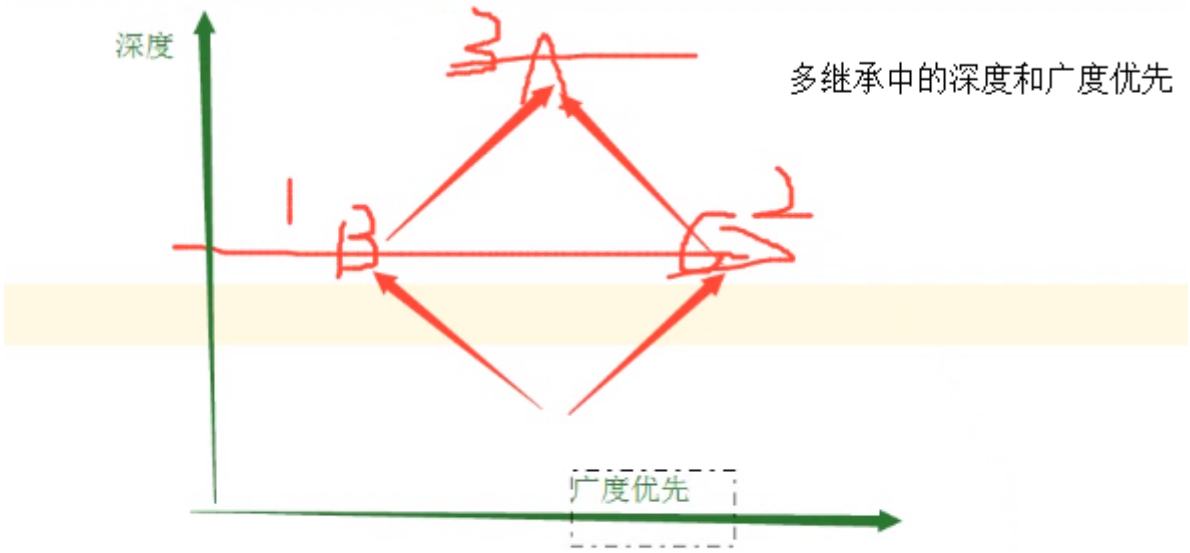
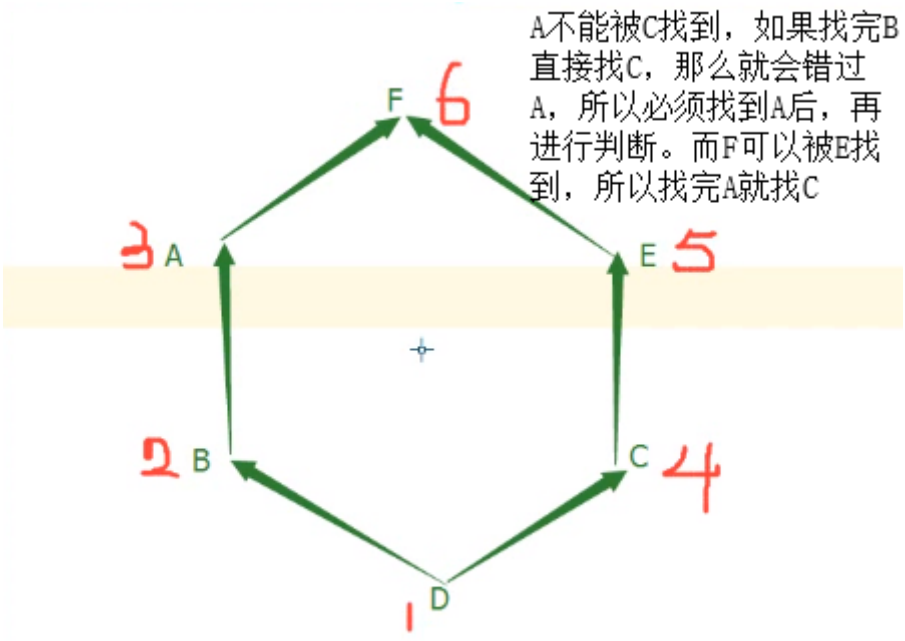
python面试题搜集（九）：史上最全python面试题详解（三）

38、面向对象深度优先和广度优先是什么?






```
d = D()
# d.func()
print(D.mro())
```

类.mro() 函数可以显示继承类的顺序

```
多继承
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\pytho
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>,
```

```python
class D(B,C):
    pass
    # def func(self): print('D')

d = D()
d.func()
#2.7
# 新式类 继承object类的才是新式类 广度优先
# 经典类 如果你直接创建一个类在2.7中就是经典类 深度优先
# print(D.mro())
# D.mro()

# 多继承中，我们子类的对象调用一个方法，默认是就近原则，找的顺序是什么？
# 经典类中 深度优先
# 新式类中 广度优先
# python2.7 新式类和经典类共存，新式类要继承object
# python3   只有新式类，默认继承object
# 经典类和新式类还有一个区别 super mro方法只在新式类中存在
```

python3全是默认继承object类，都算是新式类，都按照广度优先算法执行

39、面向对象中super的作用?

```python
class Dog(Animal):
    def __init__(self,name,aggr,hp,kind):
        super().__init__(name,aggr,hp)   # 只在新式类中有，python3中
        self.kind = kind         # 派生属性
    def eat(self):print('dog eating')
jin = Dog('金老板',200,500,'teddy')
print(jin.name)
jin.eat()
super(Dog,jin).eat()
```

当super（）函数在类的外部使用，则（类名，对象名）中需要传参数

40、是否使用过functools中的函数？其作用是什么？

```
1 Python自带的 functools 模块提供了一些常用的高阶函数，也就是用于处理其它函数的特殊函数。换言之，就是能使用该模块对可调用对象进行处理。
2
3 functools模块函数概览
4 functools.cmp_to_key(func)
5 functools.total_ordering(cls)
6 functools.reduce(function, iterable[, initializer])
7 functools.partial(func[, args][, *keywords])
8 functools.update_wrapper(wrapper, wrapped[, assigned][, updated])
9 functools.wraps(wrapped[, assigned][, updated])
```

41、列举面向对象中带双下划线的魔术方法?

1. init()

2. del()

```
1 在调用del方法的时候，实际使用的是del()
2
3 class Person(object):
4     def __del__(self):
5         print('我给干掉啦')
6
7 bill = Person()
8 del bill #我给干掉啦
```

3. new()

```
 1 new()只有继承自objectd的类才有new()这方法是在init()之前调用的，用于生成实例对象。多用于设计模式中的单例模式。单例模式是为了确保类有且只有一个对象。多用于日志记录和数据库操作，打印机后台处理程序。这样子可以避免对统一资源产生相互冲突的请求
 2 new（）负责创建一个类的对象，init（）方法负责对创建后的类对象进行默认设置
 3 class Singleton(object):
 4     def __new__(cls):
 5         if not hasattr(cls, 'instance'):
 6             cls.instance = super(Singleton, cls).__new__(cls)
 7         return cls.instance
 8
 9 s = Singleton()
10 print('Object created', s)
11 s1 = Singleton()
12 print('Object created', s1)
13
14 # output
15 # Object created <__main__.Singleton object at 0x0000018EFF662DA0>
16 # Object created <__main__.Singleton object at 0x0000018EFF662DA0>
17
```

```
18 cls是当前类，new()返回的是一个实例，和init()中的self是同一个东西
```

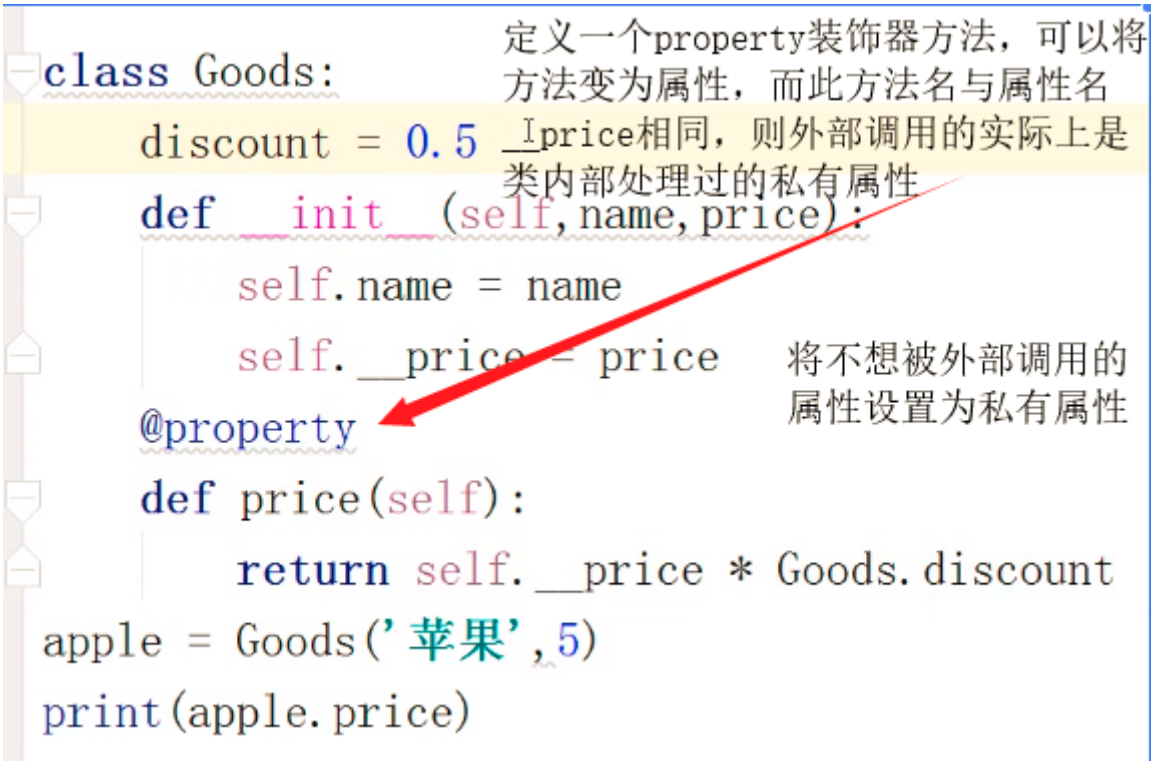
42、如何判断是函数还是方法?

一般情况下，单独写一个def func():表示一个函数，如果写在类里面是一个方法。但是不完全准确。

```
1 class Foo(object):
2     def fetch(self):
3         pass
4
5 print(Foo.fetch)   # 打印结果<function Foo.fetch at 0x000001FF37B7CF28>表示函数
6 # 如果没经实例化，直接调用Foo.fetch()括号里要self参数，并且self要提前定义
7 obj = Foo()
8 print(obj.fetch)  # 打印结果<bound method Foo.fetch of <__main__.Foo object at
0x000001FF37A0D208>>表示方法
```

43、面向对象中的property属性、类方法、静态方法?

property属性:



类方法:

```python
# 类的操作行为
class Goods:
    __discount = 0.8
    def __init__(self,name,price):
        self.name = name
        self.__price = price
    @property
    def price(self):
        return self.__price * Goods.__discount
    @classmethod  # 把一个方法 变成一个类中的方法，这个方法就直接可以被类调用，
    def change_discount(cls,new_discount):  # 修改折扣
        cls.__discount = new_discount
apple = Goods('苹果',5)
print(apple.price)
Goods.change_discount(0.5)    # Goods.change_discount(Goods)
print(apple.price)
# 当这个方法的操作只涉及静态属性的时候 就应该使用classmethod来装饰这个方法
```

类方法就是对类的直接操作

静态方法:

```python
    @staticmethod
    def get_usr_pwd():    # 静态方法
        usr = input('用户名 : ')
        pwd = input('密码 : ')
        Login(usr,pwd)

类名
Login.get_usr_pwd()
# 在完全面向对象的程序中，
# 如果一个函数 既和对象没有关系 也和类没有关系 那么就用staticmethod将这
# 类方法和静态方法 都是类调用的
# 对象可以调用类方法和静态方法么？    可以    一般情况下 推荐用类名调用
# 类方法 有一个默认参数 cls 代表这个类  cls
# 静态方法 没有默认的参数 就象函数一样
```

静态方法可以传任意参数，用类名调用

静态方法时java中的概念

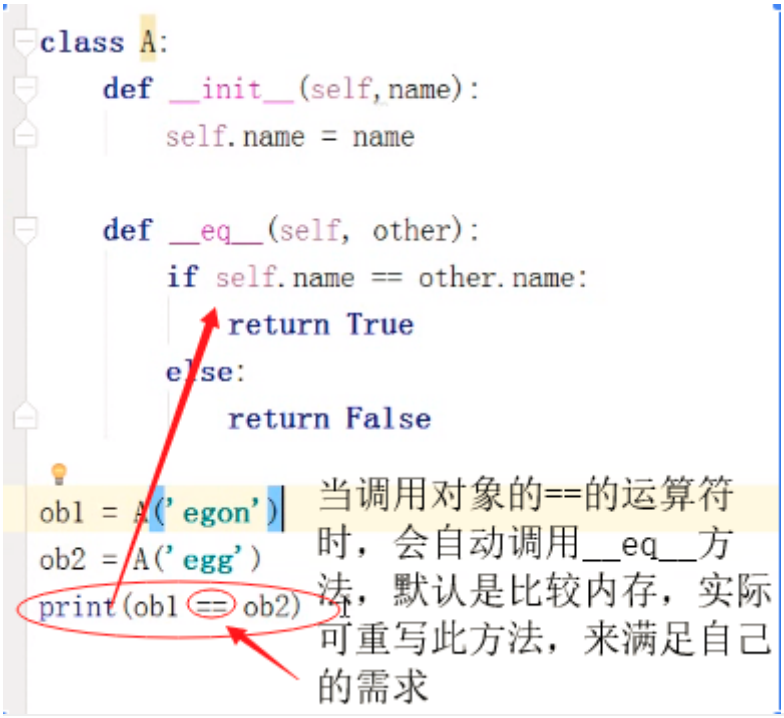
由于java中只有类的概念，没有独立函数的概念，当需要定义一个单独意义的函数的时候，又必须把函数放在类中，就需要用到静态方法

44、列举面向对象中的特殊成员以及应用场景

```
1. __doc__
    表示类的描述信息
class Foo:
    """ 描述类信息，这是用于看片的神奇 """
    def func(self):
        pass
print Foo.__doc__

==============
 描述类信息，这是用于看片的神奇
```

```
2. __module__ 和  __class__
   __module__ 表示当前操作的对象在哪个模块
    __class__      表示当前操作的对象的类是什么
3. __init__
构造方法，通过类创建对象时，自动触发执行。
4. __del__
析构方法，当对象在内存中被释放时，自动触发执行。

注：此方法一般无须定义，因为Python是一门高级语言，程序员在使用时无需关心内存的分配和释放，因为此工作都是交给Python解释器来执行，所以，析构函数的调用是由解释器在进行垃圾回收时自动触发执行的。
5. __call__
    对象后面加括号，触发执行。

注：构造方法的执行是由创建对象触发的，即：对象 = 类名() ；而对于 __call__ 方法的执行是由对象后加括号触发的，即：对象() 或者 类()()
6. __dict__
    类或对象中的所有成员
7. __str__
如果一个类中定义了__str__方法，那么在打印 对象 时，默认输出该方法的返回值。
```

8、eq



45、什么是反射？以及应用场景？

反射的意义

```
class A:
    price = 20
    def func(self):
        print('in func')
# 反射类的属性
# A.price
print(getattr(A,'price'))
```

反射使字符串具有对象或类的属性、方法的特性

类似于（A.属性）的模式都可以套用反射

```
class A:
    price = 20
    @classmethod
    def func(cls):
        print('in func')
# 反射类的属性
# A.price
print(getattr(A,'price'))

# 反射类的方法：classmethod staticmethod
# A.func()
if hasattr(A,'func'):
    getattr(A,'func')()
```

定义一个类方法（参数是一个类cls）

hasattr函数进行判断反射是否成立，成立则执行getattr

```python
def qqxing():
    print('qqxing')
year = 2018
import sys
# print(sys.modules['__main__'].year)
# 反射自己模块中的变量
# print(getattr(sys.modules['__main__'],'year'))
if "__name__" =="__main__":
    main()
# 反射自己模块中的函数
# getattr(sys.modules['__main__'],'qqxing')()
变量名 = input('>>>')
print(getattr(sys.modules['__main__'],变量名))
```

表达式可以拿出自己模块的名字

改为__name__
在其他模块导入此模块时就可以表示其他模块的名字

```python
# setattr  设置修改变量
class A:
    pass
a = A()
setattr(a,'name','nezha')
setattr(A,'name','alex')
print(A.name)
print(a.name)

# delattr 删除一个变量
delattr(a,'name')
print(a.name)
delattr(A,'name')
print(a.name)
```

实质是为对象和类增加属性值

删除对象和类的属性

46、用尽量多的方法实现单例模式。

一、模块单例

Python 的模块就是天然的单例模式，因为模块在第一次导入时，会生成 `.pyc` 文件，当第二次导入时，就会直接加载 `.pyc` 文件，而不会再次执行模块代码。

```
#foo1.py
class Singleton(object):
    def foo(self):
        pass
singleton = Singleton()

#foo.py
from foo1 import singleton
```

二、静态变量方法

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，所有我们可以基于这个，实现单例模式。

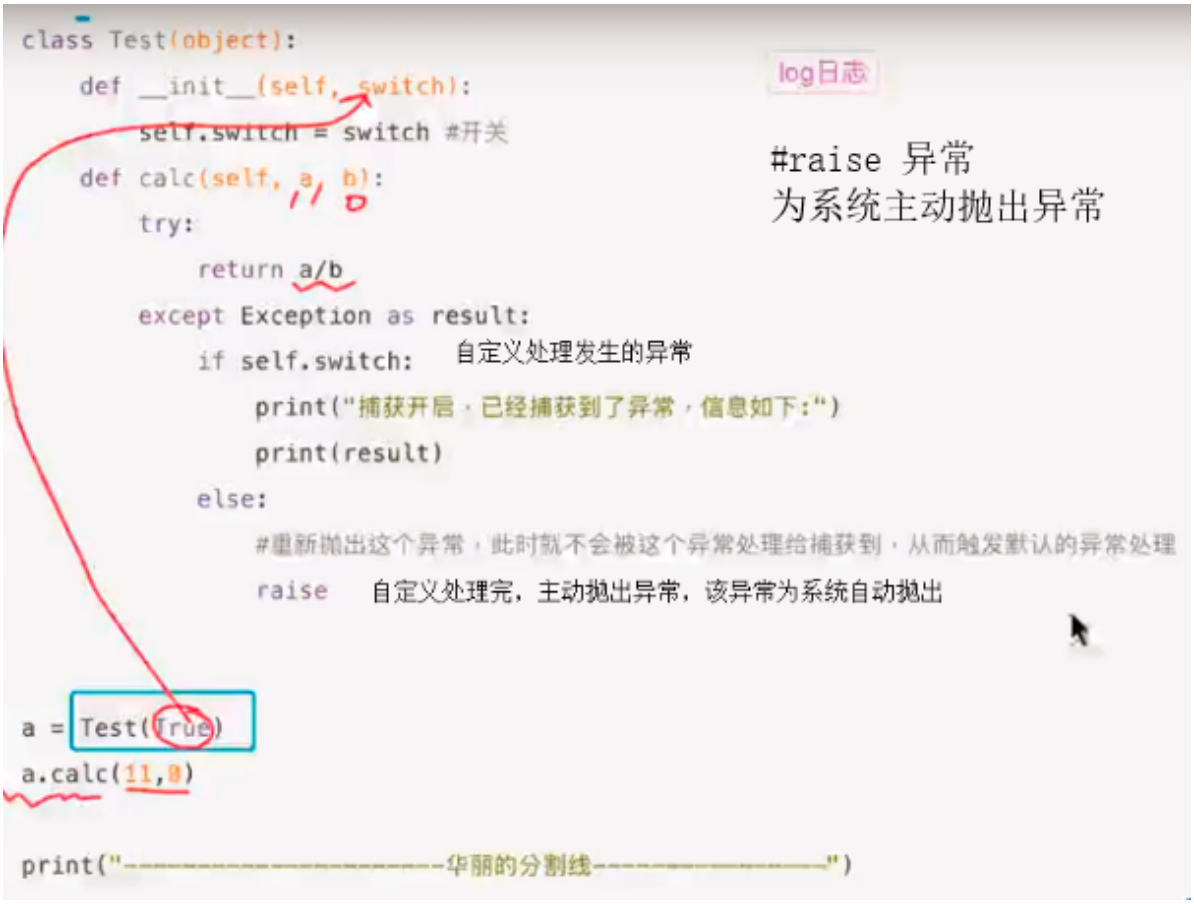
```
class Singleton(object):
    def __new__(cls,a):
        if not hasattr(cls, '_instance'):
            cls._instance = object.__new__(cls)
        return cls._instance
    def __init__(self,a):
        self.a = a
    def aa(self):
        print(self.a)

a = Singleton("a")
```

47、装饰器的写法以及应用场景。

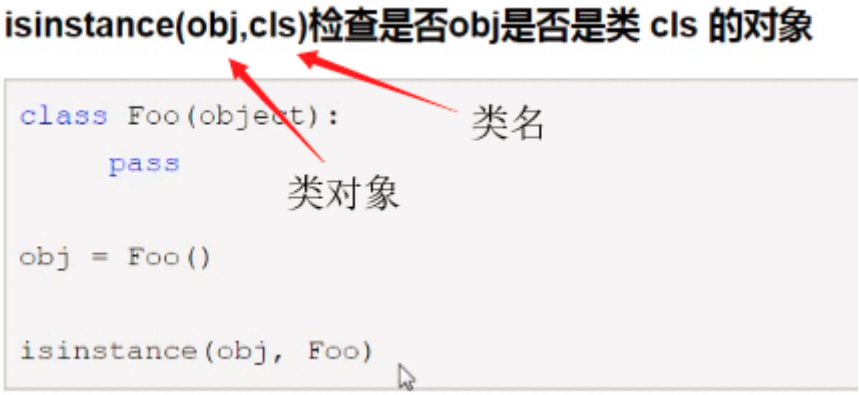
```python
from functools import wraps
def wrapper(func):  #func = holiday
    @wraps(func)
    def inner(*args,**kwargs):
        print('在被装饰的函数执行之前做的事')
        ret = func(*args,**kwargs)
        print('在被装饰的函数执行之后做的事')
        return ret
    return inner

@wrapper    #holiday = wrapper(holiday)
def holiday(day):
    '''这是一个放假通知'''
    print('全体放假%s天'%day)
    return '好开心'

print(holiday.__name__)
print(holiday.__doc__)
ret = holiday(3)    #inner
print(ret)
```

确保在加入装饰器功能外，不影响装饰对象的调用

48、异常处理写法以及如何主动跑出异常（应用场景）



49、isinstance作用以及应用场景?



50、json序列化时，可以处理的数据类型有哪些？如何定制支持datetime类型?

```
官方文档中的一个Demo：
>>> import json

>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         return json.JSONEncoder.default(self, obj)
...
>>> dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
```

```
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[', '2.0', ', ', '1.0', ']']

```

```
然后简单扩展了一个JSONEncoder出来用来格式化时间
class CJsonEncoder(json.JSONEncoder):

    def default(self, obj):
        if isinstance(obj, datetime):
            return obj.strftime('%Y-%m-%d %H:%M:%S')
        elif isinstance(obj, date):
            return obj.strftime('%Y-%m-%d')
        else:
            return json.JSONEncoder.default(self, obj)

```

```
使用时候只要在json.dumps增加一个cls参数即可：

json.dumps(datalist, cls=CJsonEncoder)
```

51、json序列化时，默认遇到中文会转换成unicode，如果想要保留中文怎么办?

json序列化时遇到中文会默认转换成unicode ，如何让他保留中文形式

```
import json
a=json.dumps({"ddf":"你好"},ensure_ascii=False)
print(a) #{"ddf": "你好"}
```

52、使用代码实现查看列举目录下的所有文件。

```
import os

if __name__ == '__main__':
    work_dir = 'C:\Program Files\MySQL\Connector ODBC 8.0'
    for parent, dirnames, filenames in os.walk(work_dir, followlinks=True):
        for filename in filenames:
            file_path = os.path.join(parent, filename)
            print('文件名：%s' % filename)
            print('文件完整路径：%s\n' % file_path)
```

53、简述 yield和yield from关键字。

1、可迭代对象与迭代器的区别

可迭代对象：指的是具备可迭代的能力，即enumerable. 在Python中指的是可以通过for-in 语句去逐个访问元素的一些对象，比如**元组**tuple,**列表**list，**字符串**string，**文件对象**file 等。

迭代器：指的是通过另一种方式去一个一个访问可迭代对象中的元素，即enumerator。在python中指的是给内置函数iter()传递一个可迭代对象作为参数，返回的那个对象就是迭代器，然后通过迭代器的next()方法逐个去访问。

```
1 from collections import Iterable
2
3 li=[1,4,2,3]
4 iterator1 = iter(li)
5 print(next(iterator1))
6 print(next(iterator1))
7 print(next(iterator1))
8 print(isinstance(iterator1,Iterable)) # 判断是否是迭代器，导入collection模块
```

```
>>>1
4
2
True
```

2、生成器

生成器的**本质**就是一个逐个返回元素的函数，即“本质——函数”

最大的好处在于它是“延迟加载”，即对于处理长序列问题，更加的节省存储空间。即生成器每次在内存中只存储一个值

3、什么又是yield from呢?

简单地说，yield from generator 。实际上就是**返回另外一个生成器**。如下所示：

```
 1 def generator1():
 2     item = range(10)
 3     for i in item:
 4         yield i
 5
 6 def generator2():
 7     yield 'a'
 8     yield 'b'
 9     yield 'c'
10     yield from generator1() #yield from iterable本质上等于 for item in
iterable: yield item的缩写版
11     yield from [11,22,33,44]
12     yield from (12,23,34)
```

```
13     yield from range(3)
14
15 for i in generator2() :
16     print(i)
```

从上面的代码可以看出，**yield from** 后面可以跟的式子有“ **生成器 元组 列表等**可迭代对象以及 **range（）**函数产生的序列”

上面代码运行的结果为：

a b c 0 1 2 3 4 5 6 7 8 9 11 22 33 44 12 23 34 0 1 2

请关注，未完待续！