### python3学习手册

**简介**：

Python官网： https://www.python.org

Python由Guido van Rossum于1989年底发明，于1991年发行第一版，Python源代码遵循GPL协议

Python是一种解释型、面向对象、动态数据类型、可交互的语言

python2.0于2000-10-16发布，于2020年1月1日停止更新2.x版本，Python-2.7成为最后一个py 2.x版本

python3.0于2008-12-03发布

**python各版本发布时间**：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2.0.1 | 2001-06-22 |  | 3.7.0 | 2018-06-27 |  | 3.14.0 | 2025-10-07 |
| 2.7.0 | 2010-07-03 |  | 3.8.0 | 2019-10-14 |  |  |  |
| 2.7.5 | 2013-05-15 |  | 3.9.0 | 2020-10-05 |  |  |  |
| 2.7.18 | 2020-04-20 |  | 3.9.16 | 2022-12-06 |  |  |  |
| 3.0.1 | 2009-02-13 |  | 3.10.0 | 2021-10-04 |  |  |  |
| 3.5.0 | 2015-09-13 |  | 3.11.0 | 2022-10-24 |  |  |  |
| 3.6.0 | 2016-12-23 |  | 3.12.0 | 2023-10-02 |  |  |  |
| 3.6.8 | 2018-12-24 |  | 3.13.0 | 2024-10-07 |  |  |  |

**版权声明**：

本文档以开源的形式发布，所有条款如下：

（1）无担保：作者不保证文档内容的准确无误，亦不承担由于使用此文档所导致的任何后果

（2）自由使用：任何人可以出于任何目的而自由地 阅读/链接/打印/转载/引用/分发/再创作 此文档，无需任何附加条件

若您 阅读/链接/打印/转载/引用/分发/再创作 本文档，则说明接受以上2个条款。

作者：李茂福

2022-03-26～2025-12-11

**★第0章、linux执行python脚本的方式**

**python安装路径**：

|  |  |  |
| --- | --- | --- |
| 操作系统 | python版本 | 安装路径 |
| windows | 2.7 | C:\Python27 |
| windows | 3.10 | C:\Users\用户名\AppData\Local\Programs\Python\Python310 |
| centos7 | 2.7.5 | /usr/bin/python --> /usr/bin/python2 --> /usr/bin/python2.7 (可执行文件) |
| centos8 | 3.6.8 | /usr/libexec/platform-python --> /usr/libexec/platform-python3.6 (可执行文件) |

①**进入交互模式**

# python3 #输入python3 回车

>>> #这里可执行python代码，类似shell脚本

>>> print("hello")

>>> exit() #退出交互模式

②**执行一次性代码**

# python3 -c "要执行的代码，多个语句之间用;分号隔开"

③**执行脚本**

# vi test.py #内容如下3行

#!/usr/bin/env python3

# coding=utf-8

print("hello world")

# chmod +x test.py #添加可执行权限

# python3脚本文件开头位置要指定以下2行（使用的python解释器及本源码文件编码）

#!/usr/bin/env python3

# coding=utf-8

# python2脚本文件里要指定以下2行

#!/usr/bin/env python

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

# ./test.py #执行脚本

#或者

# python3 test.py

**★第1章、PyCharm常用设置**

PyCharm官网： https://www.jetbrains.com/pycharm/download/

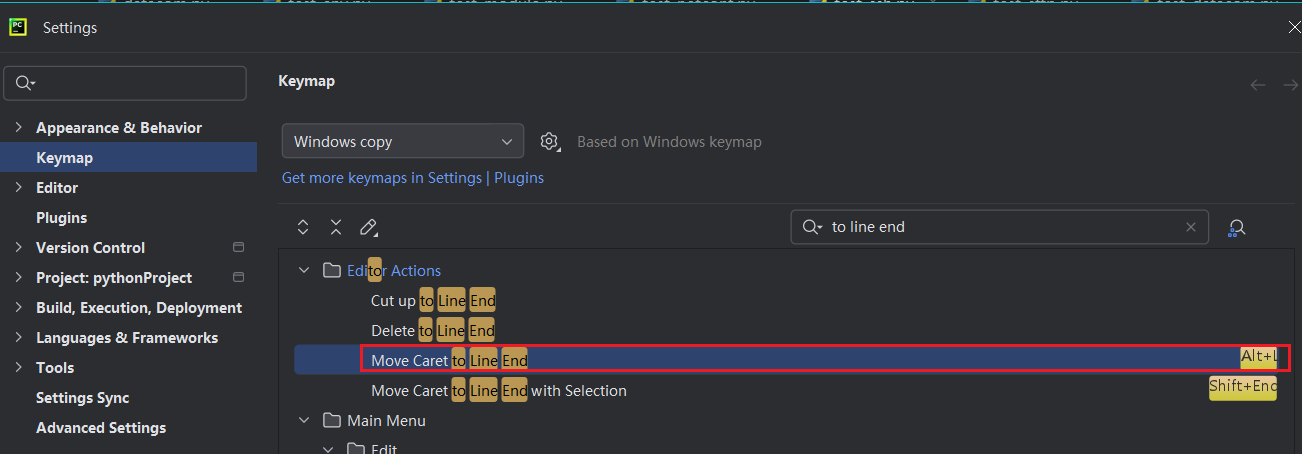
旧版本下载： https://www.jetbrains.com/pycharm/download/other.html

**★PyCharm快捷键**

|  |  |
| --- | --- |
| Alt + Shift | 列选择模式，可同时选择多列 |
| Ctrl + Alt + L | 重新格式化代码，Reformat Code |
| Ctrl + B | 跳转到鼠标指定的函数定义处 |
| Shift + F10 | 运行程序 |
| Ctrl + R | 查找替换 |
| Ctrl + / | 批量注释或取消注释（选中的所有行） |
| Shift + Enter | 直接创建新行，并跳到新行（会自动根据上下文进行缩进） |
| Ctrl + Shift + + | 展开当前代码文件的所有代码块 |
| Ctrl + Shift + - | 折叠当前代码文件的所有代码块 |
| Ctrl + + | 展开当前代码块（光标所在位置） |
| Ctrl + - | 折叠当前代码块（光标所在位置） |

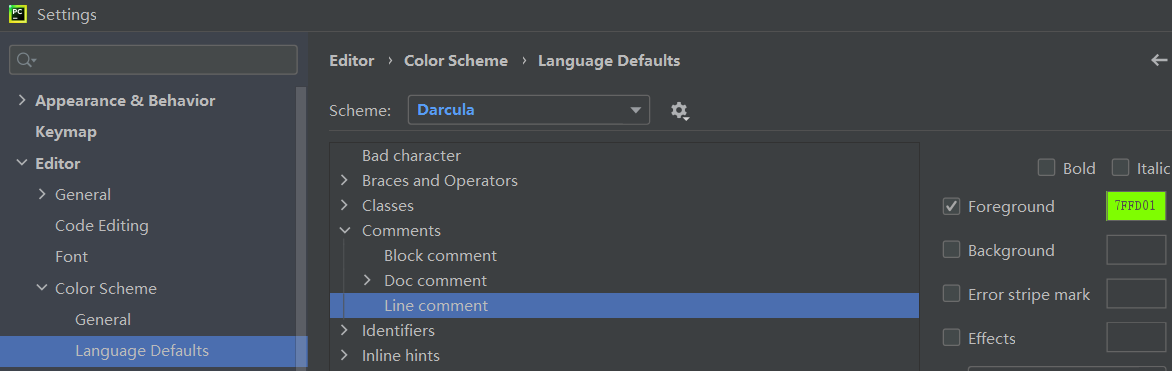
**★设置快捷键**

File → Settings → Keymap → 搜索 "to line end"，编辑Move Caret to Line End的快捷键为 Alt+L



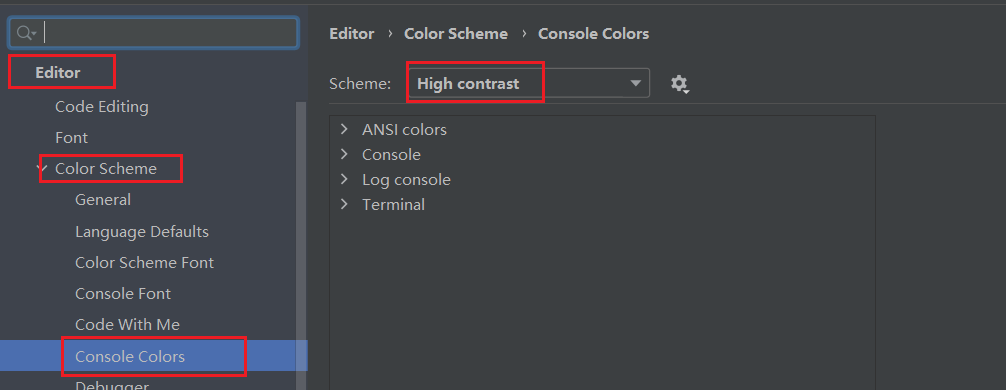
**★设置字体颜色**

File → Settings → Editor → Color Scheme → Language Defaults → 选中目标Scheme



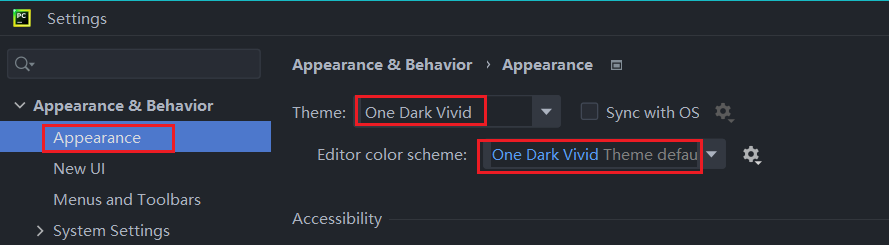
**★设置背景颜色**

File→Settings→Editor→Color Scheme→Console Colors→Scheme → 选择合适的主题



**★设置编程界面代码显示主题**

File→Settings→Appearance & Behavior→Theme：选择一个合适的主题，也可点击“get more themes”下载更多主题



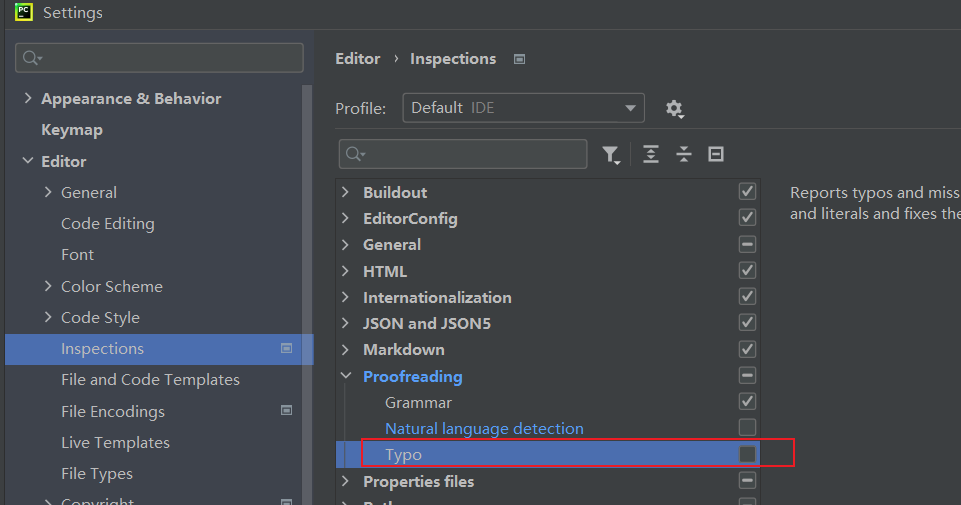
**★显示空白字符**

File → Settings → Editor → General → Appearance → 选中Show whitespaces



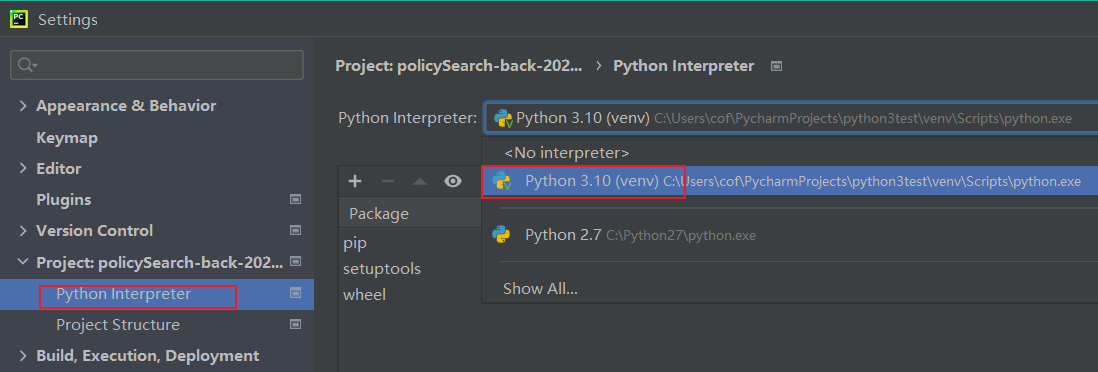
**★不进行拼写检查**

File → Settings → Editor → Inspections → Proofreading → Typo 取消勾选



**★设置项目的python解释器**

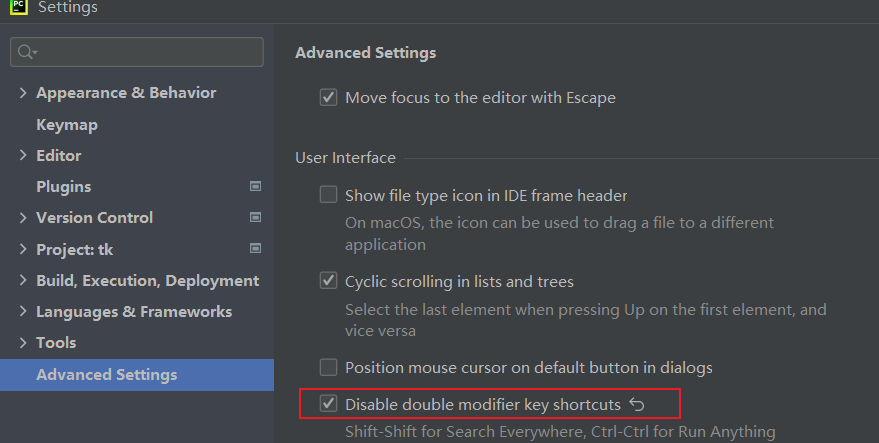
File → Settings → Project: xxx →Python Interpreter → 右侧选择目标python解释器



**★关闭双击shift全局搜索功能**

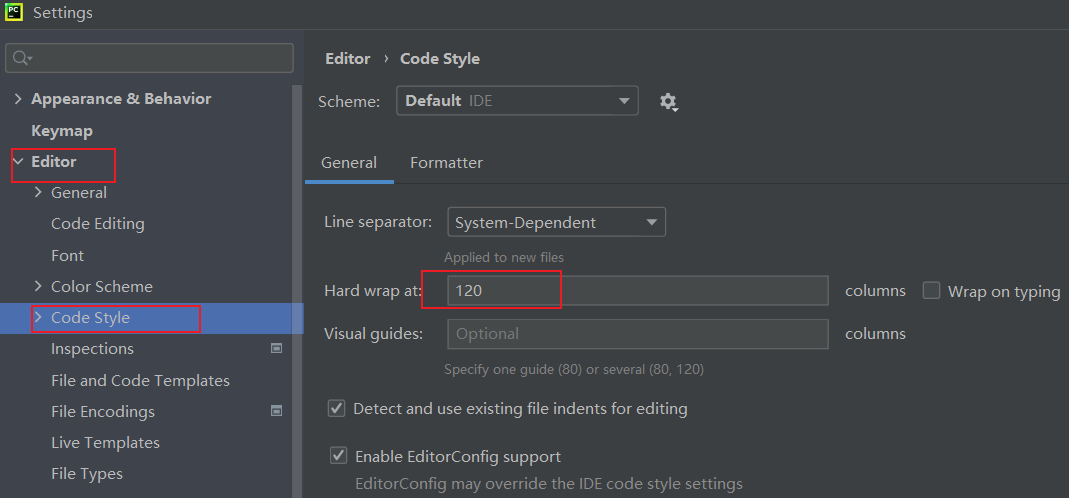
有时我们使用shift切换输入法，若不小心按的次数多了（按2次shift键），结果就弹出全局搜索框，不太友好，可以关闭此功能

File → Settings → Advanced Settings → 右侧的 User Interface ，勾选 Disable double modifier key shortcuts



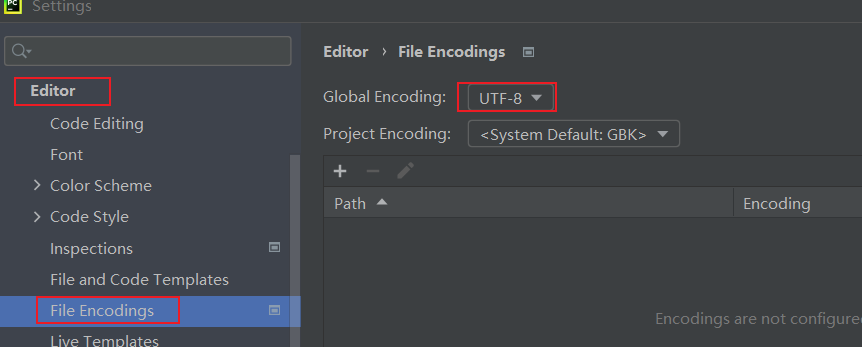
**★设置每行最大长度**

File → Settings → Editor → Code Style → 右侧的 General下的 Hard wrap at: 120（默认为120字符）



**★设置源码文件字符编码**

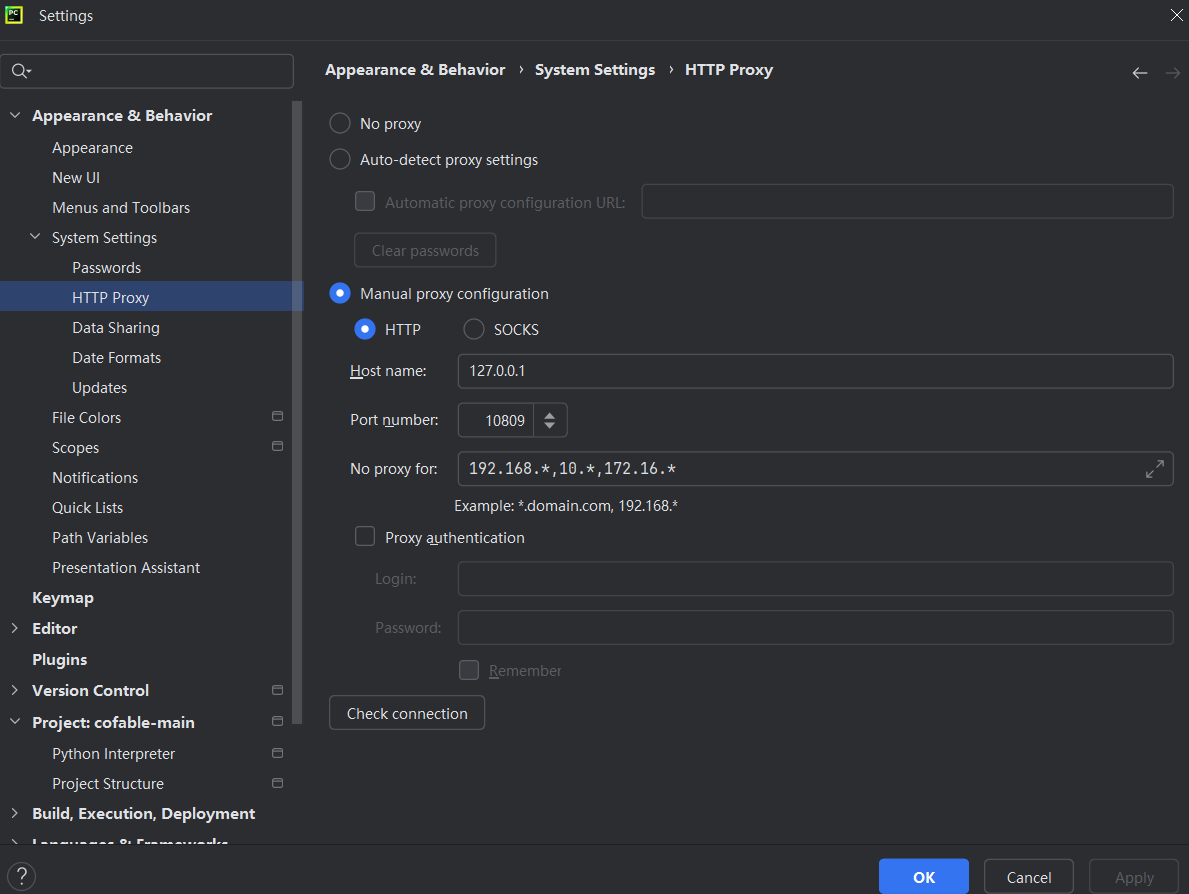
File → Settings → Editor → File Encodeings → 右侧的“Global Encoding”设置目标字符编码，默认UTF-8



**★pycharm设置代理**

通过代理进行模块的安装

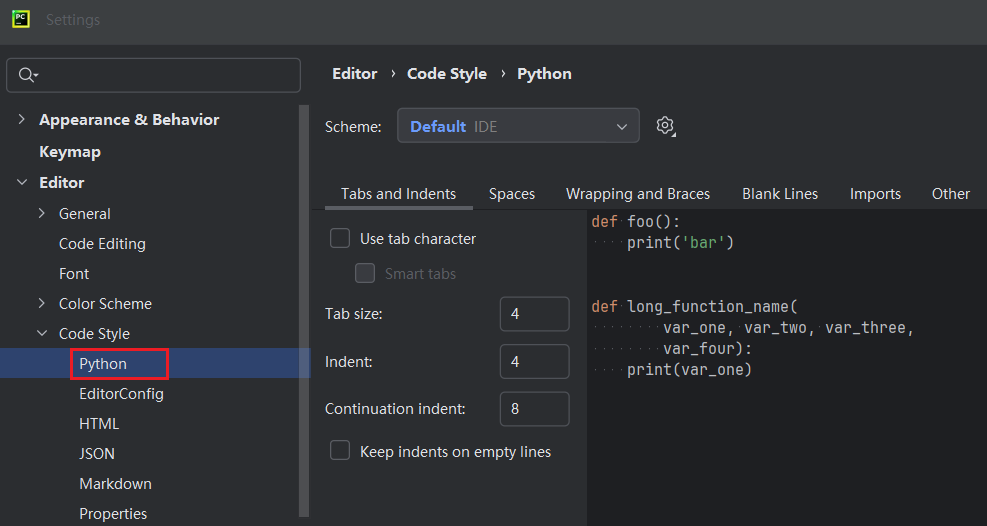
File → Settings → Appearance & Behavior → System Settings → HTTP Proxy



**★设置缩进为4空格**

File → Settings → Editor → Code Style → Python → Tabs and Indents：

取消勾选“Use tab character”，设置Tab size为4，Indent为4



**★第2章、Centos7源码安装python3**

Centos7默认自带python2.7.5版本，**无pip工具**

先去官网下载python3的源码包： https://www.python.org/downloads/source/

或者直接 wget https://www.python.org/ftp/python/3.10.3/Python-3.10.3.tgz



下载目标版本源码包，并上传到目标linux服务器上

**安装命令：**

# yum install -y zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel \

readline-devel tk-devel gcc gcc-c++ make patch libffi-devel python-devel \

gdbm-devel xz-devel libpcap-devel db4-devel

# tar -xvf Python-3.x.x.tgz #解压源码包

# cd Python-3.x.x #进入解压目录

# ./configure prefix=/usr/local/python3.x.x #设置要安装的路径，prefix为可选

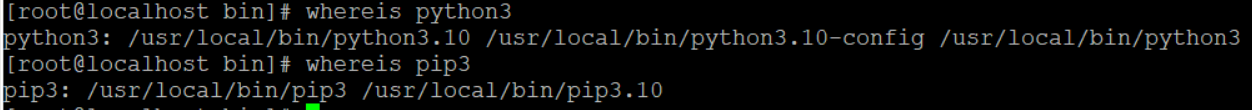
# make && make install

**创建软链接：**

# ln -s /usr/local/python3.x.x/bin/python3 /usr/bin/python3

# ln -s /usr/local/python3.x.x/bin/pip3 /usr/bin/pip3

#默认安装无需指定prefix，也无需创建链接，默认安装路径： /usr/local/bin/



# python2 -c "import sys; print sys.path"

['', #程序当前路径

'/usr/lib64/python27.zip',

'/usr/lib64/python2.7',

'/usr/lib64/python2.7/plat-linux2',

'/usr/lib64/python2.7/lib-tk',

'/usr/lib64/python2.7/lib-old',

'/usr/lib64/python2.7/lib-dynload',

'/usr/lib64/python2.7/site-packages',

'/usr/lib/python2.7/site-packages']

# python3 -c "import sys; print(sys.path)"

['', #程序当前路径

'/usr/local/python3/lib/python37.zip',

'/usr/local/python3/lib/python3.7',

'/usr/local/python3/lib/python3.7/lib-dynload',

'/usr/local/python3/lib/python3.7/site-packages']

# python3 --version #查看版本

# python3 -V #查看版本

**★第3章、基础语法**

**★保留字（关键字）**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| True | False | None | and | as | assert |
| async | await | break | class | continue | def |
| del | elif | else | except | finally | for |
| from | global | if | import | in | is |
| lambda | nonlocal | not | or | pass | raise |
| return | try | while | with | yield |  |

可以使用以下命令查看：

import keyword  
print(keyword.kwlist)

**注意：**

print 在py2中是关键字，在py3中不是关键字，是一个函数

True/False在py3中是关键字，彻底禁止赋值

**★标准数据类型**

|  |  |
| --- | --- |
| number | int bool float complex ( 1+2j ) （python3无long长整型） True=1, False=0 |
| string | 有序 strx='xxxx' stry="xxx" 不区别单引号与双引号，单个字符也是string |
| list | 有序 列表，数组 |
| tuple | 有序 元组 |
| dictionary | 无序 字典 {键值对 集合} 关键字必须互不相同 |
| set | 无序 集合，元素不重复（自动删除重复项） py3新增 |
| bytes | 有序 字节数组，其元素值不可变，其元素类型为int（0~255） py3新增 |
| bytearray | 有序 字节数组，其元素值可变，其元素类型为int（0~255） py3新增 |

**不可变类型（immutable）**： Number String Tuple bytes

**可变类型（mutable）**： List Dictionary Set bytearray

①python中的bool值也算是number数值，True等于1，False等于0

if True == int(1):  
 print("True is 1")  
else:  
 print("True is not 1")

结果：

True is 1

②set 集合是由若干个元素组成的，基本功能是 进行成员关系测试 和 删除重复元素

set可以进行数学上的集合运算（差集，并集，交集等）

可用 { ... } 和 set() 函数创建集合，创建空集合必须使用 set()而不能用 {} 因为 {}表示空的字典，例：

a=set("fldsaj")

b=set("fdslj")

a - b #a和b的差集

a | b #a和b的并集

a & b #a和b的交集

a ^ b #a和b中不同时存在的元素

python2中的8进制数是以数字0开头的，比如： num=0775

而py3中的8进制数是以 0o开头的，比如： num=0o775 （py2.6+也支持这种写法了）

**★判断变量类型**

type(x) #返回x的类型，如 <class 'int'>

isinstance(x, int) #返回**True**或False

type()不会 认为子类是父类的一种类型

isinstance(x,int)会认为子类是父类的一种类型

**★类型转换**

|  |  |
| --- | --- |
| map(type, listxx) | 将字符串str型list转为int型list  strlist = ["23","15","6","44"] #元素类型为str  strnn = list(map(int, strlist)) #map()将strlist的元素都转为int型，map()返回的是map类型，需要再使用list()转为list型  tuple(map(int, strlist)) #返回tuple型 |
| int(x) | 将x输出为10进制的整型，这里的x为number |
| int(x,base=n) | 将x输出为10进制的整型，这里的x必须是n进制的string型  如 int("A", base=16)返回值为10 |
| chr(n) | 输入0～255内的整数，返回对应ascii字符，输入可为10进制，也可为16进制，python3的chr()支持unicode（0x00到0x10FFFF） |
| unichr() | 输入任意unicode编码值，返回对应unicode字符，python3的unichr()没了，合并到chr()了 |
| ord('x') | 输入为一个字符，ascii及unicode字符都行，输出为对应的int编码值 |
| hex() | 将一个整型数字转为16进制字符串，0xnn |
| oct() | 将一个整型数字转为8进制字符串，0onn |
| bin() | 将一个整型数字转为2进制字符串，0bnn |

**★str字符串操作**

**引号与三引号**

字符串可以用 引号 括起来，单个字母也是字符串，如下：

str\_xx = "a"  
str\_yy = 'b'

如果引号里还有相同的引号，则需要使用\反斜杠转义

str\_xx = "hello, i am \"cof-lee\""  
print(str\_xx)

结果：

hello, i am "cof-lee"

3个引号（可以是3个单引号，也可以是3个双引号，一般建议只用3个双引号）表示多行注释，比如：

def funcxx():  
 *"""*

*这里是多行注释  
 本函数实现了xxx功能  
 :return:   
 """* print("test")

3个引号也可以做字符串，多行字符串，（可包含 "双引号及'单引号，而不需要使用\反斜杠进行转义）

str\_nn = """this is\n "hello" \t 'world'  
lslslslslslsls  
llllllllllllll  
"""  
print(str\_nn)

结果：（根据结果可知，三个引号里面的\反斜杠也会进行转义）

this is

"hello" 'world'

lslslslslslsls

llllllllllllll

如果不想让字符串里的\反斜杠转义，可以在字符串前加字母r或者R（表示row原始字符串），或者写双反斜杠\\

str\_nn = r"""this is\n hello \t world"""  
str\_nn2 = """this is\\n hello \\t world"""  
print(str\_nn)  
print(str\_nn2)

结果：

this is\n hello \t world

this is\n hello \t world

**大小写转换及替换过滤拆分拼接-相关操作**

strx="what cAnd Wfds sfjdsD"

strx.lower() #全转为小写，返回新的字符串，不改变原字符串

strx.upper() #全转为大写，返回新的字符串，不改变原字符串

strx.capitalize() #句首字母大写

strx.title() #将每单词首字母大写

strx.swapcase() #全部大小写翻转，互换

strx.replace("old", "new") #把strx中的old替换成new

strx.rstrip() #过滤掉换行符及行尾的所有空白符

strx.lstrip() #过滤掉行首的所有空白符

strx.strip() #过滤位于行首以及行尾的所有空白字符以及换行符

strx.split() #将strx拆分，默认以空白符（含换行符）为拆分符

strx.split(" ") #将strx拆分，""之间指定拆分符

strx.split(" ", 1) #将strx拆分，""之间指定拆分符，且只拆分1次，即分成2部分

strx.isdigit() #判断strx是否为纯数字字符，是则返回True，否则返回False

strx.isalpha() #判断strx是否为纯字母字符，是则返回True，否则返回False（仅大小写字母）

strx.islower() #判断strx是否为小写字母字符，是则返回True，否则返回False

strx.isupper() #判断strx是否为大写字母字符，是则返回True，否则返回False

strx.isalnum() #判断strx是否为字母**或**数字字符，是则返回True，否则返回False

lists=["abc","def","what"]  
strx="".join(lists) #默认直接拼接可遍历的对象，元素只可为str，中间不会加入其他字符或删除其他字符  
stry=",".join(lists) #拼接时在2元素str间添加 ""引号中指定的符号，这里是添加逗号

strx = "hello"  
newstrc = strx.center(30, "\*") # 输出总字符数30，strx显示为居中对齐，其余空位使用""引号中指定的\*补全  
newstrl = strx.ljust(30, "\*") # 左对齐，\*补全  
newstrr = strx.rjust(30, "\*") # 右对齐，\*补全

import re  
  
src\_str = """select \* from ? where name='?' and age=?; [student,cof,18]"""  
match\_obj = re.search(r'\[(.\*?)\]', src\_str)  
  
sql\_str = src\_str[0:match\_obj.start()]  
print("原始sql语句: ", sql\_str)  
  
param\_str = src\_str[match\_obj.start():]  
param\_str = param\_str.replace("[", "")  
param\_str = param\_str.replace("]", "")  
param\_list = param\_str.strip().split(",") # 到这一步，已经将方括号里的所有参数转为 str 的列表了  
print("识别到的参数list: ", param\_list)  
  
new\_sql\_str = sql\_str.replace('?', '%s').replace('%s', '%s', len(param\_list) - 1) % tuple(param\_list)  
  
print(new\_sql\_str)

结果：

原始sql语句: select \* from ? where name='?' and age=?;

识别到的参数list: ['student', 'cof', '18']

select \* from student where name='cof' and age=18;

**★十六进制字符串与bytes互换**

hex\_str = "07E9040E0B3300002B0800"  
bytes\_obj = bytes.fromhex(hex\_str) # hex字符串转为bytes  
print(bytes\_obj)  
print(bytes\_obj.hex(" ", 2)) # bytes转为hex字符串，且以空格做间隔，每2字节为一组

结果：

b'\x07\xe9\x04\x0e\x0b3\x00\x00+\x08\x00'

07 e904 0e0b 3300 002b 0800

**★print输出带颜色的字linux**

此种方法仅linux命令行下有效

print( "\033[0m 带颜色的文字 \033[0m" )

# \033[和m之间的数字为显示效果及颜色的 数字代码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数字代码 | 效果 | 颜色 | 前景色代码 | 背景色代码 |
| 0 | 默认，灰白 | 黑 | 30 | 40 |
| 1 | 粗体 | 红 | 31 | 41 |
| 4 | 下划线 | 绿 | 32 | 42 |
| 5 | 闪烁 | 黄 | 33 | 43 |
| 7 | 反白显示 | 蓝 | 34 | 44 |
|  |  | 紫 | 35 | 45 |
|  |  | 青 | 36 | 46 |
|  |  | 灰白，同0默认 | 37 | 47 |

效果+颜色用;分号隔开

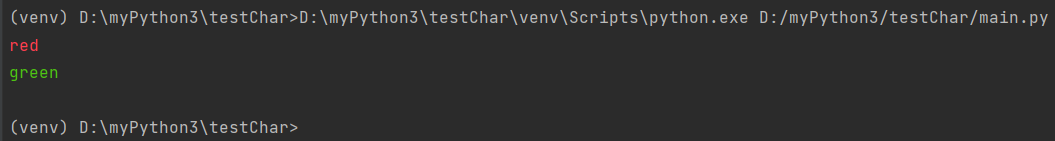
print("\033[4;31m这是带下划线红色的字\033[0m")



**★print输出带颜色的字windows**

此种方法仅windows命令行下有效

import ctypes  
  
STD\_INPUT\_HANDLE = -10  
STD\_OUTPUT\_HANDLE = -11  
STD\_ERROR\_HANDLE = -12  
FOREGROUND\_BLACK = 0x0  
FOREGROUND\_BLUE = 0x01 # text color contains blue.  
FOREGROUND\_GREEN = 0x02 # text color contains green.  
FOREGROUND\_RED = 0x04 # text color contains red.  
FOREGROUND\_INTENSITY = 0x08 # text color is intensified.  
BACKGROUND\_BLUE = 0x10 # background color contains blue.  
BACKGROUND\_GREEN = 0x20 # background color contains green.  
BACKGROUND\_RED = 0x40 # background color contains red.  
BACKGROUND\_INTENSITY = 0x80 # background color is intensified.  
  
std\_out\_handle = ctypes.windll.kernel32.GetStdHandle(STD\_OUTPUT\_HANDLE)  
ctypes.windll.kernel32.SetConsoleTextAttribute(std\_out\_handle,  
 FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE) # reset  
ctypes.windll.kernel32.SetConsoleTextAttribute(std\_out\_handle, FOREGROUND\_RED | FOREGROUND\_INTENSITY) # red  
print("red")  
ctypes.windll.kernel32.SetConsoleTextAttribute(std\_out\_handle, FOREGROUND\_GREEN | FOREGROUND\_INTENSITY) # green  
print("green")  
ctypes.windll.kernel32.SetConsoleTextAttribute(std\_out\_handle,  
 FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE) # reset



**★运算符**

①**算术运算符**

+ - \* /

% 取模，取余数

\*\* 幂，如 A \*\* 4 表示A的4次幂

// 取整除，取商的整数部分，向下取整，去尾法；若运算数为float，则结果也是flost

②**比较运算符**

== 等于 != 不等于

> 大于 >= 大于等于

< 小于 <= 小于等于

③**逻辑运算**

not and or

**★链式比较**

if 3 > 2 == True:  
 print("yes")  
else:  
 print("no")

结果：

no

原因是python会将 3 > 2 == True拆解为 3 > 2 and 3 == True: 结果是False

④**成员运算**

in # a in Listx ; a在Listx中则返回True

not in # a not in Listx ; a不在Listx中则返回True

⑤**身份运算符**

is # a is b ; 判断a和b是否引用自同一个对象，是则返回True

is not

# a is b 同 id(a) == id(b) # id(x)用于获取对象x的内存地址

**⑥位运算符**

& 与运算 ^ 异或

| 或 << 左位移

~ 取反 >> 右位移

**★运算符优先级：**

指数\*\* > 位运算 > 乘除 > 加减 > 比较 > 赋值 > 身份 > 成员 > 逻辑

**★整除运算**

在python 2中，**/** 如果参与运算的数为int，就是整除，只取整数

如果 / 参与运算的数为float，则返回float，不再是整数

在python 3中，/ 不论参与运算的数为int还是float，都不是整除，都返回float

若参与运算的数只有int，则可用 **//** 运算符，取整数商

若参与运算的数为float，则**//**也是返回float，如果要保证能返回整数，需要强制转为int，才保险一些，如：

a = 4.8  
b = 2  
c = int(a // b)  
print(c)

结果：

2

也可用 import math ; math.trunc(xx/yy) 来取整，(xx/yy)可为float，结果也是整数商

import math  
  
a = 9  
b = 2.9  
c = math.trunc(a / b)  
print(c)

结果：

3

**★小数取整**

①内置函数round()为**四舍五入**，n.5则靠偶数取整 ROUND\_HALF\_EVEN （这点和py2不同，py2是向上取整）

round(2.5) #返回2

round(3.5) #返回4

②math.ceil() **进一法**

import math

math.ceil(3.1) #返回4

③math.floor() **去尾法**

import math

math.floor(3.99) #返回3

**★条件语句**

**（python<3.10版本不支持switch语句，在3.10版本开始支持switch语句）**

if 判断句 :

语句体

elif 判断句 :

语句体

else:

语句体

**★循环语句**

① **for循环**

for i in xx : # xx为可遍历的数据，如list,tuple,bytes,bytearray

break #跳出当前循环，跳出for

#使用索引：

for i in range(10) : #生成10个数，从0开始，0到9

# To Do

# range(1,9) #从1开始，1到8

# range(1,9,2) #从1开始，步长为2，即1，3，5，7

**注意：**

py2中的 range()是列表生成器，得到的是一个列表；xrange()是惰性迭代器

优先使用xrange()做循环迭代，节省内存且速度更快

py3只有range了（但py3的range就是py2中的xrange）

>>> ll=range(5)

>>> print(type(ll))

<class 'range'>

②**while循环**

while 判断 **:**

循环体

continue #continue跳过剩下的循环体，进入下一轮loop，break跳出整个循环

**★二维list**

直接定义：

matrix = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

间接定义：

matrix = [[0 for i in range(3)] for i in range(4)] # 4个小list，每小list为3元素（4行3列）  
print(matrix)

matrix = [0 for \_ in range(10)] # 下划线表示占位符，如果写 i，有时会提示i未被使用  
print(matrix)

三维类推：

matrix = [[[0 for \_ in range(3)] for \_ in range(4)] for \_ in range(2)]  
print(matrix)

**#查看数组行数与列数**

arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]  
print(len(arr)) # 行数  
print(len(arr[0])) # 列数

**★list/set/dictionary增删改查操作**

**①list数据操作（有序，数值可重复）**

listxx.append(x) # 把元素x追加到列表的末尾  
listxx.insert(i, x) # 在索引i指定位置插入元素x，原listxx[i]后移一位  
listxx.extend(listyy) # 把listyy追加到列表的末尾  
listxx.remove(x) # 删除列表中第一个值为x的元素，若无这样的元素则返回一个error  
listxx.pop(i) # 从列表的指定索引位置移除元素，并返回此元素值，如果不指定i，则移除并返回最后一个元素

listxx.popleft() # 移除列表头元素并返回其值  
listxx.index(x) # 返回列表中第一个值为x的元素的索引  
listxx.count(x) # 返回x在列表中出现的次数  
listxx.sort() #对列表中的元素排序，改变列表本身，返回值为None或error，不返回新的list，默认升序，从小到大  
listxx.sort(reverse=True) # 排序，降序，从大到小

listxx.reverse() # 反转列表，头变尾，尾变头

del listxx[1:3] # 切割操作，删除列表的这一段，含头不含尾

newlist = sorted(listxx) # 返回新列表，且排好序了，升序，原listxx不变

list\_xx = ["a", "b", "c"]  
for i, item in enumerate(list\_xx): # 返回索引及相应的元素  
 print(f"第 {i} 个元素为 {item}")

结果：

第 0 个元素为 a

第 1 个元素为 b

第 2 个元素为 c

对列表排序（当列表的元素为tuple时）：

users = [("cof", 18),  
 ("lee", 19),  
 ("wong", 17),  
 ("tom", 20),  
 ("duck", 16)  
 ]  
new\_users = sorted(users, key=lambda x: x[1])

# users.sort(key=lambda x: x[1]) #对自己排序也一样的  
print(new\_users)

结果：

[('duck', 16), ('wong', 17), ('cof', 18), ('lee', 19), ('tom', 20)]

**②set数据操作（无序，数值不重复）**

★若在创建set时，有多个重复的元素，则只会存储其一，元素只能是不可变对象

t=("wh","fdjs","fkdsl")  
setyy=frozenset(t) # 创建不可变set，无add remove update操作

setxx.add("xxx") # 新增元素

setxx.remove("bb") # 删除元素，若元素不存在则返回KeyError，存在则返回None

setxx.update(iterablexx) #添加多个元素，这些元素是iterablexx里的，iterablexx可为list,set,tuple

def is\_xxx\_member(name):  
 return name in {"wong", "tom", "lee"} # 直接判断用户是否是此set里的成员（集合的in操作是hash查找，速度远超线性扫描的list）  
  
print(is\_xxx\_member("cof"))

结果：

False

**③dictionary数据操作（无序，key不重复）**

dictxx["new\_key"] = "new\_value" # 新增键值对，如果已存在则更新值

dictxx.update(lx) #添加多个键值对，lx为另一dictionary

value2 = dictxx["k2"] # 取指定key对应的值

k,v=dictxx.popitem() # 随机返回并删除一个键值对

del dictxx["k2"] # 删除指定key的这对键值对

**★遍历字典dictionary**

①遍历key

dictx = {"k1": "v1", "k2": "v2", "k3": "v3"}  
for key in dictx.keys():  
 print(key, "-------->", dictx[key])

**#判断字典中是否存在某个key**

dictx = {"k1": "v1", "k2": "v2", "k3": "v3"}  
if "k3" in dictx.keys():  
 print(dictx["k3"])  
else:  
 print("k3不存在字典中")

dictx = {"k1": "v1", "k2": "v2", "k3": "v3"}  
value = dictx.get("k3") # 如果key不存在，则默认返回None  
if value is None:  
 print("k3不存在字典中")  
else:  
 print(dictx["k3"])

dictx = {"k1": "v1", "k2": "v2", "k3": "v3"}  
value = dictx.get("k33", "null") # 如果key不存在，则返回指定的值"null"  
print(value)

②遍历value

dictx = {"k1": "v1", "k2": "v2", "k3": "v3"}  
for value in dictx.values():  
 print(value)

③遍历item

dictx = {"k1": "v1", "k2": "v2", "k3": "v3"}  
for k, v in dictx.items():  
 print(k, "---->", v)

访问字典下的子字典：

dict\_xx = {  
 "key1": "value1",  
 "key2": {  
 "subkey1": "xxx",  
 "subkey2": [  
 {  
 "subsubkey1": "yyy",  
 "subsubkey2": "zzz"  
 },  
 {  
 "subsubkey3": "nnn"  
 }  
 ]  
 }  
}  
print(dict\_xx["key2"]["subkey2"][0]["subsubkey1"])

结果：

yyy

dict\_xx = {  
 "key1": "value1",  
 "key2": "value2"  
}  
try:  
 value = dict\_xx["key2"][0]["subsubkey11111"]  
 print(value)  
except KeyError:  
 print("dict\_xx 没有指定的key")  
except TypeError:  
 print("dict\_xx['key2']不是一个序列")  
except IndexError:  
 print("dict\_xx['key2']没有指定的索引[1]")

结果：

dict\_xx['key2']不是一个序列

**★string和bytes转换**

strxx = "这是w" # 类型为 str 长度 3  
bytesxx = strxx.encode("utf8") # 将str编码为utf8，返回类型bytes，长度 7

print(type(bytesxx[0])) #返回int, bytesxx本身为bytes类型，其元素则为 int

★string的元素仍为str

stryy=bytesxx.decode("utf8") #将bytes类型解码为字符，返回str

**★bytes和bytearray**

bytes字节数组各元素值不可变，bytearray各元素值可变

bytes和bytearray的各元素值都是int类型，取值范围0~255

**①创建bytes数组**

b1 = b"abcd\xff" # 在字符前加修饰符b直接创建，只可以是ASCII字符  
b2 = bytes(3) # 表示创建长度为3的字节数组，默认使用0填充  
b3 = bytes(listx) # 将可遍历对象转为字节数组，该对象各元素皆为int（0~255）

b4=bytes("strxx",encoding="utf8") #将str转为bytes，等价于str.encode("utf8")

print(b3.hex()) # 输出16进制字符串，如AC0B，默认是b'\x11\x16c\xfe'这种

**②创建bytearray数组**

ba1 = bytearray(3) # 表示创建长度为3的字节数组，使用0填充  
ba2 = bytearray(listx) # 将可遍历对象转为字节数组，该对象各元素皆为int(0~255)  
ba3 = bytearray("strxx", encoding="utf8") # 将str转为bytearray, str.encode("utf8")默认返回的是bytes

print(ba2.hex()) # 输出16进制字符串，默认是bytearray(b'\x11\x16c\xfe')

★ bytes和bytearray都可以使用 .decode("utf8") 方法转为string

**★深拷贝和浅拷贝**

**深拷贝：**

deep copy深拷贝是完全复制一个对象，包括对象里的可变类型数据

import copy  
  
ori\_obj = {"mylist": [1, 2, 3]}  
new\_obj = copy.deepcopy(ori\_obj)  
new\_obj["mylist"].append(4) # 对新对象的操作，不会影响到原始对象的内容

print(f"ori\_obj: {ori\_obj['mylist']}")  
print(f"new\_obj: {new\_obj['mylist']}")

结果：

ori\_obj: [1, 2, 3]

new\_obj: [1, 2, 3, 4]

**浅拷贝：**

shallow copy浅拷贝会复制一个新的对象，但对于原对象里的可变类型数据，它只复制了引用，并未完全复制其内容本身，所以对新的对象的操作，会影响到原始对象里的可变类型的数据

ori\_obj = {"mylist": [1, 2, 3]}  
new\_obj = ori\_obj.copy()  
new\_obj["mylist"].append(4) # 对新对象的操作，会影响到原始对象的内容  
print(f"ori\_obj: {ori\_obj['mylist']}")  
print(f"new\_obj: {new\_obj['mylist']}")

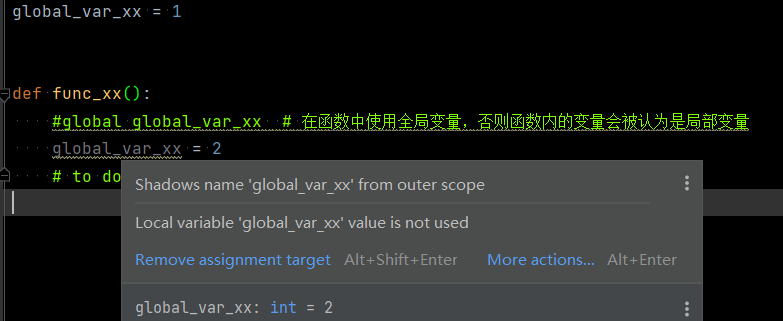
结果：

ori\_obj: [1, 2, 3, 4]

new\_obj: [1, 2, 3, 4]

**★global全局变量**

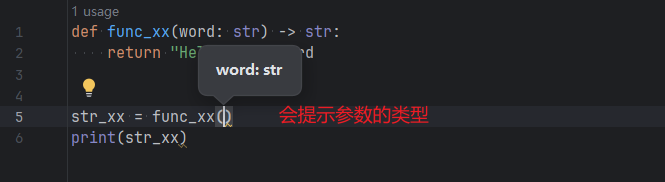
global\_var\_xx = 1  
  
def func\_xx():  
 global global\_var\_xx # 在函数中使用全局变量，否则函数内的变量会被认为是局部变量  
 global\_var\_xx = 2  
 # to do

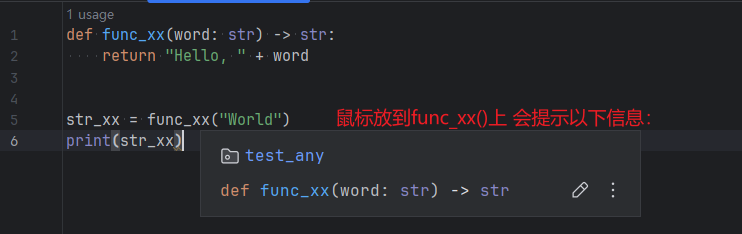


**★类型提示（py>=3.5）**

类型提示是Python3.5引入的一种新特性，用于在代码中指定变量的类型。

def func\_xx(word: str) -> str:  
 return "Hello, " + word  
  
str\_xx = func\_xx("World")  
print(str\_xx)





**★Py2与Py3的区别以及让代码兼容py2与py3**

**#python2与python3的区别：**

|  |  |  |
| --- | --- | --- |
| 区别项： | Python2 | Python3 |
| 发布/停更 | 2000年发布，2020年停止所有维护 | 2008年发布，持续维护 |
| 定位 | 仅用于维护存量系统 | 新项目开发及新版本系统首选 |
| print打印输出 | print "string" #ping是一个关键字 | print("string") #print是一个内置函数 |
| /整除运算 | 3 / 2 = 1 #2个整数相除，结果取整  3.0 / 2 = 1.5 #有浮点数相除，结果才为float  3 // 2 = 1 #整数相除返回整数结果  3.0 // 2 = 1.0 #py2和py3相同 | 3 / 2 = 1.5 #所有除法均为float  3.0 / 2 = 1.5 #所有除法均为float  3 // 2 = 1 #整数相除返回整数结果  3.0 // 2 = 1.0 #返回结果是float |
| str字符串编码 | <str> "str"实际上存储的是字节串（bytes）  <unicode> 显式指明 u"str" 才是unicode文本 | <str> "str"存储的就是unicode文本  <bytes> 存储的是字节串，显式声明 b'xxx' |
| 异常捕获 | 支持2种写法：  except 异常类 , err: #旧式写法  except 异常类 as err: #新式，2.6+支持 | 仅支持新式写法：  except 异常类 as err: |
| 八进制数写法 | 有2种写法：  num = 0755 #前缀数字0  num = 0o755 #2.6+支持，但不推荐 | 仅支持一种写法：  num = 0o755 |
| xrange/range | range() #返回list（耗内存）  xrange() #惰性生成器（省内存） | range() #等同于py2的xrange惰性生成器（省内存），py3没有xrange()这个函数了 |
| 迭代器方法 | dict.keys()及dict.items() 返回的是列表 | dict.keys()及dict.items() 返回视图对象（惰性的，省内存），如果要返回列表，需要：  list(dict.keys()) |
| 输入函数 | raw\_input() #获取输入的字符串  input() #执行输入的代码（比较危险） | input() #获取输入的字符串，等同于py2的raw\_input() |
| 编码默认值 | sys.getdefaultencoding() = 'ascii' | sys.getdefaultencoding() = 'utf-8' |
| subprocess | py2.6无check\_output()，需手动实现，  无timeout参数 | 原生支持check\_output() 及timeout参数  默认输出的是bytes，需要decode()转str |
| datetime | py2.7.9+才支持timestamp()时间戳，  无timezone内置类 | 原生支持timestamp()时间戳，  及timezone内置类 |
| 废弃模块 | commands, urllib2, ConfigParser 等仍可用 | 废弃了commands,  重构urllib为urllib.request/ urllib.parse  ConfigParser改为小写的 configparser |
| 新增模块 | 无核心新增模块 | 新增 asyncio, enum, pathlib 等模块 |
|  |  |  |

①py2的输出语句为 print "str" ，而在py3中使用print("str")函数

②py3的/除法为float除法，不是整除，而py2中/表示整除

③py2的源码文件默认使用ascii编码，除非指定了 # -\*- coding: utf-8 -\*-，py3源码文件默认使用utf8编码

若要在py2中也使用py3中的这些特性，可以导入名为 \_\_future\_\_ 的包，导入后，py2会启导入的功能使得也支持py3的这些特性。同时py2的这些原有特性会被py3覆盖

让源代码兼容python2和python3，只需要在代码开头添加以下几行，然后都用py3的语法去写代码：

from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
from \_\_future\_\_ import unicode\_literals  
  
import sys  
  
if sys.version\_info[0] < 3:  
 reload(sys)  
 sys.setdefaultencoding('utf-8')

**解析：**

★**打印函数**

导入print\_function后，让python2也支持print()函数

（以下代码在py2与py3都可正常执行）

from \_\_future\_\_ import print\_function  
  
str\_1 = "hello world"  
print(str\_1)

**★除法**

division让python2也使用py3的除法规则（浮点数运算）

（以下代码在py2与py3都可正常执行）

from \_\_future\_\_ import division  
  
print('10 / 3 =', 10 / 3)  
print('10.0 / 3 =', 10.0 / 3)  
print('10 // 3 =', 10 // 3)

结果：

('10 / 3 =', 3.3333333333333335)

('10.0 / 3 =', 3.3333333333333335)

('10 // 3 =', 3)

**★编码**

python2的字符串默认使用bytes编码，字符长度同bytes字节长度

#!/usr/bin/env python  
# -\*- coding: utf-8 -\*-  
  
str1 = "我们" # 默认同 b"我们"  
str2 = u"我们"  
print len(str1)  
print len(str2)

结果：

6 # str1默认是bytes，"我们"用utf8编码时为6字节长度

2 # str2强制指定为unicode，2字符长度

让py2也默认使用py3的unicode字面量

#!/usr/bin/env python  
# -\*- coding: utf-8 -\*-  
from \_\_future\_\_ import unicode\_literals  
  
str1 = "我们" # 导入unicode\_literals后，默认同 u"我们"  
str2 = u"我们"  
print len(str1)  
print len(str2)

结果：

2

2

**★绝对引入**

from \_\_future\_\_ import absolute\_import

py2引入包时，直接从当前目录引入；py3默认从环境变量引入

**★第4章、python推导式**

推导式表示从一个数据序列构建成另一个新的数据序列，当我们使用列表推导式时，它会立即构建整个列表，并将所有元素一次性加载到内存中，对于小规模数据或轻量级操作，比较快速。但是在处理大规模数据或者列表推导式内部包含复杂操作时，列表推导式反而会降低速度，成为性能瓶颈。

① **list推导式**

newlist=[表达式 for 变量 in 源列表 if 条件] #直接返回list类型

例：

names=["abc", "xdfjl", "fdklsaj"]

new\_names=[name.upper() for name in names if len(name) >3 ]

② **dictionary推导式**

newdic={ key表达式: value表达式 for 变量 in 列表 if 条件} #直接返回dictionary

③ **set集合推导式**

newset={表达式 for 变量 in 列表/元组} #直接返回set

④ **tuple元组推导式**

newtuple=(表达式 for 变量 in 列表) #直接返回的是生成器对象，不是tuple

例：

xx=tuple( x for x in range(1,10) ) # tuple(推导式) 可得元组

**★第5章、函数def**

**★定义函数**

def fun\_name(参数): #可有传入参数，也可没有

"描述"

语句

return xx #若未指定return，则默认返回None

**★参数传递**

**①不定长参数**

def func( \*var ): #参数以tuple形式传入，var为tuple

#或者传参数时直接传入多个值，以逗号,隔开

#若传入的变量为tuple时，不带\*号则表示只有一个参数，就是tuple本身，

带\*号则表示把tuple解开，变成多个参数，每个元素为一个参数

**②定长参数加不定长参数**

def func( var1, var2, \*var\_args\_tuple):

#前面2个为固定参数，后面可传入若干个可选参数，调用函数时，不可省略前面的定参

**③默认参数**

def func(var1, var2=10, var3=40):

#调用时若未传入参数var2和var3，则使用默认值10，40

**④解构传入tuple/dict参数**

def func\_test(\*tuple\_xx, \*\*dict\_xx):  
 print(tuple\_xx)  
 print(dict\_xx)  
  
def func\_test2(a, b, c):  
 print(a, b, c)  
  
def func\_test3(name, age, weight):  
 print(name, age, weight)  
  
func\_test(1, 2, 3, name="cof", age=18, weight=70)  
dict1 = {'name': 'cof', 'age': 18, 'weight': 70}  
func\_test2(\*dict1)  
func\_test3(\*\*dict1)

结果：

(1, 2, 3)

{'name': 'cof', 'age': 18, 'weight': 70}

name age weight

cof 18 70

def func\_test(\*args, \*\*kwargs):  
 for arg in args:  
 print(arg)  
 for key, value in kwargs.items():  
 print(key, value)  
  
func\_test(1, 2, 3, name="cof", age=18, weight=70)

结果：

1

2

3

name cof

age 18

weight 70

**说明：**

\*是对tuple的解构，\*可对dict拿到key值(str)

\*\*是对dict的解构，

**⑤传参是否可变**

不可变类型： number string tuple bytes 传递的是值

可变类型： list dictionary set bytearray <类的实例/对象> 传递的是引用

将不可变类型参数传入函数后，函数里面对参数的修改，**不会改变**函数外的变量值

将可变类型参数传入函数后，函数里面对参数的修改，会改变函数外的变量值

**★内嵌函数**

内嵌函数就是在一个函数内部定义的函数

内嵌函数可以在外部函数内部被调用，也可以在外部函数之外的地方被调用。但是，内嵌函数在外部函数之外是不可见的，即无法直接调用。

def func\_out():  
 str1 = "Hello"  
  
 def func\_inner():  
 nonlocal str1 # 指示str1不是内嵌函数的变量，而是其外部函数func\_out()里的变量  
 str1 = str1 + " World"  
  
 func\_inner()  
 return str1  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print(func\_out())

结果：

Hello World

**★第6章、python3内置函数**

内置函数是指不用import导入任何模块就能直接使用的函数

**①算术相关**

abs(x) #返回x的绝对值

min(x1,x2,x3) #返回最小值

hex(x) #将x转换成16进制数，带0x

oct(x) #将x转换成8进制数，带0o

bin(x) #将x转换成2进制数，带0b

int(x) #将x转换成10进制数，默认认为x为10进制数/str，base=16指定x字符串为16进制数

divmod() 函数接收两个数字类型（非复数）参数，返回一个包含商和余数的元组(a // b, a % b)

②生成器

dict() #用于生成一个dictionary

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

slice() 函数实现切片对象，主要用在切片操作函数里的参数传递

sorted() 函数对所有可迭代的对象进行排序操作，默认升序，reverse=True则降序，返回新的list；list 的 sort 方法返回的是对已经存在的列表进行操作

③查看帮助类

help() 查看目标对象的帮助信息

dir() 函数不带参数时，返回当前范围内的变量、方法和定义的类型列表；带参数时，返回参数的属性、方法列表。如果参数包含方法\_\_dir\_\_()，该方法将被调用。如果参数不包含\_\_dir\_\_()，该方法将最大限度地收集参数信息

④设置，获取

setatrr() 设置对象的属性

getattr() 获取对象的属性

a = A()

>>> getattr(a, 'bar') # 获取属性 bar 值

1

>>> setattr(a, 'bar', 5) # 设置属性 bar 值

>>> a.bar

5

next() 返回迭代器的下一个项目。

next() 函数要和生成迭代器的 iter() 函数一起使用

id() 函数返回对象的唯一标识符，标识符是一个整数。CPython 中 id() 函数用于获取对象的内存地址

input() 函数接受一个标准输入数据，返回为 string 类型

staticmethod() 返回函数的静态方法

⑤判断

all() 函数用于判断给定的可迭代参数 iterable 中的所有元素是否都为 TRUE，如果是返回 True，否则返回 False（元素除了是 0、空、None、False 外都算 True）

any() 函数用于判断给定的可迭代参数 iterable 是否全部为 False，则返回 False，如果有一个为 True，则返回 True。元素除了是 0、空、FALSE 外都算 TRUE

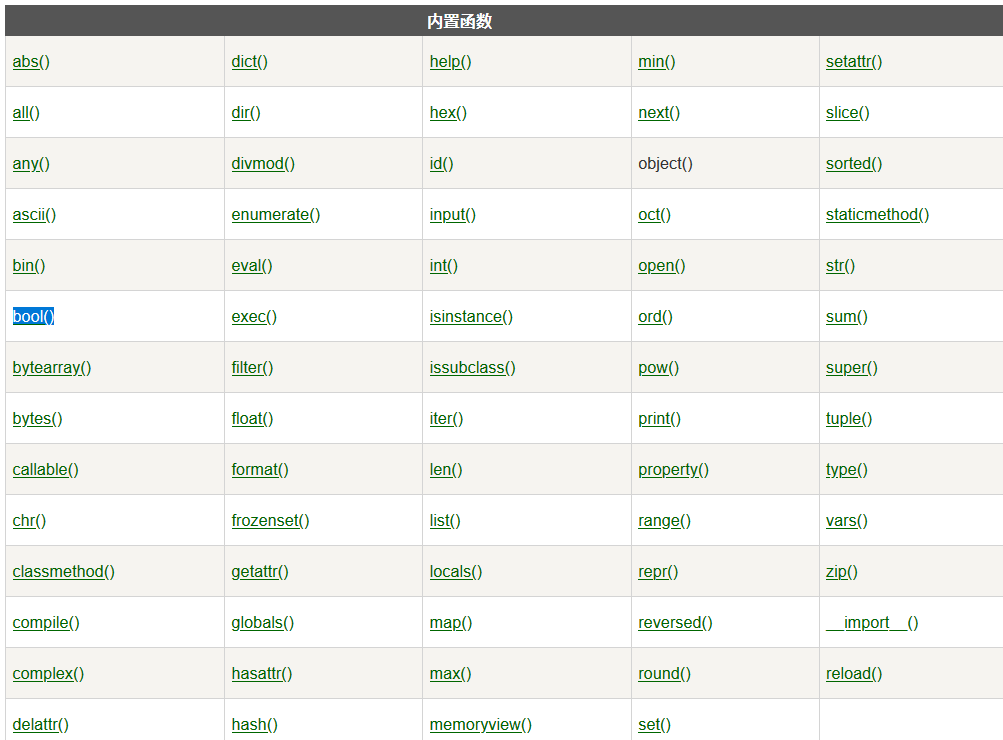
⑥其他

eval() 函数用来执行一个字符串表达式（为python表达式），并返回表达式的值

open() 函数用于打开一个文件，并返回文件对象

ascii() 函数类似 repr() 函数, 返回一个表示对象的字符串, 但是对于字符串中的非 ASCII 字符则返回通过 repr() 函数使用 \x, \u 或 \U 编码的字符

str() 函数将对象转化为适于人阅读的形式



https://www.runoob.com/python3/python3-built-in-functions.html

**★第7章、输入输出**

**★标准输入函数input()**

内置函数input()为标准输入函数，执行到此函数时程序会暂停住，让用户从console输入，读到换行符结束，返回string，返回内容不含换行符

stdin = input()

stdin2 = input("提示：")

在linux命令行执行程序脚本时，可使用 | 管道符输入到标准输入，这时就不会暂停住了

# echo "inputarg" | ./test.py

**★标准输入列表sys.stdin.readlines**

sys.stdin有2个方法：

import sys

sys.stdin.readline() #读取一行，读到换行符结束，内容包含换行符，返回<str>

sys.stdin.readlines() #读取所有行，读到Ctrl+D结束，内容包含换行符，返回<list>，每行作为一个元素<str>

**★命令行输入列表sys.argv**

import sys

print("以下为命令行参数：", sys.argv) #sys.argv为命令行输入的参数list，第一个参数永久是程序本身（脚本名称）

在linux命令行执行程序脚本时，可直接跟在脚本后的参数称为命令行参数

# ./test.py arg1 arg2 #直接传入 命令行参数

# echo "arg1" | xargs ./test.py # xargs将标准输入转为命令行参数

**★标准输出print()**

**①不指定格式**

print( arg1, arg2, arg3 ) #输出时3个参数之间默认使用空格隔开

print( arg1, arg2, sep="" ) #输出时不使用任何字符隔开，也可在""内指定分隔符

print()默认是带换行的，即默认结尾符为\n，可用end=""指定结尾符为空（不带换行）

如 print( argx, argv, end="" )

**②print()旧式字符串格式化**

同C语言的sprintf()的格式化字符串用法，用 % 操作符

print( "这是第%2d号，体重%5.2f公折" % (num, wei) )

%-6.3f 表示左对齐，共6位宽，3位小数，浮点型（默认左对齐）

|  |  |  |  |
| --- | --- | --- | --- |
| 类型码 | 含义 | 类型码 | 含义 |
| %c | 字符 | %s | 字符串 |
| %d | 十进制整数 | %u | 无符号整数 |
| %o | 八进制数 | %x | 小写十六进制数 |
| %X | 大写十六进制数 | %f | 浮点数 |
| %e | 科学计数法，小写e | %g | 自动采用最短的%f或%e |
| %F | 同%f，会把inf,nan写成INF,NAN | %E | 同%e,大写E |
| %G | 同%g，会大写 | %b | 二进制数，print()不支持 |

**★字符串.format()格式化函数**

"字符串".format(argxx) 的新式字符串格式化函数，是python 2.6开始引入的一种方法

.format()通过将strxx字符串中的{}来识别替换字段

{字段名!转换字段:格式说明}

print( "我叫{}，年龄{}岁".format("李", 20) )

print("我叫{0}，年龄{1}岁".format("李", 20))

结果：

我叫李，年龄20岁

**①字段名**

★可省，就写{}，默认从左到右对应后面.format()里的arg

★写数字，0，1，2 可重复用右边的字段，如0，1，1，1

★写变量名 xxx

.format()里**传入tuple**参数时，得带上\*号，如：

cof = ("李", 20)

print("我叫{0}，年龄{1}".format( \*cof) )

**传入dictionary**时，得带上\*\*双星号

**传入类的对象**时，不带星号，在{}里取对象的属性

a = classx("李", 20)

print("我叫{0.name}，今年{0.age}".format(a))

**传入list,tuple,dictionary时**，若不想按顺序取值，可用.[i]按下标取值

a=[1,2,3]

print("我{0.[1]}，在{0.[0]}里".format(a))

在 Python 2.x 中，ValueError: zero length field name in format 是格式化字符串时的常见错误，核心原因是使用了空占位符（{}）但当前Python版本不支持——Python 2.6 及更早版本要求格式化占位符必须指定索引或名称，而空占位符 {} 是Python 2.7+/3.1+ 才支持的语法，py2.6及以下版本 必须指定索引或名称，如 {0} {name}

**②转换字段**

!s 把参数str()一下，即调用str()去格参数

!r repr()

!a 返回ascii()，或\unnnn \xnnnn

**③格式说明 :**

前面有:冒号

顺序为 :填充 对齐方式 正负号 # 宽度 .精度 类型码

**对齐方式**

< 左对齐

> 右对齐

^ 居中

后面必须指定宽度

print("我在{:<6}号".format( n ))

**填充**

在:后，对齐符前 写指定符号，如用0填充

n = 3  
print("我在{:0>6}号".format(n))

结果：

我在000003号

n = 3.1  
print("我在{:0>6.2f}号".format(n)) # 6位宽度，小数点后保留2位

结果：

我在003.10号

**正负号**

让正数带上+号

print("我在{:0>+6}号".format( n )) #0填充默认在+号与后面数字之间了，这不是我们想要的，这时得用=指定对齐方式，默认右对齐

print("我在{:0=+6}号".format( n ))

**数字形式带前缀**

#b 输出带0b前缀

#o 输出带0o前缀

#x 输出带0x前缀

**,数字分组分隔**

让数字以3个一组带上逗号分隔符

,d 仅对应,d，不可用,f ,x之类

\_d \_o \_x \_b 可对应这4个，以每4个字符为一组带\_下划线输出

**%百分比形式**

{:%} 自动将数字乘以100带上%号，浮点数表示了，默认6位小数

{:.2%} 指定2位小数精度

**类型码**

最后是类型码，同print()旧式字符串格式化里的类型码

d 十进制整数

**★格式化输出之f-string**

python3.6开始支持新的字符串格式化语法，使用字母f加格式化的字符串，功能和"str".format()类似

f-string的优点是比str.format()更简洁，易读，快速

name = "cof-lee"  
print(f"这是{name}")

结果：

这是cof-lee

假如想在字符串里输出 {} 花括号本身，则需要写双花括号 {{ }} ，示例：

name = "cof-lee"  
str\_xx = f"{{'name': '{name}'}}"  
print(str\_xx)

结果：

{'name': 'cof-lee'}

**★对齐方式及宽度指定**

weight = 60.2  
print(f"his weight is {weight:.3f}") # 小数点后保留3位

结果：

his weight is 60.200

weight = 60.2  
print(f"his weight is {weight:5.2f}") # 5位宽度，小数点后保留2位

结果：

his weight is 60.20

weight = 139243260.2  
print(f"his weight is {weight:,}") # 添加千位分隔符

结果：

his weight is 139,243,260.2

weight = 60.2  
print(f"his weight is {weight:0>8.3f}") # 共8位，>向右对齐，用0填充，小数点后保留3位

结果：

his weight is 0060.200

**★进制转换**

num = 164  
print(f"{num:x}") # 输出十六进制数  
print(f"{num:o}") # 输出八进制数  
print(f"{num:b}") # 输出二进制数  
num2 = 0xff  
print(f"{num2:d}") # 输出十进制数

结果：

a4

244

10100100

255

**★百分比**

num = 0.2345  
print(f"{num:.2%}") # 输出百分比，2位小数

结果：

23.45%

**★argparse命令行参数识别**

Python 2 的 optparse 和 Python 3 的 argparse 均是用于解析命令行参数的标准库，argparse 是 optparse 的升级版（Python 3.2+ 正式替代 optparse，且 optparse 在 Python 3 中被标记为废弃）。二者核心目标一致，但 argparse 解决了 optparse 的诸多限制，功能更强大、易用性更高。

import argparse  
  
# 创建一个ArgumentParser对象  
parser = argparse.ArgumentParser(description="使用示例") # description="--help时的说明标题"  
  
# 添加选项参数（optional arguments）  
parser.add\_argument("-u", "--user", action='store', type=str, default="default\_username",  
 required=False, help="用户名") # action=store表示 存储传入的值，若无此选项则返回default=指定的值  
parser.add\_argument("-p", "--passwd", type=str, default="default\_password", help="密码") # 无action=参数时，默认就是store  
parser.add\_argument("-i", "--init", action="count", required=False, default=0,  
 help="初始化，多次传参则累加计数") # 累计本选项出现的次数，返回int（计数），无此选项则返回默认值0  
parser.add\_argument("--ip", action="append", type=str, required=False,  
 help="目标ip，可多次传参") # action=append时，将每个传入的值添加到一个列表中  
parser.add\_argument("-t", "--dry-run", action="store\_true", required=False,  
 help="如果该项出现，则设置值为True，否则False") # action=store\_true时，如果该项出现，则设置值为True，否则False  
parser.add\_argument("-f", "--not-dry-run", action="store\_false", required=False,  
 help="如果该项出现，则设置值为False，否则True") # action=store\_false时，如果该项出现，则设置值为False，否则True  
# 添加必填参数，位置参数（positional arguments）  
parser.add\_argument("target1", action='store', type=str) # 直接输入值，不需要也不能带选项名称'target1'  
parser.add\_argument("target2", action='store', type=str) # 直接输入值，不需要也不能带选项名称'target2'  
# 说明  
"""  
以 - 1个减号开头的，只有一个字母的选项称为 短选项  
以-- 2个减号开头的，后面跟完整的单词的选项称为 长选项  
action='' 表示有此选项时的动作，默认是store  
 store 存储参数后面的值，返回类型由 type=指定，如果无此选项也未指定default值，则返回None  
 count 累计本选项出现的次数，返回int，如果无此选项也未指定default值，则返回None（选项后不可带值）  
 append 将每个传入的值添加到一个列表中，返回 list[type]，如果无此选项也未指定default值，则返回None  
 store\_true 若有该选项，则设置值为True，否则False（选项后不可带值）  
 store\_false 若有该选项，则设置值为False，否则True（选项后不可带值）  
①store\_true与store\_false 类型的短参数可连在一起写，比如 -tf 表示同时指定了 -t -f  
②无论是短选项还是长选项，在传值时，可用空格隔开，也可用=等号连接选择与值  
"""  
args = parser.parse\_args() # 解析命令行参数  
# 获取参数值  
user = args.user # 直接使用选项名称（不含开头的--）做为属性  
passwd = args.passwd  
init\_cound = args.init  
ip\_list = args.ip  
dry\_run = args.dry\_run # 使用选项名称（不含开头的--）做为属性，（中间的-需要转为\_下划线）  
not\_dry\_run = args.not\_dry\_run  
target1 = args.target1  
target2 = args.target2  
print("your input args:")  
print(f" user: {user}")  
print(f" passwd: {passwd}")  
print(f" init\_cound: {init\_cound}")  
print(f" ip\_list: {ip\_list}")  
print(f" dry\_run: {dry\_run}")  
print(f" not\_dry\_run: {not\_dry\_run}")  
print(f" target1: {target1}")  
print(f" target2: {target2}")

执行效果：

# python.exe test.py --help

usage: test.py [-h] [-u USER] [-p PASSWD] [-i] [--ip IP] [-t] [-f] target1 target2

使用示例

positional arguments:

target1

target2

optional arguments:

-h, --help show this help message and exit

-u USER, --user USER 用户名

-p PASSWD, --passwd PASSWD

密码

-i, --init 初始化，多次传参则累加计数

--ip IP 目标ip，可多次传参

-t, --dry-run 如果该项出现，则设置值为True，否则False

-f, --not-dry-run 如果该项出现，则设置值为False，否则True

# python test.py --user=cof -p passxxx -iii --ip ip1 --ip ip2 --ip ip3 -tf tt1 tt2

your input args:

user: cof

passwd: passxxx

init\_cound: 3

ip\_list: ['ip1', 'ip2', 'ip3']

dry\_run: True

not\_dry\_run: False

target1: tt1

target2: tt2

**★第8章、面向对象**

**类是由 数据结构 及对数据进行的操作方法 组成的**

**★类的操作**

**①创建类**

class 类名称:

"""类的帮助信息，描述"""

类的内容

class 类名称(基类1, 基类2): #派生类

类的内容

**②类的方法**

类的方法即类里的函数，python的类方法必须有一个额外的第一个参数，类方法第一个参数名为self，表示类的实例(对象)，而非类本身

定义时有self，但在使用类方法时，不用再写这个参数！

**对象：**通过类定义的数据结构实例

**③构造函数 \_\_init\_\_(self)** #init前后各2个下划线

class MyClassA:  
 xxx = 4 # 类变量，它的值在这个类的所有实例间共享，可在内部/外部使用它  
 def \_\_init\_\_(self, namex, agex): #构造函数，初始化方法  
 self.name = namex  
 self.age = agex

**④创建对象，对象属性**

类的实例化

对象名=类名(x) # x参数是传给\_\_init\_\_()方法的

#对象可直接添加/删除/修改 属性

对象名.newAttr = value

del 对象名.xxAttr

**使用函数来访问属性**

getattr( obj, attrName ) #获取对象的属性

hasattr( obj, attrName ) #检查对象是否存在这个属性

setattr( obj, attrName, valuex ) #设置属性的值，若属性不存在，则创建

delattr( obj, attrName ) #删除属性

**⑤内置类属性**

\_\_dict\_\_ #类的属性，一个字典，由类的数据属性组成  
\_\_doc\_\_ #类的文档字符串  
\_\_name\_\_ #类名  
\_\_module\_\_ #类定义所在的模块  
\_\_bases\_\_ #类的所有父类，一个元组

**⑥析构函数**

\_\_del\_\_(self) #析构方法，在对象销毁时被调用

**⑦特殊属性**

\_protedAttr #以单下划线开头表示protected变量，只允许其本身与子类进行访问

\_\_privateAttr #以双下划线开头表示private变量，只允许这个类本身访问

\_\_privateMethod #以双下划线开头的方法为private方法，只允许这个类本身访问

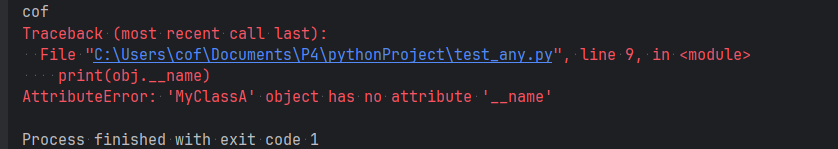
\_\_xxx\_\_() #特殊方法，以双下划线开头及双下划线结尾

对象不可访问类的私有变量（以双下划线开头），但可用以下形式来访问：

对象名.\_类名\_\_私有属性名

class MyClassA:  
 def \_\_init\_\_(self, length): # 构造函数，初始化方法  
 self.length = length  
 self.\_\_name = "cof"  
 print(self.\_\_name)  
  
  
obj = MyClassA(3)  
print(obj.\_\_name)

结果：



换成以下方式，则可成功读取到对象的私有属性：

class MyClassA:  
 def \_\_init\_\_(self, length): # 构造函数，初始化方法  
 self.length = length  
 self.\_\_name = "cof"  
 print(self.\_\_name)  
  
  
obj = MyClassA(3)  
print("对象的私有属性值为", obj.\_MyClassA\_\_name)

结果：



**★\_\_sttr\_\_()与\_\_repr\_\_()**

\_\_str\_\_()和\_\_repr\_\_()方法都用于定义对象的字符串表示，但\_\_str\_\_()更适合用于友好的打印输出，而\_\_repr\_\_()更适合用于精确重现对象。如果两者都被定义，通常情况下\_\_str\_\_()方法会被print()函数等显示函数使用，而\_\_repr\_\_()方法会被交互式解释器使用。

class MyClass:  
 def \_\_init\_\_(self, var="default"):  
 self.var = var  
  
 def \_\_str\_\_(self):  
 return "This is MyClass"  
  
 def \_\_repr\_\_(self):  
 return "MyClass()"  
  
  
obj = MyClass()  
print(obj) # 输出: This is MyClass （有定义\_\_str\_\_()时，优先输出此函数的返回值，否则次优先输出\_\_repr\_\_()，2者都没有定义，则输出对象地址（系统默认格式））  
print(obj.\_\_str\_\_()) # 输出: This is MyClass  
print(obj.\_\_repr\_\_()) # 输出: MyClass()

结果：

This is MyClass

This is MyClass

MyClass()

class MyClass:  
 def \_\_init\_\_(self, var="default"):  
 self.var = var  
  
  
obj = MyClass()  
print(obj) # 输出: 有定义\_\_str\_\_()时，优先输出此函数的返回值，否则次优先输出\_\_repr\_\_()，2者都没有定义，则输出对象地址（系统默认格式）  
print(obj.\_\_str\_\_()) # 输出: This is MyClass  
print(obj.\_\_repr\_\_()) # 输出: MyClass()

结果：

<\_\_main\_\_.MyClass object at 0x000002646EC9F1F0>

<\_\_main\_\_.MyClass object at 0x000002646EC9F1F0>

<\_\_main\_\_.MyClass object at 0x000002646EC9F1F0>

**★\_len\_\_() 对象的长度**

class MyClassA:  
 def \_\_init\_\_(self, length): # 构造函数，初始化方法  
 self.length = length  
  
 def \_\_len\_\_(self):  
 return int(self.length)  
  
  
obj = MyClassA(3)  
print(len(obj))

结果：

3

**★静态方法**

不需要生成类的实例（对象），就可访问其定义的函数（方法），这种方法称为静态方法

class MyClass:  
 version = "1.0"  
  
 def \_\_init\_\_(self, age): # 构造函数，初始化方法  
 self.age = age  
 print("my age is {}".format(self.age))  
  
 @staticmethod  
 def func\_xx(name): # 在方法的上方添加 @staticmethod 表示此方法为静态方法，静态方法没有self参数  
 print("my name is {}".format(name))  
 print("staticmethod: {}".format(MyClass.version)) # 如果要访问类本身的属性，需要指定类名  
  
 @classmethod  
 def func\_yy(cls):  
 # 接收cls参数（代表类本身），可灵活访问、修改类属性  
 print("classmethod：{}".format(cls.version))  
  
MyClass.func\_xx("cof") # 直接使用类的静态方法  
MyClass.func\_yy()

结果：

my name is cof

**★装饰器**

装饰器（Decorator）允许我们在不修改原函数代码的情况下，对原函数添加额外的功能

示例：

def my\_decorator\_1(func):  
 def wrapper(name):  
 print("这是在原函数之前要新增的功能代码")  
 new\_name = name.upper()  
 func(new\_name) # 调用原函数  
 print("这是在原函数之后要新增的功能代码")  
  
 return wrapper  
  
  
@my\_decorator\_1  
def test\_func(name): # 在原函数上面一行添加 @装饰器名称 即可使用此装饰器  
 print(f"my name is {name}")  
  
test\_func("cof")

结果：

这是在原函数之前要新增的功能代码

my name is COF

这是在原函数之后要新增的功能代码

#多次使用装饰器

def my\_decorator\_1(func):  
 def wrapper(name):  
 print("这是在原函数之前要新增的功能代码")  
 new\_name = name.upper()  
 func(new\_name) # 调用原函数  
 print("这是在原函数之后要新增的功能代码")  
  
 return wrapper  
  
  
@my\_decorator\_1  
@my\_decorator\_1  
def test\_func(name):  
 print(f"my name is {name}")  
  
test\_func("cof")

结果：

这是在原函数之前要新增的功能代码

这是在原函数之前要新增的功能代码

my name is COF

这是在原函数之后要新增的功能代码

这是在原函数之后要新增的功能代码

**★with上下文管理器**

with语句是一种用于简化资源获取和释放的语法结构，它可以确保在代码块执行完毕后，自动清理和释放相应资源（如文件、网络连接、数据库连接等）这种特性被称为“上下文管理器”

**with语句的基本语法如下：**

with **context\_manager** as variable:

# 执行代码块

# context\_manager是一个上下文管理器，它通常是一个实现了\_\_enter\_\_() 和\_\_exit\_\_()方法的对象

# variable是一个可选的变量，用于接收context\_manager的\_\_enter\_\_()方法返回值

在文件操作中，with语句非常常见，它可以确保文件在使用后被自动关闭，例如：

with open('test.txt', 'r') as file:  
 content = file.read()  
 print(content) #无需要手动关闭文件

**★自定义上下文管理器**

创建一个类，然后通过定义\_\_enter\_\_()和\_\_exit\_\_()方法来管理资源，例如：

class MyResource:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def \_\_enter\_\_(self):  
 print("MyResource.\_\_enter\_\_: Entering...")  
 return self  
  
 def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
 print("MyResource.\_\_exit\_\_: Exiting...")  
 # 这里可以处理资源释放，如关闭文件，关闭网络连接等  
 # 如果有异常发生，可以通过返回False来传播异常  
 return False  
  
with MyResource("hellow") as resource:  
 print(resource.name) # 对resource对象处理完成后，不需要手动关闭它，会自动调用\_\_exit\_\_()方法

结果：

MyResource.\_\_enter\_\_: Entering...

hellow

MyResource.\_\_exit\_\_: Exiting...

**★其他方法**

**#让类也能当函数使用**

class MyCalss:  
 def \_\_init\_\_(self, value):  
 self.value = value  
  
 def \_\_call\_\_(self, new\_value): # 类也能当函数使用  
 self.value = new\_value  
 print(f"new\_value is {self.value}")  
  
  
a = MyCalss("1111")  
a("2222")

结果：

new\_value is 2222

**★第9章、异常处理**

异常是python对象，表示一个错误，当程序发生异常时，需要捕获处理它，否则程序会终止执行。

python中的各种异常类都继承自BaseException这个基类。

BaseException派生出了4个之类：

用户中断执行时异常（keyboardinterrupt）

python解释器退出异常（systemexit）

内置及非系统退出异常（exception） # 遇到最多的就是exception类异常

生成器退出异常（generatorexit）

常见的基于Exception类的异常：

|  |  |
| --- | --- |
| NameError | 属于编译时异常，产生原因：使用了未定义的变量 |
| IndexError | 属于运行时异常，产生原因：列表/元组等访问越界，索引越界 |
| AttributeError | 属于运行时异常，产生原因：访问某个对象的不存在的属性 |
| FileNotFoundError | 属于运行时异常，产生原因：打开文件时，目标文件不存在 |
| ZeroDivisionError | 属于运行时异常，产生原因：算术运算时，除数为0了 |
| AssertionError | 属于运行时异常，产生原因：由 assert 语句抛出的 |

**①捕获异常可用try, except语句**

try:  
 #要执行的语句  
except xx异常类: #可以不匹配具体的错误类  
 #要执行的语句  
except yy异常类 as 数据:  
 #要执行的语句  
else:  
 #如果没有异常发生时 执行的语句  
finally:  
 #不管有无异常最终都要执行的语句

**②触发异常**

可以使用raise语句 自己 触发异常

def myFunc(varx):  
 if varx < 10:  
 raise Exception("提示信息", varx)  
 # 触发异常后，后面的代码不会再执行  
  
# 使用  
try:  
 myFunc(n)  
except Exception as err:  
 # 要执行的语句，err就是抛出的varx

自定义异常类，创建一个新类，继承自Exception类

class MyError(Exception): #创建自定义类，继承自Exception  
 def \_\_init\_\_(self, arg):  
 self.arg = arg  
 # 其他语句  
  
# 捕获异常  
try:  
 raise MyError("xx") # 抛出异常  
except MyError as err:  
 # to do

**★使用assert语句抛出异常**

assert语句又称为断言语句，格式为如下：

assert 表达式  
assert 表达式, "异常信息"

表达式的结果为False时，触发AssertionError异常；值为True时，不做任何操作。

示例：

def func\_xx(num1, num2):  
 assert num2 != 0, "func\_xx: 分母不能为0" #当assert后面的表达式返回True时，才继续往下执行  
 return num1 // num2  
  
try:  
 func\_xx(3, 0)  
except AssertionError as err:  
 print(err)

结果：

func\_xx: 分母不能为0

**★第10章、文件操作**

使用内置函数open()打开一个文件，返回file对象，再调用此对象的相关方法进行读写操作

**★文件打开模式**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 模式 | 文本 | 二进制 | 指针位置 | 是否覆盖 | 含义 |
| 只读 | r | rb | 开头 | 否 | 只读；若文件不存在则**报错** |
| 只写 | w | wb | 开头 | 是 | 全覆盖；打开文件时清空文件；不存在则新建 |
| 只写 | a | ab | 末尾 | 否 | 追加；不存在则新建 |
| 读写 | w+ | wb+ | 开头 | 是 | 读写，打开文件时就清空文件；若文件不存在则创建新的文件 |
| 读写 | r+ | rb+ | 开头 | 否 | 读写，写时从头插入，不覆盖；若文件不存在则**报错** |
| 读写 | a+ | ab+ | 末尾 | 否 | 读写，写时追加；若文件不存在则新建 |

例：

fileobj = open("test.txt", "r")  
fileobj.close() # 关闭文件

**★处理打开文件时的异常**

try:  
 fileobj = open("test.txt", "r")  
 # to do

fileobj.close()  
except Exception as err:  
 print(err)

也可直接使用with去打开文件，就不用手动关闭文件了

with open('test.txt', 'r') as fileobj:  
 # to do

**★打开文件时指定字符集编码**

打开文本文件时默认使用**系统本地语言**的编码，最好给open()函数传入encoding参数

例如，读取GBK编码的文件：（操作文本文件一定要指定编码）

fileobj = open('gbk.txt', 'r', encoding='gbk')

python3支持的常用字符集编码：

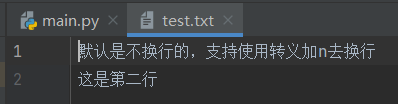
|  |  |
| --- | --- |
| utf-8 | Unicdoe，默认无BOM |
| utf-16 | Unicdoe，默认小端字节序，有BOM，支持UTF16代理对 |
| gbk | 中文-大陆，默认无BOM |
| euc-jp | 日语 |
| shift-jis | 日语 |
| euc-kr | 韩语 |
| big5 | 中文-台湾 |
| latin-1 | iso-8859-1 |

**★文本文件读写操作**

**.write()**方法写入文件，打开文件默认是跟随系统本地语言编码，写入文件也是，得在打开文件时指定文本编码；写入内容时，默认末尾不带换行符，可自己加\n

# coding=utf-8  
with open("test.txt","w+",encoding="utf8") as fileobj:  
 fileobj.write("默认是不换行的")  
 fileobj.write("，支持使用转义加n去换行\n这是第二行")

效果：



**.flush()** 方法刷新文件内部缓冲，把内部缓冲区的数据立刻写入文件

**.read(n)** 读方法，一次读n个字符（换行也算一个字符）

# coding=utf-8  
with open("test.txt","r+",encoding="utf8") as fileobj:  
 stx=fileobj.read(10)  
 print(stx)

**.readline()** 一次读一行，返回的字符串是带有换行符的

**.readlines()** 读取所有行，返回的是由每一行组成的list（带有换行符）

**★二进制文件操作→struct模块**

在打开文件时，操作符添加b就行，读写仍然使用.read()和.write()方法，读取的二进制内容是存储在bytes里，可以使用struct模块去解析成对应的数据类型

**★打开文本文件与打开二进制文件的区别：**

**打开文本文件时**：遇到EOF(0x1A)则认为文件结束了，不管文件里是\r\n还是\n换行，读取到string里只有\n，写入时也会转换\n为相应的系统换行符

fp.read()返回的是string类型

**打开二进制文件**则会读取到文件结尾，不转换\r\n等字符，且不可指定encoding="编码"

fp.read()返回的是bytes类型

**struct模块**中最重要的三个函数是pack(), unpack(), calcsize()

struct.pack('fmt', v1, v2, ...) #按照给定的格式(fmt)，把数据封装成bytes(实际上是类似于c结构体的字节流)

struct.unpack('fmt', bytesxx) #按照给定的格式(fmt)解析bytes，返回解析后的tuple

struct.calcsize('fmt') #计算给定的格式(fmt)占用多少字节的内存

struct中支持的格式

|  |  |  |  |
| --- | --- | --- | --- |
| format | 对应的C类型 | python类型 | 字节数 |
| c | char | string of length 1 | 1 |
| s | char[] | string | 1 |
| b | signed char | int | 1 |
| B | unsigned char | int | 1 |
| h | short | int | 2 |
| H | unsigned short | int | 2 |
| i l | int long | int | 4 |
| I L | unsiged int/long | int | 4 |
| q | long long | ~~long~~ int | 8 |
| Q | unsigned long long | ~~long~~ int | 8 |
| f | float | float | 4 |
| d | double | float | 8 |
| ? | \_Bool | bool | 1 |

**format指定原始数据的字节序：**

字节序默认跟随系统，x86\_64为小端字节序

|  |  |  |
| --- | --- | --- |
| 符号 | byte order | 字节对齐方式 |
| @ | native | 凑够4个字节 |
| = | native | 按原字节数 |
| < | little-endian | 按原字节数，小端字节序 |
| > | big-endian | 按原字节数，大端字节序，逻辑上是啥顺序就是啥 |
| ! | network = big-endian | 按原字节数，网络字节序 |

使用方法是在format的第一个位置指示即可

struct解析二进制串示例：

import struct  
with open("test.txt","rb+") as fileobj:  
 strb=fileobj.read(10)  
 a,b,c=struct.unpack("<2ih",strb)  
 print(a,b,c)

<2ih 表示使用小端字节序，2个int赋值给a和b两个变量，1个short赋值给c

import struct  
with open("test.txt","rb+") as fileobj:  
 strb=fileobj.read(10)  
 a,b,c=struct.unpack("<ih4s",strb)

<ih4s 表示小端字节序，一个i赋值给a，一个h赋值给b，4s表示4字节的string赋值给c

★struct.unpack()返回的是tuple，如果只有一个元素，则左边要写a,=struct.unpack()，带有个逗号，否则a就是tuple的类型了

struct将数据转换成二进制串示例

a = 5  
b = 89.4  
strb = struct.pack("if", a, b)  
print(len(strb)) #结果是8

★python3在将str="xxx"转换成3s时，会报错：

struct.error: argument for 's' must be a bytes object

需要在传入的字符参数前加b

import struct  
a=b'hello'  
b=b'world!'  
bytes=struct.pack('5s6s',a,b)

或者把str转为对应编码的bytes

strxx.encode("utf8")

★struct.pack()返回的是bytes类型，不可变量

**★第11章、模块，包**

包是一个分层次的文件目录结构，它定义了一个由模块及子包组成的python应用环境

包是一种管理python模块命名空间的形式

包是一个文件夹，包文件夹下必须存在\_\_init\_\_.py文件，该文件可以为空

\_\_init\_\_.py文件用于标识当前文件夹是一个包，包下的其他\*.py文件为模块定义文件

如果要让其他程序导入此包下的所有模块，需要在\_\_init\_\_.py里使用\_\_all\_\_变量导出相应的模块，\_\_init\_\_.py文件内容如下：（示例中的cof，lee为2个模块名称，即对应包下的cof.py及lee.py这2个模块文件）

\_\_all\_\_ = ["cof",  
 "lee"]

**★导入模块**

import 包名1.子包名.模块名  
# 使用模块里的变量/函数时得使用完整名称  
包名1.子包名.模块名.funxx()

from 包名1.子包名 import 模块名x  
# 这时可直接使用 模块加变量/函数的方式  
模块名x.funxx()

from .模块名x  
# .相对路径导入，表示 模块名x 在本模块同目录下， ..2个点表示在父目录下的  
模块名x.funxx()

from 包名x import \*  
#这时，只会导入 包名x 下的\_\_init\_\_.py里的 \_\_all\_\_=[这里列出的函数及变量]

一个模块被另一程序第一次导入时，其主程序将运行，若不想让它运行，可做个判断：

if \_\_name\_\_=="\_\_main\_\_":  
 #被导入模块的主程序写这里，只有它自己执行时才运行这里的代码，只有直接执行此模块脚本时才会运行这里的代码  
else:  
 #来自另一程序的调用，当其他脚本import此模块时，此模块的名称就是此模块的脚本文件名（不含.py）后缀

import sys as xx #将导入的模块名 sys 设置别名为 xx ，下文即可使用xx.func()  
print(xx.path)

**示例：**

我们在test\_module.py文件里写如下代码：

#!/usr/bin/env python3  
# coding=utf8  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print(f"直接执行了此脚本，\_\_name\_\_为 {\_\_name\_\_}")  
else:  
 print(f"import了此脚本，\_\_name\_\_为 {\_\_name\_\_}")

然后在另一脚本test.py里import引入此模块：

import test\_module  
  
print("hellow")

运行test.py的结果：

import了此脚本，\_\_name\_\_为 test\_module

hellow

★一个模块被导入时，解释器默认会将它编译成可执行的字节码.pyc

在模块所处包下生成一个名为\_\_pycache\_\_的目录，里面就是各模块的字节码文件

linux下可设置环境变量不生成字节码文件：

export PYTHONDONTWRITEBYTECODE=1

import sys  
sys.dont\_write\_bytecode = True #指定在导入时不生成字节码

**★python解释器查找模块顺序**

python解释器会去哪里找要导入的模块呢？

1，首先查找sys.modules字典里的模块（这是以前导入的所有模块的缓存）

2，然后匹配内置的模块sys.builtin\_module\_names

3，最后根据sys.path路径下去找（包含有程序当前路径；pip安装的包在"site-packages"目录下）

都找不到就报错ModuleNotFoundError

# python2.7 -c "import sys; print sys.path"  
['', #程序当前路径  
'/usr/lib64/python27.zip',   
'/usr/lib64/python2.7',   
'/usr/lib64/python2.7/plat-linux2',   
'/usr/lib64/python2.7/lib-tk',   
'/usr/lib64/python2.7/lib-old',   
'/usr/lib64/python2.7/lib-dynload',   
'/usr/lib64/python2.7/site-packages',   
'/usr/lib/python2.7/site-packages']  
  
# python3 -c "import sys; print(sys.path)"  
['', #程序当前路径  
'/usr/local/python3/lib/python37.zip',   
'/usr/local/python3/lib/python3.7',   
'/usr/local/python3/lib/python3.7/lib-dynload',   
'/usr/local/python3/lib/python3.7/site-packages']

★判断当前环境是否有某个模块：

如果import的模块找不到，就报ModuleNotFoundError异常

try:  
 import 模块名  
except Exception as e:  
 print(e)

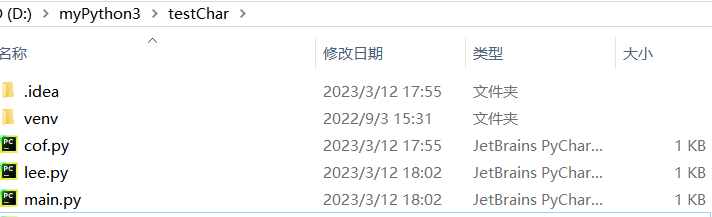
如果没有指定的模块，则结果如下：



**★打包源代码**

有时我们写好了自己的一些模块或代码包，需要像网上下载的xxx-pkg.tar.gz模块安装包一样打包我们的源代码并执行setup.py去安装我们的代码模块，可以使用distutils模块里的setup函数去打包

例如，我们写的代码有cof.py及lee.py两个模块，main.py只是用于测试的主函数



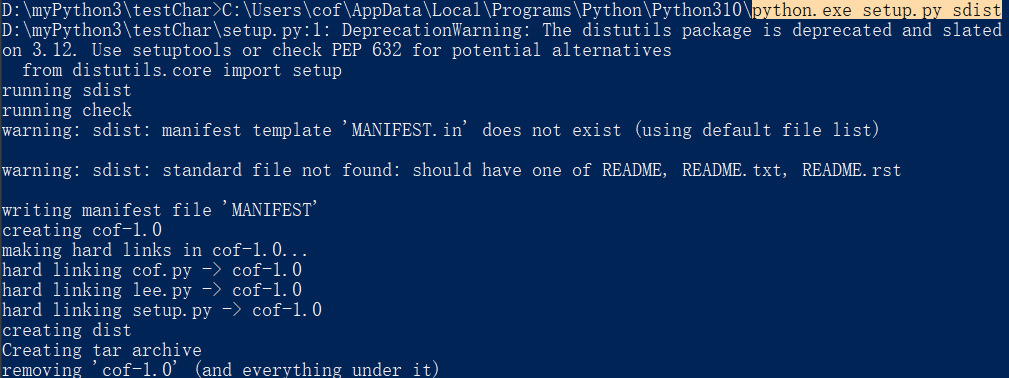
现在需要把cof.py及lee.py打包成一个安装包

在我们的源程序目录下再创建一个setup.py文件，内容如下：

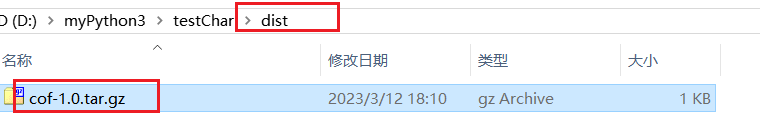
from distutils.core import setup  
setup(name='cof',  
 version="1.0",  
 py\_modules=['cof', 'lee'], # 要打包的模块名称  
 description="描述信息",  
 url="https://cof-lee.com/cof/home-page",  
 author="cof-lee",  
 author\_email="cof-lee@cof-lee.com"  
 )

最后在源程序目录下打开cmd命令行，使用python去执行此setup.py，命令如下：

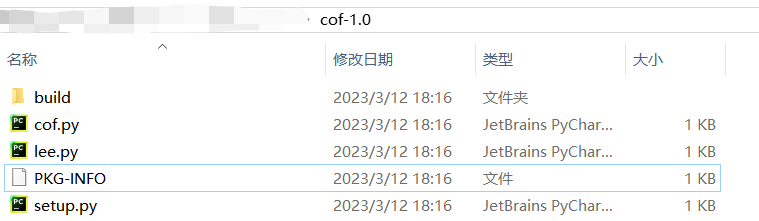
>C:\Users\cof\AppData\Local\Programs\Python\Python310\python.exe setup.py sdist



执行之后，在源程序目录下生成一个名为dist的子目录，里面有生成的安装包

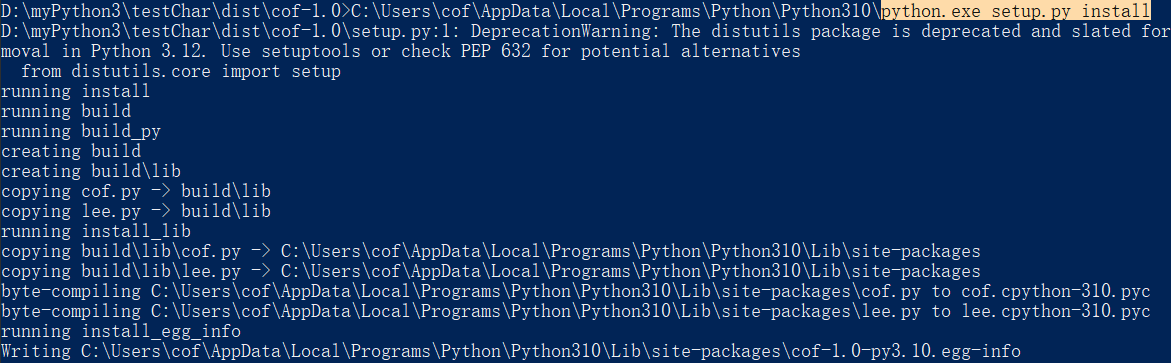


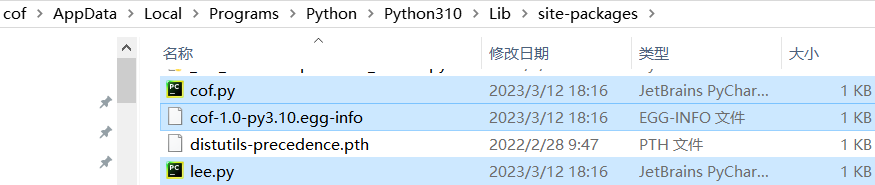
将安装包复制到其他需要使用这些程序模块的地方，再解压此安装包，



执行里面的setup.py文件进行安装，命令如下：

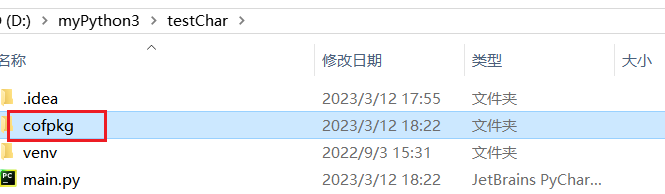
> python3 setup.py install #将模块文件安装到本python3环境的site-packages路径下





**打包模块包：**

现在需要把源程序目录下的cofpkg模块包（目录）打包成一个安装包

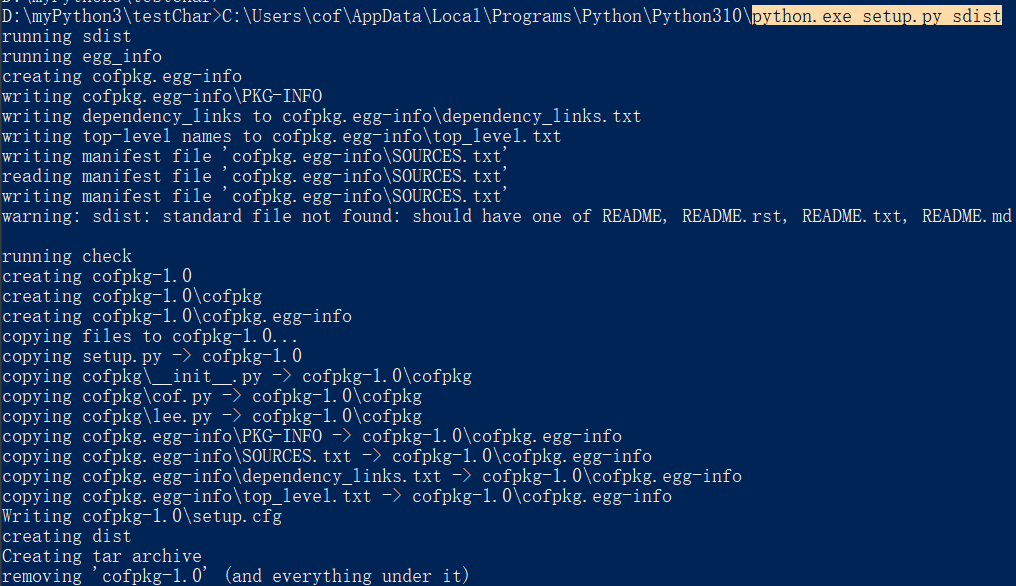


在我们的源程序目录下再创建一个setup.py文件，内容如下：

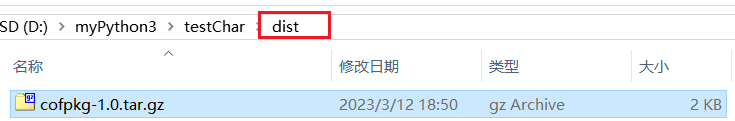
from setuptools import setup  
setup(name='cofpkg',  
 version="1.0",  
 packages=["cofpkg"], # 要打包的模块包名称（目录名称）  
 description="描述信息",  
 url="https://cof-lee.com/cof/home-page",  
 author="cof-lee",  
 author\_email="cof-lee@cof-lee.com",  
 python\_requires=">=3.6"  
 )

最后在源程序目录下打开cmd命令行，使用python去执行此setup.py，命令如下：

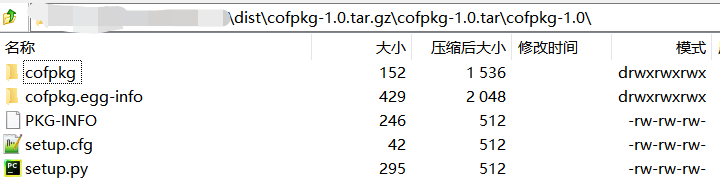
>C:\Users\cof\AppData\Local\Programs\Python\Python310\python.exe setup.py sdist



这样就成了，在源程序目录下生成一个名为dist的子目录，里面有生成的安装包



压缩包里的内容：



**★pyinstaller打包为.exe程序**

# pip3 install pyinstaller

cmd> cd 项目名称\venv\Scripts>

> pyinstaller.exe ..\..\test.py -F --hidden-import ..\..\rely\_module.py -i ..\..\test.ico -n test-001.exe

-F 将所有文件打包为一个单独的可执行文件

-D 将所有文件打包为一个目录，包含可执行文件和所有依赖的文件

--hidden-import 跟若干个依赖的程序文件

-n 指定生成的.exe文件名称，若未指定则默认同第1个主程序文件名（.py后缀改为.exe）

-w 指定程序运行时不显示命令行，仅显示图形界面（仅对Windows有效），同--windowed

-c 指定程序运行时显示命令行（仅对Windows有效），同--console

-i 指定exe程序的图标

-p 指定python导入模块的路径，同--path=

14919 INFO: checking EXE

14919 INFO: Building EXE because EXE-00.toc is non existent

14919 INFO: Building EXE from EXE-00.toc

14920 INFO: Copying bootloader EXE to D:\path\_to\项目目录\venv\Scripts\dist\test-001.exe

14933 INFO: Copying icon to EXE

14940 INFO: Copying 0 resources to EXE

14940 INFO: Embedding manifest in EXE

14948 INFO: Appending PKG archive to EXE

14958 INFO: Fixing EXE headers

15056 INFO: Building EXE from EXE-00.toc completed successfully.

在 项目名称\venv\Scripts> 目录下自动创建名为 dist的子目录，在此子目录下生成.exe可执行文件

**★将资源文件打包到同一个exe文件里**

使用--add-data参数，--add-data src\_dir;dest\_dir

尽量将所有资源文件放到同一子目录下（比如./builtin\_resource），再将此子目录打包进入exe文件里

> pyinstaller.exe ..\..\test.py -F --add-data ..\..\builtin\_resource;builtin\_resource -n test-xxx.exe

**★linux下打包为一个二进制程序**

# pyinstaller test.py -F --add-data builtin\_resource:builtin\_resource -n test-xxx-linux

**★打包后的程序执行路径与工作路径**

在pyinstaller 6.0.0以后，--add-data参数中的文件默认打包进\_internal文件夹，可以使用\_\_file\_\_定位打包后的资源

在程序里使用打包后的资源文件，需要使用\_\_file\_\_表示打包后的程序运行路径，如果使用os.getcwd则会提示找不到文件

builtin\_xxxfile\_path1 = os.path.join(os.path.dirname(\_\_file\_\_), "builtin\_resource", "fileName")

例：

import os  
  
print("当前程序执行时路径（文件）", \_\_file\_\_)  
print("当前程序执行时路径（目录）", os.path.dirname(\_\_file\_\_))  
print("当前程序工作目录",os.getcwd())

打包：

> pyinstaller.exe -F --add-data ..\..\builtin\_resource;builtin\_resource ..\..\test.py -n test-xxx.exe

> test-xxx.exe #运行test-xxx.exe，结果如下：

当前程序执行时路径（文件） C:\Users\cof\AppData\Local\Temp\\_MEI115162\test.py

当前程序执行时路径（目录） C:\Users\cof\AppData\Local\Temp\\_MEI115162

当前程序工作目录 D:\myPython3\python\_Project\_py39\.venv\Scripts\dist

可见工作目录与执行时路径是不一样的，如果需要使用打包后的资源文件，需要使用\_\_file\_\_获取**执行时路径**

**工作目录**（也叫静态路径）是程序文件（打包后的）所在目录，使用os.getcwd()获取

**执行时路径**是程序（打包后的）在执行时解压后的某个临时目录，使用os.path.dirname(\_\_file\_\_)获取

**★第12章、pip操作**

pip是一个Python包安装与管理工具

Python 2.x默认不带有pip命令

Python 3.5及以上版本都自带pip命令

**★pip操作命令**

pip安装的包默认在 site-packages子目录下

# pip freeze #查看用pip安装的包

# pip list #查看所有pip包

# pip list -o #同 --outdated 查看所有包及可更新的版本

#pip install 包名 #安装包

#pip install 包名==x.x.x #安装包时指定相应版本号

#pip install -r requireMents.txt #一次安装此txt文件里的所有包，一行一个软件包

#pip install -U 包名 #更新包，同--upgrade

#python -m pip install --upgrade pip #升级pip本身的版本

#pip download -d /下载目录 包名 #只下载包（依赖也一并下载），不安装

# pip install xxxx --proxy="http://x.x.x.x:3128" #使用代理

# pip install xxxx.whl #安装本地whl包

# pip uninstall 包名 #卸载依赖包

**#pip默认从以下地址下载软件包：**

https://pypi.org/simple #可以指定成其他的pip源地址

**①-i指定软件源**

#pip install 包名 -i https://xxx.com/simple --trusted-host xxx.com

**②写入配置文件**

**在当前用户家目录下创建.pip目录**，其下创建pip.conf文件

（windows为家目录下的pip目录下创建pip.ini文件）

# mkdir ~/.pip

# vi ~/.pip/pip.conf

[global]

index-url=https://xxx.com/simple

trusted-host=xxx.com

proxy=http://x.x.x.x:3128

[install]

trusted-host=xxx.com

**★国内的pip源地址：**

http://mirrors.aliyun.com/pypi/simple

https://pypi.tuna.tsinghua.edu.cn/simple

**★python2.7安装pip命令**

# curl https://bootstrap.pypa.io/pip/2.7/get-pip.py -o get-pip.py

# python2 get-pip.py

Installing collected packages: pip, setuptools, wheel

Successfully installed pip-20.3.4 setuptools-44.1.1 wheel-0.37.1

# pip2 list

DEPRECATION: Python 2.7 reached the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.7 is no longer maintained. pip 21.0 will drop support for Python 2.7 in January 2021. More details about Python 2 support in pip can be found at https://pip.pypa.io/en/latest/development/release-process/#python-2-support pip 21.0 will remove support for this functionality.

#或者手动下载软件包并解压到以下路径：

/usr/lib/python2.7/site-packages

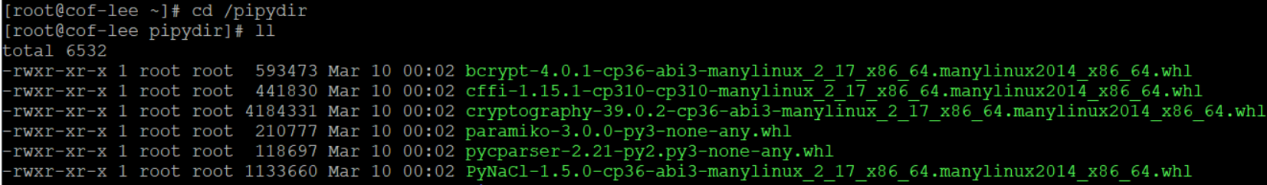
**★自制pip源服务**

在装有pip命令的服务器上执行以下命令，安装pip2pi

# pip3 install pip2pi #安装pip2pi，此工具用于创建pipy源的索引

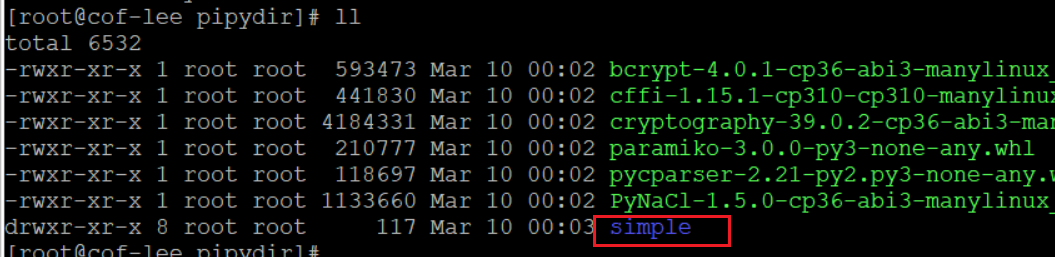
#然后将要用到的python模块包放到某目录下，如 /pypi ，所有人有rx权限

# mkdir /pypi



# dir2pi -S /pypi #创建pip源索引，成功后会在此目录下生成一个simple子目录

# dir2pi -s /pypi #-s小写s表示不创建硬链接，只创建软链接



#当目录里更新了其他模块包后，再次执行dir2pi -s /pypi命令即可

#最后创建一个web server（如nginx）监听7788端口，根目录为python模块包所在目录

server {

listen 7788;

location / {

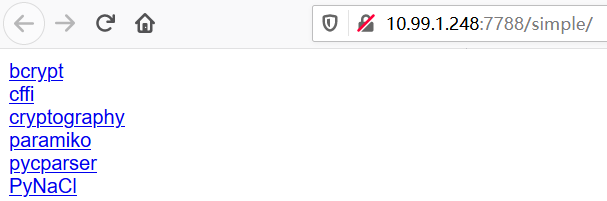
root /pypi;

}

}

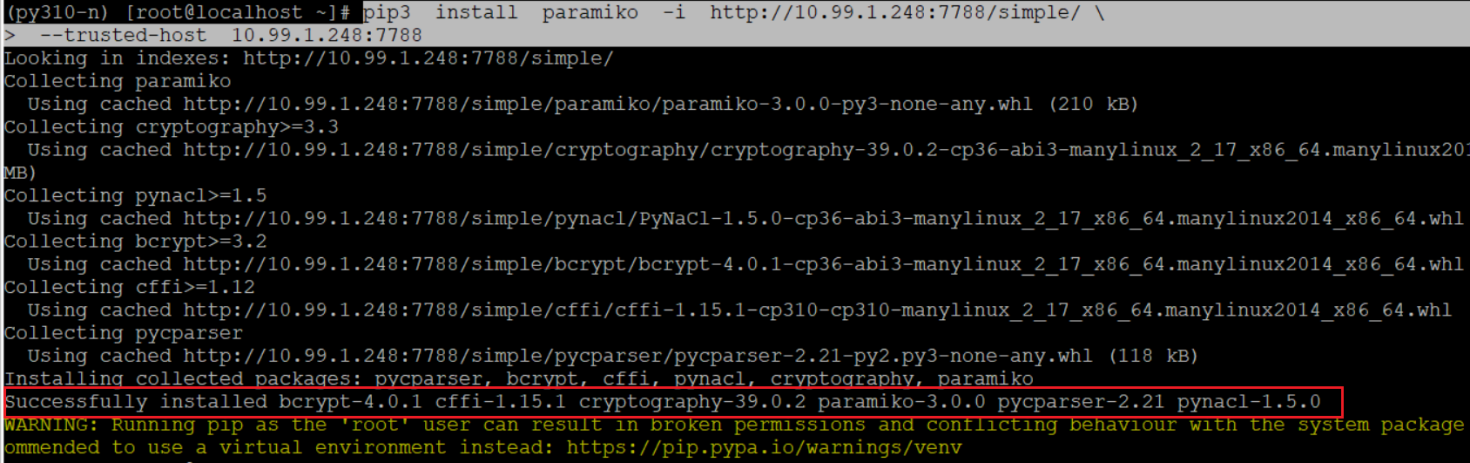
#测试，在浏览器打开以下地址（10.99.1.248为web server的ip）

http://10.99.1.248:7788/simple/



**★客户端使用此pipy源**

# pip3 install paramiko -i http://10.99.1.248:7788/simple/ --trusted-host 10.99.1.248



#上图可见成功了

**★python创建虚拟环境**

有时在同一台服务器上安装多个python包，可能会产生依赖冲突问题，可以通过创建虚拟环境来解决，python虚拟环境的原理是创建一个目录，在此目录下创建一个python软件链接，并激活一个shell环境变量，让系统优先从此虚拟环境的目录下去查找/安装依赖包

# mkdir -p ~/python3.9-venv

# python3.9 -m venv ~/python3.9-venv #会复制一份“干净”的python3.9解释器到此目录下

# source ~/python3.9-venv/bin/activate #激活指定的虚拟环境

# deactivate #退出当前虚拟环境

**★python2创建虚拟环境**

#首先安装pip2，再用pip2安装virtualenv软件包

# pip2 install virtualenv

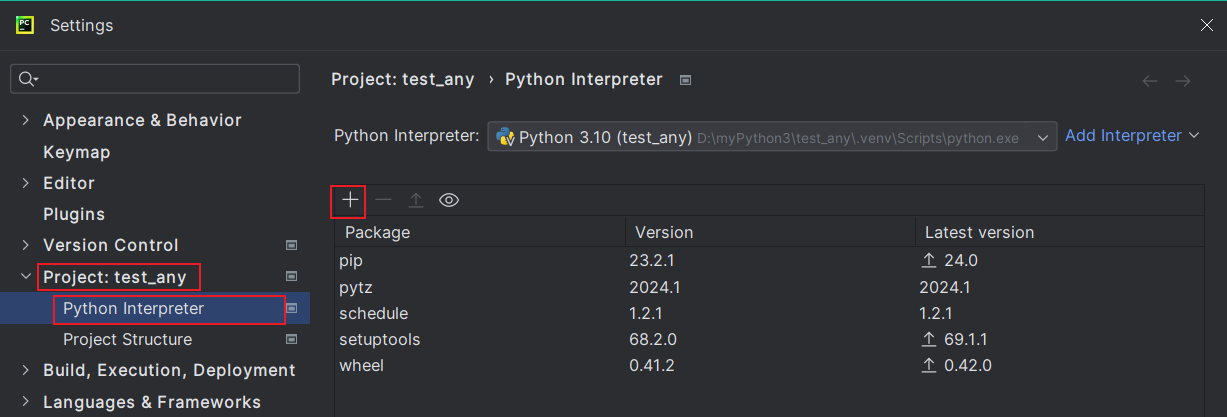
# mkdir -p ~/xxx-venv

# virtualenv -p python2 ~/xxx-venv #创建虚拟环境

# source ~/xxx-venv/bin/activate #激活指定的虚拟环境

**★pycharm安装模块**

File → Settings → Project: xxx →Python Interpreter → 右侧选择目标python解释器，再点击下面的+加号



**★第13章、anaconda**

conda是代码包及其依赖项和环境的管理工具，conda为Python项目而创造，但可适用于以下多种语言：

Python, R, Ruby, Lua, Scala, Java, JavaScript, C/C++, FORTRAN

anaconda是一个包含180+的科学包及其依赖项的发行版本，其包含的科学包包括：conda, numpy, scipy, ipython notebook等

conda包和环境管理器包含于Anaconda的所有版本当中

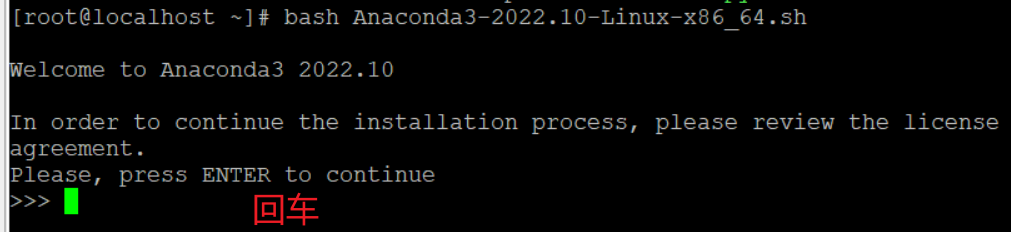
|  |  |
| --- | --- |
| pip | 是用于安装和管理软件包的 包管理器，维护多个环境难度较大 |
| virtualenv | 是用于创建一个独立的Python环境 的工具  不同的py程序需要不同的python版本及不同版本的依赖包，在共享主机时，无法在全局site-packages目录中安装依赖包。  virtualenv将会为它自己的安装目录创建一个环境，这并不与其他virtualenv环境共享库；同时也可以选择性地不连接已安装的全局库。 |
| conda | 结合了pip和virtualenv的功能，比较方便地在不同环境之间进行切换，环境管理较为简单 |
|  |  |

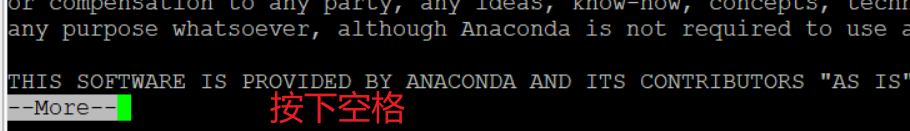
**★centos安装anaconda3**

anaconda下载地址： https://www.anaconda.com/products/distribution

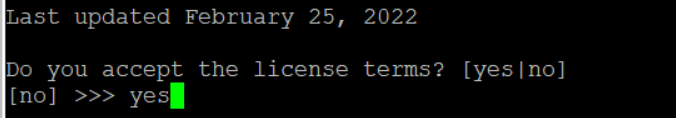
不区分linux版本，一般centos7，centos8，ubuntu2004，2204等版本都可以直接安装

# bash Anaconda3-2022.10-Linux-x86\_64.sh

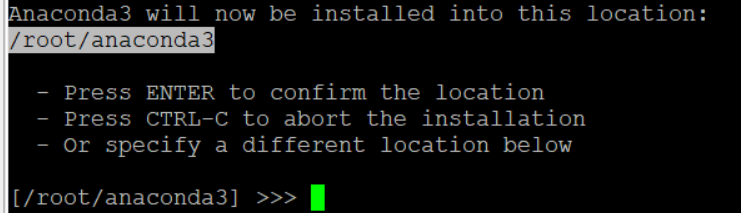




一直按空格翻页，直到出现下面提示： Do you accept the license terms? [yes|no]



输入yes

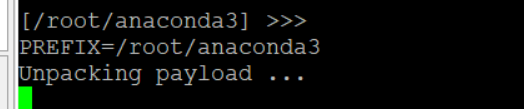


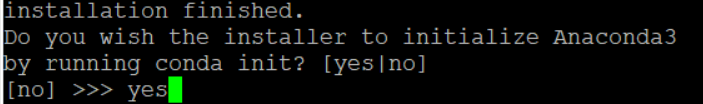
默认是安装在/root/anaconda3目录下，如果不是root用户，请输入当前用户能访问的目录

# bash Anaconda3-2022.10-Linux-x86\_64.sh -p /opt/anaconda3

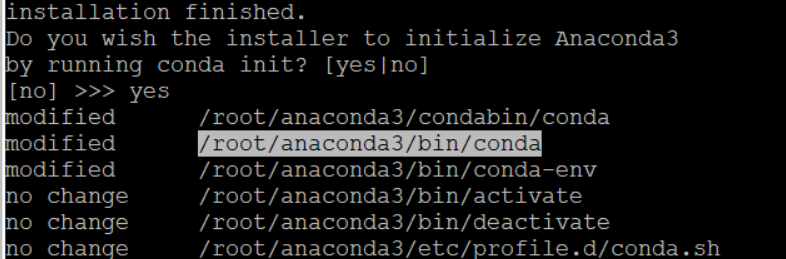
#这样可跳过软件安装路径确认阶段，直接安装到指定目录下

然后等待安装完成





#是否初始化conda的环境，直接输入yes，回车



conda命令在安装目录的bin子目录下，默认把初始命令添加到~/.bashrc文件里，由于我们没有重新登录系统，所以它没有重新加载bash配置，需要手动执行一下：

# source ~/.bashrc #之后就可以正常使用conda命令了

# >>> conda initialize >>>

# !! Contents within this block are managed by 'conda init' !!

\_\_conda\_setup="$('/root/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"

if [ $? -eq 0 ]; then

eval "$\_\_conda\_setup"

else

if [ -f "/root/anaconda3/etc/profile.d/conda.sh" ]; then

. "/root/anaconda3/etc/profile.d/conda.sh"

else

export PATH="/root/anaconda3/bin:$PATH"

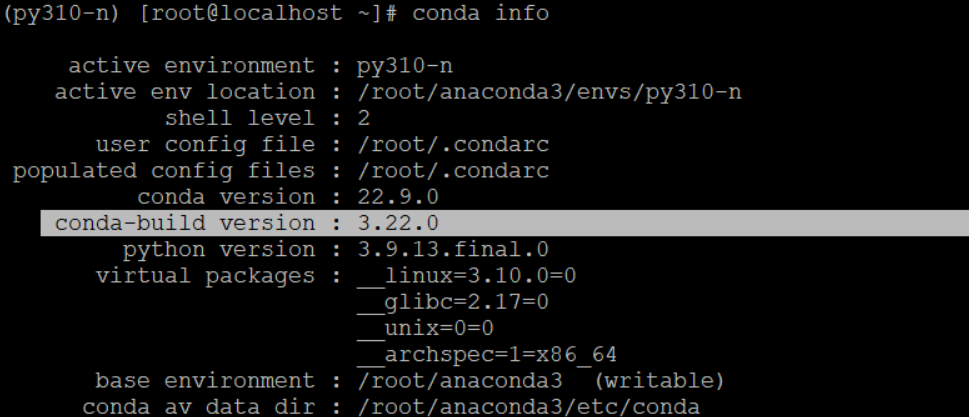
fi

fi

unset \_\_conda\_setup

# <<< conda initialize <<<

# conda info #查看conda信息



**★普通用户使用conda**

前提是安装anaconda时，安装在普通用户能访问的目录下，比如/opt/anaconda3

普通用户登陆后，需要执行一下 conda init 使conda的路径等系统环境变量信息写入当前用户的bashrc下

# /opt/anaconda3/bin/conda init bash #需要根据自己的实际情况修改bin之前的目录

# source ~/.bashrc #普通用户就可正常使用了

**★conda创建环境**

# conda env list #显示已创建的环境

# conda info --envs #显示已创建的环境



默认情况下，新创建的环境保存路径：

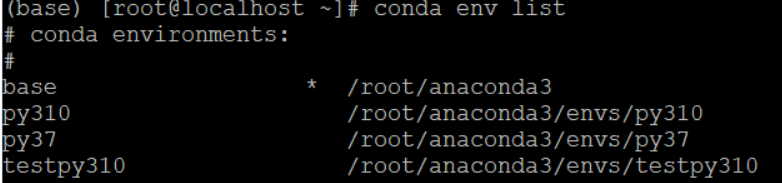
windows /Users/用户名/anaconda3/env 目录下

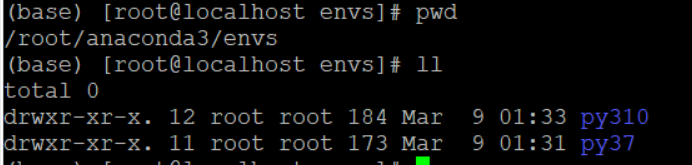
linux /path/to/anaconda3/envs/ 目录下

# conda create -n py310 python=3.10 paramiko #创建一个名为py310的环境，安装python 3.10版本及paramiko包，创建时如果本地没有相应版本的软件包，需要联网去下载

Proceed ([y]/n)? y #同意执行

如果创建环境后安装Python时没有指定Python的版本，那么将会安装与Anaconda版本相同的Python版本，即如果安装Anaconda第2版，则会自动安装Python 2.x；如果安装Anaconda第3版，则会自动安装Python 3.x

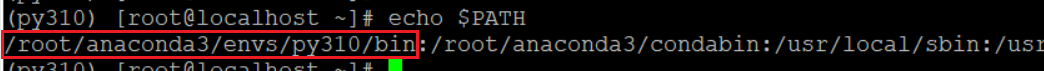




environment location: /root/anaconda3/envs/py310 创建完成后会提示环境所处路径

/root/anaconda3/envs/py310/bin/python3.10 #此环境的python解释器

/root/anaconda3/envs/py310/lib/python3.10/site-packages #此环境的第三方包路径

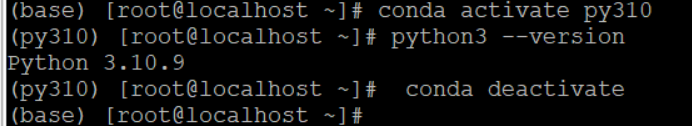


原理是conda把当前环境所处路径放到默认$PATH前面，查找python3解释器时优先找此路径下的python3程序

# source activate <env\_name> #切换环境

# conda activate base #切换环境

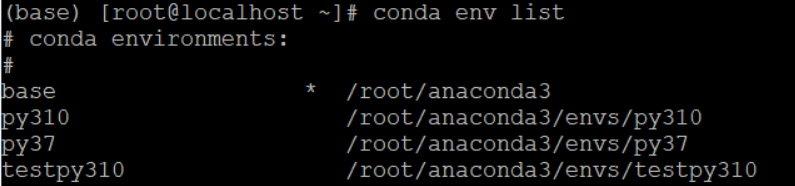
当成功切换环境之后，在命令提示符前面将显示 (env\_name) 字符串



# conda deactivate #退出当前环境

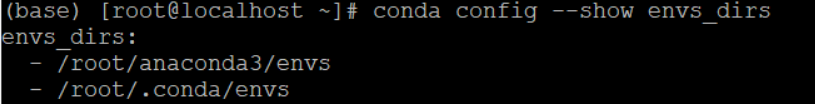
(env\_name)# conda install paramiko #在当前环境里安装其他依赖软件包

(env\_name)# conda install 软件包 -c 下载源 # -c指定下载源



# conda env remove -p /root/anaconda3/envs/testpy310 #删除指定环境

# conda config --show envs\_dirs #查看环境默认存储路径



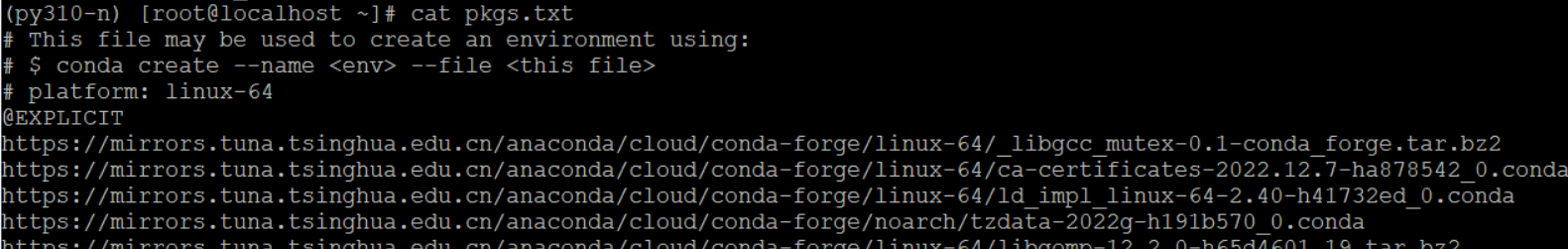
# conda config --add envs\_dirs /path/to/env #添加一个环境默认存储路径

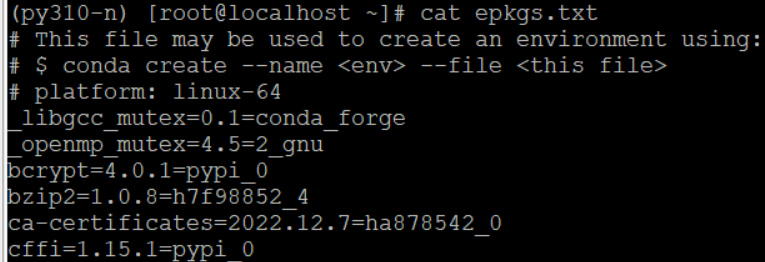
# conda config --remove envs\_dirs /path/to/env #删除一个环境默认存储路径

# conda list #查看当前环境安装的所有包

# conda list --explicit > pkgs.txt #输出当前环境所有包列表到pkgs.txt文件里

# conda list -e > epkgs.txt #输出当前环境所有包列表到epkgs.txt文件里





# conda create --name EnvXXX --file epkgs.txt #基于指定的包创建环境

**★conda设置软件源**

# conda config --show channels #查看下载时可使用的源

# conda config --show-sources #查看下载时可使用的源

# conda config --add channels $URL #添加源，如下三个：

conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/

conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/

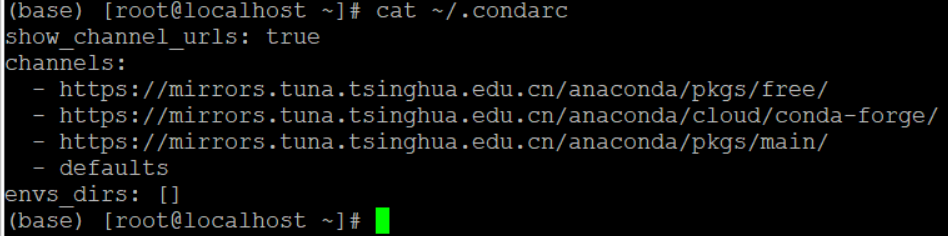
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge/

# conda config --set show\_channel\_urls yes #设置搜索时显示通道地址

# conda config --remove channels $URL #删除源

# conda config --remove-key channels #还原默认源

# conda config #生成默认配置文件，在~/.condarc



# conda search pytorch #从所有channels中查找软件

# conda install anaconda/cloud/conda-forge::pytorch==1.13.1 #从指定channel中安装包

**★conda设置代理**

# conda config --show #查看所有配置项

# vi .condarc #添加proxy\_servers配置

proxy\_servers:

http: http://10.99.1.1:10809

https: http://10.99.1.1:10809



pip的代理需要单独配置

# mkdir ~/.pip

# vi ~/.pip/pip.conf

[global]

index-url=https://xxx.com/simple

trusted-host=xxx.com

proxy=http://x.x.x.x:3128

[install]

trusted-host=xxx.com

**★第14章、python常用模块**

**★decimal精确计算模块**

import decimal  
nn=decimal.Decimal(3.141592653589793)  
nn/=decimal.Decimal(2.2)  
print(nn)

Decimal数只能和Decimal数进行算术操作

import decimal  
print(decimal.getcontext()) #获取上下文，查看精度等默认参数，默认prec=28  
decimal.getcontext().prec=60 #设置为60位精度

**★struct二进制数据处理模块**

请见第10章的“二进制文件操作→struct模块”

**★time时间日期模块**

时间是以秒为单位的浮点小数

时间戳表示从1970年1月1日0时到现在经过的秒数，传统4字节表示秒数只支持1970到2038年的时间

**①获取时间戳 time.time()**

import time  
print("当前时间戳",time.time())  
print(type(time.time()))



**②时间结构体，有9个字段**

import time  
localtime=time.localtime(time.time())  
print(localtime)  
print(type(localtime)) #结果如下

time.struct\_time(tm\_year=2022, tm\_mon=4, tm\_mday=5, tm\_hour=22, tm\_min=19, tm\_sec=7, tm\_wday=1, tm\_yday=95, tm\_isdst=0)  
<class 'time.struct\_time'>

**③格式化时间**

import time  
print(time.strftime("%Y-%m-%d %H:%M:%S",time.localtime()))  
print(time.strftime("%a %b %d %H:%M:%S %Y",time.localtime())) # %a表示周几  
print(time.strftime("%a, %d %b %Y %H:%M:%S %z",time.localtime()))

结果：

2024-11-24 12:17:05

Sun Nov 24 12:17:05 2024 #日期不足2位不以0填充

Sun, 24 Nov 2024 12:17:05 +0800

import time  
print(time.ctime(time.time()))  
print(time.asctime(time.localtime())) #同 time.ctime()

 #日期就一个 5（不足2位不以0填充）

**④时间字符串转为时间结构体**

import time  
timestr="Tue Apr 05 22:22:15 2022"  
t=time.strptime(timestr,"%a %b %d %H:%M:%S %Y")  
print(t)  
print(type(t))

**⑤时间结构体转为时间戳**

import time  
timestr="Tue Apr 05 22:22:15 2022"  
t=time.strptime(timestr,"%a %b %d %H:%M:%S %Y")  
tstamp=time.mktime(t) #时间结构体转为时间戳  
print(tstamp) #结果 1649168535.0

**⑥暂停线程的运行**

import time  
time.sleep(5) #单位为 秒

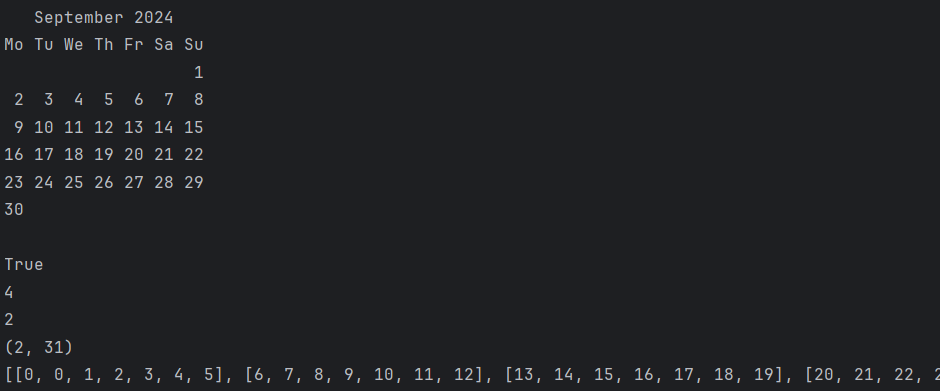
**★datetime时间日期模块**

import datetime  
  
# 注意，datetime是模块，datetime模块有一个名为datetime的类  
print(datetime.datetime.now()) # 返回当前日期及时间<datetime>（本地时间）如 2024-05-29 15:44:30.203461  
dt = datetime.datetime(2024, 5, 29, 15, 46, 0) # 构建一个<datatime>对象，2024-05-29 15:47:32.460047  
print(dt.timestamp()) # 返回时间戳<float>，单位：秒，如 1716968760.0  
ts = 1716968760.0  
print(datetime.datetime.fromtimestamp(ts)) # 根据时间戳构建<datetime>  
time\_str = "2024-05-29 15:52:00"  
print(datetime.datetime.strptime(time\_str, "%Y-%m-%d %H:%M:%S")) # 根据时间字符串构建<datetime>  
dt\_now = datetime.datetime.now()  
print(dt\_now.strftime("%Y-%m-%d %H:%M:%S")) # 把时间对象<datetime>转为时间字符串  
new\_date = dt\_now + datetime.timedelta(days=2, hours=3) # 当前时间加上2天3小时后，得到新的时间  
print(new\_date)  
# 前面获取的时间是没有时区的，即时区属性tzinfo=None  
# 下面获取UTC时间，再转为相应时间的本地时间  
date\_utc = datetime.datetime.utcnow().replace(tzinfo=datetime.timezone.utc) # 获取UTC时间，需要强制设置时区为utc  
print("UTC时间", date\_utc)  
date\_shanghai = date\_utc.astimezone(datetime.timezone(datetime.timedelta(hours=8))) # 设置时间为+8时区  
print("东8区时间", date\_shanghai)

**★calendar日历模块**

import calendar  
  
calendar.setfirstweekday(0) # 指定每周的起始日期码，默认为0（周一），指定为6时表示超始为周日  
print(calendar.month(2024, 9, w=0, l=0)) # 输出指定年份，指定月份的日历，一个月的，w表示日期之间间隔宽度，l表示行距  
print(calendar.isleap(2024)) # 判断指定年份是否为闰年，是则返回True  
print(calendar.leapdays(2011, 2025)) # 返回指定的2个年份之间的闰年总数  
print(calendar.weekday(2024, 5, 29)) # 返回指定日期是星期几，0表示星期一，6表示星期日  
print(calendar.monthrange(2024, 5)) # 计算指定年份指定月份有几天，返回一个tuple，元组第1个元素为该月第一天是星期几（0表示星期一）第2个元素为该月的天数  
print(calendar.monthcalendar(2024, 5)) # 返回指定年份指定月份的日历情况，按周输出list（没有日期的那天以0填充）

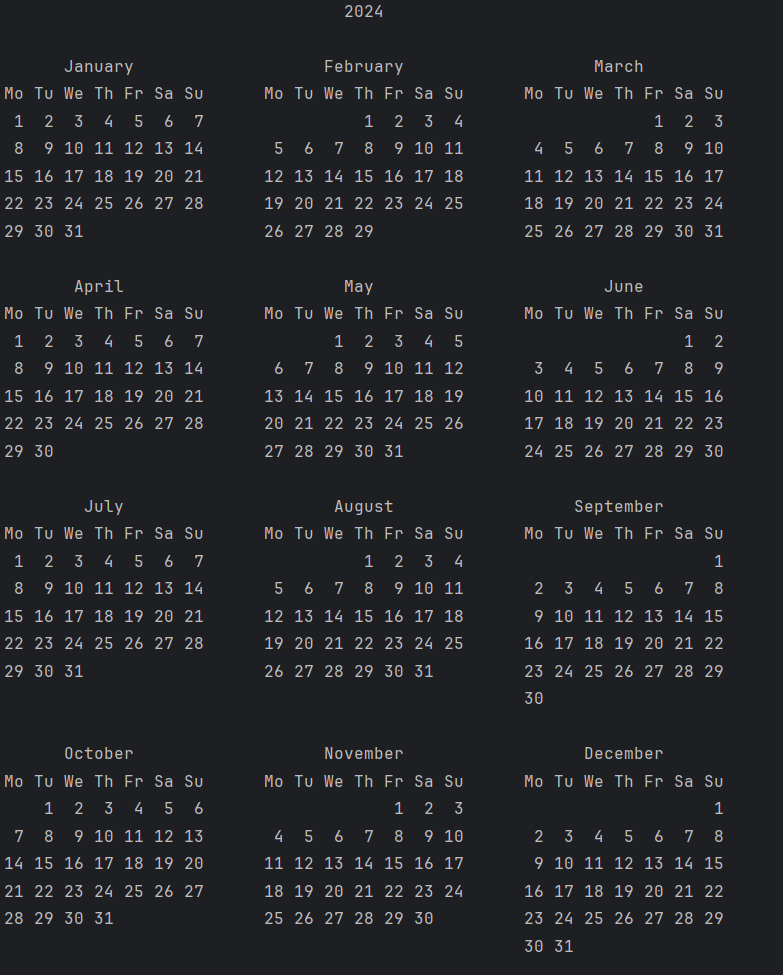
结果：



打印一年的日历（12个月）

import calendar  
  
calendar.setfirstweekday(0) # 指定每周的起始日期码，默认为0（周一），指定为6时表示超始为周日  
print(calendar.calendar(2024, w=2, l=1, c=6)) # 输出指定年份的日历，12个月的，c表示每个月之间的间隔，默认为6

结果：



**★sys模块（py解释器相关）**

import sys  
  
print(sys.argv) # sys.argv为标准输入参数列表，第0个元素为程序名本身  
  
print(sys.getdefaultencoding()) # 查看python默认字符编码  
print(sys.version\_info) # 返回python版本信息，可用sys.version\_info[0],[1],[2]获取相应版本号  
# sys.version\_info(major=3, minor=10, micro=2, releaselevel='final', serial=0)  
print(sys.platform) # win32, linux2  
print(sys.getwindowsversion()) # (major=10,minor=0,build=17763,platform=2,service\_pack='')  
my\_module\_path = '/path/to/project'  
sys.path.append(my\_module\_path) # 将项目目录添加到python模块搜索路径中（搜索列表的末尾）  
sys.path.insert(0, my\_module\_path) # 将项目目录添加到python模块搜索路径中（搜索列表的第0个元素）

**★os操作系统接口模块**

import os  
cdir = os.getcwd() # 返回当前工作目录str  
os.chdir("D:\\") # 切换当前目录  
dir1 = "D:\\tmp\\dir1"  
os.mkdir(dir1) #创建目录，若目录已存在则报错

os.makedirs(dir1, exist\_ok=True) #创建目录，若目录已存在也不报错

ret = os.system("dir") # 执行系统命令，成功执行返回0，失败返回非0数值，输出结果直接在命令行打印出来

import os  
  
current\_dir = os.getcwd()  
path1 = os.path.join(current\_dir, "child\_dir", "file.txt")  
print("拼接路径:", path1)

结果：

拼接路径: D:\myPython3\test\_any\child\_dir\file.txt

批量重命名文件：

import os  
  
file\_dir = r'D:\test'  
file\_name\_list = os.listdir(file\_dir)  
for file\_name in file\_name\_list:  
 print(file\_name)  
 old\_file\_full\_name = os.path.join(file\_dir, file\_name)  
 new\_file\_full\_name = os.path.join(file\_dir, "new-" + file\_name)  
 os.rename(old\_file\_full\_name, new\_file\_full\_name) # 重命名文件

结果：

test1.txt

test2.txt



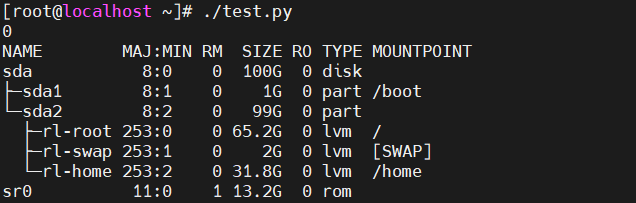
**★subprocess模块**

用于在目标系统上创建子进程执行命令，并返回状态码及标准输出/标准错误的内容

os.system("cmd")只能返回状态码，无法保存执行命令的结果到某个返回参数，而使用subprocess.run()就可以获得执行结果内容

import subprocess  
  
command = "lsblk"  
result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)  
print(result.returncode) # 执行命令返回的状态码  
print(result.stdout.decode()) # 执行命令返回的结果内容

结果：（第一行的0为执行命令返回的状态码，0表示执行成功）



**★math模块**

import math  
x = 2  
math.sqrt(x) # 返回x的平方根，float  
math.pi # 常量 pi=3.141592653589793

math.e # 常量 e=2.718281828459045  
math.log(x) # 返回log x的值，以e为底  
math.log10(x) # 返回log10 x的值，以10为底  
math.sin(x) # 返回sin x的值，x为弧度，360度为2π弧度  
math.cos(x)  
math.tan(x)

math.pow(x,y) #返回x的y次幂，同 x \*\* y

**★random模块**

import random  
random.random() # 随机生成一个[0,1)范围内的实数，float  
random.uniform(0, 5) # 随机生成一个[0,5]范围内的实数，float  
random.randint(0, 8) # 随机生成一个[0,8]范围内的整数，int  
random.randrange(1, 100, 2) # 从[1到100]随机选一个整数，step默认为1，这里指定为2，即从1，3，5，7...选一个整数

**★生成随机字符串**

import string  
import random  
  
def generate\_random\_text(size):  
 return ''.join(  
 random.SystemRandom().choice(string.ascii\_lowercase + string.ascii\_uppercase + string.digits + string.punctuation) for \_ in range(size))  
  
print(generate\_random\_text(50))

string.ascii\_letters 包含大小写字母

**★glob文件通配符模块**

import glob  
print(glob.glob("\*.py")) # 返回由 当前目录下匹配上的文件名 组成的list

**★pickle序列化模块**

序列化是指将内存中的对象 存储到文件中（二进制文件）

反序列化将文件内容读出并生成对象（恢复成对象）

import pickle  
datax = {"k1": "v1", "k2": "v2", "k3": "v3"}  
datay = ["fdas", "fkldsj", 1323]  
with open("pk.dat", "wb") as fp:  
 pickle.dump(datax, fp) # 一次存入一个对象，对象类型可不相同，但读出时得按相应顺序读出  
 pickle.dump(datay, fp)

import pickle  
with open("pk.dat", "rb") as fp:  
 datax = pickle.load(fp) # 一次读取一个对象，对象类型可不相同  
 datay = pickle.load(fp)

**★re正则匹配模块**

RE（Regular Expression）正则表达式是 用来搜索目标字符串A 中的某些字符串s的，以及查询目标字符串s在A中的位置

正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为"元字符"）组成的文字模式

正则表达式用//包含起来（正则表达式写在//双斜杠之间）如 /xxxx/ 里面的xxxx为正则表达式，（正则表达式本身不包含最外面那2个斜杠。在右边的/后面可带修饰符，如 /xxxx/igms

不过，在不同的文本编辑器或编程语言里，不是直接输入 /xxxx/igms 的，有的文本编辑器提供搜索框，可直接输入xxxx正则表达式，其他修饰符以“可选项”列出，需要时勾选相应修饰项即可

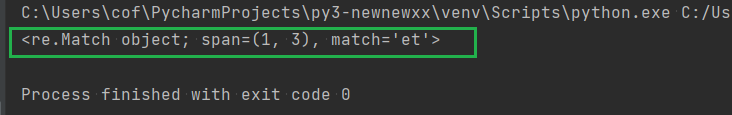
**★正则表达式的修饰符**

|  |  |  |
| --- | --- | --- |
| i | ignore，匹配时不区分大小写 | re.I |
| g | global，全局匹配，查找所有的匹配项 |  |
| m | multi line，多行匹配，使得边界符^和$匹配每一行的开头和结尾，而不是整个字符串 | re.M |
| s | 默认时 .点号 匹配除了换行符\n之外的所有字符，加上s修饰后，.点号包含换行符 | re.S |

re.search() #匹配整个字符串，直到找到一个匹配，只返回匹配上的第一个匹配对象<re.Match>

import re  
str = "net Net fldskajfl jfsd lfjasd lfj asdkfl sl x"  
ret = re.search(r"et", str, flags=re.I) #在str里搜索 et，返回一个匹配对象，如果匹配不到，返回None  
print(ret)

结果：



说明：

re.search(正则表达式, 要搜索的字符串, flags=修饰符) #只匹配第一个匹配上的字符

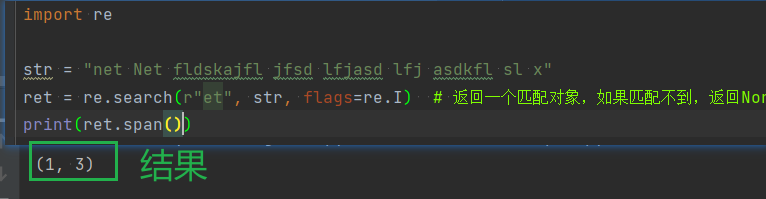
正则表达式 可用r" " 指定里面的所有字符都不识别为python转义字符

flags= 修饰符，可用 | 或运算连接多个修饰符，常用3个修饰符如下：

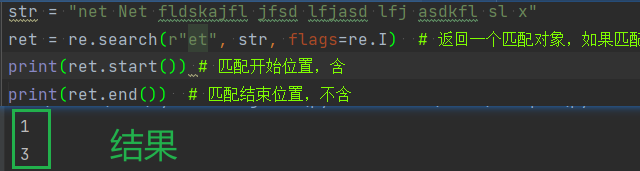
re.I表示不区分大小写，re.M表示多行匹配，re.S表示使**.**点号匹配换行符

ret.span() 为一个tuple，匹配的起始字符index及终止字符index，含起始，不含终止

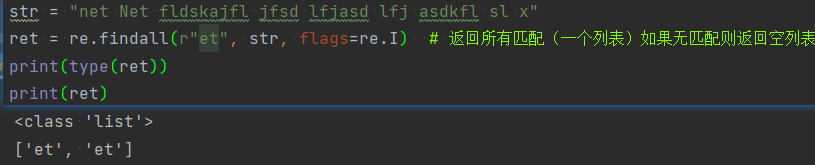
ret.group() 为匹配的一个群，一个匹配内容字符串



也可用ret.start()和ret.end()方法获取匹配到的起始位置及结束位置（含起始，不含结束）



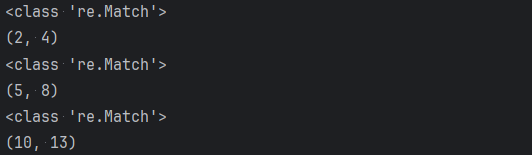
re.findall(正则表达式, 要搜索的字符串, flags=修饰符) #返回所有的匹配字符串<list>，元素为匹配的字符串



re.finditer(正则表达式, 要搜索的字符串, flags=修饰符) #返回所有的匹配的结果，元素为<re.Match>

import re  
  
str\_xx = "ab001000bc000"  
match\_pattern = r'(?:0)+'  
ret = re.finditer(match\_pattern, str\_xx, flags=re.I)  
for ret\_item in ret:  
 print(type(ret\_item))  
 print(ret\_item.span())

运行结果：



**★base64模块**

import base64  
  
def test\_base64():  
 b64\_str = 'aGVsbG8='  
 decode = base64.b64decode(b64\_str) # base64解密后返回 <bytes>  
 print(decode) # 结果为 b'hello'  
 bytes\_like = 'hello'.encode('utf8')  
 encode = base64.b64encode(bytes\_like) # base64加密内容必须为 <bytes> 或 <bytearray>  
 print(encode) # 返回 b'aGVsbG8='  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_base64()

**★json模块**

import json  
dictx = {  
 "k1": "v1",  
 "k2": "vvv2",  
 "k3": {"k3xx": "vdskl", "k3fyy": "fdksj"}  
}  
str\_json = json.dumps(dictx) # 将字典数据转为json字符串，默认是一行  
str2\_json = json.dumps(dictx, indent=4) # 将字典数据转为格式化的json字符串，缩进4空格  
newdict = json.loads(str\_json) # 将json字符串导入解析为字典数据  
with open("data.json", "w", encoding="utf8") as fp:  
 json.dump(dictx, fp, indent=4) # 将字典数据格式化为json字符串后写入到文件中  
with open("data.json", "r", encoding="utf8") as fp:  
 data = json.load(fp) # 读取文件中的json字符串后生成字典数据（文件中只可有一个json对象）  
 print(data)

import json  
  
str\_xx = r'{"name":"cof","age":18}'  
data = json.loads(str\_xx) # ★将str类型的json字符串转为dict字典类型的数据  
formatted\_str = json.dumps(data, indent=4, sort\_keys=True) # sort\_keys=True表示按键名排序输出  
print(formatted\_str)

结果：

{

"age": 18,

"name": "cof"

}

说明：读取字符串时使用json.loads()方法，读取文件时使用json.load()方法

**★getpass模块**

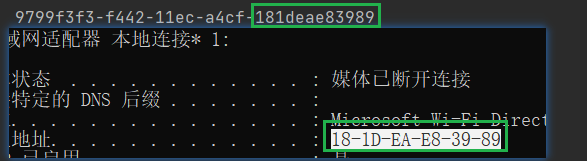
import getpass  
cusername = getpass.getuser() # 获取当前用户名  
passwd = getpass.getpass("请输入密码：") # 获取用户输入密码，不回显，该函数在IDLE/PyCharm里不生效

**★uuid模块**

import uuid  
print(uuid.uuid1()) # 返回uuid类，print则输出uuid字符串  
print(uuid.uuid4().\_\_str\_\_()) # 直接返回uuid字符串

uuid1使用到本服务器的一个网口mac地址，会泄露服务器的mac地址信息，不常用

uuid4使用时间加其他随便机数，常用



**★sqlite3模块**

import sqlite3  
  
sqlite\_conn = sqlite3.connect('test.db') # 连接数据库文件，若文件不存在则新建  
# 数据库所有数据存储在此文件中，默认数据库名称同文件名（不含.db后缀）  
sqlite\_cursor = sqlite\_conn.cursor() # 创建一个游标，用于执行sql语句  
  
# 查询是否有名为'tb\_test'的表  
sqlite\_cursor.execute(  
 'SELECT \* FROM sqlite\_master WHERE "type"="table" and "tbl\_name"="tb\_test";')  
result = sqlite\_cursor.fetchall() # fetchall()从结果中获取所有记录，返回一个list，元素为<tuple>（即查询到的结果）  
print("tables: ", result)  
if len(result) == 0: # 若未查询到有此表，则创建此表  
 sqlite\_cursor.execute("create table tb\_test (id varchar(36) NOT NULL PRIMARY KEY,name varchar(128),age int)")  
try:  
 sqlite\_cursor.execute("insert into tb\_test (id,name,age) values ('uuidxxxxxx','cof-lee',18)")  
except Exception as e:  
 print(type(e)) # 若在数据库中定义了唯一性约束，当试图插入一个重复的值时，会触发"sqlite3.IntegrityError"错误  
 print(e)  
 exit()  
sqlite\_cursor.execute("insert into tb\_test (id,name,age) values ('uuidxxxxxx2','tom',19)")  
sqlite\_cursor.execute("insert into tb\_test (id,name,age) values ('uuidxxxxxx3','ben',16)")  
print("cursor rowcount", sqlite\_cursor.rowcount) # 游标上一次执行时返回的结果，刚插入一条数据，所以显示1  
sqlite\_cursor.close()  
sqlite\_conn.commit() # 保存  
sqlite\_conn.close() # 关闭连接  
  
sqlite\_conn2 = sqlite3.connect('test.db') # 连接数据库（此时数据库文件已存在）  
sqlite\_cursor2 = sqlite\_conn2.cursor() # 创建一个游标，用于执行sql语句  
sqlite\_cursor2.execute("select \* from tb\_test limit 1000")  
result = sqlite\_cursor2.fetchall() # fetchall()从结果中获取所有记录，返回一个list，元素为<tuple>（即查询到的结果）  
print(result)  
  
sqlite\_cursor2.execute("select \* from tb\_test limit 1000")  
result2 = sqlite\_cursor2.fetchone() # fetchone()从结果中获取一条记录，返回一个<tuple>（即查询到的结果）  
print(result2)  
  
sqlite\_cursor2.execute("select \* from tb\_test limit 1000")  
result3 = sqlite\_cursor2.fetchmany(3) # fetchone()从结果中获取多条记录（3条）返回一个list，元素为<tuple>（即查询到的结果）  
print(result3)  
  
sqlite\_cursor2.close()  
sqlite\_conn2.close() # 关闭连接

**★sched模块**

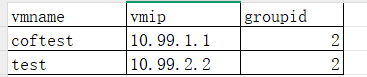
周期进行某任务，每隔一定时间就执行某任务

import time  
import sched  
  
  
def schedule\_func(index):  
 print(f'index: {index} do schedule\_func time {time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())}')  
 time.sleep(1)  
  
  
def loop\_monitor():  
 index = 0  
 while True:  
 sched1 = sched.scheduler(time.time, time.sleep) # 创建一个调度器  
 sched1.enter(3, 1, schedule\_func, (index,)) # 延迟3秒，优先级1，回调函数，参数  
 sched1.run() # 运行调度器，默认是blocking=True，阻塞模式，等时间到了才运行，运行回调函数后才继续  
 print(f'index: {index} 运行调度器回调函数之后的输出 {time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())}')  
 index += 1  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 loop\_monitor()

**★csv模块**

**读取csv文件：**

csv文件名 test.csv 内容如下：



import csv  
with open('test.csv', 'r') as file:  
 reader = csv.reader(file)  
 for row in reader:  
 print(type(row)) # <class 'list'>  
 print(row[0], row[1], row[2])

结果：

<class 'list'>

vmname vmip groupid

<class 'list'>

coftest 10.99.1.1 2

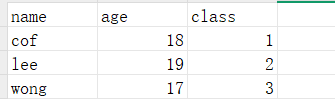
<class 'list'>

test 10.99.2.2 2

**写入文件：**

import csv  
data = [  
 ["name", "age", "class"],  
 ["cof", 18, 1],  
 ["lee", 19, 2]  
]  
with open('test2.csv', 'w', newline='') as file: # newline=''用来避免在写入时产生额外的空行  
 writer = csv.writer(file)  
 writer.writerows(data) # 写入多行数据  
 writer.writerow(["wong", 17, 3]) # 写入单行数据

结果：



**★第15章、常用非自带模块**

常用的非自带模块，需要使用pip安装

**★paramiko（SSH远程登录）**

# pip3 install paramiko

# 在pycharm里，打开工程所在目录，进入venv\Scripts\子目录下

按下Shift键同时点击鼠标右键，在此处打开PowerShell窗口

cmd> pip3 install paramiko

**★使用密码认证**

import paramiko  
import time  
  
hostname = "10.99.1.233"  
username = "root"  
password = "xxxxxx"  
port = 22  
timeout = 20 # 单位:秒  
  
client1 = paramiko.client.SSHClient()  
client1.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy()) # 允许连接不在know\_hosts文件里的主机  
try:  
 client1.connect(hostname=hostname, port=port, username=username, password=password, timeout=timeout)  
except paramiko.AuthenticationException as e:  
 print(f"Authentication Error: {e}")  
 exit()  
stdin, stdout, stderr = client1.exec\_command("ls -l") # 执行一次性命令  
# stdout.read() #返回bytes  
# stdout.readlines() #返回list,元素为str  
print(stderr.readlines()) # 返回list,元素为str  
  
for line in stdout.readlines():  
 print(line, end="") # 默认是每个元素都带有末尾换行符

**★使用密钥认证**

import paramiko  
import sys  
import time  
  
hostname = "10.99.1.233"  
username = "root"  
prikey\_file = "D:\\tmp\\id\_rsa2"  
port = 22  
timeout = 20 # 单位:秒  
  
print("## 请输入:要执行的命令，最后按Ctrl+D结束输入 ##")  
input\_list = sys.stdin.readlines() # 输入要执行的命令  
  
client1 = paramiko.SSHClient()  
client1.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy()) # 自动添加服务器的ssh密钥，允许连接不在know\_hosts文件里的主机  
try:  
 # prikey = paramiko.RSAKey(filename=prikey\_file)  
 prikey = paramiko.RSAKey.from\_private\_key\_file(prikey\_file) # 同上，创建一个rsa-key对象  
except FileNotFoundError as e:  
 print(f"File Not Found Error: {e}")  
 exit()  
try:  
 client1.connect(hostname=hostname, port=port, username=username, pkey=prikey, timeout=timeout)  
except paramiko.AuthenticationException as e:  
 print(f"Authentication Error: {e}")  
 exit()  
client1\_shell = client1.invoke\_shell() # 创建一个交互式shell  
  
index = 1  
for line in input\_list: # 解析输入  
 if line == "\n": # 如果是空行，则不执行此空命令  
 continue  
 print(f"执行命令{index} : {line.strip()}")  
 client1\_shell.send(line.encode('utf8')) # 交互式shell可多次执行命令而不断开连接  
 time.sleep(1)  
 index += 1  
 output = client1\_shell.recv(65535).decode('utf8')  
 print(f"输出结果:\n{output}")  
  
client1.close()

**★根据密钥文本字符串创建private\_key**

prikey\_content = '''-----BEGIN OPENSSH PRIVATE KEY-----  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
-----END OPENSSH PRIVATE KEY-----  
'''  
prikey\_obj = io.StringIO(prikey\_content)  
prikey2 = paramiko.RSAKey.from\_private\_key(prikey\_obj) # 创建一个rsa-key对象

**★sftp上传下载文件**

import paramiko  
import sys  
import time  
  
hostname = "10.99.1.233"  
username = "root"  
password = "xxxxxx"  
port = 22  
timeout = 20 # 单位:秒  
  
transport = paramiko.Transport((hostname, port)) # 创建SSH传输通道  
try:  
 transport.connect(username=username, password=password)  
except paramiko.AuthenticationException as e:  
 print(f"Authentication Error: {e}")  
 exit()  
sftp = paramiko.SFTPClient.from\_transport(transport) # 创建SFTP客户端  
  
# 下载远程文件到本地  
try:  
 remote\_file\_path = 'anaconda-ks.cfg'  
 local\_file\_path = 'D:\\tmp\\anaconda-ks.cfg'  
 sftp.get(remote\_file\_path, local\_file\_path)  
except FileNotFoundError as e:  
 print(f"File Not Found Error: {e}")  
  
# 上传本地文件到远端服务器  
try:  
 local\_file\_path = 'D:\\tmp\\anaconda-ks.cfg'  
 remote\_file\_path = 'anaconda-ks.cfg2'  
 sftp.put(local\_file\_path, remote\_file\_path)  
except FileNotFoundError as e:  
 print(f"File Not Found Error: {e}")  
  
sftp.close() # 关闭SFTP连接  
transport.close() # 关闭SSH传输通道

**★requests（HTTPS请求）**

# pip3 install requests

# 在pycharm里，打开工程所在目录，进入venv\Scripts\子目录下

按下Shift键同时点击鼠标右键，在此处打开PowerShell窗口

cmd> pip3 install requests

#使用

import requests  
  
response = requests.get("http://sysyear.top:4600")  
print(type(response.text)) # 类型为str  
response.encoding = "utf-8" # 指定返回数据的文本编码  
  
print(response.status\_code) # 返回http响应状态码  
print(response.headers) # 返回http头部信息，单行字典类型  
print(response.text) # 返回正文内容，多行文本

import json  
import requests  
from urllib3.exceptions import InsecureRequestWarning  
  
requests.packages.urllib3.disable\_warnings(InsecureRequestWarning) # 关闭“不安全的请求”告警提示

myurl = "https://10.99.1.69/test.html"  
  
myheaders = {  
 "Accept": "application/json",  
 "Content-Type": "application/json;charset=utf8"  
}  
  
mydata = json.dumps({  
 "username": "cof",  
 "password": "xxx"  
})  
  
response = requests.request("GET", myurl, headers=myheaders, data=mydata, verify=False, timeout=5)  
# verify=False表示不校验ssl证书，timeout=5指定超时时间，单位：秒，默认为10秒  
if response.status\_code == 200:  
 response.encoding = "utf-8" # 指定返回数据的文本编码  
 print(response.status\_code) # 输出请求的状态码  
 print(response.text) # 输出返回的内容  
 print(type(response.json())) # 将返回的json字符串内容转为dict字典  
 print(response.json())  
 print(response.headers.get("Content-Type")) # 获取返回头里的某个key对应的value  
else:  
 print("get request failed")

结果：

200

{"test":"test-info"}

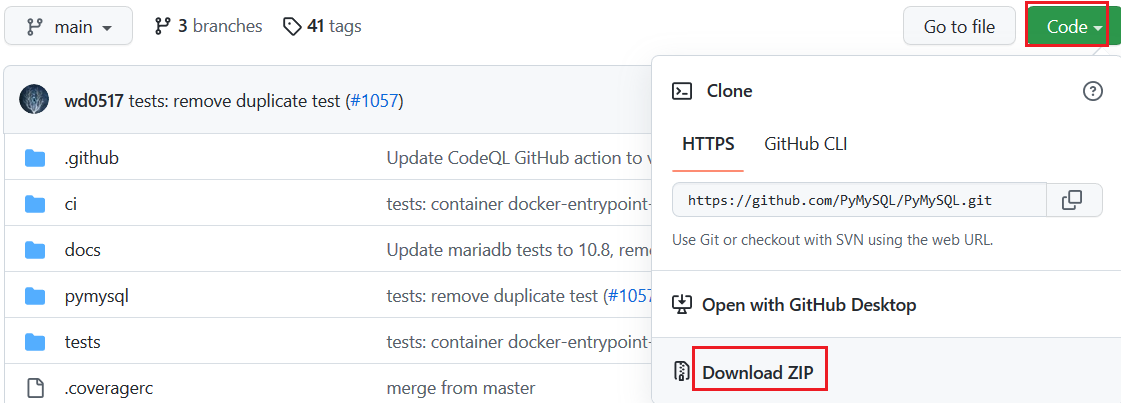
<class 'dict'>

{'test': 'test-info'}

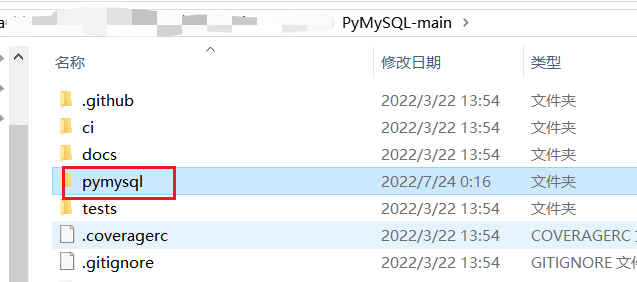
application/json

**★pymysql（数据库连接）**

开源地址： https://github.com/PyMySQL/PyMySQL



下载zip包到本地并解压



把解压目录里的pymysql子目录复制到项目的 site-packages目录下即可使用

#使用：

**★redis**

# pip3 install redis

import redis  
  
conn = redis.Redis(  
 host='10.99.1.3',  
 port=6379,  
 password='Centos123',  
 socket\_timeout=3,  
 db=0, # 若未指定，默认是0  
 decode\_responses=True # 自动将返回值转为字符串，默认返回bytes  
)  
try:  
 if conn.ping(): # 连接成功返回True，否则返回异常  
 print("连接成功")  
 conn.set('keyNamexx', 'valuexx') # 操作成功则返回True  
 data = conn.get('keyNamexx') # 返回str类型的value，若无此key则返回None  
 print(data)  
except Exception as err:  
 print(err)  
 # 可能会抛出以下异常：  
 # <class 'redis.exceptions.AuthenticationError'>  
 # <class 'redis.exceptions.TimeoutError'>  
 # <class 'redis.exceptions.ConnectionError'>

**‌连接池：**

解决的是‌连接建立和释放的开销‌。每次通过网络与Redis服务器建立连接、发送命令、接收响应、再断开连接，这个过程本身是有成本的。连接池会预先创建并维护一批连接，应用程序需要执行Redis命令时，直接从池中获取一个空闲连接来使用，用完后再归还给池。这样避免了频繁的连接/断开操作，节省了大量时间。‌

**‌管道‌：**

解决的是‌网络往返延迟（RTT）‌。当你需要连续执行多个命令时，如果逐个发送，就需要经历多次“发送-等待-接收”的网络往返。管道可以将多个命令一次性打包发送到服务器，服务器再将所有命令的执行结果一次性返回。这极大地减少了网络通信的次数，尤其适合批量操作

**#使用连接池**

连接池能复用连接，减少频繁创建连接的开销

import redis  
  
pool = redis.ConnectionPool(  
 host='10.99.1.3',  
 port=6379,  
 password='Centos123',  
 socket\_timeout=3,  
 max\_connections=20, # 最大连接数，建议设置范围10-50  
 decode\_responses=True  
)  
conn = redis.Redis(connection\_pool=pool) # 从连接池获取连接  
try:  
 if conn.ping(): # 连接成功返回True，否则返回异常  
 print("连接成功")  
 conn.set('keyNamexx', 'valuexx') # 操作成功则返回True  
 conn.expire('keyNamexx', 60) # 设置60秒后过期  
 ret = conn.exists('keyNamexx') # 判断key是否存在，存在则返回数字1，否则返回0  
 print(ret)  
 data = conn.get('keyNamexx') # 返回str类型的value，若无此key则返回None  
 print(data)  
 print(conn.ttl('keyNamexx')) # 查看剩余时间，单位：秒；-1表示永不过期，-2表示已过期  
except Exception as err:  
 print(err)  
 # 可能会抛出以下异常：  
 # <class 'redis.exceptions.AuthenticationError'>  
 # <class 'redis.exceptions.TimeoutError'>  
 # <class 'redis.exceptions.ConnectionError'>

结果：

连接成功

1

valuexx

60

建议是优先使用mset、mget等批量命令，减少网络请求次数

**#使用管道**

执行大量的写操作时，使用管道可以显著提高效率

import redis  
  
pool = redis.ConnectionPool(  
 host='10.99.1.3',  
 port=6379,  
 password='Centos123',  
 socket\_timeout=3,  
 max\_connections=20, # 最大连接数，建议设置范围10-50  
 decode\_responses=True  
)  
conn = redis.Redis(connection\_pool=pool) # 从连接池获取连接  
try:  
 if conn.ping(): # 连接成功返回True，否则返回异常  
 print("连接成功")  
 pipe = conn.pipeline() # 创建管道  
 for i in range(1000):  
 pipe.set(f'keyName{i}', f'value{i}') # 操作成功则返回True  
 response = pipe.execute() # 执行管道中的所有命令，返回每一条命令的结果到一个list中  
 print(response) # 结果list  
except Exception as err:  
 print(err)

结果：



说明：管道中的命令是按顺序执行的，不是原子性的，如果其中的某个命令失败了，后续命令仍然会执行，如果需要原子性操作，建议使用mset等命令

**虽然连接池和管道可以一起使用，但在某些特定场景下需要注意：**

‌Redis集群模式‌：在Redis集群环境下，使用管道需要特别小心。因为集群中的数据是分片存储的，如果管道中的多个命令操作的是不同槽位（slot）的数据，它们可能需要被发送到不同的Redis节点。而传统的管道机制是绑定到一个连接上的，这可能导致命令被发送到错误的节点。‌

因此，在集群模式下使用管道时，需要确保管道内的所有命令都操作的是同一个key（从而位于同一个节点上），或者使用支持集群的客户端库提供的特殊管道功能

**★selenium（浏览器自动化访问网站操作）**

# pip3 install selenium #安装selenium模块，可指定版本selenium=='4.16.0'

安装谷歌浏览器驱动 https://chromedriver.com/

<=114版本下载地址： https://chromedriver.chromium.org/downloads

>=115版本下载地址： https://googlechromelabs.github.io/chrome-for-testing/

把下载的浏览器驱动压缩包进行解压，将解压的chromedriver.exe 放到Python 的 项目\venv\Scripts 目录下

使用：

from selenium import webdriver  
  
def test\_selenium():  
 chrome\_options = webdriver.ChromeOptions()  
 # 解决谷歌浏览器提示"chrome正受到自动测试软件的控制"  
 chrome\_options.add\_experimental\_option("useAutomationExtension", False)  
 chrome\_options.add\_experimental\_option("excludeSwitches", ['enable-automation'])  
  
 # 解决密码提示  
 prefs = {"credentials\_enable\_service": False,  
 "profile.password\_manager\_enabled": False}  
 chrome\_options.add\_experimental\_option("prefs", prefs)  
  
 # 信任网站自签名证书（信任不安全的连接）  
 chrome\_options.set\_capability("acceptInsecureCerts", True)  
 # 禁用javascript  
 chrome\_options.add\_argument('--disable-javascript')  
 # 设置UserAgent  
 user\_agent = ["--user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64)",  
 "AppleWebKit/537.36 (KHTML, like Gecko)",  
 "Chrome/120.0.0.0",  
 "Safari/537.36"]  
 chrome\_options.add\_argument(" ".join(user\_agent))  
 # 创建一个浏览器驱动  
 driver = webdriver.Chrome(options=chrome\_options)  
 driver.maximize\_window() # 浏览器最大化  
 # driver.get("https://limaofu.github.io")  
 # element = driver.find\_element(by=By.CLASS\_NAME, value='active')  
 # print(element.text)  
 # print(element.get\_attribute('herf'))  
 # driver.get("https://www.baidu.com")  
 driver.get("https://10.99.1.234")  
 time.sleep(2)  
 driver.find\_element(by=By.ID, value="user\_login").send\_keys("gituser1")  
 time.sleep(0.5)  
 driver.find\_element(by=By.ID, value="user\_password").send\_keys("Centos123") # 一般可以在输入密码这里，输入\n回车进行登录  
 time.sleep(0.5)  
 driver.find\_element(by=By.CLASS\_NAME, value="gl-button-text").click() # 也可找到“登录”按钮，点击登录  
 time.sleep(3)  
 img\_path = "D:\\tmp\\test.png"  
 driver.save\_screenshot(img\_path)  
 # driver.fullscreen\_window() # 浏览器全屏化（即不显示系统状态栏），maximize\_window时仍显示系统状态栏  
 # element = driver.find\_element(By.ID, 'su')  
 # img\_path = "D:\\tmp\\test.png"  
 # element.screenshot(img\_path) # 对单个元素进行截图  
 time.sleep(3)  
 driver.quit() # 关闭控制台

if \_\_name\_\_ == '\_\_main\_\_':  
 test\_selenium()

**★PIL模块（pillow图像处理）**

PIL是指“Python Imaging Library”，PIL通过Image类来表示一个图像对象，可以对该对象进行各种操作，如调整图像大小、裁剪、旋转、滤镜、格式转换等。在将PIL图像对象保存为其他格式时，需要使用save()方法，并将文件名参数设置为要保存的文件名及格式后缀。基本上能将图片转换成任何一种常见的图片格式，比如BMP、GIF、TIFF等。

# pip3 install pillow

import tkinter  
from PIL import Image, ImageTk  
import cv2  
  
def test\_tk():  
 window = tkinter.Tk()  
 window.title("pillow")  
 # 创建一个 Tkinter 界面  
 canvas = tkinter.Canvas(window, width=640, height=480)  
 canvas.pack()  
 cap = cv2.VideoCapture(0) # 打开摄像头  
 ret, frame = cap.read() # 捕捉一帧图像  
 img = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB) # 将 OpenCV 图像转换为 PIL 图像  
 img = Image.fromarray(img)  
 imgtk = ImageTk.PhotoImage(image=img) # 创建 Tkinter 图像  
 canvas.create\_image(0, 0, anchor=tkinter.NW, image=imgtk) # 在画布上绘制图像  
 window.mainloop() # 运行主循环  
 cap.release() # 释放摄像头资源  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_tk()

**★调整图片大小**

from PIL import Image  
  
image1 = Image.open('D:\\2寸照片-白底.jpeg') # 原照片  
new\_image = image1.resize((484, 680)) # 新照片，调整大小，单位：像素  
  
# 保存新照片到文件，支持jpg及png等格式（同等质量下，jpg文件较小，png文件较大）  
new\_image.save('D:\\test.jpg', quality=75) # quality控制jpg图片的质量，这对输出文件的大小有直接影响

**★cv2模块（opencv-python图像视频处理）**

# pip3 install opencv-python

import cv2  
  
def test\_cv2\_img():  
 img1 = cv2.imread('D:\\tmp\\test.png')  
 img2 = cv2.resize(img1, (420, 360))

img3 = cv2.copyMakeBorder(img1, 2, 2, 2, 2, cv2.BORDER\_CONSTANT, value=[255, 0, 0])

# 设置图片边框，top、bottom、left、right边框厚度，value颜色顺序为BGR

cv2.imwrite('D:\\tmp\\test2.png', img2)  
 while True:  
 cv2.imshow('xxx', img2) # 显示图片  
 key = cv2.waitKey(1) # 判断退出的条件 当按下'Q'键的时候呢，就退出  
 if key == ord('q'): # 如果按下'Q'键就截图保存并退出  
 break  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_cv2\_img()

import cv2  
  
def test\_cv2\_video\_capture():  
 capture = cv2.VideoCapture(0) # 打开摄像头，0是默认的摄像头 如果有多个摄像头，可以设置1,2,3...  
 while True: # 进入无限循环，不使用循环则只捕捉一张图片就结束了  
 ret, frame = capture.read() # 将摄像头拍到的图像作为frame值  
 frame = cv2.flip(frame, 1) # 如果是电脑前置摄像头，则需要将图像左右调换  
 cv2.imshow('video', frame) # 将frame的值显示出来 有两个参数 前一个是窗口名字，后面是值  
 key = cv2.waitKey(1) # 判断退出的条件 当按下'Q'键的时候呢，就退出  
 if key == ord('q'): # 如果按下'Q'键就截图保存并退出  
 save\_path = "D:\\tmp\\test-capture.jpg" # 带有中文的保存文件路径  
 cv2.imwrite(save\_path, frame) # imwrite()不支持中文路径和文件名  
 # img\_write = cv2.imencode(".jpg", frame)[1].tofile(save\_path) #所以使用这个保存到文件  
 break  
 capture.release() # 释放摄像头资源  
 cv2.destroyAllWindows()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_cv2\_video\_capture()

**★pyautogui模块（截屏）**

# pip3 install pyautogui

import pyautogui  
  
  
def test\_pyautogui():  
 img = pyautogui.screenshot(region=(10, 10, 650, 430)) # 不带region参数，则默认截整个屏幕  
 img.save("D:\\tmp\\test-pyautogui.png")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_pyautogui()

numpy在安装pyautogui时已自动安装

import pyautogui  
import cv2  
import numpy  
  
  
def test\_pyautogui\_cv2():  
 img = pyautogui.screenshot()  
 open\_cv\_image = numpy.array(img)  
 # Convert RGB to BGR,opencv read image as BGR,but Pillow is RGB  
 # OpenCV读取的图片顺序是BGR，而Pillow是RGB，PyAutoGui用的是Pillow，所以需要把RGB转换为BGR，不然OpenCV中的颜色会发蓝  
 open\_cv\_image = cv2.cvtColor(open\_cv\_image, cv2.COLOR\_RGB2BGR)  
 cv2.imwrite('D:\\tmp\\test-pyautogui-cv2.png', open\_cv\_image)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_pyautogui\_cv2()

**★django模块**

Django是一个高级的Python Web框架，可以快速开发安全和可维护的网站。Django项目有一个内置的开发服务器，用于在没有任何外部Web服务器的情况下立即运行应用程序。

# pip3 install django==5.0.1

import django  
print(django.get\_version())

# django-admin startproject testapp #创建一个项目，在当前路径下生成testapp项目目录

# python testapp/manage.py runserver #运行应用程序

Watching for file changes with StatReloader

Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

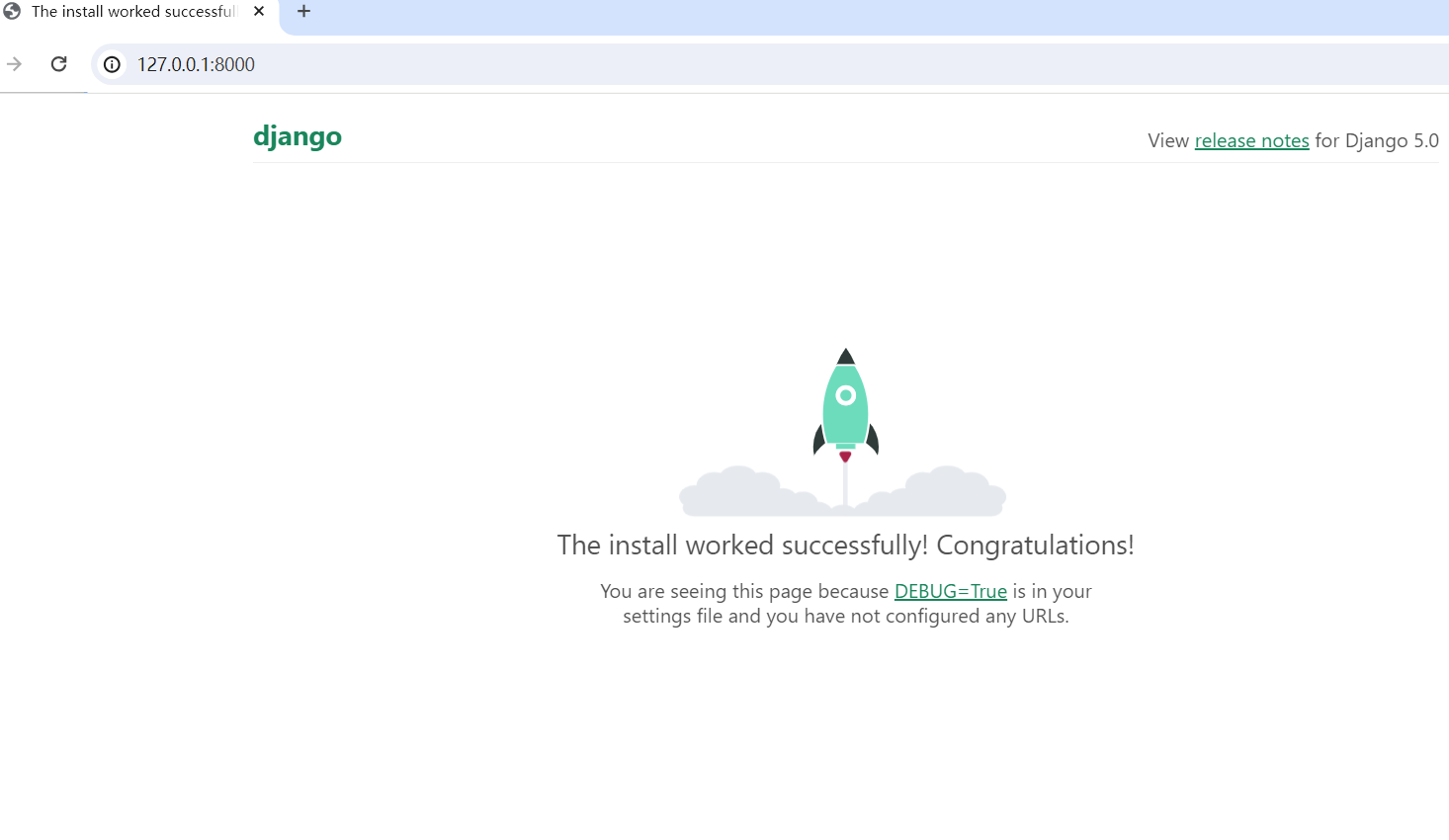
January 31, 2024 - 16:33:27

Django version 5.0.1, using settings 'testapp.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.

以上命令启动一个web服务器，监听8000端口，浏览器访问 http://127.0.0.1:8000/ 时，显示应用的欢迎页面



http://127.0.0.1:8000/admin/ 可登录django管理界面，默认没有账号密码，先创建一个

# python testapp/manage.py migrate #开始执行迁移（不用做这个）

# python testapp/manage.py createsuperuser #创建超级用户,需要输入用户名、邮箱、密码、确认密码

Username (leave blank to use 'cof-thinkpad'): cof

Email address: cof@cof-lee.com

Password:

Password (again):

Superuser created successfully.

# python testapp/manage.py runserver #再次运行应用程序

**★schedule模块**

# pip3 install schedule

import schedule  
import time  
import threading  
  
# 定义任务  
def job(var1):  
 print("Hello " + var1)  
  
# schedule.every(5).seconds.do(job,var1="World")  
schedule.every(5).seconds.until("2024-03-04 13:07").do(job, var1="World")  
# schedule.every().day.at("10:56").do(job, var1="World")  
schedule.every().day.at("10:56").do(job, var1="World").tag("jobxx")  
  
def scheduler\_thread():  
 while True:  
 schedule.run\_pending()  
 time.sleep(1)  
  
# 启动调度器线程  
thread = threading.Thread(target=scheduler\_thread)  
thread.start()  
  
schedule.run\_all() # 立即运行所有作业  
schedule.run\_all(delay\_seconds=10) # 立即运行所有作业，每次作业间隔10秒  
  
all\_jobs = schedule.get\_jobs("jobxx") # 获取指定标签的所有任务  
schedule.clear("jobxx") # 取消指定标签的任务  
  
all\_jobs = schedule.get\_jobs() # 获取目前的所有任务  
schedule.clear() # 取消所有任务

**★xlrd（xls文件处理）**

d

**★openpyxl（xlsx文件处理）**

# pip3 install openpyxl

★创建xlsx文件

import openpyxl  
from openpyxl.drawing.image import Image  
  
workbook = openpyxl.Workbook() # 创建一个工作簿对象  
sheet1 = workbook.active # 获取当前激活的工作表对象  
sheet1.title = "表格1"  
workbook.create\_sheet(index=1, title="表格2") # 创建表格，index从0开始  
data\_list1 = ['name', 'age', 'class', 'weight', 'address', 'phone']  
data\_list2 = ['cof', 18, 1, 70, 'dx', 137]  
sheet1.append(data\_list1)  
sheet1.append(data\_list2)  
sheet3 = workbook.copy\_worksheet(sheet1) # 复制表格1到新的表格（sheet3）新表格名称默认是 源表格名称+"Copy"  
sheet3.title = "表格3"  
sheet3.cell(row=2, column=1, value="ben") # 修改指定单元格的数据，行和列从1开始编号  
cell\_to\_delete = sheet3.cell(row=2, column=2)  
cell\_to\_delete.value = None # 清空指定单元格的数据  
del workbook["表格2"] # 删除指定的表格  
img\_test = Image("test.png")  
sheet3.add\_image(img\_test, anchor="C2") # 插入图片到C2单元格  
workbook.save("test8.xlsx") # 保存工作簿到文件，若文件已存在，则覆盖

★读取xlsx文件

import openpyxl  
  
workbook = openpyxl.load\_workbook("test.xlsx")  
all\_sheet\_list = workbook.sheetnames  
print("所有表格名称:", all\_sheet\_list)  
# sheet1 = workbook['Sheet1'] # 根据表名称获取表格  
sheet1 = workbook.worksheets[0] # 根据表序号获取表格，第1个表格  
print("此表名称为:", sheet1.title) # 获取表名称  
print("此表行数及列数:", sheet1.max\_row, sheet1.max\_column) # 获取表行数及列数  
cell = sheet1.cell(row=2, column=1) # 获取第2行第1列这个单元格数据 ★表格的行和列从1开始编号  
print("目标单元格内容为:", cell.value)  
row\_content = []  
for i in range(sheet1.max\_column):  
 cell = sheet1.cell(row=1, column=i + 1) # ★表格的行和列从1开始编号  
 row\_content.append(cell.value)  
print("第1行数据:", row\_content)  
  
column\_content = []  
for i in range(sheet1.max\_row):  
 cell = sheet1.cell(row=i + 1, column=2) # ★表格的行和列从1开始编号  
 column\_content.append(cell.value)  
print("第2列数据:", column\_content)  
  
# 读取所有数据，一行一行地输出  
for index, row in enumerate(sheet1.iter\_rows(values\_only=True), 1):  
 data = list(row)  
 print(f"第{index}行: ", data)

**★pyperclip剪贴板模块**

# pip3 install pyperclip

import pyperclip  
  
try:  
 selected\_text = "hello"  
 pyperclip.copy(selected\_text) # 复制到剪贴板  
except tkinter.TclError as e:  
 print("TerminalVt100.copy\_selected\_text\_on\_terminal\_text: 未选择任何文字", e)  
  
pasted\_text = pyperclip.paste() # 从剪贴板复制文本  
print(pasted\_text)

**★moviepy**

将mp4视频中的音频导出为mp3文件

# pip3 install moviepy

#!/usr/bin/env python3  
# coding: utf-8  
from moviepy.editor import VideoFileClip # 新版本是 from moviepy import VideoFileClip  
  
video\_path = 'D:\\b037.mp4' # 视频文件路径  
audio\_path = 'D:\\b037.mp3' # 提取音频的保存路径  
  
video\_clip = VideoFileClip(video\_path)  
audio\_clip = video\_clip.audio  
audio\_clip.write\_audiofile(audio\_path)  
  
audio\_clip.close()  
video\_clip.close()

★将音频与视频合并（替换视频中的原音频）

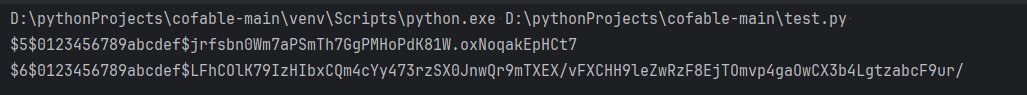
from moviepy import VideoFileClip, AudioFileClip  
  
# 加载视频文件  
video\_clip = VideoFileClip("D:\\xxx-noVoice.mp4")  
  
# 加载音频文件  
audio\_clip = AudioFileClip("D:\\xxx-voice.mp3")  
  
# 使用音频剪辑替换视频中的音频（可选）  
video\_with\_audio = video\_clip.with\_audio(audio\_clip) # 旧版本用 set\_audio  
  
# 输出新的视频文件  
video\_with\_audio.write\_videofile("D:\\output\_video.mp4", codec='libx264', audio\_codec='aac')

**★passlib（生成密码hash）**

# pip3 install passlib

#!/usr/bin/env python3  
# coding: utf-8  
from passlib.hash import sha512\_crypt  
from passlib.hash import sha256\_crypt  
  
password = "passwdxx\_"  
salt = "0123456789abcdef"  
hash\_result1 = sha256\_crypt.hash(password, salt=salt, rounds=5000)  
hash\_result2 = sha512\_crypt.hash(password, salt=salt, rounds=5000)  
print(hash\_result1)  
print(hash\_result2)

结果：



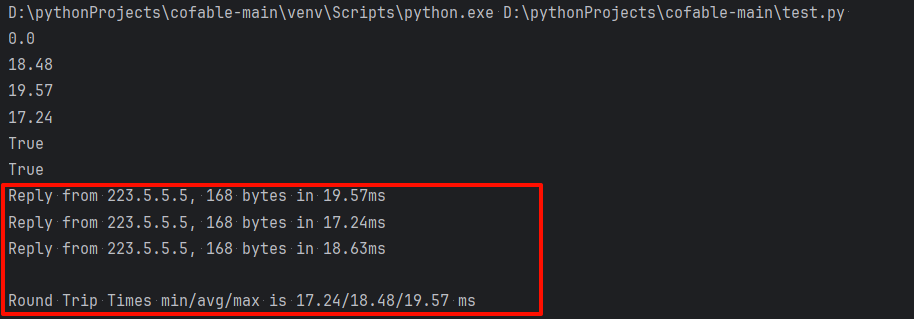
**★pythonping**

ping检测模块

# pip3 install pythonping

import pythonping  
from pythonping.executor import SuccessOn  
  
result = pythonping.ping("223.5.5.5", count=3, timeout=2, interval=3, size=140) # 发起一次ping检测  
# count 一次检测发的包数，发几个包  
# timeout 超时时间，单位：秒  
# interval 两次发包的时间间隔，单位：秒  
# size 每次发包的数据大小，单位字节  
# 只有所有包都发完且有回复或到了超时时间后 ping函数才会返回result  
print(result.packet\_loss) # 丢包率  
print(result.rtt\_avg\_ms) # RTT平均时间，单位毫秒  
print(result.rtt\_max\_ms) # RTT最大时间，单位毫秒  
print(result.rtt\_min\_ms) # RTT最小时间，单位毫秒  
print(result.success(SuccessOn.One)) # 检测是否成功，默认只要有一个包回复就算成功，返回True，失败返回False  
print(result.success(SuccessOn.All)) # 检测是否成功，要求所有包都有回复才算成功  
print(result) # 打印日志

结果：



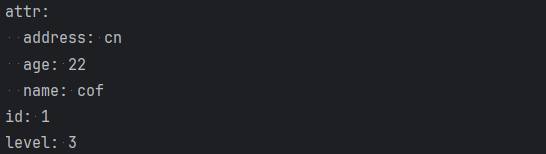
**★PyYAML模块**

# pip3 install PyYAML #安装PyYAML模块，在使用时是用yaml的名字

# 将字典转为yaml字符串

import yaml  
  
data = {'attr': {'address': 'cn', 'age': 22, 'name': 'cof'}, 'id': 1, 'level': 3}  
yaml\_str = yaml.dump(data, default\_flow\_style=False, indent=2, sort\_keys=True)  
print(yaml\_str)

结果：



# 将yaml字符串转为字典

import yaml  
  
str\_xx = """

attr:  
 address: cn  
 age: 22  
 name: cof  
id: 1  
level: 3  
"""  
data = yaml.load(str\_xx, Loader=yaml.Loader)  
print(data)

结果：

{'attr': {'address': 'cn', 'age': 22, 'name': 'cof'}, 'id': 1, 'level': 3}

说明：读取字符串以及读取文件时都是使用yaml.load()方法

**★jinja2模块**

# pip3 install jinja2 #安装jinja2模块

创建模板文件test.j2，文件内容如下：

hello, everybody

my name is {{ name }}, my age is {{ age }}

import jinja2  
  
j2file = "test.j2"  
with open(j2file, 'r') as file:  
 template\_xx = jinja2.Template(file.read()) # 导入j2模板文件  
  
context = {"name": "cof", "age": 18} # 定义模板变量  
content\_xx = template\_xx.render(context) # 渲染模板

print(content\_xx)

结果：



import jinja2  
  
j2\_str = """hello, everybody  
my name is {{ name }}, my age is {{ age }}  
"""  
  
template\_xx = jinja2.Template(j2\_str) # 导入j2模板字符串  
context = {"name": "cof", "age": 18} # 定义模板变量  
content\_xx = template\_xx.render(context) # 渲染模板  
print(content\_xx)

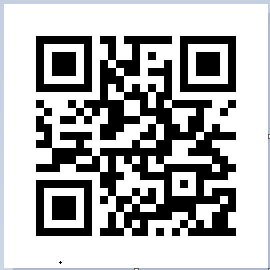
结果同上

**★qrcode生成二维码**

# pip3 install qrcode #安装qrcode模块

#!/usr/bin/env python3  
# -\*- coding: utf-8 -\*-  
  
import qrcode  
  
data = """test\_qrcode\_string""" # 二维码的内容，类型为字符串  
  
img = qrcode.make(data, version=2, box\_size=8, border=4, error\_correction=qrcode.constants.ERROR\_CORRECT\_M)  
  
img.save("test\_qrcode.png") # 将生成的图片保存到文件  
img.show() # 将生成的图片使用默认的图片浏览工具打开，正式使用时，不可在代码里写这行

生成的图片：



**qrcode.make各参数说明：**

|  |  |  |
| --- | --- | --- |
| 参数 | 取值范围 | 说明 |
| version | 1-40，默认=None | 指定二维码的版本，它决定了二维码的大小，其中1是最小的，40是最大的。如果不指定这个参数，qrcode库会根据添加的数据量自动选择一个合适的版本。 |
| box\_size | 整型，默认=10 | 指定二维码中每个小方块（模块）的像素大小，本例中指定为8，则二维码中的每个点（模块）都用8x8的像素块来表示 |
| border | 整型，默认=4 | 设置二维码周围的空白边框大小，以模块为单位 |
| error\_correction | 默认=ERROR\_CORRECT\_M | 设置二维码的纠错级别，它决定了二维码可以纠正的错误量 |

**error\_correction级别：**

|  |  |
| --- | --- |
| qrcode.constants.ERROR\_CORRECT\_L | Low，表示最低的纠错级别（可以纠正最大7%的错误模块） |
| qrcode.constants.ERROR\_CORRECT\_M | Medium，可以纠正最大15%的错误模块 |
| qrcode.constants.ERROR\_CORRECT\_Q | Quartile，可以纠正最大25%的错误模块 |
| qrcode.constants.ERROR\_CORRECT\_H | High，可以纠正最大30%的错误模块 |

二维码的数据容量取决于其版本和纠错级别。二维码有40个不同的版本（从版本1到版本40），每个版本的大小都是前一个版本的4倍。版本1的二维码是21x21模块，每增加一个版本，二维码的尺寸就增加4个模块（例如，版本2是25x25模块）。最大的版本40是177x177模块。

纠错级别越高，可以存储的有效数据就越少，因为一部分容量被用于纠错信息。

不同版本和纠错级别，二维码大约可以存储的字符数（以UTF-8编码的数字、字母和常见符号计算）：

|  |  |  |
| --- | --- | --- |
| 版本 | 纠错等级 | 数据容量（字符数） |
| 1 | L | 41 |
| 1 | M | 34 |
| 1 | Q | 27 |
| 1 | H | 17 |
| 40 | L | 7089 |
| 40 | M | 5933 |
| 40 | Q | 4689 |
| 40 | H | 3223 |

注意：如果超过了二维码的最大数据容量，会收到以下错误：

ValueError: Invalid version (was 41, expected 1 to 40)

因为库尝试增加版本号以容纳数据，但是超出了最大版本40的限制。

在这种情况下，需要减少数据量或将其分成多个二维码

**★第16章、多进程与多线程**

在python2.6之前python没有官方的线程池模块，只有第三方的theadpool模块，后来

python2.6加入了multiprocessing.dummy线程池

python3.2之后又加入了concurrent.futures模块，支持多线程及多进程；concurrent.futures底层还是用的threading和multiprocessing这2个模块，在其上又封装了一层。

**★多进程multiprocessing.Process类**

python多进程（multi-process）适用于计算密集型的任务，

对于I/O密集型任务应该使用多线程（multi-thread），比如磁盘读写，网络通信等

在python中，对于计算密集型任务，多进程占优势；对于IO密集型任务，多线程占优势。

使用多进程时，各进程拥有独立的内存空间，无法共享内存空间，所以各进程无法访问同一个全局变量，若需要各进程共享某个全局变量，可以使用multiprocessing.Value()或multiprocessing.Array()来创建共享变量

在windows系统里使用多进程时，程序里必须有入口函数main()，且程序末尾必须调用入口函数，即：

if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★★实例化multiprocessing.Process类-调用外部函数**

import multiprocessing  
import os  
import time  
  
  
def work\_func():  
 res = 0  
 for i in range(100000000):  
 res \*= i  
  
  
def main():  
 process\_list = []  
 print("本机cpu为", os.cpu\_count(), "核")  
 start = time.time()  
 for i in range(4):  
 p = multiprocessing.Process(target=work\_func) # 创建一个进程，将目标函数放入  
 process\_list.append(p)  
 p.start()  
 for p in process\_list:  
 p.join()  
 stop = time.time()  
 print("计算密集型任务，多进程耗时 %s" % (stop - start))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★继承multiprocessing.Process类-调用外部函数**

import multiprocessing  
import os  
import time  
  
  
def work\_func(argxx):  
 res = 0  
 for i in range(argxx):  
 res \*= i  
  
  
class MyProcess(multiprocessing.Process):  
 def \_\_init\_\_(self, target, args):  
 super().\_\_init\_\_()  
 self.target = target  
 self.args = args  
  
 def run(self):  
 self.target(\*self.args)  
  
  
def main():  
 process\_list = []  
 print("本机cpu为", os.cpu\_count(), "核")  
 start = time.time()  
 for i in range(4):  
 p = MyProcess(target=work\_func, args=(100000000,)) # 创建一个进程，将目标函数放入  
 process\_list.append(p)  
 p.start()  
 for p in process\_list:  
 p.join()  
 stop = time.time()  
 print("计算密集型任务，多进程耗时 %s" % (stop - start))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★继承multiprocessing.Process类-重写init/run方法**

import multiprocessing  
import os  
import time  
  
  
def work\_func(argxx):  
 res = 0  
 for i in range(argxx):  
 res \*= i  
  
  
class MyProcess(multiprocessing.Process):  
 def \_\_init\_\_(self, argxx):  
 super().\_\_init\_\_()  
 self.argxx = argxx  
  
 def run(self):  
 res = 0  
 for i in range(self.argxx):  
 res \*= i  
  
  
def main():  
 process\_list = []  
 print("本机cpu为", os.cpu\_count(), "核")  
 start = time.time()  
 for i in range(4):  
 p = MyProcess(100000000) # 创建一个进程，将目标函数放入  
 process\_list.append(p)  
 p.start()  
 for p in process\_list:  
 p.join()  
 stop = time.time()  
 print("计算密集型任务，多进程耗时 %s" % (stop - start))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多进程同步\_lock（互斥锁）**

使用multiprocessing.Lock()方法创建一个互斥锁，

互斥锁在同一时刻只允许一个线程访问共享数据

使用多进程时，各进程拥有独立的内存空间，无法共享内存空间，所以各进程无法访问同一个全局变量，若需要各进程共享某个全局变量，可以使用multiprocessing.Value()或multiprocessing.Array()或multiprocessing.Manager().list()或multiprocessing.Manager().dict()来创建共享变量

import multiprocessing  
import os  
import time  
import random  
  
  
def work\_func(lock, shared\_var):  
 for i in range(10):  
 lock.acquire()  
 print("进程 {} 已获取锁".format(multiprocessing.current\_process().name)) # 获取锁，用于线程同步  
 n = random.randint(0, 5)  
 shared\_var.value += n  
 time.sleep(random.randint(0, 1))  
 shared\_var.value -= n  
 print("进程 {} 里面查看shared\_var.value为：{}".format(multiprocessing.current\_process().name, shared\_var.value))  
 lock.release() # 释放锁，开启下一个线程  
 print("进程 {} 释放锁".format(multiprocessing.current\_process().name))  
  
  
def main():  
 process\_list = []  
 lock = multiprocessing.Lock()  
 shared\_var = multiprocessing.Value('i', 0) # 第一个参数i表示整数，还可为d（浮点数），第2个参数表示初始值  
 print("本机cpu为", os.cpu\_count(), "核")  
 for i in range(8):  
 t = multiprocessing.Process(target=work\_func, args=(lock, shared\_var)) # 创建一个进程，将目标函数放入  
 process\_list.append(t)  
 t.start()  
 for t in process\_list:  
 t.join()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多进程同步\_Semaphore（信号量）**

使用multiprocessing.BoundedSemaphore()方法创建一个信号量，

互斥锁同一时刻只允许一个进程访问共享数据，而信号量在同一时刻允许一定数量的进程访问共享数据。

比如柜台有5个窗口，允许同时有5个人办理业务，后面的人只能等待，5人中有人办理完业务，等待的人才能去办理

使用多进程时，各进程拥有独立的内存空间，无法共享内存空间，所以各进程无法访问同一个全局变量，若需要各进程共享某个全局变量，可以使用multiprocessing.Value()或multiprocessing.Array()或multiprocessing.Manager().list()或multiprocessing.Manager().dict()来创建共享变量

import multiprocessing  
import time  
import random  
  
  
# 模拟柜台业务办理  
def work\_func(semaphore, name, current\_user):  
 semaphore.acquire()  
 current\_user.append(multiprocessing.current\_process().name)  
 print("当前办理业务的顾客有：{}\n顾客{} 正在办理业务\n".format(current\_user, name))  
 time.sleep(random.randint(6, 8))  
 current\_user.remove(multiprocessing.current\_process().name)  
 semaphore.release()  
  
  
def main():  
 process\_list = []  
 c\_user\_list=[]  
 semaphore = multiprocessing.BoundedSemaphore(5) # 同时只允许有5个任务在运行  
 current\_user = multiprocessing.Manager().list() # 还可有multiprocessing.Manager().dict()  
 for i in range(18):  
 t = multiprocessing.Process(target=work\_func, args=(semaphore, i + 1, current\_user))  
 process\_list.append(t)  
  
 for process in process\_list:  
 process.start()  
 print(process.name, "已开始运行")  
  
 for process in process\_list:  
 process.join()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多进程同步\_Condition（条件对象）**

使用multiprocessing.Condition()方法创建一个条件对象，

条件对象能让一个进程A暂停下来，等待另一个进程B的通知，当进程B满足某个条件后通知进程A继续运行

import multiprocessing  
  
  
def operator\_func(cond, name):  
 cond.acquire()  
 print(name + ": 我可以变更了吗？")  
 cond.notify() # 发消息给Manager，唤醒一个挂起的线程  
 cond.wait() # 等待Manager的回复，释放内部所占用的琐，同时线程被挂起，直至接收到通知被唤醒或超时  
 print(name + ": 收到，现在修改变更方案")  
 cond.notify() # 再次发消息给Manager，唤醒一个挂起的线程  
 cond.wait() # 等待Manager的回复  
 print(name + ": 收到，现在变更")  
 cond.release()  
  
  
def manager\_func(cond, name):  
 cond.acquire()  
 cond.wait() # manager要先运行，等待操作员的请求  
 print(name + ": 变更方案有问题，先不变更，先改方案")  
 cond.notify() # 发消息给Operator，唤醒一个挂起的线程  
 cond.wait() # 等待Operator的回复  
 print(name + ": 好，可以变更了")  
 cond.notify() # 发消息给Operator，唤醒一个挂起的线程  
 cond.release()  
  
  
def main():  
 cond = multiprocessing.Condition()  
 opuser = multiprocessing.Process(target=operator\_func, args=(cond, "cof-lee",))  
 mgmt = multiprocessing.Process(target=manager\_func, args=(cond, "boss",))  
 mgmt.start()  
 opuser.start()  
 mgmt.join()  
 opuser.join()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多进程同步\_Event（事件）**

使用multiprocessing.Event()方法创建一个事件对象，

事件用于进程之间的通信。一个进程发出一个信号，其他一个或者多个进程等待，调用Event对象的wait方法，进程则会被阻塞，直到别的进程set之后才会被唤醒。

import random  
import multiprocessing  
import time  
  
  
def operator\_func(event, name):  
 print(name + ": 我可以变更了吗？")  
 event.set() # 发请求给Manager，唤醒一个挂起的线程  
 event.clear()  
 event.wait() # 等待Manager的回复，释放内部所占用的琐，同时线程被挂起，直至接收到通知被唤醒或超时  
 print(name + ": 收到，现在修改变更方案")  
 time.sleep(random.randint(1, 3))  
 event.set() # 再次发消息给Manager，唤醒一个挂起的线程  
 event.clear()  
 event.wait() # 等待Manager的回复  
 event.clear()  
 print(name + ": 收到，现在变更")  
  
  
def manager\_func(event, name):  
 event.wait() # manager要先运行，等待操作员的请求  
 print(name + ": 变更方案有问题，先不变更，先改方案")  
 time.sleep(random.randint(1, 3))  
 event.set() # 发回复给Operator，唤醒一个挂起的线程  
 event.clear()  
 event.wait() # 等待Operator的再次回复  
 print(name + ": 好，可以变更了")  
 time.sleep(random.randint(1, 3))  
 event.set() # 发消息给Operator，唤醒一个挂起的线程  
  
  
def main():  
 event = multiprocessing.Event()  
 opuser = multiprocessing.Process(target=operator\_func, args=(event, "cof-lee",))  
 mgmt = multiprocessing.Process(target=manager\_func, args=(event, "boss",))  
 mgmt.start()  
 opuser.start()  
 mgmt.join()  
 opuser.join()

if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多进程\_线程队列Queue**

使用multiprocessing.JoinableQueue()方法创建一个队列，它与queue.Queue的区别在于，它是多进程安全的，不用担心它的互斥和死锁问题。

比如，有一个仓库用来存放商品，假如最多只能放5个商品，A不停的放，B不停的取，A和B的放取速度不一致，就可以用队列来做

import multiprocessing  
import queue  
import time  
import random  
  
  
def producer\_a(test\_queue):  
 product\_id = 1  
 while True:  
 print('A 查看当前队列长度：', test\_queue.qsize())  
 test\_queue.put(f"商品 {product\_id}")  
 print(f"A 放入:[商品 {product\_id}]，放入后队列长度：{test\_queue.qsize()}") # 如果队列已达到最大值，则阻塞在此，直到有人取出  
 product\_id += 1  
 time.sleep(random.randint(2, 3))  
  
  
def consumer\_b(test\_queue):  
 while True:  
 print('B 查看当前队列长度：', test\_queue.qsize())  
 print(f"B 取出 [{test\_queue.get()}]，取出后队列长度：{test\_queue.qsize()}") # 如果队列里无内容，则阻塞在此，直到有人放入  
 time.sleep(random.randint(3, 6))  
  
  
def main():  
 test\_queue = multiprocessing.JoinableQueue(maxsize=5) # 先进先出队列  
 p = multiprocessing.Process(target=producer\_a, args=(test\_queue,))  
 c = multiprocessing.Process(target=consumer\_b, args=(test\_queue,))  
 p.start()  
 c.start()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★★多进程\_进程池multiprocessing.Pool**

from multiprocessing import Pool as ProcessPool  
import time  
  
  
def work\_func(n):  
 print(f"this is work {n}")  
 res = 0  
 for i in range(100000000):  
 res \*= i  
  
  
def main():  
 start\_time = time.time()  
 for i in range(32):  
 work\_func(i)  
 time1 = time.time() - start\_time  
  
 # 开8个 worker，没有参数时默认是 cpu 的核心数  
 pool = ProcessPool(processes=32)  
 # 在线程中执行 work\_func 并返回执行结果  
 start\_time2 = time.time()  
 results2 = pool.map(work\_func, range(32))  
 pool.close()  
 pool.join()  
 time2 = time.time() - start\_time2  
 print("单线程顺序执行耗时:", time1)  
 print("线程池并发执行耗时:", time2)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

单线程顺序执行耗时: 98.03768038749695

线程池并发执行耗时: 36.293702602386475

**★多线程threading.Thread类**

python多线程（multi-thread）适用于I/O密集型的任务，

对于计算密集型任务应该使用多进程（multi-process）

Python3有两个标准库 \_thread 和 threading 提供对线程的支持

python2为thread和threading模块

**★简单线程\_thread模块**

import \_thread  
import time  
  
def my\_thread\_func(thread\_name, count): # 定义线程函数  
 for i in range(count):  
 time.sleep(1)  
 print("线程名：{},第{}次输出".format(thread\_name, i))  
  
try:  
 \_thread.start\_new\_thread(my\_thread\_func, ("thread1", 5)) #创建子线程  
 \_thread.start\_new\_thread(my\_thread\_func, ("thread2", 5))  
except Exception as e:  
 print(e)  
  
for i in range(8): #主线程要是退出了，则子线程也结束了，要确保在子线程结束前，主线程仍在运行  
 time.sleep(1)  
 print("this is main thread")

**★threading模块Thread类**

提供复杂的方法

import threading  
import time  
  
class myThread(threading.Thread):  
 def \_\_init\_\_(self, name, delay): # 重写init初始化方法  
 threading.Thread.\_\_init\_\_(self)  
 self.name = name  
 self.delay = delay  
  
 def run(self): # 重写run方法，用户线程要执行的内容在这里  
 time.sleep(self.delay)  
 print("%s: %s" % (self.name, time.ctime(time.time())))  
 print("退出线程：" + self.name)  
  
thread1 = myThread("Thread-1", 4) # 创建子线程  
thread2 = myThread("Thread-2", 3)  
  
thread1.start() # 开始线程  
thread2.start()  
thread1.join() # 等待线程退出  
thread2.join()  
print("退出主线程")

**★★实例化threading.Thread类-调用外部函数**

调用threading.Thread类的start()方法

import time  
import threading  
  
  
def work\_func(varxx):  
 print('线程名称：{} 参数值：{}'.format(threading.current\_thread().name, varxx))  
  
  
def main():  
 print(f'主线程开始时间：{time.strftime("%Y-%m-%d %H:%M:%S")}')  
 thread\_list = []  
 for i in range(5):  
 t = threading.Thread(target=work\_func, args=(i+1,))  
 thread\_list.append(t)  
 t.start()  
 for t in thread\_list:  
 t.join()  
 print(f'主线程结束时间：{time.strftime("%Y-%m-%d %H:%M:%S")}')  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★继承threading.Thread类-调用外部函数**

import time  
import threading  
  
  
def work\_func(varxx):  
 print('线程名称：{} 参数：{}'.format(threading.current\_thread().name, varxx))  
  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self, target, args):  
 super().\_\_init\_\_()  
 self.target = target  
 self.args = args  
  
 def run(self):  
 self.target(\*self.args)  
  
  
def main():  
 print(f'主线程开始时间：{time.strftime("%Y-%m-%d %H:%M:%S")}')  
  
 # 初始化3个线程，传递不同的参数  
 t1 = MyThread(target=work\_func, args=(11,))  
 t2 = MyThread(target=work\_func, args=(22,))  
 t3 = MyThread(target=work\_func, args=(33,))  
 # 开启三个线程  
 t1.start()  
 t2.start()  
 t3.start()  
 # 等待线程运行结束  
 t1.join()  
 t2.join()  
 t3.join()  
  
 print(f'主线程结束时间：{time.strftime("%Y-%m-%d %H:%M:%S")}')  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★继承threading.Thread类-重写init/run方法**

在子类中重写run()和\_\_init\_\_()方法

import time  
import threading  
  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self, varxx):  
 super().\_\_init\_\_()  
 self.varxx = varxx  
  
 def run(self):  
 print('线程名称：{} 参数值：{}'.format(threading.current\_thread().name, self.varxx))  
  
  
def main():  
 print(f'主线程开始时间：{time.strftime("%Y-%m-%d %H:%M:%S")}')  
 # 初始化3个线程，传递不同的参数  
 t1 = MyThread(11)  
 t2 = MyThread(22)  
 t3 = MyThread(33)  
 # 开启三个线程  
 t1.start()  
 t2.start()  
 t3.start()  
 # 等待线程运行结束  
 t1.join()  
 t2.join()  
 t3.join()  
 print(f'主线程结束时间：{time.strftime("%Y-%m-%d %H:%M:%S")}')  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多线程同步\_lock（互斥锁）**

使用threading.Lock()方法创建一个互斥锁，

互斥锁在同一时刻只允许一个线程访问共享数据

import threading  
import random  
  
num = 0 # 各线程之间可访问同一内存块，即可访问同一个全局变量  
  
  
def task\_thread(lock):  
 global num  
 # 获取锁，用于线程同步  
 lock.acquire()  
 print("线程 {} 已获取锁".format(threading.current\_thread().name))  
 for i in range(1000):  
 n = random.randint(0, 99)  
 num = num + n  
 num = num - n  
 # 释放锁，开启下一个线程  
 print("线程 {} 里面查看num为：{}".format(threading.current\_thread().name, num))  
 print("线程 {} 释放锁".format(threading.current\_thread().name))  
 lock.release()  
  
  
def main():  
 lock = threading.Lock()  
 t1 = threading.Thread(target=task\_thread, args=(lock,))  
 t2 = threading.Thread(target=task\_thread, args=(lock,))  
 t3 = threading.Thread(target=task\_thread, args=(lock,))  
 t1.start()  
 t2.start()  
 t3.start()  
 t1.join()  
 t2.join()  
 t3.join()  
 print(num)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多线程同步\_Semaphore（信号量）**

使用threading.BoundedSemaphore()方法创建一个信号量，

互斥锁同一时刻只允许一个线程访问共享数据，而信号量在同一时刻允许一定数量的线程访问共享数据。

比如柜台有5个窗口，允许同时有5个人办理业务，后面的人只能等待，5人中有人办理完业务，等待的人才能去办理

import threading  
import time  
import random  
  
current\_user = [] # 各线程之间可访问同一内存块，即可访问同一个全局变量  
  
  
# 模拟柜台业务办理  
def work\_func(semaphore, name):  
 print("当前激活的线程数量：", threading.active\_count()) # 当前激活的线程数量（含主线程）  
 semaphore.acquire()  
 current\_user.append(threading.current\_thread().name)  
 print("当前办理业务的顾客有：{}\n顾客{} 正在办理业务\n".format(current\_user, name))  
 time.sleep(random.randint(1, 8))  
 current\_user.remove(threading.current\_thread().name)  
 semaphore.release()  
  
  
def main():  
 thread\_list = []  
 semaphore = threading.BoundedSemaphore(5) # 同时只允许有5个任务在运行  
 for i in range(18):  
 t = threading.Thread(target=work\_func, args=(semaphore, i + 1,))  
 thread\_list.append(t)  
  
 for thread in thread\_list:  
 thread.start()  
 print(thread.name, "已开始运行")  
  
 for thread in thread\_list:  
 thread.join()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**★多线程同步\_Condition（条件对象）**

使用threading.Condition()方法创建一个条件对象，

条件对象能让一个线程A暂停下来，等待另一个线程B的通知，当线程B满足某个条件后通知线程A继续运行

import threading  
  
  
def operator\_func(cond, name):  
 cond.acquire()  
 print(name + ": 我可以变更了吗？")  
 cond.notify() # 发消息给Manager，唤醒一个挂起的线程  
 cond.wait() # 等待Manager的回复，释放内部所占用的琐，同时线程被挂起，直至接收到通知被唤醒或超时  
 print(name + ": 收到，现在修改变更方案")  
 cond.notify() # 再次发消息给Manager，唤醒一个挂起的线程  
 cond.wait() # 等待Manager的回复  
 print(name + ": 收到，现在变更")  
 cond.release()  
  
  
def manager\_func(cond, name):  
 cond.acquire()  
 cond.wait() # manager要先运行，等待操作员的请求  
 print(name + ": 变更方案有问题，先不变更，先改方案")  
 cond.notify() # 发消息给Operator，唤醒一个挂起的线程  
 cond.wait() # 等待Operator的回复  
 print(name + ": 好，可以变更了")  
 cond.notify() # 发消息给Operator，唤醒一个挂起的线程  
 cond.release()  
  
  
def main():  
 cond = threading.Condition()  
 opuser = threading.Thread(target=operator\_func, args=(cond, "cof-lee",))  
 mgmt = threading.Thread(target=manager\_func, args=(cond, "boss",))  
 mgmt.start()  
 opuser.start()  
 mgmt.join()  
 opuser.join()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

上面程序先启动了Manager线程，mgmt虽然获取到了条件变量锁cond，但又执行了wait并释放条件变量锁，自身进入阻塞状态。

Operator线程启动后，就获得了条件变量锁cond并发出了消息，之后通过notify唤醒一个挂起的线程。

最后通过release程序释放资源

**★多线程同步\_Event（事件）**

使用threading.Event()方法创建一个事件对象，

事件用于线程之间的通信。一个线程发出一个信号，其他一个或者多个线程等待，调用Event对象的wait方法，线程则会被阻塞，直到别的线程set之后才会被唤醒。

import random  
import threading  
import time  
  
  
def operator\_func(event, name):  
 print(name + ": 我可以变更了吗？")  
 event.set() # 发请求给Manager，唤醒一个挂起的线程  
 event.clear()  
 event.wait() # 等待Manager的回复，释放内部所占用的琐，同时线程被挂起，直至接收到通知被唤醒或超时  
 print(name + ": 收到，现在修改变更方案")  
 time.sleep(random.randint(1, 3))  
 event.set() # 再次发消息给Manager，唤醒一个挂起的线程  
 event.clear()  
 event.wait() # 等待Manager的回复  
 event.clear()  
 print(name + ": 收到，现在变更")  
  
  
def manager\_func(event, name):  
 event.wait() # manager要先运行，等待操作员的请求  
 print(name + ": 变更方案有问题，先不变更，先改方案")  
 time.sleep(random.randint(1, 3))  
 event.set() # 发回复给Operator，唤醒一个挂起的线程  
 event.clear()  
 event.wait() # 等待Operator的再次回复  
 print(name + ": 好，可以变更了")  
 time.sleep(random.randint(1, 3))  
 event.set() # 发消息给Operator，唤醒一个挂起的线程  
  
  
def main():  
 event = threading.Event()  
 opuser = threading.Thread(target=operator\_func, args=(event, "cof-lee",))  
 mgmt = threading.Thread(target=manager\_func, args=(event, "boss",))  
 mgmt.start()  
 opuser.start()  
 mgmt.join()  
 opuser.join()

if \_\_name\_\_ == "\_\_main\_\_":  
 main()

Event内部默认内置了一个标志，初始值为false。上面代码manager通过wait()方法进入等待状态，直到operator调用该Event的set()方法将内置标志设置为true，对象manager再继续运行。

对象boy随后调用Event的clear方法再讲内置的标志设置为False，恢复初始状态，即让对方再次等待

**★多线程\_线程队列Queue**

使用queue.Queue()方法创建一个队列，

python的queue模块提供了同步的，线程安全的队列类。包括

先进先出的队列queue;

后进先出的LifoQueue;

优先级队列PriorityQueue

比如，有一个仓库用来存放商品，假如最多只能放5个商品，A不停的放，B不停的取，A和B的放取速度不一致，就可以用队列来做

import threading  
import queue  
import time  
import random  
  
  
def producer\_a():  
 product\_id = 1  
 while True:  
 print('A 查看当前队列长度：', test\_queue.qsize())  
 test\_queue.put(f"商品 {product\_id}")  
 print(f"A 放入:[商品 {product\_id}]，放入后队列长度：{test\_queue.qsize()}") # 如果队列已达到最大值，则阻塞在此，直到有人取出  
 product\_id += 1  
 time.sleep(random.randint(2, 3))  
  
  
def consumer\_b():  
 while True:  
 print('B 查看当前队列长度：', test\_queue.qsize())  
 print(f"B 取出 [{test\_queue.get()}]，取出后队列长度：{test\_queue.qsize()}") # 如果队列里无内容，则阻塞在此，直到有人放入  
 time.sleep(random.randint(3, 6))

def main():

test\_queue = queue.Queue(maxsize=5) # 先进先出  
 # test\_queue = queue.LifoQueue(maxsize=3) # 后进先出  
 # test\_queue = queue.PriorityQueue(maxsize=3) # 优先级队列  
 p = threading.Thread(target=producer\_a)  
 c = threading.Thread(target=consumer\_b)  
 p.start()  
 c.start()

if \_\_name\_\_ == "\_\_main\_\_":  
 main()

上面的代码就是实现生产者和消费者模型的一个比较简单的例子。

在并发编程中，使用生产者和消费之模式可以解决绝大多数的并发问题

如果生产者处理的速度很快，而消费者处理速度很慢，那么生产者就必须等消费者处理完，才能继续生产数据。

同理，如果消费者的处理能力大于生产者，那消费者就必须等待生产者。

生产者和消费者模式就是通过一个容器（队列）来解决强耦合问题,生产者和消费者之间不通信，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

**★多线程\_线程池multiprocessing.dummy.Pool**

单线程和线程池并发执行效率对比

from multiprocessing.dummy import Pool as ThreadPool  
import time  
  
  
def work\_func(n):  
 print(f"this is work {n}")  
 res = 0  
 for i in range(100000000):  
 res \*= i  
  
  
def main():  
 start\_time = time.time()  
 for i in range(5):  
 work\_func(i)  
 time1 = time.time() - start\_time  
  
 # 开5个 worker，没有参数时默认是 cpu 的核心数  
 pool = ThreadPool(processes=5)  
 # 在线程中执行 work\_func 并返回执行结果  
 start\_time2 = time.time()  
 results2 = pool.map(work\_func, range(5))  
 pool.close()  
 pool.join()  
 time2 = time.time() - start\_time2  
 print("单线程顺序执行耗时:", time1)  
 print("线程池并发执行耗时:", time2)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

对于计算密集型任务，线程池并没有优势

单线程顺序执行耗时: 14.65950894355774

线程池并发执行耗时: 14.894720315933228

from multiprocessing.dummy import Pool as ThreadPool  
import time  
  
  
def work\_func(n):  
 print(f"this is work {n}")  
 time.sleep(1)  
  
  
def main():  
 start\_time = time.time()  
 for i in range(5):  
 work\_func(i)  
 time1 = time.time() - start\_time  
  
 # 开5个 worker，没有参数时默认是 cpu 的核心数  
 pool = ThreadPool(processes=5)  
 # 在线程中执行 work\_func 并返回执行结果  
 start\_time2 = time.time()  
 results2 = pool.map(work\_func, range(5))  
 pool.close()  
 pool.join()  
 time2 = time.time() - start\_time2  
 print("单线程顺序执行耗时:", time1)  
 print("线程池并发执行耗时:", time2)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

对于I/O密集型任务，线程池有优势

单线程顺序执行耗时: 5.049410104751587

线程池并发执行耗时: 1.00626540184021

**★★多线程\_线程池ThreadPoolExecutor**

import time  
from concurrent.futures import ThreadPoolExecutor  
  
def func1(var):  
 time.sleep(10)  
 print(f"thread {var} is finished")  
  
thread\_pool\_1 = ThreadPoolExecutor(max\_workers=10)  
for i in range(10):  
 task = thread\_pool\_1.submit(func1, i) # 向线程池提交一个作业（作业函数，传给作业函数的参数）  
 print(f"task {i} is done? {task.done()}") # .done()方法返回当前任务是否完成，True表示已完成  
  
thread\_pool\_1.shutdown(wait=False) # wait=False时，此方法会立即返回，而不会等待任务完成，默认就是False  
print("end 主线程退出了") # 主线程退出了，但如果线程池里还有任务未完成，也不会终止它，而是会让它们正常执行完成

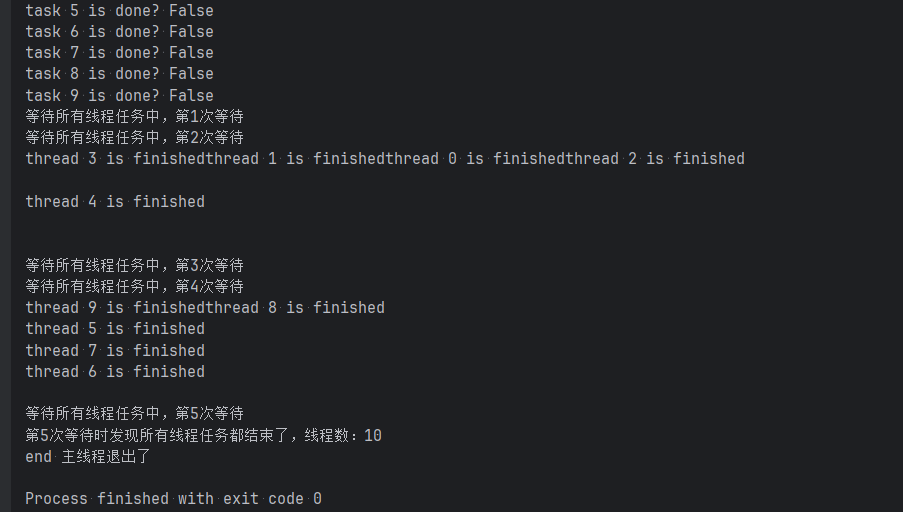
结果：



**需要等待所有线程任务结束，再退出主线程：**

import time  
from concurrent.futures import ThreadPoolExecutor  
  
def func1(var):  
 time.sleep(10)  
 print(f"thread {var} is finished")  
  
thread\_pool\_1 = ThreadPoolExecutor(max\_workers=5)  
thread\_task\_list = []  
for i in range(10):  
 task = thread\_pool\_1.submit(func1, i) # 向线程池提交一个作业（作业函数，传给作业函数的参数）  
 thread\_task\_list.append(task)  
 print(f"task {i} is done? {task.done()}") # .done()方法返回当前任务是否完成，True表示已完成  
  
thread\_pool\_1.shutdown(wait=False) # wait=False时，此方法会立即返回，而不会等待任务完成，默认就是False  
thread\_task\_len = len(thread\_task\_list)  
wait\_count = 1  
while True:  
 finished\_task\_count = 0  
 print(f"等待所有线程任务中，第{wait\_count}次等待")  
 for task in thread\_task\_list:  
 if task.done() is False:  
 break # 跳出 for循环，只要发现有一个任务未完成，则跳出for循环，然后此时 finished\_task\_count 肯定是小于 thread\_task\_len 的，需要等待几秒  
 else:  
 finished\_task\_count += 1  
 if finished\_task\_count == thread\_task\_len:  
 print(f"第{wait\_count}次等待时发现所有线程任务都结束了，线程数：{thread\_task\_len}")  
 break # 跳出 while循环  
 else:  
 time.sleep(5)  
 wait\_count += 1  
  
print("end 主线程退出了") # 主线程退出了，但如果线程池里还有任务未完成，也不会终止它，而是会让它们正常执行完成

结果：



**线程池的使用说明：**

线程池的基类是 concurrent.futures 模块中的Executor类，Executor有两个子类，ThreadPoolExecutor和ProcessPoolExecutor，其中ThreadPoolExecutor用于创建线程池，而ProcessPoolExecutor用于创建进程池。

如果使用线程池来管理并发编程，那么只要将相应的task函数提交给线程池，剩下的事情就由线程池来处理。

Exectuor提供了如下常用方法：

|  |  |
| --- | --- |
| submit(func, \*args, \*\*kwargs) | 将func函数提交给线程池。\*args代表传给func函数的参数，\*kwargs代表以关键字参数的形式为fn函数传入参数。 |
| map(func, \*iterables, timeout=None, chunksize=1) | 该函数类似于全局函数map(func, \*iterables)，会启动多个线程，以异步方式立即对iterables执行map处理。 |
| shutdown(wait=True) | 关闭线程池（wait=True时才会等待所有线程结束），否则不等待 |

程序将作业函数func提交（submit）给线程池后，submit()方法会返回一个Future对象（相当于一个任务task），Future类主要用于获取线程任务函数的返回值。由于线程任务会在新线程中以异步方式执行，因此，线程执行的函数相当于一个“将来完成”的任务，所以使用Future来表示。

Future方法：

|  |  |
| --- | --- |
| cancel() | 取消该Future代表的线程任务。如果该任务正在执行，不可取消，则该方法返回False；否则，程序会取消该任务，并返回True。 |
| cancelled() | 返回Future代表的线程任务是否被成功取消 |
| running() | 如果该Future代表的线程任务正在执行、不可被取消，该方法返回True |
| done() | 如果该Funture代表的线程任务被成功取消或执行完成，则该方法返回True |
| result(timeout=None) | 获取该Future代表的线程任务最后返回的结果。如果Future代表的线程任务还未完成，该方法将会阻塞当前线程，其中timeout参数指定最多阻塞多少秒 |
| exception(timeout=None) | 获取该Future代表的线程任务所引发的异常。如果该任务成功完成，没有异常，则该方法返回None |
| add\_done\_callback(fn) | 为该Future代表的线程任务注册一个“回调函数”，当该任务成功完成时，程序会自动触发该fn函数 |

在用完一个线程池后，应该调用该线程池的shutdown()方法，该方法将启动线程池的关闭序列。调用shutdown()方法后的线程池不再接收新任务，但会将以前所有的已提交任务执行完成。当线程池中的所有任务都执行完成后，该线程池中的所有线程都会死亡。

**★多进程和多线程性能对比**

python多线程适用于I/O密集型的任务

python多进程适用于计算密集型的任务

**计算密集型任务-多进程（快）**

from multiprocessing import Process  
import os  
import time  
  
  
# 计算密集型任务  
def work\_func():  
 res = 0  
 for i in range(100000000):  
 res \*= i  
  
  
def main():  
 process\_list = []  
 print("本机cpu为", os.cpu\_count(), "核")  
 start = time.time()  
 for i in range(4):  
 p = Process(target=work\_func) # 创建一个进程，将目标函数放入  
 process\_list.append(p)  
 p.start()  
 for p in process\_list:  
 p.join()  
 stop = time.time()  
 print("计算密集型任务，多进程耗时 %s" % (stop - start))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

本机cpu为 8 核

计算密集型任务，多进程耗时 4.6597900390625

**计算密集型任务-多线程（慢）**

from threading import Thread  
import os  
import time  
  
  
# 计算密集型任务  
def work\_func():  
 res = 0  
 for i in range(100000000):  
 res \*= i  
  
  
def main():  
 thread\_list = []  
 print("本机cpu为", os.cpu\_count(), "核")  
 start = time.time()  
 for i in range(4):  
 t = Thread(target=work\_func) # 创建一个线程，将目标函数放入  
 thread\_list.append(t)  
 t.start()  
 for t in thread\_list:  
 t.join()  
 stop = time.time()  
 print("计算密集型任务，多线程耗时 %s" % (stop - start))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

本机cpu为 8 核

计算密集型任务，多线程耗时 12.218607425689697

**I/O密集型任务-多进程（慢）**

from multiprocessing import Process  
import os, time  
  
  
# I/0密集型任务  
def work\_func():  
 print("===>", file=open("tmp.txt", "w"))  
  
  
def main():  
 process\_list = []  
 print("本机为", os.cpu\_count(), "核 CPU")  
 start = time.time()  
 for i in range(400):  
 p = Process(target=work\_func) # 多进程  
 process\_list.append(p)  
 p.start()  
 for p in process\_list:  
 p.join()  
 stop = time.time()  
 print("I/0密集型任务，多进程耗时 %s" % (stop - start))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

本机为 8 核 CPU

I/0密集型任务，多进程耗时 9.766752481460571

**I/O密集型任务-多线程（快）**

from threading import Thread  
import os, time  
  
  
# I/0密集型任务  
def work\_func():  
 print("===>", file=open("tmp.txt", "w"))  
  
  
def main():  
 thread\_list = []  
 print("本机为", os.cpu\_count(), "核 CPU")  
 start = time.time()  
  
 for i in range(400):  
 t = Thread(target=work\_func) # 多线程  
 thread\_list.append(t)  
 t.start()  
 for t in thread\_list:  
 t.join()  
 stop = time.time()  
 print("I/0密集型任务，多线程耗时 %s" % (stop - start))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

本机为 8 核 CPU

I/0密集型任务，多线程耗时 0.07439279556274414

**★结束线程**

import ctypes  
  
def cofable\_stop\_thread(thread):  
 *"""  
 结束线程，如果线程里有time.sleep(n)之类的操作，则需要等待这个时长之后，才会结束此线程  
 即此方法无法立即结束sleep及其他阻塞函数导致的休眼线程，得等线程获得响应时才结束它  
 raises the exception, performs cleanup if needed  
 注意：本函数会抛出一个SystemError异常，外部调用时需要处理此异常  
 """* if thread is None:  
 raise ValueError("cofable\_stop\_thread: thread obj is None")  
 thread\_id = ctypes.c\_long(thread.ident)  
 res = ctypes.pythonapi.PyThreadState\_SetAsyncExc(thread\_id, ctypes.py\_object(SystemExit))  
 # 正常结束线程时会返回数值1  
 if res == 0:  
 raise ValueError("cofable\_stop\_thread: invalid thread id")  
 elif res != 1:  
 # 如果返回的值不为0，也不为1，则 you're in trouble  
 # if it returns a number greater than one, you're in trouble,  
 # and you should call it again with exc=NULL to revert the effect  
 ctypes.pythonapi.PyThreadState\_SetAsyncExc(thread\_id, None)  
 raise SystemError("cofable\_stop\_thread: PyThreadState\_SetAsyncExc failed")  
  
  
def cofable\_stop\_thread\_silent(thread):  
 *"""  
 结束线程，如果线程里有time.sleep(n)之类的操作，则需要等待这个时长之后，才会结束此线程  
 即此方法无法立即结束sleep及其他阻塞函数导致的休眼线程，得等线程获得响应时才结束它  
 本函数不会抛出异常  
 """* if thread is None:  
 print("cofable\_stop\_thread\_silent: thread obj is None")  
 return  
 thread\_id = ctypes.c\_long(thread.ident)  
 res = ctypes.pythonapi.PyThreadState\_SetAsyncExc(thread\_id, ctypes.py\_object(SystemExit))  
 # 正常结束线程时会返回数值1  
 if res == 0:  
 print("cofable\_stop\_thread\_silent: invalid thread id")  
 elif res == 1:  
 print("cofable\_stop\_thread\_silent: thread stopped")  
 else:  
 # 如果返回的值不为0，也不为1，则 you're in trouble  
 # if it returns a number greater than one, you're in trouble,  
 # and you should call it again with exc=NULL to revert the effect  
 ctypes.pythonapi.PyThreadState\_SetAsyncExc(thread\_id, None)  
 print("cofable\_stop\_thread\_silent: PyThreadState\_SetAsyncExc failed")

**★第17章、网络通信**

**★将域名解析为IP地址**

**①gethostbyname**

import socket  
  
def get\_ip\_by\_domain(domain):  
 ip = socket.gethostbyname(domain)  
 return ip  
  
print(get\_ip\_by\_domain("www.cof-lee.com"))

结果：（返回一个ip地址，不论域名对应几个ip，只返回其中1个ip）

8.134.203.157

**②getaddrinfo**

import socket  
  
domain = "www.cof-lee.com"  
  
try:  
 addr\_info = socket.getaddrinfo(domain, 0, socket.AF\_INET, socket.SOCK\_STREAM, socket.IPPROTO\_TCP, socket.AI\_ADDRCONFIG)  
 # 返回一个列表，每个元素都是一个tuple元组，包含地址信息  
 print(addr\_info)  
 first\_addr\_tuple = addr\_info[0] # 只取第1个ip信息tuple  
 # 从地址信息元组中获取IP地址  
 ip\_address = first\_addr\_tuple[4][0] # ip信息tuple是一个五元组(host, port)，第5个元素是一个tuple（ip, port），这里取ip  
 print(ip\_address)  
except socket.gaierror as e:  
 print(f"Error resolving domain {domain}: {e}")

结果：

[(<AddressFamily.AF\_INET: 2>, <SocketKind.SOCK\_STREAM: 1>, 6, '', ('10.99.1.69', 0)), (<AddressFamily.AF\_INET: 2>, <SocketKind.SOCK\_STREAM: 1>, 6, '', ('10.99.1.1', 0)), (<AddressFamily.AF\_INET: 2>, <SocketKind.SOCK\_STREAM: 1>, 6, '', ('10.99.1.233', 0))]

10.99.1.69

当域名对应多个IP地址时，getaddrinfo()会返回所有相关的地址信息，可以通过遍历addr\_info来获取所有解析值

如果域名不存在或网络有问题，getaddrinfo()会抛出socket.gaierror异常。应该捕获这个异常并适当处理它。

socket.gaierror: [Errno 11001] getaddrinfo failed

**③gethostbyname\_ex**

import socket  
  
domain = "www.cof-lee.com"  
  
name, aliaslist, addresslist = socket.gethostbyname\_ex(domain)

print("name:", name)  
print("aliaslist:", aliaslist)  
print("addresslist:", addresslist)

结果：

name: www.cof-lee.com

aliaslist: []

addresslist: ['10.99.1.1', '10.99.1.69', '10.99.1.233']

**★TCP通信**

**★tcp-server**

import socket  
  
host = "0.0.0.0"  
port = 1234  
sock1 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 创建socket，ipv4,tcp  
sock1.setsockopt(socket.IPPROTO\_TCP, socket.TCP\_MAXSEG, 1300) # 设置tcp\_mss为1300  
sock1.bind((host, port)) # 监听的服务端ip,port，为一个元组  
sock1.listen(5) # 等待客户端连接  
while True:  
 con, addr = sock1.accept()  
 print("客户端的地址为：", addr)  
 con.send("发给客户端的信息".encode("utf8")) # 只能发送bytes  
 con.close()  
sock1.close() # 关闭socket

**★tcp-client**

import socket  
  
host = "127.0.0.1"  
port = 1234  
sock1 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 创建socket，ipv4,tcp  
sock1.setsockopt(socket.IPPROTO\_TCP, socket.TCP\_MAXSEG, 1300) # 设置tcp\_mss为1300  
sock1.connect((host, port)) # 连接服务端ip,port，为一个元组  
recv = sock1.recv(1024) # 接收对端发来的数据，最多收1024字节，返回bytes  
sock1.close() # 关闭socket

**★UDP通信**

**★udp-server**

import socket  
  
addr\_ser = ("0.0.0.0", 1234)  
sock1 = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) # 创建socket，ipv4,udp  
sock1.bind(addr\_ser) # 监听的服务端ip,port，为一个元组  
while True:  
 data, addr = sock1.recvfrom(1024) # 接收客户端发来的数据，最多1024字节  
 print("客户端的地址为：", addr)  
 print("客户端发来的数据：", data)  
 sock1.sendto("发给客户端的信息".encode("utf8"), addr)  
sock1.close() # 关闭socket

**★udp-client**

import socket  
  
addr\_ser = ("127.0.0.1", 1234)  
sock1 = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) # 创建socket，ipv4,tcp  
sock1.sendto("发送给server的数据".encode("utf8"), addr\_ser)  
recv = sock1.recv(1024) # 接收对端发来的数据，最多收1024字节，返回bytes  
sock1.close() # 关闭socket

**★tcp-server与多线程**

import threading  
import socket  
import time  
  
sendtouser = [  
 "HTTP/1.1 200 OK",  
 f"Date: {time.strftime('%a, %d %b %Y %H:%M:%S %z')}",  
 "Connection: Keep-Alive",  
 "Keep-Alive: timeout=5, max=100",  
 "Server: test-server",  
 "Content-Length: 7",  
 "Content-Type: text/html; charset=UTF-8",  
 "",  
 "hello\r\n"  
]  
  
def my\_thread\_func(con, addr): # 定义线程函数  
 print("本服务端为：", con.getsockname()) # 监听的地址和端口  
 print("客户端的地址为：", addr) # 客户端的源地址和源端口  
 con.send("\r\n".join(sendtouser).encode("utf8"))  
 con.close()  
  
host = "0.0.0.0"  
port = 1234  
sock1 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 创建socket  
sock1.bind((host, port)) # 监听的服务端(ip,port)，为一个元组  
sock1.listen(5) # 等待客户端连接  
while True:  
 conn, address = sock1.accept()  
 try:  
 thread = threading.Thread(target=my\_thread\_func, args=(conn, address))  
 thread.start()  
 except Exception as e:  
 print(e)

**★icmp**

#!/usr/bin/env python3  
# coding=utf-8  
# module name: cofping  
# author: Cof-Lee <cof8007@gmail.com>  
# this module uses the GPL-3.0 open source protocol  
# update: 2024-11-19  
  
import array  
import ctypes  
import struct  
import time  
import socket  
import random  
import string  
import cofnet  
  
ICMP\_TYPE\_8\_ECHO\_REQUEST = 0x08  
ICMP\_TYPE\_0\_ECHO\_RESPOND = 0x00  
ICMP\_TYPE\_3\_DESTINATION\_UNREACHABLE = 3  
ICMP\_TYPE\_11\_TIME\_TO\_LIVE\_EXCEEDED = 11  
  
  
def stop\_thread\_silently(thread):  
 *"""  
 结束线程，如果线程里有time.sleep(n)之类的操作，则需要等待这个时长之后，才会结束此线程  
 即此方法无法立即结束sleep及其他阻塞函数导致的休眼线程，得等线程获得响应时才结束它  
 本函数不会抛出异常  
 """* if thread is None:  
 print("cofping.stop\_thread\_silently: thread obj is None")  
 return  
 thread\_id = ctypes.c\_long(thread.ident)  
 res = ctypes.pythonapi.PyThreadState\_SetAsyncExc(thread\_id, ctypes.py\_object(SystemExit))  
 # 正常结束线程时会返回数值1  
 if res == 0:  
 print("cofping.stop\_thread\_silently: invalid thread id")  
 elif res == 1:  
 print("cofping.stop\_thread\_silently: thread stopped")  
 else:  
 # 如果返回的值不为0，也不为1，则:  
 ctypes.pythonapi.PyThreadState\_SetAsyncExc(thread\_id, None)  
 print("cofping.stop\_thread\_silently: PyThreadState\_SetAsyncExc failed")  
  
  
class ResultOfPingOnePacket:  
 def \_\_init\_\_(self, respond\_source\_ip="", respond\_destination\_ip="", rtt\_ms=0.0, icmp\_data\_size=0, ttl=0, is\_success=False,  
 icmp\_type=0, icmp\_code=0, icmp\_checksum=0x0000, icmp\_id=0x0000, icmp\_sequence=0x0000, icmp\_data=b'',  
 received\_a\_respond=False, failed\_info=""):  
 self.respond\_source\_ip = respond\_source\_ip  
 self.respond\_destination\_ip = respond\_destination\_ip  
 self.rtt\_ms = rtt\_ms # RTT时间，单位：毫秒  
 self.icmp\_data\_size = icmp\_data\_size # icmp数据大小，单位：字节  
 self.ttl = ttl # ip报文里的ttl  
 self.is\_success = is\_success # 检测是否成功，成功则置为True  
 self.icmp\_type = icmp\_type  
 self.icmp\_code = icmp\_code  
 self.icmp\_checksum = icmp\_checksum  
 self.icmp\_id = icmp\_id  
 self.icmp\_sequence = icmp\_sequence  
 self.icmp\_data = icmp\_data # bytes类型数据  
 self.received\_a\_respond = received\_a\_respond  
 self.failed\_info = failed\_info # 如果检测不成功，必须提示失败信息  
  
  
class PingOnePacket:  
 *"""  
 单次ping检测，只会发送1个icmp\_echo\_request报文，然后等待回复  
 """* def \_\_init\_\_(self, target\_ip="", timeout=2, size=1, ttl=128, dont\_frag=False):  
 self.target\_ip = target\_ip # 目标ip（ipv4地址）  
 self.timeout = timeout # 超时，单位：秒  
 self.size = size # 发包数据大小，单位：字节，当整个报文长度小于mac帧长度要求时，会自动以0填充  
 self.ttl = ttl  
 self.dont\_frag = dont\_frag # 置True时不分片，置False时分片  
 self.result = ResultOfPingOnePacket()  
 self.is\_finished = False  
 self.icmp\_send\_type = ICMP\_TYPE\_8\_ECHO\_REQUEST # icmp\_echo\_request  
 self.icmp\_send\_code = 0x00  
 self.icmp\_send\_checksum = 0x0000  
 self.icmp\_send\_id = 0xFFFF & random.randint(0, 0xFFFF) # 为进程号，echo响应消息与echo请求消息中的id保持一致，取值随机  
 self.icmp\_send\_sequence = 0xFFFF & random.randint(0, 0xFFFF) # 序列号，echo响应消息与echo请求消息中的sequence保持一致，取值随机  
 self.icmp\_send\_data = b''  
 self.icmp\_send\_packet = b''  
 self.icmp\_socket = None  
 self.start\_time = 0.0  
 self.recv\_thread = None  
  
 def start(self):  
 self.icmp\_send\_packet = self.generate\_icmp\_packet()  
 # 创建icmp套接字  
 self.icmp\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_RAW, socket.IPPROTO\_ICMP)  
 self.icmp\_socket.settimeout(self.timeout) # 设置socket超时时间，当收到数据包后，会重置超时时间为指定的  
 self.icmp\_socket.setsockopt(socket.IPPROTO\_IP, socket.IP\_TTL, self.ttl) # 设置ip报文的ttl  
 if self.dont\_frag:  
 self.icmp\_socket.setsockopt(socket.IPPROTO\_IP, socket.IPV6\_DONTFRAG, 2) # 设置ip报文不分片，没有IPV4\_DONTFRAG常量，这就有点坑  
 self.start\_time = time.time()  
 try:  
 self.icmp\_socket.sendto(self.icmp\_send\_packet, (self.target\_ip, 0)) # ★发送请求报文  
 except OSError as err:  
 self.is\_finished = True  
 stop\_thread\_silently(self.recv\_thread)  
 self.result.is\_success = False  
 self.result.failed\_info = err.\_\_str\_\_()  
 return  
 self.recv\_icmp\_packet() # 接收报文，阻塞型  
 self.icmp\_socket.close()  
  
 @staticmethod  
 def generate\_icmp\_checksum(packet: bytes) -> int:  
 if len(packet) & 1: # 长度的末位为1表示：长度不是2的倍数（即最后一bit不为0），则：  
 packet = packet + b'\x00' # 需要以0填充  
 words = array.array('h', packet)  
 checksum = 0  
 for word in words:  
 checksum += (word & 0xffff)  
 while checksum > 0xFFFF:  
 checksum = (checksum >> 16) + (checksum & 0xffff) # checksum只能为2字节，溢出部分需要继续进行+运算，直到不溢出为止  
 return (~checksum) & 0xffff # 反回2字节校验和的反码  
  
 def generate\_icmp\_packet(self) -> bytes:  
 self.icmp\_send\_data = "".join(random.SystemRandom().choice(string.ascii\_letters) for \_ in range(self.size)).encode('utf8')  
 # 字节序默认跟随系统，x86\_64为LE小端字节序  
 icmp\_temp\_header = struct.pack('bbHHH', self.icmp\_send\_type, self.icmp\_send\_code, self.icmp\_send\_checksum,  
 self.icmp\_send\_id, self.icmp\_send\_sequence)  
 icmp\_temp\_packet = icmp\_temp\_header + self.icmp\_send\_data  
 self.icmp\_send\_checksum = self.generate\_icmp\_checksum(icmp\_temp\_packet)  
 icmp\_header = struct.pack('bbHHH', self.icmp\_send\_type, self.icmp\_send\_code, self.icmp\_send\_checksum,  
 self.icmp\_send\_id, self.icmp\_send\_sequence)  
 return icmp\_header + self.icmp\_send\_data  
  
 def recv\_icmp\_packet(self):  
 while True:  
 print(f"{self.target\_ip} 开始接收回包，")  
 used\_time = time.time() - self.start\_time  
 if used\_time >= self.timeout:  
 print(f"PingOnePacket.recv\_icmp\_packet: {self.target\_ip}接收超时了 {used\_time}")  
 self.result.is\_success = False  
 self.result.failed\_info = "timeout"  
 self.result.rtt\_ms = self.timeout \* 1000  
 self.is\_finished = True  
 return  
 try:  
 # recv\_packet, addr = self.icmp\_socket.recvfrom(65535) # ★★接收到整个ip报文，阻塞型函数  
 recv\_packet = self.icmp\_socket.recv(65535) # ★★接收到整个ip报文，阻塞型函数  
 except Exception as e: # 超时会报异常  
 print("\nPingOnePacket.recv\_icmp\_packet: 接收报异常超时了", e)  
 self.result.is\_success = False  
 self.result.failed\_info = "timeout"  
 self.result.rtt\_ms = self.timeout \* 1000  
 self.is\_finished = True  
 return  
 # 如果接收到报文了：  
 rtt\_s = time.time() - self.start\_time  
 ipv4\_header = recv\_packet[:20]  
 icmp\_header = recv\_packet[20:28]  
 icmp\_data = recv\_packet[28:]  
 ipv4\_struct\_tuple = struct.unpack("!BBHHHBBHII", ipv4\_header)  
 icmp\_type, icmp\_code, icmp\_checksum, icmp\_id, icmp\_sequence = struct.unpack("bbHHH", icmp\_header)  
 if icmp\_id == self.icmp\_send\_id and icmp\_sequence == self.icmp\_send\_sequence and icmp\_type != ICMP\_TYPE\_8\_ECHO\_REQUEST:  
 self.result.received\_a\_respond = True  
 self.result.rtt\_ms = rtt\_s \* 1000  
 if icmp\_type == ICMP\_TYPE\_0\_ECHO\_RESPOND and icmp\_code == 0x00:  
 self.result.is\_success = True  
 else:  
 self.result.is\_success = False  
 self.result.failed\_info = self.generate\_icmp\_failed\_info(icmp\_type, icmp\_code)  
 self.result.ttl = ipv4\_struct\_tuple[5]  
 # self.result.respond\_source\_ip = addr[0] # 同 ipv4\_struct\_tuple[8]  
 self.result.respond\_source\_ip = cofnet.int32\_to\_ip(ipv4\_struct\_tuple[8])  
 self.result.respond\_destination\_ip = cofnet.int32\_to\_ip(ipv4\_struct\_tuple[9])  
 self.result.icmp\_data\_size = len(icmp\_data) # 大小为icmp数据部分的长度  
 self.result.icmp\_type = icmp\_type  
 self.result.icmp\_code = icmp\_code  
 self.result.icmp\_checkum = icmp\_checksum  
 self.result.icmp\_id = icmp\_id  
 self.result.icmp\_sequence = icmp\_sequence  
 self.result.icmp\_data = icmp\_data  
 self.is\_finished = True  
 return  
 elif icmp\_type == ICMP\_TYPE\_11\_TIME\_TO\_LIVE\_EXCEEDED: # 这种类型常见于tracepath中，由中间路由器返回的ttl超时消息  
 # 它本身是icmp报文，其icmp\_id和icmp\_sequence为空，其数据内容为 原数据包的ip报文（含ip报文中的icmp载荷）  
 carrier\_ipv4\_header = recv\_packet[28:48]  
 carrier\_icmp\_packet = recv\_packet[48:]  
 # carrier\_icmp\_header = recv\_packet[48:56]  
 # carrier\_icmp\_data = recv\_packet[56:]  
 carrier\_ipv4\_struct\_tuple = struct.unpack("!BBHHHBBHII", carrier\_ipv4\_header)  
 if carrier\_ipv4\_struct\_tuple[9] == cofnet.ip\_or\_maskbyte\_to\_int(  
 self.target\_ip) and carrier\_icmp\_packet == self.icmp\_send\_packet:  
 self.result.received\_a\_respond = True  
 self.result.rtt\_ms = rtt\_s \* 1000  
 self.result.is\_success = False  
 self.result.failed\_info = self.generate\_icmp\_failed\_info(icmp\_type, icmp\_code)  
 self.result.ttl = ipv4\_struct\_tuple[5]  
 self.result.respond\_source\_ip = cofnet.int32\_to\_ip(ipv4\_struct\_tuple[8])  
 self.result.respond\_destination\_ip = cofnet.int32\_to\_ip(ipv4\_struct\_tuple[9])  
 self.result.icmp\_data\_size = len(icmp\_data) # 大小为icmp数据部分的长度  
 self.result.icmp\_type = icmp\_type  
 self.result.icmp\_code = icmp\_code  
 self.result.icmp\_checkum = icmp\_checksum  
 self.result.icmp\_id = icmp\_id  
 self.result.icmp\_sequence = icmp\_sequence  
 self.result.icmp\_data = icmp\_data  
 self.is\_finished = True  
 return  
 time\_left = self.timeout - rtt\_s  
 if time\_left > 0:  
 self.icmp\_socket.settimeout(time\_left)  
  
 @staticmethod  
 def generate\_icmp\_failed\_info(icmp\_type, icmp\_code) -> str:  
 if icmp\_type == ICMP\_TYPE\_3\_DESTINATION\_UNREACHABLE: # ★终点不可达  
 if icmp\_code == 0:  
 failed\_info = f"终点不可达-->network\_unreachable icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 1:  
 failed\_info = f"终点不可达-->host\_unreachable icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 2:  
 failed\_info = f"终点不可达-->protocol\_unreachable icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 3:  
 failed\_info = f"终点不可达-->port\_unreachable icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 4:  
 failed\_info = f"终点不可达-->fragmentation\_required icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 5:  
 failed\_info = f"终点不可达-->source\_route\_failed icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 6:  
 failed\_info = f"终点不可达-->network\_unknown icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 7:  
 failed\_info = f"终点不可达-->host\_unknown icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 8:  
 failed\_info = f"终点不可达-->source\_host\_isolated icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 9:  
 failed\_info = f"终点不可达-->network\_administratively\_prohibited icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 10:  
 failed\_info = f"终点不可达-->host\_administratively\_prohibited icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 11:  
 failed\_info = f"终点不可达-->network\_unreachable\_tos icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 12:  
 failed\_info = f"终点不可达-->host\_unreachable\_tos icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 13:  
 failed\_info = f"终点不可达-->communication\_administratively\_prohibited icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 14:  
 failed\_info = f"终点不可达-->host\_precedence\_violation icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 13:  
 failed\_info = f"终点不可达-->precedence\_cutoff icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 else:  
 failed\_info = f"终点不可达-->UNKNOWN\_CODE icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_type == 4: # ★源点不可达  
 failed\_info = f"源点不可达 icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_type == 5: # ★路由重定向  
 if icmp\_code == 0:  
 failed\_info = f"路由重定向-->for\_network icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 1:  
 failed\_info = f"路由重定向-->for\_host icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 2:  
 failed\_info = f"路由重定向-->for\_tos\_and\_network icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 3:  
 failed\_info = f"路由重定向-->for\_tos\_and\_host icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 else:  
 failed\_info = f"路由重定向-->UNKNOWN\_CODE icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_type == ICMP\_TYPE\_11\_TIME\_TO\_LIVE\_EXCEEDED: # ★时间超时  
 if icmp\_code == 0:  
 failed\_info = f"时间超时--ttl\_超时\_传输过程中减为0了 icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 1:  
 failed\_info = f"时间超时--分片重组超时 icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 else:  
 failed\_info = f"时间超时--UNKNOWN\_CODE icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_type == 12: # ★IP头参数问题  
 if icmp\_code == 0:  
 failed\_info = f"IP头参数问题-->pointer\_indicates\_error icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 1:  
 failed\_info = f"IP头参数问题-->missing\_required\_option icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 elif icmp\_code == 2:  
 failed\_info = f"IP头参数问题-->bad\_length icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 else:  
 failed\_info = f"IP头参数问题-->UNKNOWN\_CODE icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 else: # ★未知错误类型  
 failed\_info = f"未知错误类型 icmp\_type={icmp\_type} icmp\_code={icmp\_code}"  
 return failed\_info

**★第18章、加密解密操作**

**★Hash计算**

import hashlib  
  
data = "hello".encode("utf8")  
hasher = hashlib.sha256() # 支持 md5, sha1, sha256, sha384, sha512  
hasher.update(data)  
print(hasher.hexdigest())

结果：

2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

**★DES**

Data Encryption Standard

**★AES**

AES（Advanced Encryption Standard）有3种密码长度：

128bit（16字节），192bit（24字节），256bit（32字节）

分组密码模式主要有5种：

|  |  |  |
| --- | --- | --- |
| ECB | Electronic CodeBook mode | 电子密码模式 |
| CBC | Cipher Block Chaining mode | 密码分组链式模式 |
| CFB | Cipher FeedBack mode | 密文反馈模式 |
| OFB | Output FeedBack mode | 输出反馈模式 |
| CTR | CounTeR mode | 计算器模式 |

常用模式是 CBC和CTR，这2种模式都需要一个初始化向量（IV），使用初始化向量对明文分组进行异或处理，再进行加密

AES加密算法处理的 明文分组长度都是16字节，即IV初始化向量也是16字节

密钥则可以是128bit，192bit或256bit。

**示例：**AES\_256\_CBC

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  
from cryptography.hazmat import backends  
from cryptography.hazmat.primitives import padding  
  
  
def encrypt\_message(plain\_data: bytes, key):  
 # iv = os.urandom(16) # 生成一个随机的初始化向量  
 iv\_str = 'a5f4cfeb673cf005444e19925f36c0b9'  
 iv = bytes.fromhex(iv\_str) # hex字符串转为bytes  
 # 使用AES算法和CBC模式  
 cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backends.default\_backend())

# modes.CBC(iv)表示使用CBC模式，也可使用其他模式，比如 modes.CTR(iv)  
 encryptor = cipher.encryptor()  
  
 # 添加填充以适应块大小  
 padder = padding.PKCS7(algorithms.AES.block\_size).padder()  
 padded\_data = padder.update(plain\_data) + padder.finalize()  
  
 # 加密数据  
 ciphertext = encryptor.update(padded\_data) + encryptor.finalize()  
 return iv + ciphertext # 返回IV和密文  
  
  
def decrypt\_message(encrypted\_data, key):  
 # 从密文中提取IV  
 iv = encrypted\_data[:16]  
 ciphertext = encrypted\_data[16:]  
  
 # 使用相同的密钥和IV进行解密  
 cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backends.default\_backend())  
 decryptor = cipher.decryptor()  
  
 # 解密数据并去除填充  
 padded\_data = decryptor.update(ciphertext) + decryptor.finalize()  
 unpadder = padding.PKCS7(algorithms.AES.block\_size).unpadder()  
 decrypted\_data = unpadder.update(padded\_data) + unpadder.finalize()  
 return decrypted\_data  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 示例密钥（必须对齐到16字节或24字节或32字节长度）  
 hex\_str = "1bd3e3de1e60d7379daa028dac2ea9afec8721307390a2892ac6663335e6c3c6" # 这个是32字节长度  
 key = bytes.fromhex(hex\_str) # hex字符串转为bytes  
 # key = os.urandom(32)  
  
 plain\_text = """  
{  
 "name": "cof",  
 "pass": "1234567"  
}"""  
 plain\_data = plain\_text.encode("utf8")  
 cipher\_data = encrypt\_message(plain\_data, key)  
  
 with open("passwd.encrypt", "wb+") as file:  
 file.write(cipher\_data)  
 print("Encrypted:", cipher\_data)  
  
 with open("passwd.encrypt", "rb") as file:  
 cipher\_dataxx = file.read()  
 decrypted = decrypt\_message(cipher\_dataxx, key).decode("utf8")  
 print("Decrypted:", decrypted)

结果：

Encrypted: b'\xa5\xf4\xcf\xebg<\xf0\x05DN\x19\x92\_6\xc0\xb9\x8e\x7f\xba<\x03\x8ci4+\x12\x9b\xc5\xf7\xfdz\xc05P\xf33\xcc\x9e\x15E>\x1fBE>PvP\xb6\xcd\xea\x03\xd3"0\xa2s!N\x8e\xda\x07%\x99'

Decrypted:

{

"name": "cof",

"pass": "1234567"

}

**★DSA**

Digital Signature Algorithm

理论基础：基于离散对数问题的困难性，即在有限域上求解离散对数是困难的

用途：主要用于数字签名，不常用于数据加密

**★RSA**

Rivest-Shamir-Adleman

理论基础：基于大整数分解的困难性，即，给定一个大整数和其质因数分解非常困难

用途：可用于数据加密和数字签名

from cryptography.hazmat.primitives.asymmetric import rsa, padding  
from cryptography.hazmat import backends  
from cryptography.hazmat.primitives import serialization, hashes  
from cryptography import exceptions  
  
  
def encrypt\_message(plain\_data: bytes, public\_key: rsa.RSAPublicKey) -> bytes:  
 encrypted\_data = public\_key.encrypt(  
 plain\_data,  
 padding.OAEP( # 使用OAEP填充，Optimal Asymmetric Encryption Padding 此填充方式仅用于加密解密  
 mgf=padding.MGF1(algorithm=hashes.SHA256()), # Mask Generation Fucntion 掩码生成函数  
 algorithm=hashes.SHA256(),  
 label=None  
 )  
 )  
 return encrypted\_data  
  
  
def decrypt\_message(encrypted\_data: bytes, private\_key: rsa.RSAPrivateKey) -> bytes:  
 decrypt\_data = private\_key.decrypt(  
 encrypted\_data,  
 padding.OAEP(  
 mgf=padding.MGF1(algorithm=hashes.SHA256()),  
 algorithm=hashes.SHA256(),  
 label=None  
 )  
 )  
 return decrypt\_data  
  
  
def sign\_data(plain\_data: bytes, private\_key: rsa.RSAPrivateKey) -> bytes:  
 signature = private\_key.sign(  
 plain\_data,  
 padding.PSS( # 使用PSS填充，Probabilistic Signature Scheme 此填充方式仅用于签名验签  
 mgf=padding.MGF1(hashes.SHA256()),  
 salt\_length=padding.PSS.MAX\_LENGTH  
 ),  
 hashes.SHA256()  
 )  
 return signature  
  
  
def verify\_sign\_ok(message: bytes, signature: bytes, public\_key: rsa.RSAPublicKey) -> bool:  
 try:  
 public\_key.verify(  
 signature,  
 message,  
 padding.PSS(  
 mgf=padding.MGF1(hashes.SHA256()),  
 salt\_length=padding.PSS.MAX\_LENGTH  
 ),  
 hashes.SHA256()  
 )  
 return True  
 except exceptions.InvalidSignature:  
 # print("难签失败")  
 return False  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 private\_key = rsa.generate\_private\_key(public\_exponent=65537, # 标准公开指数  
 key\_size=2048, # 密钥长度，2048，3072，4096 bit  
 backend=backends.default\_backend())  
 public\_key = private\_key.public\_key() # 从私钥里导出公钥，（私钥是包含公钥的）  
 private\_key\_bytes = private\_key.private\_bytes( # 生成的其实是文本内容 "-----BEGIN RSA PRIVATE KEY-----" 这种  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.TraditionalOpenSSL,  
 encryption\_algorithm=serialization.NoEncryption()  
 )  
 public\_key\_bytes = public\_key.public\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PublicFormat.SubjectPublicKeyInfo  
 )  
 with open("rsa\_private\_key\_test.pem", "wb") as f:  
 f.write(private\_key\_bytes)  
 with open("rsa\_public\_key\_test.pem", "wb") as f:  
 f.write(public\_key\_bytes)  
  
 with open("rsa\_private\_key\_test.pem", "rb") as key\_file:  
 private\_key2 = serialization.load\_pem\_private\_key(key\_file.read(), password=None)  
 # public\_key2 = private\_key2.public\_key() # 也可以从私钥里导出公钥  
  
 with open("rsa\_public\_key\_test.pem", "rb") as key\_file:  
 public\_key2 = serialization.load\_pem\_public\_key(key\_file.read())  
  
 message = "hello, world".encode("utf8")  
 signature = sign\_data(message, private\_key2) # 使用私钥进行签名，签名长度等于 密钥长度  
 print(f"签名长度：{len(signature)}字节，内容：{signature}")  
 if verify\_sign\_ok(message, signature, public\_key2):  
 print("验签成功")  
 else:  
 print("验签失败")  
  
 private\_key3\_str = """  
-----BEGIN RSA PRIVATE KEY-----  
MIIEogIBAAKCAQEA2lEbqCMJrCjA+vlk3Q9R7iRZUj+HryMt7WamopaJNydZNKJm  
MO4thOWBKuFl4vkDkNGJdAmvfrAVpDiZFd+RKwnrGD+AGUOSj2835iFFr+MlR8s2  
T/b28FAM5YXHXQz+yba5go92fIVM+nNPMeSIfzZaAqLr+vygPt4zWBvQxC1Q/pVO  
CAT3lEw7LCx1XGiP5af0gbxf2rLbvjA8Y8NuYI3X0IpgGQAb/q7qovFCF16A8YL9  
R2uUT/oRr136CG1u9yWA3RN41hXCcsZou1xtB8b8nPubpSP7ZLBY+ka6NphkCKgM  
t+S7n1fu0HOrLxGRy8KzPjRFKrpqN4OMhpDtcwIDAQABAoIBABfSbDP7aBr0Ms6P  
97y+vGAXG59ZFaX7hDpDOQvvuixIVmMNv3HiY27SGOHuyLkswSyvDefEBiAgTamR  
kB7DZOnvLKCE4/udvyMjQym7grOlE6nTAkD/1UU2ru7n8JpLiI0b1lU+m0gOHEQN  
4LE0hd0VDLBR9RxPhEV9Oy6yO6RUNoKQ7TjQjanyqM+MH7IObUKuchAlnJYi6BP0  
9pnHf/ldUR2Q5vbl75nQ83skiiV/MWBlgFw/jjwJ+/W6OeGHAj5CzW6L4iVULzVl  
zCF7A8vZn9Npq9HXv0qpofQgOoaKBjjXcbG+z13vzf+KXHU/r3hjFL+nCmhKO2j6  
8BdRCf0CgYEA/KIRk9pybHcPnljqNdlEN+E38fUhpX5fLagYDXs9/2Bg64HfZt7J  
RC5912CptqE1MbyT4aVwIRlhrWzlYLqK2+5owxfr6nT5zDxZi57dY6HJdagckadq  
Vp5KIRpTey/xMamQIVrMkdsTsbzxhWuiS3IpkQ8skANmNECshSFqgLUCgYEA3Tn1  
oJdpj5/v2gbsku2MVrBdGws/K9m3ggwa8ZCpmUiMoXUGcXacEBpk1DoA20XPO7w4  
ae0+zJ9KquBSwAjyhNAB1k8xkR3ERDJFvbZRCbCLEtNJWhDJD9zv43uc4JMKYPGm  
5Ak+F+yd+bcYEKsUp8XKeIJRCdWxGCqaljcplocCgYBuCSVA0ffeuGkMHkw9Ehn9  
gNI+R/fUvls0Zq+7R3+0eMUeLt4gT084rEtRgR/KlSK3PBzpPoa9U/HQeMrbu8R5  
kLj+qK+v2i1nQK37+0EQgUpfPc7bviC6A43kJzRPRo12hRVrAcJykQpd8x77buss  
k1f5f7N4MpLUlYrPHcPmtQKBgAtUGHvsxP8R1Zp4mEy91V3rEhl8bfO05mzGaerM  
2aJ70JfEAlx1EnDQqTEk9z5DSOVqMglamMipr40oh5RF4QVkEXQEAinIOHC1klSw  
rV2tqLGCGFOAYItraO1hOxtU7EcimMU8o+1dtIlBEv5nhFtDCac6JSQ3ySXPqLLg  
iESpAoGAFSmNPpIBuBYoUvdhuXvxZT9yEKGIPjUplGxZUI6D/bWo6z4GUIQm7/XI  
kIIeC6oOHAhzVrMZ1Abc3tNG6F+3tVMUb/QaCcFin0KHvnRcQsTVxhU0564+FdKV  
mcdLF8tKCFryvbQLuN49a530GrFUDNGaM9HCOvKC7yr8xCFuoJA=  
-----END RSA PRIVATE KEY-----"""  
 public\_key3\_str = """  
-----BEGIN PUBLIC KEY-----  
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2lEbqCMJrCjA+vlk3Q9R  
7iRZUj+HryMt7WamopaJNydZNKJmMO4thOWBKuFl4vkDkNGJdAmvfrAVpDiZFd+R  
KwnrGD+AGUOSj2835iFFr+MlR8s2T/b28FAM5YXHXQz+yba5go92fIVM+nNPMeSI  
fzZaAqLr+vygPt4zWBvQxC1Q/pVOCAT3lEw7LCx1XGiP5af0gbxf2rLbvjA8Y8Nu  
YI3X0IpgGQAb/q7qovFCF16A8YL9R2uUT/oRr136CG1u9yWA3RN41hXCcsZou1xt  
B8b8nPubpSP7ZLBY+ka6NphkCKgMt+S7n1fu0HOrLxGRy8KzPjRFKrpqN4OMhpDt  
cwIDAQAB  
-----END PUBLIC KEY-----"""  
 private\_key3 = serialization.load\_pem\_private\_key(private\_key3\_str.encode("utf8"), password=None)  
 public\_key3 = serialization.load\_pem\_public\_key(public\_key3\_str.encode("utf8"))  
  
 encrypted\_data = encrypt\_message(message, public\_key3) # 使用公钥进行加密  
 print(f"加密后的数据长度：{len(encrypted\_data)}字节，加密数据：{encrypted\_data}") # 加密后的数据长度是 密钥长度 的整数倍  
 decrypt\_data = decrypt\_message(encrypted\_data, private\_key3) # 使用私钥进行解密  
 print(f"解密后的数据：{decrypt\_data}")

结果：

签名长度：256字节，内容：

b'W\xa5\\\x95)\xc9\x86\xeb\xd4\xe3\xf6q\xc8\x05\x88\x0f\xb8Z7\xc4S\x87\xc4\xa6\x0c-\x00\xfb\xad\xb9\x17<\x17\xd2\xfb\xd1\x0ed\x1b.M\xf7;\xd5\xf7\x94 $\x00\x1e\x0bl\xc3\x01\xe0"On\xb3$d\x00\xc8fw\xa4\x19\xbc:\xb0^\xf4\xdc\\>\x91\x883\xb2o\xb0b(\_w(\xf7)\xe56p4J\x9e\xa5N\xb3z\x9e\x93\xfe\xc0uo;s[\xee\'dztZ\xe5uI\xc9\xe4H\x05\x92\xf6\xa5\x07J)H\xa7R\xf2\xae\xd7\x94IQ\xec\xd3\xaf\*\x1a\xc9\\\xa4\xb2%[z\x05u\xde\xa9\xab\x8f\x86\xf4\x85mA\xdbmNW\xee2\x8d\xbe`\x8a\x1fG\xc6\xed\xd2\x1aZ\*\x7f\x931s\x0f\xd7.\xd8\xe8\xa8\xb9E\x08\xa7\xa0\x9a\xa3\x8b4\xd5\xcb\x0c-IDg\xd5\xf0\x9d#\x07\xb5\x82\x8d\x02R-\xd3\x82\xe3\xd8V\_N\xfd\xd8W\xads\xf5\xf8\xd1\x1b0\xb86=\xcbfB\x0e\_\xce/\x0eb\xf7&\xcf\x05:"\x909\x17\x87\xdf\xaf\x97\''

验签成功

加密后的数据长度：256字节，加密数据：

b'\xa7m\xac\x16\xedp\x89\x98\x90\xf5;f\xf6\xa1\xb8\xb3\xa7W\*w=\x99.z\xbb\x87D\xd5\xe4\x08gS\xc5q\xc1\xc8x\xa7F$\xdf)!\x1b]\x9d\x8d\x94\xa3\x8e\xe1\xca\xf3=)\xaf>\xf8\x07\x13\t\xe7\r\xd7\xa4t\x02\xd1\xd0.&\x19\xbc\xc4\xad\x19p\xb8\x06-\xee\x1e\xd0\xfa\xb6\\~\xefV\xa1\xd8\x079\xc1\xc7\xc6\x8d\x98\x86\x8e:\x7fE:P\x11],\xe0\x1c`\xeb\xbb\xb9\xaf<\x0b\xe8\xb7\x12c\x94\x1b\x8dV\xd9e$?\x19\xcb\xc2\x06\x9a\x05\xdc \x9dt\x05#\x0c\xf2\xe2\x9dB\x81\x97r\xf6Dy\xea\_7\x9c\xd9\xfe\x88\x8e\xf8\xc1&f2\xa2\x00\xceD/\xe3/>iAa\x9567\xd4\xd9&\x86\x94\xdb+5\x03\xc7h\x98K@\xc3\xdeQ\xe6\x9f\xacf\x16\xc1\x1a\xf1\xfa\x87[;\xc1\x81\xbe\x9b&Z?\x91\x97\xfc\xe5X~\x1f\xf9w\x89Cpx7\xed\x87\xe9\xa2\x16(\xac\xec\xa7\xeaW\x95\xb3:\xf7<&\x97\x1a\xe0\x1f\xe9U\xacN\x02\xa1'

解密后的数据：b'hello, world'

**★ECC**

Elliptic Curve Cryptography

椭圆曲线密码（ECC）是基于椭圆曲线数学理论构建的非对称加密技术，其安全性依赖于椭圆曲线离散对数问题（ECDLP）的数学困难性。通过点乘运算实现密钥生成与数据加密，ECC在相同安全强度下密钥长度显著短于RSA等传统算法，适用于资源受限环境，该技术广泛应用于区块链（如比特币secp256k1曲线）

相较于RSA算法，234位ECC密钥可提供与2048位RSA等效的安全性，密钥长度缩短约87%。这一特性使其在移动设备、物联网终端等计算资源受限场景中具有显著优势。实际测试显示，ECC的签名生成速度比RSA快15倍以上，验证速度快约5倍。

中国SM2算法采用256位椭圆曲线参数，于2010年纳入国家商用密码标准。

国际标准化组织（ISO）将ECC纳入TLS 1.3协议强制支持算法列表

美国NIST推荐15条标准曲线（如secp256r1）用于政府机构通信加密。

基于有限域的椭圆曲线算法，椭圆曲线密码（ECC）提供多种不同的算法，包括

数字签名算法：

ECDSA 椭圆曲线数字签名算法

EdDSA 爱德华曲线数字签名算法

密钥协商算法：

ECDH 椭圆曲线迪弗赫尔曼

FHMQV 全哈希Menezes-Qu-Vanstone

椭圆曲线算法：

ECIES 椭圆曲线集成加密方案

ElGamal 基于EC的离散对数椭圆曲线密码

256位密钥的ECC 相当于 2072位的RSA 安全强度

推荐曲线： NIST P-256（secp256r1）

from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives.asymmetric import ec  
from cryptography.hazmat.primitives import serialization  
  
# 生成ECC密钥对  
private\_key = ec.generate\_private\_key(ec.SECP256R1(), default\_backend())  
public\_key = private\_key.public\_key()  
  
private\_key\_pem = private\_key.private\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.TraditionalOpenSSL,  
 encryption\_algorithm=serialization.NoEncryption()  
)  
public\_key\_pem = public\_key.public\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PublicFormat.SubjectPublicKeyInfo  
)  
# 输出密钥信息  
print("私钥内容：", private\_key\_pem.decode("utf8"))  
print("公钥内容：", public\_key\_pem.decode("utf8"))

结果：

私钥内容： -----BEGIN EC PRIVATE KEY-----

MHcCAQEEIKS6peHSJEsMxsm9gQmGXGq5fzzqJOGIIWhCXBl9iMaMoAoGCCqGSM49

AwEHoUQDQgAEroFEiHmPZuwiSFBatOUSv7Uvb5IlP2cKnMtBifIE8EyvyoAJD138

BXxBsBlfxRjZXrtgSnjlnwV6uBEg75HU7Q==

-----END EC PRIVATE KEY-----

公钥内容： -----BEGIN PUBLIC KEY-----

MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEroFEiHmPZuwiSFBatOUSv7Uvb5Il

P2cKnMtBifIE8EyvyoAJD138BXxBsBlfxRjZXrtgSnjlnwV6uBEg75HU7Q==

-----END PUBLIC KEY-----

**★Ed25519（EdDSA）**

tls 1.3常用

from cryptography.hazmat.primitives.asymmetric import ed25519  
from cryptography.hazmat.primitives import serialization  
from cryptography import exceptions  
  
# 生成ECC密钥对  
private\_key = ed25519.Ed25519PrivateKey.generate()  
public\_key = private\_key.public\_key()  
  
private\_key\_pem = private\_key.private\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.PKCS8,  
 encryption\_algorithm=serialization.NoEncryption()  
)  
public\_key\_pem = public\_key.public\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PublicFormat.SubjectPublicKeyInfo  
)  
# 输出密钥信息  
print("私钥内容：", private\_key\_pem.decode("utf8"))  
print("公钥内容：", public\_key\_pem.decode("utf8"))  
  
# 签名  
data = 'hello'.encode("utf8")  
signature = private\_key.sign(data)  
print(f"签名长度 {len(signature)} 字节，内容：{signature}")  
  
# 验签  
try:  
 public\_key.verify(signature, data)  
 print("验证成功")  
except exceptions.InvalidSignature:  
 print("验证失败")

结果：

私钥内容： -----BEGIN PRIVATE KEY-----

MC4CAQAwBQYDK2VwBCIEIO8XuB8Yj+kTZnAFa6GF0FYeNAxzgIE0pluhqBLov6JJ

-----END PRIVATE KEY-----

公钥内容： -----BEGIN PUBLIC KEY-----

MCowBQYDK2VwAyEADk+j2sKp4SJmpGq5wgwdouUBB3wYrpIFbMhhzQ9IKms=

-----END PUBLIC KEY-----

签名长度 64 字节，内容：

b'\xaf\xff\xfb$8=\xce!\x18E?\xbd\x95\x1d\xcb\x90\xef\xa6\xb6\xcd\xe5\xa0\xfc\x07\xe6gh0\xd36\x86\x91V\xdc\xa2\x97!A\x16=)\x01{ s9\xd0j\xe4\x16#\xb7\x0e/$\xfa\x1c4\\\xfa\xed[\x90\x08'

验证成功

**★第19章、tkinter图形界面编程**

Tkinter是Python的标准GUI库，它提供了丰富的组件和布局管理器，可用于创建图形用户界面应用程序

Tkinter库具有以下优点：

是Python的标准GUI库，无需安装第三方库即可使用

提供了丰富的组件和布局管理器，能够满足大多数应用程序的需求

具有良好的跨平台性，能够在Windows、macOS和Linux等操作系统上运行

在python 2.x中，模块名为Tkinter（首字母大写）

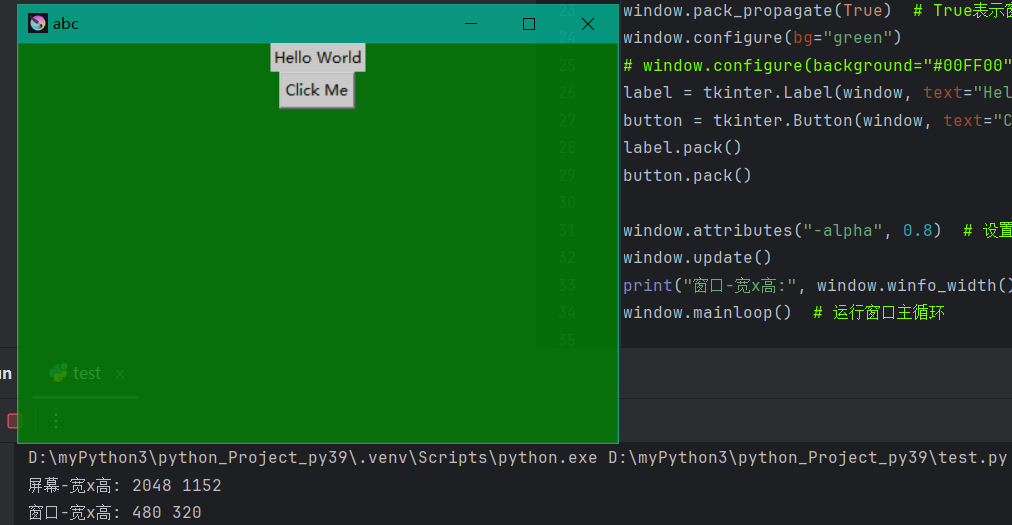
在python 3.x中，模块名为tkinter（首字母小写）

**★窗口及组件对象**

在Tkinter中，窗口和组件都是对象。我们可以创建一个窗口对象，并在其中添加各种组件对象

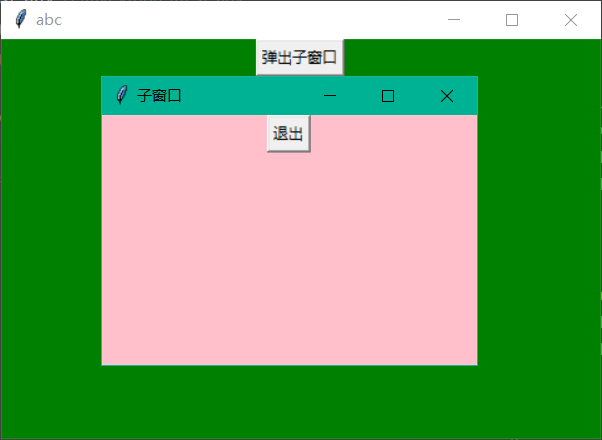
import tkinter  
  
win\_width = 480 # 窗口宽度  
win\_height = 320 # 窗口高度  
  
window = tkinter.Tk() # 创建窗口对象  
screen\_width = window.winfo\_screenwidth()  
screen\_height = window.winfo\_screenheight()  
  
window.title("abc") # 设置窗口标题  
# window.iconbitmap(bitmap="D:\\test.ico") # 设置窗口图标，默认为羽毛图标  
window.wm\_iconbitmap(bitmap="D:\\test.ico") # 设置窗口图标，默认为羽毛图标  
# iconbitmap()方法只能应用于顶级窗口（即 Tk()函数创建的窗口），而wm\_iconbitmap()方法可以应用于任何窗口，包括顶级窗口和子窗口  
# iconbitmap()方法只能处理.ico格式的图标文件，而wm\_iconbitmap()方法可以处理多种格式的图标文件，例如bmp、gif、ico、jpg、png等  
  
win\_pos = f"{win\_width}x{win\_height}+{screen\_width // 2 - win\_width // 2}+{screen\_height // 2 - win\_height // 2}"  
print("屏幕-宽x高:", screen\_width, screen\_height)  
# window.geometry("480x320+100+100") # 设置窗口大小及位置，宽x高+横坐标+纵坐标  
window.geometry(win\_pos) # 设置窗口大小及位置，居中  
window.resizable(width=True, height=True) # 宽度和高度可由用户手动调整，False表示不可调整  
window.minsize(200, 140) # 可调整的最小宽度及高度  
window.maxsize(800, 640) # 可调整的最大宽度及高度  
window.pack\_propagate(True) # True表示窗口内的控件大小自适应，False表示不自适应，需要手动设置控件坐标及大小  
window.configure(bg="green")  
# window.configure(background="#00FF00") # bg为background的缩写，颜色可写英文单词，也可写#RGB  
label = tkinter.Label(window, text="Hello World") # 创建标签组件  
button = tkinter.Button(window, text="Click Me") # 创建按钮组件  
label.pack()  
button.pack()  
  
window.attributes("-alpha", 0.8) # 设置透明度，值为[0,1]之间的浮点数，值越小越透明  
window.update() # 获取控件size之前，得先update()刷新一下，否则winfo\_width及winfo\_height返回值为1  
print("窗口-宽x高:", window.winfo\_width(), window.winfo\_height())  
window.mainloop() # 运行窗口主循环

效果：



**★Toplevel子窗口**

import tkinter  
  
main\_window = tkinter.Tk() # 创建窗口对象  
main\_window.title("abc") # 设置窗口标题  
main\_window.geometry("480x320+100+100") # 设置窗口大小及位置，宽x高+横坐标+纵坐标  
main\_window.configure(bg="green")  
  
  
def click\_exit\_button(pop\_window):  
 pop\_window.destroy()  
 main\_window.attributes("-disabled", 0) # 使主窗口可响应  
 main\_window.focus\_force() # 使主窗口获得焦点  
  
  
def on\_closing():  
 main\_window.attributes("-disabled", 0)  
 main\_window.focus\_force()  
  
  
def create\_pop\_window():  
 main\_window.attributes("-disabled", 1) # 使主窗口关闭响应，无法点击它  
 pop\_window = tkinter.Toplevel(main\_window) # 创建子窗口对象  
 pop\_window.title("子窗口")  
 pop\_window.geometry("300x200+100+100")  
 pop\_window.configure(bg="pink")  
 pop\_window.protocol("WM\_DELETE\_WINDOW", on\_closing) # 子窗口点击右上角的关闭按钮后，触发此函数  
 exit\_button = tkinter.Button(pop\_window, text="退出", command=lambda: click\_exit\_button(pop\_window)) # 子窗口创建按钮  
 exit\_button.pack()  
  
  
create\_pop\_window\_button = tkinter.Button(main\_window, text="弹出子窗口", command=create\_pop\_window) # 主窗口创建按钮  
create\_pop\_window\_button.pack()  
  
main\_window.mainloop() # 运行窗口主循环



**★布局管理器**

在Tkinter中，布局管理器负责管理组件的位置和大小。目前，Tkinter提供了三种布局管理器：pack、grid和place

**①pack布局管理器**

按照添加顺序将组件放置到窗口中

con.pack() #默认 pack 会在父窗口 window 中垂直方向按顺序包装排列控件，等价于 pack(side=tkinter.TOP)

con.pack(side=tkinter.LEFT) #水平排列，从左到右

con.pack(fill=tkinter.BOTH, expand=tkinter.TRUE) #让里边的控件跟随窗口自动拉伸大小

fill= {NONE、X、Y、BOTH} #选择当控件伸缩时按照哪个方向填充

expand= {bool 类型值} #跟着父控件一起伸缩

**②grid布局管理器**

将窗口划分为网格，并将组件放置到指定的网格中

grid 会把父窗口划分成行列，然后根据调用时传入参数 row，column 确定把控件放置在对应的行列中

每一列中，列宽由这一列中最宽的单元格确定。

每一行中，行高由这一行中最高的单元格决定。

控件并不是充满整个单元格的，可以指定单元格中剩余空间的使用，或者空出这些空间。可以在水平或竖直或两个方向上填满这些空间。也可以连接若干个单元格为一个更大空间， 这一操作被称作跨越（span），与excel的合并单元格是一个概念。使用grid的布局方法的时候，单元格必须是紧邻创建的。

**③place布局管理器**

允许开发者精确地控制组件的位置和大小

★同一顶级层次中，不同布局管理器不可混用

**★隐藏控件**

使用pack()显示的控件，使用 .pack\_forget() 方法隐藏起来，使控件不可见，并未删除

使用grid()显示的控件，使用 .grid\_forget() 方法隐藏起来

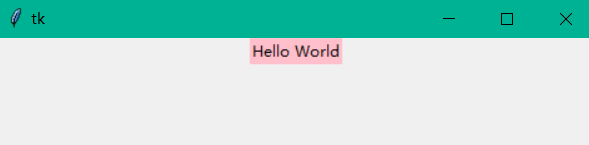
使用place()显示的控件，使用 .place\_forget() 方法隐藏起来

**★Lebel标签**

Label标签用于显示不可编辑的文本信息

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
label = tkinter.Label(window, text="Hello World", bd=1, bg="pink")  
label.pack()  
  
window.mainloop() # 运行窗口主循环

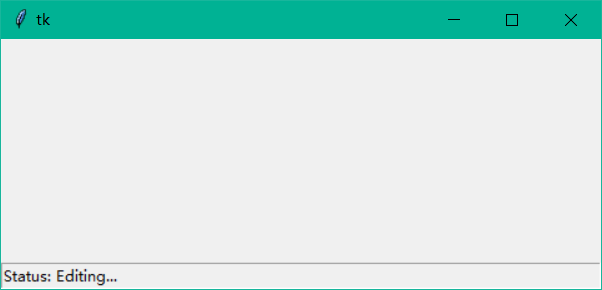
显示效果：



★模拟状态栏

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
status\_bar = tkinter.Label(window, text="Status: Editing...", bd=1, relief=tkinter.SUNKEN, anchor=tkinter.W)  
status\_bar.pack(side=tkinter.BOTTOM, fill=tkinter.X)  
  
window.mainloop() # 运行窗口主循环

显示效果：



**★文本左对齐**

**参数：**

anchor = 'w' # 文本区域左对齐，可填'n', 's', 'e', 'w'  
justify = 'left' # 行左对齐，可填'left', 'right', 'center'

#若使用了grid布局，可在grid里使用sticky参数

widget.grid(sticky='w') # 在单元格内左对齐，仅限grid，可填'n', 's', 'w', 'e'或自由搭配

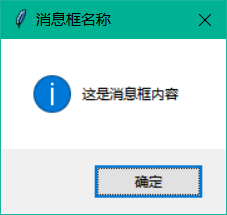
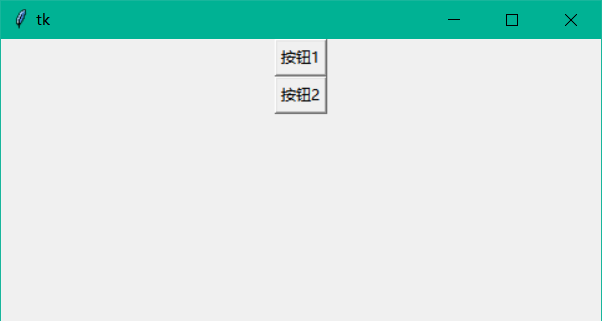
**★Button按钮**

按钮上可以显示文本或图像，并且可以与一个回调函数关联，当用户点击按钮时，自动调用该回调函数。要向按钮回调函数传递参数，得使用lambda函数：

command=lambda: 回调函数(var1, var2, varxx)

import tkinter  
from tkinter import messagebox  
  
def on\_button\_click():  
 messagebox.showinfo("消息框名称", "这是消息框内容")  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
button1 = tkinter.Button(window, text="按钮1", command=on\_button\_click) # 创建按钮组件  
button2 = tkinter.Button(window, text="按钮2", command=on\_button\_click) # 创建按钮组件  
button1.pack()  
button2.pack()  
  
window.mainloop() # 运行窗口主循环

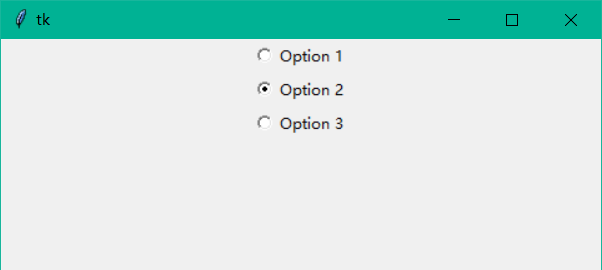
显示效果：



**★Radiobutton单选按钮**

import tkinter  
  
def on\_radiobutton\_click(var):  
 print(f"当前单选情况：var1值为 {var.get()}")  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
# 创建布尔变量  
var1 = tkinter.StringVar()  
  
# 创建复选按钮组件  
radiobutton1 = tkinter.Radiobutton(window, text="Option 1", variable=var1,value="a",  
 command=lambda: on\_radiobutton\_click(var1))  
radiobutton2 = tkinter.Radiobutton(window, text="Option 2", variable=var1, value="b",  
 command=lambda: on\_radiobutton\_click(var1))  
radiobutton3 = tkinter.Radiobutton(window, text="Option 3", variable=var1, value="c",  
 command=lambda: on\_radiobutton\_click(var1))  
var1.set("a")  
radiobutton1.pack()  
radiobutton2.pack()  
radiobutton3.pack()  
  
window.mainloop() # 运行窗口主循环

结果：



选中Option 2时，console输出：（选中哪个option，则var1的值就被设置为那个Radiobutton的value）

当前单选情况：var1值为 b

**★Checkbutton复选按钮**

复选框可以显示文本或图像，并且可以与一个布尔变量关联，当用户选中或取消选中复选框时，自动更新该布尔变量的值。要向按钮回调函数传递参数，得使用lambda函数

import tkinter  
  
def on\_checkbutton\_click(var1, var2):  
 print(f"当前复选情况：var1值为 {var1.get()}，var2值为 {var2.get()}")  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
# 创建布尔变量  
var1 = tkinter.BooleanVar()  
var2 = tkinter.BooleanVar()  
  
# 创建复选按钮组件  
checkbutton1 = tkinter.Checkbutton(window, text="Option 1", variable=var1,  
 command=lambda: on\_checkbutton\_click(var1, var2))  
checkbutton2 = tkinter.Checkbutton(window, text="Option 2", variable=var2,  
 command=lambda: on\_checkbutton\_click(var1, var2))  
  
checkbutton1.pack()  
checkbutton2.pack()  
  
window.mainloop() # 运行窗口主循环

显示效果：

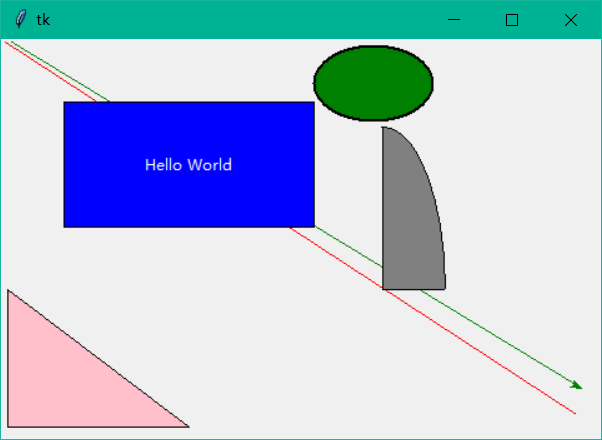


**★Canvas画布**

画布组件用于在应用程序中绘制图形和文本，画布支持多种绘图方法，如绘制直线、矩形、椭圆和文本等。

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
canvas = tkinter.Canvas(window, width=480, height=320) # 创建画布组件  
canvas.pack()  
  
canvas.create\_line(0, 0, 460, 300, fill="red") # 绘制直线，左上角坐标，右下角坐标，填充颜色  
canvas.create\_line(5, 0, 465, 280, fill="green", arrow=tkinter.LAST) # 绘制直线，带箭头的  
canvas.create\_rectangle(50, 50, 250, 150, fill="blue") # 绘制矩形，左上角坐标，右下角坐标，填充颜色  
canvas.create\_text(150, 100, text="Hello World", fill="white") # 绘制文本，左上角坐标  
canvas.create\_oval(250, 5, 345, 65, fill="green", width=2, outline="black") # width指定边框粗细，默认为1黑色，outline="black"指定边框颜色  
canvas.create\_arc(255, 70, 355, 330, fill="gray", outline="black") # 画扇形  
canvas.create\_polygon(5, 200, 150, 310, 5, 310, fill="pink", outline="black") # 画多边形（这里为三角形）  
  
window.mainloop() # 运行窗口主循环

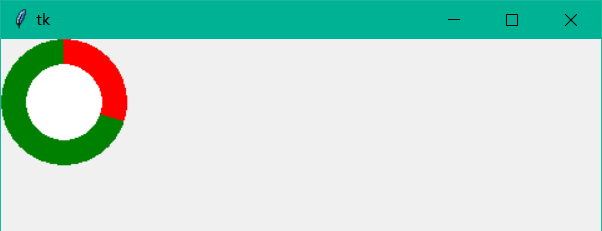
显示效果：



**★画占比图**

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
canvas = tkinter.Canvas(window, width=480, height=320, highlightthickness=0) # 创建画布组件，设置边框宽度为0  
canvas.pack()  
  
# 清空画布  
canvas.delete(\*canvas.find\_all()) # find\_all()返回所有对象id元组，delete()删除指定的id对象  
# 创建使用率占比图  
canvas.create\_oval(0, 0, 100, 100, fill="green", width=0, outline="green") # 画圆形，底座  
angle = 0.3 \* 360 # 使用率 乘以 360度  
canvas.create\_arc(0, 0, 100, 100, extent=angle, start=90 - angle, fill="red", width=0, outline="red")  
# extent指定要画的扇形角度，顺时针方向为正值，  
# start为起始角度，默认为0，即水平的X轴方向，逆时针方向为正值  
# 画圆形，中心空白  
ring\_width = 20  
canvas.create\_oval(ring\_width, ring\_width, 100 - ring\_width, 100 - ring\_width, fill="white", width=0, outline="white")  
  
window.mainloop() # 运行窗口主循环

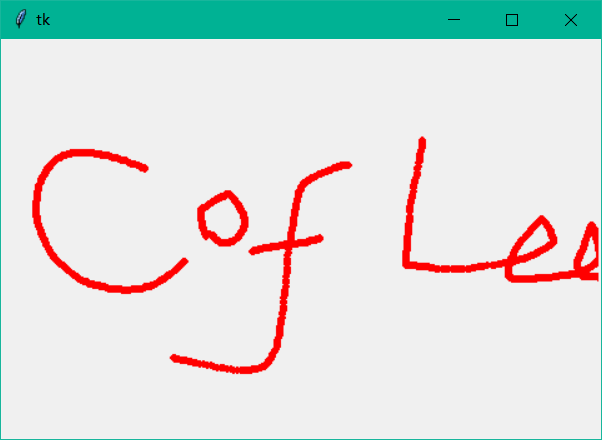
效果：



**★Canvas画点线**

import tkinter  
  
def move(event):  
 x = event.x  
 y = event.y  
 canvas.create\_oval(x, y, x + 5, y + 5, fill='red', width=0)  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
canvas = tkinter.Canvas(window, width=480, height=320) # 创建画布组件  
canvas.pack()  
canvas.bind('<B1-Motion>', move)  
  
window.mainloop() # 运行窗口主循环

显示效果：

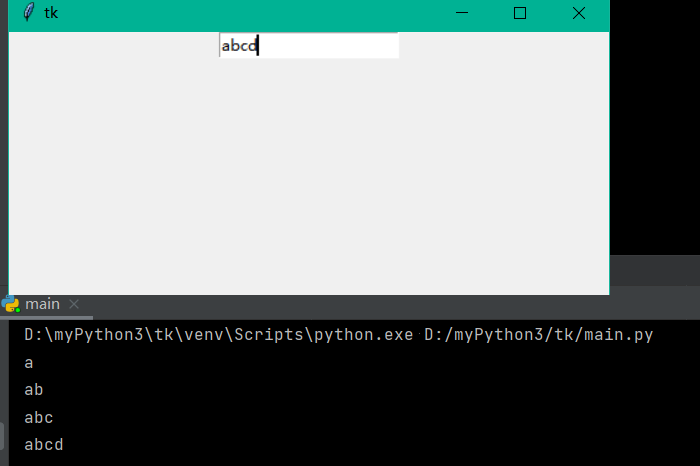


**★Entry单行文本输入框**

条目组件用于在应用程序中添加单行文本输入框，条目组件可以与一个字符串变量关联，当用户在条目组件中输入文本时，自动更新该字符串变量的值。

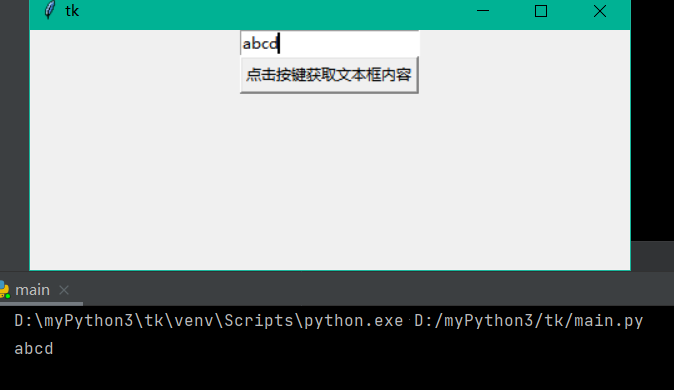
import tkinter  
  
def on\_entry\_change(sv):  
 print(sv.get())  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
sv = tkinter.StringVar() # 创建字符串变量  
sv.trace("w", lambda name, index, mode, sv=sv: on\_entry\_change(sv))  
  
entry = tkinter.Entry(window, textvariable=sv) # 创建条目组件  
entry.pack()  
  
window.mainloop() # 运行窗口主循环

显示效果：



import tkinter  
  
  
def on\_button\_click(sv):  
 print(sv.get())  
  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
sv = tkinter.StringVar() # 创建字符串变量  
entry = tkinter.Entry(window, textvariable=sv) # 创建条目组件  
button1 = tkinter.Button(window, text="点击按键获取文本框内容", command=lambda: on\_button\_click(sv)) # 创建按钮组件  
entry.pack()  
button1.pack()  
  
window.mainloop() # 运行窗口主循环

显示效果：



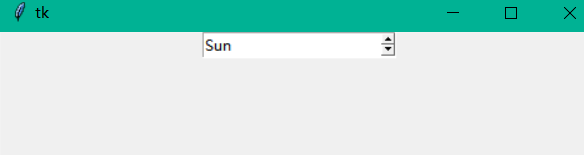
**★Spinbox**

Spinbox相当于Entry的扩展，除了直接在spinbox中编辑或修改其值外，还可以通过上、下按钮来升序或降序地选择其值。

import tkinter  
  
  
def on\_spinbox\_change(sv):  
 print(sv.get())  
  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
# 指定spinbox的值，values可以是一个列表或元组，如果指定了values， from\_,to,increment将被屏蔽  
values = ["Sun", "Mon", "Tue", "Wed", "Thu", "Sat"] # spinbox将只能在1-6范围内增减  
spinval = tkinter.StringVar()  
  
spinbox = tkinter.Spinbox(window, from\_=1, to=100, increment=2, textvariable=spinval, values=values,  
 command=lambda: on\_spinbox\_change(spinval))  
# increment 指定步进值, 默认为1

spinval.set("Sun") # 设置默认值

spinbox.pack()  
  
window.mainloop() # 运行窗口主循环

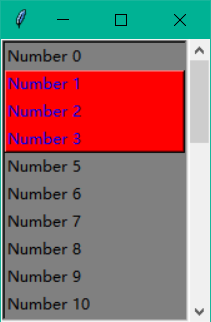


**★Listbox列表框**

Listbox控件的显示主体就是一个选项列表（items list），每个item一行。当创建一个Listbox控件对象时，该对象没有选项（item）。item可以支持两种方法创建。item可以被删除。一个或多个item可以被选中，然后触发相应的函数。

import tkinter  
  
  
def select\_item(\*args):  
 print(listbox.curselection())  
 print(args)  
  
  
window = tkinter.Tk() # 创建窗口  
scrollbar = tkinter.Scrollbar(window) # 创建窗口滚动条  
scrollbar.pack(side="right", fill="y") # 设置窗口滚动条位置  
  
# 创建列表框  
listbox = tkinter.Listbox(window, selectmode="multiple", bg="gray", bd=2, cursor="arrow", yscrollcommand=scrollbar.set,  
 selectbackground='red', selectforeground='blue', selectborderwidth=2, activestyle='dotbox',  
 exportselection=False)  
# 当Listbox组件的可视范围发生改变的时候，Listbox组件通过调用set()方法通知Scrollbar组件  
# 而当用户操纵滚动条时,就自动调用Listbox组件的yview方法  
# 添加水平滚动条方法跟上边一样,只是将yscrollcommand改为xscrollcommand，yview改成xview即可  
# exportselection=False,表示可以同时选择多个不同listbox的item，默认情况下，当用户选择了当前的listbox条目后，其他listbox条目都会被清除选择

for line in range(30):  
 listbox.insert(tkinter.END, "Number " + str(line)) # 添加item选项  
  
listbox.pack(side="left")  
scrollbar.config(command=listbox.yview) # 而当用户操纵滚动条时,就自动调用Listbox组件的yview方法  
listbox.delete(4)  
print(listbox.curselection())  
listbox.select\_set(1, 3)  
print(listbox.curselection())  
print(listbox.size())  
listbox.bind('<<ListboxSelect>>', select\_item) # 将item的选择事件和select\_item()绑定  
  
window.mainloop()



console输出：

()

(1, 2, 3)

29

**★Listbox属性**

selectmode=

browse 默认值，只能选一个选项，即使你按住鼠标拖动，也是选择最后鼠标停留的那个选项

extended 通过按住鼠标拖动，选择多个选项（连续的多个选项）

single 只能选择一个选项且只支持鼠标单击，鼠标按住并拖动无效，这也是与browse有差异的地方

multiple 选择多个选项且支持鼠标单击（可不连续的多个选项）鼠标按住并拖动无效

, selectbackground='red', selectforeground='blue', selectborderwidth=5

activestyle='dotbox' 表示当某个item被选中后，该item的显示样式

dotbox 该样式是 选中的item(s)的外围矩形边界不是常规的实线，而是点线。

none 没有样式显示

underline 选中的item(s)的文字有下划线，默认样式

★listbox的常见index有如下几种：

number item的索引，从0开始计数

active 表示listbox的当前光标选中的item。

anchor Indicates the anchor point for the selection

end 表示listbox的最后一个item

listbox.index("active") #返回选中的那个index值  
listbox.delete(4) #删除第4个选项，从0开始  
listbox.delete(4,5) #删除第4到第5个选项  
listbox.curselection() #获取当前选中的item索引，返回<tuple>  
listbox.select\_set(3) #设置第3个item被选中  
listbox.select\_set(1,3) #设置第1到第3个item被选中  
listbox.size() #获取listbox中item的个数

★Listbox没有command属性，那么需要使用bind函数来关联事件

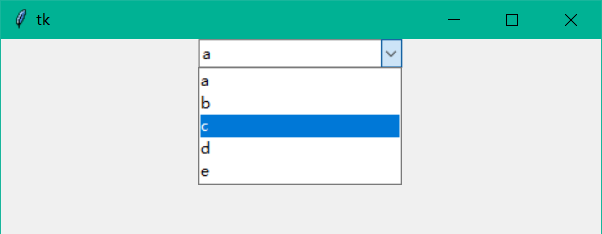
listbox.bind('<<ListboxSelect>>', select\_item) # 将item的选择事件和select\_item()绑定

**★Combobox下拉列表**

combobox是用户可用来选择某个数据条目的下拉列表。它是Entry和drop-down控件的组合。当你单击右侧的箭头时，将看到一个显示所有选项的下拉菜单，单击其中的一个，它将替换当前的Entry内容

import tkinter  
from tkinter import ttk  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
combo\_list = ["a", "b", "c", "d", "e"]  
combobox = ttk.Combobox(window, values=combo\_list, state="normal")  
combobox.pack()  
print("current1: ", combobox.current()) # 当前未选中任何条目，结果为-1  
combobox.current(0) # 设置当前选中第0个条目  
print("current2: ", combobox.current())  
print("get content: ", combobox.get())  
  
def selected(event):  
 print("current: ", combobox.current())  
 print("get content: ", combobox.get())  
  
combobox.bind("<<ComboboxSelected>>", selected)  
window.mainloop() # 运行窗口主循环

显示效果：



console输出：

current1: -1

current2: 0

get content: a

**★Frame框架**

框架组件用于在应用程序中添加矩形区域，通常用作容器来管理其他组件。框架组件可以嵌套使用，以创建复杂的布局。

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
# 创建框架1  
frame1 = tkinter.Frame(window, bg="green", width=240, height=100)  
frame1.pack(fill="both", expand=True)  
# 在框架1中添加标签  
label = tkinter.Label(frame1, text="Hello 标签1")  
label.pack()  
  
# 创建框架2  
frame2 = tkinter.Frame(window, bg="blue", width=200, height=100)  
frame2.pack(fill="both", expand=True)  
# 在框架2中添加按钮  
button2 = tkinter.Button(frame2, text="按钮2")  
button2.pack()  
  
window.mainloop() # 运行窗口主循环

显示效果：（默认填充满整个window）

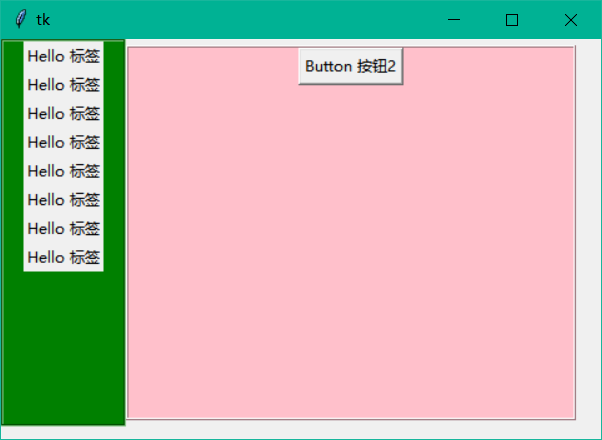


需要固定Frame的大小，则使用以下配置

frame1.grid\_propagate(False)  
frame1.pack\_propagate(False)

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
# 创建框架1  
frame1 = tkinter.Frame(window, bg="green", borderwidth=2, width=100, height=310, relief='groove')  
frame1.grid\_propagate(False)  
frame1.pack\_propagate(False)  
frame1.grid(row=0, column=0)  
# 在框架1中添加标签  
for i in range(8):  
 label = tkinter.Label(frame1, text="Hello 标签")  
 label.pack()  
  
# 创建框架2  
frame2 = tkinter.Frame(window, bg="pink", borderwidth=2, width=360, height=300, relief='ridge')  
frame2.grid\_propagate(False)  
frame2.pack\_propagate(False)  
frame2.grid(row=0, column=1)  
# 在框架2中添加按钮  
button2 = tkinter.Button(frame2, text="Button 按钮2")  
button2.pack()  
  
window.mainloop() # 运行窗口主循环

显示效果：



**★Text多行文本框**

import tkinter  
from tkinter import filedialog  
  
  
def open\_file(txt\_edit):  
 file\_path = filedialog.askopenfilename(filetypes=[("Text files", "\*.txt"), ("All files", "\*.\*")])  
 if not file\_path:  
 window.title("无标题 - 记事本")  
 return  
 window.title(f"{file\_path} - 记事本")  
 txt\_edit.delete("1.0", tkinter.END) #删除所有内容  
 with open(file\_path, mode="r", encoding="utf-8") as input\_file:  
 txt\_edit.insert(tkinter.END, input\_file.read()) # 从文本框最后新增文本  
  
  
def save\_as\_file(txt\_edit):  
 file\_path = filedialog.asksaveasfilename(defaultextension=".txt",  
 filetypes=[("Text files", "\*.txt"), ("All files", "\*.\*")])  
 if not file\_path:  
 return  
 with open(file\_path, mode="w", encoding="utf-8") as output\_file:  
 output\_file.write(txt\_edit.get("1.0", tkinter.END))# 从文本框获取文本，第1行第0列到末尾  
 window.title(f"{file\_path} - 记事本")  
  
  
window = tkinter.Tk()  
window.title("记事本")  
  
window.rowconfigure(0, minsize=100, weight=1)  
window.columnconfigure(1, minsize=100, weight=1)  
  
btn\_frame = tkinter.Frame(master=window, bd=2)  
btn\_open = tkinter.Button(master=btn\_frame, text="打开", command=lambda: open\_file(txt\_edit))  
btn\_save\_as = tkinter.Button(master=btn\_frame, text="另存", command=lambda: save\_as\_file(txt\_edit))  
  
btn\_frame.grid(row=0, column=0, sticky="ns")  
btn\_open.grid(row=0, column=0, sticky="ew", padx=5, pady=5)  
btn\_save\_as.grid(row=1, column=0, sticky="ew", padx=5)  
  
txt\_edit = tkinter.Text(master=window)  
txt\_edit.grid(row=0, column=1, sticky="nsew")  
  
window.mainloop()

显示效果：



**★设置字体颜色**

import tkinter  
from tkinter import font  
  
window = tkinter.Tk()  
window.title("记事本")  
  
txt\_edit = tkinter.Text(master=window, spacing1=0, spacing2=0, spacing3=0, bg="pink")  
txt\_edit.grid(row=0, column=1, sticky="nsew")  
  
font\_bold = font.Font(weight='bold', size=14, name="")  
txt\_edit.tag\_config("tag\_config\_name", foreground="white", backgroun="black", font=font\_bold, underline=1,  
 underlinefg="green")  
txt\_edit.insert(tkinter.END, "hello", "tag\_config\_name")  
  
window.mainloop()



**★索引及光标定位**

import tkinter  
from tkinter import font  
  
window = tkinter.Tk()  
window.title("记事本")  
  
txt\_edit = tkinter.Text(master=window, spacing1=0, spacing2=0, spacing3=0, bg="pink")  
txt\_edit.grid(row=0, column=1, sticky="nsew")  
  
txt\_edit.config(insertbackground="green") # 设置光标颜色  
font\_bold = font.Font(weight='bold', size=14, name="")  
txt\_edit.tag\_config("tag\_config\_name", foreground="white", backgroun="black", font=font\_bold, underline=1,  
 underlinefg="green")  
txt\_edit.insert(tkinter.END, "hello", "tag\_config\_name")  
  
# 选择完文本后，text\_cursor光标会停留在tkinter.SEL\_LAST  
txt\_edit.mark\_set("tkinter\_END", tkinter.END) # 使text\_cursor光标移动到末尾  
# 以下方法会使闪烁的光标(text\_cursor)移到最后一行行末，但页面不会滚动，选中的内容仍在当前界面  
txt\_edit.mark\_set(tkinter.INSERT, "end lineend")  
# .see()方法会使文本框滚动内容到末尾，选择的内容如果不在最后一页，则被滚动了，当前页面看不到了  
txt\_edit.see("tkinter\_END")  
  
txt\_edit.focus\_force() # 聚焦到Text控件上  
  
window.mainloop()

**索引：**

tkinter.SEL\_FIRST, tkinter.SEL\_LAST # 选中的文本，开始索引及结尾索引

selected\_text = txt\_edit.get(tkinter.SEL\_FIRST, tkinter.SEL\_LAST) # 获取选中的文本

# 设置选中的索引，设置tkinter.SEL后，自动会更新tkinter.SEL\_FIRST, tkinter.SEL\_LAST

txt\_edit.tag\_add(tkinter.SEL, "1.0", tkinter.END + "-1c")

**tag绑定事件：**

tag\_add()选中某段字符串，然后使用tag\_config可设置这些字符串的显示属性，也可使用tag\_bind绑定事件，如鼠标单击事件

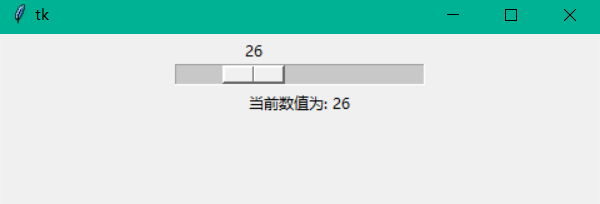
txt\_edit.tag\_bind("tag\_name",'<Button-1>',click\_func)  
def click\_func(event):  
 print("xxx")

**★Scale滑动条**

Scale滑动条用于设置/获取指定范围的数值。

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
def drag\_cb(val):  
 str\_1 = "当前数值为: " + val  
 label.config(text=str\_1)  
  
scale = tkinter.Scale(window, from\_=1, to=100, resolution=1, command=drag\_cb, orient="horizontal", showvalue=1,  
 length=200, sliderlength=50)  
# from\_=1, to=100 表示数值范围为 [1,100]  
# resolution=1 表示步长为1，滑动间隔为1  
# showvalue=1 表示在滑轨上显示当前数值  
# length=200 表示整个滑轨长度  
# sliderlength=50 滑块的长度  
scale.pack()  
label = tkinter.Label(window)  
label.pack()  
  
window.mainloop()

显示效果：



**★Treeview表格/树状结构**

**★表格**

import tkinter  
from tkinter import ttk  
  
  
def click\_func(event, treeview):  
 item\_index = treeview.focus() # 获取用户单击的item index  
 item\_content = treeview.item(item\_index, "values")  
 print("selected item: ", item\_content)  
  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
yscroll = tkinter.Scrollbar(window, orient=tkinter.VERTICAL)  
tree\_view = ttk.Treeview(window, columns=("name", "class", "age", "weight"), show="headings", height=10,  
 yscrollcommand=yscroll.set)  
yscroll.config(command=tree\_view.yview)  
# 设置每一个列的宽度和对齐的方式  
tree\_view.column("name", width=100, anchor="center")  
tree\_view.column("class", width=80, anchor="center")  
tree\_view.column("age", width=80, anchor="center")  
tree\_view.column("weight", width=100, anchor="center")  
# 设置每个列的标题  
tree\_view.heading("name", text="姓名")  
tree\_view.heading("class", text="班级")  
tree\_view.heading("age", text="年龄")  
tree\_view.heading("weight", text="体重")  
# 插入数据  
for index in range(50):  
 tree\_view.insert("", index, values=("cof", "1", "18", index))  
# 绑定单击事件到Treeview组件  
tree\_view.bind("<<TreeviewSelect>>", lambda event: click\_func(event, tree\_view))  
  
yscroll.pack(side=tkinter.RIGHT, fill=tkinter.Y)  
tree\_view.pack() # 被滚动的控件必须放在滚动条之后  
  
# 设置风格  
style = ttk.Style()  
style.theme\_use("default")  
style.configure("Treeview", background="white", foreground="red", fieldbackground="white")  
style.configure("Treeview.Heading", background="#BBB", foreground="black", font=("Arial", 10, "bold"))  
style.configure("Treeview.Row", background="white", foreground="green", font=("Arial", 10))  
  
window.mainloop() # 运行窗口主循环

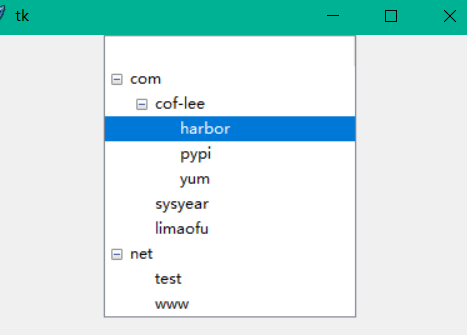


console输出：

selected item: ('cof', '1', '18', '4')

**★树状结构**

import tkinter  
from tkinter import ttk  
  
main\_window = tkinter.Tk()  
main\_window.geometry("400x300+200+200")  
  
tree\_root = ttk.Treeview(main\_window) # 顶级树根  
tree\_root.pack()  
  
# 添加一级树枝  
tree\_0 = tree\_root.insert("", 0, "com", text="com") # ★  
tree\_1 = tree\_root.insert("", 1, "net", text="net")  
tree\_2 = tree\_root.insert("", 2, "org", text="org")  
  
# 添加二级树枝  
tree\_0\_0 = tree\_root.insert(tree\_0, 0, "cof-lee", text="cof-lee") # ★★  
tree\_0\_1 = tree\_root.insert(tree\_0, 1, "sysyear", text="sysyear")  
tree\_0\_2 = tree\_root.insert(tree\_0, 2, "limaofu", text="limaofu")  
  
tree\_1\_0 = tree\_root.insert(tree\_1, 0, "test", text="test")  
tree\_1\_1 = tree\_root.insert(tree\_1, 1, "www", text="www")  
tree\_1\_2 = tree\_root.insert(tree\_1, 2, "any", text="any")  
  
tree\_2\_0 = tree\_root.insert(tree\_2, 0, "api", text="api")  
tree\_2\_1 = tree\_root.insert(tree\_2, 1, "auth", text="auth")  
tree\_2\_2 = tree\_root.insert(tree\_2, 2, "login", text="login")  
  
# 三级树枝  
tree\_0\_0\_0 = tree\_root.insert(tree\_0\_0, 0, "harbor", text="harbor") # ★★★  
tree\_0\_0\_1 = tree\_root.insert(tree\_0\_0, 1, "pypi", text="pypi")  
tree\_0\_0\_2 = tree\_root.insert(tree\_0\_0, 2, "yum", text="yum")  
  
main\_window.mainloop()



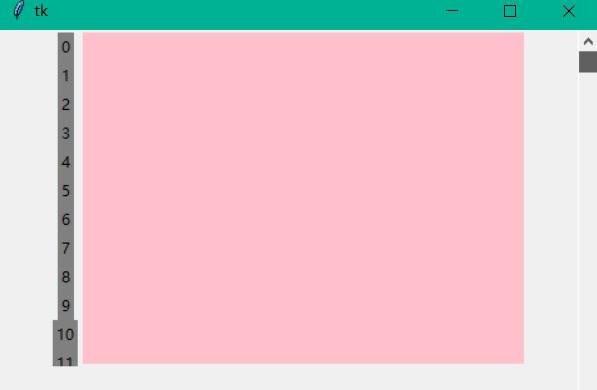
**★Canvas-Frame-滚动框实现**

import tkinter  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
scrollbar = tkinter.Scrollbar(window)  
scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)  
canvas = tkinter.Canvas(window, yscrollcommand=scrollbar.set, bg="pink")  
canvas.pack()  
scrollbar.config(command=canvas.yview)  
frame = tkinter.Frame(canvas)  
frame.pack()  
canvas.create\_window((0, 0), window=frame, anchor='nw')  
  
for i in range(200):  
 label = tkinter.Label(frame, text=str(i), bg="gray")  
 label.pack()  
  
frame.update\_idletasks() # 更新Frame的尺寸  
canvas.configure(scrollregion=(0, 0, 460, 200 \* 24))  
  
  
def proces\_mouse\_scroll(event):  
 global canvas  
 if event.delta > 0:  
 canvas.yview\_scroll(-1, 'units') # 向上移动  
 else:  
 canvas.yview\_scroll(1, 'units') # 向下移

canvas.bind("<MouseWheel>", proces\_mouse\_scroll)  
window.mainloop() # 运行窗口主循环

画布移动到最开头位置：

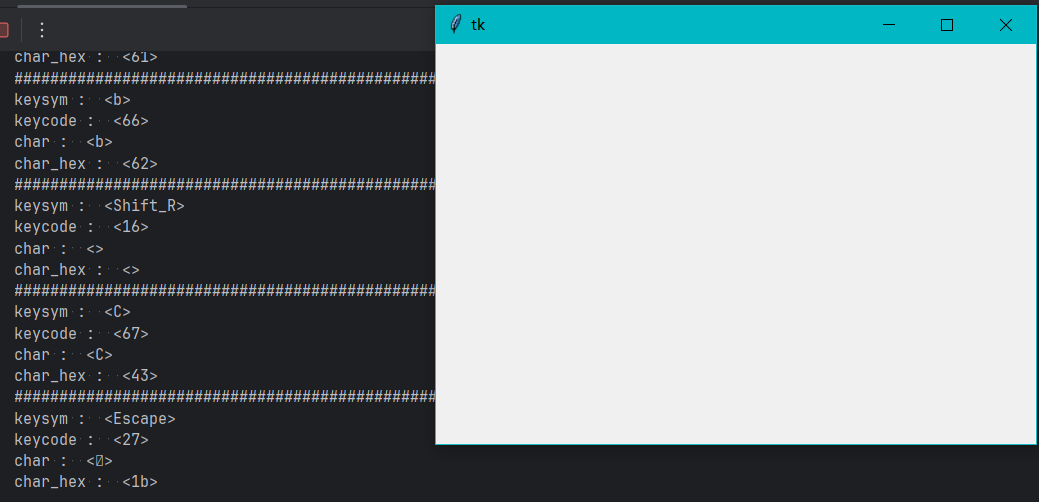
canvas.yview(tkinter.MOVETO, 0.0) # MOVETO表示移动到，0.0表示最开头，1.0表示最底端



**★键盘事件**

import tkinter  
  
  
def handle\_keypress(event):  
 print("##########################################################################")  
 print(f"keysym : <{event.keysym}>")  
 print(f"keycode : <{event.keycode}>")  
 print(f"char : <{event.char}>")  
 print(f"char\_hex : <{event.char.encode('utf8').hex()}>")  
  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
window.bind("<Key>", handle\_keypress)  
  
window.mainloop() # 运行窗口主循环

显示效果：



**其他：**

def front\_end\_input\_func\_printable\_char(self, event):  
 *"""  
 处理普通可打印字符，控制键及组合按键  
 ★★★ 按键，ascii字符，vt100控制符是3个不同的概念  
 按键可以对应一个字符，也可没有相应字符，  
 按下shift/ctrl等控制键后再按其他键，可能会产生换档字符（如按下shift加数字键2，产生字符@）  
 vt100控制符是由ESC（十六进制为\0x1b，八进制为\033）加其他可打印字符组成，比如:  
 按键↑（方向键Up）对应的vt100控制符为 ESC加字母OA，即b'\033OA'  
 ★★★* ***:param*** *event:* ***:return****:  
 """* print("普通字符输入如下：")  
 print(event.keysym)  
 print(event.keycode)  
 # 非可打印字符没有event.char，event.char为空，需要发送event.keycode或转为vt100控制序列再发送  
 if event.keysym == "BackSpace":  
 input\_byte = struct.pack('b', event.keycode)  
 elif event.keysym == "Delete":  
 input\_byte = struct.pack('b', event.keycode)  
 elif event.keysym == "Down":  
 input\_byte = b'\033OB' # ESC O B 对应 方向键(↓) VK\_DOWN (40)  
 elif event.keysym == "Up":  
 input\_byte = b'\033OA' # ESC O A 对应 方向键(↑) VK\_UP (38)  
 elif event.keysym == "Left":  
 input\_byte = b'\033OD' # ESC O D 对应 方向键(←) VK\_LEFT (37)  
 elif event.keysym == "Right":  
 input\_byte = b'\033OC' # ESC O C 对应 方向键(→) VK\_RIGHT (39)  
 elif event.keysym == "Control\_L":  
 self.ctrl\_pressed = True  
 input\_byte = event.char.encode("utf8")  
 elif event.keysym == "Control\_R":  
 self.ctrl\_pressed = True  
 input\_byte = event.char.encode("utf8")  
 else:  
 # 可打印字符只能发送event.char，因为输入!@#$%^&\*()这些换档符号时，需要先按下Shift键再按下相应数字键，  
 # Shift键本身不发送（Shift键没有event.char），要发送的是换档后的符号  
 # ctrl+字母 这类组合键也是单一字符\0x01到\0x1A  
 input\_byte = event.char.encode("utf8")  
 if len(input\_byte) != 0:  
 self.user\_input\_byte\_queue.put(input\_byte)  
 return "break" # 事件处理脚本返回 "break" 会中断后面的绑定，所以键盘输入不会被插入到文本框

**其他事件：**

<Control-c> 匹配组合键

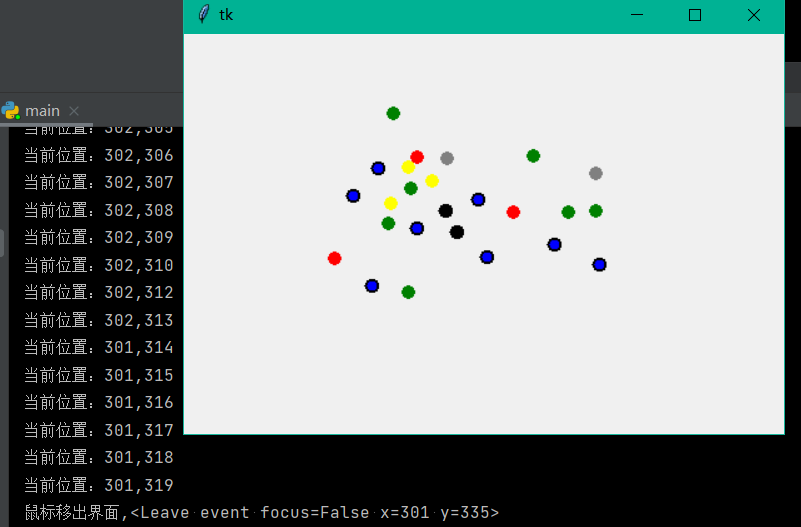
<KeyPress> 监听按键按下事件

<KeyRelease> 监听按键释放事件

**★鼠标事件**

import tkinter  
  
def left\_click(event):  
 x, y = event.x, event.y  
 canvas.create\_oval(x - 5, y - 5, x + 5, y + 5, fill="blue", width=2,  
 outline="black") # width指定边框粗细，默认为1黑色，outline="black"指定边框颜色  
  
def double\_left\_click(event):  
 x, y = event.x, event.y  
 canvas.create\_oval(x - 5, y - 5, x + 5, y + 5, fill="black", width=0)  
  
def middle\_click(event):  
 x, y = event.x, event.y  
 canvas.create\_oval(x - 5, y - 5, x + 5, y + 5, fill="red", width=0)  
  
def double\_middle\_click(event):  
 x, y = event.x, event.y  
 canvas.create\_oval(x - 5, y - 5, x + 5, y + 5, fill="gray", width=0)  
  
def right\_click(event):  
 x, y = event.x, event.y  
 canvas.create\_oval(x - 5, y - 5, x + 5, y + 5, fill="green", width=0)  
  
def double\_right\_click(event):  
 x, y = event.x, event.y  
 canvas.create\_oval(x - 5, y - 5, x + 5, y + 5, fill="yellow", width=0)  
  
def mouse\_wheel(event):  
 direction = event.delta  
 if direction > 0:  
 print("向上滚动：", direction)  
 if direction <= 0:  
 print("向下滚动：", direction)  
  
def mouse\_enter(event):  
 print(f"鼠标移入界面,{event}")  
  
def mouse\_leave(event):  
 print(f"鼠标移出界面,{event}")  
  
def current\_pos(event):  
 x = event.x  
 y = event.y  
 print(f"当前位置：{event.x},{event.y}")  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
canvas = tkinter.Canvas(window, width=480, height=320) # 创建画布组件  
canvas.pack()  
canvas.bind("<Enter>", mouse\_enter) # 绑定鼠标移入事件到Canvas画布上  
canvas.bind("<Leave>", mouse\_leave) # 绑定鼠标移出事件到Canvas画布上  
canvas.bind("<Button-1>", left\_click) # 绑定左键单击事件到Canvas画布上  
canvas.bind("<Double-Button-1>", double\_left\_click) # 绑定左键双击事件到Canvas画布上  
canvas.bind("<Button-2>", middle\_click) # 绑定中键单击事件到Canvas画布上  
canvas.bind("<Double-Button-2>", double\_middle\_click) # 绑定中键单击事件到Canvas画布上  
canvas.bind("<Button-3>", right\_click) # 绑定右键单击事件到Canvas画布上  
canvas.bind("<Double-Button-3>", double\_right\_click) # 绑定右键双击事件到Canvas画布上  
canvas.bind("<MouseWheel>", mouse\_wheel) # 绑定滚轮事件到Canvas画布上  
canvas.bind("<B1-Motion>", current\_pos) # 绑定左键长按移动事件到Canvas画布上  
canvas.bind("<B2-Motion>", current\_pos) # 绑定中键长按移动事件到Canvas画布上  
canvas.bind("<B3-Motion>", current\_pos) # 绑定右键长按移动事件到Canvas画布上  
canvas.bind("<Motion>", current\_pos) # 绑定移动事件到Canvas画布上  
window.mainloop() # 运行窗口主循环

显示效果：



**其他事件：**

<ButtonRelease-1> 鼠标左击释放事件

**★bind绑定事件**

★bind绑定事件，有的事件默认会传一个参数event

如果需要给给回调函数传入额外的参数，需要使用lambda函数，如下：

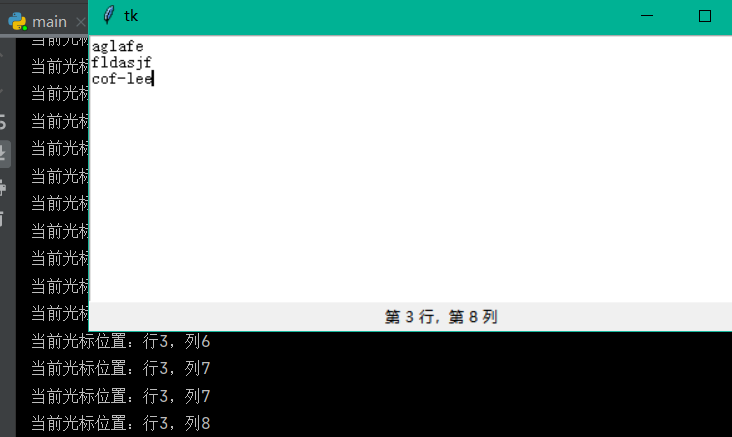
def func\_call\_back(event, var1, var2):  
 print(var1, var2)  
  
canvas.bind("<MouseWheel>", lambda event: func\_call\_back(event, "Hello", "World"))

**★文本框获取光标位置**

import tkinter  
  
# 光标位置获取与显示行列  
def cursor\_position(event):  
 line, column = event.widget.index("insert").split(".")  
 print("当前光标位置：行{}，列{}".format(line, int(column) + 1))  
 label.config(text=f'第 {line} 行，第 {int(column) + 1} 列') # 更新行列  
  
window = tkinter.Tk() # 创建窗口对象

label = tkinter.Label(text='第 1 行，第 1 列') # 光标位置获显示  
label.pack(side=tkinter.BOTTOM, fill='both')  
  
text = tkinter.Text(window) # 文本域  
text.pack(expand=True, fill='both')  
text.focus\_set() # 获取组件焦点（解决文本域初始化不显示光标）  
  
# 捆绑键盘鼠标事件  
text.bind('<KeyPress>', cursor\_position) # 键盘按下触发  
text.bind('<KeyRelease>', cursor\_position) # 键盘释放触发  
text.bind('<ButtonPress>', cursor\_position) # 鼠标按下触发  
text.bind('<ButtonRelease>', cursor\_position) # 鼠标释放触发  
  
window.mainloop() # 运行窗口主循环

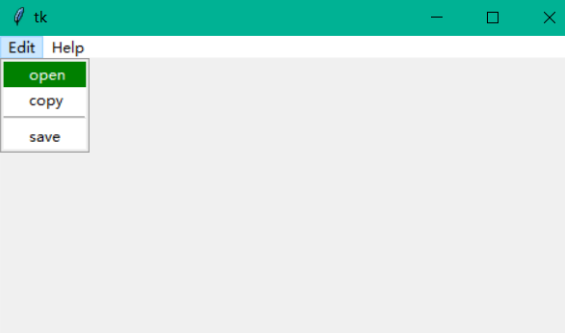
显示效果：



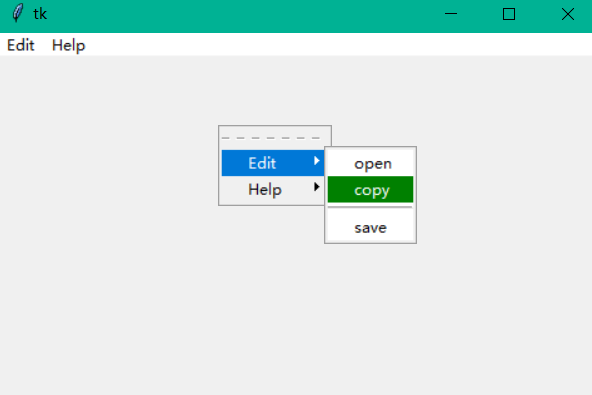
**★Menu菜单栏**

import tkinter  
  
def open\_func():  
 print("open file")  
  
def save\_func():  
 print("save file")  
  
def pop(event, menu): # 右键弹出菜单  
 menu.post(event.x\_root, event.y\_root)  
 # event.x是相对于窗口的坐标值，event.x\_root是相对屏幕的坐标值  
 # 而menu.post坐标是相对于屏幕的，所以使用event.x则从窗口外弹出了，使用event.x\_root才是从鼠标处弹出  
  
  
window = tkinter.Tk() # 创建窗口对象  
window.geometry("480x320") # 设置窗口大小，宽x高  
  
menu\_bar = tkinter.Menu(window) # 创建一个菜单，做菜单栏（menubar）  
menu\_cascade\_edit = tkinter.Menu(menu\_bar, tearoff=0, activebackground="green", activeforeground="white",  
 background="white", foreground="black") # 创建一个菜单，不分窗，tearoff=1表示分窗，即此菜单可拉出来变成一个可移动的独立弹窗  
# activebackground指定鼠标移入时的背景色，activeforeground指定鼠标移入时的前景色  
# background指定鼠标未移入时的背景色，foreground指定鼠标未移入时的前景色  
menu\_cascade\_help = tkinter.Menu(menu\_cascade\_edit, tearoff=0) # 创建一个菜单item，不分窗  
  
menu\_bar.add\_cascade(label="Edit", menu=menu\_cascade\_edit) # 将menu\_cascade\_edit添加为menu\_bar的子菜单（瀑布菜单类型）且起个入口名称：Edit  
menu\_bar.add\_cascade(label="Help", menu=menu\_cascade\_help) # 将menu\_cascade\_help添加为menu\_bar的子菜单（瀑布菜单类型）且起个入口名称：Help  
  
menu\_cascade\_edit.add\_command(label="open", command=open\_func) # 在瀑布菜单中加入子菜单项，关联触发函数  
menu\_cascade\_edit.add\_command(label="copy") # 在瀑布菜单中加入子菜单项  
menu\_cascade\_edit.add\_separator() # 在瀑布菜单中加入一条分割线  
menu\_cascade\_edit.add\_command(label="save", command=save\_func) # 在瀑布菜单中加入子菜单项，关联触发函数  
menu\_cascade\_help.add\_command(label="About")  
  
window.config(menu=menu\_bar) # 放置菜单栏到主窗口  
# menu\_bar.delete(0) # 删除第一个位置的菜单项  
window.bind("<Button-3>", lambda event: pop(event, menu\_bar)) # 监听鼠标右击事件  
  
window.mainloop() # 运行窗口主循环

显示效果：

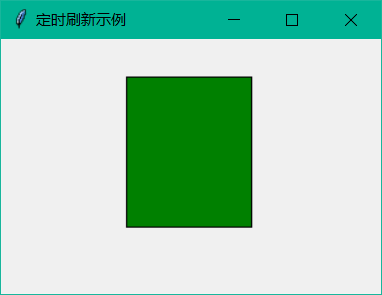


鼠标右击显示效果：



**★window.after定时刷新界面**

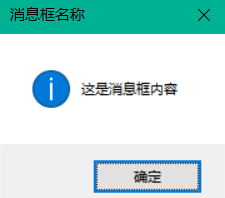
import tkinter  
import random  
  
def refresh(window, canvas): # 定义一个定时刷新方法  
 canvas.delete("all") # 清除画布  
 canvas.create\_rectangle(100, random.randint(9, 100), 200, 150, fill="green") # 绘制矩形  
 window.after(1000, refresh, window, canvas) # 注册下一次回调函数（可选）  
  
def test\_tk\_after():  
 window = tkinter.Tk()  
 window.title("定时刷新示例")  
 canvas = tkinter.Canvas(window, width=300, height=200)  
 canvas.pack()  
 after\_id = window.after(1000, refresh, window, canvas) # 等待1000毫秒，调用回调函数，给回调函数传的参数  
 # window.after\_cancel(after\_id)  
 # 运行主循环  
 window.mainloop()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_tk\_after()



**★messagebox消息提示/确认框**

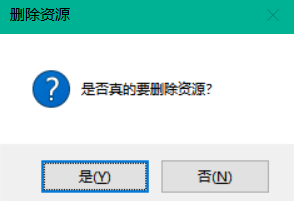
**★消息提示框**

from tkinter import messagebox  
  
def on\_button\_click():  
 messagebox.showinfo("消息框名称", "这是消息框内容")  
  
on\_button\_click()



**★消息确认框**

from tkinter import messagebox  
  
result = messagebox.askyesno("删除资源", f"是否真的要删除资源？")  
# 参数1为弹窗标题，参数2为弹窗内容，有2个按钮（是，否），点击"是"时返回True  
if result:  
 print("已删除资源")  
else:  
 print("用户取消了删除操作")



**★获取控件的子控件**

控件.winfo\_children()方法可获取控件的子控件

清空窗口里的所有控件

def claer\_tkinter\_window(window):  
 for widget in window.winfo\_children():  
 widget.destroy()

**★内置各种选择框（文件，颜色等）**

**★文件选择框**

from tkinter import filedialog  
  
file\_path = filedialog.askopenfilename(filetypes=[("Text files", "\*.txt"), ("All files", "\*.\*")])  
if not file\_path:  
 print("未选择文件")  
else:  
 print(file\_path)

#文件选择对话框返回的file\_path是一个字符串（选择的文件路径字符串）

**★文件保存对话框**

from tkinter import filedialog  
  
file\_xx = filedialog.asksaveasfile(title="保存到文件", filetypes=[("Text files", "\*.log"), ("All files", "\*.\*")], defaultextension=".log")  
if not file\_xx:  
 print("未选择文件")  
else:  
 # 保存到文本文件  
 all\_text\_n = "str"  
 with open(file\_xx.name, "a", encoding="utf8") as fileobj: # 追加，不存在文件则新建  
 fileobj.write(all\_text\_n)

#文件保存对话框返回的file\_xx是一个\_io.TextIOWrapper对象，需要使用.name属性获得其路径字符串

**★颜色选择框**

import tkinter  
from tkinter import colorchooser  
  
color = tkinter.colorchooser.askcolor()  
print(color)  
print(type(color[1]))

结果：

((192, 129, 180), '#c081b4') #返回tuple，tuple[0]为RGB的3个数值tuple；tuple[1]为颜色的RGB十六进制编码

<class 'str'> #tuple[1]为颜色的RGB十六进制编码，是一个字符串

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**★附录、常用基础算法**

**★排序算法**

**★冒泡排序法**

def bubble\_sort(listn): # listn为可变参数  
 for i in range(len(listn) -1): #元素数量为n，外循环只需要n-1次  
 for k in range(len(listn) -i -1): #内循环不会比较上一次最后一个元素  
 if listn[k] > listn[k +1]: #大的往后排，内循环结束后大的在最后一元素  
 tmp = listn[k] #以下为交换2个元素的值  
 listn[k] = listn[k + 1]  
 listn[k + 1] = tmp

**★选择排序法**

def select\_sort(listn): # listn为可变参数  
 for i in range(len(listn) - 1): #元素数量为n，外循环只需要n-1次  
 for k in range(i +1, len(listn)): #内循环不会比较上一次的第一个元素  
 if listn[i] > listn[k]: #把最小的元素放本次内循环最前面  
 tmp = listn[i] #以下为交换2个元素的值  
 listn[i] = listn[k]  
 listn[k] = tmp

**★斐波纳契数列**

def fibonacci(max):  
 first = 0  
 last = 1  
 while last < max:  
 print(last)  
 tmp = first  
 first = last  
 last += tmp # 后一数字是前2数字之和

结果： 1 1 2 3 5 8 13

**★殴几里德算法**

求num1和num2的最大公约数（最大公因数）

def gcd(num1, num2):  
 if num1 == 0 and num2 == 0:  
 return 0  
 r = num1 % num2  
 num1 = num2  
 num2 = r  
 while r != 0:  
 r = num1 % num2  
 num1 = num2  
 num2 = r  
 return num1

未完待续

**★字符编码**

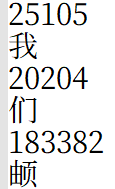
def hex\_to\_bytearray(hex\_str):  
 # 十六进制字符串 转为 bytearray列表  
 # input <str> , output <bytearray>  
 byte\_array\_length, mod = divmod(len(hex\_str), 2)  
 if mod != 0:  
 return None  
 byte\_array = bytearray(byte\_array\_length)  
 for i in range(byte\_array\_length):  
 byte\_array[i] = int(hex\_str[i \* 2:i \* 2 + 2], base=16)  
 return byte\_array

def byte\_to\_unicode(byte\_array):  
 # 将bytes/bytearray列表 转为 Unicode Rune字元列表  
 # input <bytearray> , output list<rune> , type(rune) == int  
 unicode\_runes = []  
 i = 0  
 while i < len(byte\_array):  
 if 0 <= byte\_array[i] < 128:  
 unicode\_rune = byte\_array[i]  
 i += 1  
 elif 191 < byte\_array[i] < 224:  
 unicode\_rune = ((byte\_array[i] & 0x1F) << 6) | (byte\_array[i + 1] & 0x3F)  
 i += 2  
 elif 223 < byte\_array[i] < 240:  
 unicode\_rune = ((byte\_array[i] & 0x0F) << 12) | ((byte\_array[i + 1] & 0x3F) << 6) | (  
 byte\_array[i + 2] & 0x3F)  
 i += 3  
 elif 239 < byte\_array[i] < 248:  
 unicode\_rune = ((byte\_array[i] & 0x07) << 18) | ((byte\_array[i + 1] & 0x3F) << 12) | (  
 (byte\_array[i + 2] & 0x3F) << 6) | (byte\_array[i + 3] & 0x3F)  
 i += 4  
 else:  
 return None  
 unicode\_runes.append(unicode\_rune)  
 return unicode\_runes

★使用：

byte\_a = hex\_to\_bytearray("E68891E4BBACF0ACB196")  
runes = byte\_to\_unicode(byte\_a)  
for i in runes:  
 print(i)  
 print(chr(i))

结果：



**★附录（ascii码）**

字符编码相关知识请参阅作者的“汉字标准及字符编码探究”

链接：https://limaofu.github.io/t3doc/汉字标准及字符编码探究-back-250928.docx

**★ASCII码表**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **数字** | **Hex** | **含义** | **数字** | **Hex** | **含义** | **数字** | **Hex** | **含义** | **数字** | **Hex** | **含义** | **数字** | **Hex** | **含义** |
| 0 | 00 | NUL | 26 | 1A | SUB | 52 | 34 | 4 | 78 | 4E | N | 104 | 68 | h |
| 1 | 01 | SOH | 27 | 1B | ESC | 53 | 35 | 5 | 79 | 4F | O | 105 | 69 | i |
| 2 | 02 | STX | 28 | 1C | FS | 54 | 36 | 6 | 80 | 50 | P | 106 | 6A | j |
| 3 | 03 | ETX | 29 | 1D | GS | 55 | 37 | 7 | 81 | 51 | Q | 107 | 6B | k |
| 4 | 04 | EOT | 30 | 1E | RS | 56 | 38 | 8 | 82 | 52 | R | 108 | 6C | l |
| 5 | 05 | ENQ | 31 | 1F | US | 57 | 39 | 9 | 83 | 53 | S | 109 | 6D | m |
| 6 | 06 | ACK | 32 | 20 | 空格 | 58 | 3A | : | 84 | 54 | T | 110 | 6E | n |
| 7 | 07 | BEL | 33 | 21 | ! | 59 | 3B | ; | 85 | 55 | U | 111 | 6F | o |
| 8 | 08 | BS | 34 | 22 | " | 60 | 3C | < | 86 | 56 | V | 112 | 70 | p |
| 9 | 09 | HT | 35 | 23 | # | 61 | 3D | = | 87 | 57 | W | 113 | 71 | q |
| 10 | 0A | LF | 36 | 24 | $ | 62 | 3E | > | 88 | 58 | X | 114 | 72 | r |
| 11 | 0B | VT | 37 | 25 | % | 63 | 3F | ? | 89 | 59 | Y | 115 | 73 | s |
| 12 | 0C | FF | 38 | 26 | & | 64 | 40 | @ | 90 | 5A | Z | 116 | 74 | t |
| 13 | 0D | CR | 39 | 27 | ' | 65 | 41 | A | 91 | 5B | [ | 117 | 75 | u |
| 14 | 0E | S0 | 40 | 28 | ( | 66 | 42 | B | 92 | 5C | \ | 118 | 76 | v |
| 15 | 0F | S1 | 41 | 29 | ) | 67 | 43 | C | 93 | 5D | ] | 119 | 77 | w |
| 16 | 10 | DLE | 42 | 2A | \* | 68 | 44 | D | 94 | 5E | ^ | 120 | 78 | x |
| 17 | 11 | DC1 | 43 | 2B | + | 69 | 45 | E | 95 | 5F | \_ | 121 | 79 | y |
| 18 | 12 | DC2 | 44 | 2C | , | 70 | 46 | F | 96 | 60 | ` | 122 | 7A | z |
| 19 | 13 | DC3 | 45 | 2D | - | 71 | 47 | G | 97 | 61 | a | 123 | 7B | { |
| 20 | 14 | DC4 | 46 | 2E | . | 72 | 48 | H | 98 | 62 | b | 124 | 7C | | |
| 21 | 15 | NAK | 47 | 2F | / | 73 | 49 | I | 99 | 63 | c | 125 | 7D | } |
| 22 | 16 | SYN | 48 | 30 | 0 | 74 | 4A | J | 100 | 64 | d | 126 | 7E | ~ |
| 23 | 17 | ETB | 49 | 31 | 1 | 75 | 4B | K | 101 | 65 | e | 127 | 7F | DEL |
| 24 | 18 | CAN | 50 | 32 | 2 | 76 | 4C | L | 102 | 66 | f |  |  |  |
| 25 | 19 | EM | 51 | 33 | 3 | 77 | 4D | M | 103 | 67 | g |  |  |  |

特殊字符表示：

|  |  |  |  |
| --- | --- | --- | --- |
| 转义 | hex十六进制 | vt表示 | 含义 |
| \07 | 07 | ^G | BEL 响铃 |
| \010 | 08 | ^H | BS 退格，光标左移一格 |
| \t | 09 | ^I | HT 水平制表符 |
| \v | 0B | ^K | VT 垂直制表符 |
| \r | 0D | ^M | CR 回车，光标移动到行首 |
| \n | 0A | ^J | LF 换行，光标移动到下一行 |
| \032 | 1A | ^Z | SUB EOF |
| \033 | 1B | ^[ | ESC |

1961年美国国家标准局(ANSI)制定了ASCII码（American Standard Code for Information Interchange，美国标准信息[交换码](http://www.baike.com/sowiki/%E4%BA%A4%E6%8D%A2%E7%A0%81?prd=content_doc_search" \o "交换码)），它已被国际标准化组织（ISO）定为国际标准，称为ISO 646标准，标准ASCII码为7位，扩充为8位（最高位为0）

7位二进制数可以表示(2^7)==128个数字（0到127），每个数字都唯一地对应一个字符，这些数字就是对应字符的编码，ASCII编码对应 控制字符及可打印字符：

0～32号及第127号（共34个）是控制字符或通讯专用字符（32号为空格）

33～126号（共94个）是可打印字符

ASCII码主要是给美国用的（英语文字），欧洲及亚洲其他国家的文字符号不在ACSII的收录范围内，于是他们把ASCII扩展了，使用8位的ASCII，把最高位置1，这样比原来的ASCII又多出128个编码可分配，不同的国家甚至厂商都有自己的标准，不利于规范，于是国际标准化组织（ISO）和国际电工委员会(IEC)联合制定了一系列8位字符集的标准：ISO 8859，全称ISO/IEC 8859

**ISO/IEC 8859-1** (Latin-1) 西欧语言

**ISO/IEC 8859-2** (Latin-2) 中欧语言

**ISO/IEC 8859-3** (Latin-3) 南欧语言。世界语也可用此字符集显示。

**ISO/IEC 8859-4** (Latin-4) 北欧语言

**ISO/IEC 8859-5** (Cyrillic) 斯拉夫语言

**ISO/IEC 8859-6** (Arabic) 阿拉伯语

**ISO/IEC 8859-7** (Greek) 希腊语

**ISO/IEC 8859-8** (Hebrew) 希伯来语(视觉顺序)

**ISO/IEC 8859-8-I** (Hebrew) 希伯来语(逻辑顺序)

**ISO/IEC 8859-9** (Latin-5 或 Turkish) 它把Latin-1的冰岛语字母换走，加入土耳其语字母

**ISO/IEC 8859-10** (Latin-6 或 Nordic) 北日耳曼语支，用来代替Latin-4

**ISO/IEC 8859-11** (Thai) 泰语，从泰国的 TIS620 标准字集演化而来

**ISO/IEC 8859-13** (Latin-7 或 Baltic Rim) 波罗的语族

**ISO/IEC 8859-14** (Latin-8 或 Celtic) 凯尔特语族

**ISO/IEC 8859-15** (Latin-9) 西欧语言，加入Latin-1欠缺的芬兰语字母和大写法语重音字母，以及欧元符号

**ISO/IEC 8859-16** (Latin-10) 东南欧语言。主要供罗马尼亚语使用，并加入欧元符号