小强函数进阶使用练习

113 Python 查找变量遵守什么顺序？ *LEGB* 规则是啥？

在学习 Python 函数时，我们经常会遇到变量作用域的问题，有全局变量，局部变量等，Python 查找变量的顺序遵守 *LEGB* 规则，即遇到某个变量时：

- 优先从它所属的函数(local)内查找；
- 若找不到，并且它位于一个内嵌函数中，就再到它的父函数(enclosing)中查找；
- 如果还是找不到，再去全局作用域(global)查找；
- 再找不到，最后去内置作用域 (build-in) 查找。
- 若还是找不到，报错。

如下例子，变量 `c` 在局部作用域 (local) 被发现；变量 `b` 在 `parent` 函数和 `son` 函数间(enclosing)被发现；变量 `a` 在全局作用域(global)被发现 ； `min` 函数属于 Python 中 内置函数 ，所以在搜寻完 *LEG* 三个区域后，最终在 `build-in` 域被找到。

复制

```
a = 10
def parent():
    b = 20
    def son():
        c = 30 # c: local
        print(b + c) # b: enclosing
        print(a + b + c ) # a: global
        print(min(a,b,c)) # min : built-in

    son()


In [72]: parent()
50
60
10
```

如下变量 `d`，在 `LEGB` 四个域都被搜寻一遍后，还是未找到，就会抛出 `d` 没有被发现的异常。

复制

```
a = 10
def parent():
    b = 20
    def son():
        c = 30 # c: local
        print(b + c) # b: enclosing
        print(a + b + c ) # a: global
        print(min(a,b,c)) # min : built-in
        print(d) # 在 LEGB 四个域都未找到后，报错！

    son()

parent()

# NameError: name 'd' is not defined
```

114 可变对象与可哈希的关系？

可变对象英文名称 mutable，如 list, dict 都是不可哈希的(unhashable);只有不可变对象才是可哈希的。

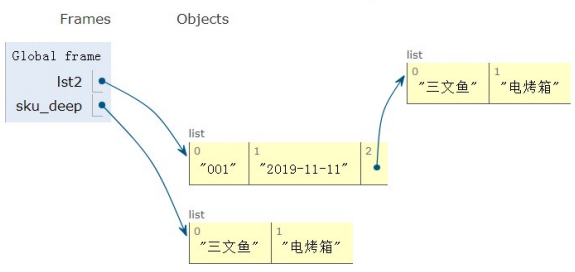
115 什么是浅拷贝？ Python 中列表对象怎么实现浅拷贝？

仅仅实现 list 对象中内嵌对象的一层拷贝，属于浅拷贝，英文： `shallow copy`.

复制

```
lst2 = ['001','2019-11-11',['三文鱼','电烤箱']]
sku_deep = lst2[2].copy()
```

可视化图如下，拷贝 lst2[2] 后，sku_deep 位于栈帧中指向一块新的内存空间：



此时，再对 sku_deep 操作，便不会影响 lst2[2] 的值。

116 什么是深拷贝？ Python 中怎么实现深拷贝？

要想实现深度拷贝，需要使用 `copy` 模块的 `deepcopy` 函数：

复制

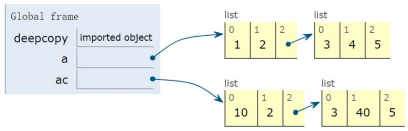
```
from copy import deepcopy

a = [1,2,[3,4,5]]
ac = deepcopy(a)
ac[0] = 10
ac[2][1] = 40
print(a[0] == ac[0])
print(a[2][1] == ac[2][1])
```

打印结果，都为 `False`，结合下图，也能看出内嵌的 list 全部完成复制，都指向了不同的内存区域。

Print output (drag lower right corner to resize)

```
False
False
```



117 Python 函数常见的有哪五类参数

- 位置参数
- 关键字参数
- 默认参数
- 可变位置参数
- 可变关键字参数

118 Python 函数常见的五类参数案例

定义函数 `f`，只有一个参数 `a`，`a` 既可能为位置参数，也可能为关键字参数，这取决于调用函数 `f` 的传参。

复制
```
def f(a):
  print(f'a:{a}')
```

下面这样调用 `f`，`a` 就是位置参数，英文 positional argument ，并且 `a` 被赋值为 1

复制
```
In [8]: f(1)
a:1
```

如果这样调用 `f`，`a` 就是关键字参数，英文 keyword argument，`a` 同样被赋值为1：

复制
```
In [9]: f(a=1)
a:1
```

如果函数 `f` 定义为如下，有一个默认值 0

复制
```
def f(a=0):
    print(f'a:{a}')
```

那么，`a` 就是默认参数，且默认值为 0.

有以下两种不同的调用方式：

- 按照 `a` 的默认值调用

复制
```
In [11]: f()
a:0
```

- 默认参数 a 值为 1

复制
```
In [12]: f(1)
a:1
```

如果函数 `f` 定义为，如下：

复制
```
def f(a,*b,**c):
    print(f'a:{a},b:{b},c:{c}')
```

函数 `f` 的参数稍显复杂，但也是最常用的函数参数定义结构。

出现带一个星号的参数 `b`，这是可变位置参数；

带两个星号的参数 `c`，这是可变关键字参数。

可变表示函数被赋值的变量个数是变化的

例如，可以这样调用函数：

复制
```
f(1,2,3,w=4,h=5)
```

参数 `b` 被传递 2 个值，参数 `c` 也被传递 2 个值。

可变位置参数 `b` 被解析为元组，可变关键字参数 `c` 被解析为字典。

复制
```
In [17]: f(1,2,3,w=4,h=5)
a:1,b:(2, 3),c:{'w': 4, 'h': 5}
```

如果这样调用函数：

复制
```
f(1,2,w=4)
```

参数 `b` 被传递 1 个值，参数 `c` 被传递 1 个值。

复制
```
In [18]: f(1,2,w=4)
a:1,b:(2,),c:{'w': 4}
```

所以，称带星号的变量为可变的。

119 Python 中如何查看参数的类型

借助 Python 的 `inspect` 模块查看参数的类型。

首先，导入 `inspect` 模块：

复制
```
from inspect import signature
```

函数 `f` 定义为：

```
def f(a,*b):
  print(f'a:{a},b:{b}')
```

查看参数类型：

```
In [24]: for name,val in signature(f).parameters.items():
    ...:     print(name,val.kind)

a POSITIONAL_OR_KEYWORD
b VAR_POSITIONAL
```

看到，参数 `a` 既可能是位置参数，也可能是关键字参数。

参数 `b` 为可变位置参数。

但是，如果函数 `f` 定义为：

```
def f(*,a,**b):
  print(f'a:{a},b:{b}')
```

查看参数类型：

```
In [22]: for name,val in signature(f).parameters.items():
    ...:     print(name,val.kind)
    ...:
a KEYWORD_ONLY
b VAR_KEYWORD
```

看到参数 `a` 只可能为 `KEYWORD_ONLY` 关键字参数。

因为参数 `a` 前面有个星号，星号后面的参数 `a` 只可能是关键字参数，而不可能是位置参数。

不能使用如下方法调用 `f`：

```
f(1, b=2, c=3)

TypeError: f() takes 0 positional arguments but 1 was given
```

120 Python 函数参数传值 6 大规则及常见错误解析

Python 强大多变，原因之一在于函数参数的多样化。

方便的同时，也要求开发者遵守更多的约束规则。

如果不了解这些规则，函数调用时，可能会出现各种各样的调用异常。

常见的有以下六类：

SyntaxError: positional argument follows keyword argument，位置参数位于关键字参数后面

TypeError: f() missing 1 required keyword-only argument: 'b'，必须传递的关键字参数缺失

SyntaxError: keyword argument repeated，关键字参数重复

TypeError: f() missing 1 required positional argument: 'b'，必须传递的位置参数缺失

TypeError: f() got an unexpected keyword argument 'a'，没有这个关键字参数

TypeError: f() takes 0 positional arguments but 1 was given，不需要位置参数但却传递 1 个

下面总结， 6 个主要的参数使用规则。

规则 1：不带默认值的位置参数缺一不可

函数调用时，根据函数定义的参数位置来传递参数，是最常见的参数类型。

```
def f(a):
  return a

f(1) # 这样传递值，参数 a 为位置参数
```

如下带有两个参数，传值时必须两个都赋值。

```
def f(a,b):
    pass

In [107]: f(1)

TypeError: f() missing 1 required positional argument: 'b'
```

规则 2：关键字参数必须在位置参数右边

在调用 `f` 时，通过**键--值**方式为函数形参传值。

```
def f(a):
  print(f'a:{a}')
```

参数 `a` 变为关键字参数

```
f(a=1)
```

但是，下面调用，就会出现：位置参数位于关键字参数后面的异常。

```
def f(a,b):
    pass

In [111]: f(a=1,20.)

SyntaxError: positional argument follows keyword argument
```

规则 3：对同一个形参不能重复传值

下面调用也不OK:

```
def f(a,**b):
    pass

In [113]: f(1,width=20.,width=30.)

SyntaxError: keyword argument repeated
```

规则 4：默认参数的定义应该在位置形参右面

在定义函数时，可以为形参提供默认值。

对于有默认值的形参，调用函数时，如果为该参数传值，则使用传入的值，否则使用默认值。

如下 `b` 是默认参数：

```
def f(a,b=1):
  print(f'a:{a}, b:{b}')
```

默认参数通常应该定义成不可变类型。

规则 5：可变位置参数不能传入关键字参数

如下定义的参数 a 为可变位置参数：

```
def f(*a):
  print(a)
```

调用方法：

```
In [115]: f(1)
(1,)

In [116]: f(1,2,3)
(1, 2, 3)
```

但是，不能这么调用：

```
In [117]: f(a=1)

TypeError: f() got an unexpected keyword argument 'a'
```

规则 6：可变关键字参数不能传入位置参数

如下，`a` 是可变关键字参数：

```
def f(**a):
  print(a)
```

调用方法：

```
In [119]: f(a=1)
{'a': 1}

In [120]: f(a=1,b=2,width=3)
{'a': 1, 'b': 2, 'width': 3}
```

但是，不能这么调用：

```
In [121]: f(1)

TypeError: f() takes 0 positional arguments but 1 was given
```

121 高阶函数 filter 使用案例解读

过滤器，过滤掉不满足函数 `function` 的元素，重新返回一个新的迭代器。

这个函数大概等价于下面自定义函数 `filter_self` ：

```
def filter_self(function,iterable):
    return iter([ item for item in iterable if function(item)])
```

`filter_self` 函数接收一个 `function` 作为参数，满足条件的元素才得以保留。

调用 `filter_self` ，筛选出满足指定身高的学生。其条件是，男生身高超过 1.75， 女生身高超过 1.65.

```
class Student():
    def __init__(self,name,sex, height):
        self.name = name
        self.sex = sex
        self.height = height

def height_condition(stu):
    if stu.sex == 'male':
        return stu.height > 1.75
    else:
        return stu.height > 1.65

students = [Student('xiaoming','male',1.74),
           Student('xiaohong','female',1.68),
           Student('xiaoli','male',1.80)]
students_satisfy = filter_self(height_condition,students)
for stu in students_satisfy:
    print(stu.name)
```

打印结果：

```
xiaohong
xiaoli
```

解释下 `height_condition` 函数，其第一个参数是可迭代对象 `iterable` 中的一个元素，这是值得注意的。

以上使用自定义过滤函数，下面使用 Python 内置的 `filter` 函数，也实现一遍学生身高筛选功能。

```
students_satisfy = filter(height_condition,students)
for stu in students_satisfy:
    print(stu.name)
```

打印结果如下，与上面使用自定义过滤函数实现的结果相同。

```
xiaohong
xiaoli
```

122 高级函数之 map 函数使用案例

它将 function 映射于 iterable 中的每一项，并返回一个新的迭代器。

如下，`map` 函数实现每个元素加 1 ：

```
In [10]: mylst = [1,3,2,4,1]
In [12]: result = map(lambda x: x+1, mylst)
In [13]: result
Out[13]: <map at 0x214b38353c8>
In [14]: list(result)
Out[14]: [2, 4, 3, 5, 2]
```

同时注意到，`map` 函数支持 传入多个可迭代对象。

当传入多个可迭代对象时，输出元素个数等于较短序列长度。

如下，传入两个列表，`function` 就需要接受两个参数，取值分别对应第一、第二个列表中的元素。

找到同时满足第一个列表的元素为奇数，第二个列表对应位置的元素为偶数的元素。

```
In [63]: xy = map(lambda x,y: x%2==1 and y%2==0, [1,3,2,4,1],[3,2,1,2])

In [64]: for i in xy:
    ...:     print(i)

False
True
False
False
```

借助 `map` 函数，还能实现向量级运算。

```
In [65]: lst1 = [1,2,3,4,5,6]
In [66]: lst2 = [3,4,5,6,3,2]

In [68]: def vector_add(x,y):
    ...:     return list(map(lambda i,j: i+j, x,y))

In [69]: vector_add(lst1,lst2)
Out[69]: [4, 6, 8, 10, 8, 8]
```

同时还支持，向量长度不等的加法运算：

```
In [65]: lst1 = [1,2,3,4,5,6]
In [70]: lst3 = [1,2]

In [71]: vector_add(lst1,lst3)
Out[71]: [2, 4]
```

123 高阶函数之 reduce 使用案例

reduce(function, iterable[, initializer])

提到 `map` ，就会想起 `reduce` ，前者生成映射关系，后者实现归约。

`reduce` 函数位于 `functools` 模块中，使用前需要先导入。

```
In [72]: from functools import reduce
```

`reduce` 函数中第一个参数是函数 `function`。`function` 函数，参数个数必须为 **2**，是可迭代对象 `iterable` 内的连续两项。

计算过程，从左侧到右侧，依次归约，直到最终为单个值并返回。

```
In [73]: reduce(lambda x,y: x+y,list(range(10)))
Out[73]: 45
```

以上 `reduce` 的完整过程，参考下面的演示动画：

124 常用函数之 reversed

reversed(seq)

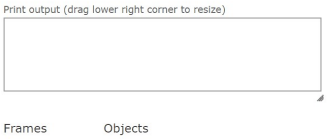
重新生成一个反向迭代器，对输入的序列实现反转。

复制

```
In [155]: rev = reversed([1,4,2,3,1])

In [156]: for i in rev:
     ...:     print(i)
1
3
2
4
1
```

反转过程，动画演示如下：



125 常用函数之 sorted

sorted(iterable, *, key=None, reverse=False)

实现对序列化对象的排序

`key` 参数和 `reverse` 参数必须为关键字参数，都可省略。

复制

```
In [1]: a = [1,4,2,3,1]
In [2]: sorted(a,reverse=True)
Out[2]: [4, 3, 2, 1, 1]
```

如果可迭代对象的元素也是一个复合对象，如下为字典。

依据依据为字典键的值，`sorted` 的 `key` 函数就会被用到。

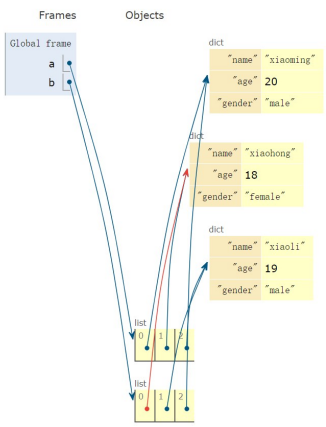
复制

```
In [80]: a = [{'name':'xiaoming','age':20,'gender':'male'},
    ...: {'name':'xiaohong','age':18,'gender':'female'},
    ...: {'name':'xiaoli','age':19,'gender':'male'}]

In [81]: b = sorted(a,key=lambda x: x['age'],reverse=False)

In [82]: b
Out[82]:
[{'name': 'xiaohong', 'age': 18, 'gender': 'female'},
 {'name': 'xiaoli', 'age': 19, 'gender': 'male'},
 {'name': 'xiaoming', 'age': 20, 'gender': 'male'}]
```

排序的可视化图，如下所示：



126 常用函数之 iter

iter(object[, sentinel])

返回一个严格意义上的可迭代对象，其中，参数 `sentinel` 可有可无。

复制

```
In [21]: lst = [1,3,5]
In [22]: it = iter(lst)
In [23]: it
Out[23]: <list_iterator at 0x214b3886188>

In [26]: it.__next__()
Out[26]: 1

In [27]: it.__next__()
Out[27]: 3

In [28]: it.__next__()
Out[28]: 5

In [29]: it.__next__()
---------------------------------------------------------------------------
--
StopIteration                             Traceback (most recent call las
t)
<ipython-input-29-74e64ed6c80d> in <module>
----> 1 it.__next__()

StopIteration:
```

`it` 迭代结束后，再 `__next__` 时，触发 `StopIteration` 异常，即迭代器已经执行到最后。

下一章

互动评论

说点什么

评论

The Scrapper

1 个月前

练习题也不错

鼓掌

存