**关于Python**

1. Python的函数参数传递方法？
2. 普通函数，按顺序传递；
3. 关键字传递，参数名=值；
4. 有默认值的参数传递，不传为默认值，传值为所传的值；
5. 函数参数带\*，传入元组；
6. 函数参数带\*\*，传入字典。
7. Python中的元类(metaclass)有哪些？
8. 用类形式

创建元类（metaclass）继承type，重写\_\_new\_\_方法：

class OneMetaClass(type):

\_\_new\_\_(cls, name, bases, attrs)

cls: 将要创建的类，类似与self，但是self指向的是instance，而这里cls指向的是class

name: 将要创建的类的名字，也就是我们通常用类名.\_\_name\_\_获取的。

bases: 要继承的父类

attrs: 类将有的属性和方法，是一个dict。

返回一个type方法：

return type.\_\_new\_\_(cls, name, bases, attrs)

使用创建一个class：

class classname(basicclass,metaclass=OneMetaclass)

classname: 将要创建的类名

basicclass:要继承的父类名

metaclass=:要使用的元类（类生成器）

1. 用函数形式

Def fn(class\_name, basic\_name, attrs):

Return type(class\_name, (basic\_name,), attrs)

调用返回是一个class，需要实例化为一个instance，然后使用。

3、@staticmethod和@classmethod的区别？

类的方法有三种：

实例方法（instance method）类中的普通方法 def 名字(self,参数) self 是实例化的对象

类方法（class method）用@classmethod修饰的方法 def 名字(cls,参数) cls是自身的类

静态方法（static method）用@staticmethod修饰的方法 def 名字(参数)

实例方法只能被实例化的对象调用

类方法和静态方法可以被实例化的对象或直接被类调用

类方法：

可以进行类判断时；（父子类，父类判断子类是哪个）

用实例和类调用，都是将类作为隐藏参数传递过去；

可以避免类实例化的开销。

静态方法：

当函数逻辑上属于一个类但是不依赖于类的属性时，使用；

用实例和类调用，但是都不依赖于实例和类；

可以避免类实例化的开销。

1. 类变量和实例变量区别？

Python的变量：

普通的python变量(非类相关变量)被赋值后即变量存在，可读可写。

Python类里边的变量（类变量和实例对象变量）以某种方式在某地方被赋值，即存在，可被读写

Python类变量（定义在class里def外）

在class里def外，通过变量名能被赋值

其它地方，通过类名字点变量名可被赋值

Python实例变量（定义在class里def里通过self点变量名）

def里，通过self点变量名能被赋值，不一定非在init里，其他已被调用的方法里也行

其它地方，通过实例对象点变量名被赋值

类变量是该类所有实例共享的属性。（类名点变量名）

当类变量有初值，它的子类这个变量的值先访问本身，再访问父类，直到找到这个值。（这个为初始值）。

当类变量被赋值，它的子类还没被赋值（或只有初始值），它和子类都将被赋值。

当类变量被赋值，它的子类已被赋值，它的类变量将修改，子类的类变量不会改变。

实例变量是对于每个实例都独有的数据。（实例对象点变量名）

每个对象都有独立空间，修改其中一个其他不会变

1. Python中单下划线和双下划线？

"单下划线"开始的成员变量叫做保护变量,意思是只有类对象和子类对象自己能访问到这些变量:

可以访问，但是不希望去创建对象去访问，一般类中会定义自己的接口，让你去访问接口，而不是直接访问"单下划线"开始的成员变量和方法。

"双下划线"开始的是私有成员,意思是只有类对象自己能访问,连子类对象也不能访问到这个数据：

这样定义是防止子类对象对父类对象成员的修改。可以 对象名.\_类名\_\_成员名，访问。

"双下划线开头，双下划线结尾"。一种约定，Python内部的名字，用来区别其他用户自定义的命名,以防冲突。是一些 Python 的“魔术”对象，表示这是一个特殊成员，例如：定义类的时候，若是添加\_\_init\_\_方法，那么在创建类的实例的时候，实例会自动调用这个方法，一般用来对实例的属性进行初使化，Python不建议将自己命名的方法写为这种形式。即以双下划线开头的方法和变量不会被继承

1. 字符串格式化:%和.format？

Python中用'{}'.format()代替原来的’%s’% 参数名（功能更加强大）

%：

如果要传递一组值，需要’%s’% (参数名,)

.format：

1. 使用位置参数（列表用\*名字）

>>> data = ['jack',25]

>>> 'my name is {} ,age {}'.format(data[0],data[1])

'my name is jack ,age 25'

>>> 'my name is {1} ,age {0}{1}'.format(data[0],data[1])

'my name is 25 ,age jack25'

>>> 'my name is {1} ,age {0}'.format(\*data)

'my name is 25 ,age jack'

>>> 'my name is {} ,age {}'.format(\*data)

'my name is jack ,age 25'

1. 使用关键字参数（字典用\*\*名字）

>>> hash = {'name':'hoho','age':18}

>>> 'my name is {name},age is {age}'.format(name='hoho',age=19)

'my name is hoho,age is 19'

>>> 'my name is {name},age is {age}'.format(\*\*hash)

'my name is hoho,age is 18'

class Person(object):

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

self.name = name

self.age = age

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

return "This guy is {self.name}, {self.age} years old.".format(self=self)

In[2]: p = Person("Q1mi", 18)

In[3]: str(p)

Out[3]: 'This guy is Q1mi, 18 years old.'

1. 填充与格式化:[填充字符][对齐方式 <^>][宽度]

>>> '{0:\*>10}'.format(10) ##右对齐

'\*\*\*\*\*\*\*\*10'

>>> '{0:\*<10}'.format(10) ##左对齐

'10\*\*\*\*\*\*\*\*'

>>> '{0:\*^10}'.format(10) ##居中对齐

'\*\*\*\*10\*\*\*\*'

1. 精度与进制

>>> '{:.2f}'.format(1/3)

'0.33'

>>> '{:b}'.format(10) #二进制

'1010'

>>> '{:o}'.format(10) #八进制

'12'

>>> '{:x}'.format(10) #16进制

'a'

>>> '{:,}'.format(12369132698) #千分位格式化

'12,369,132,698'

1. 使用索引

>>> li

['hoho', 18]

>>> 'name is {0[0]} age is {0[1]}'.format(li)

'name is hoho age is 18

1. 迭代器和生成器的区别？

迭代对象iterable object：

可以使用for...in...语句进行循环的对象，比如字符串、列表、元组、字典以及迭代器、生成器都是可迭代对象。

迭代器iterator：

任何实现了iter和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。迭代器实现有三种方法：

1. 内置函数 iter() 将可迭代对象（str,list,tuple,dict…）转化为迭代器
2. 创建类，重写\_\_iter\_\_和\_\_next\_\_方法。

# Create iterator Object

class Container:

def \_\_init\_\_(self, start = 0, end = 0):

self.start = start

self.end = end

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

print("[LOG] I made this iterator!")

return self

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

print("[LOG] Calling \_\_next\_\_ method!")

if self.start < self.end:

i = self.start

self.start += 1

return i

else:

raise StopIteration()

1. 构造生成器。生成器是一种特殊的迭代器。

生成器generator：

yield 语句可以让普通函数变成一个生成器，省略了复杂的 \_\_iter\_\_() & \_\_next\_\_() 方式，并且相应的 \_\_next\_\_() 方法返回的是 yield 后面的值。一种更直观的解释是：程序执行到 yield 会返回值并暂停，再次调用 next() 时会从上次暂停的地方继续开始执行。

def container(start, end):

while start < end:

yield start

start += 1

c = container(0, 5)

print(type(c))

print(next(c))

next(c)

for i in c:

print(i)

1. 说一说面向切面编程AOP和装饰器？

面向过程(Procedure Oriented)是一种以过程为中心的编程思想。

面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）计算机程序是由单个能够起到子程序作用的单元或对象组合而成。

面向对象与面向过程明显的不同就是封装、继承、类。OOP 达到了软件工程的三个主要目标：重用性、灵活性和扩展性。

面向切面编程（Aspect Oriented Programming(AOP)），在运行时，编译时，类和方法加载时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。我们管切入到指定类指定方法的代码片段称为切面，而切入到哪些类、哪些方法则叫切入点。这样的做法，对原有代码毫无入侵性。

装饰器 Decorators

装饰器是一个很著名的设计模式，经常被用于有切面需求的场景，较为经典的有插入日志、性能测试、事务处理等。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量函数中与函数功能本身无关的雷同代码并继续重用。

概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

1. 无参装饰器

def outer(func): # 将index的地址传递给func

def inner():

print('开始')

func() # fun = index 即func保存了外部index函数的地址

print("结束")

return inner # 返回inner的地址

def index():

print("welcome to index page")

run = outer(index)

run()

1. 有参装饰器

def outer(func): # 将index的地址传递给func

def inner(\*args, \*\*kwargs):

print('开始')

res = func(\*args, \*\*kwargs)

print("结束")

return res

return inner # 返回inner的地址

def index(name):

print("welcome to index page", name)

return 1234567890

run = outer(index)

print(run('hahaha'))

@outer # 语法糖 index=outer(index).

def index(name):

print("welcome to index page", name)

return 1234567890

当一个对象有多个装饰器，也就是有多个语法糖的时候，它的执行顺序是从下到上执行。

执行顺序按照普通装饰类很好理解。

1. 类装饰器

类装饰器具有灵活度大、高内聚、封装性等优点。使用类装饰器主要依靠类的\_\_call\_\_方法，当使用 @ 形式将装饰器附加到函数上时，就会调用此方法。

class Foo(object):

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

self.\_func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print('class decorator runing')

res = self.\_func(\*args, \*\*kwargs)

print('class decorator ending')

return res

@Foo

def bar(name):

print('bar', name)

return 123456

print(bar('aaa'))

9、Python自省

自省就是面向对象的语言所写的程序在运行时,所能知道对象的类型.简单一句就是运行时能够获得对象的类型.比如type(),dir(),getattr(),hasattr(),isinstance().

10、Python is是对比地址,==是对比值

11、\_\_new\_\_和\_\_init\_\_的区别

\_\_new\_\_元类创建类时使用。返回一个类。

\_\_init\_\_是类初始化的时候使用，没用返回值。

12、Python函数式编程

lambda函数

函数名 = lambda 参数列：函数体

13、单例模式

单例模式（Singleton Pattern）确保某个类只能有一个实例存在。eg.读取文件，一个实例就够用，创建多余的实例会严重浪费内存资源。

1. 使用模块

将类和创建这个类的实例对象放在一个py文件中，然后需要这个实例的时候用import导入创建的这个实例对象。\*\*这个对象也可以自己实例化，但是就不是单例了。

test.py中：

class Singleton(object):

def foo(self):

pass

singleton = Singleton()

另一个py文件中：

from test import singleton

1. 使用装饰器（进行普通的实例化就是单例。）

def Singleton(cls):

instance = {}

def \_singleton(\*args, \*\*kargs):

if cls not in instance:

instance[cls] = cls(\*args, \*\*kargs)

return instance[cls]

return \_singleton

# 语法糖标注，就是闭包函数第一层已经执行，在第一层定义的变量和传入的参数都会赋给返回对象。

@Singleton

class A(object):

def \_\_init\_\_(self, x=0):

self.x = x

a1 = A(2)

a2 = A(3)

# a2获取的是a1的实例对象，不再新建对象。

1. 使用类方法（可以直接自己实例化，但是不属于单例）

class Singleton(object):

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

pass

@classmethod

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

if not hasattr(Singleton, "\_instance"):

Singleton.\_instance = Singleton(\*args, \*\*kwargs)

return Singleton.\_instance

obj1 = Singleton.instance()

obj2 = Singleton.instance()

1. 基于\_\_new\_\_方法实现\*\*\*不错（将类方法改为\_\_new\_\_，进行普通的实例化就是单例）

当我们实例化一个对象时，是先执行了类的\_\_new\_\_方法（我们没写时，默认调用object.\_\_new\_\_），实例化对象；然后再执行类的\_\_init\_\_方法，对这个对象进行初始化。

class Singleton(object):

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

pass

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not hasattr(Singleton, "\_instance"):

# 在\_\_new\_\_中获取当前类的实例要用：

# super.\_\_new\_\_(cls,\*arg,\*\*kwargs)（使用父类创建）

Singleton.\_instance = super().\_\_new\_\_(cls)

return Singleton.\_instance

obj1 = Singleton()

obj2 = Singleton()

1. 基于metaclass元类方式实现（进行普通的实例化就是单例）

# 可知在类的实例函数调用时会运行（instance\_name()）当前类的\_\_call\_\_方法；

# Foo类是通过元类创建，所以在这个类实例化时（class\_name(\*args, \*\*kwargs)）会调用当前元类的\_\_call\_\_方法。

class SingletonType(type):

def \_\_call\_\_(cls, \*args, \*\*kwargs):

if not hasattr(cls, "\_instance"):

cls.\_instance = super().\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instance

class Foo(metaclass=SingletonType):

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

self.name = name

# 元类的\_\_call\_\_方法返回的一般是调用元类的类的实例。

# 使用super().\_\_call\_\_(\*args, \*\*kwargs)（元类的父类）创建

obj1 = Foo('name')

obj2 = Foo('name')

\*\*\*\*这样的单例在多线程的时候会出现问题（实例有时会出现多个），所以要在创建实例的地方

创建锁。让在创建实例的时候进不去。

（5种创建单例的方法，1（import方法）基本不用；2（装饰器）单例多线程不好实现；3（使用类方法），4（使用\_\_new\_\_方法），5（使用元类中的\_\_call\_\_方法）多线程好锁。其中优先使用\_\_new\_\_方法，其次元类\_\_call\_\_方法，再其次类方法（类方法单例包装不严密））

import threading

class Singleton(object):

instance\_lock = threading.Lock()

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

pass

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not hasattr(Singleton, "\_instance"):

with Singleton.instance\_lock:

if not hasattr(Singleton, "\_instance"):

Singleton.\_instance = object.\_\_new\_\_(cls)

return Singleton.\_instance

def task(arg):

obj = Singleton()

print(obj)

for i in range(10):

t = threading.Thread(target=task,args=[i,])

t.start()

14、GIL线程全局锁

I/O密集型，多线程效率高

线程全局锁(Global Interpreter Lock),即Python为了保证线程安全而采取的独立线程运行的限制,说白了就是一个核只能在同一时间运行一个线程（单核多线程）。可以使用协程。

计算密集型，多进程效率高

绕过GIL，使用多进程。用multiprocessing模块创建进程池。利用协程和进程配合。

或者使用C编写计算密集型任务。

15、协程

协程，英文Coroutines，是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。

协程的暂停完全由程序控制；进程和线程的阻塞状态是由操作系统内核来进行切换，面临着内核态和用户态的切换问题而耗费许多切换时间

因此，协程的开销远远小于进程和线程的开销。

python可以通过 yield/send 的方式实现协程（生成器generator）。在python 3.5以后，async/await 成为了更好的替代方案。