Python基础教程

第8章 内存管理与编解码

8.1 内存管理

内存管理，在很大程度上决定了Python的执行效率，因为在Python的运行中，会创建和销毁大量的对象，这些都涉及到内存的管理。

- 变量无需事先声明；
- 变量无需指定类型；
- 程序员不用关心内存管理；
- 变量名会被“回收”；
- del语句能够直接释放资源；

8.2.1 变量定义

大多数编译型语言，变量在使用前必须先声明。

在**Python**中，无需显式变量声明语句，变量在第一次被赋值时自动声明。和其它大多数语言一样，变量只有被创建和赋值后才能被使用。

```
>>> a
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

```
>>> x = 4
>>> y = 'this is a string'
>>> x
4
>>> y
'this is a string'
```

变量一旦被赋值，就可以通过变量名来访问它。

8.2.2 动态类型

（1）Python中不但变量名无需事先声明，也无需类型声明！

（2）对象的类型和内存占用都是运行时确定的。

（3）在创建，也就是赋值时，解释器会根据语法和右侧的操作数来决定新对象的类型。

（4）在对象创建后，一个该对象的引用会被赋值给左侧的变量。

（5）变量名没有类型，类型属于对象而不是变量名。从另一方面讲，对象知道自己的类型。每个对象都包含了一个头部信息，其中标记了这个对象的类型。

- >>> a = 5
- >>> b = 5
  - 问题：a 和 b是否引用同一个对象？

  >>> id(a) == id(b)

  >>> a is b
  - 对大整数和浮点数呢？
- >>> a = [1, 2, 3]
- >>> b = [1, a]
  - 问题：b是否引用了a?

8.2.3 内存分配

一个优秀的程序员，应该清楚使用完变量后，要释放系统资源！

Python解释器承担了内存管理的复杂任务。

许多时候申请的内存都是小块的内存，这些小块内存在申请后，很快又会被释放，由于这些内存的申请并不是为了创建对象，所以并没有对象一级的内存池机制。

这就意味着运行期间会大量地执行**malloc**和**free**的操作，频繁地在用户态和核心态之间进行切换，这将严重影响**Python**的执行效率。

为了加速执行效率，引入了一个内存池机制，用于管理对小块内存的申请和释放。

8.2.3 Pymalloc机制管理内存分配

Python内部默认的小块内存与大块内存的分界点定在256个字节，这个分界点由名为SMALL_REQUEST_THRESHOLD的符号控制。

Python中的内存分配管理机制有两套实现：

- 针对小对象，当申请的内存小于256字节时，PyObject_Malloc会在内存池中申请内存；
- 当申请的内存大于256字节时，PyObject_Malloc的行为将蜕化为malloc的行为，即会直接执行new/malloc的行为来申请内存空间。

8.2.3 基本数据类型的内存分配

- Python中有一小部分的对象是不使用Pymalloc进程内存分配的，主要是integer/float/list/dict。
- 为了提升这些常用对象的内存使用效率，会保存在单独的列表中。
- Python为list/dict采用不同的策略，会最多保留80个空闲的list/dict，如果多余80个，多出的会被释放。

8.2.3 基本数据类型的内存分配

- Python通过malloc()为Integer/Float两种类型分配大约1KB大小的内存块列表，这些列表被当做Integer/Float的数组使用，而不是使用Pymalloc的分配的8字节的整数倍大小的Block，以减少内存消耗。
- 在创建一个新的Integer/Float对象时，字直接从这个内存列表中获取数据，或是重新分配一块新的Block；
- 在释放时对应的Block重新加入到列表中。这些Block也是不会被返还给操作系统的。

8.2.4 引用计数

- 每个对象各有多少个引用，简称引用计数；
- 为了追踪内存中的对象，**Python**使用了引用计数技术。也就是说**Python**内部记录着所有使用中的对象各有多少引用；
- 一个内部跟踪变量，称为一个引用计数器。当对象被创建时，就创建了一个引用计数，当这个对象不再需要时，即这个对象的引用计数变为**0**时，它就被垃圾回收（释放内存）。

8.2.4 引用计数-增加引用计数

当对象被创建并（将其引用）赋值给变量时，该对象的引用计数被设置为1。

当同一个对象（的引用）又被赋值给其他变量时，或作为参数传递给函数、方法、或类的实例时，或者被赋值为一个窗口的成员时，该对象的一个新的引用，或者称作别名，就被创建，则该对象的引用计数自动加1。

- 对象被创建

  例: ***x = 5***
- 或另外的别名被创建

  例 ***y = x***
- 或被作为参数传递给函数（新的本地引用）

  例 ***foo(x) foo() is a function***
- 或成为容器对象的一个元素

  例 ***myList = [sdf, x, 656]***

```
>>> from sys import getrefcount
>>> a = [1, 2, 3]
>>> getrefcount( a )
2
>>> b = a
>>> getrefcount( a )
3
>>> c = [a, a]
>>> getrefcount( a )
5
>>> d = [a, a, a]
>>> getrefcount( a )
8
```

8.2.4 引用计数-减少引用计数（1）

（1）当对象的引用计数被撤销时，引用计数就会减少；

（2）当变量被赋给另外一个对象时,原对象的计数也会自动减1；

（3）其他造成对象的引用计数减少的方式包括使用del语句删除一个变量，或者当一个对象被移出一个窗口对象时（或该容器对象本身的引用计数变成了0时）。

8.2.4 引用计数-减少引用计数（2）

一个对象的引用计数在以下情况下会减少：

- 一个本地引用离开了其作用范围
- 对象的别名被显式销毁

  del y　　**y** 是一个对象

- 对象的一个别名被赋值给其他对象

  x = 123

- 对象被从一个窗口对象中移除

  myList.remove(x)

- 窗口对象本身被销毁

  del myList

8.2.4 引用计数- del语句

del语句会删除对象的一个引用，语法为：

```
del obj1, obj2, obj3,...
```

执行del x（x的引用计数为1）语句就会删除该对象的最后一个引用，也就是该对象的引用计数会减为0，这会导致该对象从此“无法访问”或“无法抵达”。

从此刻起，该对象就成为垃圾回收机制的回收对象。

8.2.5 垃圾收集（1）

不再使用的内存会被一种称为垃圾收集的机制释放。

垃圾收集器负责释放内存，垃圾收集器用来寻找引用计数为0的对象，它也负责检查那些虽然引用计数大于0但也应该被销毁的对象。

但特殊情况会导致循环引用，如：

- 一个循环引用发生在当你有至少两个对象互相引用时，也就是说所有的引用都消失了，这些引用仍然存在；
- 这说明只靠引用计数是不够的。

8.2.5 垃圾收集（2）

Python的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。

当一个对象的引用计数变为**0**，解释器会暂停，释放掉这个对象和仅有这个对象可访问的其他对象。

作为引用计数的补充，垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象，这时，解释器会试图清理所有未引用的循环。

8.2.5 垃圾收集（3）

- 垃圾回收时，**Python**不能进行其它的任务。
- 频繁的垃圾回收将大大降低**Python**的工作效率。如果内存中的对象不多，就没有必要总启动垃圾回收。
- **Python**只会在特定条件下，自动启动垃圾回收。
  - 当**Python**运行时，会记录其中分配对象(**object allocation**)和取消分配对象(**object deallocation**)的次数。当两者的差值高于某个阈值时，垃圾回收才会启动。
  - 可以通过**gc**模块的**get_threshold()**方法，查看该阈值，也可以手动启动垃圾回收，即使用**gc.collect()**。

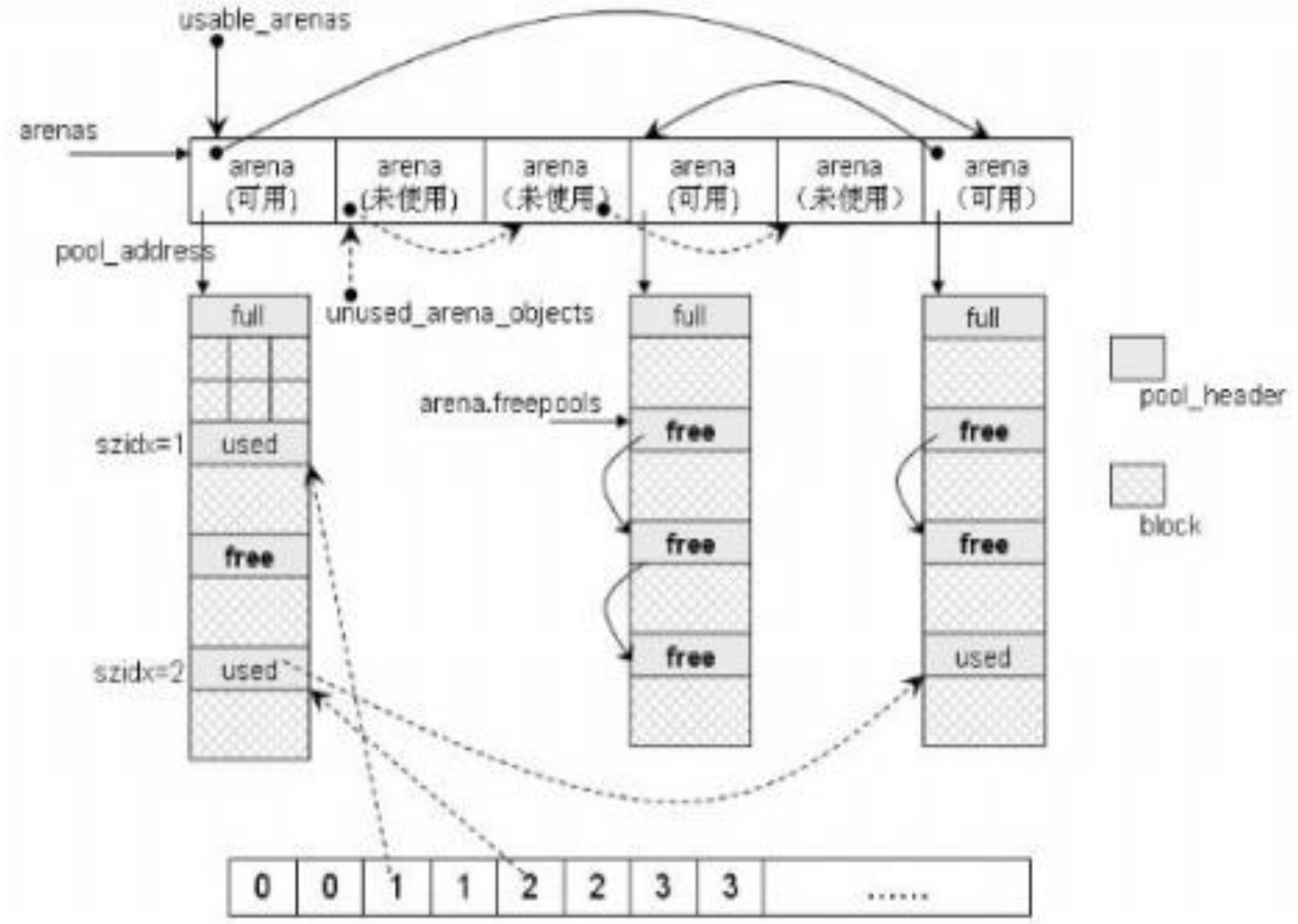
8.2.5 垃圾收集（3）

- Python同时还采用了分代(generation)回收的策略。
- 这一策略的基本假设是，存活时间越久的对象，越不可能在后面的程序中变成垃圾。
- 程序往往会产生大量的对象，许多对象很快产生和消失，但也有一些对象长期被使用。
- 出于信任和效率，对于这样一些“长寿”对象，编程者相信它们的用处，所以减少在垃圾回收中扫描它们的频率。

8.2.5 垃圾收集- Pymalloc机制管理内存释放

在一个对象的引用计数减为0时，与该对象对应的析构函数就会被调用，调用析构函数并不意味着最终一定会调用free释放内存空间，因为如果真是这样的话，那频繁地申请、释放内存空间会使 Python的执行效率大打折扣。

一般来说，Python中大量采用了内存对象池的技术，使用这种技术可以避免频繁地申请和释放内存空间。因此在析构时，通常都是将对象占用的空间归还到内存池中。



Python内存池全景

8.3 ASCII编码

八个二进制位就组合出256种状态，这被称为一个字节(byte)。也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从0000000到11111111。

上个世纪60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为ASCII码，一直沿用至今。

ASCII码一共规定了128个字符的编码，比如空格“SPACE”是32(二进制00100000)，大写的字母A是65(二进制01000001)。这128个符号(包括32个不能打印出来的控制符号)，只占用了一个字节的后面7位，最前面的1位统一规定为0。

8.4 非ASCII编码

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。比如，在法语中，字母上方有注音符号，它就无法用ASCII码表示。于是，一些欧洲国家就决定，利用字节中闲置的最高位编入新的符号。比如，法语中的é的编码为130(二进制10000010)。

但是，这里又出现了新的问题，不同的国家有不同的字母，即使都使用256个符号的编码方式，代表的字母却不一样。比如，130在法语编码中代表了é，在希伯来语编码中却代表了字母Gimel (ג)，在俄语编码中又会代表另一个符号。

所有这些编码方式中，0-127表示的符号是一样的，不一样的只是128-255的这一段。

8.5 GB2312字符集

汉字多达10万左右，其中有6000多个常用汉字。

于是规定：一个小于127的字符的意义与原来相同，但两个大于127的字符连在一起时，就表示一个汉字，前面的一个字节（称之为高字节）从0xA1用到 0xF7，后面一个字节（低字节）从0xA1到0xFE，这样就可以组合出大约7000多个简体汉字了。

在这些编码里，还把数学符号、罗马希腊的字母、日文的假名们都编进去了，连在 ASCII 里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的“全角”字符，而原来在127号以下的那些就叫“半角”字符了，这种汉字方案叫做"GB2312"。

GB2312 是对 ASCII 的中文扩展。

8.6 GB2312字符集的扩展

但是中国的汉字还是太多了，如朱镕基的“镕”字，不得不继续把GB2312 没有用到的码位找出来用上。

后来还是不够用，于是不再要求低字节一定是127号之后的内码，只要第一个字节是大于127就固定表示这是一个汉字的开始，不管后面跟的是不是扩展字符集里的内容。扩展之后的编码方案被称为 GBK 标准，GBK 包括了 GB2312 的所有内容，同时又增加了近20000个新的汉字（包括繁体字）和符号。

再次扩展，又加了几千个新的少数民族的字，GBK 扩成了 GB18030。

8.7 DBCS

中国的程序员们把这一系列汉字编码的标准为"**DBCS**"（**Double Byte Charecter Set** 双字节字符集）。

在**DBCS**系列标准里，最大的特点是两字节长的汉字字符和一字节长的英文字符并存于同一套编码方案里。

因此程序为了支持中文处理，必须要注意字串里的每一个字节的值，如果这个值是大于**127**的，那么就认为一个双字节字符集里的字符出现了。

8.8 Unicode

世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。

如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。

Unicode，就像它的名字都表示的，这是一种所有符号的编码，是一个很大的集合，现在的规模可以容纳**100**多万个符号。每个符号的编码都不一样，比如，**U+0639**表示阿拉伯字母**Ain**，**U+0041**表示英语的大写字母**A**，**U+4E25**表示汉字“严”。

```
>>> hex( ord( '严') )
'0x4e25'
```

8.9 Unicode的问题

需要注意的是，Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

比如，汉字“严”的Unicode是十六进制数4E25，也就是说这个符号的表示至少需要2个字节。表示其他更大的符号，可能需要3个字节或者4个字节，甚至更多。

8.9 Unicode的问题

- （1）如何才能区别**Unicode**和**ASCII**？
- （2）计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号呢？
- （3）英文字母只用一个字节表示就够了，如果**Unicode**统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必然有二到三个字节是**0**，这对于存储来说是极大的浪费，文本文件的大小会因此大出二三倍，这是无法接受的。
- 造成的结果是：

1）出现了**Unicode**的多种存储方式，也就是说有许多种不同的二进制格式，可以用来表示**Unicode** 。

2）**Unicode**在很长一段时间内无法推广，直到互联网的出现。

8.10 UTF-8

UTF-8就是在互联网上使用最广的一种Unicode的实现方式。

注意：UTF-8是Unicode的实现方式之一。

UTF-8最大的一个特点，就是它是一种变长的编码方式。

- 使用1~6个字节表示一个符号，根据不同的符号而变化字节长度；
- 如果Unicode字符由2个字节表示（UCS-2），则编码成UTF-8很可能需要3个字节。而如果Unicode字符由4个字节表示（UCS-4），则编码成UTF-8可能需要6个字节。

8.11 UTF-8编码规则

UTF-8的编码规则很简单，只有两条：

1）对于单字节的符号，字节的第一位设为0，后面7位为这个符号的Unicode码。因此对于英语字母，UTF-8编码和ASCII码是相同的。

2）对于n字节的符号（n>1），第一个字节的前n位都设为1，第n+1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的Unicode码。

Unicode/UCS-4	bit数	UTF-8	byte数	备注
0000 ~ 007F	0~7	0XXX XXXX	1	
0080 ~ 07FF	8~11	110X XXXX 10XX XXXX	2	
0800 ~ FFFF	12~16	1110XXXX 10XX XXXX 10XX XXXX	3	基本定义范围：0~FFFF
1 0000 ~ 1F FFFF	17~21	1111 0XXX 10XX XXXX 10XX XXXX 10XX XXXX	4	Unicode6.1定义范围：0~10 FFFF
20 0000 ~ 3FF FFFF	22~26	1111 10XX 10XX XXXX 10XX XXXX 10XX XXXX 10XX XXXX	5	说明：此非unicode编码范围，属于UCS-4 编码 早期的规范UTF-8可以到达6字节序列，可以覆盖到31位元（通用字符集原来的极限）。尽管如此，2003年11月UTF-8 被 RFC 3629 重新规范，只能使用原来Unicode定义的区域， U+0000到U+10FFFF。根据规范，这些字节值将无法出现在合法 UTF-8序列中
400 0000 ~ 7FFF FFFF	27~31	1111 110X 10XX XXXX 10XX XXXX 10XX XXXX 10XX XXXX 10XX XXXX	6	

8.12 BOM

BOM —Byte Order Mark，中文名译作“字节顺序标记”。

UCS 规范建议在**传输字节流前**，先传输**BOM**。

（1）如果头为 **FEFF**，就表明这个字节流是 **Big-Endian** 的；

（2）如果头为 **FFFE**，就表明这个字节流是 **Little- Endian** 的；

（3）如果头为**EF BB BF**，就表明这个字节流是 **UTF-8** 编码的.

注：对于**UTF-8**来说，**BOM**标记的有无并不是必须的，是可选的，因为**UTF-8**字节没有顺序。

Windows 就是使用 BOM 来标记文本文件的编码方式的。

byte

由于Python的字符串类型是str，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把str变为以字节为单位的bytes。

Python对bytes类型的数据用带b前缀的引号表示：

```
x = b'ABC'
type(x)
```

```
>>> x = b'ABC'
>>>
>>> type( x )
<class 'bytes'>
```

- 以Unicode表示的str通过encode()方法可以编码为指定的bytes，例如：
  - >>> '中文'.encode('utf-8') #默认
  - >>> '中文'.encode('gbk')

```
>>> '中文'.encode('utf-8')
b'\xe4\xb8\xad\xe6\x96\x87'
>>> '中文'.encode('gbk')
b'\xd6\xd0\xce\xc4'
```

- 用Python3写文件的时候如果没有指定编码模式，在中文Windows系统中其默认使用的是encoding =‘cp936’。
- 微软的CP936通常被视为等同GBK。故在写文件的时候记得要指定编码格式：

```
>>> import sys, locale
>>> print(sys.getdefaultencoding())
>>> print(locale.getdefaultlocale())
```

打开记事本，输入‘123456’,然后按不同编码方式保存：

- ANSI
- Unicode
- Unicode big endian
- UTF-8

然后用一个二进制查看工具（如winhex, powershell命令等）打开查看：

Format-Hex .\1-u.txt

或

- bom = open(r'123-utf8-bom.txt','rb')
- d2 = bom.read()
- d2.hex()

- 思考

1）Python如何进行内存管理?

2）如何进行Unicode编码?