Python 基础部分

一、面向对象

1.简述面向对象的三大特征

封装:

- 对属性的封装：对象.属性 = 值
- 对方法的封装：类中定义的方法

继承：(drf视图)

- 子类自动拥有父类中除了私有内容外的所有内容
- 多继承 -> MRO: C3算法来完成

多态:

- 鸭子模型

2.什么是鸭子模型

```
class Foo(object):
    def xxxxx(self):
        pass

class Base(object):
    def xxxxx(self):
        pass

def func(arg):  #arg可以是任意类型，但必须实现xxxx方法
    arg.xxxxx()

obj1 = Foo()
obj2 = Base()

func(obj1)
func(obj2)
```

3.super的作用

MRO顺序的上一个

4.MRO是什么，什么是C3算法?

5.列举面向对象中带双下划线的特殊方法

```
# __getattr__
  CBV
  django配置文件
  wtforms中的Form()示例化中 将"_fields中的数据封装到From类中"
# __mro__
```

```
  wtform中 FormMeta中继承类的优先级
# __dict__
是用来存储对象属性的一个字典，其键为属性名，值为属性的值
# __new__
实例化但是没有给当前对象
wtforms，字段实例化时返回：不是StringField，而是UnboundField
est frawork many=Turn  中的序列化
# __call__
flask 请求的入口app.run()
字段生成标签时：字段.__str__ => 字段.__call__ => 插件.__call__
# __iter__
循环对象是，自定义__iter__
wtforms中BaseForm中循环所有字段时定义了__iter__
# -metaclass
作用：用于指定当前类使用哪个类来创建
场景：在类创建之前定制操作
示例：wtforms中，对字段进行排序。
```

```
class Foo(object):

    def __setitem__(self, key, value):
        pass

    def __getitem__(self, item):
        pass

    def __delitem__(self, key):
        pass


obj = Foo()
obj['k1'] = 'alex'  #自动调用setitem方法
obj['k1']           #自动调用getitem方法

del obj['k1']       #自动调用delitem方法

#Django中request.session内部使用的是__setitem__
```

6.双下划线和单下划线的区别?

- _xxx "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（即类实例）和子类对象自己能访问到这些变量，需通过类提供的接口进行访问；不能用'from module import *'导入
- __xxx 类中的私有变量/方法名 （Python的函数也是对象，所以成员方法称为成员变量也行得通。）," 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。
- `__xxx__` 系统定义名字，前后均有一个“双下划线” 代表python里特殊方法专用的标识，如 **init**() 代表类的构造函数

7.实例变量和类变量的区别

- 实例变量是对于每个实例都独有的数据，而类变量是该类所有实例共享的属性和方法。
- 类变量又叫全局变量，是属于类的特性，**实例先找实例化变量，然后再去找类变量**.

- **类变量最好是用类名来访问**,当然, 我们通过对象名也可以访问. 但只能看, 不能改变它,**想要改变它, 需要用类名来改变它.**

8.静态方法和类方法的区别?

静态方法

- 静态方法是定义在类名称空间的一个函数，定义形式是在def行前加修饰符@staticmethod
- 静态方法的参数可以是任意参数，不需要self参数
- 可以通过类名或者值为实例对象的变量，以属性的方式调用静态方法

类方法

- 类方法定义形式是在def行前加修饰符@classmethod，这种方法必须有一个表示其调用类的参数，一般用**cls**作为参数名，也可以有多个其他的参数
- 类方法也是类对象的属性，可以以属性访问的形式调用
- 调用它的类将自动约束到方法的cls参数，可以通过这个参数访问该类的其他属性

9.isinstance和type的作用?

10.有用过with statement语句吗，他的好处是?

11.实现一个单例类

```
import threading
class singleton(object):
    _instance = None
    _lock = threading.RLock()

    def __new__(cls, *args, **kwargs):
        if cls._instance:
            return cls._instance


        with cls._lock:
            if not cls._instance:
                cls._instance = super().__new__(cls, *args, **kwargs)
            return cls._instance
```

12.上下文管理

```
class Foo(object):

    def __enter__(self):
        print('进入')
        return 123

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('退出')

obj = Foo()
with obj as f:
    print(f)
#__exit__ 方法中有三个参数，用来接收处理异常，如果代码在运行时发生异常，异常会被保存到这里。
```

```
#exc_type : 异常类型
#exc_val : 异常值
#exc_tb : 异常回溯追踪
```

13.请描述with的用法,如果自己的类需要支持with语句,应该如何书写?

```
class Person:
    def __enter__(self):
        print("我是enter")

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("我是exit ")

    def __call__(self, *args, **kwargs):
        print("我是call")

p = Person()

p()

with p: # 自动调用__enter__
    print("哈哈哈")

    # 从with出去的时候. 自动调用__exit__
```

14.python中如何判断一个对象是否可调用? 哪些对象是可调用对象? 如何定义一个类,使其对象本身就是可调用对象?

如何判断是否可调用

```
#使用内置的callable函数
    callable(func)
#判断对象类型是否是FunctionType
    type(func) is FunctionType
        # 或者
    isinstance(func, FunctionType)
#判断对象是否实现__call__方法
    hasattr(func, '__call__')
```

可调用对象

- python中分为函数(function)和方法(method)，函数是python中的一个可调用对象(用户定义的课调用对象,及lambda表达式创建的函数,都是函数,其类型都是Functiontype),方法是一种特殊的类函数
- 类方法和类进行绑定，实例方法与实例进行绑定，所以两者的类型都是method。
- 而静态方法，本身即不和类绑定，也不和实例绑定，不符合上述定义，所以其类型应该是function
- 如果一个类实现了`__call__`方法，那么其实例也会成为一个可调用对象，其类型为创建这个实例的类，而不是函数或方法。

15.请实现一个栈

```
class Stack(object):
```

```
    def __init__(self):
        self.data = []

    def push(self,value):
        self.data.append(value)

    def pop(self):
        return self.data.pop()

obj = Stack()
obj.push('alex')
obj.push('eric')
val = obj.pop()
print(val)
```

16.python面向对象中能给属性赋值吗?

17.如何禁止从外部修改实例属性:

- 我们可以通过`__slots__`来严格防止别人访问定义的实例属性
- Python在存储实例的过程实际上是存储到`__dict__`字典里
- `__slots_`的作用就是一旦在类里定义了该属性，那么Python将再不会创建**dict**了，而是把实例属性存储到`__slots__`里面(实际上是定义了一个描述器来存储)

```
>>> class C:
...     __slots__ = 'x', 'y'
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
...
>>> c = C()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'x' and 'y'
>>> c= C(3,4)
>>> c.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute '__dict__'
>>> c.__slots__
('x', 'y')
```

定义类，绑定__slots__

该类的实例不再创建__dict__

https://blog.csdn.net/weixin_4268186

18.如何创建私有属性？可以从外部访问私有属性吗？私有属性可以继承吗?

提到Python的**私有属性**，即是在属性前加双下划线，最大的作用就是防止别人访问该私有属性，如果访问就会报错：AttributeError： module object has no attribute '__x'，举例如下：

```
class A:
    def __init__(self,x,y):
        self.__x = x
        self.y = y

a = A(3,4)
print(a.y)              #4

print(a.__dict__)       #{'_A__x': 3, 'y': 4}
```

- Python是如何存储实例属性(定义在**init**下的属性)。当我们定义实例属性时，Python会把其**存入dict中**，实际上就是个字典，以上图为例，当我们访问y属性时，实际上访问的是a.`__dict__`['y']
- `__dict__`里还有`_A__x`，也就是说通过a.`__dict__`[''`_A__x`'']就能**访问到我们不想让人访问的**私有属性`__x`，而`__dict__`默认是省略的，所以直接a.`_A__x`就可以访问私有属性了，所以才说python的私有属性并不那么私有，阻挡不了非开发人员的硬性访问。

19.私有属性可以被继承吗?

- 私有的属性、方法，不会被子类继承，也不能被访问

20.可以给属性方法设置值吗?

二、函数

1.函数

- 当python遇到 def 语句时候 ,会在内存中生成一个函数对象,并且这个函数是靠将函数名来引用,但是这个函数体内部的语句只有在函数的调用的时候才会被执行，而函数调用结束了，就是函数返回时,函数执行完后内部变量将会被回收
- 函数是一个对象,所以可以以对象的形式作为某个函数的结果返回

2.什么是闭包

简单来说,外层函数outer()里面包含了一个内部函数 inner(),并且外部函数返回内部函数的对象

```
def outer():
    name = "alex"
    # 内部函数
    def inner():
        print(name)
    return inner
fn = outer()          # 访问外部函数，获取到内部函数的函数地址
fn()                  # 访问内部函数
```

- 内部函数inner() 存在对外部函数outer()的引用 ,此时这个内部函数 inner()就叫做闭包
- 在闭包中,由于内部函数存在对外部函数变量的引用,所以即使外部函数执行完毕,该变量依然存在但是一定是内层函数调用的外层函数的变量，如果内层函数不会引用，那么这个变量也是不存在的

3. 装饰器

1. 装饰器的作用

- 装饰器本质上是一个 Python 函数或类，它可以让其他函数或类在不需要做任何代码修改的前提下增加额外功能，
- 装饰器的返回值也是一个函数/类对象。
- 它有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码到装饰器中并继续重用

2.不带参数的装饰器

```
def wrapper(func):
    def inner(*args,**kwargs):
        print('--函数前添加功能--')
        ret = func(*args,**kwargs)
```

```
        print('--函数后添加功能--')
        return ret
    return inner

#定义一个函数,并添加装饰器
@wrapper   #等价于  func1 = timer(func1)
def func1(m):
    print('--func1--')
    return m
#调用函数
f = func1(20)
print(f)
```

3.带参数的装饰器

```
def outer(flag):
    def timer(func):
        def inner(*args,**kwargs):
            if flag:
                print('--添加功能--')
                ret = func(*args,**kwargs)
            else:
                ret = func(*args,**kwargs)
            return ret
        return inner
    return timer
#定义函数并添加装饰器
@outer(True)  # func1 = timer()
def func1(m):
    print('--func1--')
    return m
#调用函数
print(func1(20))
```

4.多个装饰器装饰一个函数

```
def wrapper1(func):
    print('--wrapper1装饰--')
    def inner1(*args,**kwargs):
        print('--wrapper1 前--')
        ret = func(*args,**kwargs)
        print('--wrapper1 后--')
        return ret
    return inner1

def wrapper2(func):
    print('--wrapper2装饰--')
    def inner2(*args,**kwargs):
        print('--wrapper2 前--')
        ret = func(*args,**kwargs)
        print('--wrapper2 后--')
        return ret
```

```
    return inner2

#定义函数并添加装饰器
@wrapper2   #func1 = wrapper2(func1)  inner2  func = inner1
@wrapper1   #func1 = wrapper1(func1)  inner1  func = func1
def func1(m):
    print ('--func1--')
    return m
#调用函数
print(func1(20))

#执行结果:
# --开始装饰wrapper1--
# --开始装饰wrapper2--
# --wrapper2 前--
# --wrapper1 前--
# --func1--
# --wrapper1 后--
# --wrapper2 后--
# 20
```

5.装饰器修复技术

```
import time
from functools import wraps
def timer(func):
    @wraps(func)
    def inner():
        print(time.time())
        ret = func()
        return ret
    return inner
```

4.迭代器

存在的意义: 为了让所有的数据类型拥有相同的遍历方式

`__iter__`获取到迭代器 `__next__`获取到下一个元素. 获取到最后的时候会报错. StopIteration

list. dict, range, open(), str, tuple, set

都可以被迭代. 都可以使用for循环

特点:

1. 省内存 -> 生成器
2. 惰性机制
3. 只能向前. 不能反复

5.生成器

1. 生成器函数: yield

```
def func():
   yield

func() 创建生成器对象 不会执行函数
g.__next__()   运行到下一个yield. 把yield后的数据返回
g.send(123)  给上一个yield位置传值
```

2.生成器表达式

[结果 for if] {key:value for if} {key for if}

(结果 for if) 直接获取到生成器对象

6.列表推导式

```
v = [i for i in range(10)]

v = [lambda:i for i in range(10)]
print(v[0]())
print(v[9]())


def func():
    return [lambda:i for i in range(5)]
v = [m() for m in func()] # [4,4,4,4,4]
```

7.(i for i in range(10)) 和 [i for i in range(10)] 的区别?

8.lambda表达式格式以及应用场景?

```
# 格式:
    匿名函数: res = lambda x: i*x   □ print(res(2))
# 应用场景:
    Filter(),map(),reduce(),sorted()函数中经常用到，它们都需要函数形参数;
    一般定义调用一次。
     (reduce()对参数序列中元素进行累积)
```

9.列举常见的内置函数

```
# map:遍历序列，为每一个序列进行操作，返回一个结果列表
l = [1, 2, 3, 4, 5, 6, 7]
def pow2(x):
   return x * x
res = map(pow2, l)
print(list(res)) #[1, 4, 9, 16, 25, 36, 49]
-------------------------------------------------------------
# reduce: 对于序列里面的所有内容进行累计操作
from functools import reduce
def add(x, y):
   return x+y
print(reduce(add, [1,2,3,4])) #10
-------------------------------------------------------------
```

```
# filter: 对序列里面的元素进行筛选，最终获取符合条件的序列。
l = [1, 2, 3, 4, 5]
def is_odd(x):  # 求奇数
   return x % 2 == 1
print(list(filter(is_odd, l))) #[1, 3, 5]
---------------------------------------------------------------
#zip用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表
a = [1,2,3]
b=[4,5,6]
c=[4,5,6,7,8]
ziped1 = zip(a,b)
print('ziped1>>>',list(ziped1)) #[(1, 4), (2, 5), (3, 6)]
ziped2 = zip(a,c)
print('ziped2>>>',list(ziped2)) #[(1, 4), (2, 5), (3, 6)],以短的为基准
```

8.将空列表作为参数

```
def func(a=[]):
  a.append(3)
  return a

# ------------ 调用 ---------------
def func(a=[]):
  print id(a)
  a.append(3)
  return a

> print func()
4365426272
[3]
> print func()
4365426272
[3, 3]
> print func()
4365426272
[3, 3, 3]
#原来在python中，*默认参数不是每次执行时都创建的！*
```

9. *args，kwargs**

```
def func(*args):
  print id(args)
> a = [1,2]
> print id(a)
4364874816
> func(*a)
4364698832
> func(*a)
4364701496
# 实际上args也会产生一个新的对象。但是值是填入的传入参数。那么每一个item也会复制吗
```

10.内置函数

1.lamda匿名函数

为了解决一些简单的需求而设计的一句话函数, 语法: 函数名 = lambda 参数: 返回值

- 函数的参数可以有多个, 多个参数之间用逗号隔开
- 匿名函数不管多复杂. 只能写一行, 且逻辑结束后直接返回数据
- 返回值和正常的函数一样, 可以是任意数据类型

```
zed= lambda a,b : a+b
ret=zed(2,3)
print(ret)
```

2. sorted()排序函数.

语法: sorted(Iterable, key=None, reverse=False)

- **Iterable**: 可迭代对象
- **key:** 排序规则(排序函数), 在sorted内部会将可迭代对象中的每一个元素传递给这个函数的参数. 根据函数运算的结果进行排序
- **reverse:** 是否是倒序. True: 倒序, False: 正(不写,默认正序)

```
lst=["天龙八部","倚天屠龙记","鹿鼎记","雪山飞狐","功夫","射雕英雄传"]
def func(name):
    return len(name)
l1 = sorted(lst,key=func)
print(l1)
lst=["天龙八部","倚天屠龙记","鹿鼎记","雪山飞狐","功夫","射雕英雄传"]
l2 = sorted(lst,key=lambda name:len(name)%3)              #sorted与lambda配合使用
print(l2)
```

```
lst = [{"id": 1, "name": 'alex', "age": 18},
    {"id": 2, "name": 'wusir', "age": 16},
    {"id": 3, "name": 'taibai', "age": 17}]
# 按照年龄对学生信息进行排序
def func(dic):
    return dic['age']
l2 = sorted(lst, key=func)          # 流程: 把可迭代对象的每一项传递给函数. 函数返回一个数字. 根据这个数字完成排序
print(l2)
l3 = sorted(lst, key=lambda dic: dic['age'])
print(l3)
l4 = sorted(lst, key=lambda dic: len(dic['name']))                      #按照名字长度排序
l5 = sorted(lst, key=lambda dic: ascii(dic['name'][0]))  # ord()        #按照名字ascii排序
print(l4)
print(l5)
```

3. filter()筛选函数

语法: filter(function. Iterable)

- **function:** 用来筛选的函数, 在filter中会自动的把iterable中的元素传递给function. 然后根据function返回的True或者False来判断是否保留此项数据

- **Iterable:** 可迭代对象

```
lst = [23, 28, 15, 27, 24, 22]
def func(age):
    return age > 18 and age % 2 == 0
f = filter(func, lst)
print("__iter__"in dir(f))                  #判断f是否为可迭代对象  True
print(list(f))                              #  [28, 24, 22]
lst = [23, 28, 15, 27, 24, 22]
print(list(filter(lambda age:age >18 and age % 2 ==0,lst)))
lst = [23, 28, 15, 27, 24, 22]
f = filter(lambda age: age > 18 and age % 2 == 0, lst)
print(list(f))                      #  [28, 24, 22]
print(sorted(f))                    #[]                  迭代器的惰性机制
lst = [23, 28, 15, 27, 24, 22]
f = filter(lambda age: age % 2 == 0, filter(lambda age: age > 18, lst))
print(list(f))
```

```
lst = [{"id":1, "name":'alex', "age":18},
 {"id":2, "name":'wusir', "age":16},
 {"id":3, "name":'taibai', "age":17}]

# 筛选出年龄大于等于17岁的人,并按照年龄排序
f1=filter(lambda dic: dic['age'] >= 17, lst)
f2=sorted(f1, key=lambda dic: dic['age'])
print(list(f2))            #[{'id': 3, 'name': 'taibai', 'age': 17}, {'id': 1, 'name':
'alex', 'age': 18}]
# print(list(sorted(filter(lambda dic: dic['age'] >= 17, lst), key=lambda dic:
dic['age'])))
```

4. map() 映射函数

语法: map(function, iterable) 可以对可迭代对象中的每一个元素进行映射,分别取执行 function

```
lst = [1,5,9,3]
# l1 = [i**2 for i in lst]
m = map(lambda x: x**2, lst)
print(list(m))
```

计算两个列表中相同位置的数据的和

```
lst1 = [1, 2, 3, 4, 5]
lst2 = [2, 4, 6, 8, 10]
print(list(map(lambda x, y: x + y , lst1, lst2)))          #[3, 6, 9, 12, 15]
```

三、基础数据类型

1.解释型语言和编译型语言的区别?

```
编译型:代码文件  -> 文件(机器码)  -> 计算机运行. 例如: C/go/java/c#
解释性:由解释器边解释边执行.                    例如: python/php/js
```

2.你除了python以外还会哪些编程语言?

```
大学学过:C语言/java,全都还给老师了.
了解:go

go和python的区别?
- 并发(网络)
    - go: 编程语言已经帮你写好了.
    - python: 自己写并发.
- 解释性编译型
- 简易程度
    - py:简单.
    - go:难.
```

3.你为什么选择python?

```
- 语法简洁
- 从诞生之初,类库丰富,不用重复造轮子.
- 大势所趋:
    - 底层算法/api
    - 应用开发

购买现成的解决方案.
```

4.列举你常用数据类型及其中的方法

```
str:split/strip/upper/replace/join
list:append/insert/pop
dict:get/keys/values/items
tuple:...
set:add/交集intersection/并集union/差集 [符号] *****
```

5.深浅拷贝

```
# 针对可变类型: list,dict,set
# 浅拷贝:拷贝第一层
# 深拷贝:所有层的可变类型都会被拷贝.
```

6.is 和 == 的区别?

```
is是内存
==值
```

7.看代码写结果

```
v = 1 or 2
v = 0 or 6
v = 1 and 2
v = 0 and 2
```

8.给你一个大文件,如何查看所有内容?

```
# for 文件对象

# tell 和 seek
```

四、其他

1.内存的简单理解

- 栈区（stack）： 由编译器自动分配释放 ， 存放函数的参数值，局部变量的值等。其操作方式先进后出类似于数据结构中的栈；
- 堆区（heap） ： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由系统回收，**但在程序运行期间可能造成内存的泄露（比如循环new出来的对象却总是没有被delete掉）**；
- 3、全局变量区（也称静态存储区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域，程序结束后有系统释放；
- 4、常量区： 常量字符串就是放在这里的， 程序结束后由系统释放；
- 5、代码段：存放函数体的二进制代码；

2.python 的垃圾回收机制

- 引用计数：记录每个对象的引用计数，如果是0就被回收，但是循环引用处理不了
- 标记清除：从变量触发找对象，找到了就标记，把没有被标记的对象清除掉，可以处理循环引用，但还是必须将所有的引用过一遍，太耗时。
- 分代计数
  - 第一代： 从头到尾，10遍第一代，之后，如果还没有被清除，晋升为第二代
  - 第二代：从头到尾10遍，晋升为第三代
  - 第三代：从头到尾。。。。。。

3.深浅拷贝

2.2 浅拷贝 :增加了一个指针指向一个存在的内存地址

- 两种方式

```
lst1 = ["太白","日天","哪吒","银角大王","金角大王"]
lst2 = lst1
lst1.append("女神")
print(lst1,id(lst1))     # ['太白', '日天', '哪吒', '银角大王', '金角大王', '女神']
2420170040264
print(lst2,id(lst2))      #['太白', '日天', '哪吒', '银角大王', '金角大王', '女神']
2420170040264
#指向同一个内存地址,所以二者相同
```

```
lst1 = ["太白","日天","哪吒","银角大王","金角大王"]
lst2 = lst1.copy()      # 会创建新对象，创建对象的速度会很快.  lst2 = lst1[:] 创建了新列表
lst1.append("女神")
print(lst1,id(lst1))    #['太白', '日天', '哪吒', '银角大王', '金角大王', '女神']
2606709613704
print(lst2,id(lst2))    #['太白', '日天', '哪吒', '银角大王', '金角大王'] 2606709613832
```

- 多层情形

```
lst1 = ["太白","日天",["盖浇饭", "锅包肉", "吱吱冒油的猪蹄子"],"哪吒","银角大王","金角大王"]
lst2 = lst1.copy() # 会创建新对象，创建对象的速度会很快.
lst1[2].append("油泼扯面")
print(lst1, id(lst1[2]))     #['太白', '日天', ['盖浇饭', '锅包肉', '吱吱冒油的猪蹄子', '油泼
扯面'], '哪吒', '银角大王', '金角大王'] 1440371189000
print(lst2, id(lst2[2]))     #['太白', '日天', ['盖浇饭', '锅包肉', '吱吱冒油的猪蹄子', '油泼
扯面'], '哪吒', '银角大王', '金角大王'] 1440371189000
```

浅拷贝. 只会拷贝第一层. 第二层的内容不会拷贝. 所以被称为浅拷贝

2.3 深拷贝 增加一个指针并且开辟了新的内存，这个增加的指针指向这个新的内存

- 单层

```
import copy
li1 = [1, 2, 3]
li2 = copy.deepcopy(li1)
li1.append(4)
print(li1, id(li1))     # [1, 2, 3, 4]      3112618579592
print(li2, id(li2))     # [1, 2, 3]         3112618579784
```

- 多层

```
import copy

li1 = [1, 2, 3,[4,5],6]
li2 = copy.deepcopy(li1)
li1[3].append(7)
print(li1, id(li1))      # [1, 2, 3, [4, 5, 7], 6]    2575562397512
print(li2, id(li2))      # [1, 2, 3, [4, 5], 6]       2575562398792
```

注:深拷贝. 把元素内部的元素完全进行拷贝复制.,不会产生一个改变另一个跟着改变的问题