[Python常识及环境搭建 3](#_Toc169748892)

[python3 安装 3](#_Toc1019943442)

[python开发环境: pytharm 3](#_Toc1900371879)

[使用pypi(安装开发模块) 3](#_Toc209545030)

[本地安装 4](#_Toc882512898)

[python2的不同之处 4](#_Toc437660427)

[PYcharm 5](#_Toc1276240638)

[pycharm快捷键操作 5](#_Toc1080519217)

[PyCharm的常用方法 5](#_Toc2128027823)

[Python基本语法 8](#_Toc1796201047)

[异常处理 11](#_Toc1559599833)

[基本数据类型 13](#_Toc485328586)

[数字 13](#_Toc1864714168)

[字符串 13](#_Toc1948915071)

[内置函数 18](#_Toc1747222344)

[列表 18](#_Toc988241576)

[列表解析 19](#_Toc390907904)

[列表内建函数 20](#_Toc1787943884)

[元组 21](#_Toc1938803105)

[字典 21](#_Toc1248018992)

[字典相关函数 22](#_Toc549973466)

[集合 23](#_Toc2030219144)

[数据类型比较 24](#_Toc1492645109)

[流程控制 26](#_Toc1424065525)

[if语句语法结构 26](#_Toc1432786610)

[while循环 27](#_Toc462007197)

[for循环 27](#_Toc2102981309)

[函数基本操作 28](#_Toc543611542)

[匿名函数lambda 31](#_Toc919676779)

[filter()函数,map函数 31](#_Toc1811588616)

[偏函数 32](#_Toc520786911)

[生成器 32](#_Toc1089425671)

[内部函数闭包 33](#_Toc684048410)

[装饰器 34](#_Toc273675143)

[OOP基础 35](#_Toc1298970701)

[基本概念 35](#_Toc1566561309)

[组合 37](#_Toc711335570)

[派生/继承 37](#_Toc427727691)

[通过继承覆盖方法 38](#_Toc499596878)

[多重继承 39](#_Toc691879746)

[特殊方法 39](#_Toc76445091)

[魔法方法 40](#_Toc2059196711)

[多线程多进程 41](#_Toc1177208332)

[多进程及案例 41](#_Toc1941159259)

[多线程及案例 44](#_Toc1860628135)

[Ansible编程基础 49](#_Toc776947029)

[Python使用ansible模块 50](#_Toc781917187)

[python调用playbook 53](#_Toc104052391)

[编写ansible模块(了解,一般调用已有的官方模块) 55](#_Toc417407265)

[模块基础 56](#_Toc573236644)

[hashlib模块(md5 sha 1,sha256,sha512,加密解密,完整性校验) 59](#_Toc1352071383)

[re模块(正则) 60](#_Toc967380732)

[tarfile模块(解压,压缩) 62](#_Toc455972140)

[collections的Counter模块(带统计排序功能的字典) 62](#_Toc697232844)

[shutil模块 (文件目录移动拷贝,权限) 63](#_Toc243962609)

[keywork模块(查看关键字,检查关键字) 64](#_Toc1888758750)

[subprocess模块(调用系统命令) 65](#_Toc1159240042)

[socket模块 65](#_Toc199460270)

[创建TCP服务器 65](#_Toc284886644)

[创建UDP服务器及客户端 68](#_Toc2078916821)

[time模块 69](#_Toc2011048887)

[datetime模块 70](#_Toc805673556)

[os模块简介(跟文件相关的操作) 71](#_Toc1020858844)

[pickle模块简介(读写python对象) 71](#_Toc547613649)

[pymysql模块(操作mysql数据) 72](#_Toc1079348699)

[SQLAlchemy模块(支持各种数据库) 74](#_Toc172345898)

[建立连接(deconn.py) 76](#_Toc2114174958)

[查询操作 78](#_Toc1790684269)

[更新数据 81](#_Toc600073589)

[删除记录 82](#_Toc466288188)

[urllib基础 (获取网页内容) 82](#_Toc335080367)

[案例:爬取网页 82](#_Toc676518680)

[案例3:爬取图片 83](#_Toc378001252)

[模拟客户端 84](#_Toc1512288700)

[urllib进阶 85](#_Toc470194292)

[HTTP异常处理 85](#_Toc91145739)

[request模块 (urllib 高级版,获取网页内容) 86](#_Toc141752081)

[request对象 86](#_Toc1252111479)

[Response 对象 87](#_Toc195198130)

[paramiko模块(并发远程执行命令) 89](#_Toc559159346)

[smtplib模块(邮件编程) 92](#_Toc1825348124)

[JSON 模块 94](#_Toc1547269513)

[zabbix编程 95](#_Toc1526540078)

[文件对象 100](#_Toc133836616)

[文件读取 101](#_Toc97018710)

[文件输出 102](#_Toc1770502688)

[文件指针移动 103](#_Toc2022595367)

[标准文件 103](#_Toc1256258752)

[不知道放哪里 104](#_Toc1969962958)

[print用法 104](#_Toc159998363)

[input用法 104](#_Toc1187691925)

# Python常识及环境搭建

python2 跟python3不兼容,2是主流.我们学习python3

python 跨平台的

python官方站点 [www.python.org](http://www.python.org)

python 软件仓库 https://pypi.python.org

## python3 安装

: 从官网下载源码,安装依赖包.编译安装

yum -y install gcc gcc-c++ zlib-devel openssl-devel readline-devel libffi-devel sqlite-devel tcl-devel tk-devel

tar xzf Python-3.6.1.tar.gz

./configure --prefix=/usr/local

make&&make install

/系统usr/bin/python 是python2的软连接,不要修改成python3,因为系统很多程序是python2写的.

## python开发环境: pytharm

. 下载地址 https://www.jetbrains.com/pycharm/download

新建工程. 注意解释器那边选择python3. 工程目录默认放在家目录下pythonProject/untitled

https://yiyibooks.cn //python官方文档翻译网站

## 使用pypi(安装开发模块)

• pypi即python package index

• 是python语言的软件仓库

• 官方站点为https://pypi.python.org

安装系统不存在的python模块，可以使用**pip**，相当于yum

1、安装gcc，有些软件包是C的源码

# yum -y install gcc

2、为了加速下载，可以使用国内开元镜像站点

# mkdir /root/.pip

# vim /root/.pip/pip.conf

[global]

index-url = http://pypi.douban.com/simple/

[install]

trusted-host=pypi.douban.com

3、安装pymysql

# pip3 install pymysql

## 本地安装

安装paramiko模块

• 本地安装

# yum install -y gcc gcc-c++ python-devel

# tar xzf paramiko-1.15.4.tar.gz

# python setup.py install

## python2的不同之处

#!/usr/bin/env python

# coding: utf8 ---------------> python2 默认ascii，这里改成utf8.必须紧跟着第一行

python2 中默认不支持tab补齐,需要以下操作:

vim /usr/local/bin/tab.py

from rlcompleter import readline

readline.parse\_and\_bind('tab: complete')

vim ~/.bash\_profile

export PYTHONSTARTUP=/usr/local/bin/tab.py

source ~/.bash\_profile

python2中，print不是函数

>>> print 'hello world!'

>>> print 10

python2中，输入不采用input()，而是raw\_input()

# PYcharm

## pycharm快捷键操作

ctrl +/ 群体注释/反注释

划中所选区域，按table键就把对应的行前加一个table

划中所选区域，按shift +table键就把对应的行前减一个table

神奇的工具,pycham ->code ->reformat code 自动调整代码风格

小技巧

def push\_it():

psss

def pot\_it():

pass

def view\_it():

pass

cmds={‘0’:push\_it,’1’:pop\_it,’3’=view\_it}

choice=input(“请输入012”)

cmds[choice]() //直接执行对应的函数

新建工程的时候,选择解释其的时候,需要勾选”Inherit global sit-pachages”,不然找不到新安装的包. 或者选择下面的’existing interpteter’

## [PyCharm的常用方法](https://www.cnblogs.com/xyc211/p/7986064.html)

Alt+Enter 自动添加包  
Ctrl + Shift + NumPad+ 展开所有的代码块

Ctrl + Shift + NumPad- 收缩所有的代码块

Ctrl + NumPad+/- 展开或者收缩代码块

CTRL+F 查找

Ctrl + R 替换  
Ctrl + Shift + F 全局查找  
Ctrl + Shift + R 全局替换

3、运行(Running)  
Alt + Shift + F10 运行模式配置  
Alt + Shift + F9 调试模式配置  
Shift + F10 运行  
Shift + F9 调试  
  
4、调试(Debugging)  
F8 跳过  
F7 进入  
Shift + F8 退出  
Alt + F9 运行游标  
Alt + F8 验证表达式  
Ctrl + Alt + F8 快速验证表达式  
F9 恢复程序  
Ctrl + F8 断点开关  
Ctrl + Shift + F8 查看断点  
  
5、导航(Navigation)  
Ctrl + N 跳转到类  
Ctrl + Shift + N 跳转到符号  
Alt + Right/Left 跳转到下一个、前一个编辑的选项卡(上/下一个打开的文件)  
F12 回到先前的工具窗口  
Esc 从工具窗口回到编辑窗口  
Shift + Esc 隐藏运行的、最近运行的窗口  
Ctrl + Shift + F4 关闭主动运行的选项卡  
Ctrl + G 查看当前行号、字符号  
Ctrl + E 当前文件弹出(最近使用的文件.各种浮动框)  
Alt + F1 查找当前文件或标识  
Ctrl+B / Ctrl+Click 跳转到声明  
Ctrl + Alt + B 跳转到实现  
Ctrl + Shift + I查看快速定义  
Ctrl + Shift + B跳转到类型声明  
Ctrl + U跳转到父方法、父类  
Alt + Up/Down跳转到上一个、下一个方法  
Ctrl + ]/[跳转到代码块结束、开始  
Ctrl + F12弹出**文件**结构  
Ctrl + H**类**型层次结构  
Ctrl + Shift + H方法层次结构  
Ctrl + Alt + H调用层次结构  
F2 / Shift + F2下一条、前一条高亮的错误  
  
  
6、搜索相关(Usage Search)  
Alt + F7/Ctrl + F7文件中查询用法  
Ctrl + Shift + F7文件中用法高亮显示  
Ctrl + Alt + F7显示用法  
  
7、重构(Refactoring)  
F5复制F6剪切  
Alt + Delete安全删除  
Shift + F6重命名  
Ctrl + F6更改签名  
Ctrl + Alt + N内联  
Ctrl + Alt + M提取方法  
Ctrl + Alt + V提取属性  
Ctrl + Alt + F提取字段  
Ctrl + Alt + C提取常量  
Ctrl + Alt + P提取参数  
  
8、控制VCS/Local History  
Ctrl + K提交项目  
Ctrl + T更新项目  
Alt + Shift + C查看最近的变化  
Alt + BackQuote(’)VCS快速弹出  
  
9、模版(Live Templates)  
Ctrl + Alt + J当前行使用模版  
Ctrl +Ｊ插入模版  
  
10、基本(General)  
Alt + #[0-9]打开相应的工具窗口  
Ctrl + Alt + Y同步  
Ctrl + Shift + F12最大化编辑开关  
Alt + Shift + F添加到最喜欢  
Alt + Shift + I根据配置检查当前文件  
Ctrl + BackQuote(’)快速切换当前计划  
Ctrl + Alt + S　打开设置页  
Ctrl + Shift + A查找编辑器里所有的动作  
Ctrl + Tab在窗口间进行切换

**一些常用设置：**

1. pycharm默认是自动保存的，习惯自己按ctrl + s 的可以进行如下设置：  
1. file -> Setting -> General -> Synchronization -> Save files on frame deactivation 和 Save files automatically if application is idle for .. sec 的勾去掉  
2. file ->Setting -> Editor -> Editor Tabs -> Mark modified tabs with asterisk 打上勾  
2. Alt + Enter: 自动添加包

3. 对于常用的快捷键，可以设置为visual studio(eclipse...)一样的：  
file -> Setting -> Keymap -> Keymaps -> vuisual studio -> Apply

4. Pycharm中默认是不能用Ctrl+滚轮改变字体大小的，可以在file -> Setting ->Editor-〉Mouse中设置

5. 要设置Pycharm的字体，要先在file -> Setting ->Editor-〉Editor中选择一种风格并保存，然后才可以改变

6. 在setting中搜索theme可以改变主题，所有配色统一改变

# Python基本语法

print('hello world!') #顶格写,不然报错

if 3 > 0: # 关键字后面有冒号

print('OK') #以缩进控制流程

print('yes')

x = 3; y = 4 # 不推荐，还是应该写成两行

print(x + y)

**运算符 不分数字还是字符**

= += 等

• 标准算术运算符

+ - \* / // % \*\*

• 比较运算符

< <= > >= == != <>

• 逻辑运算符

and not or

print(5 / 2) # 2.5

print(5 // 2) # 丢弃余数，只保留商

print(5 % 2) # 求余数

print(5 \*\* 3) # 5的3次方

print(5 > 3) # 返回True

print(3 > 5) # 返回False

print(20 > 10 > 5) # python支持连续比较

print(20 > 10 and 10 > 5) # 与上面相同含义

print(not 20 > 10) # False

**合法标识符**

• python标识符字符串规则和其他大部分用C编写的高级语言相似

• 第一个字符必须是字母或下划线(\_)

• 剩下的字符可以是字母和数字或下划线

• 大小写敏感

**变量赋值**

• python支持链式多重赋值

x = y = 10

• 另一种将多个变量同时赋值的方法称为多元赋值,采用这种方式赋值时,等号两边的对象都是元组

a, b = 10, 20

**名字空间**

• 任何时候,总有一个到三个活动的作用域(内建、全局和局部)

• 标识符的搜索顺序依次是局部、全局和内建

• 提到名字空间,可以想像是否有这个标识符

• 提到变量作用域,可以想像是否可以“看见”这个标识符函数式编程

**变量作用域全局变量**

• 标识符的作用域是定义为其声明在程序里的可应用范围,也就是变量的可见性

• 在一个模块中最高级别的变量有全局作用域

• 全局变量的一个特征是除非被删除掉,否则它们的存活到脚本运行结束,且对于所有的函数,他们的值都是可以被访问的局部变量

• 局部变量只时暂时地存在,仅仅只依赖于定义它们的函数现阶段是否处于活动

• 当一个函数调用出现时,其局部变量就进入声明它们的作用域。在那一刻,一个新的局部变量名为那个对象创建了

• 一旦函数完成,框架被释放,变量将会离开作用域

• 如果局部与全局有相同名称的变量,那么函数运行时,局部变量的名称将会把全局变量名称遮盖住

>>> x = 4

>>> def foo():

... x = 10

... print('in foo, x =', x)

...

>>> foo()

in foo, x = 10

>>> print('in main, x =', x)

in main, x = 4

python **中global 跟nonlocal的作用域**

python引用变量的顺序：

当前作用域局部变量->外层作用域变量->当前模块中的全局变量->python内置变量 。

a=3

b=3

c=3

def func1():

global a #声明全局变量a  
 global d # 定义全局变量d.在函数外面也可以使用d  
 a = 3 #修改全局变量a  
 b = 3 #定义局部变量b  
 print(c) #不修改全局变量的值,就可以直接使用全局变量,不用声明  
 d='d'  
 def func2():  
 global a #声明全局变量a  
 nonlocal b #声明非局部变量b. nonlocal关键字用来在函数或其他作用域中使用外层(非全局)变量.此处为func中定义的b  
 b=4  
 a += 10  
 #nonlocal e #会报错,nonlocal只声明.不定义.  
 print('fun2 a=%s'%a)  
 print('fun2 b=%s' % b)  
 print('fun2 c=%s' % c)  
 print('fun2 d=%s' % d)  
 func2()  
 print('fun1 a=%s' % a)  
 print('fun1 b=%s' % b)  
 print('fun1 c=%s' % c)  
 print('fun1 d=%s' % d)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 func1()  
 print(d) #全局变量d尽管在函数中定义,但是d(全局变量)的作用域是整个程序,所以在此时还是有效的

**关键字**

• 和其他的高级语言一样,python也拥有一些被称作关键这字的保留字符

• 任何语言的关键字应该保持相对的稳定,但是因为python是一门不断成长和进化的语言,其关键字偶尔会更新

• 关键字列表和iskeyword()函数都放入了keyword模块以便查阅

import keyword

keyword.kwlist # 查看所有关键字

keyword,iskeyword(‘pass’) # 查看是否是关键字

**内建**

• 除了关键字之外,python还有可以在任何一级代码使用的“内建”的名字集合,这些名字可以由解释器设置或使用

• 虽然built-in不是关键字,但是应该把它当作“系统保留字”

• 保留的常量如:True、False、None

**模块结构及布局**

• 编写程序时,应该建立一种统一且容易阅读的结构,并将它应用到每一个文件中去

#!/usr/bin/env python #起始行

“this is a test module” #模块文档字符串

import sys #导入模块

import os

debug = True #全局变量声明

class FooClass(object): #类定义

'Foo class'

pass

def test(): #函数定义

"test funcDon"

foo = FooClass()

if \_\_name\_\_ == ‘\_\_main\_\_’: #程序主体

test()

## 异常处理

什么是异常

• 当python检测到一个错误时,解释器就会指出当前流已经无法继续执行下去,这时候就出现了异常

• 异常是因为程序出现了错误而在正常控制流以外采取的行为

• 这个行为又分为两个阶段:

– 首先是引起异常发生的错误

– 然后是检测(和采取可能的措施)阶段

python中的异常

• 当程序运行时,因为遇到未解的错误而导致中止运行,便会出现traceback消息,打印异常

异常 描述

NameError 未声明/初始化对象

IndexError 序列中没有没有此索引

SyntaxError 语法错误

KeyboardInterrupt 用户中断执行

EOFError 没有内建输入,到达EOF标记

IOError 输入/输出操作失败

valueerror int()转换的时候类型不对

FileNotFoundError

Exception 匹配所有的exception

try-except语句

• 定义了进行异常监控的一段代码,并且提供了处理异常的机制

try:

try\_suite #监控这里的异常

except Exception [as reason]: 把异常的说明保存到变量reason中

except\_suite #异常处理代码

>>> try:

... f = open('foo.txt')

... except FileNotFoundError:

... print('No such file')

...

No such file

带有多个expect的try语句

• 可以把多个except语句连接在一起,处理一个try块中可能发生的多种异常

try:

n = int(input("number: "))

result = 100 / n

except (ValueError, ZeroDivisionError):

print('invalid number')

except (KeyboardInterrupt, EOFError):

print('\nBye-bye')

else:

print(result) # 异常不发生时才执行else子句

finally:

print('Done') # 不管异常是否发生都必须执行的语句

# 常用形式有try-except和try-finally

**raise语句**

• 要想引发异常,最简单的形式就是输入关键字raise,

后面跟要引发的异常的名称

• 执行raise语句时,Python会创建指定的异常类的一个对象

• raise语句还可指定对异常对象进行初始化的参数

**断言**

• 断言是一句必须等价于布尔值为真的判定

• 此外,发生异常也意味着表达式为假

案例5:自定义异常

1. 编写第一个函数,函数接收姓名和年龄,如果年龄

不在1到120之间,产生ValueError异常

2. 编写第二个函数,函数接收姓名和年龄,如果年龄

不在1到120之间,产生断言异常

def set\_age(name, age):

if not 0 < age < 120:

raise ValueError('年龄超过范围')

print("%s is %d years old" % (name, age))

def set\_age2(name, age):

assert 0 < age < 120, '年龄超过范围' //如果不在这个范围就抛异常

print("%s is %d years old" % (name, age))

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

set\_age('zhangsan', 20)

set\_age2('lisi', 200)

# 基本数据类型

## 数字

基本数字类型

• int:有符号整数

• bool:布尔值

– True:1 //True False必须这么写

– False:0

• float:浮点数

• complex:复数

数字表示方式

• python默认以十进制数显示

• 数字以0o或0O开头表示为8进制数

• 数字以0x或0X开头表示16进制数

• 数字以0b或0B开头表示2进制数

## 字符串

定义字符串

• python中字符串被定义为引号之间的字符集合

• python支持使用成对的单引号或双引号

• 无论单引号,还是双引号,表示的意义相同

• python还支持三引号(三个连续的单引号或者双引号),可以用来包含特殊字符,支持换行回车，可以保持输入格式

• python不区分字符和字符串

>>> sentence = "tom's cat is dog" #单引号双引号意义想用

>>> sentence = ‘tom\’s cat is a dog’ #使用转义

>>> words = ''' #三个单引号

... haha

... xix

... tang'''

>>> words

'\nhaha\nxix\ntang'

>>> print (words)

haha

xix

tang

**序列类型操作符 (字符串,列表元祖)**

序列操作符 作用

seq[ind] 获得下标为ind的元素,第一个字符的索引是0,最后一个字符的索引是-1.子字符串包含切片中的起始下标,但不包含结束下标

seq[ind1:ind2] 获得下标从ind1到ind2间的元素集合

seq \* expr 序列重复expr次

seq1 + seq2 连接序列seq1和seq2

obj in seq 判断obj元素是否包含在seq中

obj not in seq 判断obj元素是否不包含在seq中

py\_str = 'python'

len(py\_str) # 取长度

py\_str[0] # 第一个字符,不能修改

'python'[0] #第一个字符,不推荐

py\_str[-1] # 最后一个字符

# py\_str[6] # 错误，下标超出范围

py\_str[2:4] # 切片，起始下标包含，结束下标不包含

py\_str[2:] # 从下标为2的字符取到结尾

py\_str[:2] # 从开头取到下标是2之前的字符

py\_str[:] # 取全部

py\_str[::2] # 步长值为2，默认是1,得到pyo. **[起始位置:结束位置:步长]**

py\_str[1::2] # 取出yhn

py\_str[::-1] # 步长为负，表示自右向左取

py\_str[1:3:-1] # 空

py\_str[3:1:-1] #'ht'

**内建函数**

函 数 含 义

list(iter) 把可迭代对象转换为列表

str(obj) 把obj对象转换成字符串

tuple(iter) 把一个可迭代对象转换成一个元组对象,同理还有int(),float()

len(seq) 返回seq的长度

max(iter,key=None):返回iter中的最大值

enumerate 接受一个可迭代对象作为参数,返回一个enumerate对象.就是方便得到容器的下标跟内容

reversed(seq) 接受一个序列作为参数,返回一个以逆序访问的迭代器

sorted(iter) 接受一个可迭代对象作为参数,返回一个有序的**列表**

**例子:**

from random import randint

alist = list() # []

list('hello') # ['h', 'e', 'l', 'l', 'o']

list((10, 20, 30)) # [10, 20, 30] 元组转列表

astr = str() # ''

str(10) # '10'

str(['h', 'e', 'l', 'l', 'o']) # 将列表转成字符串 得到”['h', 'e', 'l', 'l', 'o']”

tmp\_str=’’.join(['h', 'e', 'l', 'l', 'o']) # 得到”hello”

atuple = tuple() # ()

tuple('hello') # ('h', 'e', 'l', 'l', 'o')

num\_list = [randint(1, 100) for i in range(10)]

max(num\_list)

min(num\_list)

**enumerate:**

alist = [10,'john']

>>> list(enumerate(alist))

[(0, 10), (1, 'john')]

>>> for ind in range(len(alist)) :

... print('%s:%s' % (ind,alist[ind]))

...

0:10

1:john

>>> for item in enumerate(alist) :

... print('%s:%s'%(item[0],item[1]))

...

0:10

1:john

>>> for ind,val in enumerate(alist) :

... print('%s:%s'%(ind,val))

...

0:10

1:john

**reversed(seq):**

**sorted(iter):**

>>> atuple = [randint(1,100) for i in range(10)]

>>> atuple

[29, 23, 31, 93, 29, 23, 61, 73, 5, 16]

>>> sorted(atuple)

[5, 16, 23, 23, 29, 29, 31, 61, 73, 93]

>>> reversed(atuple)

<list\_reverseiterator object at 0x7fb2112f20f0>

>>> list(reversed(atuple))

[16, 5, 73, 61, 23, 29, 93, 31, 23, 29]

>>> for i in reversed(atuple) :

... print(i,end=' ')

...

16 5 73 61 23 29 93 31 23 29

案例2:检查标识符

1. 程序接受用户输入

2. 判断用户输入的标识符是否合法

3. 用户输入的标识符不能使用关键字

4. 有不合法字符,需要指明第几个字符不合法

import string

import keyword

import sys

first\_char = string.ascii\_letters + '\_'

all\_char = string.digits + first\_char

def check\_id(char) :

if char[0] not in first\_char :

return "1st is invalid!"

if keyword.iskeyword(char) :

return '%s is keyword' % char

for i in enumerate(char) :

if i[1] not in all\_char :

return '第%s个字符不合法！！！' % (i[0]+1)

return '%s is valid' % char

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

print(check\_id(sys.argv[1]))

**格式化操作符**

• 字符串可以使用格式化符号来表示特定含义

**格式化字符 转换方式**

%c 转换成字符

%s 优先用str()函数进行字符串转换

%d / %i 转成有符号十进制数

%o 转成无符号八进制数

%#o 带0o的八进制数

%e / %E 转成科学计数法

%f / %F 转成浮点数

**辅助指令 作用**

\* 定义宽度或者小数点精度

- 左对齐

+ 在正数前面显示加号

<sp> 在正数前面显示空格

# 在八进制数前面显示零0,在十六进制前面显示'0x'或者'0X'

0 显示的数字前面填充0而不是默认的空格

"%s is %s years old" % ('bob', 23) # 常用

"%s is %d years old" % ('bob', 23) # 常用

"%s is %d years old" % ('bob', 23.5) # %d是整数 常用

"%s is %f years old" % ('bob', 23.5)

"%s is %5.2f years old" % ('bob', 23.5) # %5.2f是宽度为5，2位小数

"97 is %c" % 97 #转换成ascii

"11 is %#o" % 11 # %#o表示有前缀的8进制

"11 is %#x" % 11

"%10s%5s" % ('name', 'age') # %10s表示总宽度为10，右对齐, 常用

"%-10s%-5s" % ('name', 'age') # %-10s表示左对齐, 常用

"%-10s%-5s" % ('bob', 25)

"%10d" % 123

"%010d" % 123

**format函数**

• 使用位置参数

– 'my name is {} ,age {}'.format('hoho',18)

• 使用关键字参数

– 'my name is {name},age is

{age}'.format({'name':'bob', 'age':23})

• 填充与格式化

– {:[填充字符][对齐方式 <^>][宽度]}

• 使用索引

– 'name is {0[0]} age is {0[1]}'.format(['bob', 23])

"{} is {} years old".format('bob', 25)

"{1} is {0} years old".format(25, 'bob')

"{:<10}{:<8}".format('name', 'age')

**原始字符串操作符**

• 原始字符串操作符是为了对付那些在字符串中出现的特殊字符

• 在原始字符串里,所有的字符都是直接按照字面的意思来使用,没有转义特殊或不能打印的字符

>>> winPath = "c:\windows\temp"

>>> print(winPath)

c:\windows emp

>>> newPath = r"c:\windows\temp"

>>> print(newPath)

c:\windows\temp

### 内置函数

py\_str = 'hello world!'

py\_str.capitalize() #首字母大写

py\_str.title() #每个单词首字母大写

py\_str.center(50) #居中,默认用空格填充

py\_str.center(50, '#') #居中,两边用#填充

py\_str.ljust(50, '\*') #左对齐,\*填充

py\_str.rjust(50, '\*') #右对齐,\*填充

py\_str.count('l') # 统计l出现的次数

py\_str.endswith('!') # 以!结尾吗？

py\_str.endswith('d!')

py\_str.startswith('a') # 以a开头吗？

py\_str.islower() # 字母都是小写的？其他字符不考虑

py\_str.isupper() # 字母都是大写的？其他字符不考虑

'Hao123'.isdigit() # 所有字符都是数字吗？

'Hao123'.isalnum() # 所有字符都是字母数字？

' hello\t '.strip() # 去除两端空白字符，常用

str.strip(‘\r\n’) #去掉两端的\\r\n字符

' hello\t '.lstrip() #右端去掉空白符

' hello\t '.rstrip()

'how are you?'.split() #默认以空格切割,返回列表

'hello.tar.gz'.split('.') #以”.”切割

'.'.join(['hello', 'tar', 'gz']) #返回hello.tar.gz

'-'.join(['hello', 'tar', 'gz']) #返回hello-tar-gz

## 列表

• 可以将列表当成普通的“数组”,它能保存任意数量任意类型的python对象

• 像字符串一样,列表也支持下标和切片操作

• 列表中的项目可以改变

创建及访问列表

• 列表是有序、可变的数据类型

• 列表中可以包含不同类型的对象

• 列表可以由[]或工厂函数创建

• 支持下标及切片操作

a = b = 10

a = 20 # b的值不变，因为数字是不可变的

alist = blist = [1, 2]

alist[0] = 10 # blist也会改变，因为列表是可变的

alist = [10, 20, 30, 'bob', 'alice', [1,2,3]]

len(alist)

alist[-1] # 取出最后一项

alist[-1][-1] # 因为最后一项是列表，列表还可以继续取下标 ,取出3.

[1,2,3][-1] # [1,2,3]是列表，[-1]表示列表最后一项

alist[-2][2] # 列表倒数第2项是字符串，再取出字符下标为2的字符,取得 i

alist[3:5] # ['bob', 'alice']

10 in alist # True

'o' in alist # False

100 not in alist # True

alist[-1] = 100 # 修改最后一项的值

alist.append(200) # 向列表中追加一项

alist + [200] #可以,但alist不会变

更新列表

• 通过下标只能更新值,不能使用下标添加新值

>>> alist = [10, 35, 20, 80]

>>> alist[-1] = 100

>>> alist[1:3] = [30, 50]

>>> alist[2:2]

[]

>>> alist[2:2] = [11,22,33,44]

>>> alist

[10, 35, 11, 22, 33, 44, 20, 80]

**成员关系判断**

'o' in py\_str //True

'to' in py\_str //False

'to' not in py\_str // True

### 列表解析

• 它是一个非常有用、简单、而且灵活的工具,可以用来**动态地创建列表**

• 语法:

[expr for iter\_var in iterable]

• 这个语句的核心是for循环,它迭代iterable对象的所有条目

• expr应用于序列的每个成员,最后的结果值是该表达式产生的列表

[10+5]

[10+5 for i in range(10)]

[10+i for i in range(10)]

[10+i for i in range(10) if i % 2 == 1]

[10+i for i in range(10) if i % 2]

['192.168.1.%s' % i for i in range(1,20)]

ips= ('172.40.58.%s' % i for i in range(1, 255)) # 生成列表对象,创建生成器,不是直接生成列表,减少内存空间

**list与string互相转换**

list函数把字符串转换成列表

s = 'Python2best'

s1 = [3,5,4,2]

s2 = []

for ch in s :

s2.append(ch)

print(s2)

print(list(s))

string的join函数可以把列表转换成string

str=’’.join(s1)

### 列表内建函数

列表方法 操作

list.append(obj) 向列表中添加一个对象obj

list.count(obj) 返回一个对象obj 在列表中出现的次数

list.extend(seq) 把序列seq的内容添加到列表中

list.index(obj) 返回obj对象的下标

list.insert(index, obj) 在索引量为index 的位置插入对象obj

list.reverse() 原地翻转列表

list.sort() 排序

list.pop() 弹出最后一项

list.pop(2) 弹出第二项

alist = [1, 2, 3, 'bob', 'alice']

alist[0] = 10

alist[1:3] = [20, 30]

alist[2:2] = [22, 24, 26, 28] #在第二个位置后加入这些元素

alist.append(100)

alist.remove(24) # 删除第一个24

alist.index('bob') # 返回下标

blist = alist.copy() # 相当于blist = alist[:]

alist.insert(1, 15) # 向下标为1的位置插入数字15

alist.pop() # 默认弹出最后一项

alist.pop(2) # 弹出下标为2的项目

alist.pop(alist.index('bob'))#弹出bob

alist.sort()

alist.reverse()

alist.count(20) # 统计20在列表中出现的次数

alist.clear() # 清空

alist.append('new') #加入new

alist.extend('new') #加入三个字母

alist.extend(['hello', 'world', 'hehe'])

## 元组

• 通过()或工厂函数tuple()创建元组

• 元组是有序的、不可变类型

• 与列表类似,作用于列表的操作,绝大数也可以作用于元组

atuple = (10, 20, 30, 'bob', 'alice', [1,2,3]) #注意是圆括号

len(atuple)

10 in atuple #True

atuple[2]

atuple[3:5]

# atuple[-1] = 100 # 错误，元组是不可变的

单元素的元组

a = (10) 使用type 查看这是一个数值

应该这样写

a = (10,)

如果元组的元素是可变的,那元组也可以改变的

a=(1,2,[3,4,6])

a[-1][-1]=5 //以上操作正确,不常用

## 字典

# 字典是key-value(键－值）对形式的，**没有顺序**，通过键取出值

创建字典

• 通过{ }操作符创建字典

• 通过dict()工厂方法创建字典

• 通过fromkeys()创建具有相同值的默认字典

adict = dict()

adict //{}

dict(['ab','cd']) //{'a': 'b', 'c': 'd'}

bdict = dict([('name','bob'),('age',43)]) //{'name': 'bob', 'age': 43}

{}.fromkeys(['zhangsan','kusu','bob'],11)//{'zhangsan': 11, 'kusu': 11, 'bob': 11}

访问字典

• 字典是映射类型,意味着它没有下标,访问字典中的值需要使用相应的键

for key in dic:

print 'key is %s,value is %s'%(key,dic[key])

for key,value in dic.items():

print 'key is %s,value is %s'%(key,value)

print('%(name)s:%(age)s' % bdick) //bob:43

更新字典

• 通过键更新字典

– 如果字典中有该键,则更新相关值

– 如果字典中没有该键,则向字典中添加新值

>>> bdick['name'] = 'tom'

>>> bdick

{'name': 'tom', 'age': 43}

>>> bdick['email'] = 'tom@tedu.cn'

>>> bdick

{'name': 'tom', 'age': 43, 'email': 'tom@tedu.cn'}

adic['name'] #取得 ’bob’

adic[‘score’]=90 # 添加新的键值对,如果key已经存在,则改变value的值

adic.setdefault(‘sex’,’male’) #添加键值对. 可以使用变量

删除字典

• 通过del可以删除字典中的元素或整个字典

• 使用内部方法clear()可以清空字典

• 使用pop()方法可以“弹出”字典中的元素

del adict['email’] //{'age': 22, 'name': 'bob’}

adict.pop('age’) //返回22 {'name':'bob’}

adict.clear() //{}

字典操作符

• 使用字典键查找操作符[ ],查找键所对应的值

• 使用in和not in判断键是否存在于字典中

adict = {'age': 23, 'name': 'bob'}

print(adict['name’]) //Bob

'bob' in adict //False

'name' in adict //True

### 字典相关函数

• len():返回字典中元素的数目

• hash():本身不是为字典设计的,但是可以判断某个对象是否可以作为字典的键

字典内建方法

• dict.copy():返回字典(深复制)的一个副本

• dict.get(key, default=None):对字典dict中的键key,返回它对应的值value,如果字典中不存在此键,则返回default的值

• dict.setdefault(key, default=None):如果字典中不存在key键,由dict[key]=default为它赋值

• dict.items():返回一个包含字典中(键,值)对元组的列表

• dict.keys():返回一个包含字典中键的列表

• dict.values():返回一个包含字典中所有值的列表

• dict.update(dict2):将字典dict2的键-值对添加到字典dict

adict = dict([('name','bob'),('age','23')]) //{'name': 'bob', 'age': '23'}

len(adict) //2

hash(10) # 判断给定的数据是不是不可变的，不可变数据才能作为key 返回10

hash('10') #可以 返回-8525507676581654654

>>> hash([1,2]) #不行

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

adict.keys() //dict\_keys(['name', 'age'])

adict.values() //dict\_values(['bob', '23'])

adict.items() //dict\_items([('name', 'bob'), ('age', '23')])

# get方法常用，重要

adict.get('name') //'bob'

adict.get('qq') # 取出字典中name对应的value，没有返回None

print(adict.get('qq','not found')) //可以指定返回值not found

print(adict.get('age','not found')) //23

adict.update({'name':'job'}) //{'name': 'job', 'age': '23'}

adict.update({'phone':'1234124124'}) //{'name': 'job', 'age': '23', 'phone': '1234124124'}

案例:显示10秒计时，且后面显示的会替换掉前面的

import time

import sys

for i in range(10) :

print('\r %s' % i ,end='') //\r光标切换到行首

sys.stdout.flush()

time.sleep(1)

## 集合

创建集合

• 数学上,把set称做由不同的元素组成的集合,集合(set)的成员通常被称做集合元素

**set 元素不可重复**

• 集合对象是一组无序排列的可哈希的值

• 集合有两种类型

– 可变集合set

– 不可变集合frozenset

集合相当于没有key的字典,也用{}表示

**集合类型操作符**

• 集合支持用in和not in操作符检查成员

• 能够通过len()检查集合大小

• 能够使用for迭代集合成员

• 不能取切片,没有键

aset = set('abc') //{‘a’,’b’,’c’}

bset = set('cde') //{‘c’,’d’,’e’}

aset & bset # 交集{’c’}

aset.intersection(bset) # 交集 返回一个set, a/bset 不变 {‘c’}

aset | bset # 并集 {‘a’,’b’,’c’,’d’,’e’}

aset.union(bset) # 并集 {‘a’,’b’,’c’,’d’,’e’}

aset - bset # 差补 跟b-a结果不一样. {‘a’,’b’}

aset.difference(bset) # 差补 {‘a’,’b’}

bset-aset # 插补 {’d’,’e’}

aset.add('new') #{‘a’,’b’,’c’,’new’}

aset.update(['aaa', 'bbb']) #{‘a’,’b’,’c’,’aaa’,’new’,’bbb’}

aset.remove('bbb') #{‘a’,’b’,’c’,’aaa’,’new’}

cset = set('abcde')

dset = set('bcd')

cset.issuperset(dset) # cset是dset的超集么？

cset.issubset(dset) # cset是dset的子集么？

案例： 比对2个大文件的不同的部分

file1 file2

with open('file1') as fobj :

aset = set(fobj) //把每一行放在集合里面,自动取出重复项

with open('file2') as fobj :

bset = set(fobj)

with open('diff.txt','w') as fobj :

fobj.writelines(bset - aset)

## 数据类型比较

python中不存在所谓的传值调用，一切传递的都是对象的引用

**• 按存储模型分类**

– 标量类型:数值、字符串 //不能放其他类型

– 容器类型:列表、元组、字典 //可以存放混合烈性

**• 按更新模型分类:**

– **可变类型**:列表、字典 //允许通过下标修改.变量指向的内存可以修改.

– **不可变类型**:数字、字符串、元组 //不允许透过下标修改内容.变量名指向的内存不可修改.要修改变量的内容,需要重新该分配内存,原先内存会被系统收回

**不可变类型:**整形来说，目前仅支持(-1,100)。

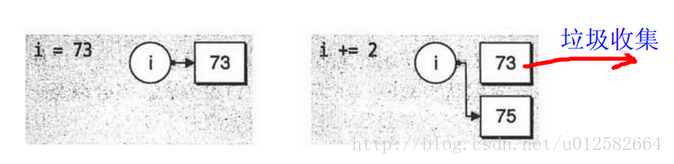
　　　　优点是，这样可以减少重复的值对内存空间的占用。

　　　　缺点呢，如例1所示，我要修改这个变量绑定的值，如果内存中没用存在该值的内存块，那么必须重新开辟一块内存，把新地址与变量名绑定。而不是修改变量原来指向的内存块的值，这回给执行效率带来一定的降低。

看下面的例子（例1）

i = 73

i += 2



从上图可知，不可变对象的特征没有变，变的只是创建了新对象，改变了变量的对象引用。

　　看一个例子（例2）

>>>x = 1

>>>y = 1

>>>z = 1>>> x is yTrue

>>>y is zTrue

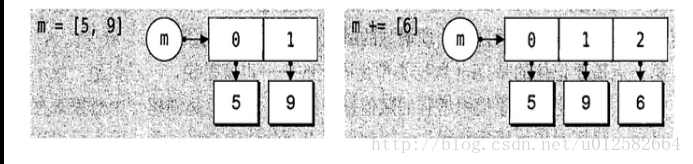
如上所示，因为整数为不可变，x,y,z在内存中均指向一个值为1的内存地址，也就是说，x,y,z均指向的是同一个地址.

**可变对象:**

下面看一个可变对象的例子（例3）

m=[5,9]

m+=[6]



**函数参数：**

Python函数参数对于可变对象，函数内对参数的改变会影响到原始对象；对于不可变对象，函数内对参数的改变不会影响到原始参数。原因在于：

1、可变对象，参数改变的是可变对象，其内容可以被修改。

2、不可变对象，改变的是函数内变量的指向对象。

比如存在2个列表 a 和 b. 如果a=b的话， a和b的地址是相同的；如果只是想拷贝，那么就得用 a=b[:]

**另外的**

blist=alist 这两个list指向同一块内存,此时修改blist的内容,alist也会改变.

blist is alist //True

clist=alist[:] 这两个list指向不同的内存,互不影响.

id(alist) #查看内存地址.

**• 按访问模型分类**

– 直接访问:数字

– 顺序访问:字符串、列表、元组

– 映射访问:字典

# 流程控制

## if语句语法结构

• 标准if条件语句的语法

if expression:

if\_suite

else:

else\_suite

• 如果表达式的值非0或者为布尔值True, 则代码组if\_suite被执行;否则就去执行else\_suite

• 代码组是一个python术语,它由一条或多条语句组成,表示一个子代码块

if 'i' in 'print' :

print('haha')

if 0:

print('no') #不执行,任何值为0的数字就是False

if 10 not in ('9','8','7'):

print('duide') 非空对象都是True

if ' ':

print('shikongde ') 空格字符也是字符，条件为True

“” in “nN” # 空字符串,返回True

“” in [‘n’,’N’] #返回False

alist=list(“hello”)

if alist

alist.pop //如果alist有内容,就执行

• 只要表达式数字为非零值即为True

• 空字符串、空列表、空元组,空字典的值均为False

**条件表达式**

• 从Python 2.5集成的语法确定为: X if C else Y //C返回True,则表达式返回X,否则返回Y

>>>x,y=3,4

>>>smaller = x if x < y else y

>>>print (smaller)

3

## while循环

循环语句基础

• 一组被重复执行的语句称之为循环体,能否继续重复,决定循环的终止条件

• Python中的循环有while循环和for循环

• 循环次数未知的情况下,建议采用while循环

• 循环次数可以预知的情况下,建议采用for循环

**while循环语法结构**

• 当需要语句不断的重复执行时,可以使用while循环

while expression:

while\_suite

• 语句while\_suite会被连续不断的循环执行,直到expression的值变成0或False

**break语句**

• break语句可以结束当前循环然后跳转到下条语句

• 写程序的时候,应尽量避免重复的代码,在这种情况下可以使用while-break结构

**continue语句**

• 当遇到continue语句时,程序会终止当前循环,并忽略剩余的语句,然后回到循环的顶端

• 如果仍然满足循环条件,循环体内语句继续执行,否则退出循环

**else语句**

• python中的while语句也支持else子句

• else子句只在循环完成后执行

• break语句也会跳过else块

## for循环

for循环语法结构

• python中的for接受可迭代对象(例如序列或迭代器)作为其参数,每次迭代其中一个元素

for iter\_var in iterable:

suite\_to\_repeat

• 与while循环一样,支持break、continue、else语句

• 一般情况下,循环次数未知采用while循环,循环次数已知,采用for循环

astr = 'hello'

alist = [10, 20, 30]

atuple = ('bob', 'tom', 'alice')

adict = {'name': 'john', 'age': '23'}

for ch in astr :

print(ch)

for i in alist :

print(i)

for name in atuple :

print(name)

for key in adict :

print('%s:%s' %(key,adict[key]))

**range函数**

• for循环常与range函数一起使用

• range函数提供循环条件，

• range函数的完整语法为:

range(start, end, step =1) 不包含末尾，默认开头是0，步长是1

range(10) #[0,1,2,3,4,5,6,7,8,9]

range(6,11) #[6,7,8,9,10]

range(1,10,2) #[1,3,5,7,9]

range(10,0,-1) #[10,9,8,7,6,5,4,3,2,1]

案例2:九九乘法表

1. 创建mtable.py程序

2. 乘序运行后,可以在屏幕上打印出九九乘法表课

3. 修改程序,由用户输入数字,可打印任意数字的乘法表

num = int(input('请输入要实现n乘法表：'))

for i in range(1,num+1) :

for j in range(1,i+1) :

print('%s\*%s=%s' % (i,j,i\*j),end='\t')

print()

# 函数基本操作

函数基本概念

• 函数是对程序逻辑进行结构化或过程化的一种编程方

• 将整块代码巧妙地隔离成易于管理的小块

• 把重复代码放到函数中而不是进行大量的拷贝,这样既能节省空间,也有助于保持一致性

• 通常函数都是用于实现某一种功能

创建函数

• 函数是用def语句来创建的,语法如下:

def funcOon\_name(arguments):

"funcOon\_documentaOon\_string"

funcOon\_body\_suite

• 标题行由def关键字,函数的名字,以及参数的集合(如果有的话)组成

• def子句的剩余部分包括了一个虽然可选但是强烈推荐的文档字串,和必需的函数体

标题行由def关键字,函数的名字,以及参数的集合(如果有的话)组成

• def子句的剩余部分包括了一个虽然可选但是强烈推荐的文档字串,和必需的函数体

前向引用

• 函数不允许在函数未声明之前对其进行引用或者调用

内部函数

• 在函数体内创建另外一个函数是完全合法的,这种函数叫做内部/内嵌函数

**调用函数**

• 同大多数语言相同,python用一对圆括号调用函数

• 如果没有加圆括号,只是对函数的引用

def get\_age(name, age):

print('%s is %s years old' % (name, age))

get\_age('bob', 25) # 参数按顺序传递

get\_age(25, 'bob')

get\_age(age=25, name='bob') //**关键字参数**,允许参数缺失或者不按顺序

**参数组**

• python允许程序员执行一个没有显式定义参数的函数

• 相应的方法是通过一个把元组(非关键字参数)或字典(关键字参数)作为参数组传递给函数func(\*tuple\_grp\_nonkw\_args, \*\*dict\_grp\_kw\_args)

\*args 表示args是一个元祖,参数个数不定

\*\*kwawgs 表示kwawgs是一个字典,参数不定

def func1(\*args): # \*表示args是个元组

print(args)

def func2(\*\*kwargs): # \*\*表示kwargs是个字典

print(kwargs)

def func3(x, y):

print(x \* y)

def func4(name, age):

print("%s is %s years old" % (name, age))

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

func1()

func1(10)

func1(10, 'bob')

func2()

func2(name='bob', age=25)

func3(\*[10, 5]) # 调用的时候，\*表示拆开后面的数据类型

func4(\*\*{'name': 'bob', 'age': 25}) # name='bob', age=25

**函数的返回值**

• 多数情况下,函数并不直接输出数据,而是向调用者返回值

• 函数的返回值使用return关键字

• 没有return的话,函数默认返回None

**定义参数**

• **形式参数**:函数定义时,紧跟在函数名后(圆括号内)的参数被称为形式参数,简称形参。由于它不是实际存在变量,所以又称虚拟变量

•  **实际参数**: 在主调函数中调用一个函数时,函数名后面括弧中的参数(可以是一个表达式)称为“实际参数”,简称实参

传递参数

• 调用函数时,实参的个数需要与形参个数一致

• 实参将依次传递给形参

在定义函数时，尽量不要使用input，尽量使用传参，因为数据的输入不一样是从键盘，也可以从文件或者其他

函数参数传递**：**

Python函数参数对于可变对象，函数内对参数的改变会影响到原始对象；对于不可变对象，函数内对参数的改变不会影响到原始参数。原因在于：

1、可变对象，参数改变的是可变对象，其内容可以被修改。

2、不可变对象，改变的是函数内变量的指向对象。

比如存在2个列表 a 和 b. 如果a=b的话， a和b的地址是相同的；如果只是想拷贝，那么就得用 a=b[:]

**位置参数**

• 与shell脚本类似,程序名以及参数都以位置参数的方式传递给python程序

• 使用sys模块的argv列表接收

position\_args.py

import sys

print(sys.argv)

[root@room9pc01 python]# python3 position\_args.py 1 jlasf 10

['position\_args.py', '1', 'jlasf', '10']

**默认参数**

• 默认参数就是声明了默认值的参数

• 因为给参数赋予了默认值,所以在函数调用时,不向该参数传入值也是允许的

函数定义时不能把没有默认值的参数放在有默认值的参数后面

def pstar(n=50) :

print('\*' \* n)

pstar()

pstar(30)

## 匿名函数lambda

• python允许用lambda关键字创造匿名函数

• 匿名是因为不需要以标准的def方式来声明

• 一个完整的lambda“语句”代表了一个表达式,这个表达式的定义体必须和声明放在同一行

见filter函数

lambda [arg1[, arg2, ... argN]]: expression

>>> a = lambda x, y: x + y

>>> print(a(3, 4))

## filter()函数,map函数

• filter(func, seq):调用一个布尔函数func来迭代遍历每个序列中的元素;返回一个使func返回值为true的元素的序列

• 如果布尔函数比较简单,直接使用lambda匿名函数就显得非常方便了

data = filter(lambda x: x % 2, [num for num in range(10)])

print(data) #过滤出10以内的奇数

[1, 3, 5, 7, 9]

from random import randint

def func1(x):

return x % 2

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

alist = [randint(1, 100) for i in range(10)]

print(alist)

# filter要求第一个参数是函数，该函数必须返回True或False(其实所有值都ok啦,非0就保留,0/none就丢弃)

# 执行时把alist的每一项作为 func1的参数，返回真留下，否则过滤掉

result = filter(func1, alist) # 高阶函数

print(list(result))

result2 = filter(lambda x: x % 2, alist)

print(list(result2))

类似的还有map函数

result = map(func1, alist) //filter筛选fun返回True的项,最终返回符合条件的源列表.map返回一个列表,列表是进过fun处理的全部项

print(list(result))

也可以用匿名函数

result2=map(lambda x : x\*2+1,alist)

print(list(result2))

## 偏函数

• 偏函数的概念是将函数式编程的概念和默认参数以及可变参数结合在一起

• 一个带有多个参数的函数,如果其中某些参数基本上固定的,那么就可以通过偏函数为这些参数赋默认值

from functools import partial

def foo(a, b, c, d, f):

return a + b + c + d + f

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

print(foo(10, 20, 30, 40, 5))

print(foo(10, 20, 30, 40, 25))

print(foo(10, 20, 30, 40, 69))

print(foo(10, 20, 30, 40, 32))

add = partial(foo, a=10, b=20, c=30, d=40)

print(add(f=5)) # foo(10, 20, 30, 40, 5)

print(add(f=8)) # foo(10, 20, 30, 40, 8)

**递归函数**

• 如果函数包含了对其自身的调用,该函数就是递归的

• 在操作系统中,查看某一目录内所有文件、修改权限等都是递归的应用

def func(num):

if num == 1:

return 1

else:

return num \* func(num - 1)

print(func(5)) //返回120

print(func(10)) //返回3628800

## 生成器

• 从句法上讲,生成器是一个带yield语句的函数

• 一个函数或者子程序只返回一次,但一个生成器能暂停执行并返回一个中间的结果

• yield 语句返回一个值给调用者并暂停执行

• 当生成器的next()方法被调用的时候,它会准确地从离开地方继续

• 与迭代器相似,生成器以另外的方式来运作

• 当到达一个真正的返回或者函数结束没有更多的值返回,StopIteration异常就会被抛出

def mygen():

yield 'hello'

a = 10 + 20

yield a

yield [1, 2, 3]

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

m = mygen() //先调用生成器生成一个generator对象

for i in m:

print(i)

for i in m:

print(i) # 无值，因为生成器对象只能用一次

案例: fileblock

读取文件并处理.但是读取整个文件,太大,读取一行,太小.这是就可以使用生成器

def blocks(fobj):

block = []

counter = 0

for line in fobj:

block.append(line)

counter += 1

if counter == 10:

yield block # 返回中间结果，下次取值，从这里继续向下执行

block = []

counter = 0

if block: //返回最后不到10行的部分

yield block

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

fobj = open('/tmp/passwd') # cp /etc/passwd /tmp

for lines in blocks(fobj):

print(lines)

print()

fobj.close()

## 内部函数闭包

• 闭包将内部函数自己的代码和作用域以及外部函数的作用结合起来

• 闭包的词法:变量不属于全局名字空间域或者局部的--而属于其他的名字空间,带着“流浪"的作用域

• 闭包对于安装计算,隐藏状态,以及在函数对象和作用域中随意地切换是很有用的

• 闭包也是函数,但是他们能携带一些额外的作用域

也就是函数包含函数,外部函数返回值是内部函数的地址. 外部函数利用形参改变内部函数的执行效果.

函数式编程,提高代码的可重复使用率. 也就是说可以用来生成函数

python一切皆对象，函数这一语法结构也是一个对象。在[函数对象](http://www.cnblogs.com/vamei/archive/2012/07/10/2582772.html)中，我们像使用一个普通对象一样使用函数对象，比如更改函数对象的名字，或者将函数对象作为参数进行传递。

实例 创建通用的计数器

**def** counter(start=0):  
 count = start  
 **def** incr():  
 **nonlocal** count  
 count +=1  
 **return** count  
 **return** incr  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 a=counter()  
 b=counter(10)  
 print(a()) //1  
 print(b()) //11  
 print(a()) //2 python一切皆对象,函数也是对象.python没有传值赋值,只有引用赋值. 那么a保存的是一个对象incr的引用. 因为count属于流浪域,出了函数不消失. 所以a对象的count一直是有值的.

print(b()) //12 两个流浪域不同

print(counter()()) //1

## 装饰器

• 装饰器是在函数调用之上的修饰

• 这些修饰仅是当声明一个函数或者方法的时候,才会应用的额外调用

• 使用装饰器的情形有:

– 引入日志

– 增加计时逻辑来检测性能

– 给函数加入事务的能力

def color(func):

def red():

return '\033[31;1m%s\033[0m' % func()

return red

def hello():

return 'Hello World!'

@color

def welcome():

return 'Hello China!'

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

print(color(hello)())

hello = color(hello)

print(hello())

print(welcome()) //,有装饰器,实际执行效果是 color(welcome)()

# welcome因为有装饰器，所以调用时不是调用welcome函数,而是相当于color(welcome)() ..

# color(welcome)返回red，color(welcome)(),# 等价于red()

利用参数组,做更方便的装饰器:

def color(func):

def red(\*args):

return '\033[31;1m%s\033[0m' % func(\*args)

return red

@color

def hello(word):

return 'Hello %s!' % word

@color

def welcome():

return 'How are you?'

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

print(hello('China'))

print(welcome())

# OOP基础

## 基本概念

• 类(Class):用来描述具有相同的属性和方法的对象的集合。

它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。

• 实例化:创建一个类的实例,类的具体对象。

• 方法:类中定义的函数。

• 对象:通过类定义的数据结构实例。对象包括两个数据成员(类变量和实例变量)和方法。

创建类

• 使用 class 语句来创建一个新类,class 之后为类的名称并以冒号结尾

• 类名建议使用驼峰形式

class BearToy:

pass

创建实例

• 类是蓝图,实例是根据蓝图创建出来的具体对象

tidy = BearToy()

定义变量:

class A:

count=0 //类的变量,可以A.count, a.count访问

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

self.name=’bob’ //实例的变量,只能a.count访问

a=A()

python中的\_\_dict\_\_属性了,\_\_dict\_\_是一个字典，键是属性名，值为属性值。

Python的实例有自己的\_\_dict\_\_，它对应的类也有自己的\_\_dict\_\_   （但是有些特殊的对象是没有\_\_dict\_\_属性的，这里不做讨论）

一个对象的属性查找顺序遵循首先查找实例对象自己，然后是类，接着是类的父类。

print(A.\_\_dict\_\_) //{'\_\_module\_\_': '\_\_main\_\_', 'count': 0, '\_\_init\_\_': <function A.\_\_init\_\_ at 0x7f989b7cb1e0>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'A' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'A' objects>, '\_\_doc\_\_': None}

print(a.\_\_dict\_\_) //{'name': 'bob'}

构造器方法

• 当实例化类的对象时,构造器方法默认自动调用

• 实例本身作为第一个参数,传递给self

析构函数是\_\_del\_\_

class BearToy:

def \_\_init\_\_(self, nm, size, color):

'''\_\_init\_\_在实例化时自动执行，实例本身自动作为第一个参数传递给self'''

self.size = size #绑定属性到实例

self.color = color

self.name = nm

def sing(self) ；

print('lalala....')

def speak(self) :

print('My name is %s' % self.name)

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

tidy = BearToy('Tiny','small', 'orange') #调用\_\_init\_\_

print(tiny.size)

print(tiny.color)

tidy.sing()

tidy.speak()

## 组合

什么是组合

• 类被定义后,目标就是要把它当成一个模块来使用,并把这些对象嵌入到你的代码中去

• 组合就是让不同的类混合并加入到其它类中来增加功能和代码重用性

• 可以在一个大点的类中创建其它类的实例,实现一些其它属性和方法来增强对原来的类对象

组合应用

• 两个类明显不同

• 一个类是另一个类的组件

class Vendor:

def \_\_init\_\_(self, phone, email):

self.phone = phone

self.email = email

def call(self):

print('calling %s' % self.phone)

class BearToy:

def \_\_init\_\_(self,color,size,phone,email):

self.color = color

self.size = size

self.vendor = Vendor(phone, email)

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

bigbear = BearToy('Brown', 'Middle','4008009999', 'sale@tedu.cn')

print(bigbear.color)

bigbear.vendor.call()

## 派生/继承

• 当类之间有显著的不同,并且较小的类是较大的类所需要的组件时组合表现得很好;

但当设计“相同的类但有一些不同的功能”时,派生就是一个更加合理的选择了

• OOP 的更强大方面之一是能够使用一个已经定义好的类,扩展它或者对其进行修改,

而不会影响系统中使用现存类的其它代码片段

• OOD(面向对象设计)允许类特征在子孙类或子类中进行继承

• 创建子类只需要在圆括号中写明从哪个父类继承即可

继承

• 继承描述了基类的属性如何“遗传”给派生类

• 子类可以继承它的基类的任何属性,不管是数据属性还是方法

class BearToy :

def \_\_init\_\_(self,nm, color, size):

self.name = nm

self.color = color

self.size = size

def sing(self):

print('lalala.....')

def speak(self):

print('my nams is %s' % self.name)

class NewBear(BearToy):

def run(self):

print('running...')

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

b1 = NewBear('venie','Brown','Large')

b1.sing()

b1.run()

### 通过继承覆盖方法

• 如果子类中有和父类同名的方法,父类方法将被覆盖

• 如果需要访问父类的方法,则要调用一个未绑定的父类方法,明确给出子类的实例

class BearToy :

def \_\_init\_\_(self,nm, color, size):

self.name = nm

self.color = color

self.size = size

def sing(self):

print('lalala.....')

def speak(self):

print('my nams is %s' % self.name)

class NewBear(BearToy):

def \_\_init\_\_(self,nm,color,size,date):

#BearToy.\_\_init\_\_(self,nm, color, size)

super(NewBear,self).\_\_init\_\_(nm,color,size) //调用父类构造器

self.date = date

def run(self):

print('running...')

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

b1 = NewBear('venie','Brown','Large','1992-01-01')

b1.sing()

b1.run()

print(b1.date)

### 多重继承

• python允许多重继承,即一个类可以是多个父类的子类,子类可以拥有所有父类的属性

**如果多个父类里有重名的方法，则使用的顺序是：自下向上，自左向右**

class A:

def foo(self):

print('in A foo')

def hello(self):

print('A hello')

class B:

def bar(self):

print('in B bar')

def hello(self):

print('B hello')

class C(B,A):

pass

# def hello(self):

# print('C hello')

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

c.foo()

d.bar()

c.hello()

输出的记过是B hello

如果C 有hello方法，则打印C hello

## 特殊方法

**类方法**

• 使用classmethod装饰器定义

• 第一个参数cls表示类本身

class Date :

def \_\_init\_\_(self,year, month, date):

self.year = year

self.month = month

self.date = date

@classmethod #类方法，不用创建实例即可调用

def create(cls,dstr): #cls表示类本身，class

y, m, d = map(int,dstr.split('-')) //列表元素全部转换成int

dt = cls(y, m, d)

return dt

**静态方法**

• 基本上就是一个函数

• 在语法上就像一个方法

• 没有访问对象和它的字段或方法

• 使用staticmethod装饰器定义

@staticmethod #一个和此类不相关的函数，强行加入此类，要加上静态

def is\_date\_valid(dstr):

y, m, d = map(int,dstr.split('-'))

return 1<= d <= 31 and 1<= m <=12 and y<4000

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

bith\_date = Date(1995,10,3)

day = Date.create('2017-01-02')

print(day)

## 魔法方法

\_\_init\_\_方法: 构造函数,在对象被创建的时候自动调用

\_\_del\_\_方法：属于python语言的析构函数，它在对象消逝的时候被调用。

当对象不再被使用时，\_\_del\_\_方法运行，但是很难保证这个方法究竟在 什么时候 运行。如果你想要指明它的运行，你就得使用del语句

\_\_str\_\_方法

\_\_call\_\_方法

class Book:

def \_\_init\_\_(self, title, author, pages):

self.title = title

self.author = author

self.pages = pages

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

return '《%s》' % self.title

def \_\_call\_\_(self):

print('《%s》is written by %s' % (self.title, self.author))

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

py\_book = Book('Core Python', 'Wysley', 800) //自动调用\_\_init\_\_

print(py\_book) #自动调用\_\_str\_\_

py\_book() # 自动调用\_\_call\_\_

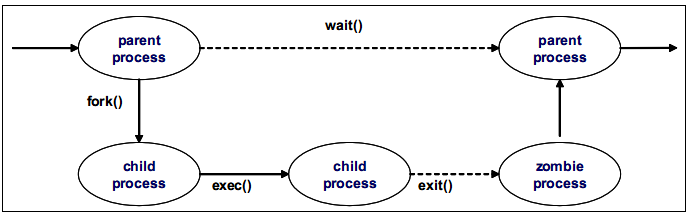
# 多线程多进程

## 多进程及案例

什么是forking

• fork(分岔)在Linux系统中使用非常广泛. 当某一命令执行时,父进程(当前进程)fork出一个子进程. 父进程将自身资源拷贝一份,命令在子进程中运行时,就具有和父进程完全一样的运行环境进程的生命周期.

• 父进程fork出子进程并挂起(系统默认的流程,通过wait()挂起,一般不挂起), 子进程运行完毕后,释放大部分资源并通知父进程,这个时候,子进程被称作僵尸进程, 父进程获知子进程结束,子进程所有资源释放僵尸进程.



• 僵尸进程没有任何可执行代码,也不能被调度,不消耗cpu,消耗内存. 如果系统中存在过多的僵尸进程,将因为没有可用的进程号而导致系统不能产生新的进程.• 对于系统管理员来说,可以试图杀死其父进程或重启系统来消除僵尸进程

forking编程

• 需要使用os模块

• os.fork()函数实现forking功能

• python中,绝大多数的函数只返回一次,os.fork将返回两次

• 对fork()的调用,针对父进程返回子进程的PID;对于子进程,返回PID0

• 因为所有的父子进程拥有相同的资源,所以在编写程序时要避免资源冲突

import os

print("starting........")

pid=os.fork() //创建子进程,返回值是个数字,对于父进程,返回值是PID,子进程是0

if pid:

print('in parent')

else:

print('in child')

**forking炸弹**

import os

for i in range(5):

pid = os.fork() # 父进程的工作是生成子进程

if not pid: # 如果是子进程，工作完后，结束，不要进入循环

print('hello')

#exit() 如果子进程没有exit,子进程也将进入循环,继续产生子进程.将以指数级别增加子进程数量

案例2:扫描存活主机

1. 通过ping测试主机是否可达

2. 如果ping不通,不管什么原因都认为主机不可用

3. 通过fork方式实现并发扫描

import subprocess

import os

def ping(host):

rc = subprocess.call(

'ping -c2 %s &> /dev/null' % host,

shell=True

)

if rc:

print('%s: down' % host)

else:

print('%s: up' % host)

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

ips = ['172.40.58.%s' % i for i in range(1, 255)]

for ip in ips:

pid = os.fork()

if not pid:

ping(ip)

exit()

使用轮询解决zombie问题

• 父进程通过os.wait()来得到子进程是否终止的信息

• 在子进程终止和父进程调用wait()之间的这段时间,子进程被称为zombie(僵尸)进程

• 如果子进程还没有终止,父进程先退出了,那么子进程会持续工作。系统自动将子进程的父进程设置为init进程(根进程,rhel7为systemd),init将来负责清理僵尸进程

• python可以使用waitpid()来处理子进程

• waitpid()接受两个参数,第一个参数设置为-1,表示与wait()函数相同;第二参数如果设置为0表示挂起父进程,直到子程序退出,设置为1表示不挂起父进程

• waitpid()的返回值:如果子进程尚未结束则返回0,否则返回子进程的PID

import os

import time

pid = os.fork()

if pid:

print('In parent. sleeping...')

print(os.waitpid(-1, 1)) # 无僵尸进程可以处理，返回0

time.sleep(20)

print(os.waitpid(-1, 1)) # 处理僵尸进程，返回子进程PIP

time.sleep(60)

print('parent done.')

else:

print('in child. sleeping...')

time.sleep(10)

print('child done')

# watch -n1 ps a 当子进程成为僵尸进程时，显示为Z

# kill 试图杀死僵尸进程、父进进程，查看结果

forking服务器

• 在网络服务器中,forking被广泛使用

• 如果服务器需要同时响应多个客户端,那么forking是解决问题最常用的方法之一

• 父进程负责接受客户端的连接请求

• 子进程负责处理客户端的请求

案例3:利用fork创建TCP服务器

• 编写TCP服务器

1. 服务器监听在0.0.0.0的21567端口上

2. 收到客户端数据后,将其加上时间戳后回送给客户端

3. 如果客户端发过来的字符全是空白字符,则终止与客户端的连接

4. 服务器能够同时处理多个客户端的请求

5. 程序通过forking来实现多线程编程

import socket

import os

from time import strftime

class TcpTimeServer:

def \_\_init\_\_(self, host='', port=12345):

self.addr = (host, port)

self.serv = socket.socket()

self.serv.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.serv.bind(self.addr)

self.serv.listen(1)

def chat(self, c\_sock):

while True:

data = c\_sock.recv(1024)

if data.strip() == b'quit':

break

data = '[%s] %s' % (strftime('%H:%M:%S'), data.decode('utf8'))

c\_sock.send(data.encode('utf8'))

c\_sock.close()

def mainloop(self):

while True:

cli\_sock, cli\_addr = self.serv.accept()

pid = os.fork()

if pid:

cli\_sock.close() #父进程不需要了

while True:

result = os.waitpid(-1, 1)[0] # 优先处理僵尸进程,返回0表示没有僵尸进程了

if result == 0:

break

else:

self.serv.close() #因为子进程复制了父进程的资源,所以需要关闭

self.chat(cli\_sock)

exit()

self.serv.close()

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

s = TcpTimeServer()

s.mainloop()

## 多线程及案例

**多线程**的动机

• 在多线程(MT)编程出现之前,电脑程序的运行由一个执行序列组成,执行序列按顺序在主机的中央处理器(CPU)中运行

• 无论是任务本身要求顺序执行还是整个程序是由多个子任务组成,程序都是按这种方式执行的

• 即使子任务相互独立,互相无关(即,一个子任务的结果不影响其它子任务的结果)时也是这样

• 如果并行运行这些相互独立的子任务可以大幅度地提升整个任务的效率

**多线程任务的工作特点**

• 它们本质上就是异步的,需要有多个并发事务

• 各个事务的运行顺序可以是不确定的,随机的,不可预测的

• 这样的编程任务可以被分成多个执行流,每个流都有一个要完成的目标

• 根据应用的不同,这些子任务可能都要计算出一个中间结果,用于合并得到最后的结果

什么是进程

• 计算机程序只不过是磁盘中可执行的、二进制(或其它类型)的数据

• 进程(有时被称为重量级进程)是程序的一次执行

• 每个进程都有自己的地址空间、内存以及其它记录其运行轨迹的辅助数据

• 操作系统管理在其上运行的所有进程,并为这些进程公平地分配时间

什么是线程

• 线程(有时被称为轻量级进程)跟进程有些相似。不同的是,所有的线程运行在同一个进程中,共享相同的运行环境

• 线程有开始,顺序执行和结束三部分

• 线程的运行可能被抢占(中断),或暂时的被挂起(也叫睡眠),让其它的线程运行,这叫做让步

• 一个进程中的各个线程之间共享同一片数据空间,所以线程之间可以比进程之间更方便地共享数据以及相互通讯

• 线程一般都是并发执行的,正是由于这种并行和数据共享的机制使得多个任务的合作变为可能

• 需要注意的是,在单CPU 的系统中,真正的并发是不可能的,每个线程会被安排成每次只运行一小会,然后就把CPU 让出来,让其它的线程去运行

**多线程编程**

多线程相关模块

• thread和threading模块允许程序员创建和管理线程

• thread模块提供了基本的线程和锁的支持,而threading提供了更高级别、功能更强的线程管理功能

• 推荐使用更高级别的threading模块

• 只建议那些有经验的专家在想访问线程的底层结构的时候,才使用thread模块传递函数给Thread类

• 多线程编程有多种方法,

**传递函数给threading模块的Thread类**

• Thread对象使用start()方法开始线程的执行,使用join()方法挂起程序,直到线程结束

import threading

import time

nums = [4, 2]

def loop(nloop, nsec): #定义函数,打印运行的起止时间

print('start loop %d, at %s' % (nloop, time.ctime()))

time.sleep(nsec)

print('loop %d done at %s' % (nloop, time.ctime()))

def main():

print('starting at: %s' % time.ctime())

threads = []

for i in range(2): #创建两个线程,放入列表

t = threading.Thread(target=loop, args=(0,nums[i]))

threads.append(t)

for i in range(2):

threads[i].start() #同时运行两个线程

for i in range(2):

threads[i].join() #主程序挂起,直到所有线程结束

print('all Done at %s' % time.ctime())

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

main()

•  **传递可调用类给Thread类**

• 相对于一个或几个函数来说,由于类对象里可以使用类的强大的功能,可以保存更多的信息,这种方法更为灵活

#!/usr/bin/env python

import threading

import time

nums = [4, 2]

class ThreadFunc(object): #定义可调用的类

def \_\_init\_\_(self, func, args, name = ''):

self.name = name

self.func = func

self.args = args

def \_\_call\_\_(self):

apply(self.func, self.args)

def loop(nloop, nsec): #定义函数,打印运行的起止时间

print('start loop %d, at %s' % (nloop, time.ctime()))

time.sleep(nsec)

print('loop %d done at %s' % (nloop, time.ctime()))

def main():

print('starting at: %s' % time.ctime())

threads = []

for i in range(2):

t = threading.Thread(target = ThreadFunc(loop, (i, nums[i]),

loop.\_\_name\_\_))

threads.append(t) #创建两个线程,放入列表

for i in range(2):

threads[i].start()

for i in range(2):

threads[i].join()

print('all Done at %s' % time.ctime())

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

main()

含有线程的服务器

• 多数的线程服务器有同样的结构

• 主线程是负责侦听请求的线程

• 主线程收到一个请求的时候,新的工作线程会被建立起来,处理客户端请求

• 客户端断开时,工作线程将终止

• 线程划分为用户线程和后台(daemon)进程,setDaemon将线程设置为后台进程

**案例4:**扫描存活主机

1. 通过ping测试主机是否可达

2. 如果ping不通,不管什么原因都认为主机不可用

3. 通过多线程方式实现并发扫描

threading 方法调用

import subprocess

import threading

def ping(host):

rc = subprocess.call(

'ping -c2 %s &> /dev/null' % host,

shell=True

)

if rc:

print('%s: down' % host)

else:

print('%s: up' % host)

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

ips = ['172.40.58.%s' % i for i in range(1, 255)]

ts=[]

for ip in ips:

# 创建线程，ping是上面定义的函数, args是传给ping函数的参数

t = threading.Thread(target=ping, args=(ip,)) args必须是元组

t.start() # 执行ping(ip)

ts.append(t)

for t in ts:

t.join() #等待线程结束

threading类 调用类

import threading

import subprocess

class Ping:

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

self.host = host

def \_\_call\_\_(self):

rc = subprocess.call(

'ping -c2 %s &> /dev/null' % self.host,

shell=True

)

if rc:

print('%s: down' % self.host)

else:

print('%s: up' % self.host)

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

ips = ('172.40.58.%s' % i for i in range(1, 255)) # 列表解析使用()创建生成器

for ip in ips:

t = threading.Thread(target=Ping(ip)) # 创建Ping的实例

t.start() # target()

python的多线程有一个GIL(全局解释器),使得多个线程,在某一时刻只有一个线程发送cpu处理. 不能有效的减少运行时间.

计算密集型应用:多线程不适合. 1+500000 .不能有效减少时间

IO密集型应用:适合多线程.可以有效减少运行时间

多进程就没有这个问题

windows系统不支持fork，只支持多线程

案例5:创建多线程时间戳服务器

• 编写一个TCP服务器

1. 服务器监听在0.0.0.0的12345端口上

2. 收到客户端数据后,将其加上时间戳后回送给客户端

3. 如果客户端发过来的字符全是空白字符,则终止与客户端的连接

4. 要求能够同时处理多个客户端的请求

5. 要求使用多线程的方式进行编写总结和答疑

import socket

import threading

from time import strftime

class TcpTimeServer:

def \_\_init\_\_(self, host='', port=12345):

self.addr = (host, port)

self.serv = socket.socket()

self.serv.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.serv.bind(self.addr)

self.serv.listen(1)

def chat(self, c\_sock):

while True:

data = c\_sock.recv(1024)

if data.strip() == b'quit':

break

data = '[%s] %s' % (strftime('%H:%M:%S'), data.decode('utf8'))

c\_sock.send(data.encode('utf8'))

c\_sock.close()

def mainloop(self):

while True:

cli\_sock, cli\_addr = self.serv.accept()

t = threading.Thread(target=self.chat, args=(cli\_sock,))

t.start()

self.serv.close()

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

s = TcpTimeServer()

s.mainloop()

# Ansible编程基础

ansible的安装使用见 服务.docx

ansible官方文档查看url

https://docs.ansible.com/ansible/latest/index.html

搜索Python API查看相关内容,有例子

命名元组

• 命名元组与普通元组一样,有相同的表现特征,其添加的功能就是可以根据名称引用元组中的项

• collections 模块提供了namedtuple()函数,用于创建自定义的元组数据类型

from collections import namedtuple

u1 = ('bob', 25, 'bob@tedu.cn')

u1[0] //只能通过下标访问

user = namedtuple('user', ['name', 'age', 'email']) //定义命名元组. 注意两个user需要命名一致

u2 = user('alice', 23, 'alice@tarena.com') //实例化

print(u2[1])

print(u2[1:])

print(u2.email) //可以通过下标访问,也可以通过属性访问

--------------------------------------------------------------------------------------------------

user(name='alice', age=23, email='alice@tedu.cn')

23

alice@tedu.cn

Ansible常用属性

• from ansible.parsing.dataloader import DataLoader

– 用来加载解析yaml文件或JSON内容,并且支持vault的解密

• from ansible.vars.manager import VariableManager

– 管理变量的类,包括主机,组,扩展等变量

• from ansible.inventory.manager import InventoryManager

– 用于创建主机清单,主机清单的源采用配置文件或是逗号分开主机名字符串

• from ansible.playbook.play import Play

– 用于创建play对象,能够通过play\_source提供的信息自动创建任务对象

• from ansible.executor.task\_queue\_manager import TaskQueueManager

– 用于处理进程池中的多进程。队列管理器负责加载play策略插件,以便在选定的主机上执行任务

• import ansible.constants as C

– 存储ansible一些预定义变量

ansible只支持python2. pycharm修改解释器,file->setting->projectname ->interpreter

python2 中默认不支持tab补齐,需要以下操作:

vim /usr/local/bin/tab.py

from rlcompleter import readline

readline.parse\_and\_bind('tab: complete')

vim ~/.bash\_profile

export PYTHONSTARTUP=/usr/local/bin/tab.py

source ~/.bash\_profile

python2中，print不是函数

>>> print 'hello world!'

>>> print 10

python2中，输入不采用input()，而是raw\_input()

## Python使用ansible模块

**使用ansible 模块，以下内容从ansible官方文档 Python API里的第一个例子获取**

#!/usr/bin/env python

# coding: utf8 ---------------> python2 默认ascii，这里改成utf8

import json

import shutil

from collections import namedtuple

from ansible.parsing.dataloader import DataLoader

from ansible.vars.manager import VariableManager

from ansible.inventory.manager import InventoryManager

from ansible.playbook.play import Play

from ansible.executor.task\_queue\_manager import TaskQueueManager

import ansible.constants as C

# 删除了CallBack部分，暂不使用自定义输出

Options = namedtuple('Options', ['connection', 'module\_path', 'forks', 'become', 'become\_method', 'become\_user', 'check', 'diff'])

#ansible命令的选项,编程的时候也得涉及

# 连接可以设置为local/ssh/smart

# local表示本地执行，ssh表示使用ssh连接，smart表示智能判断

#forks=10 表示始终保持10个子进程

options = Options(connection='local', module\_path=['/to/mymodules'], forks=10, become=None, become\_method=None, become\_user=None, check=False, diff=False)

loader = DataLoader() # 用于分析json/yml文件

passwords = dict(vault\_pass='secret') #把括号里面删除，因为是免密登录,变成passwords=dict()

#主机清单,有两种表示方法，可以使用字符串，使用逗号分割，也可以使用列表，填写的是清单文件地址

#inventory = InventoryManager(loader=loader, sources='localhost,node4.tedu.cn')

inventory = InventoryManager(loader=loader, sources=['/root/myansi/hosts'])

#变量管理器，可用于连接的端口不是默认端口时

variable\_manager = VariableManager(loader=loader, inventory=inventory)

play\_source = dict(

name = "Ansible Play",

hosts = 'localhost', #这里是执行的主机，可以填写上面主机清单里的主机或组，例如 webservers

gather\_facts = 'no',

tasks = [

dict(action=dict(module='shell', args='ls'), register='shell\_out'),

# dict(action=dict(module='shell', args='ls'), register='shell\_out'),

# dict(action=dict(module='shell', args='id zhangsan'), register='shell\_out'),

# dict(action=dict(module='shell', args='useradd zhangsan'), register='shell\_out'),

dict(action=dict(module='yum', args='name=vsftpd state=latest'), register='shell\_out'),

# dict(action=dict(module='debug', args=dict(msg='{{shell\_out.stdout}}')))

dict(action=dict(module='debug', args=dict(msg='{{shell\_out.stdout}}')))

]

#task里面存放着执行的命令.debug模块保存执行输出信息.创建用户用user模块

)

# 这里是把上面设置的执行数据载入为Play()

play = Play().load(play\_source, variable\_manager=variable\_manager, loader=loader)

tqm = None

try:

tqm = TaskQueueManager(

inventory=inventory,

variable\_manager=variable\_manager,

loader=loader,

options=options,

passwords=passwords,

)

result = tqm.run(play) 最终依靠tqm.run来执行所有设置的命令

finally:

if tqm is not None:

tqm.cleanup() 清除tqm的执行

shutil.rmtree(C.DEFAULT\_LOCAL\_TMP, True) 删除执行过程中产生的临时文件

××××××××××××××××××××××××××××××××××××××××××

上述代码，有4个地方需要修改的，

1、connection 部分的连接方式 local ssh smart， fork 的进程数

2、passwords=dict() 括号里面的可以删除，因为已经设置免密登录

3、inventory里面的主机清单修改

4、play\_source 里面需要执行命令的hosts 和对应的命令

**回调函数**

def foo(func, \*args):

print 'processing...'

func(\*args)

def loop(num):

for i in range(num):

print i

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

foo(loop, 10)

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

## python调用playbook

Playbook编程概述

• Playbooks 是 Ansible的配置、部署、编排语言

• 它们可以被描述为一个需要希望远程主机执行命令的方案或者一组IT程序运行的命令集合

• 可以通过python编程的方式执行playbook

• 与执行ad-hoc命令一样,playbook的执行也需要相关模块

from collections import namedtuple

from ansible.parsing.dataloader import DataLoader

from ansible.vars.manager import VariableManager

from ansible.inventory.manager import InventoryManager

from ansible.executor.playbook\_executor import PlaybookExecutor

Options = namedtuple(

'Options', [

'connection',

'remote\_user',

'ask\_sudo\_pass',

'verbosity',

'ask\_pass',

'module\_path',

'forks',

'become',

'become\_method',

'become\_user',

'check',

'listhosts',

'listtasks',

'listtags',

'syntax',

'sudo\_user',

'sudo',

'diff'

]

)

options = Options(

connection='smart',

remote\_user=None,

ask\_pass=None,

sudo\_user=None,

forks=5,

sudo=None,

ask\_sudo\_pass=None,

verbosity=5,

module\_path=None,

become=None,

become\_method=None,

become\_user=None,

check=False,

diff=False,

listhosts=None,

listtasks=None,

listtags=None,

syntax=None

)

loader = DataLoader()

passwords = dict()

inventory = InventoryManager(loader=loader, sources=['/root/myansi/hosts'])

variable\_manager = VariableManager(loader=loader, inventory=inventory)

def playbook\_run(playbook\_path):

playbook = PlaybookExecutor(

playbooks=playbook\_path,

inventory=inventory,

variable\_manager=variable\_manager,

loader=loader,

options=options,

passwords=passwords

)

result = playbook.run()

return result

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

playbook\_run(playbook\_path=['/root/myansi/web\_db.yml'])

## 编写ansible模块(了解,一般调用已有的官方模块)

官方文档: docs.ansible.com

官方模块

• Ansible官方已经提供了大量模块,在编写模块之前,可以查看是否已有现成模块

• 官方已发布模块

– http://docs.ansible.com/ansible/modules.html

• 官方正在开发的模块

– https://github.com/ansible/ansible/labels/module

模块执行流程

• 将模块文件读入内存,然后添加传递给模块的参数,最后将模块中所需要的类添加到内存,由zipfile压缩后,再由base64进行编码,写入到模板文件内

• 通过默认的连接方式(一般是ssh),ansible连接到远程主机,创建临时目录,并关闭连接

• 打开另外一个ssh连接,将模板文件以sftp方式传送到刚刚创建的临时目录中,写完后关闭连接

• 打开一个ssh连接将任务对象赋予可执行权限,执行成功后关闭连接

• 最后,ansible将再打开一个新连接来执行模块,并删除临时目录及其所有内容

• 模块的结果是从标准输出stdout中获取json格式的字符串。ansible将解析和处理此字符串

管道连接

• 使用了管道连接后,与远程主机只有一个连接,命令通过数据流的方式发送执行

• 此模式与有些系统程序兼容不太好

• 配置方式:

[root@room9pc01 myansi]# vim /etc/ansible/ansible.cfg（这里应该在当前目录下的ansible.cfg文件里添加一行）

pipelining = True //只建立一个连接,把所有的活都干了

模块开发(创建拷贝文件的模块)

模块库目录

• 可以使用 ANSIBLE\_LIBRARY环境变量来指定模块的存放位置

• 也可以在playbook当前目录下创建library目录

1、在工作目录下创建模块目录

[root@room8pc16 myansi]# mkdir library

2、在模块目录下创建自己的模块remote\_copy.py

[root@room8pc16 myansi]# cd library

[root@room8pc16 library]# vim remote\_copy.py

#!/usr/bin/env python

import shutil

from ansible.module\_utils.basic import AnsibleModule

def main():

module = AnsibleModule(

argument\_spec=dict(

source=dict(required=True, type='str'),

dest=dict(required=True, type='str')

)

)#编写模块头,创建模块入口, 使用AnsibleModule类中的argument\_spec来接受参数

shutil.copy(module.params['source'], module.params['dest']) #使用shutil.copy拷贝文件

module.exit\_json(changed=True) #拷贝完成后,返回json数据

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

main()

3、设置ansible查找模块的路径

[root@room8pc16 myansi]# export ANSIBLE\_LIBRARY=./library

4、测试

[root@room8pc16 myansi]# ansible node2.tedu.cn -m remote\_copy -a "source=/etc/passwd dest=/var/tmp/mima"

查看ansible 的选项 -M 如果不设置环境变量，可以把已经写好的模板放入下面的2个路径里面去

-M MODULE\_PATH, --module-path=MODULE\_PATH

prepend colon-separated path(s) to module library

(default=[u'/root/.ansible/plugins/modules',

u'/usr/share/ansible/plugins/modules'])

编写测试playbook

• 编写playbook,测试编写的模块

[root@localhost myansi]# vim mytest.yml

---

- name: test remote\_copy module

hosts: myself

tasks:

- name: remote\_copy file

remote\_copy:

source: /etc/hosts

dest: /tmp/zj.txt

• 执行playbook

[root@localhost myansi]# ansible-playbook mytest.yml

# 模块基础

**定义模块**

模块基本概念

• 模块是从逻辑上组织python代码的形式

• 当代码量变得相当大的时候,最好把代码分成一些有组织的代码段

• 这些代码片段相互间有一定的联系,可能是一个包含数据成员和方法的类,也可能是一组相关但彼此独立的操作函数

• 这些代码段是共享的,所以python允许“调入”一个模块,允许使用其他模块的属性来利用之前的工作成果,实现代码重用

创建模块

• 模块物理层面上组织模块的方法是文件,每一个以.py作为结尾的python文件都是一个模块

• 模块名称切记不要与系统中已存在的模块重名

• 模块文件名字去掉后面的扩展名(.py)即为模块名

**名称空间**

• 名称空间就是一个从名称到对象的关系映射集合

• 给定一个模块名之后,只可能有一个模块被导入到python解释器中,所以在不同模块间不会出现名称交叉现象

• 每个模块都定义了它自己的唯一的名称空间

>>> import foo

>>> import bar

>>> print(foo.hi) #调用foo模中的hi变量

hello

>>> print(bar.hi) #调用bar模块中的hi变量

greet

**搜索路径**

• 模块的导入需要一个叫做“路径搜索”的过程

• python在文件系统“预定义区域”中查找要调用的模块

• 搜索路径在sys.path中定义

>>> import sys

>>> print(sys.path)

['', '/usr/local/lib/python36.zip', '/usr/local/lib/python3.6', '/usr/

local/lib/python3.6/lib-dynload', '/usr/local/lib/python3.6/site-

packages']

**/usr/local/lib/python3.6/site-packages** //我们自己写的程序可以放到这里去.

模块导入方法

• 使用import导入模块

• 可以在一行导入多个模块,但是可读性会下降

• 可以只导入模块的某些属性

• 导入模块时,可以为模块取别名

import time, os, sys

from random import choice

import pickle as p //取别名

模块加载(load)

• 当导入模块时,模块的顶层代码会被执行

• 一个模块不管被导入(import)多少次,只会被加载(load)一次

[root@py01 ~]# cat foo.py

hi = 'hello'

print(hi)

[root@py01 ~]# python3

>>> import foo

Hello #第一次导入,执行print语句

>>> import foo #再次导入,print语句不再执行

**从zip文件中导入**

• 在2.3版中,python加入了从ZIP归档文件导入模块的功能

• 如果搜索路径中存在一个包含python模块(.py、.pyc、或.pyo文件)的.zip文件,

导入时会把ZIP文件当作目录处理

#导入sys模块,在搜索路径中加入相应的zip文件

>>> import sys

>>> sys.path.append('/root/pymodule.zip')

>>> import foo #导入pymodule.zip压缩文件中的foo模块

# foo.py -> hi = 'hello'

#print(hi)

#zip mytest.zip foo.py

#rm -rf foo.py

#python3

#>>> import foo ->Error

#>>> import sys

#>>> sys.path.append('./mytest.zip')

#>>> import foo

**包**

目录结构

• 包是一个有层次的文件目录结构,为平坦的名称空间加入有层次的组织结构

• 允许程序员把有联系的模块组合到一起

• 包目录下必须有一个\_\_init\_\_.py文件 python2必须有,python3不是必须的

phone/

\_\_init\_\_.py

common\_uDl.py

voicedata/

\_\_init\_\_.py

post.py

**绝对导入**

• 包的使用越来越广泛,很多情况下导入子包会导致和真正的标准库模块发生冲突

• 因此,所有的导入现在都被认为是绝对的,也就是说这些名字必须通过python路径(sys.path或PYTHONPATH)来访问

**相对导入**

• 绝对导入特性使得程序员失去了import的自由,为此出现了相对导入

• 因为import语句总是绝对导入的,所以相对导入只应用于from-import语句

[root@py01 ~]# ls -R phone/

phone/:

common\_util.py \_\_init\_\_.py voicedata

phone/voicedata:

\_\_init\_\_.py post.py

[root@py01 ~]# cat phone/voicedata/post.py

from .. import common\_util

**模块导入的特性**

• 模块具有一个\_\_name\_\_特殊属性

• 当模块文件直接执行时,\_\_name\_\_的值为'\_\_main\_\_'

• 当模块被另一个文件导入时,\_\_name\_\_的值就是该模块的名字

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

函数() //写成这样的格式,在import此模块的时候,函数不会被执行.如果直接运行模块,函数才会被执行

## hashlib模块(md5 sha 1,sha256,sha512,加密解密,完整性校验)

• hashlib用来替换md5和sha模块,并使他们的API一致,专门提供hash算法

• 包括md5、sha1、sha224、sha256、sha384、sha512,使用非常简单、方便

# 加密：对称加密、非对称加密、单向加密

# 单向加密：加密只能向一个方向进行，相同的数据总是得到相同的“乱码”

# 不能根据结果回推源。用途：文件完整性校验、加密密码

**案例:得到文件的MD5值**

import hashlib

f = open('/etc/passwd', 'rb')

data = f.read()

f.close()

m = hashlib.md5(data)//读入全部文件内容

print(m.hexdigest()) //生成文件md5值

//得到的54d21e3107b30a485e61cf0d4a15c6db

import hashlib

import sys

def check\_md5(fname):

m = hashlib.md5()

with open(fname, 'rb') as fobj:

while True:

data = fobj.read(4096)

if not data:

break

m.update(data)

return m.hexdigest()

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

print(check\_md5(sys.argv[1]))

//得到的54d21e3107b30a485e61cf0d4a15c6db

## re模块(正则)

贪婪匹配

• \*、+和?都是贪婪匹配操作符,在其后加上?可以取消其贪婪匹配行为

• 正则表达式匹配对象通过groups函数获取子组

>>> data = 'My phone number is: 150888899999'

>>> m = re.search('.+(\d+)', data)

>>> print m.groups()

('9',)

>>>

>>> m = re.search('.+?(\d+)', data)

>>> m.groups()

('150888899999',)

**核心函数和方法**

m = re.match('f..', 'food') # 尝试用正则表达式模式从字符串的开头匹配,如果匹配成功,则返回一个匹配对象;否则返回None

print(re.match('f..', 'seafood')) # 匹配不到返回None,返回

<\_sre.SRE\_Match object; span=(0, 3), match='foo'>

m.group() # 返回匹配的值,可以用print打印,得到foo

m = re.search('f..', 'seafood') #在字符串中查找正则表达式模式的第一次出现,如果匹配成功,则返回一个匹配对象;否则返回None

re.findall('f..', 'seafood is food') # 返回所有匹配项组成的列表 得到 ['foo', 'foo']

result = re.finditer('f..', 'seafood is food') # 返回匹配对象组成的迭代器 得到<callable\_iterator object at 0x7f026b81a470>

for m in result: # 从迭代器中逐个取出匹配对象

print(m.group())

re.sub('f..', 'abc', 'fish is food') # 替换, 用abc替换f..,返回字符串

re.split('\.|-', 'hello-world.tar.gz') # 用.和-做切割符号,返回成功匹配的列表. string也有这个功能,re模块更灵活

patt = re.compile('f..') # 先把要匹配的模式编译，回一个正则表达式对象.在大量匹配的情况下,可以提升效率

m = patt.search('seafood') # 指定在哪个字符串中匹配

**案例1:分析apache访问日志**

• 编写一个apche日志分析脚本

1. 统计每个客户端访问apache服务器的次数

2. 将统计信息通过字典的方式显示出来

3. 分别统计客户端是Firefox和MSIE的访问次数

4. 分别使用函数式编程和面向对象编程的方式实现

**函数式编程**

import re

def count\_patt(fname,patt):

result = {}

cpatt = re.compile(patt)

with open(fname) as fobj :

for line in fobj:

m = cpatt.search(line) #如果匹配不到,返回none

if m:

key = m.group()

result[key] = result.get(key, 0) +1

return result

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

fname = 'access\_log'

ip = '^(\d+\.){3}\d+' 统计ip

print(count\_patt(fname,ip))

br = 'Firefox|MSIE|Chrome' 统计浏览器

print(count\_patt(fname,br))

**面向对象编程**

import re

from collections import Counter

class anylize:

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

self.fname = fname

def count\_patt(self,patt):

cpatt = re.compile(patt)

result = Counter()

with open(self.fname) as fobj:

for line in fobj:

m = cpatt.search(line)

if m:

result.update([m.group()])

return result

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

A=anylize('access\_log')

ip = '^(\d+\.){3}\d+'

br = 'Firefox|MSIE|Chrome'

print(A.count\_patt(ip))

print(A.count\_patt(br))

## tarfile模块(解压,压缩)

• tarfile模块允许创建、访问tar文件

• 同时支持gzip、bzip2格式 (gz,bz))

调试的时候做好把文件删除,因为文件不正确的话,读进来为空.

**生成压缩文件**

import tarfile

tar = tarfile.open('/tmp/demo.tar.gz','w:gz') #gzip压缩

tar.add('/etc/passwd')

tar.add('/etc/hosts')

tar.close()

#tar tvzf /tmp/demo.tar.gz

**解压缩**

tar = tarfile.open('/tmp/demo.tar.gz','r:gz') # 解压所有文件到当前目录

tar.extractall() ##里面有path 和member的选项功能

tar.close()

arfile.open的mode有很多种：

|  |
| --- |
|  |
| 'r' or 'r:\*' | Open for reading with transparent compression (recommended). |
| 'r:' | Open for reading exclusively without compression. |
| 'r:gz' | Open for reading with gzip compression. |
| 'r:bz2' | Open for reading with bzip2 compression. |
| 'a' or 'a:' | Open for appending with no compression. The file is created if it does not exist. |
| 'w' or 'w:' | Open for uncompressed writing. |
| 'w:gz' | Open for gzip compressed writing. |
| 'w:bz2' | Open for bzip2 compressed writing. |

## collections的Counter模块(带统计排序功能的字典)

c = Counter('abcdeabcdabcaba') # count elements from a string //得到Counter({'a': 5, 'b': 4, 'c': 3, 'd': 2, 'e': 1})

c.most\_common(3) # three most common elementsCounter// [('a', 5), ('b', 4), ('c', 3)]

sorted(c) # list all unique elements // ['a', 'b', 'c', 'd', 'e']

sorted(c.elements()) # list elements with repetitions //['a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c', 'd', 'd', 'e']

可以使用字典的函数

c.values() #total of all counts //dict\_values([5, 4, 3, 2, 1])

c['a'] # count of letter 'a' // 5

for elem in 'shazam': # update counts from an iterable

c[elem] += 1 # by adding 1 to each element's count

c['a'] # now there are seven 'a'

del c['b'] # remove all 'b'

c['b'] # now there are zero 'b'

d = Counter('simsalabim') # make another counter

c.update(d) # add in the second counter

c['a'] # now there are nine 'a'

c.clear() # empty the counter

Note: If a count is set to zero or reduced to zero, it will remain

in the counter until the entry is deleted or the counter is cleared:

>>> c = Counter('aaabbc')

>>> c['b'] -= 2 # reduce the count of 'b' by two

>>> c.most\_common() # 'b' is still in, but its count is zero

[('a', 3), ('c', 1), ('b', 0)]

## shutil模块 (文件目录移动拷贝,权限)

**文件复制和移动**

• **shutil.copyfileobj**(fsrc, fdst[, length])

将类似文件的对象fsrc的内容复制到类似文件的对象fdst。文件对象可以是普通文件(文本文件,二进制文件),也可以是网络设备(socket)等

•  **shutil.copyfile**(src, dst, \*, follow\_symlinks=True)

将名为src的文件的内容(无元数据)复制到名为dst的文件,然后返回dst。src/dest 为string

• **shutil.copy**(src, dst, \*, follow\_symlinks=True)

将文件src复制到文件或目录dst。src和dst应为字符串。

如果dst指定目录,则文件将使用src的基本文件名复制到dst中。返回新创建的文件的路径。

• **shutil.copy2(**src, dst, \*, follow\_symlinks=True)

与copy()相同,但copy2()也尝试保留所有文件元数据(描述文件属性的数据)。

• **shutil.move**(src, dst, copy\_function=copy2)

递归地将文件或目录(src)移动到另一个位置(dst),并返回目标。 (删除文件可以移动到//dev/null)

**目录操作**

• **shutil.copytree**(src, dst, symlinks=False, ignore=None, copy\_function=copy2, ignore\_dangling\_symlinks=False)

递归地复制以src为根的整个目录树,返回目标目录。由dst命名的目标目录不能已经存在。

• **shutil.rmtree** (path, ignore\_errors=False, onerror=None)

删除整个目录树; 路径必须指向目录(而不是指向目录的符号链接)。

import shutil

with open('/etc/passwd','rb') as sfobj:

with open('/tmp/mima.txt','wb') as dfobj:

shutil.copyfileobj(sfobj,dfobj) #拷贝文件对象

shutil.copyfile('/etc/passwd', '/tmp/mima2.txt') # cp 1.txt 2.txt

shutil.copy('/etc/shadow', '/tmp/') # cp 1.txt /root/

shutil.copy2('/etc/shadow', '/tmp/') # cp -p 1.txt /root/

shutil.move('/tmp/mima.txt', '/var/tmp') # mv a.txt /root/

shutil.copytree('/var/log', '/tmp/logs') # cp -r /var/log /tmp/logs

shutil.rmtree('tmp/logs') # rm -rf /tmp/logs

**权限管理**

• **shutil.copymode**(src, dst, \*, follow\_symlinks=True)

将权限位从src复制到dst。文件内容,所有者和组不受影响。src和dst是以字符串形式给出的路径名称。

• **shutil.copystat**(src, dst, \*, follow\_symlinks=True)

将权限位,最后访问时间,上次修改时间和标志从src复制到dst。

• **shutil.chown**(path, user=None, group=None)

更改给定路径的所有者用户和/或组

shutil.copymode('/etc/shadow','/tmp/mima2.txt')###将mima2.txt的权限设置成与/etc/shadow一样

shutil.copystat('/etc/shadow','/tmp/mima2.txt')###将mima2.txt的元数据设置成与/etc/shadow一样，元数据使用stat /etc/shadow 查看

shutil.chown('/tmp/passwd',user='zhangsan',group='zhangsan')

## keywork模块(查看关键字,检查关键字)

import keyword

keyword.kwlist # 查看所有关键字

keyword,iskeyword(‘pass’) # 查看是否是关键字

## subprocess模块(调用系统命令)

subprocess.call('useradd %s' % username,shell=True) //调用系统命令

## socket模块

什么是C/S架构

• 服务器是一个软件或硬件,用于提供客户需要的“服务”

• 硬件上,客户端常见的就是平时所使用的PC机,服务器常见的有联想、DELL等厂商生产的各种系列服务器

• 软件上,服务器提供的服务主要是程序的运行,数据的发送与接收、合并、升级或其它的程序或数据的操作

套接字

• 套接字是一种具有“通讯端点”概念的计算机网络数据结构

• 套接字起源于20世纪70年代加利福尼亚大学伯克利分校版本的Unix

• 一种套接字是Unix套接字,其“家族名”为AF\_UNIX

• 另一种套接字是基于网络的,“家族名”为AF\_INET

• 如果把套接字比做电话的插口,那么主机与端口就像区号与电话号码的一对组合

面向连接与无连接

• 无论你使用哪一种地址家族,套接字的类型只有两种。一种是面向连接的套接字,另一种是无连接的套接字

• 面向连接的主要协议就是传输控制协议TCP,套接字类型为SOCK\_STREAM

• 无连接的主要协议是用户数据报协议UDP,套接字类型为SOCK\_DGRAM

• python中使用socket模块中的socket函数实现套接字的创建

socket函数与方法

### 创建TCP服务器

• 创建TCP服务器的主要步骤如下:

1. 创建服务器套接字:s = socket.socket()

2. 绑定地址到套接字:s.bind()

3. 启动监听:s.listen()

4. 接受客户连接:s.accept()

5. 与客户端通信:recv()/send()

6. 关闭套接字:s.close()

import socket

import threading

from time import strftime

class TcpTimeServer:

def \_\_init\_\_(self, host='', port=12345): # ‘’表示本机所有地址 0.0.0.0

# port应该大于1024

self.addr = (host, port)

self.serv = socket.socket() # 默认值就是基于tcp的网络套接字

self.serv.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1) # 设置选项,程序结束之后可以立即再运行,否则要等60秒

self.serv.bind(self.addr) # 绑定地址到套接字

self.serv.listen(1) 启动侦听进程

def chat(self, c\_sock):

while True:

data = c\_sock.recv(1024) 设置一次最多读1024字节数据

if data.strip() == b'quit':

break

data = '[%s] %s' % (strftime('%H:%M:%S'), data.decode('utf8'))

c\_sock.send(data.encode('utf8')) 发送的数据要求是bytes类型,发送网络数据一般都要加上\r\n. 把str类型转换成bytes类型.从网络接受到的数据需要decode,转换成str.

c\_sock.close()

def mainloop(self):

while True: //循环等待客户端链接

cli\_sock, cli\_addr = self.serv.accept()

t = threading.Thread(target=self.chat, args=(cli\_sock,))

t.start()

self.serv.close()

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

s = TcpTimeServer()

s.mainloop()

**创建TCP客户端**

• 创建TCP客户端的步骤主要如下:

1. 创建客户端套接字:cs = socket.socket()

2. 尝试连接服务器:cs.connect()

3. 与服务器通信:cs.send()/cs.recv()

4. 关闭客户端套接字:cs.close()

import socket

host = '192.168.1.254'

port = 12345

addr = (host, port)

c = socket.socket()

c.connect(addr)

while True:

data = input('> ') + '\r\n'

c.send(data.encode('utf8')) #服务器收到end结束,所以要先发送再判断

if data.strip() == 'end' :

break

data = c.recv(1024)

print(data.decode('utf8'))

c.close()

案例3:创建TCP时间戳服务端

import socket

from time import strftime

class TcpTimeServer:

def \_\_init\_\_(self, host='', port=12345):

self.addr = (host, port)

self.serv = socket.socket()

self.serv.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.serv.bind(self.addr)

self.serv.listen(1)

def chat(self, c\_sock):

while True:

data = c\_sock.recv(1024)

if data.strip() == b'quit':

break

data = '[%s] %s' % (strftime('%H:%M:%S'), data.decode('utf8'))

c\_sock.send(data.encode('utf8'))

c\_sock.close()

def mainloop(self):

while True:

cli\_sock, cli\_addr = self.serv.accept()

self.chat(cli\_sock)

self.serv.close()

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

s = TcpTimeServer()

s.mainloop()

### 创建UDP服务器及客户端

• 创建UDP服务器的主要步骤如下:

1. 创建服务器套接字:s = socket.socket()

2. 绑定服务器套接字:s.bind()

3. 接收、发送数据:s.recvfrom()/ss.sendto()

4. 关闭套接字:s.close()

import socket

from time import strftime

host = ''

port = 12345

addr = (host, port)

s = socket.socket(type=socket.SOCK\_DGRAM)

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

s.bind(addr)

while True:

data, cli\_addr = s.recvfrom(1024)

clock = strftime('%H:%M:%S')

data = data.decode('utf8')

data = '[%s] %s' % (clock, data)

s.sendto(data.encode('utf8'), cli\_addr)

s.close()

**创建UDP客户端**

• 创建UDP客户端的步骤主要如下:

1. 创建客户端套接字:cs = socket.socket()

2. 与服务器通信:cs.sendto()/cs.recvfrom()

3. 关闭客户端套接字:cs.close()

import socket

host = '192.168.4.254'

port = 12345 //可以同时开启tcp&udp 12345端口

addr = (host, port)

c = socket.socket(type=socket.SOCK\_DGRAM)

while True:

data = input('> ')

if data.strip() == 'quit':

break

c.sendto(data.encode('utf8'), addr)

print(c.recvfrom(1024)[0].decode('utf8'))

# print(c.recvfrom(1024))

c.close()

## time模块

时间表示方式

• 时间戳timestamp:表示的是从1970年1月1日00:00:00开始按秒计算的偏移量

• UTC(Coordinated Universal Time,世界协调时)亦即格林威治天文时间,世界标准时间。

在中国为UTC+8。DST(Daylight Saving Time)即夏令时

• 元组(struct\_time):由9个元素组成

struct\_time元组

索引 属性 值

0 tm\_year 2000

1 tm\_mon 1-12

2 tm\_mday 1-31

3 tm\_hour 0-23

4 tm\_min 0-59

5 tm\_sec 0-61

6 tm\_wday 0-6(0表示周一)

7 tm\_yday(一年中的第几天) 1-366

8 tm\_isdst(是否为dst时间) 默认为-1

time模块方法.蓝色需要掌握

• time.localtime([secs]):将一个时间戳转换为当前时区的struct\_time。secs参数未提供,则以当前时间为准

• time.gmtime([secs]):和localtime()方法类似,

gmtime()方法是将一个时间戳转换为UTC时区(0时区)的struct\_time

• time.time():返回当前时间的时间戳

• time.mktime(t):将一个struct\_time转化为时间戳

• time.sleep(secs):线程推迟指定的时间运行。单位为秒

• time.asctime([t]):把一个表示时间的元组或者struct\_time表示为

这种形式:‘Sun Jun 20 23:21:05 1993’。如果没有参数,将会将time.localtime()作为参数传入

• time.ctime([secs]):把一个时间戳(按秒计算的浮点数)转化为time.asctime()的形式

• time.strftime(format[, t]):把一个代表时间的元组或者struct\_time(如由time.localtime()和time.gmtime()返回)转化为格式化的时间字符串。如果t未指定,将传入time.localtime()

• time.strptime(string[, format]):把一个格式化时间字符串转化为struct\_time。

实际上它和strftime()是逆操作

>>>time.strftime('%Y-%m-%d %X', time.localtime())

'2017-12-12 12:58:19'

时间样式

格式 含义 格式 含义

%a 本地简化星期名称 %m 月份(01-12)

%A 本地完整星期名称 %M 分钟数(00-59)

%b 本地简化月份名称 %p 本地am或者pm的相应符

%B 本地完整月份名称 %S 秒(01-61)

%c 本地相应的日期和时间 %U 一年中的星期数(00–53,星期日是一个星期的开始)

%d 一个月中的第几天(01-31) %w 一个星期中的第几天(0-6,0是星期天)

%H 一天中的第几个小时(24小时制,00-23) %x 本地相应日期

%I 第几个小时(12小时制,01-12) %X 本地相应时间

%j 一年中的第几天(001-366) %y 去掉世纪的年份(00-99)

%Z 时区的名字 %Y 完整的年份

## datetime模块

• datetime.today():返回一个表示当前本地时间的datetime对象

• datetime.now([tz]):返回一个表示当前本地时间的datetime对象,

如果提供了参数tz,则获取tz参数所指时区的本地时间

• datetime.strptime(date\_string, format):将格式字符串转换为datetime对象

• datetime.ctime(datetime对象):返回时间格式字符串

• datetime.strftime(format):返回指定格式字符串

时间计算

• 使用timedelta可以很方便的在日期上做天days,小时hour,分钟,秒,毫秒,微妙的时间计算

>>> dt = datetime.datetime.now()

>>> days = datetime.timedelta(days=100, hours=3)

>>> dt + days

datetime.datetime(2050, 6, 10, 20, 41, 20, 106546)

## os模块简介(跟文件相关的操作)

• 对文件系统的访问大多通过python的os模块实现

• 该模块是python访问操作系统功能的主要接口

• 有些方法,如copy等,并没有提供,可以使用shutil模块作为补充

函数

symlink() 创建符号链接

chmod() 改变权限模式

getatime() 返回最近访问时间

os.getcwd() # 显示当前路径

os.listdir() # ls -a

os.listdir('/tmp') # ls -a /tmp

os.mkdir('/tmp/mydemo') # mkdir /tmp/mydemo

os.chdir('/tmp/mydemo') # cd /tmp/mydemo

os.mknod('test.txt') # touch test.txt

os.symlink('/etc/hosts', 'zhuji') # ln -s /etc/hosts zhuji

os.path.isfile('test.txt') # 判断test.txt是不是文件

os.path.islink('zhuji') # 判断zhuji是不是软链接

os.path.isdir('/etc')

os.path.exists('/tmp') # 判断是否存在

os.path.basename('/tmp/abc/aaa.txt') #取得文件名

os.path.dirname('/tmp/abc/aaa.txt') #取得路径

os.path.split('/tmp/abc/aaa.txt') #返回元组,路径+文件名

os.path.join('/home/tom', 'xyz.txt') ##返回string,完整路径

os.path.abspath('test.txt') # 返回当前目录test.txt的绝对路径

os.path.getsize(target) #獲取目標大小

os.walk(目录) 获取目录及子目录下的内容,返回一个列表,列表元素为元组.元组有三项,分别是’目录名’,[子目录],[文件]

os.fork() 创建子进程,返回值是个数字,对于父进程,返回值是PID,子进程是0

os.waitpid(-1,1) 父进程处理僵尸进程,返回一个元组(子进程pid,?).waitpid()接受两个参数,第一个参数设置为-1,表示与wait()函数相同;第二参数如果设置为0表示挂起父进程,直到子程序退出,设置为1表示不挂起父进程.waitpid()的返回值:如果子进程尚未结束则返回0,否则返回子进程的PID

## pickle模块简介(读写python对象)

• 把数据写入文件时,常规的文件方法只能把字符串对象写入。其他数据需先转换成字符串再写入文件 。

• python提供了一个标准的模块,称为pickle。使用它可以在一个文件中储存任何python对象,之后又可以把它完整无缺地取出来

import pickle

# shop\_list = ['eggs', 'apple', 'peach']

# with open('/tmp/shop.data','wb') as fobj :

# pickle.dump(shop\_list,fobj)

with open('/tmp/shop.data','rb') as fobj :

mylist = pickle.load(fobj)

print(mylist)

## pymysql模块(操作mysql数据)

pymysql模块不是系统自带的模块,需要使用pip3命令安装.具体见**安装python模块.** pymysql 实现了针对mysql数据库的增删改查操作. 建库建表还需要人手动在数据库操作.下面将以例子来解释pymysql的用法.

**规划数据库**

一、确定字段

姓名、性别、出生日期、部门、职位、联系方式、基本工资、奖金、工资总和、发工资日期

二、分析字段

1、根据第一范式，要把一些字段进一步拆分

所有的域都应该是原子性的，即数据库表的每一列都是不可分割的原子数据项，而不能是集。联系方式要拆分：电话号码，email

2、只有满足了1NF才能进行2NF

在1NF的基础上，非主属性必须完全依赖于候选键

员工表：员工标号、姓名、性别、出生日期、部门ID、电话号码、email

部门表：部门ID、部门名称

工资表：auto\_id员工编号、日期、基本工资、奖金、工资总和

3、3NF：任何非主属性不依赖与其他非主属性

工资总和是基本工资加奖金，所以不能出现在表中

三、创建数据库

1、建库

[root@room9pc01 ~]# mysql -uroot -p123456

mysql> create database tedu default charset 'utf8';

1. 创建部门表

mysql> create table departments(dep\_id int primary key,dep\_name varchar(20));

3、创建员工表

mysql> create table employees(emp\_id int primary key, emp\_name varchar(20) not null,birth\_date date,phone char(11),email varchar(50),dep\_id int,foreign key(dep\_id) references departments(dep\_id));

4、创建工资表

mysql> create table salary(auto\_id int auto\_increment primary key,date date,emp\_id int,basic int,awards int,foreign key(emp\_id) references employees(emp\_id));

**pymysql插入数据**

import pymysql

conn = pymysql.connect( #连接数据库

host='127.0.0.1',

port=3306,

user='root',

passwd='123456',

db='tedu',

charset='utf8'

)

cursor = conn.cursor() #创建游标.一条sql取出对应n条结果资源的接口/句柄,就是游标,沿着游标可以一次取出一行

insert1 = 'INSERT INTO departments(dep\_id,dep\_name) VALUES(%s, %s)'

result = cursor.execute(insert1,(1,'人事部')) #插入一行

conn.commit() #对数据库表做修改操作,必须要commit

data= [(2,'运维部'),(3,'开发部')]

cursor.executemany(insert1,data) #插入多行

conn.commit()

cursor.close()

conn.close()

**查询数据**

cursor = conn.cursor() #创建游标

select1 = 'select \* from departments' #sql查询语句

cursor.execute(select1)

result = cursor.fetchone() #查询一行，游标到这里

print(result) (1,’人事部’)

print('-' \* 30)

result = cursor.fetchmany(2) #查询2行，游标向下2行

print(result) #((2,’财务部’),(3,’运维部’))

print('-' \* 30)

result = cursor.fetchall() #这里还剩一个

print(result) #((4,’开发部’),) 如果没有就打印()

cursor.close()

conn.close()

**移动游标**

cursor = conn.cursor() #创建游标

select1 = 'select \* from departments'

cursor.execute(select1)

cursor.scroll(2) #默认相对当前位置移动

print(cursor.fetchall())

print('-' \* 30)

cursor.scroll(0,mode='absolute') #绝对移动，从0开始

print(cursor.fetchall())

cursor.close()

conn.close()

((3, '开发部'),)

------------------------------

((1, '人事部'), (2, '运维部'), (3, '开发部'))

**修改数据**

cursor = conn.cursor()

update1 = 'update departments set dep\_name =%s where dep\_name=%s '

result = cursor.execute(update1,('人力资源部','人事部'))

print(result)

conn.commit()

cursor.close()

conn.close()

**删除记录**

• 通过delete删除记录

cursor = conn.cursor()

delete1 = 'delete from departments where dep\_id =%s'

cursor.execute(delete1,3)

conn.commit()

cursor.close()

conn.close()

## SQLAlchemy模块(支持各种数据库)

SQLAlchemy安装

• SQLAlchemy由官方收录,可以直接安装

[root@localhost packages]# pip3 install sqlalchemy

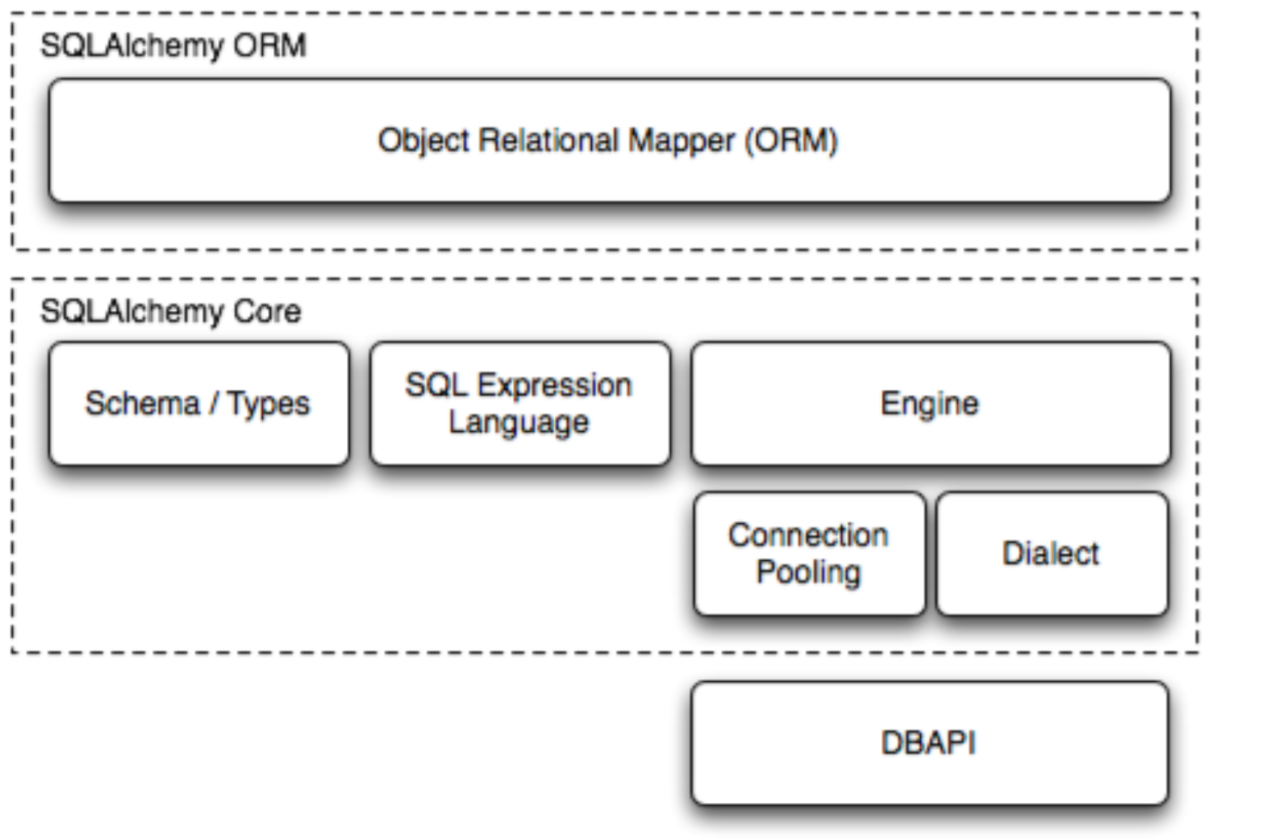
SQLAlchemy简介

• SQLAlchemy是Python编程语下的一款开源软件。提供SQL工具包及对象关系映射(ORM) 工具,使用MIT许可证发

• SQLAlchemy“采用简单的Python语言,为高效和高性能的数据库访问设计,实现了完整的企业级持久模型”

• SQLAlchemy的理念是,SQL数据库的量级和性能重要于对象集合;而对象集合的抽象又重要于表和行

• 目标是提供能兼容众多数据库(如SQLite、MySQL、Postgresql、Oracle、MS-SQL、SQLServer和Firebird)的企业级持久性模型



ORM模型

• ORM即对象关系映射

• 数据库表是一个二维表,包含多行多列。把一个表的内容用Python的数据结构表示出来的话,

可以用一个list表示多行,list的每一个元素是tuple,表示一行记录

[

('1', 'Michael'),

('2', 'Bob'),

('3', 'Adam')

]

• 用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示,就可以更容易地看出表的结构来

class User(object):

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

self.id = id

self.name = name

[

User('1', 'Michael'),

User('2', 'Bob'),

User('3', 'Adam')

]

需要手动建库,其他可以通过sqlAlchemy实现.数据库的设计方案见pymysql模块

### 建立连接(deconn.py)

from sqlalchemy import create\_engine,Column,String,Integer,ForeignKey,Date  
from sqlalchemy.ext.declarative import declarative\_base  
from sqlalchemy.orm import sessionmaker  
#通过create\_engine实现数据库的连接  
engine=create\_engine(  
 'mysql+pymysql://root:123456@localhost/tarena?charset=utf8',

//用户名:密码@主机/库. utf8加了才能支持中文.   
 encoding='utf8'  
 # echo=True

//echo=True表示将日志输出到终端屏幕,默认为False //生产环境不用加  
)

声明映射

• 当使用ORM的时候,配置过程从描述数据库表开始

• 通过自定义类映射相应的表

• 通过声明系统实现类映射

• 首先通过声明系统,定义基类  
Base=declarative\_base() #生成ORM映射所需的基类

创建会话类

• ORM访问数据库的句柄被称作Session  
Session=sessionmaker(bind=engine)

>>> Session = sessionmaker()

>>> Session.configure(bind=engine) //创建engine后执行

创建映射类

class Departments(Base): //必须继承与base  
 \_\_tablename\_\_='departments' //库的表名

#每个属性都是表中的一个字段,都是类属性  
 dep\_id=Column(Integer,primary\_key=True)  
 dep\_name=Column(String(20),nullable=False,unique=True)  
 def \_\_str\_\_(self):  
 return '[部门ID:%s, 部门名称:%s]'%(self.dep\_id,self.dep\_name)  
  
class Employees(Base):  
 \_\_tablename\_\_='employees'  
 emp\_id=Column(Integer,primary\_key=True)  
 name=Column(String(20),nullable=False)  
 gender=Column(String(6))  
 phone=Column(String(11))  
 email=Column(String(50))  
 dep\_id = Column(Integer,ForeignKey('departments.dep\_id'))  
  
 def \_\_str\_\_(self):  
 return '员工:%s'%self.name  
  
class Salary(Base):  
 \_\_tablename\_\_='salary'  
 auto\_id=Column(Integer,primary\_key=True)  
 date=Column(Date)  
 emp\_id = Column(Integer,ForeignKey('employees.emp\_id'))  
 basic=Column(Integer)  
 awards=Column(Integer)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 Base.metadata.create\_all(engine) 通过表的映射类,在数据库中创建表

创建映射类实例(增加dep表的记录add\_dep.py)

from deconn import Departments,Session  
  
hr=Departments(dep\_id=1,dep\_name='hr')  
ops=Departments(dep\_id=2,dep\_name='operations')  
dev=Departments(dep\_id=3,dep\_name='development')  
finance = Departments(dep\_id=4,dep\_name='财务部')  
  
deps=[ops,dev]  
  
session=Session()  
session.add(hr) 插入一个记录  
session.add\_all(deps) 插入多个记录  
session.add(finance)插入中文(有三处需要制定支持中文.1. 创建库,2 创建engine的时候两处: engine = create\_engine(

'mysql+pymysql://root:tedu.cn@localhost/tarena?charset=utf8',

//用户名:密码@主机/库. utf8加了才能支持中文

encoding='utf8',

echo=True)

)

session.commit()  
session.close()

向员工表中添加记录

from dbconn import Employees, Session

lll = Employees(emp\_id=1,name ='刘亮亮', gender ='男',phone ='15600001234',email ='lll@163.com',dep\_id =3)

sf = Employees(emp\_id=2,name ='施芳', gender ='男',phone ='15611001234',email ='sf@163.com',dep\_id =1)

dzj = Employees(emp\_id=3,name ='董枝俊', gender ='男',phone ='15611013456',email ='dzj@163.com',dep\_id =2)

wxy = Employees(emp\_id=4,name ='王秀燕', gender ='男',phone ='15611015555',email ='wxy@163.com',dep\_id =4)

dyx = Employees(emp\_id=5,name ='董优秀', gender ='男',phone ='13322341234',email ='dyx@163.com',dep\_id =1)

hsm = Employees(emp\_id=6,name ='黄少明', gender ='男',phone ='13300009999',email ='hsm@163.com',dep\_id =3)

glp = Employees(emp\_id=7,name ='龚丽萍', gender ='女',phone ='13322348888',email ='glp@163.com',dep\_id =2)

pj = Employees(emp\_id=8,name ='彭健', gender ='男',phone ='13322346666',email ='pj@163.com',dep\_id =4)

jjr = Employees(emp\_id=9,name ='金建荣', gender ='男',phone ='13322344444',email ='jjr@163.com',dep\_id =1)

emps= [lll,sf,dzj,wxy,dyx,hsm,glp,pj,jjr]

session = Session()

session.add\_all(emps)

session.commit()

session.close()

### 查询操作

基本查询

• 通过作用于session的query()函数创建查询对象

• query()函数可以接收多种参数

from dbconn import Departments, Employees, Salary, Session

**session = Session()**

qset = session.query(Departments).order\_by(Departments.dep\_id)

#order\_by(Departments.dep\_id)函数可以实现按指定字段排序

print(qset) # qset此时只是一条SQL语句

for dep in qset: # 向qset取值时，得到一个个实例

print("%s: %s" % (dep.dep\_id, dep.dep\_name))

####################################

qset = session.query(Employees.name, Employees.phone)

print(qset)

for name, phone in qset: # qset执行后返回的是元组.查询的是字段,就返回字段组成的元组

print('%s: %s' % (name, phone))

#######################################

qset = session.query(Departments.dep\_name.label('部门'))显示的字段名可以通过label()函数进行修改

print(qset)

for row in qset:

print(row.部门)

提取部分数据

• 通过“切片”的方式,实现部分数据的提取

qset = session.query(Employees.name, Employees.email).\

order\_by(Employees.emp\_id)[3:6]

print(qset) # qset因为切片的原因，已经是元组组成的列表了，不再是SQL语句

结果过滤

• 通过**filter()**函数实现结果过滤(相当于where子句)

qset = session.query(Employees.name).\

filter(Employees.dep\_id==2).filter(Employees.gender=='女')

print(qset)

for name in qset:

print(name)

常用过滤操作符

• 相等

query.filter(Employees.name=='john')

• 不相等

query.filter(Employees.name!='john')

• 模糊查询

query.filter(Employees.name.like(' %j ')) %匹配任意个字符,\_匹配一个字符

qset = session.query(Employees.name).\

filter(Employees.name.like('王%'))

print(qset)

for name in qset:

print(name)

#######################################

• in

query.filter(new\_emp.name.in\_(['bob', 'john'])

• not in

query.filter(~new\_emp.name.in\_(['bob', 'john'])

• 字段为空

query.filter(new\_emp.name.is\_(None))

• 字段不为空

query.filter(new\_emp.name.isnot(None))

qset = session.query(Employees.name).\

filter(Employees.name.in\_(['吴伟超', '李通达', '董枝俊']))

print(qset)

for name in qset:

print(name)

#######################################

qset = session.query(Employees.name).\

filter(~Employees.name.in\_(['吴伟超', '李通达', '董枝俊']))

print(qset)

for name in qset:

print(name)

#######################################

qset = session.query(Employees.name).\

filter(Employees.name.isnot(None))

print(qset)

for name in qset:

print(name)

#######################################

多重对象 and,or

qset = session.query(Employees.name).\

filter(and\_(Employees.dep\_id==2,Employees.gender=='女'))

print(qset)

for name in qset:

print(name)

##################################

qset = session.query(Employees.name).\

filter(or\_(Employees.dep\_id==2, Employees.gender=='女'))

print(qset)

for name in qset:

print(name)

################################

多表查询

• 通过join()方法实现多表查询

qset = session.query(Employees.name,Departments.dep\_name).\

join(Departments,Employees.dep\_id == Departments.dep\_id)

#前面查询的两项和后面join的表顺序要一致:

qset = session.query(Departments.dep\_name,Employees.name).\

join(Employees,Employees.dep\_id == Departments.dep\_id)

print(qset.all())

##################################

查询对象返回值

• all()返回列表

• first()返回结果中的第一条记录

• one()取出所有记录,如果不是一条记录则抛出异常

• scalar()调用one(),返回第一列的值

qset = session.query(Employees.name, Employees.phone)

print(qset.all()) # 返回列表

print(qset.first()) # 返回满足条件的第一个值

##################################

qset = session.query(Employees.name, Employees.phone).\

filter(Employees.emp\_id==1)

print(qset.one()) # 查询必须只有一项，否则报错

print(qset.scalar()) # 调用one，返回第一列的值

##################################

qset = session.query(Employees)

print(qset.count())

##################################

qset = session.query(Employees.name, Departments.dep\_name).\

join(Departments, Employees.dep\_id==Departments.dep\_id)

print(qset.all())

聚合

• 通过count()方法,统计行数

qset = session.query(Employees)

print(qset.count())

################################

**session.close()**

### 更新数据

• 通过会话的update()方法更新

• 通过会话的字段赋值更新

from dbconn import Departments,Session

session = Session()

ops = session.query(Departments).filter(Departments.dep\_id ==2)

ops.update({Departments.dep\_name:'运维部'})

dev = session.query(Departments).get(3) #返回主键是3的条目

geth函数总是查询主键

dev.dep\_name = '开发部'

session.commit()

session.close()

### 删除记录

• 通过会话的delete()方法进行记录删除

from dbconn import Departments, Session

session = Session()

# xz = Departments(dep\_id=5, dep\_name='行政部')

# session.add(xz)

xz = session.query(Departments).get(5)

session.delete(xz)

session.commit()

session.close()

## urllib基础 (获取网页内容)

• 在Python2版本中,有urllib和urlib2两个库可以用来实现request的发送。

而在Python3中,已经不存在urllib2这个库了,统一为urllib

• urllib中包括了四个模块

– urllib.request可以用来发送request和获取request的结果

– urllib.error包含了urllib.request产生的异常

– urllib.parse用来解析和处理URL

– urllib.robotparse用来解析页面的robots.txt文件

### 案例:爬取网页

1. 爬取的网页为http://www.tedu.cn

2. 保存的文件名为/tmp/tedu.html

get\_web.py

from urllib.request import urlopen

import sys

def get\_web(url, fname):

with urlopen(url) as html: #打开并爬取一个网页

with open(fname,'wb') as fobj:

while True :

data = html.read(1024\*4) #read()会把读取到的内容赋给一个字符串变量. readlines()读取文件的全部内容,readlines()会把读取到的内容赋值给一个列表变量。 readline()读取文件的一行内容。

if not data:

break

fobj.write(data)

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

get\_web(sys.argv[1],sys.argv[2])

### 案例3:爬取图片

1. 将http://www.tedu.cn所有的图片下载到本地

2. 本地的目录为/tmp/images

3. 图片名与网站上图片名保持一致

get\_url.py 从爬取的网页文件中找到图片地址,并保存到列表

import sys

import re

def get\_url(patt,fname):

cpatt = re.compile(patt)

result=[]

with open(fname) as fobj:

for line in fobj:

m = cpatt.search(line)

if m :

result.append(m.group())

return result

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

url = r'http://[-.\w/]+\.(jpg|png|jpeg|gif)'

get\_url(url,sys.argv[1])

download\_img.py 从图片地址的列表中下载图片

from get\_web import get\_web

from get\_url import get\_url

import os

get\_web('http://www.netbian.com/', '/tmp/tedu.html')

img\_url = r'http://[.\w/-]+\.(jpg|png|jpeg|gif)'

urls = get\_url(img\_url, '/tmp/tedu.html')

img\_dir = '/tmp/images'

if not os.path.exists(img\_dir):

os.mkdir(img\_dir)

for url in urls:

fname = os.path.join(img\_dir, url.split('/')[-1])

try:

get\_web(url, fname)

except:

pass

### 模拟客户端

• 有些网页为了防止别人恶意采集其信息所以进行了一些反爬虫的设置,而我们又想进行爬取

• 可以设置一些Headers信息(User-Agent),模拟成浏览器去访问这些网站

import sys

from urllib.request import Request

from urllib.request import urlopen

def get\_web(url, fname):

header = {

'User-Agent':"Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36"

} #模拟消息头,可以从网站访问日志里面得到

html = Request(url,headers=header)

with urlopen(html) as html:

with open(fname,'wb') as fobj:

while True :

data = html.read(1024\*4)

if not data:

break

fobj.write(data)

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

get\_web(sys.argv[1],sys.argv[2])

### urllib进阶

• 一般来说,URL标准中只会允许一部分ASCII字符,比如数字、字母、部分符号等

• 而其他的一些字符,比如汉字等,>是不符合URL标准的。此时,我们需要编码。

• 如果要进行编码,可以使用urllib.request.quote()进行

from urllib.request import quote

import webbrowser

words = quote('hello world!')

search = 'https://www.baidu.com/s?wd=' + words

print(search)

webbrowser.open\_new\_tab(search)

### HTTP异常处理

urllib.error.HTTPError

urllib.error.URLError

• 如果访问的页面不存在或拒绝访问,程序将抛出异常

• 捕获异常需要导入urllib.error模块

>>> html = urllib.request.urlopen('hep://172.40.50.116/a.html')

urllib.error.HTTPError: HTTP Error 404: Not Found

>>> html = urllib.request.urlopen('hep://172.40.50.116/aaa')

urllib.error.HTTPError: HTTP Error 403: Forbidden

**案例4**:处理下载错误

1. 起动一个web服务

2. 在web服务器的文档目录下创建目录ban,权限设置为700

3. 编写python程序访问不存在的路径和ban目录,处理404和403错误

4. 404错误打印“无此页面”,403错误打印“无权访问”

import sys

from urllib.request import urlopen

from urllib.error import HTTPError

def get\_web(url, fname):

try:

html= urlopen(url)

except HTTPError as e: #把异常的说明字符串保存到变量e中

print(e)

if e.code == 403 :

print('权限不足')

elif e.code == 404 :

print('没有那个地址')

return

with open(fname,'wb') as fobj:

while True :

data = html.read(1024\*4)

if not data:

break

fobj.write(data)

html.close()

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

get\_web(sys.argv[1],sys.argv[2])

## request模块 (urllib 高级版,获取网页内容)

**requests基础**

• Requests是用Python语言编写的、优雅而简单的HTTP库

• Requests内部采用来urillib3

• Requests使用起来肯定会比urillib3更简单便捷

• Requests需要单独安装

**requests特性**

• 支持keep-alive的连接池

• 支持通用的域名以及URL地址

• 支持使用cookie

• 支持使用类似浏览器的SSL验证

• 文件上传、下载

### request对象

**GET和POST**

• 通过requests发送一个GET请求,需要在URL里请求的参数可通过params传递

r = requests.get(url="", params={}, headers={}, cookies={})

• 与GET不同的是,POST请求新增了一个可选参数data,需要通过POST请求传递的body里的数据可以通过data传递

r = requests.post(url="", data ={}, params={}, file={}, headers={}, cookies={})

• 其他 HTTP 请求类型:PUT,DELETE,HEAD 以及OPTIONS使用起来一样简单

r = requests.put('http://httpbin.org/put', data = {'key':'value'})

r = requests.delete('http://httpbin.org/delete')

r = requests.head('http://httpbin.org/get')

r = requests.options('http://httpbin.org/get')

import requests

payload = {'wd': 'hello world!'}

# http://www.baidu.com/s?wd=hello%20world%21

r = requests.get('http://www.baidu.com/s', params=payload)

data = r.content

with open('/tmp/bbb.html', 'wb') as fobj:

fobj.write(data)

**params 请求参数**

• 当访问一个URL时,我们经常需要发送一些查询的字段作为过滤信息,例如:httpbin.com/get?key=val,这里的key=val就是限定返回条件的参数键值对

• 当利用python的requests去发送一个需要包含这些参数键值对时,可以将它们传给params

payload = {'key1':'value1', 'key2':'value2'}

r = requests.get('http://httpbin.org/get, params = payload)

**headers 设定头部**

• 用户也可以自己设定请求头

url = 'https://api.github.org/some/endpoint'

headers = {'Accept': 'applicaPon/json'}

r = requests.get(url, headers = headers)

**data发送请求数据**

• 有时候,用户需要将一些数据放在请求的body内;这时候,就需要对data传参了(仅POST, DELETE,PUT等方法有该参数,GET没有,因为GET请求没有body)

payload = {'key1' : 'value1', 'key2' : 'value2'}

r = requests.post('http://hapbin.org/post', data = payload)

**files传递文件**

• 可以通过requests传一些文件,使用的是file参数

url = 'http://hapbin.org/post'

files = {'file' : open('report.xls', 'rb')}

r = requests.post(url, files = files)

### **[Response](http://docs.python-requests.org/zh_CN/latest/api.html" \l "requests.Response" \o "requests.Response) 对象**

**响应内容**

1. r.text

• 读取服务器响应的内容.

>>> r = requests.get('http://www.baidu.com')

>>> r.text

• 请求发出后,Requests 会基于 HTTP 头部对响应的编码作出有根据的推测

• 可以找出 Requests 使用了什么编码,并且能够使用r.encoding 属性来改变它

>>> r.encoding

'ISO-8859-1'

>>> r.encoding='utf8'

>>> r.text

2. r.content

• 也可以用字节的方式访问请求响应体,尤其是非文本请求(如图片)

3. r.json()

• Requests 中还有一个内置的 JSON 解码器,助你处理 JSON 数据

**获取来原始套接字响应, r.raw**

• 在罕见的情况下,你可能想获取来自服务器的原始套接字响应,那么你可以访问 r.raw

• 如果你确实想这么干,那请你确保在初始请求中设置了 stream=True

>>> r = requests.get('haps://api.github.com/events', stream=True)

>>> r.raw

<requests.packages.urllib3.response.HTTPResponse object at

0x101194810>

>>> r.raw.read(10)

**响应状态码 r.status\_code**

• 检测响应状态码

>>> r.status\_code

• 为方便引用,Requests还附带了一个内置的状态码查询对象

>>> r.status\_code == requests.codes.ok

• 如果发送了一个错误请求(一个 4XX 客户端错误,或者 5XX 服务器错误响应),可以通过Response.raise\_for\_status() 来抛出异常

>>> bad\_r = requests.get('http://hapbin.org/status/404')

>>> bad\_r.status\_code

404

>>> bad\_r.raise\_for\_status()

**响应头 r.headers**

• 可以查看以字典形式展示的服务器响应头

>>> r.headers

{

'content-encoding': 'gzip',

'transfer-encoding': 'chunked',

'connecPon': 'close',

'server': 'nginx/1.0.4',

'x-runPme': '148ms',

'etag': '"e1ca502697e5c9317743dc078f67693f"',

'content-type': 'applicaPon/json'

}

**使用Cookie r.cookies**

• 版本0定义了Set-Cookie响应首部、Cookie请求首部

• Set-Cookie响应首部,其实就是服务端返回的Cookie信息,具体的语法如下:

>>> r = requests.get('http://www.baidu.com')

>>> r.cookies

>>> r1 = requests.get('http://www.baidu.com', cookies=r.cookies)

**案例4**:使用requests获取天气

1. 运行程序时,屏幕将出现你所在城市各区县名字

2. 用户指定查询某区县,屏幕上将出现该区县当前的气温、湿度、风向、风速等

from urllib.request import urlopen

import json

import requests

def get\_weather(city\_code):

url = 'http://www.weather.com.cn/data/sk/%s.html' % city\_code

r = requests.get(url)

r.encoding = 'utf8'

data = r.json()

output = '风向：%s, 风力: %s， 温度：%s, 温度：%s' % (

data['weatherinfo']['WD'],

data['weatherinfo']['WS'],

data['weatherinfo']['temp'],

data['weatherinfo']['SD']

)

return output

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

city\_codes = { '0': '101010100', '1': '101121404'}

prompt = """(0) 北京

(1) 台儿庄

请选择(0/1): """

choice = input(prompt)

print(get\_weather(city\_codes[choice]))

## paramiko模块(并发远程执行命令)

安装paramiko模块

• 本地安装

# yum install -y gcc gcc-c++ python-devel

# tar xzf paramiko-1.15.4.tar.gz

# python setup.py install

• 网络安装

# pip3 install paramiko

**实例**

• 编写用于实现ssh访问的脚本

– 创建SSHClient实例

– 设置添加主机密钥策略

– 连接ssh服务器

– 执行指定命令

– 在shell命令行中接受用于连接远程服务器的密码以及在远程主机上执行的命令

import paramiko

ssh = paramiko.SSHClient() #创建用于连接ssh服务器的实例

ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy()) #设置自动添加主机密钥,相当于回答yes

ssh.connect(

hostname='127.0.0.1',

username='root',

port=22

password='Taren1',

) #连接ssh服务器

a=ssh.exec\_command('ls /home') #在ssh服务器上执行指定命令

#a 是由类文件对象组成的列表，共三项，分别是输入、输出、错误

print(len(a))

out = a[1].read()

error = a[2].read()

print(out.decode('utf8'))

print(error.decode('utf8'))

ssh.close()

**案例**5:利用多线程实现ssh并发访问

• 编写脚本程序

1. 在文件中取出所有远程主机IP地址

2. 在shell命令行中接受远程服务器IP地址文件、远程服务器密码以及在远程主机上执行的命令

3. 通过多线程实现在所有的远程服务器上并发执行命令

import os  
import threading  
  
def remote\_exec(hostname,username,password,cmd):  
 ssh=paramiko.SSHClient()  
 ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())  
 ssh.connect(hostname=hostname,username=username,password=password)  
  
 a=ssh.exec\_command(cmd)  
 # print(len(a))  
 out = a[1].read()  
 error=a[2].read()  
 if out:  
 print('%s result :'%hostname+out.decode('utf8'))  
 else:  
 print('%s error:'%hostname+error.decode('utf8'))  
 ssh.close()  
  
def remote\_exec\_all(fname,password,cmd):  
 if not os.path.exists(fname):  
 return  
 with open(fname) as fobj:  
 ts=[]  
 for line in fobj:  
 line=line.strip('\r\n')  
 t=threading.Thread(target=remote\_exec,args=(line,'root',password,cmd))  
 t.start()  
 ts.append(t)  
 for t in ts:  
 t.join()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 fname='/tmp/hosts'  
 password='123456'  
 cmd='ls /home'  
 # password=getpass.getpass()  
 remote\_exec\_all(fname,password,cmd)

[python 使用paramiko模块上传本地文件到ssh](https://www.cnblogs.com/haq5201314/p/7632561.html)

我们要了解几个函数：

paramiko.Tranport(("目标ip，端口"))#这是上传目标的IP和端口

paramiko.SFTPClient.from\_tranport()#安全文件传输凭证

sftp.put(loaclfile,remotefile)#loaclfile是要上传的文件,remotefile是上传后要保存的文件名

import paramiko

ssh=paramiko.Transport(("192.168.223.129",22))

ssh.connect(username="root",password="root")

sftp=paramiko.SFTPClient.from\_transport(ssh)

localfile="使用说明.txt"

remotefile="使用说明.txt"try:

sftp.put(localfile,remotefile)except Exception:

print("[-]put Error:User name or password error or uploaded file does not exist")

print("上传")

ssh.close()

## smtplib模块(邮件编程)

**SMTP概述**

• SMTP(Simple Mail Transfer Protocol)即简单邮件传输协议,使用TCP协议25端口

• 它是一组用于由源地址到目的地址传送邮件的规则,由它来控制信件的中转方式

• python的smtplib提供了一种很方便的途径发送电子邮件。它对smtp协议进行了简单的封装

**SMTP对象**

• Python发送邮件,第一步是创建SMTP对象

import smtplib

smtp\_obj = smtplib.SMTP( [host [, port [, local\_hostname]]] )

• 创建SMTP对象也可以不给定参数,之后再通过对象的其他方法进行绑定

设置邮件

• 标准邮件需要三个头部信息

– From:发件人

– To:收件人

– Subject:主题

from email.mime.text import MIMEText #MIMEText:多用途internet扩展

from email.header import Header

message = MIMEText('Python 邮件发送测试...', 'plain', 'utf-8') 邮件正文

message['From'] = Header("zzg", 'utf-8') # 发送者

message['To'] = Header("root", 'utf-8') # 接收者

subject = 'Python SMTP 邮件测试' #标题

message['Subject'] = Header(subject, 'utf-8')

sendmail方法

• Python SMTP 对象使用 sendmail 方法发送邮件

SMTP.sendmail(from\_addr, to\_addrs, msg[, mail\_opPons, rcpt\_opPons])

• sendmail方法三个必须的参数有:

– 收件人

– 发件人

– 消息主体msg是一个字符串,表示邮件

• 将准备好的邮件发送

**案例1:通过本机发送邮件**

1. 创建bob和alice帐户

2. 编写发送邮件件程序,发件人为root,收件人是本机的bob和alice帐户

import smtplib

from email.mime.text import MIMEText

from email.header import Header

message = MIMEText('Python邮件发送测试\n','plain','utf8') #utf-8 utf8都行,mysql里面只允许utf8

message['From'] = Header('root@localhost','utf8')

message['To'] = Header('bob@localhost','utf8')

message['Subject'] = Header('mail test','utf8')

sender = 'root@redhat.com'

receivers = ['bob@localhost','alice@localhost']

smtp\_obj = smtplib.SMTP('localhost') #25端口监听在127.0.0.1上,所以只能写localhost

smtp\_obj.sendmail(sender,receivers,message.as\_string())

SMTP认证

• 如果本机没有SMTP功能,也可以使用第三方的邮件服务器

• 第三方邮件服务器往往需要认证

**通过互联网服务器发送邮件**

1. 通过自己互联网注册的邮箱,为其他同学互联网邮箱发邮件

import smtplib

from getpass import getpass

from email.mime.text import MIMEText

from email.header import Header

mail\_host = 'smtp.126.com'

mail\_user = 'gaopengfei12@126.com'

mail\_pwd = getpass()

message = MIMEText('Python邮件发送测试\n','plain','utf8')

message['From'] = Header('root@localhost','utf8') 这里是显示的发件人

message['To'] = Header('haha@localhost','utf8') 这是是显示的收件人

message['Subject'] = Header('python mail test','utf8')

sender = 'gaopengfei12@126.com' 这是是代发人邮箱

receivers = ['gaopengfei12@126.com'] 这是代收人邮箱

smtp\_obj = smtplib.SMTP()

smtp\_obj.connect(mail\_host,25) #25为smtp端口号，可以不写

以上两句可以合并成smtp\_obj=smtplib.SMTP(mail\_host,25)

smtp\_obj.login(mail\_user,mail\_pwd)

smtp\_obj.sendmail(sender,receivers,message.as\_string())

## JSON 模块

JSON概述

• JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式

• 易于人阅读和编写,同时也易于机器解析和生成

• 基于JavaScript Programming Language

• JSON采用完全独立于语言的文本格式,但是也使用了类似于C语言家族的习惯(包括C, C++, C#, Java,JavaScript, Perl, Python等)

• 这些特性使JSON成为理想的数据交换语言

JSON结构

• JSON主要有两种结构

– “键/值”对的集合:python中主要对应成字典

– 值的有序列表:在大部分语言中,它被理解为数组

数据类型对应表

**Python JSON**

dict object

list,tuple array

str string

int,float number

True true

False false

None null

json.dumps()方法: 对简单数据类型进行编码,得到json对象

json.loads()方法:对编码后的json对象进行decode解码,得到原始数据(python对象)

注意区别: dumps, loads 得到转换后的对象.

dump和load也是类似的功能，只是与文件操作结合起来了.把转换后的对象写入文件.这个比较少用.

import json

number = json.dumps(100)

print(number)

json.loads(number)

100

**案例:天气预报查询**

• 搜索“中国天气网 城市代码查询”,查找城市代码

• 城市天气情况接口

– 实况天气获取:http://www.weather.com.cn/data/sk/城市代码.html

– 城市信息获取:http://www.weather.com.cn/data/cityinfo/城市代码.html

– 详细指数获取:http://www.weather.com.cn/data/zs/城市代码.html

图片位置:

http://m.weather.com.cn/img/c0.gif

http://m.weather.com.cn/img/b0.gif

http://www.weather.com.cn/m/i/weatherpic/29x20/d0.gif

http://www.weather.com.cn/m2/i/icon\_weather/29x20/n00.gif

#城市代码

http://www.cnblogs.com/wf225/p/4090737.html

案例3:天气预报查询

1. 运行程序时,屏幕将出现你所在城市各区县名字

2. 用户指定查询某区县,屏幕上将出现该区县当前的气温、湿度、风向、风速等

from urllib.request import urlopen

import json

def get\_weather(city\_code):

url = 'http://www.weather.com.cn/data/sk/%s.html' % city\_code

html = urlopen(url)

data = json.loads(html.read())

output = '风向：%s, 风力: %s， 温度：%s, 温度：%s' % (

data['weatherinfo']['WD'],

data['weatherinfo']['WS'],

data['weatherinfo']['temp'],

data['weatherinfo']['SD']

)

html.close()

return output

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

city\_codes = { '0': '101010100', '1': '101121404'}

prompt = """(0) 北京

(1) 台儿庄

请选择(0/1): """

choice = input(prompt)

print(get\_weather(city\_codes[choice]))

## zabbix编程

Zabbix api简介

部署zabbix

• Zabbix是一个基于WEB界面的提供分布式系统监视以及网络监视功能的企业级的开源解决方案

• 能监视各种网络参数,保证服务器系统的安全运营;并提供灵活的通知机制以让系统管理员快速定位/解决存在的各种问题

• 部署方式参见云计算监控课程,不再赘述

Zabbix api概述

• Zabbix API允许你以编程方式检索和修改Zabbix的配置,并提供对历史数据的访问。它广泛用于:

– 创建新的应用程序以使用Zabbix

– 将Zabbix与第三方软件集成

– 自动执行常规任务

JSON-RPC

• Zabbix API是基于Web的API,作为Web前端的一部分提供。

它使用JSON-RPC 2.0协议,这意味着两件事:

– 该API包含一组独立的方法

– 客户端和API之间的请求和响应使用JSON格式进行编码

API结构

• Zabbix API包含许多方法,这些方法都名义上分组为单组的API

• 每个方法执行一个特定任务。例如,方法host.create 隶属于 host 这个API ,用于创建新主机

• 历史上,API有时被称为“类”

• 大多数API至少包含四种方法: get, create,update 和 delete ,分别是检索,创建,更新和删除数据。但是某些API提供一套完全不同的一组方法。

执行请求

• 设置前端后,你就可以使用远程HTTP请求来调用API。为此,需要向 api\_jsonrpc.php 位于前端目录中的文件发送HTTP POST请求。例如,如果你的Zabbix前端安装在 http://company.com/zabbix , 那么用HTTP请求来调用 apiinfo.version 方法就如下面这样:

POST http://company.com/zabbix/api\_jsonrpc.php HTTP/1.1

Content-Type: applicaPon/json-rpc

{"jsonrpc":"2.0","method":"apiinfo.version","id":1,"auth":null,"params":

{}}

• 请求的 Content-Type 头部必须设置为以下值之一:

– application/json-rpc

– application/json

– application/jsonrequest

使用API

• 通过zabbix提供的API接口,就可以使用python与其交互了

zabbix\_apiversion.py 查询api版本信息

import requests

import json

url = 'http://192.168.1.2/api\_jsonrpc.php'

headers = {'Content-Type': 'application/json-rpc'}

# https://www.zabbix.com/documentation/3.4/zh/manual/api

# data数据是从上面的官方文档网址查询

data = {

"jsonrpc":"2.0",

"method":"apiinfo.version",

"id":1,

"params":[]

}

r = requests.post(url,headers = headers, data=json.dumps(data))

print(r.json()) #返回的内容是json

工作流程

• 在访问大多数Zabbix中的任何数据之前,需要登录并获取身份验证令牌

• 取得令牌后,访问其他数据只要出示该令牌即可,不需要再进行身份验证

• 通过zabbix api提供的各种方法实现数据的检索、项目的创建等

**获取令牌**

• 使用 user.login 方法登录并获取身份验证令牌

• jsonrpc - API使用的JSON-RPC协议的版本; ZabbixAPI实现JSON-RPC版本2.0

• method - 调用的API方法

• params - 将被传递给API方法的参数

• id - 请求的任意标识符

• auth -用户认证令牌; 因为我们还没有一个,它的设置None

**案例6**:获取令牌

• 编写get\_token函数

• 该函数接受zabbix服务器url、用户名和密码作为参数

• 函数返回值为用户令牌token

查看官网地址: API->用户->user login

get\_token.py

import requests

import json

url = 'http://192.168.1.2/api\_jsonrpc.php'

headers = {'Content-Type': 'application/json-rpc'}

data = {

"jsonrpc": "2.0",

"method": "user.login",

"params": {

"user": "Admin",

"password": "zabbix"

},

"id":1,

}

r = requests.post(url,headers = headers, data=json.dumps(data))

print(r.json())

得到的身份验证令牌

{

"jsonrpc": "2.0", JSON-RPC协议的版本

"result": "0424bd59b807674191e7d77572075f33", 方法返回的数据

"id": 1 相应请求的标识符

}

**检索主机**

• 有一个有效的用户身份验证令牌,可以用来访问Zabbix中的数据

• 使用 **host.get** 方法检索所有已配置主机的ID,主机名和接口。auth 属性设置为获得的身份验证令牌

API->主机->host.get

get\_zabbix\_host.py

import requests

import json

url = 'http://192.168.1.2/api\_jsonrpc.php'

headers = {'Content-Type': 'application/json-rpc'}

data = {

"jsonrpc": "2.0",

"method": "host.get", 指定调用的函数

"params": { 这里传递函数的参数

"output": "extend",

"filter": { 指定条件

"host": [ 指定主机

"Zabbix server",

]

}

},

"auth": "e7bf6cdfd078e9cb65e5a13896f7fe11", #这个验证id是上一步的输出结果

"id":1,

}

r = requests.post(url,headers = headers, data=json.dumps(data))

print(r.json())

get\_host\_int.py 查看主机的接口和ip

import requests

import json

url = 'http://192.168.1.2/api\_jsonrpc.php'

headers = {'Content-Type': 'application/json-rpc'}

data = {

"jsonrpc": "2.0",

"method": "host.get",

"params": {

"output": [ "hostid", "host"],

"selectInterfaces": ["interfaceid", "ip" ]

},

"auth": "e7bf6cdfd078e9cb65e5a13896f7fe11",

"id":2,

}

r = requests.post(url,headers = headers, data=json.dumps(data))

print(r.json())

结果输出

{

'jsonrpc': '2.0',

'result': [

{

'hostid': '10084',

'host': 'Zabbix server',

'interfaces': [{'interfaceid': '1', 'ip': '127.0.0.1'}]

},

{

'hostid': '10254',

'host': 'zabbix\_client\_node1',

'interfaces': [{'interfaceid': '2', 'ip': '192.168.1.11'}]

}

],

'id': 2

}

获取主机组

• 获取主机组的方法与检索主机一样,只要修改请求数据即可

data = {

"jsonrpc": "2.0",

"method": "hostgroup.get",

"params": {

"output": "groupids",

"filter": {

"name": [

"Zabbix servers",

]

}

},

"auth": "e7bf6cdfd078e9cb65e5a13896f7fe11",

"id":2,

}

创建主机 在官方文档的host.create查看模板,传入的参数

• 创建主机操作与获取信息操作完全一样,只是传递的请求参数不一样而已

data = {

"jsonrpc": "2.0",

"method": "host.create",

"params": {

"host": "Linux server",

"interfaces": [

{

"type": 1,

"main": 1,

"useip": 1,

"ip": "192.168.1.3",

"dns": "",

"port": "10050"

}

],

"groups": [

{

"groupid": "4" 这里需要查看上面的主机组id

}

],

},

"auth": "e7bf6cdfd078e9cb65e5a13896f7fe11",

"id": 1

}

# 文件对象

**文件打开方法**

open及file内建函数

• 作为打开文件之门的“钥匙”,内建函数open()以及file()提供了初始化输入/输出(I/O)操作的通用接口

• 成功打开文件后时候会返回一个文件对象,否则引发一个错误

• open()方法和file()方法可以完全相互替换

• 基本语法:

file\_object = open(file\_name, access\_mode='r', buffering=-1)

文件对象访问模式

打开模式 操作

r 以读方式打开(文件不存在则报错)

w 以写方式打开(文件存在则清空,不存在则创建)

a 以追加模式打开(必要时创建新文件)

r+ 以读写模式打开(参见r)

w+ 以读写模式打开(参见w)

a+ 以读写模式打开(参见a)

b 以二进制模式打开

文件操作的三个步骤：打开、读写、关闭

## 文件读取

**read方法** 适合所有文件

• read()方法用来直接读取字节到字符串中,最多读取给定数目个字节，把所有内容读取出来

• 如果没有给定size参数(默认值为-1)或者size值为负,文件将被读取直至末尾

>>> data = fobj.read()

>>> print(data)

**readline方法** 适合文本文件

• 读取打开文件的一行(读取下个行结束符之前的所有字节)

• 然后整行,包括行结束符,作为字符串返回

• 它也有一个可选的size参数,默认为-1,代表读至行结束符

• 如果提供了该参数,那么在超过size个字节后会返回不完整的行

>>> data = fobj.readline()

>>> print(data)

**readlines方法**

• readlines()方法读取所有(剩余的)行然后把它们作为一个字符串**列表**返回

data = fobj.readlines()

print(data)

f = open('/tmp/passwd')

data = f.read()

print(data)

f.close()

#随着读写的进行，文件指针向后移动。因为第一个f.read()已经把文件指针移动到结尾了，所以再读就没有数据了所以再执行data = f.read()时，这里为空

硬盘的一个block 在分区后，默认是4k 也就是4096个字节.所以在读取的时候，建议使用

f.read(4096)

f = open('/tmp/passwd')

data = f.read(4) # 读4字节

f.readline() # 读到换行符\n结束

f.readlines() # 把每一行数据读出来放到列表中

f.close()

**文件迭代**

• 如果需要逐行处理文件,可以结合for循环迭代文件

• 迭代文件的方法与处理其他序列类型的数据类似

>>> fobj = open('star.py')

>>> for eachLine in fobj:

print(eachLine, end= '')

f = open('/tmp/passwd')

for line in f:

print(line,end='') #因为默认read的时候每行都有\n换行，所以这里把print默认的换行取消就可以了.不取消会有一行空行

f.close()

#######################################

f = open('图片地址', 'rb') # 打开非文本文件要加参数b

f.read(4096)

f.close()

## 文件输出

**write方法**

• write()内建方法功能与read()和readline()相反。它把含有文本数据或二进制数据块的字符串写入到文件中去

• 写入文件时,不会自动添加行结束标志,需要程序员手工输入

. 返回写入的字节数

>>> fobj.write('Hello World!\n')

13 //写入的字节

**writelines方法**

• 和readlines()一样,writelines()方法是针对列表的操作

• 它接受一个字符串列表作为参数,将它们写入文件

• 行结束符并不会被自动加入,所以如果需要的话,必须在调用writelines()前给每行结尾加上行结束符

>>> fobj.writelines(['Hello World!\n', 'python programing\n'])

**例子**

f = open('/tmp/myfile','w') #以w打开文件，如果文件不存在则创建

f.write('hello world\n')

f.flush() #立即将缓存中的数据同步到磁盘

f.writelines(['2nd line.\n','new line.\n'])

f.close() #关闭文件的时候，数据保存到磁盘

**with子句**

• with语句是用来简化代码的

• 在将打开文件的操作放在with语句中,代码块结束后,文件将自动关闭

>>> with open('foo.py') as f:

...... data = f.readlines()

...

>>>f.closed #True

## 文件指针移动

• seek(offset[, whence]):移动文件指针到不同的位

– offset是相对于某个位置的偏移量

– whence的值,0表示文件开头,1表示当前位置,2表示文件的结尾

需要以b方式打开文件,seek才能支持从0/1/2位置偏移. 当不是以b打开文件,则只支持从0偏移.

• tell():返回当前文件指针的位置

>>> f = open('/tmp/passwd')

>>> f.tell()

0

>>> f.read()

>>> f.tell()

2525

>>> f.seek(0,0)

0

>>> f.readline()

>>> f.tell()

142

>>> f.close

## 标准文件

• 程序一执行,就可以访问三个标准文件

– 标准输入:一般是键盘,使用sys.stdin

– 标准输出:一般是显示器缓冲输出,使用sys.stdout

– 标准错误:一般是显示器的非缓冲输出,使用sys.stderr

>>> import sys

>>> sys.stdout.write('hello world!\n')

hello world!

>>> hi = sys.stdin.readline()

hello

>>> hi

'hello\n'

# 不知道放哪里

## print用法

print('hello world!')

print('hello', 'world!') # 逗号自动添加空格

print('hello' + 'world!') # 加号表示字符拼接

print('hello', 'world', sep='\*\*\*') # 单词间用\*\*\*分隔

print('#' \* 50) # \*号表示重复50遍

print('how are you?', end='') # 默认print会打印回车，end=''表示不要回车

### input用法

number = input("请输入数字: ") # input用于获取键盘输入

print(number) # input获得的数据全是字符. 可以直接使用变量,不用加$

print(number + 10) # 报错，不能把字符和数字做运算

a=’10’ #a是字符 a+10 报错

python的变量需要先定义再使用

dir(mutable) //查看函数属性

id(vir) //查看变量vir地址

getpass.getpass('password: ') #输入的密码不显示

type(a) 可以查看a的类型

查看ascII的值 ord(‘a’)