### 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 |
| 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 |  |  |  |

**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 ~ 2023-12-04

**★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，语句用;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("hellow")

# chmod +x test.py #添加可执行权限

#脚本文件里要指定以下2行

#!/usr/bin/env python3

# 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/

**★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 |

**★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



**★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 → Type 取消勾选



**★第2章、Centos7源码安装python3**

Centos7默认自带python2.7.5版本，**无pip**

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

**安装依赖：**

# 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，pefix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 | as | nonlocal |  |
| and | exec | not | assert | finally | or |
| break | for | pass | class | from | print |
| continue | global | raise | def | if | return |
| del | import | try | elif | in | while |
| else | is | with | except | lambda | yiel |

**★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 |
| set | 无序 不重复（自动删除重复项） 集合 py3新增 |
| dictionary | 无序 字典 {键值对 集合} 关键字必须互不相同 |
| 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**： Number String Tuple bytes

**可变类型**： List Dictionary Set bytearray

set 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 {}

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

**★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字符串大小写转换及替换过滤拆分拼接**

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

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

**★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，则结果为n.0

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

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

**★条件语句（python不支持switch）**

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

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

**★整除**

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，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，则返回float

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

**★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

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

**★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

三维类推：

matrix = [ [ [0 for i in range(3)] for i in range(4) ] for i in range(2) ]

**#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

**★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])

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)

**★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

**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

**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

**★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

**★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

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**

**1.定义函数**

def fun\_name(参数):

"描述"

语句

return xx #若未指定return，则默认返回None

**2.参数传递**

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 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

**★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

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**

建议使用 "字符串".format(argxx) 的新式字符串格式化函数，返回格式化后的字符串

.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) )

**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))

**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

print("我在{:0>6}号".format( n ))

**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"str"**

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

name = "cof-lee"  
print(f"这是{name}")

结果：

这是cof-lee

**★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**

**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

\_\_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

**★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。

**①捕获异常可用try, except语句**

try:  
 #要执行的语句  
except xx类: #可以不匹配具体的错误类  
 #要执行的语句  
except yy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

**★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：

|  |  |  |
| --- | --- | --- |
| 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）

\_\_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，只要它自己执行时才运行这里的代码  
else:  
 #来自另一程序的调用

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)

★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 #在导入时指定不生成字节码

**★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【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：

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

如果没有指定的模块，则报错



**★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



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



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里





**打包模块包：**

现在需要把源程序目录下的cofpkg模块包打包成一个安装包



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



压缩包内容：



**★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命令行指定软件源**

#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



#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

**★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**

请见第九章 文件操作 ⑤二进制文件操作，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

 #日期有05

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

 #日期就一个 5

**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

**★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

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

**★sys模块**

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='')

**★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

**★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

**★re正则匹配模块**

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

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" " 指定里面的字符不识别为python转义字符

flags修饰符可用 | 或运算连接多个修饰符：

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

re.search()返回值为一个object，有个span属性，是一个tuple，表示匹配到的子字符串et在目标字符串str里的具体位置，含头不含尾，从0开始。



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（不含）



re.findall(正则表达式, 要搜索的字符串, flags=修饰符) #返回一个含所有的匹配字符串的 list



在

**★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)

**★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)

**★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



**★csv模块**

在

**★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

**★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：

**★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

#离线安装时可 按以下顺序安装各依赖包（假如下载在paramiko-win10-py310/目录下）

pip install paramiko-win10-py310/pycparser-2.21-py2.py3-none-any.whl

pip install paramiko-win10-py310/cffi-1.15.1-cp310-cp310-win\_amd64.whl

pip install paramiko-win10-py310/cryptography-37.0.4-cp36-abi3-win\_amd64.whl

pip install paramiko-win10-py310/PyNaCl-1.5.0-cp36-abi3-win\_amd64.whl

pip install paramiko-win10-py310/six-1.16.0-py2.py3-none-any.whl

pip install paramiko-win10-py310/bcrypt-3.2.2-cp36-abi3-win\_amd64.whl

pip install paramiko-win10-py310/paramiko-2.11.0-py2.py3-none-any.whl

#使用：

import paramiko  
import time  
  
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  
client1.connect(hostname="sysyear.top", port=22, username="cof", password="passwdxx", timeout=20)  
stdin, stdout, stderr = client1.exec\_command("ls -l")  
time.sleep(1) #需要暂停一下，否则主程序退出就会报错  
  
# 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

**★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

#离线安装时可 按以下顺序安装各依赖包（假如下载在paramiko-win10-py310/目录下）

pip install requests-win10-py310/pycparser-2.21-py2.py3-none-any.whl

pip install requests-win10-py310/cffi-1.15.1-cp310-cp310-win\_amd64.whl

pip install requests-win10-py310/cryptography-37.0.4-cp36-abi3-win\_amd64.whl

pip install requests-win10-py310/PyNaCl-1.5.0-cp36-abi3-win\_amd64.whl

pip install requests-win10-py310/six-1.16.0-py2.py3-none-any.whl

#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 requests  
  
myurl = "http://sysyear.top:4600"  
  
myheaders = {  
 "Accept": "application/json",  
 "Content-Type": "application/json;charset=utf8"  
}  
  
mydata = "dddd"  
  
response = requests.post(myurl, headers=myheaders, data=mydata, verify=False)  
  
response.encoding = "utf-8" # 指定返回数据的文本编码

**★xlrd（xls文件处理）**

d

**★openpyxl（xlsx文件处理）**

在

**★第16章、多进程与多线程**

**★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()

**★多进程同步之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()

**★多进程同步之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()

**★多进程同步之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()

**★多进程同步之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()

**★多进程之线程队列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()

**★多进程之进程池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()

**★多线程同步之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()

**★多线程同步之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()

**★多线程同步之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

**★多线程同步之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

**★多线程之线程队列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。

**★多线程之线程池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

**★第17章、网络通信**

**★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-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 \_thread  
import socket  
  
def my\_thread\_func(con, addr): # 定义线程函数  
 print("本服务端为：",con.getsockname())  
 print("客户端的地址为：",addr)  
 con.send("发给客户端的信息".encode("utf8"))  
 con.close()  
  
host = "0.0.0.0"  
port = 1234  
sock1 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 创建socket

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()  
 try:  
 \_thread.start\_new\_thread(my\_thread\_func, (con, addr))  
 except Exception as e:  
 print(e)

**★第18章、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

**★第n章、常用杂学算法**

**★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：



**★网络IP地址相关cofnet模块**

作者：Cof-Lee，即本文档的作者

#!/usr/bin/env python3  
# coding=utf-8  
# author: Cof-Lee  
# update: 2023-11-30  
# module name: cofnet  
# 本模块使用cof-lee开源协议 v1.0  
# 所有条款及内容如下：  
# （1）无担保：作者不保证源代码内容的准确无误，亦不承担由于使用此源代码所导致的任何后果  
# （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/再创作此源代码，无需任何附加条件  
  
*"""  
术语解析:  
maskint 掩码数字型 ，如 24 ，子网掩码位数， 类型: int  
maskbyte 掩码字节型 ，如 255.255.255.0 ，子网掩码， 类型: str  
ip ip地址称，如 10.1.1.2 ，不含掩码 类型: str  
netseg 网段，如 10.1.0.0 ，不含掩码 类型: str  
cidr 地址块，网段及掩码位数 ，如 10.1.0.0/16 类型: str  
"""*def is\_ip\_addr(input\_str):  
 # 判断 输入字符串 是否为 ip地址，返回bool值，是则返回True，否则返回False  
 # input <str> , output <bool>  
 seg\_list = input\_str.split(".")  
 if len(seg\_list) != 4:  
 return False  
 if seg\_list[0].isdigit():  
 if 0 > int(seg\_list[0]) or int(seg\_list[0]) > 255:  
 return False  
 if seg\_list[1].isdigit():  
 if 0 > int(seg\_list[1]) or int(seg\_list[1]) > 255:  
 return False  
 if seg\_list[2].isdigit():  
 if 0 > int(seg\_list[2]) or int(seg\_list[2]) > 255:  
 return False  
 if seg\_list[3].isdigit():  
 if 0 > int(seg\_list[3]) or int(seg\_list[3]) > 255:  
 return False  
 else:  
 return True  
 else:  
 return False  
  
  
def is\_cidr(input\_str):  
 # 判断 输入字符串 是否为 cidr地址块，返回bool值，是则返回True，否则返回False  
 # 输入 "10.99.1.0/24" 输出 True  
 # 输入 "10.99.1.1/24" 输出 False ，不是正确的cidr地址块写法，24位掩码，的最后一字节必须为0  
 # input <str> , output <bool>  
 seg\_list = input\_str.split(".")  
 if len(seg\_list) != 4:  
 return False  
 if seg\_list[0].isdigit():  
 if 0 > int(seg\_list[0]) or int(seg\_list[0]) > 255:  
 return False  
 if seg\_list[1].isdigit():  
 if 0 > int(seg\_list[1]) or int(seg\_list[1]) > 255:  
 return False  
 if seg\_list[2].isdigit():  
 if 0 > int(seg\_list[2]) or int(seg\_list[2]) > 255:  
 return False  
 if seg\_list[3].isdigit():  
 return False  
 seg\_list2 = seg\_list[3].split("/")  
 if len(seg\_list2) == 2:  
 if seg\_list2[1].isdigit():  
 if 0 > int(seg\_list2[1]) or int(seg\_list2[1]) > 32:  
 return False  
 else:  
 return False  
 seg\_list3 = input\_str.split("/")  
 seg\_list4 = seg\_list3[0].split(".")  
 ip\_mask\_int = int(seg\_list4[0]) << 24 | int(seg\_list4[1]) << 16 | int(seg\_list4[2]) << 8 | int(seg\_list4[3])  
 ip\_mask\_int\_and = ip\_mask\_int & (0xFFFFFFFF << (32 - int(seg\_list3[1])))  
 if ip\_mask\_int != ip\_mask\_int\_and:  
 return False  
 return True  
  
  
def is\_ip\_range(input\_str):  
 # 判断 输入字符串 是否为 ip地址范围，返回bool值，是则返回True，否则返回False  
 # 输入 "10.99.1.33-55" 输出 True  
 # 输入 "10.99.1.22-10" 输出 False ，不是正确的地址范围，首ip大于了尾ip  
 # input <str> , output <bool>  
 seg\_list = input\_str.split(".")  
 if len(seg\_list) != 4:  
 return False  
 if seg\_list[0].isdigit():  
 if 0 > int(seg\_list[0]) or int(seg\_list[0]) > 255:  
 return False  
 if seg\_list[1].isdigit():  
 if 0 > int(seg\_list[1]) or int(seg\_list[1]) > 255:  
 return False  
 if seg\_list[2].isdigit():  
 if 0 > int(seg\_list[2]) or int(seg\_list[2]) > 255:  
 return False  
 if seg\_list[3].isdigit():  
 return False  
 seg\_list2 = seg\_list[3].split("-")  
 if len(seg\_list2) == 2:  
 if seg\_list2[0].isdigit():  
 if 0 > int(seg\_list2[0]) or int(seg\_list2[0]) > 255:  
 return False  
 if seg\_list2[1].isdigit():  
 if 0 > int(seg\_list2[1]) or int(seg\_list2[1]) > 255:  
 return False  
 if int(seg\_list2[0]) >= int(seg\_list2[1]):  
 return False  
 return True  
 else:  
 return False  
  
  
def maskint\_to\_maskbyte(maskint):  
 # 将子网掩码数字型 转为 子网掩码字节型，例如：  
 # 输入 16 输出 "255.255.0.0"  
 # 输入 24 输出 "255.255.255.0"  
 # input <int> , output <str>  
 if maskint < 0 or maskint > 32:  
 raise Exception("子网掩码数值应在[0-32]", maskint)  
 mask = [0, 0, 0, 0]  
 i = 0  
 while maskint >= 8:  
 mask[i] = 255  
 i += 1  
 maskint -= 8  
 if i < 4:  
 mask[i] = 255 - (2 \*\* (8 - maskint) - 1)  
 mask\_str\_list = map(str, mask)  
 return ".".join(mask\_str\_list)  
  
  
def local\_mask\_seg\_to\_cidr(mask\_seg):  
 # 将掩码其中一个字节的数字 转为 二进制数最开头的1的个数, 例如：  
 # 输入 "192" 输出 2 ，即 1100 0000  
 # 输入 "248" 输出 5 ，即 1111 1000  
 # 输入 "255" 输出 8 ，即 1111 1111  
 # input <str> , output <int>  
 mask\_seg\_1\_number = 0  
 mask\_seg\_int = int(mask\_seg)  
 while mask\_seg\_int != 0:  
 mask\_seg\_1\_number += 1  
 mask\_seg\_int = (mask\_seg\_int << 1) & 0xFF  
 return mask\_seg\_1\_number  
  
  
def maskbyte\_to\_maskint(maskbyte):  
 # 将子网掩码字节型 转为 子网掩码数字型，例如：  
 # 输入 "255.255.255.0" 输出 24  
 # 输入 "255.255.0.0" 输出 16  
 # input <str> , output <int>  
 if not is\_ip\_addr(maskbyte):  
 raise Exception("不是正确的子网掩码,E1", maskbyte)  
 mask\_seg\_list = maskbyte.split(".")  
 mask\_seg\_index = 0  
 maskint = 0  
 while mask\_seg\_list[mask\_seg\_index] == "255":  
 maskint += 8  
 mask\_seg\_index += 1  
 if mask\_seg\_index == 4:  
 break  
 if mask\_seg\_index < 4 and mask\_seg\_list[mask\_seg\_index] != "":  
 maskint += local\_mask\_seg\_to\_cidr(mask\_seg\_list[mask\_seg\_index]) # 依赖上面的 local\_mask\_seg\_to\_cidr()  
 if maskbyte != maskint\_to\_maskbyte(maskint):  
 raise Exception("不是正确的子网掩码,E2", maskbyte)  
 return maskint  
  
  
def ip\_mask\_to\_int(ip\_or\_mask):  
 # 将 ip地址或掩码byte型 转为 32 bit的数值，例如：  
 # 输入 "255.255.255.0" 输出 4294967040  
 # 输入 "192.168.1.1" 输出 3232235777  
 # input <str> , output <int>  
 if not is\_ip\_addr(ip\_or\_mask):  
 raise Exception("不是正确的ip地址或掩码", ip\_or\_mask)  
 seg\_list = ip\_or\_mask.split(".")  
 ip\_mask\_int = int(seg\_list[0]) << 24 | int(seg\_list[1]) << 16 | int(seg\_list[2]) << 8 | int(seg\_list[3])  
 return ip\_mask\_int  
  
  
def int32\_to\_ip(int32):  
 # 将 32bit数值 转为 ipv4地址，例如:  
 # 输入 174260481 输出 "10.99.1.1"  
 # input <int> , output <str>  
 if int32 < 0 or int32 > 4294967295:  
 raise Exception("ip地址数值应在[0-4294967295]", int32)  
 ipaddress = [0, 0, 0, 0]  
 ipaddress[0] = 0xFF & (int32 >> 24)  
 ipaddress[1] = 0xFF & (int32 >> 16)  
 ipaddress[2] = 0xFF & (int32 >> 8)  
 ipaddress[3] = 0xFF & int32  
 ipaddress\_str\_list = map(str, ipaddress)  
 return ".".join(ipaddress\_str\_list)  
  
  
def get\_netseg\_int(ip, maskintorbyte):  
 # 根据 子网掩码 获 取ip地址的 网段（int值），子网掩码可为int型或byte型，例如：  
 # 输入 "10.99.1.1","24" 输出 174260480  
 # 输入 "10.99.1.1","255.255.255.0" 输出 174260480  
 # input <str,int/str> , output <int>  
 if not is\_ip\_addr(ip):  
 raise Exception("不是正确的ip地址,E1", ip)  
 maskintorbyte\_seg = str(maskintorbyte).split(".")  
 if len(maskintorbyte\_seg) == 1:  
 if int(maskintorbyte\_seg[0]) < 0 or int(maskintorbyte\_seg[0]) > 32:  
 raise Exception("子网掩码数值应在[0-32]", maskintorbyte\_seg)  
 else:  
 maskint2bin = 0xFFFFFFFF << (32 - int(maskintorbyte\_seg[0]))  
 return ip\_mask\_to\_int(ip) & maskint2bin  
 if len(maskintorbyte\_seg) == 4:  
 if not is\_ip\_addr(maskintorbyte):  
 raise Exception("不是正确的掩码,E2", maskintorbyte)  
 maskint2bin = 0xFFFFFFFF << (32 - int(maskbyte\_to\_maskint(maskintorbyte)))  
 return ip\_mask\_to\_int(ip) & maskint2bin  
 else:  
 raise Exception("不是正确的掩码,E3", maskintorbyte)  
  
  
def get\_netseg\_byte(ip, maskintorbyte):  
 # 根据 子网掩码 获 取ip地址的 网段（byte值），子网掩码可为int型或byte型，例如：  
 # 输入 "10.99.1.1","24" 输出 10.99.1.0  
 # 输入 "10.99.1.1","255.255.255.0" 输出 10.99.1.0  
 # 依赖上面的2个函数: get\_netseg\_int() 以及 int32\_to\_ip()  
 # input <str,int/str> , output <str>  
 return int32\_to\_ip(get\_netseg\_int(ip, maskintorbyte))  
  
  
def is\_ip\_in\_cidr(ip, cidr):  
 # 判断 ip地址 是否在 网段cidr内，此ip是否属于某网段地址块，返回bool值: True表示ip在网段内，False不在网段内  
 # 输入 "10.99.1.1","10.99.1.0/24" 输出 True  
 # 输入 "10.99.3.1","10.99.1.0/24" 输出 False  
 # input <str, str> , output <bool>  
 if not is\_ip\_addr(ip):  
 raise Exception("不是正确的ip地址,E1", ip)  
 if not is\_cidr(cidr):  
 raise Exception("不是正确的cidr地址块,E2", cidr)  
 netseg\_maskint = cidr.split("/")  
 netseg = netseg\_maskint[0]  
 maskint = netseg\_maskint[1]  
 ipnetsegint = get\_netseg\_int(ip, maskint)  
 netsegint = get\_netseg\_int(netseg, maskint)  
 if ipnetsegint == netsegint:  
 return True  
 else:  
 return False  
  
  
def is\_ip\_in\_net\_maskbyte(ip, net, maskbyte):  
 # 判断 ip地址 是否在 网段 net maskbyte内，是否属于某网段地址块，返回bool值: True表示ip在网段内，False不在网段内  
 # 输入 "10.99.1.1","10.99.1.0","255.255.255.0" 输出 True  
 # 输入 "10.99.3.1","10.99.1.0","255.255.255.0" 输出 False  
 # input <str, str, str> , output <bool>  
 if not is\_ip\_addr(ip):  
 raise Exception("不是正确的ip地址,E1", ip)  
 if not is\_ip\_addr(net):  
 raise Exception("不是正确的网段,E2", net)  
 if not is\_ip\_addr(maskbyte):  
 raise Exception("不是正确的掩码,E3", maskbyte)  
 ipnetsegint = get\_netseg\_int(ip, maskbyte)  
 netsegint = get\_netseg\_int(net, maskbyte)  
 if ipnetsegint == netsegint:  
 return True  
 else:  
 return False  
  
  
def is\_ip\_in\_range(targetip, start\_ip, end\_ip):  
 # 判断 ip地址 是否在 ip地址范围内，返回bool值: True表示ip在ip-range内，False不在ip-range内  
 # 输入 "10.99.1.88","10.99.1.1","10.99.2.22" 输出 True  
 # 输入 "10.99.1.88","10.99.1.1","10.99.1.22" 输出 False  
 # input <str, str, str> , output <bool>  
 if not is\_ip\_addr(targetip):  
 raise Exception("不是正确的ip地址,E1", targetip)  
 if not is\_ip\_addr(start\_ip):  
 raise Exception("不是正确的ip地址,E2", start\_ip)  
 if not is\_ip\_addr(end\_ip):  
 raise Exception("不是正确的ip地址,E3", end\_ip)  
 if ip\_mask\_to\_int(end\_ip) >= ip\_mask\_to\_int(targetip) >= ip\_mask\_to\_int(start\_ip):  
 return True  
 else:  
 return False  
  
  
# ################################# end of module's function ##############################  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("this is cofnet.py")

★

**★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/汉字标准及字符编码探究-back-0921.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 |  |  |  |

\t 09 HT

\r 0D CR

\n 0A LF

EOF 1A SUB

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