# 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

python中一切皆对象

可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

不可变类型：变量赋值 a=5 后再赋值 a=10，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变a的值，相当于新生成了a。

可变类型：变量赋值 la=[1,2,3,4] 后再赋值 la[2]=5 则是将 list la 的第三个元素值更改，本身la没有动，只是其内部的一部分值被修改了。

python 函数的参数传递：

不可变类型：类似 c++ 的值传递，如 整数、字符串、元组。如fun（a），传递的只是a的值，没有影响a对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。

可变类型：类似 c++ 的引用传递，如 列表，字典。如 fun（la），则是将 la 真正的传过去，修改后fun外部的la也会受影响

python 中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。

# 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

Input() # 允许获取输入

# 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]倒序

# 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\_\_**

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

**\_\_dict\_\_ 序列化时调用**

A = AA()

A.\_\_dict\_\_ # 类的序列化方法，只能序列化**实列属性,无法序列化类属性**

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

**\_\_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”]

**\_\_getitem\_\_ 但类使用[]调用时触发**

o = YY()

print(o[‘item’]) **# 调用YY.\_\_getitem\_\_() 方法**

# 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

类方法与静态方法主要区别在于参数

<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

## dict序列化字典

当dict()序列化一个类时，会触发如下方法

1\ 先出发类中的keys方法，获取所有允许序列化的类

2\ 触发\_\_getitem\_\_方法，根据keys()返回的值获取类中的类属性

class YY(object):

name = “aaa” # 不会因为

def keys(self):

return (‘name’,’age’)  **# dict只会序列化name,age字段**

def \_\_getitem\_\_(self, item):

return getattr(self, item) **#当YY[‘name’]触发**

## 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

**##类方法创建**

@classmethod

def instance(cls, \*args, \*\*kwargs):

if not hasattr(dog, “\_instance”):

dog.\_instance = dog()

return dog.\_instance

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/模块导入

**注意：**

1、python包导入只能导入**子目录**中的**模块**（必须定义为包）

2、像django、flask、tornado等一些框架能**不同包**之间**随意**导入，是因为他的**入口文件**server.py（项目启动文件）在**目录顶层。**

## 情况一

**|--server.py**

**|--Bao**

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

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

**/server.py**

from Bao import Module

## 情况二

**|--Bao**

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

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

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

**/Module\_one.py**

import Module\_two

from Module\_two impor \*

from . import Module\_two

## 情况三

**|--Bao1**

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

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

**|--Bao2**

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

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

**/Bao2/Modeule\_one.py**

from Bao2 import Module\_two

from .Bao2 import Module\_two

## 情况四（不可导入）

**|--Bao1**

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

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

**|--Bao2**

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

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

**注意：Module\_one与Module\_two直接不能互相导入**

***解决方案1：添加同级目录***

import sys   
sys.path.append("绝对路径")

***解决方案2：组织* 情况一 *的目录结构***

## 情况五（不可导入）

**|--Bao2**

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

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

**|--Bao3**

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

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

**注意：Module.two不能导入Module\_one，Module\_one可导入Module\_two**

***解决方案1：添加同级目录***

import sys   
sys.path.append("绝对路径")

***解决方案2：组织* 情况一 *的目录结构***

# 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 **# 返回了line函数的指针**，

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

**import functools**

**def timefun(func):** #timefun装饰器名称，

@functools.wraps(func) # 将foo函数名传递过来

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

.........

**return func(\*args, \*\*kwargs)** #对于有返回值的函数

**return wrappedfunc** #这个名称要与自定义名称一样

@timefundef #调用 ，**会立刻调用**

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。**

## 带有参数的装饰器

**def func\_arg(arg = 'hello'):** //装饰器函数的参数

**def func (funNume):** //代指 f1 函数

**def func\_in(\*args,\*\*kargs):** // f1 函数的参数

**return funNume(\*args,\*\*kargs)** // f1函数的返回值

**return func\_in**

**return func**

#1.先执行func\_arg('world')函数，这个函数return的结果是func这个函数的引用

[#2.@func#3.使用@func对f1进行装饰](mailto:#2.@func#3.使用@func对f1进行装饰)

@func\_arg('world')

def f1(a,b):

return 'haha'

## 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