python中对象引用

1. 引言

引用在各种编程语言中都有涉及，如java中值传递和引用传递。python的对象引用也是学习python过程中需要特别关注的一个知识点，特别是对函数参数传递，可能会引起不必要的BUG。本文将对引用做一个梳理，内容涉及如下：

- 变量和赋值
- 可变对象和不可变对象
- 函数参数的引用
- 浅拷贝和深拷贝
- 垃圾回收
- 弱引用

2. python引用

2.1 变量和赋值

任何一个python对象都有标签，类型和值三个属性。标签在对象创建后直到内存回收就保持不变，可以理解为内存地址。

python在给变量赋值，会把赋值的对象的内存地址赋值给变量。也就是说python的变量是地址引用式的变量。引用语义的方式，存储的只是一个变量的值所在的内存地址，而不是这个变量的值本身。

可以通过`is`或者比较`id()`的判断是否引用的是同一个内存地址的变量。

- `==` 是比较两个对象的内容是否相等，即两个对象的值是否相等
- `is`同时检查对象的值和内存地址。可以通过is判断是否是同一个对象
- `id()` 列出变量的内存地址的编号

```
# 这个例子a 和 b 两个变量共同指向了同一个内存空间
a = [1, 2, 3]
c = [1, 2, 3]
print(a is c) # False
print(a == c) # True
b = a
a.append(5)
print(a is b) # True
```

- 初始化赋值：变量的每一次初始化，都开辟了一个新的空间，将新内容的地址赋值给变量
- 变量赋值：引用地址的传递

2.2 可变对象和不可变对象

python中的一切都是对象。python对象又分为可变对象和不可变对象。二者的区别在于对象的值在不改变内存地址的情况下是否可修改。

- 可变对象包括字典dict、列表list、集合set、手动声明的类对象等
- 不可变对象包括数字int float、字符str、None、元组tuple等

下面举几个典型的例子：

- list 可变对象，内容变更地址不变

```
a = [1, 2, 3]
print(id(a))
a.append(5)
print(id(a))
```

- 不可变对象（常用的共享地址或缓存）

```
# 较小整数频繁被使用，python采用共享地址方式来管理
a = 1
b = 1
print(a is b) # True

# 对于单词类str，python采用共享缓存的方式来共享地址
a = 'hello'
b = 'hello'
print(a is b) # True
```

- 不可变对象（不共享地址）

```
a = (1999, 1)
b = (1999, 1)
print(a is b) # False

a = 'hello everyone'
b = 'hello everyone'
print(a is b) # False
```

- 元组的相对不可变型

```
# 元组的里元素是可变，改变可变的元素，不改变元组的引用
a = (1999, [1, 2])
ida = id(a)
a[-1].append(3)
idb = id(a)

print(ida == idb) # True
```

这里之所以要提到变量可变和不可变的特性，其实主要是想说明变量的引用其实和变量的可变和不可变没有直接的关系。变量可变和不可变主要着眼点是变量可不可以修改，注意这个修改不是通过赋值的操作来完成。

```
a = [1, 2, 3]
print(id(a))
a = a + [5]
print(id(a))
# 前后两个变量a,  已经不是同一个地址了
```

2.3 函数参数的引用

python中函数的传参方式是共享传参，即函数的形参是实参中各个引用的副本（别名）。函数会修改是可变对象的实参（表示的同一个对象）；而不会改变实参的引用。

```
def func(d):
    d['a'] = 10
    d['b'] = 20    # 改变了外部实参的值
    d = {'a': 0, 'b': 1}  # 赋值操作，  局部d贴向了新的标识
    print(d) # {'a': 0, 'b': 1}

d = {}
func(d)
print(d) # {'a': 10, 'b': 20}
```

建议不要写上面例子的代码，局部变量和全局变量的名称一样，尽量编码，否则很容易出bug而不自知。

函数的参数的默认值避免使用可变参数，尽量用None来代替。原因是函数的默认值是作为函数对象的属性，如果默认值是可变对象，而且修改了它，那边后续的函数对象都会受到影响。

```
class bus():
    def __init__(self, param=[]):
        self.param = param

    def test(self, elem):
        self.param.append(elem)
b = bus([2, 3])
b.param # [2, 3]

c = bus()
c.test(3)
c.param # [3]

d = bus()
d.param # [3]  # c 中修改了默认值的引用的内容
```

2.4 浅拷贝和深拷贝

对于可变对象，我们要时刻注意它的可变性，特别是对赋值或者拷贝后的变量做内容修改操作的时候，需要考虑下是否会影响到原始变量的值，如果程序有bug，可以往这方面想一想。这里讲一下拷贝即建立副本。拷贝有两种：

- 浅拷贝： 只复制顶层的对象，对于有嵌套数据结构，内部的元素还是原有对象的引用，这时候需要特别注意
- 深拷贝：复制了所有对象，递归式的复制所有对象。复制后的对象和原来的对象是完全不同的对象。 对于不可变对象来说，浅拷贝和深拷贝都是一样的地址。但是对于嵌套了可变对象元素的情况，就有所不同

```
test_a = (1, 2, 3)
test_b = copy.copy(test_a)
test_c = copy.deepcopy(test_a)
print(test_a is test_b) # True
print(test_a is test_c) # True

test_a[2].append(5) # 改变不可变对象中可变元素的内容
print(test_a is test_b) # True
print(test_a is test_c) # False
print(test_c) # (1, 2, [3, 4])
```

对于可变对象，只要拷贝，就创建了一个新的顶层对象。如果是浅拷贝，内部嵌套的可变对象只是拷贝引用，这样就会相互影响。深拷贝就不会有这种问题。

```
l1 = [3, [66, 55, 44], (2, 3, 4)]
l2 = list(l1) # l2是l1的浅拷贝

# 顶层改变不会相互影响，因为是两个不同对象
l1.append(50)
print(l1) # 3, [66, 55, 44], (2, 3, 4), 50]
print(l2) # [3, [66, 55, 44], (2, 3, 4)]

# 嵌套可变元素，浅拷贝共享一个地址
l1[1].append(100)
print(l1) # [3, [66, 55, 44, 100], (2, 3, 4), 50]
print(l2) # [3, [66, 55, 44, 100], (2, 3, 4)]

# 嵌套不可变元素，不可变元素的操作是创建一个新的对象，所以不影响
l1[2] += (2,3)
print(l1) # [3, [66, 55, 44, 100], (2, 3, 4, 2, 3), 50]
print(l2) #[3, [66, 55, 44, 100], (2, 3, 4)]
```

2.5 垃圾回收

python对于垃圾回收，采取的是**引用计数**为主，标记-清除+分代回收为辅的回收策略。

- 引用计数： python可以给所有的对象（内存中的区域）维护一个引用计数的属性，在一个引用被创建或复制的时候，让python,把相关对象的引用计数+1；相反当引用被销毁的时候就把相关对象的引用计

数-1。当对象的引用计数减到0时，认为整个python中不会再有变量引用这个对象，所以就可以把这个对象所占据的内存空间释放出来了。可以通过`sys.getrefcount()`来查看对象的引用

- 分代回收: 分代回收主要是为了提高垃圾回收的效率。对象的创建和消费的频率不一样。由于python在垃圾回收前需要检测是否是垃圾，是否回收，然后再回收。当对象很多的时候，垃圾检测的耗时变得很大，效率很低。python采用的对对象进行分代，按不同的代进行不同的频率的检测。 代等级的规则根据对象的生命时间来判断，比如一个对象连续几次检测都是可达的，这个对象代的等级高，降低检测频率。python中默认把所有对象分成三代。第0代包含了最新的对象，第2代则是最早的一些对象
- 循环引用： 一个对象直接或者间接引用自己本身，引用链形成一个环。这样改对象的引用计数永远不可能为0。所有能够引用其他对象的对象都被称为容器(container). 循环引用只能发生容器之间发生. Python的垃圾回收机制利用了这个特点来寻找需要被释放的对象。

```
import sys
a = [1, 2]
b = a

print(sys.getrefcount(a)) # 3 命令本身也是一次引用
del b
print(sys.getrefcount(a)) # 2
```

3. 弱引用

弱引用在许多高级语言中都存在。如前所述，当对象的引用计数为0时，垃圾回收机制就会销毁对象。但有时候需要引用对象，但不希望增加引用计数。这样有什么好处？

- 应用在缓存中，只存在一定的时间存在。当它引用的对象存在时，则对象可用，当对象不存在时，就返回None
- 不增加引用计数，在循环引用使用，就降低内存泄露的可能性

这就是弱引用weak reference，不会增加对象的引用数量。引用的目标对象称为 所指对象 （referent）。

```
import weakref
a_set = {0,1}
wref = weakref.ref(a_set) # 建立弱引用
print(wref()) # {0,1}
a_set = {2, 3, 4} # 原来的a_set 引用计数为0，垃圾回收
print(wref()) # None # 所值对象被垃圾回收， 弱引用也消失为None
```

弱引用一般使用时weakref集合, weakref.WeakValueDictionary, weakref.WeakKeyDictionary两者的区别是一个是值进行弱引用，一个是可以进行弱引用；另外还有weakref.WeakSet

4. 总结

本文描述python中引用相关的几个方面，希望对大家有帮助。总结如下：

- 对象赋值就完成引用，变量是地址引用式的变量
- 要时刻注意，所以引用可变对象对象的改变，是否导致共同引用的变量值得变化
- 函数会修改是可变对象的实参

- 浅拷贝只是copy顶层，如果存在内部嵌套可变对象，要注意，copy的还是引用
- 对象的引用计数为0时，就开始垃圾回收
- 弱引用为增加引用计数，与被所指对象共存亡，而不影响循环引用