否同一身份比较is，不能判断大小关系，

VIP.YELLOW == VIP.YELLOW #返回True

VIP.YELLOW == VIP.GREEN #返回False

VIP.YELLOW == 1 #会返回False，不是枚举类型之间比较

VIP.YELLOW is VIP.GREEN #返回True

**枚举转换**

当已知某个值，要根据该值匹配出某个枚举类中的成员

a = 1

VIP(a) #返回的是VIP.YELLOW 枚举类型

python进阶知识

**1.一切皆对象**

a=1

a='2'

a = fn

python中能将函数作为参数传入到另一个函数里，也可以把函数作为另一个函数的返回结果。

**闭包**

函数及函数声明时所在的环境变量的集合叫闭包。可在函数外部调用函数内部的局部变量。闭包可以保存函数的环境变量，避免函数执行后环境变量被垃圾回收。

查看某个函数的闭包使用：**\_\_closure\_\_[0].cell\_contents**

def curve\_pre():

a= 25

def fn(x):

return a \* x

return fn

f = curve\_pre()

f.\_\_closure\_\_[0].cell\_contents

记录行走步数示例：

def save\_step(x):

origin = x

def walk\_step(step):

**nonlocal** origin

origin += step

print(origin)

return origin

return walk\_step

fn = save\_step(0)

fn(2)

print(fn.\_\_closure\_\_[0].cell\_contents)

fn(6)

print(fn.\_\_closure\_\_[0].cell\_contents)

fn(12)

print(fn.\_\_closure\_\_[0].cell\_contents)

**lambda表达式**

定义普通函数

def add(x,y):

return x + y

**lambda**表达式**定义匿名函数**

lambda parameter\_list: expression **#lambda表达式的后面只能是一个简单的表达式不能是复杂的代码块，返回值不需要使用return**

lambda x,y: x+y

匿名函数的调用

f = lambda x,y: x+y

f(1,2)

(lambda x,y: x+y)(1,2)

**三元表达式**

**条件为真时的返回结果** if 条件判断 else **条件为假时的返回结果**

result = x if x> y else y #返回x必须满足条件为x>y，否则返回y

**函数式编程**

**1.map函数，返回一个map类，使用list(map\_x) 转成列表**

list\_x = [1,2,3,4,5,6,7,8]

list\_y = [1,2,3,4,5,6]

方法一

for x in list\_x:

square(x)

方法二

def square(x):

return x \* x

r= map(square, list\_x) #返回结果【1，4，9，16，25，36，49，64】

list(r)

使用lambda表达式定义匿名函数，**简化代码**

r=map(lambda x: x\*x, list\_x)

r=map(lambda x, y: x\*x + y, list\_x, list\_y)

list(r)

**2.reduce 每次的计算结果作为下一次计算的参数**

from functools import reduce

list\_x = [1,2,3,4,5,6,7,8]

r = reduce(lambda x,y:x+y, list\_x, 10) #reduce函数的第三个参数10是初始值

**3.filter，返回一个过滤后的集合，结果使用list(filterx)转成列表**

list\_x = [1,0,1,0,0,1]

r = filter(lambda x: **True** if x ==1 else **False**, list\_x) #挑选出x为1的元素

#filter函数中的lambda表达式的返回结果必须为布尔值或者0，1类布尔值，必须是能返回真假的表达式。

list(r)

**函数式编程与命令式编程**

1. 命令式编程就是编写控制过程，包含函数，条件判断， 循环
2. 函数式编程是使用map, reduce, filter, lambda表达式替换函数，循环

**装饰器**

开闭原则：**对修改是封闭的，对扩展是开放的。**不改变原有函数的基础上，增加新的功能

使用场景：

【1】当需要对某一个封装的单元比如某个函数做出修改，可以在不修改原有函数的基础上使用装饰器的形式扩展原有函数的功能进而达到间接修改原有函数。

【2】如果需要复用某个函数的功能

示例：打印函数的同时，打印当前的时间戳。

import time

def f1():

print('This is fn1')

def f2():

print('This is fn2')

def print\_current\_time(fnc):

print(time.time())

fnc()

print\_current\_time(f1)

print\_current\_time(f2)

#以上的方式让函数f1、f2和print\_current\_time没有具体的关联性仍然是独立的个体，没有内聚性。

使用装饰器解决问题的思路，不改变原有函数的基础上，增加新功能

import time

def decorator(**func**):

def wrapper(**\*args, \*\*kw**): #\*args表示不确定的多个参数, \*\*kw表示关键字参数

print(time.time())

**func(\*args, \*\*kw) #\*args收敛可变参数，\*\*kw收敛关键字参数**

return wrapper

**@decorator** #不改变原有函数，增加新功能，代码没有改变f1的内容

def f1(fun\_name):

print('This is a function1'+ fun\_name)

f1('a')

**@decorator**

def f2(fun\_name1, fun\_name2):

print('This is a function2' + fun\_nam1)

print('This is a function2' + fun\_nam2)

f2('a','b')

**@decorator**

def f3(fun\_nam1, fun\_nam2, **\*\*kw**):

print('This is a function2' + fun\_nam1)

print('This is a function2' + fun\_nam2)

print(**kw**)

f3('a','b',a=1,b=2,c='123')

装饰器的副作用：**使用装饰器后原有函数的名称会改变为wrapper，解决增加装饰器后函数名称改变的方法**

from functools import wraps

def my\_decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kw):

func()

return func(\*args, \*\*kw)

return wrapper

@my\_decorator

def example():

"""this is fn doc"""

print('this is fn')

example.\_\_name\_\_ #函数的名称仍然是example,

example.\_\_doc\_\_#函数的注释还是原函数的注释

**多线程**

程序在运行时一次智能执行一个任务（单线程），让程序同时执行多个任务就要使用多线程技术。

1.程序进入执行状态后就是一个进程，每个进程有自己的独立的内存空间、系统资源，每一个进程的内部数据和状态都是完全独立的。windows中的进程局势exe或者dll程序，进程之间相互独立也可以通信。

2.一个进程中可以包含多个线程，多个线程共享一块内存空间和一组系统资源，所以系统在各线程之间切换时，系统开销比进程小得多，因此线程称为轻量级进程。

3.python程序至少有一个线程即主线程，python程序启动后由python解释器负责创建主线程，程序结束后由python解释器停止主线程。多线程中，主线程负责其他子线程的调度，启动、挂起、停止等。

多线程编程时，需要给每个子线程执行分配机会，通过让当前子线程休眠暂停（**延迟当前子线程的后续执行**）执行，让其他线程有机会执行。如果当前子线程没有休眠，只能等待当前线程执行后再执行第二个线程。

多线程可以把空闲时间利用起来，比如有两个进程函数 func1、func2，func1函数里使用sleep休眠一定时间，如果使用单线程调用这两个函数，那么会顺序执行这两个函数，也就是直到第一个函数执行完后，才会执行第二个函数，这样需要很长时间；如果使用多线程，会发现这两个函数是同时执行的，这是因为多线程会把空闲的时间利用起来，在第一个函数休眠的函数就开始执行第二个函数

python多线程使用场景：如果程序时cpu密集型的，使用python的多线程是无法提升效率的，如果程序时IO密集型的，使用python多线程可以提高程序的整体效率。

CPU密集型（CPU-bound）：CPU密集型也叫计算密集型，指的是系统的硬盘、内存性能相对CPU要好很多，此时，系统运作大部分的状况是CPU Loading 100%，CPU要读/写I/O(硬盘/内存)，I/O在很短的时间就可以完成，而CPU还有许多运算要处理，CPU Loading很高。

IO密集型（I/O bound）：IO密集型指的是系统的CPU性能相对硬盘、内存要好很多，此时，系统运作，大部分的状况是CPU在等I/O (硬盘/内存) 的读/写操作，此时CPU Loading并不高，I/O bound的程序一般在达到性能极限时，CPU占用率仍然较低。这可能是因为任务本身需要大量I/O操作，而pipeline做得不是很好，没有充分利用处理器能力

多线程的应用有很多，一些阻塞主线程的操作应该被放到子线程中处理，比如打开文件，网络爬虫。多线程会产生并发问题，多个线程如果同时读取某个变量导致相互干扰产生并发问题，所以实际开发中尽量避免多个线程读取或写入相同的变量。

import \_thread as thread  
from time import sleep, ctime  
  
def fun1():  
 print('开始运行func1', ctime())  
 # 休眠4秒  
 sleep(4)  
 print('func1运行结束', ctime())  
def fun2():  
 print('开始运行func2', ctime())  
 # 休眠4秒  
 sleep(2)  
 print('func2运行结束', ctime())  
def main():  
 print('开始运行时间', ctime())  
 # 启动一个线程运行func1函数  
 thread.start\_new\_thread(fun1, ())  
 thread.start\_new\_thread(fun2, ())  
 # 休眠6秒  
 sleep(6)  
 print('运行结束时间', ctime())  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()  
  
E:\python\python.exe E:/progect/untitled1/untitled1/urls.py  
开始运行时间 Sat Feb 16 09:34:00 2019  
开始运行func1 Sat Feb 16 09:34:00 2019  
开始运行func2 Sat Feb 16 09:34:00 2019  
func2运行结束 Sat Feb 16 09:34:02 2019  
func1运行结束 Sat Feb 16 09:34:04 2019  
运行结束时间 Sat Feb 16 09:34:06 2019

线程模块 threading ，线程类Thread

线程模块相关函数：

avtive\_count() 返回当前处于活动状态的线程个数

current\_thread() 返回当前的Thread对象

main\_thread() 返回主线程对象，主线程是python解释器启动的线程

import threading

threading.current\_thread()

threading.active\_count()

threading.main\_thread()

创建子线程

1. 线程对象，由threading模块的Thread类或Thread的子类构建的对象
2. 线程体，即子线程要执行的代码，通常封装到一个函数中。子线程启动后会执行线程提。

实现线程体有以下两种方式

1.自定义函数中实现线程体

Thread(target=fnName, name='threadname', args=[x1,x2])

target参数指向自定义的线程体函数

name参数可自定义线程名称

args为线程体函数提供的参数，列表类型

示例

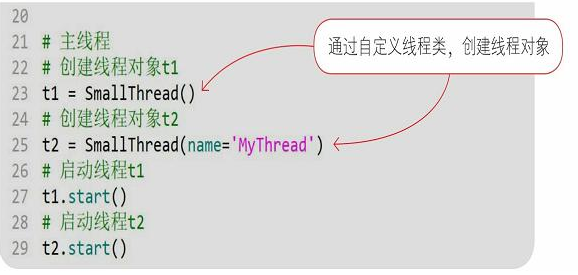
import threading

t1 = threading.Thread(target=Fn1, name ='myThread')

t1.start()

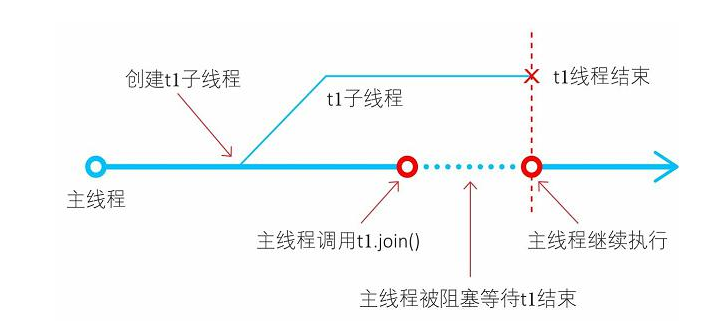
2.自定义线程类实现线程体，run()方法就是线程体函数



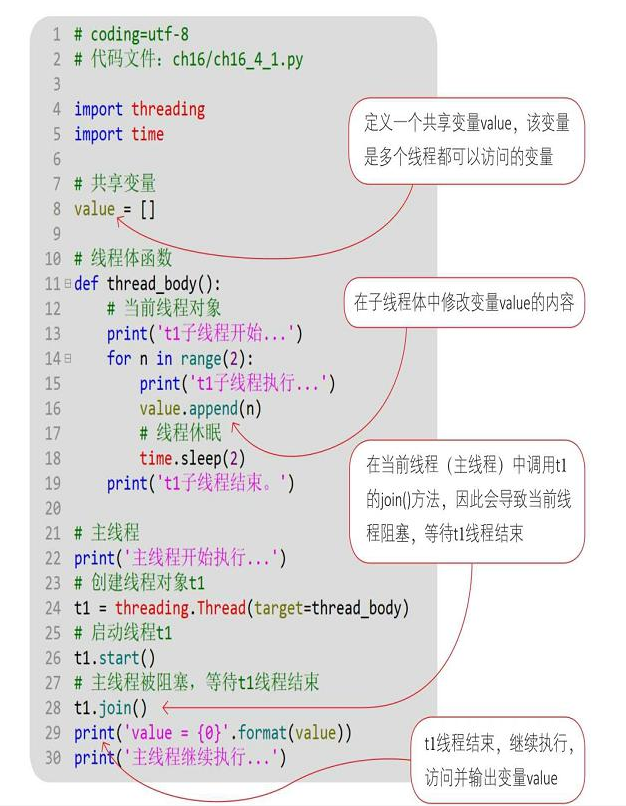


线程管理

1. 等待线程结束，某些场景可以控制主线程等待另一个子线程t1执行结束后才能继续执行

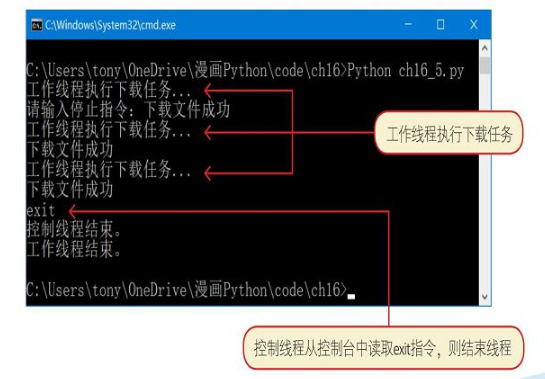


join(timeout= None) #设置超时时间单位秒，不设置默认一直等待，调用join()方法让主进程阻塞，等待t1子进程执行完毕后再继续执行。



2.





**pythonic**

1. 使用字典代替switch case语句，python没有switch case

def get\_sunday:

return 'Sunday'

def get\_monday:

return 'Monday'

def get\_tuesday:

return 'Tuesday'

def get\_default:

return 'Unknow'

dict1 = {

0: get\_sunday,

1: get\_monday,

2: get\_tuesday

}

day = 2

dict1.get(day, get\_default)() #当获取不到指定day所对应的值时，默认返回get\_default

1. 列表推导式 根据已经存在的列表创建一个新的列表

a = [1, 3, 5, 7, 9]

b = [for i in a] #返回一个新的列表

b1 = [**i\*\*2** for i in a] #返回a列表每个元素的**平方**，生成新的列表

条件筛选推到出列表

c = [i\*\*2 for i in a **if i>=5**] #筛选出原列表中大于等于5的元素的平方，生成新的列表

列表、元组、集合都可以被推导

a = {1,2,3,4,5,6,7,8}

bb = (1,2,3,4,5,6,7,8)

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

a1 = **[**i\*\*2 for i in a**]** #集合推导出列表

a2 = **{**i\*\*2 for i in a**}** #集合推导出集合

a3 = **[**i\*\*2 for i in bb**]** #元组推导出列表

a4 = **{**i\*\*2 for i in cc**}** #列表推导出集合

字典推导出列表

students = {

'name1': 18,

'name2': 22,

'name3': 24

}

b= **[**key for key,value in students.items()**]** #将原字典的key提取推导出新的列表

b2= **{**key:value for key,value in students.items()**}**

1. **iterator 与generator 迭代器与生成器**

iterator迭代器

可迭代对象iterable：凡是可以被for...in循环遍历的对象就是可迭代对象。比如列表、元组、集合都具有可迭代接口。

迭代器iterator即迭代接口，是一个可迭代的对象，但可迭代对象(列表、元组、集合)不一定是迭代器。

迭代器有两个内置方法\_\_iter\_\_，\_\_next\_\_，迭代器可以使用next函数控制迭代过程，迭代器是一次性的，第一次遍历完毕后就无法再遍历了。

可迭代对象可以多次遍历，没有next函数控制迭代过程。

使用迭代器创建一个可迭代的class类

class BookCollection:

def \_\_init(self):

self.data = ['往事','只能','回味']

self.cur = 0 #定义游标

pass

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

return self

def \_\_next\_\_(self):

if self.cur >= len(self.data):

**raise StopIteration()** #如果游标值超过data列表长度即数据已全部遍历则抛出异常（迭代完成）

result = self.data[self.cur]

self.cur += 1

return result

books= BookCollection()

#迭代器可使用for...in遍历

for boo in books:

print(book)

#通过next函数控制迭代过程

print(next(books))

print(next(books))

print(next(books))

生成器函数generator

n = [i for i in range(0,10001)]

def gen(max):

n = 0

while n <=max:

n += 1

yield n

g = gen(10000)

print(next(g))

print(next(g))

print(next(g))

for i in g:

print(i)

4.None，None是单独的对象类型NoneType，表示此处有值但为空。 JS null值

判断None 操作

if var1 is None:

if var1： 或者 if not var1：

类在做真假值判断的时候，会调用bool()方法将类转换成bool类型，类其定义内置的\_\_len\_\_(self)、\_\_bool\_\_(self)方法会作为判断真假值的依据。

class Test():

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

return False

bool(Test()) #此时判断为False

判断一个类型是否存在

a= Test()

if a:

pass

5. 海象运算符，python 3.8版本以上支持

:=

**python爬虫**

**爬虫的基本原理**

模拟http请求，向服务器发送请求，获取服务器返回的html，用正则表达式提取所需要的数据。

1. 明确目的，需要哪些数据
2. 找到数据对应的网页
3. 分析网页的结构找到数据所在的标签位置

HTML结构分析基本原则

1. 尽量选择贴近目标数据的标签。
2. 尽量选择具有唯一标识的标签来定位目标数据。
3. 尽量选择成对闭合标签。

爬虫库：beautiful soup，scrapy

from urllib import request

oop思想

封装、继承和多态是面向对象的三大特征。这三大特征与语言本身无关，这是一种面向对象的编程思想。

封装是为了提高程序的安全性；将数据（属性）和行为（方法）包装到类对象中。而在方法内部对属性进行操作，在类对象的外部调用方法。这样我们只需要用合适的方式去用它，而不用关心刚方法时如何具体实现的。

继承是为了提高代码的复用性；继承就是定义子类，子类继承父类的属性和方法。一个类没有继承任何类，默认继承object；Python支持多继承；

多态是为了提高程序的可扩展性和可维护性。Python中的多态指的是，在运行过程中根据变量所引用对象的类型，动态的决定调用哪个类对象中的方法。

class Student:  
 provice\_code = '0717'  
  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.\_\_age = age #前边加两\_表示不希望再类外部被访问  
  
 def show(self):  
 print(self.name, self.\_\_age)  
  
  
stu = Student('张', 15)  
print(stu.name)  
print(stu.age) #外部访问私有变量会提示错误，但通过stu.\_Student\_\_age可访问到  
stu.show()

Object类是所有类的父类，一个类没有继承任何类，默认继承object；

类的浅拷贝,Python拷贝一般都是浅拷贝，拷贝时，对象包含的子对象内容不拷贝，因此，源对象与拷贝对象会引用同一个子对象

import copy

a2 = copy.copy(a1)

类的深拷贝,使用copy模块的deepcopy函数，递归拷贝对象中包含的子对象，源对象和拷贝对象所有的子对象也不相同。

import copy

a3 = copy.deepcopy(a1)

**python 数据读写**

csv是有分隔符的text文件

路径的书写三种方式

path1 =C:\\Users\\Administrator\\Downloads\\text.txt

path2 = C:/Users/Administrator/Downloads\\text.txt

path = r'C:\\Users\\Administrator\\Downloads\\text.txt'

处理文件路径模块os

import os

os.name #系统类型

os.getcwd() #当前运行脚本的绝对路径

os.listdir() #返回当前脚本所在目录下的所有文件

os.chdir('c:\\abc') #切换路径到c:\abc下

os.remove('c:\\text.txt') #删除文件

os.rmdir('c:\\abc') #删除文件夹

os.path.split('c:\\user\\text.txt') #将文件路径转成元素tuple('c:\\user', 'text.txt')

os.path.exists('c:\\user\\text.txt') #判断文件是否存在，也可以判断文件夹是否存在

读取文件

open(路径, 模式, 编码格式),

模式r读取，w写入，rw读写，a追加，w+打开并清空，wb 二进制模式写入，

file = open(path1, 'r', encoding='utf8')

file.read(10) #读取10个字符

file.readline() #读取行

file.readline(5) #读取行的前5个字符

**file.readlines()** #读取所有行，并转成列表

**file.seek(0)** #上一次读取完后将光标移动到开始位置，便于下次读取

file.close()

写入文件（只能写入字符串）

file = open('text.txt', 'w', encoding='utf8')

file.write('abcd')

file.close()

file.writelines(list) #列表**list的成员依次传入writelines方法**，水平书写，如果希望换行书写则遍历list在每个成员后 + '\n'

pickle存储

可将数据处理过程中的任何类型数据格式保存为系统中的dump快照文件，**不像写入txt文件时会将原来数据格式转换为字符串**，pickle模块保存的dump快照文件保留的原有的数据类型。

#将数据保存为快照文件

import pickle

data = {'a':1,'b':[1,2,3], 'c':'hello world'}

dump\_file = open('data.pkl', 'wb') #创建文件名为data.pkl的快照文件，写入模式

**pickle.dump**(data, dump\_file)

dump\_file.close()

#读取快照文件，还原数据

dump\_file = open('c:\\user\\data.pkl', 'rb')

dt1 = pickle.load(dump\_file)

print(dt1)

Numpy科学计算工具包

强大的N维数组对象，重点对二维数组结果进行运算（不用遍历循环），包含随机数，线性代数，傅里叶变化等功能。

import numpy as np

ar = np.array([[1,2,3],[2,3,4]])

np.ndim #数组维度

一维数组：单行

二维数组：多行多列

三维数组：多个二维数组构成一个三维数组

四维数组：多个三维数组构成一个四维数组

创建数组时指定的行列个数不一样时，会自动转换成一维数组

numpy常用方法

ar = numpy.array([1,2,3]) #生成数组

ar.ndim #数组维度的个数，轴数，或者说秧，维度的数量也称rank

ar.shape #数组的行列个数

ar.size #数组元素的总个数，n行m列元素个数是n\*m

ar.dtype 查看数组中元素的类型，python中的type()函数查看的是数组变量的类型

ar.itemsize 数组中每个元素的字节长度，int32 类型是4字节，float类型是8字节

ar 查看数组详情

创建数组

ar0 = numpy.array([1,2,3,4,5,6,7,8,9])

ar1 = numpy.array(range(10))

ar2 = numpy.arange(10) numpy自带生成数组的方法

ar3 = numpy.random.rand(10),**reshape(2,5)** #创建10个随机数，**2行5列**

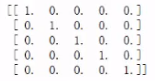
ar4 = numpy.linspace(10, 21, num=20, endpoint= True, retstep=True) #生成10-21之间的数据，均分成20个，endpoint参数指定是否包含最后一个边界数，retstep是否显示步长

ar5 = numpy.zeros(10) 创建长度为10的数组，用0填充，ones(10)方法类似

ar6=numpy.zeros((2\*5), dtype=numpy.float) 创建2行5列，用0填充的数组，元素为浮点数，ones((2\*5), dtype=numpy.int)方法类似

ar7 = numpy.zeros\_like(ar3) 创建一个数组ar7 复制ar3的数组结构，ones\_like(ar3)方法类似

ar8 = numpy.eye(5) #创建一个正方形n\*n的单位矩阵，对角线值为1，其余数为0



常用

str.split(',') #以什么分割字符串为列表

str.strip() #去除空格

dict(data) # 转data为字典格式，data的数据是列表[key, value]格式才行