# Python使用技巧

# 一/概念

python有指针的概念，如果多次初始化一个类，python会指向那个变量，并不会再创建一个相同的对象，使用id()方法即可验证

引用传参

变量 值传递，

数值 指针传递

# 二/print方法

“x”\*10 // 连续打印10个x，用于一些需要重复打印的地方

Print(“name=%s,age=$d”%(name,age))

//打印不换行

Print( “\*\*\*\*”,end=““)；

//打印换行

Print(“”)

//字符串连接新方式

“===%s===”%(a+b)

A,b是变量，

Get()与[] :get不会报错

.Startwith（）以多少开头

.Endwith()以结尾

# 三/下标和切片

字符串可以通过下标获取字符

Nam[2:6]截取第2个(包含)到第5个

Name[-1::-1]倒叙

# 四/全局变量

1、列表，字典不用加global 在函数外定义就是全局变量

2、函数内使用global，代表要修改全局变量，在函数内不修改变量内容，可直接读取变量

# 五/不定长参数

#args 是一个变量名，\*类似于指针

Def function\_name(one, two, **\*args**):

Print(args);//这是一个元组

Function\_name(1,2,3,4,5,6,7)

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

Def function\_name(one, two, **\*\*keys**):

Print(keys) //这是一个字典

Function\_name(1,2,’aa’=3,’bb’=4)

首先满足没有命名参数的，再满足带有命名参数的变量

# 六/拆包

创建进程，线程时使用

形参传递元组里的内容

Def function\_name(\*args, \*\*args)

A=(1,2,3,4,5)

B=(‘a’=1,’b’=2)

Function\_name(\*A,\*\*B)

# 七/匿名函数

# 八/魔术方法

**\_\_new\_\_ 在init之前调用，类的实例化方法，并返回该实例**

**\_\_init\_\_ 初始化时运行，实例的构造方法， 接受类的实例(self)并对其进行构造**

Class = one(one,two)在类中定义参数

Def \_\_init\_\_(self,one,two) 方法中自动接受参数

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

**\_\_str\_\_ 对象的描述信息,打印对象调用**

Def \_\_str\_\_(self)

Return “ssss”

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

**\_\_main\_\_**

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

**\_\_name\_\_ 执行文件的名称**

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

**\_\_all\_\_ 禁止调用all以外的函数，对类也有效**

\_\_all\_\_ = [“one”,”two”]

Def one()

Def two()

Def three() //不能调用

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

**\_\_slots\_\_ 限定类中只能有几个属性**

\_\_slots\_\_ = [“name”,”age”]

# 九/面向对象

注：

**继承**会继承父类的\_\_init\_\_方法，继承就会执行**父类的\_\_init\_\_**

**如果父类定义了\_\_init\_\_子类要使用super().\_\_init\_\_()方法，确保\_\_init\_\_方法执行，防止父类\_\_init\_\_方法中初始化一些变量**

## 定义

**对象中的self指的是当前类本身，self名字可便**

Class Name:

Def function\_name(self):

创建对象：Tom = Name()

调用：Tom.function\_name()

## 公有方法

**定义：**Def name()

**调用：**

ClassName.name() #类外

**Self**.name() #类内

## 私有方法

**定义：**Def \_\_name() #两个下划线

**调用：**Self.\_\_name()

## 继承

1. 定义：

Class cat(object): #新式类

Class Dog(Anmal): #继承Anmal类

Def ……

2、在子类调用父类方法

Anmal.drak(self) #self不可少

Super().drak() #super一个方法指向父类

## 实例属性、类属性

**<类属性不用实例化，就能调用>**

**<实例属性必须实例化>**

**--类外部修改类属性需要这样修改className.name=”123”**

**--不然修改的是实例属性**

**修改：**

**self修改的只会是实例属性**

**className.name=”123” #修改类属性**

**调用：**

**类属性与实例属性同名时，先读取实例属性，没有则找类属性**

**<实例属性优先级高，同时存在相同名称的类属性和实例属性，只读取实例属性>**

类属性：

Class Tool(object):

name=“one”

实例属性：

Class Tool(object):

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

self.nameo = “one”

## 类方法

<cls只调用类属性>

<self可以调用实例属性，实例方法>

<静态方法，不用额外参数>

Class Name(object):

Name = “ss”

@classmethod #类方法定义

Def function\_name(cls): #cls代表类

Cls.name

@staticmethod

Def function\_one(): #这里不用填写self,cls

#静态方法

#调用第一种

Name.function\_name() //类方法调用

#调用第二种

One = Name()

One.function\_name() //实例方法调用

## 单例模式

Class dog(object):

\_\_interface = none

Def \_\_new\_\_(cls):

If cls.\_\_interface == none:

Cls.\_\_interface = object.\_\_new\_\_(cls)

Return cls.\_\_interface

Else:

Return cls.\_\_interface

创建多次，但操作的都是一个对象

## 只初始化1次\_\_init\_\_

class dog(object):

\_\_first\_init = True

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

if self.\_\_first\_init:

SesActive.\_\_first\_init = False

# 十/异常处理

Try:

Xxxxx

Xxxxx

Except 错误类型:

…….

Except 错误类型:

…….

Except (NameError, filenotfounderror):

……..

Except exception as xxx:

捕获所有异常xxx原来的变量名

Else：

没有异常打印

Finally：

没有错打印

错误类型是特定的

# 十一/模块

系统目录->

## 同级目录

**|--Bao**

**|--Module\_one.py**

**|--Module\_two.py**

1. Import sys #导入模块、导入系统模块

2、import Module\_one #导入同级文件

3、From 文件名 import 函数名

from Module\_one import one #在module\_two中

4、Import Model\_one as aa #别名

#自己调用执行one，导入不执行。主要用于测试

If \_\_name\_\_ == “\_\_main\_\_”

Main() #主函数

## 调用包里的类

目录

**|--main.py**

**|--Bao**

**|--\_\_init\_\_.py**

**|--Module\_one.py**

**|--Module\_two.py**

1. -------->外部导入包内文件

**\_\_init\_\_.py:**

\_\_all\_\_ = [‘Module\_one’] **#只允许调用module\_one**

**Main.py:**

From ***Bao*** import \* # \_\_init\_\_.py中需要导入各个文件

from **Bao** import Module\_one

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

**Module\_one**.functionName() #调用

1. -------->包内，同级目录导入

\_\_init\_\_.py: #可以为空

\_\_all\_\_=[‘Module\_one’]

**Module\_one.py:**

from **Bao** import Module\_two

import **Module\_two**

1. ------->通过*\_\_init\_\_导入文件*

**\_\_init\_\_.py**:

\_\_all\_\_ = [‘Module\_one’]

From . import Module\_one # . 只在\_\_init\_\_.py中使用

**Main.py**:

Import ***Bao***

***Bao.Module\_one***.functionName()

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

Mo = ***Bao.Module\_one***.ClassName() #包名.模块名/文件名.类名

1. -------->导入\_\_init\_\_.py中创建的全局变量参数

**\_\_init\_\_.py:**

from flask import Flask

flask\_app = Flask(\_\_name\_\_) #在\_\_init\_\_.py生成一个flask\_app参数

**Module\_one.py:**

from Bao import flask\_app

import Bao

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

flask\_app.run()

Bao.flask\_app.run() # import Bao

**Main.py:**

from **Bao** import flask\_app

import Bao

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

flask\_app.run()

Bao.flask\_app.run()

# 十二/包

1. 创建setup.py文件，复制代码
2. Setup.py文件中添加py\_modules添加模块文件名
3. 运行命令python3 setup.py build 生成build文件目录
4. 运行命令python3 setup.py sdist 打包文件

包的安装

1. 解压包
2. 进入解压目录，运行python3 setup.py install

包内导入需要填写**包名.模块名**详细阅读有道云笔记包内容

# 十三/迭代器and生成器

共有方法：

next()/send()/\_\_next\_\_()/\_\_iter()\_\_/

## 1、列表生成式

A = [ X for x in range(10) ] #列表

A =(x for x in range(10)) #列表生成式

## 2、函数生成器(特殊的迭代器）

#生成方式比较复杂，可以填写一个函数

**#但函数中出现yield 就代表这个函数是一个生成器**

Def create():

A,b = 0,1

For I in range(50):

Yield b #一开始要打印的数字，根据这个数进行累积

A,b = a,a+b

当生成器不能再生成数据时，会产生一个异常stoplteration错误

使用for循环生成器可以避免这个错误

***For bb in a:***

***Print(bb)*** #不使用next，直接打印

## 3、迭代器

**（1）可迭代对象（iterable）**

一类是集合数据类型，如 list 、 tuple 、 dict 、 set 、 str 等；

一类是 generator ，包括生成器和带 yield 的generator function。

这些可以直接作用于 for 循环的对象统称为可迭代对象： Iterable 。

1. **itertor**

**所有可迭代对象使用iter()方法转化为迭代器**

1. **迭代器**

任何实现了\_\_iter\_\_和\_\_next\_\_()**（python2中实现next()）**方法的对象都是迭代器

\_\_iter\_\_返回迭代器自身

\_\_next\_\_返回容器中的下一个值

class Fib:

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

self.prev = 0

self.curr = 1

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

return self

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

value = self.curr

self.curr += self.prev

self.prev = value

return value

>>> f = Fib()

>>> list(islice(f, 0, 10))

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

## 4、send使用

如果yield的值返回给一个变量，例

Def create()

A = 0

For bb in range(5)

Tem = yield bb

Print Tem #这里打印不出来，要使用send(“值”)这样的方法进行打印

A = create()

a.send(“xxx”) ##这里会打印生成器的值，还会打印send的值

#第一次使用send会造成程序崩溃，后几次运行才可以，或者第一次传值none;a.send(“none”)

## 5、生成器有什么用呢

**1/协程（多任务其中一种，进程，线程，协程最快）**

Def t1():

While True:

Print(“1”)

Yiled None #暂停

Def t2():

While True:

Print(“2”)

Yiled none #暂停

Te = t1()

Tt =t2()

While True

Te.\_\_next\_\_()

Tt.\_\_next\_\_()

# 十四/闭包 and 装饰器

## 闭包

（1）函数中包含一个函数

（2）**被包含的**函数用到了**外面函数**的变量

def line\_conf(a, b): #定义闭包

def line(x):

return a\*x + b #使用了函数外的参数，资源不会被释放

return line

调用方法

**line1** = line\_conf(1, 1) #调用闭包 **return line**

print(**line1**(5)) #调用闭包 return a\*x +b

## 装饰器

1. 当一个函数加载装饰器的时候，执行流程就是装饰器定义
2. timefun装饰器名称
3. func 指向foo函数
4. wrappedfunc 自定义函数名,\*args,\*\*kargs是foo的参数
5. func(\*args,\*\*kargs) 在装饰器中调用foo函数
6. return func(\*args,\*\*kargs) 执行一次foo函数

def timefun(func): #timefun装饰器名称，func是要返回的

def wrappedfunc(\*args, \*\*kargs): #名称可自定义

.........

return func(\*args, \*\*kwargs) #通用性更好

return wrappedfunc #这个名称要与自定义名称一样

@timefundef #调用

foo(a, b, c):

print(a+b+c)

**注：**

**装饰器要么是函数，要么是一个类，不可能是一个类中的函数。**

## 类装饰器

# 十五/元类

创建类的东西-》元类，type()

Type(“名称”,(继承),{方法:函数}

# 十六/动态语言

# 十七/垃圾回收（GC）

## 大整数对象池，小整数对象池

小整数：-5到256在内存中早定义完好

大整数：python对其开启新内存

## intern机制

相同的字符串指向同一块变量

## 引用计数

Python垃圾处理机制的主要方式

缺点：循环引用内存不能被释放；python使用隔代回收进行解决

## 隔代回收

Python将对象都挂在链表中，每当计数达到预定的时候，就会进行一次清理

## 注意：

类中默认的\_\_del\_\_方法是垃圾回收机制，如果要重写\_\_del\_\_方法需要调用父类的\_\_del\_\_方法进行清除垃圾

# 十八/内建属性

在获得属性时进行调用，返回值时返回类中的属性

调用过程，先将字符串

Def \_\_getattribute\_\_(self,obj)

Return ….. #自定义返回值

return object.\_\_getattribute\_\_(self,obj) #系统默认返回的属性

# 十九/内建函数：

Lambda x:x+x,[1,2,3]

X = 1+1,x=2+2,x=3+3

Lambda x,y:x+y,[1,2,3],[3,2,1]

X,y = 1+3,xy=2+2,xy=3+1

Map

对指定方法进行映射

Filter

对指定的数据进行过滤

Filter（函数，数据）

对filter数据进行过滤，经函数判断为真的时候才保留

Reduce

对函数进行累积

# 二十/集合

1/去重

A=[1,2,3,4,5,6,1,2,3]

B= set(A)#去重

A=list(B)#转换为列表

A=’bcde’

B = Set(A)#将字符串转换为集合

C= {4，2，3}

A&B交集 A有B也有的

A|B 并集 所有不重复的

A-B 差集

A^B 对称差集只存在其中

# 二十一/Functools

Wraps显示以前的help信息

# 二十二/线程和进程

程序：保存在计算机中的代码

进程：运行起来的程序即为进程，从操作系统中获得了资源

线程：程序运行时代码执行的进度，cpu调度的最小单位

## 进程

### 1/fork

ret = os.fork()

if ret>0:

父进程

else:

子进程

Os.fork会返回2个值一个0，一个大于0，其中大于零的时父进程，等于0的时子进程，父类进程不会等待子进程结束

### 2/process

from multiprocessing import Process

import time

def test():

for i in range(5):

print("---test---")

time.sleep(1)

p = Process(target=test, args=(100,))#这里一定要加,

p.start() #让这个进程开始执行test函数里的代码

p.join()

默认不会等待子进程结束，可设置参数

**.join([time])** 等待子进程结束，或等待多少秒

**.start()** 启动进程实例

**.run()** 如果没有target方法，则默认调用的方法，这个方法在multiprocessing.process中，可以进行重写

### 3/进程池

from multiprocessing import Pool

import os

import random

import time

def worker(num):

for i in range(5):

print("===pid=%d==num=%d="%(os.getpid(), num))

time.sleep(1)

#3表示 进程池中对多有3个进程一起执行

pool = Pool(3)

for i in range(10):

print("---%d---"%i)

#注意：如果添加的任务数量超过了进程池中进程的个数的话，那么不会导致添加不进入

pool.apply\_async(worker, (i,))

#pool.apply(worker, (i,)) 阻塞

pool.close()#关闭进程池，相当于不能够再次添加新任务了

pool.join()

注意：

1/主进程默认**不会等待**进程池中的任务执行完后才结束，而是当主进程的任务做完之后立马结束

2/添加join使主进程等待子进程

### 4/通信

进程之间数据是不共享的，需要进行通信才可以数据共享

**队列->使用process**

From multiprocessing import Queue

Q = Queue(3)#最多存储3条数据

Q.put(“xx”)#添加数据

Q.get()#吐出数据

Q.full()#判断数据是否放满

不同进程之间需要传递Queue对象，如：

from multiprocessing import Process, Queue

def write(q):

q.put(value)

def read(q):

value = q.get(True)

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

# 父进程创建Queue，并传给各个子进程：

q = Queue()

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

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

**队列->使用pool**

1/如果要使用Pool创建进程，就需要使用**multiprocessing.Manager()**中的Queue()，而不是multiprocessing.Queue()，否则会得到一条如下的错误信息：

RuntimeError: Queue objects should only be shared between processes through inheritance.

## 线程

### 1/Thread

.join()

->主线（进程）等待子线程结束，才回到进程。可设置时间

->

thread模块比较底层，一般使用threading模块

from threading import Thread

t = Thread(target=test)#test函数名称

t.start()

# t.join()

通过类调用

import threading

import time

class MyThread(threading.Thread):

def run(self):

for i in range(3):

time.sleep(1)

msg = "I'm "+self.name+' @ '+str(i) #name属性中保存的是当前线程的名字

print(msg)

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

t = MyThread()#初始化类的时候自动调用

t.start()

### 2/互斥锁

多个线程对同一个资源进行竞争

**from threading import** Thread, **Lock**

import time

g\_num = 0

def test1():

global g\_num

#这个线程和ｔｅｓｔ2线程都在抢着　对这个锁　进行上锁，如果有１方成功的上锁，那么导致另外

#一方会堵塞（一直等待）到这个锁被解开为止

**mutex .acquire() #上锁**

for i in range(1000000):

g\_num += 1

**mutex .release()** #用来对mutex指向的这个锁进行解锁，只要开了锁，那么接下来会让所有因为这个锁被上了锁而堵塞的线程进行抢着上锁

def test2():

global g\_num

**mutex .acquire() #上锁**

for i in range(1000000):

g\_num += 1

**mutex .release() #解锁**

#创建一把互斥锁，这个锁默认是没有上锁的

**mutex = Lock()**

p1 = Thread(target=test1)

p1.start()

p2 = Thread(target=test2)

p2.start()

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

**from threading import** **Lock #主要的方法**

**mutex = Lock()**

**mutex .acquire([timeout])**

**mutex .release()**

注意：加锁的代码越少约好

## 三、线程注意

使用队列，栈，线程池缓存数据，处理与生产数据速度不匹配，解耦作用

Import queue import Queue #线程中是使用的队列，与进程中的对类不同

Import threading from local #线程中类似session.localhost的东西，保存数据，实现数据共享

Local\_p = local() #创建

Local\_p.one = “xxxx”#这是一个对象，可以存放函数

## 四、GIL锁

由于GIL锁的存在，导致python的**线程**是**假的多任务**，在执行多任务时线程效率远不及进程效率高，但进程与进程之间通信比较复杂

解决方法使用C语言进行函数的调用，涉及到调用第三方函数

GCC编译成动态库文件

Gcc xxx.c -shared -o -libxxx.so

from ctypes import \*

from threading import Thread

#加载动态库

lib = cdll.LoadLibrary("./libdeadloop.so")

#创建一个子线程，让其执行ｃ语言编写的函数，此函数是一个死循环

t = Thread(target=lib.DeadLoop)

t.start()

# 二十三/Socket

## 服务端

from socket import \*

udpSocket = socket(AF\_INET, SOCK\_DGRAM)

udpSocket.bind(("", 6789))#绑定IP，端口；这里是一个元组

---------UDP不用下面2句

UdpSocker.listen() #监听

UdpSocker.accept() #创建一个客户端，这里返回的是2个值

----------

recvInfor = udpSocket.recvfrom(1024)#接受数据，1024字节

recvInfor[0].decode("gb2312")#解码

SOCK\_DGRAM:UDP协议

SOCK\_STREAM:TCP协议

AF\_INET:IPV4协议

Recv/send:TCP协议

Recvfrom/sendto : UDP协议

Send(b””)#必须加个B

## 客户端

From socket import \*

udpSocket = socket(AF\_INET,SOCK\_DGRAM)

udpSocket.sendto(sendData.encode("gb2312"), (IP, Port))#发送，设置编码

Udpsocker.recvfrom(1024)

类如在收数据的同时还要发数据，多线程/进程

# 二十四/加密解密方法

pip install cryptography

def get\_3rd\_session(self, openid):

"""加密"""

key = Fernet.generate\_key().decode("utf-8") #得到一个获取key

cipher = Fernet(key) #根据key获取，一个加密

#加密数据

encrypted\_data = cipher.encrypt(openid.encode('utf-8')).decode("utf-8")

return encrypted\_data, key

def get\_openid\_from\_encrypted\_data(self, encrypted\_data, key):

"""解密"""

cipher = Fernet(key)

raw\_data = cipher.decrypt(encrypted\_data)

return raw\_data