More Standard Modules

> *"Now, imagine that your friend kept complaining that she didn't want to visit you since she found it too hard to climb up the drain pipe, and you kept telling her to use the friggin' stairs like everyone else..."*
>
> *eff-bot, June 1998*

Overview

This chapter describes a number of modules that are used in many Python programs. It's perfectly possible to write large Python programs without using them, but they can help you save a lot of time and effort.

Files and streams

The **fileinput** module makes it easy to write different kinds of text filters. This module provides a wrapper class, which lets you use a simple **for-in** statement to loop over the contents of one or more text files.

The **StringIO** module (and the **cStringIO** variant) implements an in-memory file object. You can use **StringIO** objects in many places where Python expects an ordinary file object.

Type wrappers

UserDict, **UserList**, and **UserString** are thin wrappers on top of the corresponding built-in types. But unlike the built-in types, these wrappers can be subclassed. This can come in handy if you need a class that works almost like a built-in type, but has one or more extra methods.

Random numbers

The **random** module provides a number of different random number generators. The **whrandom** module is similar, but it also allows you to create multiple generator objects.

Digests and encryption algorithms

The **md5** and **sha** modules are used to calculate cryptographically strong message signatures (so-called "message digests").

The **crypt** module implements a DES-style one-way encryption. This module is usually only available on Unix systems.

The **rotor** module provides simple two-way encryption.

The fileinput module

This module allows you to loop over the contents of one or more text files.

Example: Using the fileinput module to loop over a text file

```
# File:fileinput-example-1.py

import fileinput
import sys

for line in fileinput.input("samples/sample.txt"):
    sys.stdout.write("-> ")
    sys.stdout.write(line)
```

```
-> We will perhaps eventually be writing only small
-> modules which are identified by name as they are
-> used to build larger ones, so that devices like
-> indentation, rather than delimiters, might become
-> feasible for expressing local structure in the
-> source language.
->      -- Donald E. Knuth, December 1974
```

The module also allows you to get metainformation about the current line. This includes **isfirstline**, **filename**, and **lineno**:

Example: Using the fileinput module to process multiple files

```
# File:fileinput-example-2.py

import fileinput
import glob
import string, sys

for line in fileinput.input(glob.glob("samples/*.txt")):
    if fileinput.isfirstline(): # first in a file?
        sys.stderr.write("-- reading %s --\n" % fileinput.filename())
    sys.stdout.write(str(fileinput.lineno()) + " " + string.upper(line))
```

```
-- reading samples\sample.txt --
1 WE WILL PERHAPS EVENTUALLY BE WRITING ONLY SMALL
2 MODULES WHICH ARE IDENTIFIED BY NAME AS THEY ARE
3 USED TO BUILD LARGER ONES, SO THAT DEVICES LIKE
4 INDENTATION, RATHER THAN DELIMITERS, MIGHT BECOME
5 FEASIBLE FOR EXPRESSING LOCAL STRUCTURE IN THE
6 SOURCE LANGUAGE.
7    -- DONALD E. KNUTH, DECEMBER 1974
```

Processing text files in place is also easy. Just call the **input** function with the **inplace** keyword argument set to 1, and the module takes care of the rest.

Example: Using the fileinput module to convert CRLF to LF

```
# File:fileinput-example-3.py

import fileinput, sys

for line in fileinput.input(inplace=1):
    # convert Windows/DOS text files to Unix files
    if line[-2:] == "\r\n":
        line = line[:-2] + "\n"
    sys.stdout.write(line)
```

The shutil module

This utility module contains some functions for copying files and directories. The **copy** function copies a file in pretty much the same way as the Unix **cp** command.

Example: Using the shutil module to copy files

```
# File:shutil-example-1.py

import shutil
import os

for file in os.listdir("."):
    if os.path.splitext(file)[1] == ".py":
        print file
        shutil.copy(file, os.path.join("backup", file))
```

```
aifc-example-1.py
anydbm-example-1.py
array-example-1.py
...
```

The **copytree** function copies an entire directory tree (same as **cp -r**), and **rmtree** removes an entire tree (same as **rm -r**).

Example: Using the shutil module to copy and remove directory trees

```
# File:shutil-example-2.py

import shutil
import os

SOURCE = "samples"
BACKUP = "samples-bak"

# create a backup directory
shutil.copytree(SOURCE, BACKUP)

print os.listdir(BACKUP)

# remove it
shutil.rmtree(BACKUP)

print os.listdir(BACKUP)
```

```
['sample.wav', 'sample.jpg', 'sample.au', 'sample.msg', 'sample.tgz',
...
Traceback (most recent call last):
 File "shutil-example-2.py", line 17, in ?
   print os.listdir(BACKUP)
os.error: No such file or directory
```

The tempfile module

This module allows you to quickly come up with unique names to use for temporary files.

Example: Using the tempfile module to create filenames for temporary files

```
# File:tempfile-example-1.py

import tempfile
import os

tempfile = tempfile.mktemp()

print "tempfile", "=>", tempfile

file = open(tempfile, "w+b")
file.write("*" * 1000)
file.seek(0)
print len(file.read()), "bytes"
file.close()

try:
    # must remove file when done
    os.remove(tempfile)
except OSError:
    pass
```

```
tempfile => C:\TEMP\~160-1
1000 bytes
```

The **TemporaryFile** function picks a suitable name, and opens the file. It also makes sure that the file is removed when it's closed (under Unix, you can remove an open file and have it disappear when the file is closed. On other platforms, this is done via a special wrapper class).

Example: Using the tempfile module to open temporary files

```
# File:tempfile-example-2.py

import tempfile

file = tempfile.TemporaryFile()

for i in range(100):
    file.write("*" * 100)

file.close() # removes the file!
```

The StringIO module

This module implements an in-memory file object. This object can be used as input or output to most functions that expect a standard file object.

Example: Using the StringIO module to read from a static file

```
# File:stringio-example-1.py

import StringIO

MESSAGE = "That man is depriving a village somewhere of a computer scientist."

file = StringIO.StringIO(MESSAGE)

print file.read()
```

```
That man is depriving a village somewhere of a computer scientist.
```

The StringIO class implements memory file versions of all methods available for built-in file objects, plus a **getvalue** method that returns the internal string value.

Example: Using the StringIO module to write to a memory file

```
# File:stringio-example-2.py

import StringIO

file = StringIO.StringIO()
file.write("This man is no ordinary man. ")
file.write("This is Mr. F. G. Superman.")

print file.getvalue()
```

```
This man is no ordinary man. This is Mr. F. G. Superman.
```

StringIO can be used to capture redirected output from the Python interpreter:

Example: Using the StringIO module to capture output

```
# File:stringio-example-3.py

import StringIO
import string, sys

stdout = sys.stdout

sys.stdout = file = StringIO.StringIO()

print """
According to Gbaya folktales, trickery and guile
are the best ways to defeat the python, king of
snakes, which was hatched from a dragon at the
world's start. -- National Geographic, May 1997
"""

sys.stdout = stdout

print string.upper(file.getvalue())
```

```
ACCORDING TO GBAYA FOLKTALES, TRICKERY AND GUILE
ARE THE BEST WAYS TO DEFEAT THE PYTHON, KING OF
SNAKES, WHICH WAS HATCHED FROM A DRAGON AT THE
WORLD'S START. -- NATIONAL GEOGRAPHIC, MAY 1997
```

The cStringIO module

(Optional). This module contains a faster implementation of the **StringIO** module It works exactly like **StringIO**, but it cannot be subclassed.

Example: Using the cStringIO module

```
# File:cstringio-example-1.py

import cStringIO

MESSAGE = "That man is depriving a village somewhere of a computer scientist."

file = cStringIO.StringIO(MESSAGE)

print file.read()
```

```
That man is depriving a village somewhere of a computer scientist.
```

To make your code as fast as possible, but also robust enough to run on older Python installations, you can fall back on the **StringIO** module if **cStringIO** is not available:

Example: Falling back on the StringIO module

```
# File:cstringio-example-2.py

try:
    import cStringIO
    StringIO = cStringIO
except ImportError:
    import StringIO

print StringIO
```

```
<module 'cStringIO' (built-in)>
```

The mmap module

(New in 2.0) This module provides an interface to the operating system's memory mapping functions. The mapped region behaves pretty much like a string object, but data is read directly from the file.

Example: Using the mmap module

```
# File:mmap-example-1.py

import mmap
import os

filename = "samples/sample.txt"

file = open(filename, "r+")
size = os.path.getsize(filename)

data = mmap.mmap(file.fileno(), size)

# basics
print data
print len(data), size

# use slicing to read from the file
print repr(data[:10]), repr(data[:10])

# or use the standard file interface
print repr(data.read(10)), repr(data.read(10))
```

```
<mmap object at 008A2A10>
302 302
'We will pe' 'We will pe'
'We will pe' 'rhaps even'
```

Under Windows, the file must currently be opened for both reading and writing (**r+**, or **w+**), or the **mmap** call will fail.

Memory mapped regions can be used instead of ordinary strings in many places, including regular expressions and many string operations:

Example: Using string functions and regular expressions on a mapped region

```
# File:mmap-example-2.py

import mmap
import os, string, re

def mapfile(filename):
    file = open(filename, "r+")
    size = os.path.getsize(filename)
    return mmap.mmap(file.fileno(), size)

data = mapfile("samples/sample.txt")

# search
index = data.find("small")
print index, repr(data[index-5:index+15])

# regular expressions work too!
m = re.search("small", data)
print m.start(), m.group()
```

```
43 'only small\015\012modules '
43 small
```

The UserDict module

This module contains a dictionary class which can be subclassed (it's actually a Python wrapper for the built-in dictionary type).

The following example shows an enhanced dictionary class, which allows dictionaries to be 'added' to each other, and be initialized using the keyword argument syntax.

Example: Using the UserDict module

```
# File:userdict-example-1.py

import UserDict

class FancyDict(UserDict.UserDict):

    def __init__(self, data = {}, **kw):
        UserDict.UserDict.__init__(self)
        self.update(data)
        self.update(kw)

    def __add__(self, other):
        dict = FancyDict(self.data)
        dict.update(b)
        return dict

a = FancyDict(a = 1)
b = FancyDict(b = 2)

print a + b
```

```
{'b': 2, 'a': 1}
```

The UserList module

This module contains a list class which can be subclassed (simply a Python wrapper for the built-in list type).

In the following example, **AutoList** instances work just like ordinary lists, except that they allow you to insert items at the end by assigning to it.

Example: Using the UserList module

```
# File:userlist-example-1.py

import UserList

class AutoList(UserList.UserList):

    def __setitem__(self, i, item):
        if i == len(self.data):
            self.data.append(item)
        else:
            self.data[i] = item

list = AutoList()

for i in range(10):
    list[i] = i

print list
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The UserString module

(New in 2.0) This module contains two classes, **UserString** and **MutableString**. The former is a wrapper for the standard string type which can be subclassed, the latter is a variation that allows you to modify the string in place.

Note that **MutableString** is not very efficient. Most operations are implemented using slicing and string concatenation. If performance is important, use lists of string fragments, or the **array** module.

Example: Using the UserString module

```
# File:userstring-example-1.py

import UserString

class MyString(UserString.MutableString):

    def append(self, s):
        self.data = self.data + s

    def insert(self, index, s):
        self.data = self.data[index:] + s + self.data[index:]

    def remove(self, s):
        self.data = self.data.replace(s, "")

file = open("samples/book.txt")
text = file.read()
file.close()

book = MyString(text)

for bird in ["gannet", "robin", "nuthatch"]:
    book.remove(bird)

print book
```

```
...
C: The one without the !
P: The one without the -!!! They've ALL got the !! It's a
Standard British Bird, the , it's in all the books!!!
...
```

The traceback module

This module allows you to print exception tracebacks inside your programs, just like the interpreter does when you don't catch an exception yourself.

Example: Using the traceback module to print a traceback

```
# File:traceback-example-1.py

# note! importing the traceback module messes up the
# exception state, so you better do that here and not
# in the exception handler
import traceback

try:
    raise SyntaxError, "example"
except:
    traceback.print_exc()
```

```
Traceback (innermost last):
  File "traceback-example-1.py", line 7, in ?
SyntaxError: example
```

To put the traceback in a string, use the **StringIO** module:

Example: Using the traceback module to copy a traceback to a string

```
# File:traceback-example-2.py

import traceback
import StringIO

try:
    raise IOError, "an i/o error occurred"
except:
    fp = StringIO.StringIO()
    traceback.print_exc(file=fp)
    message = fp.getvalue()

    print "failure! the error was:", repr(message)
```

```
failure! the error was: 'Traceback (innermost last):\012  File
"traceback-example-2.py", line 5, in ?\012IOError: an i/o error
occurred\012'
```

If you wish to format the traceback in a non-standard way, you can use the **extract_tb** function to convert a traceback object to a list of stack entries:

Example: Using the traceback module to decode a traceback object

```
# File:traceback-example-3.py

import traceback
import sys

def function():
    raise IOError, "an i/o error occurred"

try:
    function()
except:
    info = sys.exc_info()
    for file, lineno, function, text in traceback.extract_tb(info[2]):
        print file, "line", lineno, "in", function
        print "=>", repr(text)
    print "** %s: %s" % info[:2]
```

```
traceback-example-3.py line 8 in ?
=> 'function()'
traceback-example-3.py line 5 in function
=> 'raise IOError, "an i/o error occurred"'
** exceptions.IOError: an i/o error occurred
```

The errno module

This module defines a number of symbolic error codes, such as **ENOENT** ("no such directory entry"), **EPERM** ("permission denied"), and others. It also provides a dictionary mapping from platform dependent numerical error codes to symbolic names.

In most cases, the **IOError** exception provides a 2-tuple with the numerical error code, and an explanatory string. If you need to distinguish between different error codes, use the symbolic names where possible.

Example: Using the errno module

```
# File:errno-example-1.py

import errno

try:
    fp = open("no.such.file")
except IOError, (error, message):
    if error == errno.ENOENT:
        print "no such file"
    elif error == errno.EPERM:
        print "permission denied"
    else:
        print message
```

```
no such file
```

The following example is a bit contrived, but it shows how to use the **errorcode** dictionary to map from a numerical error code to the symbolic name.

Example: Using the errorcode dictionary

```
# File:errno-example-2.py

import errno

try:
    fp = open("no.such.file")
except IOError, (error, message):
    print error, repr(message)
    print errno.errorcode[error]
```

```
2 'No such file or directory'
ENOENT
```

The getopt module

This module contains functions to extract command line options and arguments. It can handle both short and long option formats.

The second argument specifies the short options that should be allowed. A colon (:) after an option name means that option must have an additional argument.

Example: Using the getopt module

```
# File:getopt-example-1.py

import getopt
import sys

# simulate command line invocation
sys.argv = ["myscript.py", "-l", "-d", "directory", "filename"]

# process options
opts, args = getopt.getopt(sys.argv[1:], "ld:")

long = 0
directory = None

for o, v in opts:
    if o == "-l":
        long = 1
    elif o == "-d":
        directory = v

print "long", "=", long
print "directory", "=", directory
print "arguments", "=", args
```

```
long = 1
directory = directory
arguments = ['filename']
```

To make it look for long options, pass a list of option descriptors as the third argument. If an option name ends with an equal sign (=), that option must have an additional argument.

Example: Using the getopt module to handle long options

```
# File:getopt-example-2.py

import getopt
import sys

# simulate command line invocation
sys.argv = ["myscript.py", "--echo", "--printer", "lp01", "message"]

opts, args = getopt.getopt(sys.argv[1:], "ep:", ["echo", "printer="])

# process options
echo = 0
printer = None

for o, v in opts:
    if o in ("-e", "--echo"):
        echo = 1
    elif o in ("-p", "--printer"):
        printer = v

print "echo", "=", echo
print "printer", "=", printer
print "arguments", "=", args
```

```
echo = 1
printer = lp01
arguments = ['message']
```

The getpass module

This module provides a platform-independent way to enter a password in a command-line program.

getpass(prompt) -> string prints the prompt string, switches off keyboard echo, and reads a password. If the prompt argument is omitted, it prints "**Password:** "

getuser() -> string gets the current username, if possible.

Example: Using the getpass module

```
# File:getpass-example-1.py

import getpass

usr = getpass.getuser()

pwd = getpass.getpass("enter password for user %s: " % usr)

print usr, pwd
```

```
enter password for user mulder:
mulder trustno1
```

The glob module

This module generates lists of files matching given patterns, just like the Unix shell.

File patterns are similar to regular expressions, but simpler. An asterisk (*****) matches zero or more characters, and a question mark (**?**) exactly one character. You can also use brackets to indicate character ranges, such as **[0-9]** for a single digit. All other characters match themselves.

glob(pattern) returns a list of all files matching a given pattern.

Example: Using the glob module

```
# File:glob-example-1.py

import glob

for file in glob.glob("samples/*.jpg"):
    print file
```

```
samples/sample.jpg
```

Note that **glob** returns full path names, unlike the **os.listdir** function. **glob** uses the **fnmatch** module to do the actual pattern matching.

The fnmatch module

This module allows you to match filenames against a pattern.

The pattern syntax is the same as that used in Unix shells. An asterisk (*) matches zero or more characters, and a question mark (**?**) exactly one character. You can also use brackets to indicate character ranges, such as **[0-9]** for a single digit. All other characters match themselves.

Example: Using the fnmatch module to match files

```
# File:fnmatch-example-1.py

import fnmatch
import os

for file in os.listdir("samples"):
    if fnmatch.fnmatch(file, "*.jpg"):
        print file
```

```
sample.jpg
```

The **translate** function converts a file pattern to a regular expression:

Example: Using the fnmatch module to convert a pattern to a regular expression

```
# File:fnmatch-example-2.py

import fnmatch
import os, re

pattern = fnmatch.translate("*.jpg")

for file in os.listdir("samples"):
    if re.match(pattern, file):
        print file

print "(pattern was %s)" % pattern
```

```
sample.jpg
(pattern was .*\.jpg$)
```

This module is used by the **glob** and **find** modules.

The random module

> *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"*
>
> *John von Neumann, 1951*

This module contains a number of random number generators.

The basic random number generator (after an algorithm by Wichmann & Hill, 1982) can be accessed in several ways:

Example: Using the random module to get random numbers

```
# File:random-example-1.py

import random

for i in range(5):

    # random float: 0.0 <= number < 1.0
    print random.random(),

    # random float: 10 <= number < 20
    print random.uniform(10, 20),

    # random integer: 100 <= number <= 1000
    print random.randint(100, 1000),

    # random integer: even numbers in 100 <= number < 1000
    print random.randrange(100, 1000, 2)
```

```
0.946842713956 19.5910069381 709 172
0.573613195398 16.2758417025 407 120
0.363241598013 16.8079747714 916 580
0.602115173978 18.386796935 531 774
0.526767588533 18.0783794596 223 344
```

Note that **randint** function can return the upper limit, while the other functions always returns values smaller than the upper limit.

The **choice** function picks a random item from a sequence. It can be used with lists, tuples, or any other sequence (provided it can be accessed in random order, of course):

Example: Using the random module to chose random items from a sequence

```
# File:random-example-2.py

import random

# random choice from a list
for i in range(5):
    print random.choice([1, 2, 3, 5, 9])
```

```
2
3
1
9
1
```

In 2.0 and later, the **shuffle** function can be used to shuffle the contents of a list (that is, generate a random permutation of a list in-place). The following example also shows how to implement that function under 1.5.2 and earlier:

Example: Using the random module to shuffle a deck of cards

```
# File:random-example-4.py

import random

try:
    # available in Python 2.0 and later
    shuffle = random.shuffle
except AttributeError:
    def shuffle(x):
        for i in xrange(len(x)-1, 0, -1):
            # pick an element in x[:i+1] with which to exchange x[i]
            j = int(random.random() * (i+1))
            x[i], x[j] = x[j], x[i]

cards = range(52)

shuffle(cards)

myhand = cards[:5]

print myhand
```

```
[4, 8, 40, 12, 30]
```

This module also contains a number of random generators with non-uniform distribution. For example, the **gauss** function generates random numbers with a gaussian distribution:

Example: Using the random module to generate gaussian random numbers

```
# File:random-example-3.py

import random

histogram = [0] * 20

# calculate histogram for gaussian
# noise, using average=5, stddev=1
for i in range(1000):
   i = int(random.gauss(5, 1) * 2)
   histogram[i] = histogram[i] + 1

# print the histogram
m = max(histogram)
for v in histogram:
   print "*" * (v * 50 / m)
```

```
****
**********
*************************
***********************************
************************************************
**************************************************
*************************************
***************************
*************
***
*
```

See the *Python Library Reference* for more information on the non-uniform generators.

Warning:

The random number generators provided in the standard library are pseudo-random generators. While this might be good enough for many purposes, including simulations, numerical analysis, and games, but it's definitely not good enough for cryptographic use.

The whrandom module

This module provides a pseudo-random number generator (based on an algorithm by Wichmann & Hill, 1982). Unless you need several generators that do not share internal state (for example, in a multi-threaded application), it's better to use the functions in the **random** module instead.

Example: Using the whrandom module

```
# File:whrandom-example-1.py

import whrandom

# same as random
print whrandom.random()
print whrandom.choice([1, 2, 3, 5, 9])
print whrandom.uniform(10, 20)
print whrandom.randint(100, 1000)
```

```
0.113412062346
1
16.8778954689
799
```

To create multiple generators, create instances of the **whrandom** class:

Example: Using the whrandom module to create multiple random generators

```
# File:whrandom-example-2.py

import whrandom

# initialize all generators with the same seed
rand1 = whrandom.whrandom(4,7,11)
rand2 = whrandom.whrandom(4,7,11)
rand3 = whrandom.whrandom(4,7,11)

for i in range(5):
    print rand1.random(), rand2.random(), rand3.random()
```

```
0.123993532536 0.123993532536 0.123993532536
0.180951499518 0.180951499518 0.180951499518
0.291924111809 0.291924111809 0.291924111809
0.952048889363 0.952048889363 0.952048889363
0.969794283643 0.969794283643 0.969794283643
```

The md5 module

This module is used to calculate message signatures (so-called "message digests").

The MD5 algorithm calculates a strong 128-bit signature. This means that if two strings are different, it's highly likely that their MD5 signatures are different as well. Or to put it another way, given an MD5 digest, it's supposed to be nearly impossible to come up with a string that generates that digest.

Example: Using the md5 module

```
# File:md5-example-1.py

import md5

hash = md5.new()
hash.update("spam, spam, and eggs")

print repr(hash.digest())
```

```
'L\005J\243\266\355\243u`\305r\203\267\020F\303'
```

Note that the checksum is returned as a binary string. Getting a hexadecimal or base64-encoded string is quite easy, though:

Example: Using the md5 module to get a hexadecimal or base64-encoded md5 value

```
# File:md5-example-2.py

import md5
import string
import base64

hash = md5.new()
hash.update("spam, spam, and eggs")

value = hash.digest()

print hash.hexdigest()

# in Python 1.5.2 and earlier, use this instead:
# print string.join(map(lambda v: "%02x" % ord(v), value), "")

print base64.encodestring(value)
```

```
4c054aa3b6eda37560c57283b71046c3
TAVKo7bto3VgxXKDtxBGww==
```

Among other things, the MD5 checksum can be used for challenge-response authentication (but see the note on random numbers below):

Example: Using the md5 module for challenge-response authentication

```
# File:md5-example-3.py

import md5
import string, random

def getchallenge():
    # generate a 16-byte long random string.  (note that the built-
    # in pseudo-random generator uses a 24-bit seed, so this is not
    # as good as it may seem...)
    challenge = map(lambda i: chr(random.randint(0, 255)), range(16))
    return string.join(challenge, "")

def getresponse(password, challenge):
    # calculate combined digest for password and challenge
    m = md5.new()
    m.update(password)
    m.update(challenge)
    return m.digest()

#
# server/client communication

# 1. client connects.  server issues challenge.

print "client:", "connect"

challenge = getchallenge()

print "server:", repr(challenge)

# 2. client combines password and challenge, and calculates
# the response

client_response = getresponse("trustno1", challenge)

print "client:", repr(client_response)

# 3. server does the same, and compares the result with the
# client response.  the result is a safe login in which the
# password is never sent across the communication channel.

server_response = getresponse("trustno1", challenge)

if server_response == client_response:
    print "server:", "login ok"
```

```
client: connect
server: '\334\352\227Z#\272\273\212KG\330\265\032>\311o'
client: "l'\305\240-x\245\237\035\225A\254\233\337\225\001"
server: login ok
```

A variation of this can be used to sign messages sent over a public network, so that their integrity can be verified at the receiving end.

Example: Using the md5 module for data integrity checks

```
# File:md5-example-4.py

import md5
import array

class HMAC_MD5:
    # keyed MD5 message authentication

    def __init__(self, key):
        if len(key) > 64:
            key = md5.new(key).digest()
        ipad = array.array("B", [0x36] * 64)
        opad = array.array("B", [0x5C] * 64)
        for i in range(len(key)):
            ipad[i] = ipad[i] ^ ord(key[i])
            opad[i] = opad[i] ^ ord(key[i])
        self.ipad = md5.md5(ipad.tostring())
        self.opad = md5.md5(opad.tostring())

    def digest(self, data):
        ipad = self.ipad.copy()
        opad = self.opad.copy()
        ipad.update(data)
        opad.update(ipad.digest())
        return opad.digest()

#
# simulate server end

key = "this should be a well-kept secret"
message = open("samples/sample.txt").read()

signature = HMAC_MD5(key).digest(message)

# (send message and signature across a public network)

#
# simulate client end

key = "this should be a well-kept secret"

client_signature = HMAC_MD5(key).digest(message)

if client_signature == signature:
    print "this is the original message:"
    print
    print message
else:
    print "someone has modified the message!!!"
```

The **copy** method takes a snapshot of the internal object state. This allows you to precalculate partial digests (such as the padded key, in this example).

For details on this algorithm, see *HMAC-MD5: Keyed-MD5 for Message Authentication* by Krawczyk et al.

Warning:

Don't forget that the built-in pseudo random number generator isn't really good enough for encryption purposes. Be careful.

The sha module

This module provides an alternative way to calculate message signatures. It's similar to the **md5** module, but generates 160-bit signatures instead.

Example: Using the sha module

```
# File:sha-example-1.py

import sha

hash = sha.new()
hash.update("spam, spam, and eggs")

print repr(hash.digest())
print hash.hexdigest()
```

```
'\321\333\003\026I\331\272-j\303\247\240\345\343Tvq\364\346\311'
d1db031649d9ba2d6ac3a7a0e5e3547671f4e6c9
```

See the **md5** examples for more ways to use SHA signatures.

The crypt module

(Optional). This module implements one-way DES encryption. Unix systems use this encryption algorithm to store passwords, and this module is really only useful to generate or check such passwords.

To encrypt a password, call **crypt.crypt** with the password string, plus a "salt", which should consist of two random characters. You can now throw away the actual password, and just store the encrypted string somewhere.

Example: Using the crypt module

```
# File:crypt-example-1.py

import crypt

import random, string

def getsalt(chars = string.letters + string.digits):
    # generate a random 2-character 'salt'
    return random.choice(chars) + random.choice(chars)

print crypt.crypt("bananas", getsalt())
```

```
'py8UGrijma1j6'
```

To verify a given password, encrypt the new password using the two first characters from the encrypted string as the salt. If the result matches the encrypted string, the password is valid. The following example uses the **pwd** module to fetch the encrypted password for a given user.

Example: Using the crypt module for autentication

```
# File:crypt-example-2.py

import pwd, crypt

def login(user, password):
    "Check if user would be able to login using password"
    try:
        pw1 = pwd.getpwnam(user)[1]
        pw2 = crypt.crypt(password, pw1[:2])
        return pw1 == pw2
    except KeyError:
        return 0 # no such user

user = raw_input("username:")
password = raw_input("password:")

if login(user, password):
    print "welcome", user
else:
    print "login failed"
```

For other ways to implement authentication, see the description of the **md5** module.

The rotor module

(Optional). This module implements a simple encryption algorithm, based on the WWII Enigma engine.

Example: Using the rotor module

```
# File:rotor-example-1.py

import rotor

SECRET_KEY = "spam"
MESSAGE = "the holy grail"

r = rotor.newrotor(SECRET_KEY)

encoded_message = r.encrypt(MESSAGE)
decoded_message = r.decrypt(encoded_message)

print "original:", repr(MESSAGE)
print "encoded message:", repr(encoded_message)
print "decoded message:", repr(decoded_message)
```

```
original: 'the holy grail'
encoded message: '\227\271\244\015\305sw\3340\337\252\237\340U'
decoded message: 'the holy grail'
```

The zlib module

(Optional). This module provides support for "zlib" compression. (This compression method is also known as "deflate".)

The **compress** and **decompress** functions take string arguments:

Example: Using the zlib module to compress a string

```
# File:zlib-example-1.py

import zlib

MESSAGE = "life of brian"

compressed_message = zlib.compress(MESSAGE)
decompressed_message = zlib.decompress(compressed_message)

print "original:", repr(MESSAGE)
print "compressed message:", repr(compressed_message)
print "decompressed message:", repr(decompressed_message)
```

```
original: 'life of brian'
compressed message: 'x\234\313\311LKU\310OSH*\312L\314\003\000!\010\004\302'
decompressed message: 'life of brian'
```

The compression rate varies a lot, depending on the contents of the file.

Example: Using the zlib module to compress a group of files

```
# File:zlib-example-2.py

import zlib
import glob

for file in glob.glob("samples/*"):

    indata = open(file, "rb").read()
    outdata = zlib.compress(indata, zlib.Z_BEST_COMPRESSION)

    print file, len(indata), "=>", len(outdata),
    print "%d%%" % (len(outdata) * 100 / len(indata))
```

```
samples\sample.au 1676 => 1109 66%
samples\sample.gz 42 => 51 121%
samples\sample.htm 186 => 135 72%
samples\sample.ini 246 => 190 77%
samples\sample.jpg 4762 => 4632 97%
samples\sample.msg 450 => 275 61%
samples\sample.sgm 430 => 321 74%
samples\sample.tar 10240 => 125 1%
samples\sample.tgz 155 => 159 102%
samples\sample.txt 302 => 220 72%
samples\sample.wav 13260 => 10992 82%
```

You can also compress or decompress data on the fly:

Example: Using the zlib module to decompress streams

```
# File:zlib-example-3.py

import zlib

encoder = zlib.compressobj()

data = encoder.compress("life")
data = data + encoder.compress(" of ")
data = data + encoder.compress("brian")
data = data + encoder.flush()

print repr(data)
print repr(zlib.decompress(data))
```

```
'x\234\313\311LKU\310OSH*\312L\314\003\000!\010\004\302'
'life of brian'
```

To make it a bit more convenient to read a compressed file, you can wrap a decoder object in a file-like wrapper:

Example: Emulating a file object for compressed streams

```
# File:zlib-example-4.py

import zlib
import string, StringIO

class ZipInputStream:

    def __init__(self, file):
        self.file = file
        self.__rewind()

    def __rewind(self):
        self.zip = zlib.decompressobj()
        self.pos = 0 # position in zipped stream
        self.offset = 0 # position in unzipped stream
        self.data = ""

    def __fill(self, bytes):
        if self.zip:
            # read until we have enough bytes in the buffer
            while not bytes or len(self.data) < bytes:
                self.file.seek(self.pos)
                data = self.file.read(16384)
                if not data:
                    self.data = self.data + self.zip.flush()
                    self.zip = None # no more data
                    break
                self.pos = self.pos + len(data)
                self.data = self.data + self.zip.decompress(data)

    def seek(self, offset, whence=0):
        if whence == 0:
            position = offset
        elif whence == 1:
            position = self.offset + offset
        else:
            raise IOError, "Illegal argument"
        if position < self.offset:
            raise IOError, "Cannot seek backwards"

        # skip forward, in 16k blocks
        while position > self.offset:
            if not self.read(min(position - self.offset, 16384)):
                break

    def tell(self):
        return self.offset
    def read(self, bytes = 0):
        self.__fill(bytes)
        if bytes:
            data = self.data[:bytes]
            self.data = self.data[bytes:]
        else:
            data = self.data
            self.data = ""
        self.offset = self.offset + len(data)
        return data

    def readline(self):
        # make sure we have an entire line
        while self.zip and "\n" not in self.data:
            self.__fill(len(self.data) + 512)
        i = string.find(self.data, "\n") + 1
        if i <= 0:
            return self.read()
        return self.read(i)

    def readlines(self):
        lines = []
        while 1:
            s = self.readline()
            if not s:
                break
            lines.append(s)
        return lines

#
# try it out

data = open("samples/sample.txt").read()
data = zlib.compress(data)

file = ZipInputStream(StringIO.StringIO(data))
for line in file.readlines():
    print line[:-1]
```

> We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like indentation, rather than delimiters, might become feasible for expressing local structure in the source language.
>
> -- Donald E. Knuth, December 1974

The code module

This module provides a number of functions that can be used to emulate the behavior of the standard interpreter's interactive mode.

The **compile_command** behaves like the built-in **compile** function, but does some additional tests to make sure you pass it a complete Python statement.

In the following example, we're compiling a program line by line, executing the resulting code objects as soon as we manage to compile. The program looks like this:

```
a = (
  1,
  2,
  3
)
print a
```

Note that the tuple assignment cannot be properly compiled until we've reached the second parenthesis.

Example: Using the code module to compile statements

```
# File:code-example-1.py

import code
import string

#
SCRIPT = [
    "a = (",
    "  1,",
    "  2,",
    "  3 ",
    ")",
    "print a"
]

script = ""

for line in SCRIPT:
    script = script + line + "\n"
    co = code.compile_command(script, "<stdin>", "exec")
    if co:
        # got a complete statement.  execute it!
        print "-"*40
        print script,
        print "-"*40
        exec co
        script = ""
```

```
----------------------------------------
a = (
  1,
  2,
  3
)
----------------------------------------
----------------------------------------
print a
----------------------------------------
(1, 2, 3)
```

The **InteractiveConsole** class implements an interactive console, much like the one you get when you fire up the Python interpreter in interactive mode.

The console can be either active (it calls a function to get the next line) or passive (you call the **push** method when you have new data). The default is to use the built-in **raw_input** function. Overload the method with the same name if you prefer to use another input function.

Example: Using the code module to emulate the interactive interpreter

```
# File:code-example-2.py

import code

console = code.InteractiveConsole()
console.interact()
```

```
Python 1.5.2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
(InteractiveConsole)
>>> a = (
...     1,
...     2,
...     3
... )
>>> print a
(1, 2, 3)
```

The following script defines a function called **keyboard**. It allows you to hand control over to the interactive interpreter at any point in your program.

Example: Using the code module for simple debugging

```
# File:code-example-3.py

def keyboard(banner=None):
    import code, sys

    # use exception trick to pick up the current frame
    try:
        raise None
    except:
        frame = sys.exc_info()[2].tb_frame.f_back

    # evaluate commands in current namespace
    namespace = frame.f_globals.copy()
    namespace.update(frame.f_locals)

    code.interact(banner=banner, local=namespace)

def func():
    print "START"
    a = 10
    keyboard()
    print "END"

func()
```

```
START
Python 1.5.2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
(InteractiveConsole)
>>> print a
10
>>> print keyboard
<function keyboard at 9032c8>
^Z
END
```