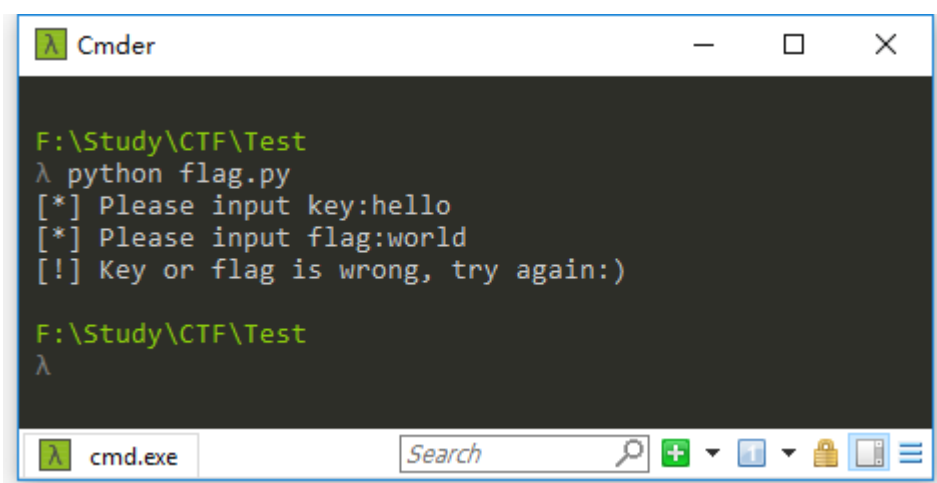


## 群题目：you\_need\_python\_write\_up

### 0x00 获取 key

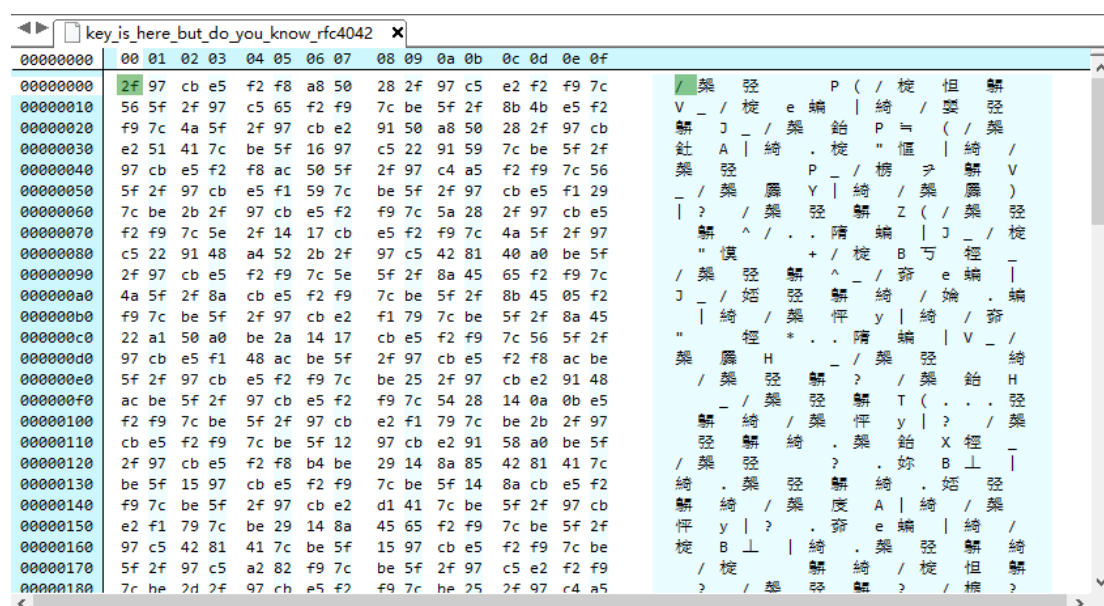
下载得到题目包 you\_need\_python.7z，解压得到 flag.py 和 key\_is\_here\_but\_do\_you\_know\_rfc4042 和两个文件，尝试运行 flag.py：



```
F:\Study\CTF\Test
λ python flag.py
[*] Please input key:hello
[*] Please input flag:world
[!] Key or flag is wrong, try again:)

F:\Study\CTF\Test
λ
```

要求输入 key，由另一个文件名 key\_is\_here\_but\_do\_you\_know\_rfc4042，可知 key 需要从 key\_is\_here\_but\_do\_you\_know\_rfc4042 中获得，查看 key\_is\_here\_but\_do\_you\_know\_rfc4042 文件的内容：



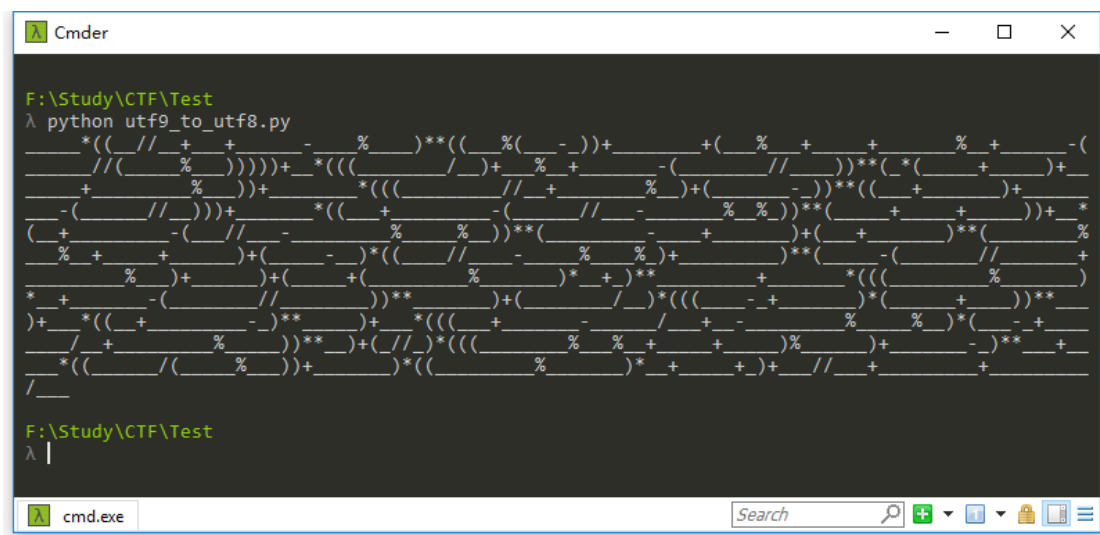
文件内容是一些乱码，但是文件名提示了 rfc4042，于是网上查询相关资料，知道 rfc4042 中定义了 utf9 和 utf18 两种 Unicode 转换编码格式，在了解转换原理之后，根据题意推测是将 utf9 编码转化了 utf8 编码，通过资料查询得知 python 已有 utf9 模块（如果没有查询到也可以自己动手编写转换代码），使用 pip 安装

utf9 模块:

1. pip search utf9
2. pip install utf9

编写代码将文件内容的 utf9 编码转化 utf8 编码:

```
1. #coding utf-8
2. #utf9_to_utf8.py
3. import utf9
4.
5. utf9_file = open('key_is_here_but_do_you_know_rfc4042','rb')
6. utf9_data = utf9_file.read()
7. decoded_data = utf9.utf9decode(utf9_data)
8. print decoded_data
9. decoded_file = open('decoded','w')
10. decoded_file.write(decoded_data)
11. decoded_file.close()
```



得到一堆符号串,但是经常仔细观察,除了“\_”符号外,其他符号都是 Python 中的算数运算符,“(”,“)”括号表示优先级,然后开脑洞“\_”为数字“1”,“\_\_”为数字“2”,依次类推“\_\_\_\_\_”为数字“9”,在熟悉了 utf9 模块的使用后尝试编写转换代码,代码执行后得到数字:5287002131074331513,尝试转换为 16 进制然后转换为 ASCII 字符:

```
1. #coding=utf-8
2. #key_show.py
3. import binascii
4.
5. _ = 1
6. __ = 2
```

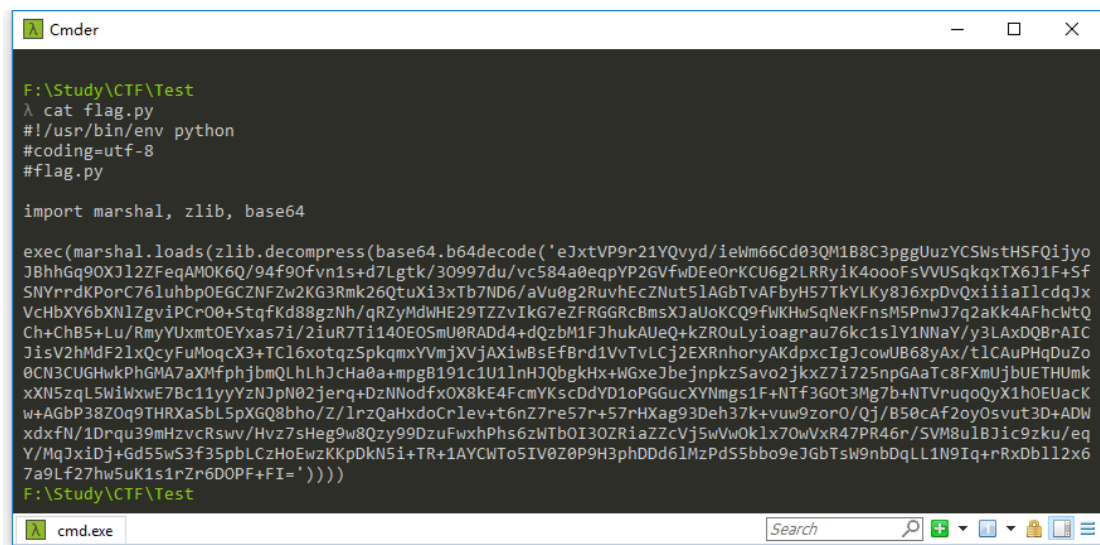
```

7. ____ = 3
8. ____ = 4
9. ____ = 5
10. ____ = 6
11. ____ = 7
12. ____ = 8
13. ____ = 9
14.
15. print binascii.a2b_hex(hex(eval("____*((__//__+__+____-____%____))*((____
(____-__))+(____+((____%____+____+____%____+____-((____//((____%____)))))+
_*(____((____//____)+____%____+____-((____//____))*((____+____)+____+
____%____))+____*((____//____+____%____)+(____-____))*((____+____
____)+____-((____//____))+____*((____+____-((____//____-____%
____%____))*((____+____+____))+____*((____+____-((____//____-____%____%____))*
*((____-____+____)+(____+____))*((____%____%____+____+____)+(____
-____)*((____//____-____%____%____)+____))*((____-((____//____+____
____%____)+____)+(____+((____%____)*____+____))*____+____*((____
____%____)*____+____-((____//____))*____)+(____//____)*((____
-____+____)*(____+____))*____+____*((____+____-____))*____+____*((____+____
____-____//____+____-____%____%____)*(____-____+____//____+____%____))*
____)+(____//____)*((____%____%____+____+____)%____)+____-____))*____+____*((____
____//((____%____))+____)*((____%____)*____+____+____)+____//____+____
+____//____"))[2:][:-1])

```

运行得到 key。

## 0x01 分析 flag.py



```

C:\Test>cat flag.py
#!/usr/bin/env python
#coding=utf-8
#flag.py

import marshal, zlib, base64

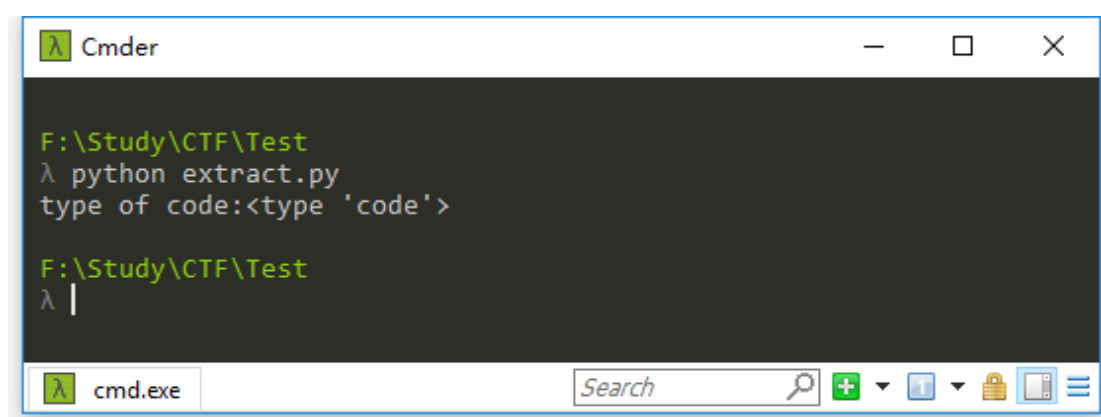
exec(marshal.loads(zlib.decompress(base64.b64decode('eJxtVP9r21YQvyd/ieWm66Cd03QM1B8C3pggUuzYCSWstHSFQijyo
JBhhGq90XJ12ZFeqAMOK6Q/94f90fvn1s+d7Lgtk/30997du/vc584a0eqpYP2GVfwDEe0rKCU6g2LRRyik4oooFsvVUSqkqxTX6J1F+Sf
SNYrrdKPorC761uhbpOEGCZNFZw2KG3Rmk26QtuXi3xTb7ND6/aVu0g2RuvhEcZNut51AGbTvAFbyH57TkYLYkY8J6xpDvQxiiaI1cdqJx
VcHbXY6bXN1ZgviPCr00+StqfKd88gzNh/qRZyMdwHE29TZZvIkG7eZFRGGRcBmsXJaUoKCQ9fWKhW5qNeKfnsM5PnwJ7q2aKk4AFhcWtQ
Ch+ChB5+Lu/RmyYUxmtOEYxas7i/2iuR7Ti140E0SmU0RADD4+dQzbM1FjhukAUeQ+kZR0uLyioagru76kc1s1Y1NNaY/y3LaxDQ8rAIC
JisV2hMdF21xQcyFuMoqcX3+TC16xotqzSpkqmxYVmJxVjAXiWBSefBrd1VvTvLCj2EXRnhoryAKdpxcIgJcowUB68yAx/t1CAuPHQDuZo
0CN3CUGHwKPhGMA7aXmFphjbmQLhLhJcHa0a+mpgB191c1U11nHJQbgkHx+WGxeJbejnpkzSavo2jKxZ7i725npGAaTc8FXmUjbuETHUmK
xXN5zqL5WiWxE7Bc11yyYzNjPn02jerq+DzNNodfxOX8kE4FcmYKscDdYD1oPGGucXYNmgs1F+NTf3G0t3Mg7b+NTVruqoQyX1h0EUaCk
w+AGbP38Z0q9THRxaSbL5pXGQ8bho/Z/1rzQaHxdoCr1ev+t6nZ7re57r+57rHXag93Deh37k+vuw9zorO/Qj/B50cAf2oy0svut3D+ADW
xdxfN/1Drqu39mHzvcRswv/Hvz7sHeg9w8Qzy99DzuFwxhPhs6zWTbOI3OZRiaZZcVj5wVw0k1x70wVxR47PR46r/SVM8u1Bjic9zku/eq
Y/MqJxIdj+Gd55wS3f35pbLCzHoEwzKKpDkN5i+TR+1AYCWT05IV0Z0P9H3phDDd61MzPd55bbo9eJGbTsW9nbDqLL1N9Iqr+RxDb1l2x6
7a9Lf27hw5uK1s1rZr6DOPF+FI=''))))

```

查询有关 marshal, zlib, base64 模块和 exec 函数的资料, 反向推测源代码或字节码先是使用 marshal 模块序列化, 之后使用 zlib 压缩, 最后使用 base64 编码,

而 `exec` 语句可以用来执行储存在字符串或者文件中的 `python` 语句或 `python` 字节码。所以尝试提取 `exec` 语句执行的内容：

```
1. import marshal, zlib, base64
2.
3. code = marshal.loads(zlib.decompress(base64.b64decode('eJxtVP9r21YQvyd/ieWm66
Cd03QM1B8C3pggUuzYCSWstHSFQijyoJBhhGq90XJ12ZFeqAMOK6Q/94f90fvn1s+d7Lgtk/30997
du/vc584a0eqpYP2GVfwDEeOrKCU6g2LRRyiK4ooofSVVUSqkqXTX6J1F+SfSNYrrdKPorC761uhb
pOEGCZNFZw2KG3Rmk26QtuXi3xTb7ND6/aVu0g2RuvhEcZNut51AGbTvAFbyH57TkYlKy8J6xpDvQ
xiiaIldcqJxVcHbXY6bXN1ZgviPCrO0+Stqfkd88gzNh/qRZyMdWHE29TZZvIkG7eZFRGGRcBmsX
JaUoKCQ9fWKHwSqNeKFnsM5PnwJ7q2aKk4AFhcWtQCh+ChB5+Lu/RmyYUxmtOEYxas7i/2iuR7Ti1
40EOsmU0RADd4+dQzbM1FJhukAUeQ+kZR0uLyioagrau76kc1s1Y1NNaY/y3LAXDQBrAICJisV2hM
dF21xQcyFuMoqcX3+TC16xotqzSpkqmxYVmJXVjAXiwBsEfBrd1VvTvLCj2EXRnhoryAKdpxcIgJc
owUB68yAx/tlCAuPHqDuZo0CN3CUGHwkPhGMA7aXmfphjbmQLhLhJcHa0a+mpgB191c1U1lnHJQbg
kHx+WGxeJbejnpkzSavo2jkxZ7i725npGAaTc8FXmUjBUETHUmKxXN5zqL5WiWxwE7Bc11yyYzNJp
N02jerq+DzNNodfxOX8kE4FcmYKscDdYD1oPGGucXYNmgs1F+NTf3G0t3Mg7b+NTVruqoQyX1h0EU
acKw+AGbP38ZOq9THRxaSbL5pXGQ8bho/Z/1rzQaHxdoCrlev+t6nZ7re57r+57rHXag93Deh37k+
vuW9zorO/Qj/B50cAf2oy0svut3D+ADWxdxfN/1Drqu39mHzvcRswv/Hvz7sHeg9w8Qzy99DzuFwx
hPhs6zWTb0I30ZRiaZZcVj5wVw0k1x70wVxR47PR46r/SVM8u1BJic9zku/eqY/MqJxiDj+Gd55wS
3f35pbLCzHoEwzKKpDkN5i+TR+1AYCWt05IV0Z0P9H3phDDd61MzPdS5bbo9eJGbTsw9nbDqLL1N9
Iq+rRxDbl12x67a9Lf27hw5uK1s1rZr6DOPF+FI=')))
4. print "type of code:%s" %type(code)
```



得知 `code` 类型为 `Python` 代码对象 (`PyCodeObject`)，为了便于题目讲解，以下简单阐述个人对 `Python` 解释器运行原理，`pyc` 文件格式以及 `Python` 代码对象的理解。

当你使用命令 `python demo.py`（这里的 `python` 其实是 `python` 解释器中的 `CPython`）时，会启动 `Python` 解释器，`Python` 解释器首先会检查当前目录是否存在相应的 `demo.pyc` 或 `demo.pyo` 文件（`pyo` 文件是经过 `Python` 解释器编译优化，然后将内存中的字节码对象序列化并加上 `pyo` 文件头信息的可储存的二进制文件），如果都存在优先检查运行 `demo.pyo`，如果不存在相应的 `demo.pyc` 或

demo.py 那么 Python 解释器就会编译 demo.py, 因为 Python 解释器检查运行 pyc 文件和 py 文件两者过程类似, 下面以 pyc 文件为例, 如果存在相应的 pyc 文件 Python 解释器则验证 demo.pyc 文件头的前四个字节幻数 (magic number) 的值, 以确保当前文件是 pyc 文件格式且当前 Python 解释器版本能支持, 如果验证不通过则给出提示退出, 如果验证通过继续检查 demo.pyc 文件头的后四个字节 py 源文件修改时间 (mtime, modify time), 如果 demo.pyc 文件头的 mtime 和现在 demo.py 源文件的修改时间一样, 那么 Python 解释器就会把 demo.pyc (不包括文件头的 8 个字节) 加载到内存并将 demo.pyc 文件反序列化为字节码对象 (PyCodeObject) 交给 Python 虚拟机 (Python 虚拟机只是 Python 解释器中实现的一部分, 为了便于理解人们把这部分抽象命名为 Python 虚拟机) 处理执行, 运行完毕 Python 解释器退出。如果 demo.pyc 文件头的 mtime 早于现在 py 源文件的修改时间, 说明之前的 demo.py 源文件经过了修改需要重新编译。

Python 解释器所做的工作就不深入了, 可以简单理解为 Python 解释器将 demo.py 加载到内存编译一个对应的字节码对象, 然后交由 Python 虚拟机程序处理执行, Python 虚拟机会从编译得到的 PyCodeObject 对象中依次读入每一条字节码指令, 并在当前的上下文环境中执行这条字节码指令。而执行完毕之后 Python 编译器会根据情况生成相应的 pyc 文件, 生成过程为 Python 解释器将内存的 Python 代码对象反序列化并加上相应的 py 文件头信息一起写入到硬盘, 所以 pyc 文件只是 Python 代码对象 (PyCodeObject) 在硬盘上的表现形式, 生成 pyo 文件过程也类似, 只是多了 Python 解释器优化的过程。

## 0x02 提取得到 pyc 文件

所以我们要做的便是编写代码将 code 这个 Python 代码对象加上相应 pyc 文件头信息提取出来写入磁盘生成 pyc 文件, 生成 pyc 文件目的是便于我们反编译 pyc 文件得到相应的 py 源码。当然你也可以通过 Python 自带的 dis 模块慢慢分析 pyc 文件中的字节码指令。

```
1. #!/usr/bin/env python
2. #coding = utf-8
3. #PyCodeObject_to_pyc.py
4. import py_compile, imp, os, marshal, zlib, base64
5.
6. code = marshal.loads(zlib.decompress(base64.b64decode('eJxtVP9r21YQvyd/iewM66
Cd03QM1B8C3pggUuzYCSWstHSFQijyoJBhhGq90XJ12ZFeqAMOK6Q/94f90fvn1s+d7Lgtk/30997
du/vc584a0eqpYP2GVfwDEeOrKCU6g2LRRyiK4ooofSVVUSqkqxTX6J1F+SfSNYrrdKPorC76luhb
pOEGCZNFZw2KG3Rmk26QtuXi3xTb7ND6/aVu0g2RuvhEcZNut5lAGbTvAFbyH57TkYlKy8J6xpDvQ
xiiiaIlcdqJxVcHbXY6bXN1ZgviPCrO0+StqfKd88gzNh/qRZyMdWHE29TZZvIkG7eZFRGGRcBmsX
JaUoKCQ9fWKHwSqNeKFnsM5PnwJ7q2aKk4AFhcWtQCh+ChB5+Lu/RmyYUxmtOEYxas7i/2iuR7Ti1
40EOSmU0RADd4+dQzbM1FJhukAUeQ+kZR0uLyioagrau76kc1s1Y1NNaY/y3LAXDQBraICJisV2hM
```

```

dF21xQcyFuMoqcX3+TC16xotqzSpkqmxYVmJXVjAXiwBsEfBrd1VvTvLCj2EXRnhoryAKdpxcIgJc
owUB68yAx/t1CAuPHqDuZo0CN3CUGHwkPhGMA7aXmfphjbmQLhLhJcHa0a+mpgB191c1U1lnHJQbg
kHx+WGxeJbejnpkzSavo2jkxZ7i725npGAATc8FXmUjBUETHUmKxXN5zqL5WiWxE7Bc11yyYzNJp
N02jerq+DzNNodfxOX8kE4FcmYKscDdYD1oPGGucXYNmgs1F+NTf3G0t3Mg7b+NTVruqoQyX1h0EU
acKw+AGbP38Z0q9THRxaSbL5pXGQ8bho/Z/1rzQaHxdoCrlev+t6nZ7re57r+57rHXag93Deh37k+
vuW9zor0/Qj/B50cAf2oy0svut3D+ADWxdxfN/1Drqu39mHzvcRswv/Hvz7sHeg9w8Qzy99DzuFwx
hPhs6zWTb0I3OZRiaZZcVj5wVw0klx70wVxR47PR46r/SVM8ulBJic9zku/eqY/MqJxiDj+Gd55wS
3f35pbLCzHoEwzKKpDkN5i+TR+1AYCWT05IV0Z0P9H3phDDd61MzPdS5bbo9eJGbTsw9nbDqLL1N9
Iq+rRxDb1l2x67a9Lf27hw5uK1s1rZr6DOPF+FI=''))
7.
8. def PyCodeObject_to_pyc(py_code_obj, pyc_file):
9.     with open(pyc_file, 'wb') as pyc:
10.         pyc_magic = imp.get_magic()
11.         pyc.write(pyc_magic)
12.         mtime = long(os.fstat(pyc.fileno()).st_mtime)
13.         py_compile.wr_long(pyc, mtime)
14.         marshal.dump(py_code_obj, pyc)
15.         pyc.flush()
16.         pyc.close()
17.
18. def main():
19.     PyCodeObject_to_pyc(code, 'extract.pyc')
20.
21. if __name__ == '__main__':
22.     main()

```

### 0x03 反编译 pyc 文件得到 py 源文件

将生成的 pyc 通过 Python 工具如 uncompyle2 等反编译得到 py 源码, 这里直接通过在线的 pyc 反编译[网站](#)得到 py 源码:

```

1. #!/usr/bin/env python
2. # encoding: utf-8
3.
4. import hashlib
5.
6. def sha1(string):
7.     return hashlib.sha1(string).hexdigest()
8.
9. def calc(strSHA1):
10.     r = 0
11.     for i in strSHA1:
12.         r += int('0x%s' % i, 16)
13.     return r
14.

```

```

15. def encrypt(plain, key):
16.     keySHA1 = sha1(key)
17.     intSHA1 = calc(keySHA1)
18.     r = []
19.     for i in range(len(plain)):
20.         r.append(ord(plain[i]) + int('0x%s' % keySHA1[i % 40], 16) - intSHA1)
21.         intSHA1 = calc(sha1(plain[:i + 1])[:20] + sha1(str(intSHA1))[:20])
22.
23.     return ''.join(map((lambda x: str(x)), r))
24.
25. if __name__ == '__main__':
26.     key = raw_input('[*] Please input key:')
27.     plain = raw_input('[*] Please input flag:')
28.     encryptText = encrypt(plain, key)
29.     cipherText = '-185-147-211-221-164-217-188-169-205-174-211-225-191-234-14
    8-199-198-253-175-157-222-135-240-229-201-154-178-187-244-183-212-222-164'
30.     if encryptText == cipherText:
31.         print '[>] Congratulations! Flag is: %s' % plain
32.         exit()
33.     else:
34.         print '[!] Key or flag is wrong, try again:)'
35.         exit()

```

#### 0x04 分析 py 源文件中加密算法

sha1 函数使用 sha1 算法计算返回 40 位 16 进制散列值，calc 函数计算 40 位 16 进制散列值中每位的整型值并相加，最后返回整型的总和值。

encrypt 函数为核心加密算法函数：

```

1. def encrypt(plain, key):
2.     keySHA1 = sha1(key)
3.     intSHA1 = calc(keySHA1)
4.     r = []
5.     for i in range(len(plain)):
6.         r.append(ord(plain[i]) + int('0x%s' % keySHA1[i % 40], 16) - intSHA1)
7.         intSHA1 = calc(sha1(plain[:i + 1])[:20] + sha1(str(intSHA1))[:20])
8.
9.     return ''.join(map((lambda x: str(x)), r))

```

由第 4 行的 for 可以知道到明文长度和密文长度相同，核心加密语句为第 6，7 行，算法使用 ord 函数取得明文每个字符的 ASCII 整型值，int 函数内容为明文每个字符位置模 40 访问由调用 sha1 函数返回的 40 位 16 进制 keySHA1 字符串中的 16 进制数并转化为 10 进制数与由调用 calc 函数返回的整型值相减，然后将 ord 函数和 int 计算所得值作为密文添加到 r 列表，第 7 行更新 intSHA1 值，



第 9 行转换为“-185-147-211...”格式并返回。

这里我们知道了密文 `cipherText`，密钥 `key`，加密算法 `encrypt`，从而能逆推出解密算法，只要把密文值减去 `int` 函数中的值并对结果使用 `chr` 函数取得明文 `plain`。

### 0x05 编写解密代码

```
1. #coding:utf-8
2. #decrypt.py
3.
4. import hashlib
5.
6. def sha1(string):
7.     return hashlib.sha1(string).hexdigest()
8.
9. def calc(strSHA1):
10.    r = 0
11.    for i in strSHA1:
12.        r += int("0x%s" % i, 16)
13.    return r
14.
15. def decrypt(strCipher, strKey):
16.    listCipher = map(lambda x: int(x), strCipher.replace('-', ' ')[1:].split(' '))
17.    strKeySHA1 = sha1(strKey)
18.    intSHA1 = calc(strKeySHA1)
19.    strPlain = ''
20.    for i in range(len(listCipher)):
21.        strPlain += chr(listCipher[i] + intSHA1 - int("0x%s" % strKeySHA1[i%4
22.        0], 16))
23.        intSHA1 = calc(sha1(strPlain[:i + 1]))[:20] + sha1(str(intSHA1))[:20
24.        ])
25.    return strPlain
26.
27. if __name__ == '__main__':
28.    strCipher= '-185-147-211-221-164-217-188-169-205-174-211-225-191-234-148-
29.    199-198-253-175-157-222-135-240-229-201-154-178-187-244-183-212-222-164'
30.    strKey = 'xxxxxx'
31.    strPlain = decrypt(strCipher, strKey)
32.    print strPlain
```

最终获得 flag。