Python数据操作

Python3有两种表示字符序列的类型：`bytes`和`str`。前者的实例包含原始的8位值（就是原始的字节，由于每个字节有8个二进制位，就是原始的8位值），后者的实例包含Unicode字符。

文本字符串

Unicode

Unicode编码是一种正在发展中的国际化规范，它可以包含世界上所有语言以及来自数学领域和其他领域的各种符号。

Unicode Code Charts页面（ http://www.unicode.org/charts/ ）包含了通往目前已定义的所有字符集的链接，且包含字符图示。最新的版本（9.0）定义了超过128 000个字符，每一种都有自己独特的**名字**和**标识符**。这些字符被分成了若干8比特的集合，称之为平面（plane）。前256个平面为基本多语言平面（basic multilingual plane）。

Python3中的Unicode字符串

Python3中的字符串是**由一系列Unicode码位（code point）所组成的不可变序列**。

通过某个字符的Unicode ID，可以直接获取其对应的字符。方法：

- 用`\u`及4个十六进制的数字从Unicode 256个基本多语言平面中指定某一特定字符。其中，前两个十六进制数字用于指定平面号（00到FF），后面两个数字用于指定该字符在平面中的位置索引。
- 使用更多的比特位来存储位于更高平面的字符。Python使用以`\U`开头的转义序列来处理，后面紧跟着8个十六进制的数字，其中最左一位需为0。
- 使用`\N{name}`来引用某一字符，其中**name**为该字符的标准名称，这对所有平面的字符都适用。

Python中的**unicodedata**模块提供了下面两个方向的准换函数：

- `lookup()` 接受不区分大小写的标准名称，返回一个Unicode字符；
- `name()` 接受一个Unicode字符，返回大写形式的名称。

```
def unicode_test(value):
    import unicodedata
    name = unicodedata.name(value)
    value2 = unicodedata.lookup(name)
    print('value="%s", name="%s", value2="%s"' % (value, name, value2))


unicode_test('A')
unicode_test('$')
unicode_test('¥')
unicode_test('\u00a2')
unicode_test('\u20ac')
print(ord('¥'))
print('\u00a5')
help(ord)
```

由于字体限制（没有任何一种字体涵盖了所有的Unicode字符），当缺失对应字符的图片时，会以占位符的形式显示。

```
unicode_test('\u2603')
```

```
# 存储 café (E WITH ACUTE, LATIN SMALL LETTER, 00E9)
import unicodedata
unicodedata.name('\u00e9')  # 'LATIN SMALL LETTER E WITH ACUTE'

unicodedata.lookup('E WITH ACUTE, LATIN SMALL LETTER')
# KeyError: "undefined character name 'E WITH ACUTE, LATIN SMALL LETTER'"
```

将Unicode字符索引名称转换为Python使用的Unicode名称，需将逗号舍去，并将逗号后面的内容移到最前面即可。

```
E WITH ACUTE, LATIN SMALL LETTER --> LATIN SMALL LETTER E WITH ACUTE
```

```
unicodedata.lookup('LATIN SMALL LETTER E WITH ACUTE')  # 'é'

place = 'caf\u00e9'
place2 = 'caf\N{LATIN SMALL LETTER E WITH ACUTE}'
place  # 'café'
place2  # 'café'
```

字符串函数`len`可计算字符串中Unicode字符的个数：

```
len('caf\u00e9')  # 4
len('\U0001f47b')  # 1
```

编码与解码

编码是将字符串转化为一系列字节的过程。维基百科字符编码的定义：

> 字符编码（英语：Character encoding）、字集码是把字符集中的字符编码为指定集合中某一对象（例如：比特模式、自然数序列、8位组或者电脉冲），以便文本在计算机中存储和通过通信网络的传递。

简单来说就是把人类通用的语言符号翻译成计算机通用的对象，而反向的翻译过程就是解码。Python中的字符串类型代表人类通用的语言符号，因此字符串类型有`encode()`方法；而字节类型代表计算机通用的对象（二进制数据），因此字节类型有`decode()`方法。

```
print(' '.encode())
```

既然说编码和解码都是翻译的过程，那么就需要一本字典将人类和计算机的语言一一对应起来，这本字典的名字叫做字符集，从最早的 **ASCII** 到现在最通用的 **Unicode**，它们的本质是一样的，只是两本字典的厚度不同而已。ASCII 只包含了26个基本拉丁字母、阿拉伯数字和英式标点符号一共128个字符，因此只需要（不占满）一个字节就可以存储，而 Unicode 涵盖的数据除了视觉上的字形、编码方法、标准的字符编码外，还包含了字符特性，如大小写字母，共可包含 1.1M 个字符，到现在只填充了其中的 128K 个位置。

字符集中字符所存储的位置（或者说对应的计算机通用的数字）称之为码位（code point），例如在 ASCII 中字符 '$' 的码位就是：

```
print(ord('$'))
```

ASCII 只需要一个字节就能存下所有码位，而 Unicode 则需要几个字节才能容纳，但是对于具体采用什么样的方案来实现 Unicode 的这种映射关系，也有很多不同的方案（或规则）。例如最常见（也是 Python 中默认的）UTF-8，还有 UTF-16、UTF-32 等。当然，在 ASCII 与 Unicode 之间还有很多其他的字符集与编码方案，例如中文编码的 GB2312、繁体字的 Big5 等等，这并不影响我们对编码与解码过程的理解。

这里简单了解一下UTF-8动态编码方案（变长编码方式），其会动态地为每一个Unicdoe字符分配1到4字节不等：

- 为ASCII字符分配1字节；
- 为拉丁语系（除西里尔语）的语言分配2字节；
- 为其他的位于基本多语言平面的字符分配3字节（包含中文在内的CJK）；
- 为剩下的字符集分配4字节。

UTF-8是Python、Linux和HTML的标准文本编码格式，其具有简单快速、字符覆盖面广、出错率低等特点。

Unicode*Error

明白了字符串与字节，编码与解码之后，让我们手动制造上面两个 Unicode*Error 试试，首先是编码错误：

```
def try_encode(s, encoding="utf-8"):
    try:
        print(s.encode(encoding))
    except UnicodeEncodeError as err:
        print(err)

s = "$" # UTF-8 String
try_encode(s) # 默认用 UTF-8 进行编码
try_encode(s, "ascii") # 尝试用 ASCII 进行编码

s = "雨" # UTF-8 String
try_encode(s) # 默认用 UTF-8 进行编码
try_encode(s, "ascii") # 尝试用 ASCII 进行编码
try_encode(s, "GB2312") # 尝试用 GB2312 进行编码
```

由于 UTF-8 对 ASCII 的兼容性，`"$"` 可以用 ASCII 进行编码；而 `"雨"` 则无法用 ASCII 进行编码，因为它已经超出了 ASCII 字符集的 128 个字符，所以引发了 `UnicodeEncodeError`；而 "雨" 在 GB2312 中的码位是 `b'\xd3\xea'`，与 UTF-8 不同，但是仍然可以正确编码。因此如果出现了 `UnicodeEncodeError` 说明你用错了字典，要翻译的字符没办法正确翻译成码位！

解码是将字节序列转化为Unicode字符串的过程。从外界文本源（文件、数据库、网站、网络API等）获取的所有的文本都是经过编码的字节串。需要知道其是使用何种方式进行编码的，才能正确解码以获得Unicode字符串。

```
place = 'caf\u00e9'
place_bytes = place.encode('utf-8')  # b'caf\xc3\xa9'
type(place_bytes)  # bytes
len(place_bytes)  # 5个字节，最后两个字节用于编码 é
```

再来看解码错误：

```
def try_decode(s, decoding="utf-8"):
    try:
        print(s.decode(decoding))
    except UnicodeDecodeError as err:
        print(err)


b = b'$' # Bytes
try_decode(b) # 默认用 UTF-8 进行解码
try_decode(b, "ascii") # 尝试用 ASCII 进行解码
try_decode(b, "GB2312") # 尝试用 GB2312 进行解码
b = b'\xd3\xea' # 上面例子中通过 GB2312 编码得到的 Bytes
try_decode(b) # 默认用 UTF-8 进行解码
try_decode(b, "ascii") # 尝试用 ASCII 进行解码
try_decode(b, "GB2312") # 尝试用 GB2312 进行解码
try_decode(b, "GBK") # 尝试用 GBK 进行解码
try_decode(b, "Big5") # 尝试用 Big5 进行解码

try_decode(b.decode("GB2312").encode())
```

一般后续出现的字符集都是对 ASCII 兼容的，可以认为 ASCII 是它们的一个子集，因此可以用 ASCII 进行解码（编码）的，一般也可以用其它方法；对于存在非子集关系的编码，强行解码有可能会导致错误或乱码！

实践中的策略

1. 记清楚编码与解码的方向；
2. 在 Python 中的操作尽量采用 UTF-8，输入或输出的时候再根据需求确定是否需要编码成二进制：

```
# cat utf8.txt
# 你好，世界！
# file utf8.txt
# utf8.txt: UTF-8 Unicode text

with open("utf8.txt", "rb") as f:
    content = f.read()
    print(content)
    print(content.decode())
with open("utf8.txt", "r") as f:
    print(f.read())


# cat gb2312.txt
# 你好，Unicode！
# file gb2312.txt
# gb2312.txt: ISO-8859 text

with open("gb2312.txt", "r") as f:
    try:
        print(f.read())
    except:
        print("Failed to decode file!")
with open("gb2312.txt", "rb") as f:
    print(f.read().decode("gb2312"))
```

编写Python程序的时候，一定要把编码和解码操作放在界面最外围来做。程序的核心部分应该使用Unicode字符类型（也就是Python3中的**str**），而且不要对字符编码做任何假设。这种办法既可以令程序接受多种类型的文本编码，又可以保证输出的文本信息只采用一种编码形式（最好用UTF-8）。

所以Python代码中经常会出现两种常见的使用情境：

1. 开发者需要原始8位值，这些8位值表示以UTF-8格式（或其他编码形式）来编码的字符；
2. 开发者需要操作没有特定编码形式的Unicode字符。

一般我们会编写两个辅助函数，以便在这两种情况之间进行转换。第一个函数接受**str**或**bytes**，并总是返回**str**：

```
def to_str(bytes_or_str, encoding='utf-8'):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode(encoding)
    else:
        value = bytes_or_str
    return value  # instance of str
```

第二个函数接受str和bytes，并总是返回bytes：

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value  # instance of bytes
```

二进制数据

- **字节序（endianness）**
- **整数符号位（sign bit）**

字节和字节数组

Python3使用两种方式以8比特序列存储小整数，每8比特可存储从0-255的值：

- 字节（bytes），不可变
- 字节数组（bytearray），可变

```
blist = [1, 2, 3, 255]
the_bytes = bytes(blist)
the_bytes  # b'\x01\x02\x03\xff'
```

```
the_byte_array = bytearray(blist)
the_byte_array  # bytearray(b'\x01\x02\x03\xff')
```

`bytes`（字节）类型值的表示：以`b`开头，接着是单引号`'`，后面跟着由十六进制数（例如`\x02`）或ASCII码组成的序列，最后是`'`。Python会将十六进制数或ASCII码转换成整数，如果该字节的值为有效ASCII码会显示ASCII字符。

```
b'\x01abc\xff'
```

```
print(the_byte_array)
the_byte_array[1] = 127
the_byte_array
```

打印bytes或bytearray数据时，Python会以`\xnn`的形式表示不可打印的字符，以ASCII字符的形式表示可打印的字符（以及一些转义字符）。

```
the_bytes = bytes(range(0, 256))
the_bytes
```

转换二进制数据

使用`struct`模块将二进制数据转换为Python中的数据结构。下面代码将一个Python元组列表写入一个二进制文件，并使用struct将每个元组编码为一个结构体。

```
from struct import Struct

def write_records(records, format, f):
    record_struct = Struct(format)
    for r in records:
        f.write(record_struct.pack(*r))


# example
records = [(1, 2.3, 4.5),
           (6, 7.8, 9.0),
           (12, 13.4, 56.7)]
with open('data.bin', 'wb') as f:
    write_records(records, '<idd', f)
```

读取上面代码中的文件并返回一个元组列表。

```
from struct import Struct

def read_records(format, f):
    record_struct = Struct(format)
    chunks = iter(lambda: f.read(record_struct.size), b'')
    return (record_struct.unpack(chunk) for chunk in chunks)


with open('data.bin', 'rb') as f:
    for rec in read_records('<idd', f):
        print(rec)
```

在函数`read_records`中，`iter()`被用来创建一个返回固定大小数据块的迭代器，这个迭代器会不断的调用一个用户提供的可调用对象，比如`lambda: f.read(record_struct.size)`，直到它返回一个特殊的值，如`b''`，这时候迭代停止。创建一个可迭代对象的一个原因是它能允许使用一个生成器推导来创建记录。

上面两个代码示例中通过创建一个Struct实例来声明一个新的结构体，结构体通常会使用一些结构码值。这些代码分别代表某个特定的二进制数据类型如32位整数，64位浮点数，32位浮点数等。第一个字符`<`指定了字节顺序。上面的例子中，它表示“低位在前”（小端方案）。更改这个字符为`>`表示高位在前（大端方案），或者是`!`表示网络字节顺序。更多的介绍请参考Python3文档。

类型标识符紧跟在字节序标识符的后面。任何标识符的前面都可添加数字用于指定需要匹配的数量。

标识符	描述	字节大小
x	跳过一个字节	1
b	有符号字节	1
B	无符号字节	1
h	有符号短整数	2
H	无符号短整数	2
i	有符号整数	4
I	无符号整数	4
l	有符号长整数	4
L	无符号长整数	4
Q	无符号long long型整数	8
f	单精度浮点数	4
d	双精度浮点数	8
p	数量和字符	1+数量
s	字符	数量

更多内容请参考Python官方文档。

产生的Struct实例有很多属性和方法用来操作相应类型的结构。`size`属性包含了结构的字节数，这在I/O操作时非常有用。`pack()`和`unpack()`方法被用来打包和解包数据。

```
from struct import Struct
record_struct = Struct('<idd')
record_struct.size  # 20

record_struct.pack(1, 2.0, 3.0)
record_struct.unpack(_)
```

`pack()`和`unpack`可以通过模块直接调用：

```
import struct
struct.pack('<i2d', 1, 2.0, 3.0)
```

格式化

如何用不同的格式化方法将变量插值（interpolate）到字符串中，即将变量的值嵌入到字符串中。

Python通常有两种格式化字符串的方式。

使用`%`进行格式化

形式为`string % data`。其中`string`是待插值的序列，常见的转换类型如下表所示：

类型	意义
%s	字符串
%d	十进制数
%x	十六进制数
%o	八进制数
%f	十进制数
%e	以科学计数法表示的浮点数
%g	十进制或科学计数法表示的浮点数
%%	文本值%本身

```
# 格式化整数
'%s' % 42  # '42'
'%d' % 42  # '42'
'%x' % 42  # '2a'
'%o' % 42  # '52'

# 格式化浮点数
'%s' % 7.03  # '7.03'
'%f' % 7.03  # '7.030000'
'%e' % 7.03  # '7.030000e+00'
'%g' % 7.03  # '7.03'

# 整数和字面值%
'%d%%' % 100  # '100%'

cat = 'Chester'
weight = 28

'Our cat %s weighs %s pounds' % (cat, weight)
# 'Our cat Chester weighs 28 pounds'
```

字符串中出现`%`的次数要与`%`之后所提供的数据项个数相同。如果需要插入多个数据，则需要将它们封装进一个元组。

另外，可以在`%`和指定类型的字母之间设定最大和最小宽度、排版以及填充字符等：

```
n = 42
f = 7.03
s = 'string cheese'

# 以默认宽度格式化
'%d %f %s' % (n, f, s)  # '42 7.030000 string cheese'

# 最小域宽10个字符，右对齐，左侧不够空格填充
'%10d %10f %10s' % (n, f, s)  # '        42   7.030000 string cheese'

# 将上例  对齐
'%-10d %-10f %-10s' % (n, f, s)  # '42         7.030000   string cheese'

# 将上例  对齐，最大字符宽度为4
'%10.4d %10.4f %10.4s' % (n, f, s)  # '      0042     7.0300       stri'

# 去掉最小域宽为10的限制
'%.4d %.4f %.4s' % (n, f, s)  # '0042 7.0300 stri'

# 将域宽、字符宽度等设定为参数
'%*.*d %*.*f %*.*s' % (10, 4, n, 10, 4, f, 10, 4, s)
# '      0042     7.0300       stri'
```

使用`{}`和`format`格式化

```
'{} {} {}'.format(n, f, s)  # '42 7.03 string cheese'

# 指定插入的顺序
'{2} {0} {1}'.format(f, s, n)  # '42 7.03 string cheese'

# 参数为字典或命名变量，格式串中标识符可以饮用这些名称
'{n} {f} {s}'.format(n=42, f=7.03, s='string cheese')  # '42 7.03 string cheese'

d = dict(n=42, f=7.03, s='string cheese')  # {'f': 7.03, 'n': 42, 's': 'string cheese'}
# {0} 代表整个字典，{1}代表字典后面的字符串'other'
'{0[n]} {0[f]} {0[s]} {1}'.format(d, 'other')  # '42 7.03 string cheese other'
```

将格式标识符放在`:`后：

```
'{0:d} {1:f} {2:s}'.format(n, f, s)  # '42 7.030000 string cheese'

'{n:d} {f:f} {s:s}'.format(n=42, f=7.03, s='string cheese')  # '42 7.030000 string cheese'

# 最小域宽设为10、右对齐
'{0:10d} {1:10f} {2:10s}'.format(n, f, s)  # '        42   7.030000 string cheese'
# 使用 > 设定右对齐
'{0:>10d} {1:>10f} {2:>10s}'.format(n, f, s)  # '        42   7.030000 string cheese'
#  对齐
'{0:<10d} {1:<10f} {2:<10s}'.format(n, f, s)  # '42         7.030000   string cheese'
# 居中
'{0:^10d} {1:^10f} {2:^10s}'.format(n, f, s)  '    42      7.030000  string cheese'
```

注意：在`{}`和`format`格式化时无法对整数设定精度。

```
'{0:>10.4d} {1:>10.4f} {2:>10.4s}'.format(n, f, s)
```

设定填充字符，将空格以外的字符放在`:`之后、其余任何排版字符和宽度标识符之前。

```
'{0:!^20s}'.format('BIG SALE')  # '!!!!!!BIG SALE!!!!!!'
```

Python3.6引入了新的 格式化字符串变量 语法，通过字符串的前缀`f`，实现类似于Swift / Scala等语言的字符串插值：

```
name = 'Frank'
f'My name is {name}'

date = datetime.datetime.now().date()
f'{date} was on a {date:%A}'

f'{"quoted string"}'

def foo():
    return 20

f'result={foo()}'
```