小强必备更优对象和避坑练习

234 defaultdict创建带初始值的字典用

defauldict 能自动创建一个被初始化的字典，也就是每个键都已经被访问过一次。

首先，导入 defaultdict

复制
```
In [44]: from collections import defaultdict
```

创建一个字典值类型为 `int` 的默认字典：

复制
```
In [45]: d = defaultdict(int)
```

创建一个字典值类型为 `list` 的默认字典：

复制
```
In [46]: d = defaultdict(list)

In [47]: d
Out[47]: defaultdict(list, {})
```

统计下面字符串

`from collections import defaultdict`

每个字符出现的位置索引

复制
```
d = defaultdict(list)
s = 'from collections import defaultdict'
for index,i in enumerate(s):
    d[i].append(index)
print(d)

defaultdict(<class 'list'>, {'f': [0, 26], 'r': [1, 21], 'o': [2, 6, 13,
20], 'm': [3, 18], ' ': [4, 16, 23], 'c': [5, 10, 33], 'l': [7, 8, 29], '
e': [9, 25], 't': [11, 22, 30, 34], 'i': [12, 17, 32], 'n': [14], 's': [15
], 'p': [19], 'd': [24, 31], 'a': [27], 'u': [28]})
```

当尝试访问一个不在字典中的键时，将会抛出一个异常。但是，使用 defaultdict 帮助我们初始化。

如果不使用 defaultdict，就需要写 if -else 逻辑。

如果键不在字典中，手动初始化一个列表：[]，并放入第一个元素：字符的索引 index

就像下面这样：

复制
```
d = {}
s = 'from collections import defaultdict'
for index,i in enumerate(s):
    if i in d:
        d[i].append(index)
    else:
        d[i] = [index]
print(d)
# 结果如下：
{'f': [0, 26], 'r': [1, 21], 'o': [2, 6, 13, 20], 'm': [3, 18], ' ': [4,
16, 23], 'c': [5, 10, 33], 'l': [7, 8, 29], 'e': [9, 25], 't': [11, 22, 30
, 34], 'i': [12, 17, 32], 'n': [14], 's': [15], 'p': [19], 'd': [24, 31],
'a': [27], 'u': [28]}
```

虽然也能得到同样结果，但是，很显然，使用 defaultdict，代码更加简洁。

235 defaultdict 应用之排序词

排序词(permutation)：两个字符串含有相同字符，但字符顺序不同。

复制
```
from collections import defaultdict

def is_permutation(str1, str2):
    if str1 is None or str2 is None:
        return False
    if len(str1) != len(str2):
        return False
    unq_s1 = defaultdict(int)
    unq_s2 = defaultdict(int)
    for c1 in str1:
        unq_s1[c1] += 1
    for c2 in str2:
        unq_s2[c2] += 1

    return unq_s1 == unq_s2
```

`defaultdict`，字典值默认类型初始化为 `int`，计数默次数都为0.

统计出的两个defaultdict：unq*s1*，*unqs2*，如果相等，就表明 str1、 str2 互为排序词。

下面，测试：

复制
```
r = is_permutation('nice', 'cine')
print(r)  # True

r = is_permutation('', '')
print(r)  # True

r = is_permutation('', None)
print(r)  # False

r = is_permutation('work', 'woo')
print(r)  # False
```

236 Counter 和 defaultdict 结合求单词频次

使用 yield 解耦数据读取 `python_read` 和数据处理 `process`

`python_read`：逐行读入

`process`： 正则替换掉空字符，并使用空格，分隔字符串，保存到 `defaultdict` 对象中。

复制
```
from collections import Counter, defaultdict
import re

def python_read(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        for line in f:
            yield line

d = defaultdict(int)

def process(line):
    for word in re.sub('\W+', " ", line).split():
        d[word] += 1
```

调用两个函数

使用 `Counter` 类统计出频次的排序

复制
```
for line in python_read('test.txt'):
    process(line)

frequency = Counter(d).most_common()
print(frequency)
```

237 坑点之列表与 * 操作

Python 中， `*` 操作符与 list 结合使用，实现元素复制。

复制 10 个 `|` 字符：

复制
```
In [32]: ['|'] * 10
Out[32]: ['|', '|', '|', '|', '|', '|', '|', '|', '|', '|']
```

复制 5 个 空列表：

复制
```
In [33]: [[]] * 5
Out[33]: [[], [], [], [], []]
```

创建一个空列表 a 后，

复制
```
a = []
```

发现 a 中的元素又是一个 list，a 的长度为 5 ，使用 `*` 复制 ：

复制
```
a = [[]] * 5
```

根据业务规则，如下填充元素：

复制
```
In [2]: a[0].extend([1,3,5])

In [3]: a[1].extend([2,4,6])
```

按照本来的想法，a 应该被填充为：

复制
```
[[1,3,5],[2,4,6],[],[]]
```

但是，实际上运行代码发现，a 为：

复制
```
In [4]: a
Out[4]:
[[1, 3, 5, 2, 4, 6],
 [1, 3, 5, 2, 4, 6],
 [1, 3, 5, 2, 4, 6],
 [1, 3, 5, 2, 4, 6],
 [1, 3, 5, 2, 4, 6]]
```

原来 `*` 操作复制出的 `a[0]` ， `a[1]` ， `...` ， `a[5]` ，在内存中标识符是相等的，实现的仅仅是浅复制。

复制
```
In [6]: a
Out[6]: [[], [], [], [], []]

In [7]: id(a[0])
Out[7]: 1958135807304

In [8]: id(a[1])
Out[8]: 1958135807304

In [9]: id(a[2])
Out[9]: 1958135807304
```

在这种场景下，希望实现 `id[0]` ，`id[1]` 不相等，修改 a[1] 不会影响 a[0]

不使用 `*` ，使用列表生成式，复制出 5 个不同 id 的内嵌列表 ，这样就能避免赋值互不干扰的问题。

复制
```
In [10]: b = [[] for _ in range(5)]

In [11]: b[0].extend([1,3,5])

In [12]: b[1].extend([2,4,6])

In [13]: b
Out[13]: [[1, 3, 5], [2, 4, 6], [], [], []]
```

238 坑点之删除列表元素

列表内元素可重复出现，讨论如何删除列表中的某个元素。

如下方法，遍历每个元素，如果等于删除元素，使用 `remove` 删除元素。

```
def del_item(lst,e):
    for i in lst:
        if i == e:
            lst.remove(i)
    return lst
```

调用 `del_item` 函数，删除成功：

```
In [19]: del_item([1,3,5,3,2],3)
Out[19]: [1, 5, 2]
```

这代表删除元素的方法是正确的吗？

考虑，删除序列 [1,3,3,3,5] 中的元素 3，结果中仍有元素 3！

```
In [20]: del_item([1,3,3,3,5],3)
Out[20]: [1, 3, 5]
```

这是为什么？遍历 `lst`，`remove` 一次，移掉位置 `i` 后的所有元素索引都要减一。

所以，一旦删除的元素，重复出现在列表中，就总会漏掉一个该删除的元素。

正确做法，找到被删除元素后，删除，同时下次遍历索引不加一；

若未找到，遍历索引加一，如下所示：

```
def del_item2(lst,e):
    i = 0
    while i < len(lst):
        if lst[i] == e:
            lst.remove(lst[i])
        else:
            i += 1
    return lst
```

调用函数，删除操作都正确：

```
In [25]: del_item2([1,3,5,3,2],3)
Out[25]: [1, 5, 2]

In [26]: del_item2([1,3,3,3,5],3)
Out[26]: [1, 5]
```

239 坑点之函数默认参数为空

Python 函数的参数可设为默认值。

如果一个默认参数类型为 list，默认值为设置为 [].

这种默认赋值，会有问题吗？

几年前，我参加面试时，就被面到这个问题。

```
def delta_val(val, volume=[]):
    if volume is None:
        volume = []
    size = len(volume)
    for i in range(size):
        volume[i] = i + val
    return volume
```

调用 `delta_val` 函数，`val` 值为 `10`，`volume` 默认值，函数返回 `rtn` 为空列表。

```
In [3]: rtn = delta_val(10)

In [4]: rtn
Out[4]: []
```

然后，我们向空列表 `rtn` 中，分别添加值 1, 2，打印 `rtn`，结果符合预期。

```
In [5]: rtn.append(1)

In [6]: rtn.append(2)

In [7]: rtn
Out[7]: [1, 2]
```

同样方法，再次调用 `delta_val` 函数，第二个参数还是取默认值。

预期返回值 `rtn` 还是空列表，但是结果却出人意料！

```
In [7]: rtn = delta_val(10)

In [8]: rtn
Out[8]: [10, 11]
```

为什么返回值为 [10, 11] 呢？ 按照出现的结果，我们猜测 `[1, 2]` + `10` 后，不正是 `[11, 12]`。

原来调用函数 `delta_val` 时，默认参数 `volume` 取值为默认值时，并且 `volume` 作为函数的返回值。再在函数外面做一些操作，再次按照默认值调用，并返回。整个过程，默认参数 `volume` 的 `id` 始终未变。

```
def delta_val(val, volume=[]):
    print(id(volume)) # 打印 volume 的 id
    size = len(volume)
    for i in range(size):
        volume[i] = i + val
    return volume
```

还原上面的调用过程：

```
rtn = delta_val(10)
rtn.append(1)
rtn.append(2)
rtn = delta_val(10)
```

可以看到 2 次调用 `delta_val`，`volume` 的内存标识符从未改变。

```
1812560502088
1812560502088
```

为了避免这个隐藏的坑，函数的默认参数值切记不能设置为 []，而是为 None

这样即便按照默认值调用多次，也会规避此风险。

```
def delta_val(val, volume= None):
    if volume is None:
        volume = []
    size = len(volume)
    for i in range(size):
        volume[i] = i + val
    return volume
```

重复前面的调用过程：

```
In [19]: rtn = delta_val(10)

In [20]: rtn.append(1)

In [21]: rtn.append(2)

In [22]: rtn
Out[22]: [1, 2]

In [23]: rtn = delta_val(10)

In [24]: rtn # 输出符合预期
Out[24]: []
```

243 坑点之{} 和()

Python 中，下面 `point` 是一个元组对象：

```
point = (1.0,3.0)
```

但是，初始创建的元组对象，若只有一个元素，只用一对括号是不够的，下面 `single` 对象不会被解释为元组，而是 `float` 型。

```
single = (1.0)

In [14]: type(single)
Out[14]: float
```

要想被解释为元组，在后面必须要加一个逗号：

```
single = (1.0,)
```

之所以单独说这个问题，是因为在函数调用时，传入参数类型要求为元组。但是在传参时，若不注意拉下逗号，就会改变值的类型。

```
def fix_points(pts):
    for i in range(len(pts)):
        t = pts[i]
        if isinstance(t,tuple):
            t = t if len(t) == 2 else (t[0],0.0)
            pts[i] = t
        else:
            raise TypeError('pts 的元素类型要求为元组')
    return pts
```

如下调用 `fix_points` 函数，第二个元素 (2.0) 实际被解析为浮点型。

```
fix_points([(1.0,3.0),(2.0),(5.0,4.0)])
```

这样传参才是正确的：

```
In [16]: fix_points([(1.0,3.0),(2.0,),(5.0,4.0)])
Out[16]: [(1.0, 3.0), (2.0, 0.0), (5.0, 4.0)]
```

与之类似的，还有创建集合与字典，它们都用一对 `{}`，但是默认返回字典，而不是集合。

```
In [18]: d = {}

In [19]: type(d)
Out[19]: dict
```

要想创建空集合，可使用内置函数 `set()`。

```
In [21]: s = set()

In [22]: type(s)
Out[22]: set
```

244 坑点之解包

Python 中，支持多值赋值给多变量的操作。最常见的用法，一行代码交换两个变量：

```
In [34]: a, b = 1, 2

In [35]: a, b = b, a
```

但是，面对稍微复杂点的类似操作，如果不搞懂多值赋值的执行顺序，就会掉入陷阱。

如下例子，如果心算出的结果等于 `a = 3, b = 5` ，那么就说明未弄明白执行顺序。

```
In [38]: a, b = 1, 2

In [39]: a, b = b+1, a+b
```

记住一点：多值赋值是先计算出等号右侧的所有变量值后，再赋值给等号左侧变量。所以，答案应该是：`a = 3, b = 3`

这种多值赋值，是一种解包 (unpack) 操作。

既然是解包，那么就得先有打包。

的确，等号右侧的多个变量，会被打包 (pack) 为一个可迭代对象。

赋值操作，就相当于解包。

这种解包操作，有时非常有用。比如，`foo` 函数返回一个list，如下：

```
def foo():
    result = [1,'xiaoming','address','telephone',['',''],'...']]
    return result
```

但是，我们只需要列表中的前两项。

更为简洁、紧凑的做法：等号左侧定义两个我们想要的变量，其他不想要的项放到 `others` 变量中，并在前加一个 `*`，如下所示：

```
sid, name, *others = foo()

In [64]: sid
Out[64]: 1

In [65]: name
Out[65]: 'xiaoming'
```

`*others` 会被单独解析为一个 list:

```
In [66]: others
Out[66]: ['address', 'telephone', ['', ''], '...']]
```

245 坑点之访问控制

Python 是一门动态语言，支持属性的动态添加和删除。而 Python 面向对象编程 (OOP) 中，提供很多双划线开头和结尾的函数，它们是系统内置方法，被称为魔法方法。如 `__getattr__` 和 `__setattr__` 是关于控制属性访问的方法。

重写 `__getattr__` 方法，会定义不存在属性时的行为。如下，访问类不存在属性时，程序默认会抛出 `AttributeError` 异常。

```
class Student():
    def __init__(self,idt,name):
        self.id = idt
        self.name = name
```

如果想改变以上这种默认行为，就可以使用 `__getattr__`. 如下，创建一个 `Student` 实例，调用一个不存在 的 `address` 属性时，给它自动赋值 `None`，需要注意只有某个属性不存在时，`__getattr__` 才会被调用。

```
class Student():
    def __init__(self, idt, name):
        self.id = idt
        self.name = name

    def __getattr__(self, prop_name):
        print('property %s not existed, would be set to None automaticall
y' %
                  (prop_name,))
        self.prop_name = None

xiaoming = Student(1, 'xiaoming')
print(xiaoming.address) # 读取
xiaoming.address = 'beijing'
print(xiaoming.address)
```

打印结果如下，Student 拥有属性 `address` 后，不再调用 `__getattr__` .

```
property address not existed, would be set to None automatically
None
beijing
```

还有一个关于属性赋值时行为定义的魔法方法：`__setattr__`，而它不管属性是否存在，属性赋值前都会调用此函数。

```
class Student():
    def __init__(self,idt,name):
        self.id = idt
        self.name = name

    def __getattr__(self,prop_name):
        print(' %s not existed' %(prop_name,))
    def __setattr__(self,prop_name,val):
        print('%s would be set ro %s'%(prop_name,str(val)))
```

只要涉及属性赋值，赋值前都会调用 `__setattr__` 方法：

```
In [2]: xiaoming = Student(1,'xiaoming')
id would be set ro 1
name would be set ro xiaoming

In [3]: xiaoming.prop2 = 1.
prop2 would be set ro 1.0
```

但是，使用它很容易掉进一个坑，`__setattr__` 里再次涉及属性赋值，这样会无限递归下去。

```
    def __setattr__(self,prop_name,val):
        print('%s would be set ro %s'%(prop_name,str(val)))
        self.prop2 = 1.0  # 导致无限递归！
```

为保险起见，不要在 `__setattr__` 方法中再做属性赋值。

246 坑点之中括号访问

经常看到，某个对象具有 `[index]` ，返回某个元素值。那么，它们是怎么实现这种中括号索引的呢？只要重写魔法方法 `__getitem__` ，就能实现 `[index]` 功能。

如下，类 `Table` 是一个最精简的具备中括号索引的类。构造函数 `__init__` 传入一个字典，`__getitem__` 返回字典键为 `column_name` 的字典值。

```
class Table(object):
    def __init__(self,df:dict):
        self.df = df
    def __getitem__(self,column_name):
        return self.df[column_name]

t = Table({'ids':list(range(5)),'name':'li zhang liu guo song'.split()})
```

使用 `Table` 类，`['column_name']` 返回对应的列：

```
print(t['name'])
print(t['ids'])
```

打印结果：

```
['li', 'zhang', 'liu', 'guo', 'song']
[0, 1, 2, 3, 4]
```

248 元类

元类，会被 Pythoner 经常提起，元类确实也有一些使用场合。但是，它又是很高深的、偏底层的抽象类型。 Python 界的领袖 `Tim Peters` 说过：

“元类就是深度的魔法，99%的用户应该根本不必为此操心。”

今天，我们只讲一些元类的基本知识，带你理解元类是什么，怎么使用元类做一个初步介绍。

`xiaoming`, `xiaohong`, `xiaozhang` 都是学生，这类群体叫做 `Student`.

Python 定义类的常见方法，使用关键字 `class`

```
In [36]: class Student(object):
    ...:     pass
```

`xiaoming`, `xiaohong`, `xiaozhang` 是类的实例，则：

```
xiaoming = Student()
xiaohong = Student()
xiaozhang = Student()
```

创建后，xiaoming 的 `__class__` 属性，返回的便是 `Student` 类

```
In [38]: xiaoming.__class__
Out[38]: __main__.Student
```

问题在于，`Student` 类有 `__class__` 属性吗？如果有，返回的又是什么？

```
In [39]: xiaoming.__class__.__class__
Out[39]: type
```

返回 `type` 那么，我们不妨猜测：`Student` 类的类型就是 `type` 换句话说，`Student` 类就是一个对象，它的类型就是 `type` 。因此，类也是对象。

相信，读者朋友们今天可能会对 Python 中一切皆对象，会有一个更深刻的认识。

Python 中，将描述 `Student` 类的类被称为：元类。

既然 `Student` 类可创建实例，那么 `type` 类能创建实例吗？ 如果能，它创建的实例就叫：类了。说对了，`type` 类一定能创建实例，如下所示，`type` 创建的 `Student` 类。

```
In [40]: Student = type('Student',(),{})

In [41]: Student
Out[41]: __main__.Student
```

它与使用 `class` 关键字创建的 `Student` 类一模一样。

249 对象序列化

对象序列化，是指将内存中的对象转化为可存储或传输的过程。很多场景，直接一个类对象，传输不方便。但是，当对象序列化后，就会更加方便，因为约定俗成的，接口间的调用或者发起的 web 请求，一般使用 `json` 串传输。

实际使用中，一般对类对象序列化。先创建一个 `Student` 类型，并创建两个实例。

```
class Student():
    def __init__(self,**args):
        self.ids = args['ids']
        self.name = args['name']
        self.address = args['address']
xiaoming = Student(ids = 1,name = 'xiaoming',address = '北京')
xiaohong = Student(ids = 2,name = 'xiaohong',address = '南京')
```

导入 `json` 模块，调用 `dump` 方法，就会将列表对象 `[xiaoming,xiaohong]` ，序列化到文件 `json.txt` 中。

```
import json

with open('json.txt', 'w') as f:
    json.dump([xiaoming,xiaohong], f, default=lambda obj: obj.__dict__, e
nsure_ascii=False, indent=2, sort_keys=True)
```

生成的文件内容，如下：

```
[
  {
    "address": "北京",
    "ids": 1,
    "name": "xiaoming"
  },
  {
    "address": "南京",
    "ids": 2,
    "name": "xiaohong"
  }
]
```

250 仅 print? 还有日志

在线下，调试代码，我们往往习惯使用 print 函数。通过 print，一些异常信息、变量值信息就会显示在控制台中，然后帮助我们锁定 bug, 找出问题。

但是，当项目上线后，程序一般运行在 linux 服务器上。如果程序出现异常行为，要想通过 print 函数找出问题，可能还得安装调试代码的 IDE，在服务器上做这些事情，可能不太方便。

一般的解决方案，在代码中想 print 的信息，也要写入到日志文件中，在磁盘上保存起来。此时，遇到问题后，找到并分析对应的日志文件就行，这种解决问题的方法更可取，效率也会更高。

日志写入不是我们想象的这般简单。如果一直向同一个文件里写，文件就会变得很大很大；也不方便分析。更糟糕的是，文件越来越大，当大小等于磁盘容量时，后面的日志信息就无法再写入。当然，还有更多问题会出现。

所以，别小看写日志，我们得需要设计一套行之有效的管理体系，对日志实施有效的管理。

像大名鼎鼎的，适用于 JAVA 开发的 `log4j` ，便是一套设计优秀的日志管理包。

Python 中，也有一个模块 `logging` ，也能做到高效的日志管理。

例如，`logging` 模块，能按照指定周期切分日志文件。这一切的规则，都是为了实现对日志的高效管理。这些需求背后，对应着一套解决方案，也就是 logging 库，和它的四大组件：记录器、处理器、过滤器和格式化器。

下面是一个基本的日之类，同时将日志显示在控制台和写入文件中，同时按照天为周期切分日志文件。

```
import logging
from logging import handlers


class Logger(object):
    kv = {
        'debug': logging.DEBUG,
        'info': logging.INFO,
        'warning': logging.WARNING,
        'error': logging.ERROR,
        'crit': logging.CRITICAL
    }  # 日志级别关系映射

    def __init__(self, filename, level='info', when='D', backCount=3, fmt=
'%(asctime)s - %(pathname)s[line:%(lineno)d] - %(levelname)s: %(message)s'
):
        self.logger = logging.getLogger(filename)
        format_str = logging.Formatter(fmt)  # 设置日志格式
        self.logger.setLevel(self.kv.get(level))  # 设置日志级别
        sh = logging.StreamHandler()  # 往屏幕上输出
        sh.setFormatter(format_str)  # 设置屏幕上显示的格式
        th = handlers.TimedRotatingFileHandler(
            filename=filename, when=when, backupCount=backCount, encoding=
'utf-8')
        th.setFormatter(format_str)  # 设置文件里写入的格式
        self.logger.addHandler(sh)  # 把对象加到logger里
        self.logger.addHandler(th)
```

创建 `log` 对象，日志级别为 `debug` 及以上的写入日志文件

```
log = Logger('all.log', level='debug').logger
```

创建 `Student` 类，`score` 属性取值只能为整型。

```
class Student:
    def __init__(self, id, name):
        self.id = id
        self.name = name
        log.info('学生 id: %s, name: %s' % (str(id), str(name)))

    @property
    def score(self):
        return self.__score

    @score.setter
    def score(self, score):
        if isinstance(score, int):
```

```
            self.__score = score
            log.info('%s得分:%d' % (self.name, self.score))
        else:
            log.error('学生分数类型为 %s, 不是整型' % (str(type(score))))
            raise TypeError('学生分数类型为 %s, 不是整型' % (str(type(score)))
)
```

下一章

互动评论

说点什么

评论

The Scrapper 1 个月前

工作日志这块，看的不太懂了

鼓掌

存

1