第四章字符编码

> 本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务器进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的水平更上一层楼。

一 理解字符编码理论储备知识

1. 字符编码说明

字符编码这个知识点其实只是涉及到一行代码，但是它非常重要，据不完全统计，现在软件30%的损失都是由于乱码问题所导致的，这个问题是最容易被大家所忽视的，因为使用的时候只是一行代码的问题，但是它的里面包含很多的知识，大部分人更加倾向于直接掌握结果，而不考虑它内部的知识，这就导致了一旦遇到字符编码的乱码问题，就会手足无措。你之前可能看过一些相关的介绍，正确与否我们先不做评论，在这篇文章中，我们会对字符编码进行全方位的介绍。字符编码的特点是理论非常多，而结论非常少，但是如果不知道理论，结论可能永远也无法理解，而且以后遇到字符编码问题就会不知所措。目前在网上几乎没有人能够清晰的把它说明白，因为字符编码的发展史涵盖了整个计算机发展的过程，我们不会直接拿出现成的结论或者猜测的结论，而是会从客观展示出来的打印结果来论证：我所讲的就是正确的。

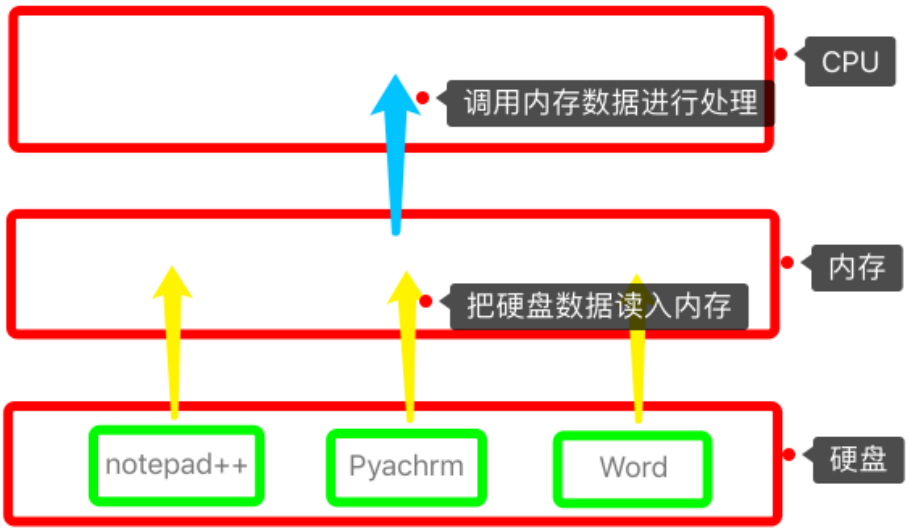
2. 计算机运行应用软件原理

在正式介绍字符编码之前，我们需要先了解计算机的运行与哪些核心部件相关：

> CPU：运行程序
> 内存：临时存储数据，一个软件要想运行，必先加载到内存
> 硬盘：软件的数据要想永久存储，一定要存入硬盘

如下图所示，一个应用软件的启动过程是：

1. 应用软件存放于硬盘上
2. 应用软件程序从硬盘读入到内存
3. CPU调用内存中与该应用软件相关的数据进行处理

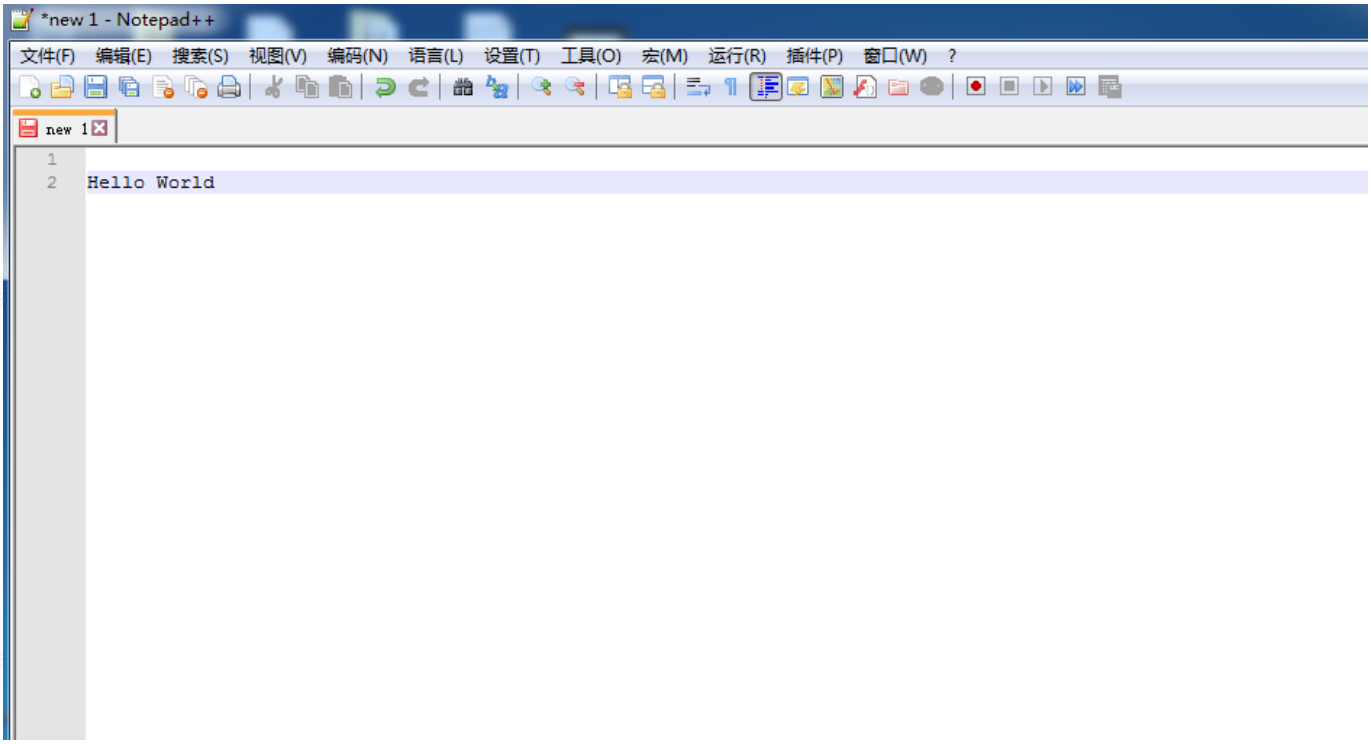


3. 保存文件的原理

接下里来我们再来看一下一个文本编辑器保存文件的过程（你可以下载安装一个notepad++编程软件）：

1. 如下图所示，当我启动notepad++程序之后，就是把这个程序读入到内存
2. 我在notepad++程序中输入“Hello World”之后，这个内容临时保存到了内存中
3. 你现在看到的屏幕上的内容是，notepad++程序在你写入的同时返回的结果

存在内存中的特点是：只要一断电，数据就会丢失，所以，我们要想永久的保存数据，需要在notepad程序上执行一个操作，把数据永久保存到硬盘（notepad程序会给你附加一个自动保存到硬盘的功能，这是为了提升用户的使用体验，早期的程序没有这个功能）。



4. 执行Python程序的原理

接下来我们再来看一下Python解释器执行Python程序的原理，也是分为三个阶段（以Python3解释器和test.py文件为例）：

1. 先启动Python解释器（把Python3解释器这个应用软件读入内存）
2. 把test.py文件读入内存
3. Python3解释器识别Python语法，解释执行test.py程序

注意：我们写的Python代码如果没有解释器解释执行和写一个普通的文件没有任何区别，这也就是说，你可以使用Pycharm写Python代码，也可以使用WPS写Python代码，在编写Python程序时候没有语法的概念（检测语法Pycharm给你附加的功能，但其实本质也是一个文本编辑器），编写的结果和编写一个普通文件是没有任何区别的， 只有在第三阶段执行Python程序的时候才会监测语法。

二 字符编码介绍

1. 字符编码初次登场

计算机是基于电工作的， 高电平用数字1表示，低电平用数字0表示，计算机也只能识别010101这种东西，这就是二进制，计算机的工作原理就是基于二进制工作的。 我们平时在使用计算机的时候并不是使用

二进制控制的，使用的都是人类的字符（中国人使用汉语，美国人使用英语），但是这些人类的字符计算机是看不懂的，要想让计算机能够看得懂，必先经历一个过程：

```
人类的字符====>翻译====>二进制数字
```

我们在notepad程序中写了一个“你好”，就把这个内容写入到了内存中，计算机要想识别必先经历一个翻译的过程，这个翻译肯定不能随便翻译的，因为在取这个数据的时候仍需按照二进制数字与人类的字符一一对应的取出来，所以必须要遵循一个标准，这个标准就是字符编码表。

我们在notepad程序中写了一个“Hello World”，在计算机内部已经事先存好了这张表，每一个字母（包含大写字母和小写字母）和空格回车标点符号这些东西都对应一个数字，然后再把这些数字转化成二进制，这样一个字符就会对应一组二进制数字，也就完成了写入的过程，当打印的时候再反过来，一组二进制数字对应一个字符。所以，我们就清楚了内存上应该有这样一张字符编码表，早期的时候硬盘上保存的也是二进制，所以硬盘上无需有字符编码表。

2. 字符编码发展史

计算机起源于美国，美国人说英语，美国人当时设计的时候根本就没考虑过中国人有一天也能用的起计算机，那么当时美国人设计的时候就只需要考虑计算能识别英文符号就可以了，所以当时的字符编码表就只是英文字符与数字的对应关系，所有这些加起来一共120多种就够了，那么也就应该有120多个不同的数字来表示这些字符。计算机的数字是二进制的，要想用010101这种东西表示出120多个数字应该用几位二进制数？

> 一位二进制数：只能表示0或者1两个数字，除去0之外，只能表示1个数字
> 两位二进制数：00，01，10，11能表示3个数字
> 三位二进制数：000，001，010，011，100，101，110，111能表示7个数字
> 四位二进制数：能表示2**4-1 = 15个数字

最开始设计的时候设计了八位二进制数，最小的是0000 0000，最大的是1111 1111，它能够表示2 8 - 1 = 255个数字（不包含0），但是最早期只是使用了后面的七位二进制位能够表示1～127的数字就够了，留下一位二进制位为了以后的扩展留一些余地。一个二进制位称为一个比特位，8个比特位称为一个字节：8bit = 1bytes，美国人用8个比特位来表示一个英文字符，所以一个英文字符占一个字节。这个字符编码表就是ASCII表，这也是最早的字符编码表，如下图所示。

ASCII表

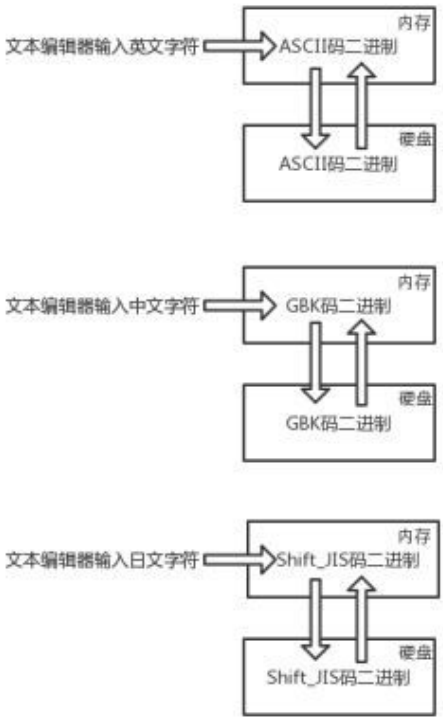
(American Standard Code for Information Interchange 美国标准信息交换代码)

高四位 / 低四位		ASCII控制字符 0000 0						ASCII控制字符 0001 1						ASCII打印字符 0010 2		ASCII打印字符 0011 3		ASCII打印字符 0100 4		ASCII打印字符 0101 5		ASCII打印字符 0110 6		ASCII打印字符 0111 7	
		十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	Ctrl
0000	0	0		^@	NUL	\0	空字符	16	▶	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A	SOH		标题开始	17	◀	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☻	^B	STX		正文开始	18	↕	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C	ETX		正文结束	19	‼	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F	ACK		肯定应答	22	▬	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	•	^G	BEL	\a	响铃	23	↨	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	8	◘	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◙	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K	VT	\v	纵向制表	27	←	^[	ESC	\e	溢出	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	♀	^L	FF	\f	换页	28	∟	^\	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	⌂	^Backspace 代码：DEL

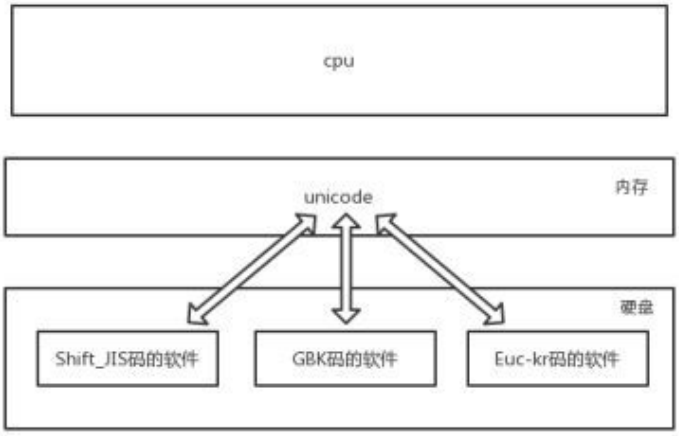
后来中国人也开始使用计算机了，ASCII只有英文字符的对应关系，而且8个比特位最多能表示200多个字符的对印关系，如果有一个人说他认识200多个汉字，估计这个人小学还未毕业。在这样的场景下，中国人也定制出了自己的字符编码表，叫做GBK（gb2312），为了表示出更多的数字，GBK编码表用16个比特位来表示一个中文字符，那么他所能表示的汉字个数就是2 16 – 1 = 65535个，这个基本上也就涵盖了所有我们常用的汉字。所以GBK使用两个字节来表示一个中文字符，但是GBK来表示一个英文字符还是用一个字节（为了和ASCII统一）。

> 不只是中国人能用电脑，日本人和韩国人也能用电脑，日本人规定了自己的Shift_JIS编码，韩国人规定了自己的Euc-kr编码（另外，韩国人说，计算机是他们发明的，要求世界统一用韩国编码，但世界人民没有搭理他们）。

到了这个阶段，如果每个国家的东西都是自己国家的人看，这当然没有问题，但是显然我们有这样的需求，这就出现了一个问题，你的硬盘上可能有日本人和韩国人编码的软件，但是内存中的字符编码表只有GBK，那么就会出现乱码，你不是秦始皇，自然做不出他那么伟大是事情，所以必须找到一种能够兼容万国语言的字符编码（其实6万多就足够表示了，各国语言虽然不同，但是所用的符号都非常类似），这个字符编码就是Unicode，它使用16个比特位也就是两个字节来表示一个字符。这也就意味着任何国家的数据到了内存中都是Unicode编码，这样内存中就不会出现乱码的问题了。除此之外还有一个很重要的问题，之前用各国编码写的保存在硬盘上的文件不能废弃，这是历史遗留问题，所以Unicode还必须要有一个非常重要的功能：把各国编码的文件转化成Unicode，如下图所以。这样有了Unicode之后，一方面可以兼容万国语言，另一方面老的软件也可以在不同国家的机器上运行。



很多地方或老的系统、应用软件仍会采用各种各样的编码，这是历史遗留问题。需要强调：软件是存放于硬盘的，而运行软件是要将软件加载到内存的，面对硬盘中存放的各种编码的软件，想让我们的计算机能够将它们全都正常运行而不出现乱码，内存中必须有一种兼容万国的编码，并且该编码需要与其他编码有相对应的映射/转换关系，这就是unicode



作者：马一特

至此，Unicode字符编码已经解决的大部分的乱码问题，但是还存在一个可以优化的空间，内存中用Unicode编码，硬盘上面有各国编码的软件，以后写程序肯定是趋向于全部使用Unicode编码，但是如果一篇文档当部分都是英文，写一个同样的内容原来的ASCII一个英文字符占用一个字节，而Unicode一个英文字符就会占用两个字节，这样的话就会增加硬盘的占用并且加大了IO操作的时间（IO操作暂且先理解为读写操作，在第五阶段最后一章单独讲解IO操作）。本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的`UTF-8（可变长，全称Unicode Transformation Format）`编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间。所以，现在硬盘上的字符编码一般是“UTF-8”，内存中用的字符编码一般是Unicode，以后肯定会趋向于内存和硬盘的存储都是用UTF-8这种字符编码，但是现阶段都是被逼的，Unicode编码还要负责转换历史遗留问题的编码。

三 乱码问题的产生与解决

1. 乱码问题的成因

接下来就是我们做实验的过程了，内存中的编码都是Unicode，如果忽略硬盘，在内存中随便写什么编码都不会出现乱码，但是因为硬盘的存在就会出现由内存向硬盘保存的时候你要指定一个字符编码，比如说是GBK，这时就是由Unicode转化成GBK，当把这个硬盘文件重新在内存读取的时候你也要告诉计算机按照GBK编码来读取，它才会对应的把数据由GBK编码反解成Unicode编码写入到内存。如果你在这时告诉

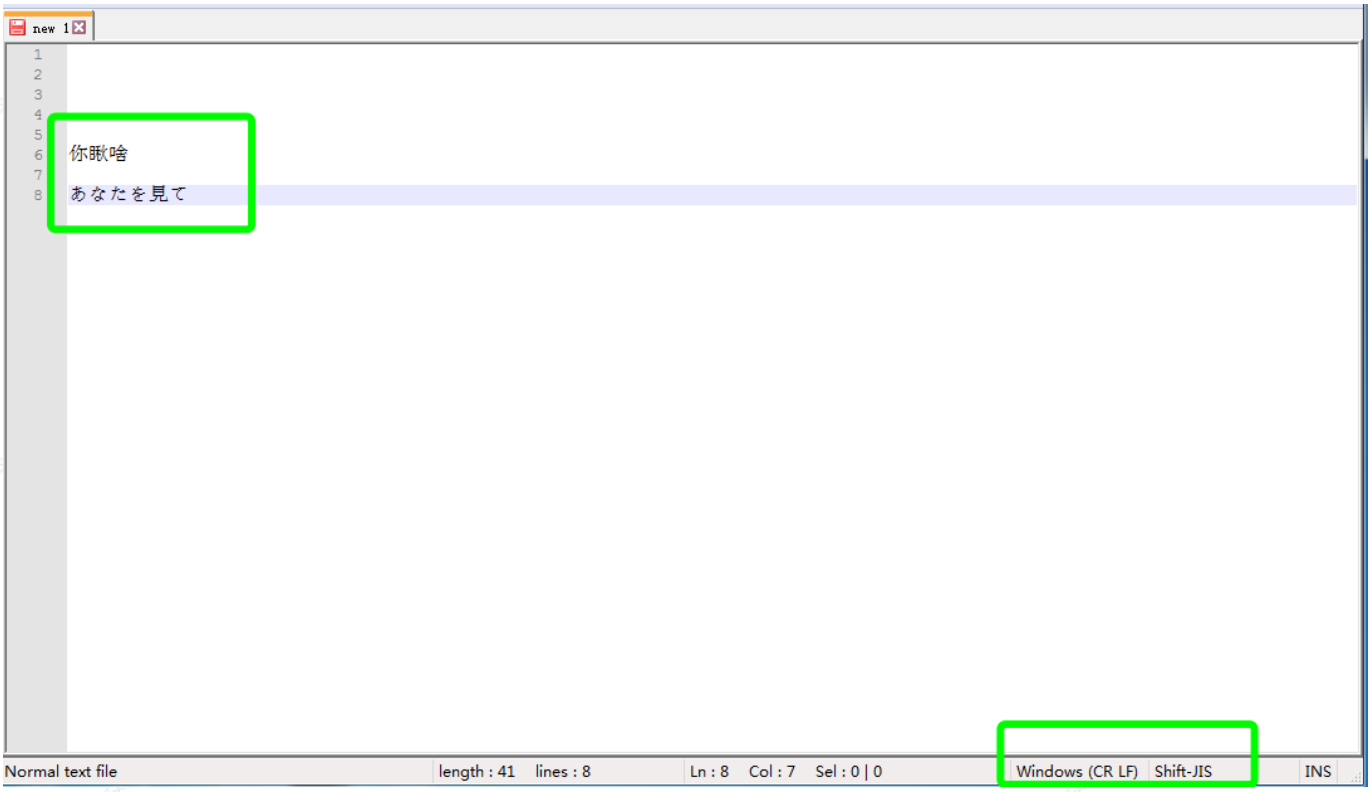
你计算机用ASCII的标准来反解数据，那么就无法反解出原来保存的数据内容，计算机懵圈了，呈现给你的就是它懵圈后的结果。

2. 保证不乱码的方案

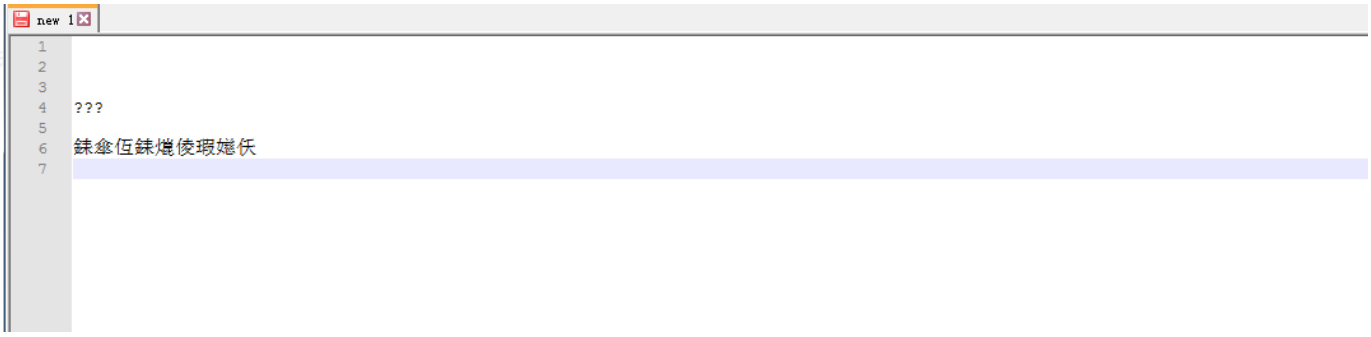
保证不出现乱码问题其实结论就只有一个：文件用什么编码保存的，就用什么编码读取，注意：我们能控制的只是文件由内存保存到硬盘的编码。

3. 人为制造乱码

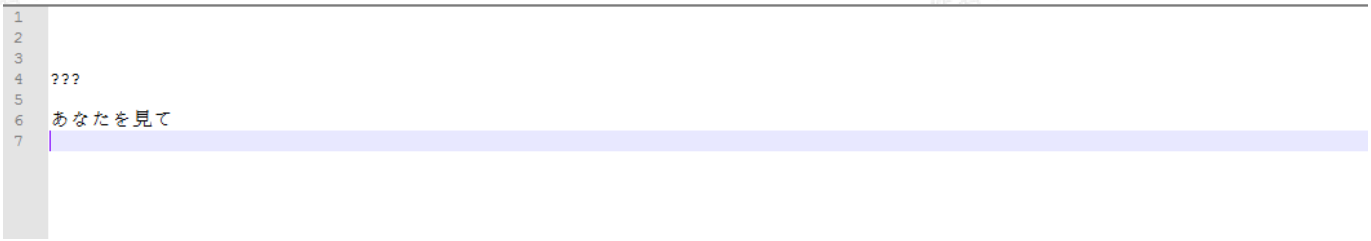
接下来我们会在notepad++程序上演示人为制造乱码的过程，首先把程序保存文件的编码改成日文的编码（见参考https://jingyan.baidu.com/article/154b46314b68da28ca8f412b.html），然在在程序里面先后写入中文的“你瞅啥”和日文的“あなたを見て”（这是日文的“瞅你咋地”），接下里来保存。注意：在保存过程中，其实计算机已经不能识别中文的“你瞅啥”了，但是它不能报错呀，他一定要硬存，所以这个保存的过程并没有什么问题，你现在看到的结果还是在内存中的。



保存之后，我们关掉再重新读取，先使用使用ASCII试一下会看到什么结果。



全都是乱码，那么我们再改成日文的编码试试。



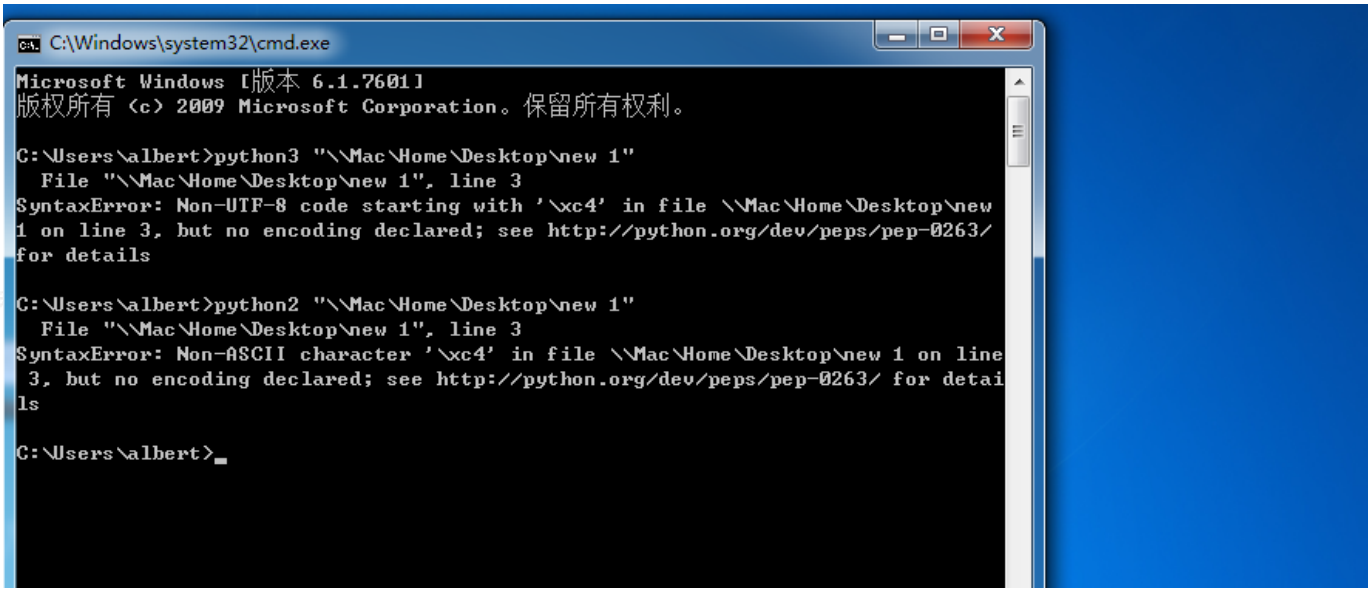
日文“瞅你咋地”已经读取出来了，但是我们存的汉字就读不出来了，这时无论我们改成任何编码汉字都是读不出来的，因为保存的时候计算机就已经无法识别了，他并没有完成有效的数据保存。所以，以后写程序写文件都应该用UTF-8编码来写，这样就不会发生乱码了。

四 Python中的字符编码问题

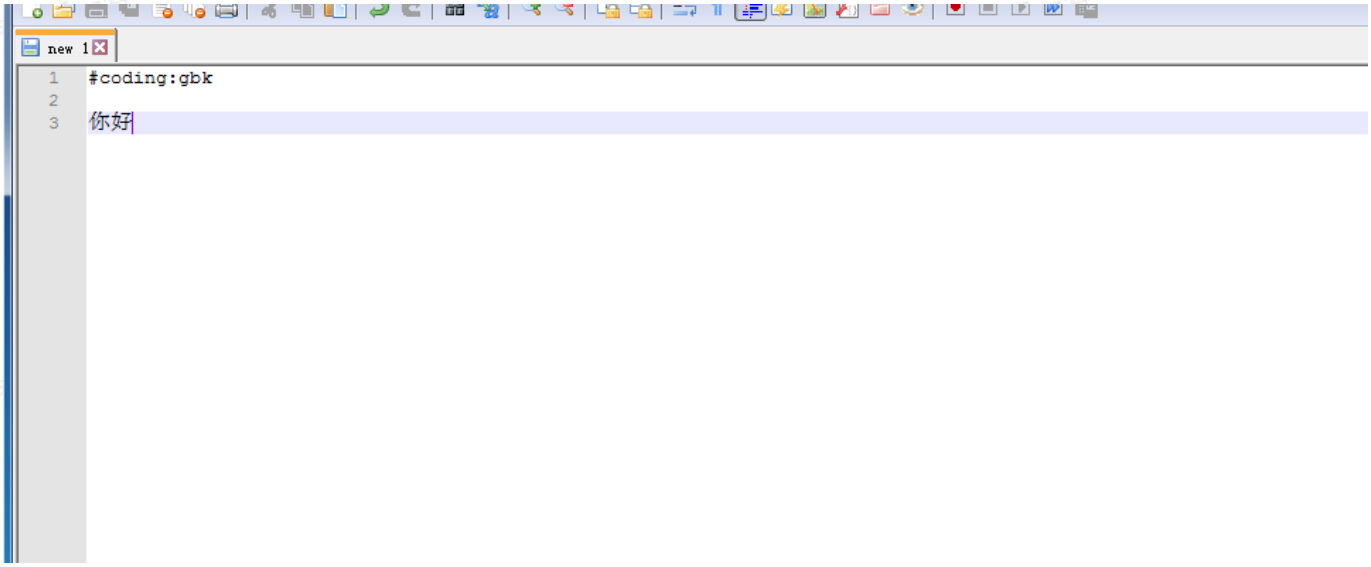
1. 解释器读取文件时字符编码异同

我们现在用GBK编码在文件中写一个“你好”，然后用Python解释器来解释执行（不用Pycharm），你会发现在Python2 和Python3 中分别提示以下错误信息，如下图所示，这就说明如果你不指定字符编码：

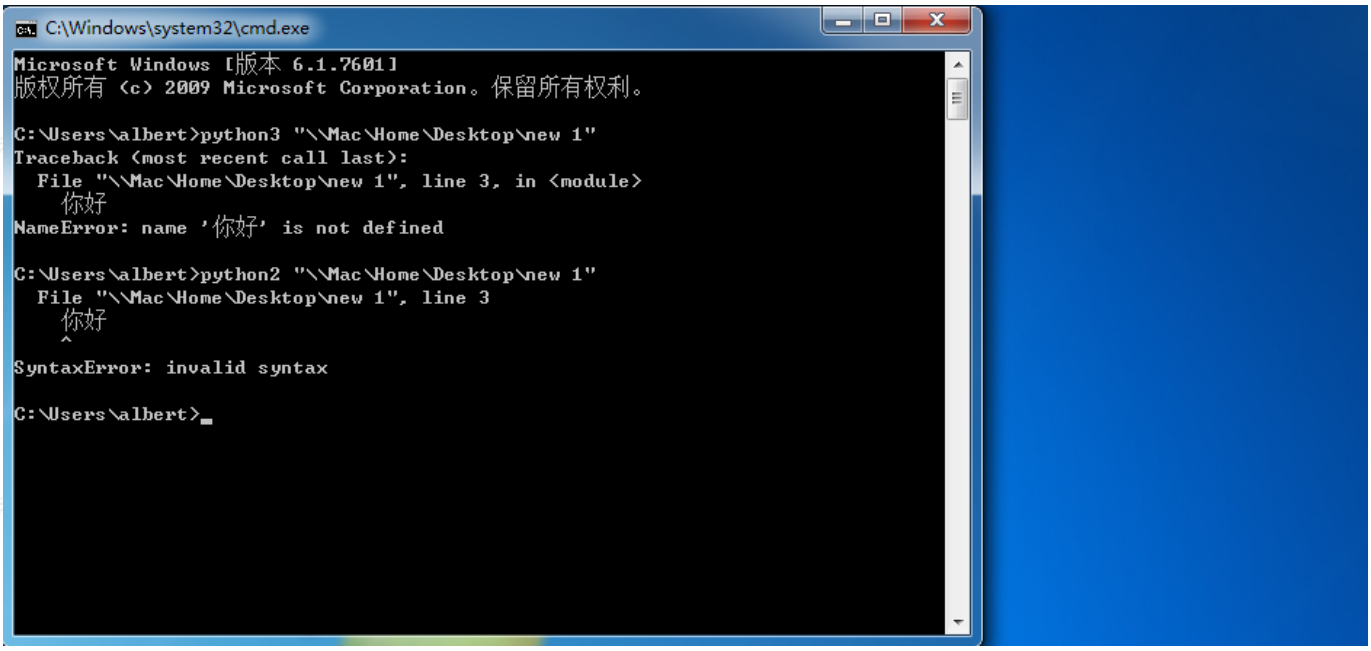
> Python3解释器默认使用UTF-8编码来读
> Python2解释器默认使用ASCII来读



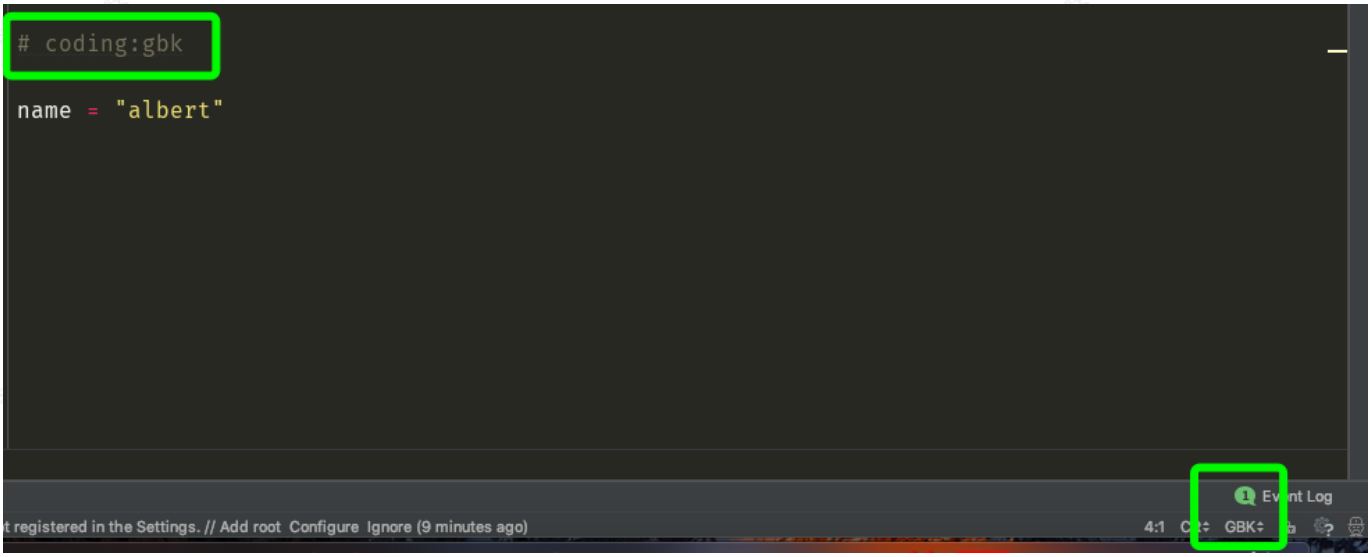
为了解决这个问题，我们需要在写Python文件开头就指定好字符编码，如下图所示。



再次调用解释器解释执行的时候就不会发生字符编码的错误了。



文件头的作用就是告诉Python解释器用指定的字符编码去读取文件内容。
在Pycharm中，当我指定好了读取文件的字符编码，它会自动改变保存写入到硬盘的字符编码，如下图所示。



很多人都知道在写代码的时候，文件最上方写一行指定字符编码的文件头：

```
# coding:utf-8
```

但是可能并不知道原因，在Pycharm中，只要你这样写了就不会发生任何的字符编码问题，但是如果换一个其他的IDE工具，这一行代码代表的是读文件所使用的字符编码，如果这个IDE工具默认保存写入到硬盘的字符编码是“GBK”，而且他不会随着你的文件头而改变，那么它保存的时候就是以GBK保存的，而读的时候指定了UTF-8的字符编码，计算机就会读不出来，换了人来解决这个问题，如果不清楚它的成因，也可能会很懵逼。好就好在大部分的文本编辑器默认用的字符编码都是UTF-8，一般人们写文件头一般也是指定UTF-8为读取的标准。所以，大部分不懂字符编码的人出现错误可能性也比较低。

2.解释器执行文件时字符编码异同

Python中有一种数据类型会涉及到字符的概念，这种数据类型就是字符串。我们在写程序时候如果不指定字符编码，保存的时候默认就是UTF-8的字符编码，而Python3解释器默认也是使用UTF-8来读取的，所以，它在Python3中运行是没有任何问题的，但是Python2中就不行了，所以为了执行Python程序的前两个阶段不出现问题，我们统一的都会加上文件头指定UTF-8字符编码，但是到了第三个阶段，就会开始识别语法了，我们在Pycharm中写入如下代码内容，计算机在执行的时候会先把这些代码以Unicode编码读入内存，然后Python解释器检测到代码中要保存一个“上”这个字符串变量，解释器就会调用计算机申请内存空间，把“上”这个字符存入到计算机内存中，保存的方式自然还是二进制，但是应该用什么字符编码保存呢?

```
# coding:gbk

x = '上'
```

涉及到Python中的数据类型，这是龟叔写的解释器，自然是他说了算。

> Python2中的字符串分为两种类型：1 str，2 unicode

在Python2中如果指定字符编码，那么字符串的保存会按照你指定的字符编码来完成，正如上面的代码，我们使用GBK编码来指定，那么在内存中就会用两个字节来保存一个汉字，接下来我们就来验证一下，如果我们直接打印的这个x的话，从理论上讲，x对应的就是一串二进制数，而打印的时候Python2解释器帮你做了一个转化，目的是为了让你能够更加直观的看到它的打印结果，但是会出现乱码（这个乱码问题我们最后再讲），我们可以使用以下代码这种形式来进行打印，就可以看到还没有转化之前的结果（龟叔还没来得及在Python2中的做转化，就写了Python3）。

```
# coding:gbk

# 解释器已经切换到Python2

x = "上"
# print(x)  # Python2加括号也能打印
print([x, ])
"""
输出：
['\xc9\xcf']
"""
```

我们从输出结果上来看，“\x”代表的是十六进制，“c9”和“cf”分别代表两个十六进制位，一个十六进制位对应四个二进制位，那么打印结果'\xc9\xcf'就是16个二进制位，也就是16bit=2bytes，这也就证明了GBK编码保存中文用两个字节。接下来用同样的方式我们再来验证一下UTF-8保存常见汉字用3个字节，代码如下。

```
# coding:utf-8

# 解释器已经切换到Python2
```

```
x = "上"
# print(x)  # Python2加括号也能打印
print([x, ])
"""
输出：
['\xe4\xb8\x8a']
"""
```

3. 字符编码的转换

通过前面的讲解我们已经清楚了unicode编码可以转换成GBK或者UTF-8，相反也可以转换，他们之间的转换过程如下：

> unicode =>编码encode=>GBK/UTF-8
> GBK/UTF-8 =>解码decode=>unicode

所以，必然会有以下代码的执行：

```
# coding:utf-8

# 解释器已经切换到Python2

x = "上"
# print(x.decode('gbk'))  # 使用utf-8编码，使用gbk无法解码
print(x.decode('utf-8'))  # 使用utf-8编码，使用utf-8解码
print([x.decode('utf-8'), ])  # 在列表打印出龟叔没有转换之前的unicode编码
"""
输出：
上
[u'\u4e0a']  # \u 代表unicode
"""
```

我们来比较一下unicode字符编码表中“上”这个字符，下图是unicode字符编码表中部分内容，左侧4开头的如“4E05”就是unicode的十六进制表示形式，很明显，通过对比，我们可以找到字符“上”对应unicode的十六进制就是“4e0a”。

在unicode中你可以看到有6个像“上”一样的字符，第一个指的是简体中文，第二个指的是香港繁体字，第三个是台湾繁体字，第四个是日本字，这些字符都是看起来类似的，所以在unicode中统一都用“4e0a”来存储，但是每一个不同编码的字符为了区分又添加了类似“494F”这样的标识，unicode编码也是通过这种方式来完成与万国编码的转换过程。

4. Python2两种字符串类型的区别

```
# coding:gbk

# 解释器已经切换到Python2

x = "上"
print(type(x))  # str类型
print([x, ])  # 以gbk编码保存
print([x.decode('gbk'), ])  # 解码后就是unicode，与下面保存的字符编码一致

y = u'上'  # 定义字符串的时候前面加"u"
print(type(y))  # unicode类型
print([y, ])  # 以unicode编码保存
```

5. 字符编码的保存与取值原理

我们以GBK编码为例，保存一个字符“上”，那么保存的结果应该是与上图unicode编码中对应的GBK编码“494F”相对应，但是，请看如下代码，打印结果并不对应，这又是为何呢?

```
# coding:gbk

# 解释器已经切换到Python2

x = "上"
print([x, ])  # 以gbk编码保存
"""
输出
['\xc9\xcf']
"""
```

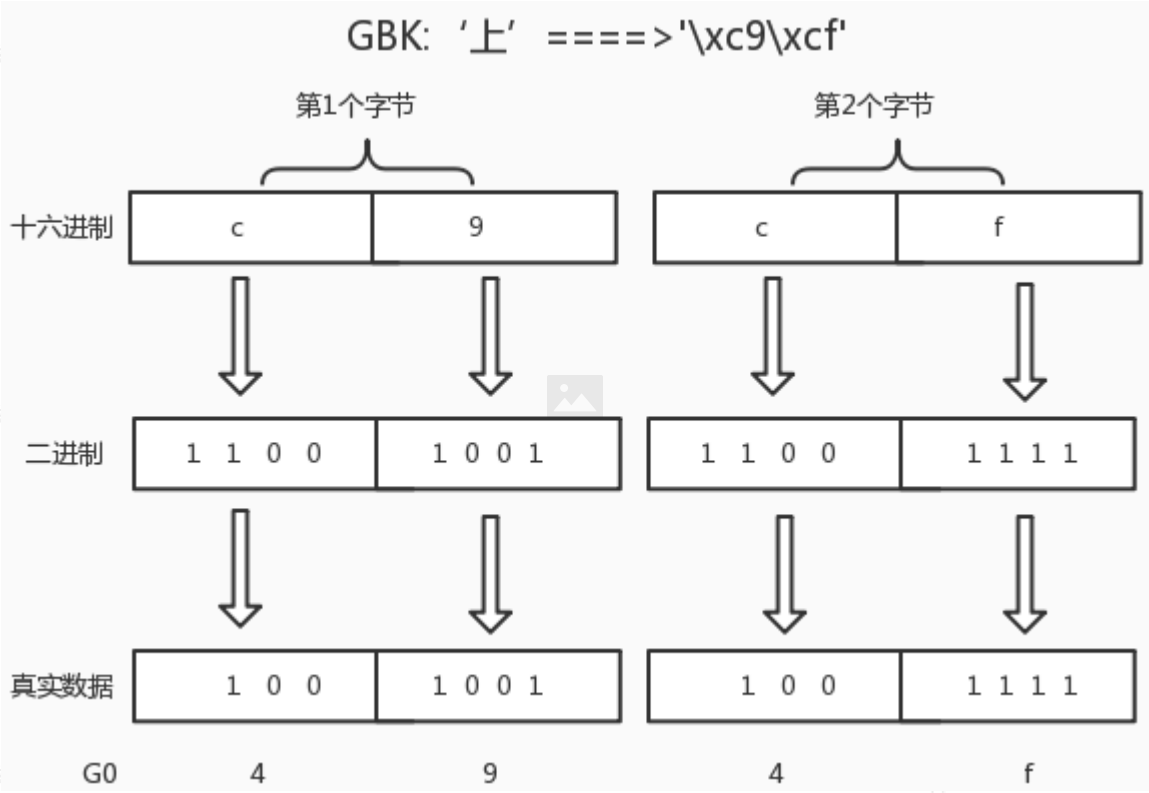
GBK可以保存中文或者英文字符，假如我们要保存一串字符“你a好”，应该是使用8bit+8bit+8bit+8bit+8bit一共40个比特位来保存。这样保存的时候没有问题，但是取值却成了问题，计算机并不知道从那一个比特位开始到哪一个比特位结束是第一个字符，所以这样连在一起无法取值（韩愈《师说》中写到：“句读之不知”也是一样的道理，这句话指的是不理解标点符号，那么自然不理解句子的开始和结束，也就无法读懂一句话）。计算机在取值的时候也一定要有一个明确的开始和结束，所以，其实计算机在保存的时候并不是8个比特位都用来表示字符的，这也是GBK表面上是保存65535个汉字其实并没有那么多，它只能保存3万多个。GBK能表示两种字符，分别是中文和英文，其实它的第一个字符是用来区分中文和英文的，所以要保存一个“你a好”字符串，应该是如下保存方式：

> (1+7bit)+(1+7bit)+(1+7bit)+(1+7bit)+(1+7bit)

假如第一个比特位是1代表中文字符，那么接下来他就会先读第一个字节第一个比特位代表中文，接下来把这个字节读完了之后，需要再一次读取第二个字节的第一个比特位，刚好第一个比特位也是1，那么他就把后面的7个比特位读完了就能准确地读出这个中文字符，同理，如果一个字符的第一个比特位是0，那么它代表一个英文字符，那么它只需要把这个字节的内容读完就能准确地读出这个英文字符。

> 计算机原本都是机器，它之所以智能都是源自于人类的智慧。

如果你使用GBK编码保存而是UTF-8编码去读取，由于两种编码每个字符所占用的字节数不同，那么关于每个字符的第一个字节的第一个比特位的标识也就不一致，就可能会产生第一个字符没有读完而后面的比特位标识与前面刚读过的字符标识不一致，那么每个字符都不能准确的读出来，这就会产生乱码。
既然已经清楚了它的存取原理那么现在我们再来看一下GBK编码保存一个中文字符“上”的过程，如下图所示。



这个过程主要分为四个阶段：

1. 先“上”这个字符转化十六进制分别是“c”，“9”，“c”，“f”
2. 把“c”，“9”，“c”，“f”分别转化成二进制对应的01010这些东西
3. 取出每个字节的第一个二进制位
4. 把剩余的每个十六进制位对应的多个二进制位再次转化成十六进制

所以你看到的结果和unicode字符编码表无法对应是由于计算机的存取原理所导致的。

6. 字符编码总结

最后再回到第二小节直接打印x会出现乱码的问题，龟叔在转化的时候按照你在内存保存的时候的字符编码GBK来转化，但是Pycharm默认使用的打印到屏幕的字符编码是UTF-8（这个可以改，但是不建议你修改），所以直接在Pycharm打印x你会看到乱码，如果是Windows系统的用户，可以在Windows终端以Python2执行代码，他就不会出现乱码，因为Windows平台默认打印到终端的编码也是GBK，如果是MacOS系统的用户，在终端以Python2执行代码还是会出现乱码，因为MacOS系统默认打印到到终端的编码是和Pycharm一样的UTF-8。

所以很多写Python2程序的人都会在str前面加一个“u”（可能他们自己也不知道这是为什么），这就是希望能够把所有的字符串按照unicode字符编码来保存，这样可以和任意编码转换。 有的人可能会有疑问，Python为什么要来两种字符串类型，这不是给使用者徒增麻烦吗?

你能想到的龟叔自然也能想到，这里有一点关于时间先后的问题。

> Python语言写于1989年，1991年Python2才正式诞生，unicode是1990年开始研发，1994年才真正诞生，UTF-8的诞生就更加的晚了，所以最开始的Python解释器一定是使用ASCII编码，那时候还没有unicode字符编码。

我们花费了大量的篇幅讲解了字符编码，这里面有一定的理论知识，但需要你记住的结论非常少，其他的深入的东西没有人会问你（详细知道的也没几个人），在Python3中所有的字符串都是用unicode编码来保存（不需要前面加“u”），字符串的数据类型也只有一个，就是str，只要是用unicode来保存的，那么所有的字符串在任何情况下都不会出现乱码，在Python3中代码示例如下：

```
# coding:gbk

# 解释器已经切换到Python3

x = "上"
print(x)
# unicode===>编码encode===>gbk
code_gbk = x.encode('gbk')
code_utf8 = x.encode('utf-8')

print(code_gbk)
print(code_utf8)
print(type(code_gbk))
print(type(code_utf8))

print(code_gbk.decode('gbk'))
print(code_utf8.decode('utf-8'))

"""
输出：
上
b'\xc9\xcf'
```

```
b'\xe4\xb8\x8a'
<class 'bytes'>
<class 'bytes'>
上
上
"""
```

从Python3中打印x.encode('gbk') 的结果中你可以看到是：‘\xc9\xcf’，这正是Python2中的str类型的值,而在Python3是bytes类型，在Python2中则是str类型，他们数据类型虽然不一致，但是存储的结果确是一致的，他们他们之间必然存在一种关联就是：Python2中的str类型就是Python3的bytes类型，我们可以查看Python2的str类型的源码，看到部分代码如下图所示：

```
2257
2258        def __sizeof__(self): # real signature unknown; restored from __doc__
2259            """ S.__sizeof__() -> size of S in memory, in bytes """
2260            pass
2261
2262        def __str__(self): # real signature unknown; restored from __doc__
2263            """ x.__str__() <==> str(x) """
2264            pass
2265
2266
2267    bytes = str
2268
2269
2270    class classmethod(object):
2271        """
2272        classmethod(function) -> method
2273
2274        Convert a function to be a class method.
2275
2276        A class method receives the class as implicit first argument,
```

编码之后的结果数据类型是bytes，看起来像是字节，其实是十六进制，计算机会自动把十六进制转化成二进制，你可以把bytes类型等同于二进制去看待，网络传输也是基于二进制来传输的，你在上网的过程中网速很慢的时候可以看到1b/s，这就是指的每秒传输一个字节。