**内建模块笔记**

**datetime**

1. datetime : python处理日期和时间的标准库。
2. 字符串'%Y-%m-%d %H:%M:%S'规定了日期和时间部分的格式 str => datetime
3. '%a, %b %d %H:%M' datetime => str
4. Datetime 加减运算 : 导入timedelta 模块，在进行”+””-”运算符运算！
5. 时区转换的关键在于，拿到一个datetime时，要获知其正确的时区，然后强制设置时区，作为基准时间。
6. 利用带时区的datetime，通过astimezone()方法，可以转换到任意时区。
7. 注：不是必须从UTC+0:00时区转换到其他时区，任何带时区的datetime都可以正确转换，例如上述bj\_dt到tokyo\_dt的转换。

**Collections**

**namedtuple**

namedtuple是一个函数，它用来创建一个自定义的tuple对象，并且规定了tuple元素的个数，并可以用属性而不是索引来引用tuple的某个元素。

namedtuple('名称', [属性list])

**deque**

deque是为了实现高效插入和删除操作的双向列表，适用于队列和栈

deque除了实现list的append()和pop()外，还支持appendleft()和popleft()，这样就可以非常高效地往头部添加或删除元素

**defaultdict**

使用dict时，如果引用的Key不存在，就会抛出KeyError。如果希望key不存在时，返回一个默认值，就可以用defaultdict。注意默认值是调用函数返回的，而函数在创建defaultdict对象时传入(lambda:’N/A’).  
**OrderedDict**

使用dict时，Key是无序的。在对dict做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用OrderedDict

注意，OrderedDict的Key会按照插入的顺序排列，不是Key本身排序：

OrderedDict可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key 代码在 : <do_collections_OrderedDict_FIFO.py>

popitem(self, /, last=True)

Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

last=False是先入先出(FIFO)，last=True是后入先出(LIFO).

Py3中可以用super().\_\_init\_\_()代替super(xxx,self).\_\_init\_\_()

关于教程上的FIFO例子的具体思路可以参考url="https://blog.csdn.net/lilong117194/article/details/76252057"

**ChainMap**

什么时候使用ChainMap最合适？举个例子：应用程序往往都需要传入参数，参数可以通过命令行传入，可以通过环境变量传入，还可以有默认参数。我们可以用ChainMap实现参数的优先级查找，即先查命令行参数，如果没有传入，再查环境变量，如果没有，就使用默认参数。

关于教程中最后一个例子(同时传入命令行参数和环境变量，命令行参数的优先级较高:)未能在win10下成功运行，有建议是说源程序是在Linux下运行的，win10环境中无法识别’user’和’color’两个参数；还有人建议说直接在windows下添加环境变量，但自己试了下发现并不能成功，所以此小例子未能运行成功！源代码如下：

$ user=admin color=green python3 use\_chainmap.py -u bob

color=green

user=bob

**Counter**

Counter是一个简单的计数器，例如，统计字符出现的个数

Counter实际上也是dict的一个子类

用法：1.先创建一个Counter类，再用for循环调用

2.直接用Counter(‘xxxxxxxx’)

**Base64**

Base64是一种用64个字符来表示任意二进制数据的方法。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用\x00字节在末尾补足后，**再在编码的末尾加上1个或2个=号**，表示补了多少字节，解码的时候，会自动去掉。

由于标准的Base64编码后可能出现字符+和/，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符+和/分别变成-和\_。

由于=字符也可能出现在Base64编码中，但=用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把=去掉：

# 标准base64：

‘abcd’ -> ‘YWJjZA==’

# 自动去掉=:

‘abcd’ -> ‘YWJjZA’

去掉=后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上=把Base64字符串的长度变为4的倍数，就可以正常解码了。

关于作业题 是否用4 – len(s) % 4 以下为评论：

**为什么不是n=4-len(s)%4呢？**

这个问题问的是对的，但是你也没有发现一个问题，就是无论我是几个字节的，我想补全3的倍数，要么加1，要么加2 那么我加密出来的base64 要么加一个= 要么加两个= 所以base64的长度在对4求余数以后 要么是2（加2个=就是4的倍数） 要么是3 （加1个=就是4的倍数）， base64解码的时候 等号的数量多于2个 就能成功解码 所以他这么写，虽然思路有点问题，但是也能成功解码。

原理类似于：第一个刚好是3的整数倍，第二个差一个，所以补了=，第三个差2个所以补了2个==。(还不是很懂！)（3字节的二进制数据编码为4字节的文本数据）

（还有作业题的assert 为什么中间隔了个逗号！）

**Struct**

Python提供了一个struct模块来解决bytes和其他二进制数据类型的转换

struct的pack函数把任意数据类型变成bytes: struct.pack(‘>I’,s1)

pack的第一个参数是处理指令，'>I'的意思是：

>表示字节顺序是big-endian(大端)，也就是网络序，I表示4字节无符号整数。

unpack把bytes变成相应的数据类型： struct.unpack(‘>IH’,s2)

根据>IH的说明，后面的bytes依次变为I：4字节无符号整数和H：2字节无符号整数。

struct模块定义的数据类型可以参考Python官方文档：<https://docs.python.org/3/library/struct.html#format-characters>

Windows的位图文件（.bmp）是一种非常简单的文件格式

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节：'BM'表示Windows位图，'BA'表示OS/2位图；

一个4字节整数：表示位图大小；

一个4字节整数：保留位，始终为0；

一个4字节整数：实际图像的偏移量；

一个4字节整数：Header的字节数；

一个4字节整数：图像宽度；

一个4字节整数：图像高度；

一个2字节整数：始终为1；

一个2字节整数：颜色数。

举例：

所以，组合起来用unpack读取：

>>> struct.unpack('<ccIIIIIIHH', s)

(b'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)

结果显示，b'B'、b'M'说明是Windows位图，位图大小为640x360，颜色数为24。

关于大小端参照评论：

大端模式，是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；这和我们的阅读习惯一致。

小端模式，是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低。

[大小端模式CSDN详解](https://blog.csdn.net/ce123_zhouwei/article/details/6971544)(点击即可)(阅读数14万)

**Hashlib(难!)**

hashlib提供了常见的摘要(哈希,散列)算法：MD5，SHA1等等。

运用格式：

md5 = hashlib.md5()

md5.update('how to use md5 in python hashlib?'.encode('utf-8'))

print(md5.hexdigest())

或者也可以：

hashlib.md5(s.encode(‘utf-8’)).hexdigest()

调用SHA1和调用MD5完全类似

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示。

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

**hmac(难!)**

如果salt是我们自己随机生成的，通常我们计算MD5时采用md5(message + salt)。但实际上，把salt看做一个“口令”，加salt的哈希就是：计算一段message的哈希时，根据不通口令计算出不同的哈希。要验证哈希值，必须同时提供正确的口令。

这实际上就是Hmac算法：Keyed-Hashing for Message Authentication。它通过一个标准算法，在计算哈希的过程中，把key混入计算过程中。

方法：

>>> import hmac

>>> message = b'Hello, world!'

>>> key = b'secret'

>>> h = hmac.new(key, message, digestmod='MD5')

>>> # 如果消息很长，可以多次调用h.update(msg)

>>> h.hexdigest()

'fa4ee7d173f2d97ee79022d1a7355bcf'

可见使用hmac和普通hash算法非常类似。hmac输出的长度和原始哈希算法的长度一致。需要注意传入的key和message都是bytes类型，str类型需要首先编码为bytes。

Python内置的hmac模块实现了标准的Hmac算法，它利用一个key对message计算“杂凑”后的hash，使用hmac算法比标准hash算法更安全，因为针对相同的message，不同的key会产生不同的hash。

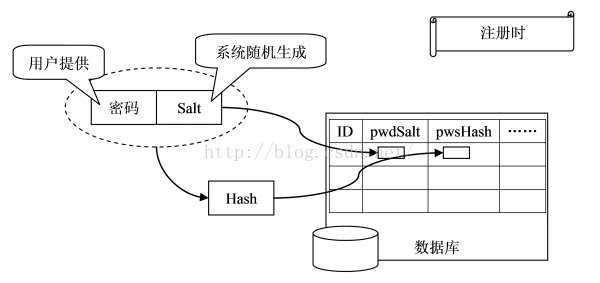


图1.注册时

<http://ww2.sinaimg.cn/large/0060lm7Tly1fnt80yuntfg30go07vt8s.gif>

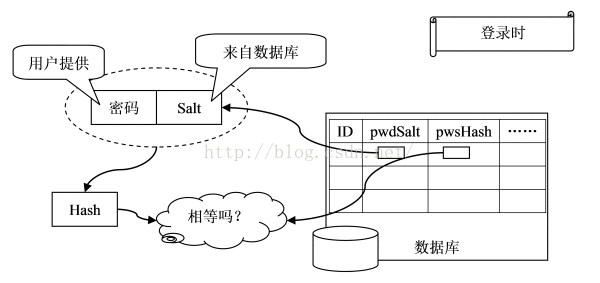


图2.登陆时

[来源地址！！！](http://ww4.sinaimg.cn/large/0060lm7Tly1fnt83vbgqvg30go07vmxa.gif)

评论区大佬：

hmac算法并不是拼凑

hashlib的加salt只是把原始密码字符串和一个salt字符串拼凑成一个新的字符串再去获取md5

a=hashlib.md5('qq'.encode('utf-8')).hexdigest()

a

'099b3b060154898840f0ebdfb46ec78f'

+++++++++++++++分隔符+++++++++++++++++++++++

b=hmac.new('q'.encode('utf-8'),'q'.encode('utf-8'),'MD5').hexdigest()

b

'18fbebb00df535b536f7f8710fc6a22b'

贴上一段代码 说明hmac并不是拼凑字符串 显然更安全

[python之数据加密与解密及相关操作](https://www.cnblogs.com/yyds/p/7072492.html)(直接点击即可)

https://blog.csdn.net/Rozol/article/details/72566661

**itertools**