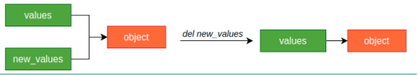
Weakref

del与垃圾回收

del

del语句销毁的是一个名称，而不是一个对象。del命令可能会导致对象被当做垃圾回，但是前提条件是删除的变量保存的是对象的最后一个引用

```
In [19]: values = [i for i in range(5)]
In [20]: new_values = values

In [21]: new_values
Out[21]: [0, 1, 2, 3, 4]

In [22]: del new_values

In [23]: new_values
---------------------------------------------------------------------------
NameError
<ipython-input-23-bf2d15c7d78c> in <module>()
----> 1 new_values
NameError: name 'new_values' is not defined

In [24]: values
Out[24]: [0, 1, 2, 3, 4]
```



垃圾回收

在CPython实现中，垃圾回收算法主要以引用计数为主，分代回收为辅的实现，其中分代回收主要是为了处理循环引用的对象回收

不必过于纠结分代回收机制，反正每个对象都会统计有多少个引用指向自己，当引用计数归零时，对象将被立即销毁

弱引用

查看引用计数

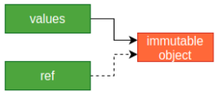
在sys模块中，提供了getrefcount()方法来查看某一个对象的引用计数。不过需要注意的是，调用该方法会使得原有的引用计数加1

```
In [4]: import sys

In [5]: values = [i for i in range(5)]

In [6]: sys.getrefcount(values)
Out[6]: 2  # 实际上为1

In [7]: numbers = values

In [8]: sys.getrefcount(values)
Out[8]: 3  # 实际上为2
```

基本概念

弱引用是相对于强引用的概念而言的，对于强引用，每添加一个变量指向对象时，其引用计数加1。但是弱引用在指向某一个对象时，其引用计数并不会增加

```
In [1]: import sys, weakref

In [2]: value_set = {1, 2, 3}

In [3]: sys.getrefcount(value_set)
Out[3]: 2

In [4]: ref = weakref.ref(value_set)

In [5]: sys.getrefcount(value_set)
Out[5]: 2   # 引用计数并没有增加

In [6]: sys.getrefcount(ref)
Out[6]: 2
```



weakref.ref()将会在某一个对象上添加一个引用，前提是对象为不可变对象，这一点稍后会提及

从示例中可以看到，weakref.ref()在指向一个对象时，其引用计数没有增加，那么就不会妨碍对象的垃圾回收

weakref.ref()算是最底层的低级接口，Python据此封装了WeakKeyDictionary、WeakValueDictionary、WeakSet等weakref集合

WeakValueDictionary

顾名思义，WeakValueDictionary首先是一个字典，并且字典中的值是对象的一个弱引用。被引用的对象在程序中其它地方被当做垃圾回收时，对应的key-value条目将会自动地从WeakValueDictionary中清除

基本使用

```
In [1]: import weakref

In [2]: class CacheItem(object):
   ...:     def __init__(self, key, value):
   ...:         self.key = key
   ...:         self.value = value

In [3]: caches = [CacheItem("smart", {"name": "smart"})]
In [4]: storage = weakref.WeakValueDictionary()

In [5]: for cache in caches:
   ...:     storage[cache.key] = cache


In [6]: storage["smart"].value
Out[6]: {'name': 'smart'}

In [7]: del caches[0]

In [8]: storage.get("smart") is None
Out[8]: True
```

WeakValueDictionary的应用

singledispatch

在functools.py中，定义了singledispatch这么一个函数，该函数本质上是一个闭包，也就是说，可以使用@语法糖来对另一个函数进行装饰

该函数的作用就是将一个普通的函数编程一个泛函数(generic function)，简单来说就是支持能够让某一个函数获得诸如Java一样的函数重载功能

```
In [23]: from functools import singledispatch

In [25]: @singledispatch
    ...: def overload():
    ...:     pass

In [26]: @overload.register(str)
    ...: def process(value):
    ...:     print(type(value))
    ...:

In [27]: @overload.register(list)
    ...: def process(value):
    ...:     print(type(value))


In [28]: process("smart")
<class 'str'>

In [29]: process([1, 2, 3])
<class 'list'>
```

如此一来即可在Python中通过取巧的方式来实现函数的重载。当然，在Python语言中，这东西其实挺鸡肋的...

在singledispatch函数内部，即使用了WeakValueDictionary

finalize

这玩意儿和Java里面的finalize是一样的，当对象要被GC回收时，调用finalize()内部的方法(Java)或者是调用由用户指定的方法(Python)

Java

在Java Object类中，有一个空的finalize方法，该方法将会在对象准备被GC回收时调用。自然而然地，许多Java程序员认为可以在此处进行一些资源清理工作，例如关闭文件描述符等工作

实际上该方法不应该进行由Java本身所创建资源的清理工作，最典型的例子就是关闭文件描述符

因为一个对象何时被清理是完全不可知的，同时也可能永久的存在与JVM堆中，那么此时finalize()方法将永远不会被执行，相应的资源也不会被释放

finalize()真正应该清理的是堆外内存资源，而不是堆内资源

Python

在Python的object类中虽然没有finalize方法，但是却定义了__del__方法，同Java一样，该方法将会在对象被GC回收之前进行调用

与Java的finalize()方法一样，__del__一定是释放外部资源，而不是内部资源

```
In [1]: s = {1, 2, 3}

In [2]: def say_bye():
    ...:     print("bye~")

# 仅作为示例使用
In [3]: weakref.finalize(s, say_bye)

In [4]: del s
bye~
```