# Python Grimoire

Revised: 2004-12-14



## Table of Contents



# 1 Introduction

> grimoire /grim**waar**/ (noun) ::= a book of magic spells and invocations. Origin: French, alteration of *grammaire* 'grammar'. A grammar is a description of a set of symbols and how to combine them to create well-formed sentences. A grimoire is a description of a set of magical symbols and how to combine them properly. It is sort of a recipe-book for magic spells. See also http://en.wikipedia.org/wiki/Grimoire

The *Python Grimoire* explains how to perform common programming tasks in Python. It is a good place to go after you've read the *Python Tutorial* and have a reasonable grasp of the basics of the language. In essence, the *Grimoire* is a collection of small recipes for very basic tasks. Its purpose is similar to that of the *Python Cookbook*, but the *Grimoire's* recipes are much simpler and more basic than those in the *Cookbook*. It does not cover advanced and/or specialized topics such as GUI toolkits and Jython.

The core of the *Grimoire* was originally developed and released by Andrew M. Kuchling in May, 1999. However, it never reached a stage where Andrew felt that it was ready for publication, and eventually he withdrew it. Steve Ferg, however, had found the *Grimoire* very helpful as he was learning Python, and he persuaded Andrew to allow him to take over maintenance of the document in August, 2002. In December 2004, Andrew and Steve released the Grimoire under the Creative Commons Attribution-NonCommercial-ShareAlike License 2.0 and Rui Carmo began hosting it at http://the.taoofmac.com .

The *Grimoire*, then, is an unpolished document. It may still contain occasional typos, formatting bloopers, code errors, and out-of-date material.

# 2 Numbers

## 2.1 Introduction

Python offers a variety of different numeric types: integers, which are fast but limited in size; long integers, which are slower but can be of arbitrary size; floating-point numbers; and complex numbers.

The simplest and most basic type are integers, which are represented as a C `long`. Their size is therefore

dependent on the platform you're using; on a 32-bit machine, they can range from -2147483647 to 2147483647. Python programs can determine the highest possible value for an integer by looking at `sys.maxint`; the lowest possible value will usually be `-sys.maxint - 1`.

Long integers are an arbitrary-precision integer type. They're slower than regular integers, but don't have the size limitations. Long integer literals have the letter "`L`" as a suffix; a lowercase "`l`" is also legal, but should be avoided because it's often difficult to distinguish a lowercase letter "`l`" from the numeral "`1`". If integers and long integers are combined in an expression, the regular integer is converted to a long integer automatically:

```
>>> 2L + 3
5L
>>> 5L ** 100
78886090522101180541172856528278622967320643510902300477027893066406 25L
```

Python's floating point numbers are represented as C's `double` type, and behave pretty much as you'd expect:

```
>>> 5.4 / 2.3
2.34782608696
>>> 6.4 * 2
12.8
>>> 7.5 % 5
2.5
```

In operations that mix floating-point numbers and integers, the integers will be converted to floating-point before computing the result, which will be left as a float.

Python also supports a complex number type. (It's possible to compile the Python interpreter without complex numbers, but few people bother to do so; saving 20K of code isn't that important.) Complex numbers are represented by a floating point constant with a suffix of "`j`"or "`J`" suffix, "`j`" being used to represent the square root of -1 in many engineering contexts.

```
>>> 1j * 1J
(-1+0j)
>>> (3 - 7j) + (2 + 1j)
(5-6j)
>>> (1 + 3j) / (2 - 4j)
(-0.5+0.5j)
```

Note that an expression like `1 + 3j` isn't a constant, but an expression that adds two constants: `1` and `3j`. This is a minor semantic quibble, whose major significance is that you should make liberal use of parentheses in complex expressions. `1+3j / 2-4j` is not the same as `(1+3j) / (2-4j)`, because the first expression will evaluate `3j/2` first.

## 2.2 Mathematical Functions

You want to perform a calculation that requires operations beyond the basic ones of addition, subtraction, multiplication, division, and exponentiation. For example, you need to evaluate a sine or cosine.

Import the `math` module, and use the functions that it provides.

```
import math

angle = math.pi / 6
x = math.cos( angle )
y = math.sin( angle )
```

If you want to get results that are complex numbers, use the `cmath` module.

```
import cmath
result = cmath.sin( 1+3j )
```

**Discussion:**

Mathematical functions such as sine and cosine are contained in the `math` module. The functions are identical to those defined by the C standard, since they're simply Python wrappers on top of those functions. The `math` module also defines two constants: `pi`and `e`.

The functions in `math` always return floating point numbers, whether the input was an integer or a float. If a result would be a complex number (`math.sqrt(-1)`, the square root of -1, is the common example), this is considered an error and triggers an exception. To get complex numbers as answers, you need to use the `cmath` module, which is similar (though not identical) to the `math` module, and can handle complex parameters and results:

```
>>> math.sqrt(-1)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OverflowError: math range error
>>> cmath.sqrt(-1)
1j
```

Here's a list of the functions contained in the `math` module. The `cmath` module defines analogous functions that operate on complex numbers.

| Name | Purpose |
|---|---|
| `acos(x)` | Arc cosine of *x* |
| `asin(x)` | Arc sine of *x* |
| `atan(x)` | Arc tangent of *x* |
| `atan2(x,y)` | Arc tangent of *x/y* |
| `ceil(x)` | Ceiling of *x*; the largest integer equal to or greater than *x* |
| `cos(x)` | Cosine of *x* |
| `cosh(x)` | Hyperbolic cosine of *x* |
| `exp(x)` | *e* raised to the power of *x* |
| `fabs(x)` | Absolute value of *x* |
| `floor(x)` | Floor of *x*; the largest integer equal to or less than *x* |
| `fmod(x)` | *x* modulo *y* |
| `frexp(x)` | The matissa and exponent for *x*. |
| `hypot(x, y)` | Euclidean distance, `sqrt(x*x + y*y)` |
| `ldexp(x, i)` | `x * (2**i)` |
| `log(x)` | Natural logarithm of *x* |
| `log10(x)` | Base 10 logarithm of *x* |
| `modf(x)` | Return the fractional and integer parts of *x* |
| `pow(x, y)` | *x* raised to the power of *y* |
| `sin(x)` | Sine of *x* |
| `sinh(x)` | Hyperbolic sine of *x* |
| `sqrt(x)` | Square root of *x* |
| `tan(x)` | Tangent of *x* |
| `tanh(x)` | Hyperbolic tangent of *x* |

## 2.3 Generating Random Numbers

You need to generate random numbers for an application, such as generating test data or selecting an item from a list.

The standard library module `random` implements a random number generator. Using it is simple:

```
import random

# Generate a random number between 0 and 1
randval = random.random()

# Generate a random integer between 45 and 55
randval = random.randint(45, 55)

# Choose a random value from a list
randname = random.choice( ["Glenna", "Jody", "Natalie"] )
```

**Discussion:**

`random.random()` returns a random floating point number in the range [0, 1), generated using a Wichmann-Hill random number generator. There are also other specialized generators in this module, built on top of the `random()` function:

- `randint(a, b)` chooses an integer in the range [*a*, *b*)
- `choice(S)` chooses a random item from the given sequence *S*
- `uniform(a, b)` chooses a floating point number in the range [*a*, *b*)

To specify the random number generator's initial setting, use `random.seed(x, y, z)`, which sets the seed from three integers in the range [1, 256) There's also a `random` class, which you can instantiate to create independent multiple random number generators.

The `random` module contains functions that approximate various standard distributions, such as a normal or Gaussian distribution, lognormal, gamma and beta distributions, and a few others. These functions are specialized enough to make me refer you to the Python Library Reference for the details.

Note that the `rand` module, which is occasionally used in some examples and demo programs, is now considered obsolete. Use `random` instead.

The Wichman-Hill random number generator is described in: "An efficient and portable pseudo-random number generator (Algorithm AS 183)", Wichmann, B. A. & Hill, I. D., *Applied Statistics* 31 (1982) 188-190.

## 2.4 Generating a Random Permutation

A related problem is generating a random permutation of numbers between 1 and N, with no repeats. For example, you might want to rearrange a list into a random order.

The solution is to generate a list containing the numbers from 1 to N, and use the `random.choice()` function to pick a random element. Remove the chosen element, and then repeat until the list is empty.

```
numbers = range(1, N+1)
while numbers:
    j = random.choice(numbers)
    numbers.remove(j)
    print j
```

## 2.5 Converting Numbers to and from Binary Data

**Problem:**

You need to convert an integer value v to a 2-byte or 4-byte string representation. For example, the number 1 would be the bytes 0,1; 1956 would be 164, 7 because 1956 = 7 * 256 + 164. This is often needed when reading or writing binary data files, or implementing network protocols.

**Solution:**

You can write code to perform the conversion by taking the low byte of `v`, shifting v down 8 bits and taking the low byte of the result, and so forth. However, the code is long and relatively slow:

```
s = ( chr( (V)       & 255 ) +
      chr( (V >> 8)  & 255 ) +
      chr( (V >> 16) & 255 ) +
      chr( (V >> 24) & 255 ) )
```

A faster and simpler solution uses the `struct` module, which is intended for such conversions:

```
import struct
s = struct.pack('i', V )
```

To reverse the conversion, and go from a binary string to a tuple of integers, use `struct.unpack()`:

```
byte, short = struct.unpack('BH', '\377\000\377\377')
```

**Discussion:**

You're not limited to only packing a single value at time. Instead, the `pack()` function in the `struct` module takes a format string containing several characters, along with the values to be packed:

```
>>> struct.pack('iii',  127, 1972, 1234567890)
'\177\000\000\000\264\007\000\000\322\002\226I'
```

Repeated format characters can be written as a single character preceded by an integer repeat count; the previous example could have been written as `struct.pack('3i', ...)`, and would produce identical results.

Different-sized values can be packed using several different format characters, whose output will vary in size. For example, the "`b`" (byte) character will produce a single byte of output, and therefore is limited to values between -128 and 127. "`h`" (short) produces 2 bytes, "`i`" (int) produces 4 bytes, and "`l`" (long) will be 4 bytes on 32-bit machines, and 8 bytes on most 64-bit machines.

Here's a table of some of the more commonly used format characters supported by `pack()` and `unpack()`. Consult the Library Reference's section on the `struct` module for a complete list:

| Format | C Type | Python |
|---|---|---|
| b | signed char | integer |
| B | unsigned char | integer |
| h | short | integer |
| i | int | integer |
| l | long | integer |
| f | float | float |

Because the `struct` module is often used for packing and unpacking C structs, it obeys the compiler's

alignment rules. This means that padding bytes may be inserted between format characters of different types. For example, the zero byte in the following example is padding:

```
>>> struct.pack('BH',  255, 65535)
'\377\000\377\377'
```

# 3 Strings

## 3.1 Introduction

Relatively few tasks can be performed by only dealing with numbers; programs will usually print out reports, modify the contents of text files, parse HTML and XML documents, and perform other operations on strings of characters. Strings are therefore an important data type in modern programming languages, and Python is no exception.

In the source code for a Python program, strings can be written in several ways. They can be surrounded by either single or double quotes:

```
if version[:5] != 'HTTP/':
    send_error(400, "Bad request version(%s)" % `version`)
```

Strings are one of Python's sequence types. This means that strings can be sliced, and the `for` statement can be used to iterate over the individual characters in a string:

```
>>> s = "Hello, world"
>>> s[0:5]
'Hello'
>>> s[-5:]
'world'
>>> for char in s:
...   print char,
...
H e l l o ,   w o r l d
```

Strings are immutable; once a string has been created, whether as a literal in a program's source code or in the course of a program's operation, you can't modify the string in-place. Trying to change the first character of a string by slicing, as in the code `s[0] = 'Y'`, fails; to create a modified version of the string, you have to assemble a whole new string with code like `s = 'Y' + s[1:]`. This means that Python has to make temporary copies of the string, which will be slow and memory-consuming if you're modifying very large strings. See the section on mutable strings to learn how to achieve some of the effect of a mutable string type.

## 3.2 Converting from Numbers to Strings

You wish to convert a number to the corresponding string representation. For example, you want to convert a number like 144 to the string `'144'`.

The simplest way is to use the built-in function `str()`, or the backquote notation `` `144` ``. Both solutions will convert the number to a string in the most straightforward way.

```
V = 187.5
S = str(V)  # Produces '187.5'
```

If v contains an integer or long integer, and you want to convert the number to hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`.

```
>>> hex(187)
'0xbb'
>>> oct(187)
'0273'
```

If fancier formatting is required, such as rounding a floating-point number to a given number of decimal places, or padding a number with zeros, use the % operator on strings, which allows more precise control of the output.

```
>>> "%04d" % (87,)
'0087'
>>> "%.2f" % (187.375,)
'187.38'
```

(See the Library Reference manual for the complete details of the available format characters.)

There's no library function to convert a number to its base-2 representation, so here's a function to do the job.

```
def atob(number):
    """Returns a string containing the binary representation of a number"""
    if number < 0:
        prefix = "-" ; number = -number
    elif number == 0: return "0"
    else:
        prefix = ""

    # Loop, looking at the lowest bit of the number an
    s = ""
    while number > 0:
        s = chr( 48+ (number & 1) ) + s
        number = number >> 1
    return prefix + s
```

## 3.3 Converting from Strings to Numbers

Now you wish to perform the opposite of the conversion in the previous section; that is, you have a string such as `'533'` and wish to convert it to the number 533.

The built-in functions `int()`, `long()`, and `float()` can perform this conversion from string to a numeric type:

```
>>> int( '533' )
533
>>> long( '37778931862957161709568' )
37778931862957161709568L
>>> float( '45e-3'), float('12.75')
(0.045, 12.75)
```

Note that these functions are restricted to decimal interpretation, so that `int('0144')` will return a result of 144, and `int('0x144')` will raise a `ValueError` exception.

To support different bases, or for use with versions of Python earlier than 1.5, the `string` module contains the functions `atoi()`, `atol()`, and `atof()`, which convert from ASCII to integer, long integer, or floating point, respectively.

```
>>> import string
>>> string.atoi('45')
```

```
>>> string.atol( '37778931862957161709568' )
3777893186295716170956 8L
>>> string.atof('187.54')
187.54
```

`atoi()` and `atol()` have an optional second argument which can be used to specify the base for the conversion. If the specified base is 0, the functions will follow Python's rules for integer constants: a leading "`0`" will cause the string to be interpreted as an octal number, and a leading "`0x`" will cause base-16 to be used.

```
>>> string.atoi('255')
255
>>> string.atoi('255', 16) # 255 hex = 597 decimal
597
>>> string.atoi('0255', 0) # Assumed to be octal
173
>>> string.atol('0x4000000000000000000', 0)
302231454903657293676544L
```

While you could use the built-in function `eval()` instead of the above functions, this is not recommended, because someone could pass you a Python expression that might have unwanted side effects. For example, a caller might pass a string containing a call to `os.system()` to execute an arbitrary command; this is a serious danger for applications such as CGI scripts that need to handle data from an unknown source. `eval()` will also be slower than a more specialized conversion operation.

## 3.4 Splitting Strings into Equal Sections

**Problem:**

You wish to split a string into *N* equally-sized parts. The last part might be smaller than the previous ones when the length of the string isn't an exact multiple of *N*. For example, dividing the 9-character string `'Woodhenge'` into 2 parts would result in the list `['Woodh', 'enge']`, containing a 5-character and a 4-character string.

**Solution:**

The following function takes a string *S* and a number *N*, and returns an *N*-element list containing the different sections of the string. The function works by taking the length of the string and computing how long the sections must be. The list is then constructed by looping over the string and extracting each section by slicing the string.

```
def equal_split(S, N):
    """Split up the string S into N parts, returning a list containing
    the parts.  The last part may be smaller than the others."""
    part = (len(S) + N - 1)  /N
    L = []
    for i in range(0, N):
        L.append( S[part*i : part*i+part] )
    return L
```

`equal_split('this is a test', 3)` will return `['this ', 'is a ', 'test']`.

## 3.5 Searching and Replacing

**Problem:**

Given a string, you wish to replace all the occurrences of one substring with different text. For example, you wish to replace all occurrences of the string `'USER-NAME'` with the string `'Joseph Addison'` in the string contained in the variable `data`.

**Solution:**

For simple string substitutions, the `string.replace` function will be the simplest and fastest solution.

```
import string
newdata = string.replace(data, 'USER-NAME', 'Joseph Addison')
```

If `data` contains `"USER-NAME has been added to the list."`, then after the above line is executed, `newdata` will contain `'Joseph Addison has been added to the list.'`.

**Discussion:**

String replacement is a common operation, and there are several ways to do it. The fastest and simplest way is the `string.replace()` function, which can only replace a fixed string with another fixed string. For example, the following line replaces the Latin1 character 219, a capital U with a circumflex accent (), with the HTML entity `&Ucirc;`.

```
newdata = string.replace(data, chr(219), '&Ucirc;')
```

The `chr()` built-in function takes an integer between 0 and 255, and returns a string of length 1 that contains the character with that byte value.

Multiple replacements will require multiple calls to `string.replace()`:

```
newdata = string.replace(data, chr(219), '&Ucirc;')
newdata = string.replace(data, chr(233), '&eacute;')
```

The replacement string is fixed, so it can't be varied depending on the string that was matched. For cases that require matching variable strings, such as "match anything between square brackets", or that require varying the replacement string, you'll have to use regular expression matching, available through the built-in `re` module.

The following example uses regular expressions to replace URLs in a string with the HTML for a link to that URL, with the URL as the link text.

```
import re

data = """http://www.freshmeat.net
  ftp://ftp.python.org/pub/python/src/
"""

newdata = re.sub(r"""(?x)
 (  # Start of group 1
  (http|ftp|mailto)  # URL scheme
  :                  # Separating colon
  \S+                # Everything up to the next whitespace character
 )  # End of group 1
""", r'<a href="\g<1>">\g<1></a>', data)
```

Notice that the order of the arguments to `re.sub` is different from the arguments for `string.replace()`. `string.replace()` takes the arguments (*string*, *substring*, *replacement*), while `re.sub()` takes the arguments (*pattern*, *replacement*, *string*). `re.sub()` has a different ordering of its arguments for consistency with the other functions in the `re` module; the regular expression pattern is viewed as the most important argument, so it's always passed as the first argument.

Regular expression patterns have a complicated syntax. Let's dissect the above pattern into its

components.

- `(?x)` Specifies that this pattern is expressed in verbose mode. Most whitespace will be ignored, so you can format the pattern neatly, and comments can be embedded in the pattern by preceding them with a "#".

- `(http|ftp|mailto)` The parenthesized group lists several alternative strings, separated by "|" characters. Any one of these strings can produce a successful match for this component of the pattern.

- `\S+` The `\S` special sequence matches any character that isn't a whitespace character. Whitespace characters are space " ", tab "`\t`",newline "`\n`", carriage return "`\r`", form feed "`\f`", and vertical tab "`\v`". The "+" character is a qualifier that specifies how many times the previous component should be repeated; "+" indicates that the `\S` should be repeated one or more times.

The replacement string can contain sequences which will contain pieces of the matching string. For example, `\g<1>` will be replaced by the contents of the first group, which in this case will contain the whole URL that matches the regular expression. The replacement string is therefore `'<a href="\g<1>">\g<1></a>'`, which will contain the text of the URL at two different places, along with the required HTML.

The `re` module can perform simple substitutions that are equally possible with `string.replace()`. This is simply a matter of writing a pattern that doesn't use any of the regular expression syntax:

```
data = re.sub('USER-NAME', 'Joseph Addison', data)
```

Try not to use regular expressions when a simpler fixed-string replacement can do the job, because you're paying a speed penalty for the extra generality of regular expressions.

For much more information on regular expressions in Python, consult the Regular Expression HOWTO, at http://www.python.org/doc/howto/regex/.

## 3.6 Parsing Input

You wish to parse some sort of structured input, such as a configuration file or some data file.

There are many possible ways to go about this.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using `string.split()`, and to subsequently convert decimal strings to numeric values using `string.atoi()`, `string.atol()` or `string.atof()` If you want to use a delimiter other than whitespace, `string.split()` can handle that, and can be combined with `string.strip()` which removes surrounding whitespace from a string.

For more complicated input parsing, the `re` module's regular expressions are better suited for the task, and are more powerful than C's `sscanf()`.

**Discussion:**

A parser for configuration files is included as the `ConfigParser` module in the standard library; you should take a look at it and see if it meets your needs.

Python programmers often choose to decree that software configuration files should written in RFC-822 format, specifying names and their corresponding values; this allows using the parser in the `rfc822` module to read the files. A sample configuration file might then look like this:

```
Title: Index to Python Information
Description: Python code, information, and documentation.
Keywords: Python, Python articles, Python documentation
Palette: gold
Sidebar: none
```

There's a contributed module that emulates `sscanf()`, written by Steve Clift and availabe from the Contributed Software section on http://www.python.org/. See http://www.python.org/ftp/python/contrib-09-Dec-1999/Misc/sscanfmodule.c.Z for the module.

If you're trying to parse some sort of well-known file format, it's possible that a Python module has already been written to deal with it. Some common cases are:

- The Python Imaging Library (http://www.pythonware.com/) can read many different graphics formats, ranging from well-known ones such as GIF and JPEG, to more specialized formats such as DCX and TIFF.

- Support for many scientific file formats has been implemented for use with Numeric Python; consult the Scientific Computing topic guide at http://www.python.org/topics/scicomp/ for more information.

- The XML topic guide tracks the available software for processing XML with python; see http://www.python.org/topics/xml/.

Before spending a lot of effort implementing a module for a new file format, do some research first and see if someone has already done it; you might save yourself a lot of work.

## 3.7 Mutable Strings

**Problem:**

Python's strings are immutable, so you can't change a single character of the string without constructing a new string. Sometimes this is a problem, particularly when dealing with very large strings; constructing a new string will require copying most of the original string, and can be slow.

**Solution:**

Use the `array` module, which provides a mutable array data type that can only hold values of a single type. If used to hold only characters, an array behaves similarly to a mutable string.

```
import array

A = array.array('c')
A.fromstring('hello there!')
print A
```

This prints:

```
array('c', 'hello there!')
```

The array object `A` is mutable, so you can modify an element in place:

```
A[0] = 'j'
print A
```

This will print:

```
array('c', 'jello there!')
```

Most functions that require strings, such as `string.split()`, won't accept array objects, so you'll have to convert the array to a string in order to pass the data to certain functions:

```
print string.split( A.tostring() )
```

# 4 Tuples and Lists

## 4.1 Introduction

Python's lists and tuples are the simplest compound data types. They can contain any number of other Python objects, no matter what the type of the subobjects. One tuple or list can contain numbers, strings, other tuples and lists, class instances, or any other data type that Python can handle.

Tuple constants are written as a sequence of values separated by commas; optionally they can be surrounded by parentheses. A zero-length tuple is written as an empty pair of parentheses `()`, and a tuple of length 1 is indicicated by a trailing comma, to distinguish a one-element tuple from an expression..

```
1,2,3
('subject', 'from', 'date', 'to')
()        # Empty tuple
(1,)      # Single-element tuple
2,        # Another single-element tuple
2         # The number 2 (no trailing ',')
```

Lists are always surrounded by square brackets:

```
[]        # Empty list
[1]       # List of length 1
[1,2,3]   # List of length 3
["this", 1, "is", "a", "list"]
```

You can retrieve the *i*th element in a list or tuple by indexing it: `a[i]`. Retrieving a single element is a fast operation, and is indepent of the size of the list/tuple or the value of *i*. A smaller list or tuple can be retrieved by slicing: `a[i:j]` returns a sublist or subtuple containing elements *i* up to *j*-1.

```
>>> L = [0,1,2,3,4,5,6]
>>> L[0:2], L[3:5], L[0:-1]
([0, 1], [3, 4], [0, 1, 2, 3, 4, 5])
```

If an index is a negative number, then the length of the sequence is added to it. This means that `L[-1]` is the last element of the sequence `L`, L[-2] is the next-to-last element, and so forth. You can mix positive and negative indexes as you like. Omitting *i*, the starting index of a slice, will default to 0, and omitting the ending index *j* defaults to the length of the sequence.

```
>>> L[:2], L[-2:], L[:-1]
([0, 1], [5, 6], [0, 1, 2, 3, 4, 5])
```

Tuples and lists are similar in many ways. The most important difference between them is that lists are mutable, or changeable objects; it's possible to replace any given list element, or to change the list's length. By way of contrast, once a tuple has been created it's not possible to change it in any way.

To modify a list, you can simply assign to an element or a slice, or use the `del` statement.

```
>>> L=[1,2,3]
>>> L[ -1 ] = 7 ; print L
[1, 2, 7]
>>> L[0:2] = ['a', 'b', 'c'] ; print L
['a', 'b', 'c', 7]
>>> del L[1] ; print L
['a', 'c', 7]
```

Because tuples and lists are so similar, you might wonder why Python supports them both. Why not only have lists, since they're more flexible than tuples?

Immutable tuples are useful in situations where you need to pass a few items to a function and don't want the function to modify the tuple; for example,

```
point1 = (120, 140)
point2 = (200, 300)
record(point1, point2)
```

You don't want to have to think about what would happen if the `record()` function changed the coordinates -- it can't, because tuples are immutable. In the section on dictionaries, you'll learn that tuples can be used as dictionary keys, while lists can't.

On the other hand, the mutability of lists makes it easier to change them, so they're useful when you're constructing a collection of objects over time, or when objects need to be freely removed and added. For example, if you're iterating through the lines of text in a file and saving lines that match a certain criterion such as a regular expression, a list is the natural data type for storing the matching lines. Adding a new line is fast -- just call the list's `append()` method -- while building a new tuple by concatentation (`tuple1 + (line,)`) would be slow since it copies the entire tuple every time.

When to use a list or a tuple is ultimately a stylistic question, and you'll arrive at your own opinions about it. Personally, I generally use tuples to hold a collection of dissimilar data; for example, a function might return a numeric error code and a string describing an error as a tuple (*`code, errmsg`*). In this style, tuples wind up being used like Cs structures or Pascal's records. A function that returned a number of similiar objects, such as lines from a text file, would return a list. In other words, while lists can contain data of different types, I tend to only use them to contain similar objects: all strings, all numbers, all instances of the same class, or whatever. (On the other hand, I don't follow this rigidly; if I had some different data types collected together and sometimes needed to change only one of them, I'd probably use a list instead of a tuple, even though the data in that list wouldn't be homogeneous.)

## 4.2 Iterating Over a List

You have a sequence, and you wish to loop over its contents.

This is the purpose of the `for` statement:

```
for item in [1,2,3,4,5]:
        print item
```

This will work for tuples and strings, too.

**Discussion:**

To iterate over a list in reverse order, the fastest solution is to simply reverse the list:

```
List.reverse()
for item in list:
    # ... do something with item ...
```

Here's another way to iterate backwards when you have some other sequence type like a tuple or a string,

or if reversing the list is unacceptable to you. You'll have to loop over the indices, from the length of the sequence minus one, down to zero.

```
for i in range(len(seq)-1, -1, -1):
    item = seq[i]
    # ... do something with item ...
```

## 4.3 Removing Duplicates from a List

You have a list, and wish to ensure that each item in the list is unique, only occurring once. For example, if your list is `[1,4,5,2,4]`, you would want to remove one of the 4's.

If you can reorder the list, sort it to put duplicate values next to each other. If the list is sorted, then removing duplicates requires just a single loop. Note that the opening test for the length of the list is required - without it, if the list is empty, this code will raise an exception.

```
if len(List) > 1:
    List.sort()         # Sort the list

    # Walk from the end of the list to the beginning;
    # if two elements are identical, delete one of them.
    last = List[-1]     # Start with the last element

    for i in range(len(List)-2, -1, -1):
        if last==List[i]:
            del List[i]
        else:
            last=List[i]
```

If all the elements of the list can be used as dictionary keys, you can create a dictionary with the list elements as keys, and take the result of the dictionary's `keys()` method as the new list.

```
d = {}
for x in List: d[x] = x
List = d.keys()
```

## 4.4 Converting Between Tuples and Lists

You have a tuple, and want to turn it into a list. Or, the opposite problem: a list that you want to turn into a tuple.

The built-in functions `tuple()` and `list()` convert any sequence type into a tuple or list containing the same items in the same order.

```
>>> tuple( [1,2,3] )
(1, 2, 3)
>>> list ( ('monty', "python's", "fliegende", "circus") )
['monty', "python's", 'fliegende', 'circus']
>>> tuple('Dr Gumby!')
('D', 'r', ' ', 'G', 'u', 'm', 'b', 'y', '!')
```

**Discussion:**

`list()` was introduced in Python 1.5; in previous versions, the following idiom was used to convert things to lists:

```
list = map(None, (1,2,3,4) )
```

Here's how the above line of code works: the `map()` built-in function is used to evaluate a function for every value in a sequence type like a tuple, and returns a list containing the function's output for each value in the input sequence. Put another way, `map(F, seq)` returns a new list containing `[F( seq[0] ), F( seq[1] ), ... ]`. If `None` is passed as the function `F`, then `map()` simply returns the elements of the sequence `[seq[0], seq[1], ...]`.

You may still see the above use of `map()` in programs that attempt to be compatible with older versions of Python; if you don't care about backward compatibility, simply use `list()` which is clearer.

## 4.5 Converting a String to a Tuple

You want to convert a tuple in a string into the corresponding tuple object; such a `str2tuple()` function would be used like this:

```
>>> myString = "(12, '  abc  '  ,   '  b,  c', 'd\\'e')"
>>>print str2tuple(myString)
(12, 1.0, "'XX'", '  abc  ', '  b,  c', "d'e", "'f")
```

The following `str2tuple()` function will do the job.

```
def str2tuple(s):
    return eval(s, {'__builtins__': {}})
```

**Discussion:**

The above function uses the `eval()` built-in function, along with a special trick. `eval()` takes a string and evaluates the contents as a Python expression, returning the value of the expression. One or two additional parameters can be supplied that must be dictionaries that will be used as the table of global and local variables, respectively.

The `str2tuple()` function uses `eval()` and provides a dictionary containing a single key-value pair, mapping the key "`__builtins__`" to an empty dictionary. "`__builtins__`" is a special name internal to Python,

The net effect of this is to evaluate the string in a highly restricted environment. This denies access to Python's standard built-in functions such as `open()`, and thus cannot access anything that isn't hardcoded in the language. For example:

```
>>> str2value(' (1,"test",10.2)')
(1, 'test', 10.2)
>>> str2value('11+13')
24
>>> str2value('open("/etc/passwd")')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in str2value
  File "<string>", line 0, in ?
NameError: open
>>>
```

Notice that `str2value()` can accept expressions such as "`11+13`" thanks to its use of `eval()`, returning an integer value. If you really want to make sure that the result is a tuple, you must use the `type()` built-in function to determine the type of the result and compare it with the type object for tuples.

```
v = str2value( text )
if type(v) is not type( () ):
    raise ValueError, "Text %s does not contain a tuple" % (text,)
```

A clearer way to do this is to import the `types` module, which contains predefined variables for the most common Python types, and compare the type of the result to `types.TupleType`.

```
if type(v) is types.TupleType:
    raise ValueError, "Text %s does not contain a tuple" % (text,)
```

## 4.6 Multidimensional Lists

You wish to create a list of lists.

This obvious solution for creating a list of *N* lists is wrong; see the Discussion for an explanation.

```
LL = [ [] ] * N
```

Instead, build an *N*-element list first, and then replace each element with a list.

```
LL = [None] * N
for i in range(N): LL[i] = []
```

If you want to fill out the contained lists with default values, simply replace `[]` with an expression like `N*[0]`. Elements could then be retrieved by accessing `LL[i][j]`.

**Discussion:**

Why doesn't `[ [] ] * N` work as expected? When multiplying a list, Python copies the contents of the list by reference. This means that you'll wind up with a list containing *N* references to the same objects, which is fine for immutable objects such as floating point numbers or tuples, but can be confusing with mutable objects such as lists. To make this clearer, consider the list constructed for `N=5`. What happens when you append an element to the first list?

```
>>> LL = [ [] ] * N
>>> print LL
[[], [], [], [], []]
>>> LL[0].append(1)
>>> LL
[[1], [1], [1], [1], [1]]
```

All of the lists are modified, not just the first one! The `is` operator will tell you that all the elements of `LL` are actually the same list; a true value is returned by `LL[0] is LL[1]`.

A common use of multi-dimensional lists is to store matrices. Before reinventing the wheel and writing yet another matrix type, take a careful look at Numeric Python (http://www.python.org/topics/scicomp), which includes a matrix type, interfaces to libraries of functions such as LINPACK, and lots of other modules useful for numeric work.

# 5 Dictionaries

## 5.1 Introduction

Dictionaries are associative data structures, storing key/value pairs. Given a key, the corresponding value can be retrieved. For example, the following dictionary maps English colour names to French ones.

```
fr_colours = { 'red': 'rouge', 'blue': 'bleu',
               'black': 'noir', 'white': 'blanc',
               'yellow': 'jaune'}
```

To retrieve the value corresponding to a given key, the notation is the same as when retrieving an element from a list or tuple. When indexing an array, you can only For example, to get the French word for 'white', you would write `fr_colours['white']` in Python code.

A dictionary can use any immutable Python value as the key, and there are no restrictions on what you can store as values. That means that keys can be integers, floating point numbers, strings, or tuples; lists are the most important type that *can't* be used as keys.

Dictionaries are implemented as resizable hash tables. Hash tables are data structures optimized for quick retrieval; the result of a semi-random hash function is computed for each key, and the value is stored in a bin given by the hash function's result. Retrieving a key is then a matter of calculating the hash function and looking in the corresponding bin, so retrieval always takes the same amount of time no matter how large the dictionary is. (At least, this is true unless you have a pathological set of keys that all hash to the same value, or to a very small set of values. This is highly unlikely, unless you deliberately engineer your keys to beat the hash function.)

The hash table implementation of dictionaries explains the reason for the restriction to immutable keys. If the key were mutable like a list, it could be changed in-place which would also modify the hash value. The value would then no longer be stored in the right bin, and lookups would incorrectly fail.

## 5.2 Checking Whether A Key is in a Dictionary

You wish to check whether a given key is stored in a dictionary. For example, you wish to know if the key `'green'` is in the `fr_colours` dictionary.

There are two ways of doing this. The `has_key()` dictionary method returns true or false depending on whether the key is in the dictionary or not.

```
>>> fr_colours.has_key('blue')
1
>>> fr_colours.has_key('purple')
0
```

If a dictionary retrieval fails, Python raises a `KeyError` exception. You can catch this exception and do something appropriate in the exception handler.

```
try:
    fcolr = fr_colours[ ecolr ]
except KeyError:
    fcolr = "<Untranslated colour %s>" % (ecolr,)
```

**Discussion:**

Deciding whether to use `has_key()` or catch the exception is a question of both style and speed. Either way is fine from a stylistic point of view, though you may have a preference for one or the other. For example, I rarely catch the `KeyError` exception, preferring to write `if dict.has_key(key): ...` before a block of code which can then assume that *key* is present. You can do whatever you like.

The comparative speed of the two solutions depends on how often you expect the key to be present. Raising an exception and then catching it is slower than a call to the `has_key()` method. Consider a program where you're looping over 100,000 keys, and doing something different depending on whether the key is present or not. If you expect only a few keys to be missing, the exception will almost never be raised, while calling the `has_key()` method would a little bit of time on every pass through the loop. In this case, catching the infrequent exceptions will be faster.

On the other hand, if 50,000 or 90,000 of the keys won't be present, the overhead of raising exceptions and catching them will be larger because an exception will have to be raised more often, so checking the result of `has_key()` is probably a better idea. The exact crossover point will vary a bit; in a presentation at the sixth Python conference, Barry Warsaw found it to be around 5 or 10 percent. See http://barry.warsaw.us/papers/ecshort.pdf.

## 5.3 Iterating Over a Dictionary's Contents

After creating a dictionary, you want to loop over the dictionary's contents and do something for each entry, such as printing them.

The `keys()` method of dictionaries returns a list of the dictionary's keys, arranged in essentially random order:

```
>>> for colr in fr_colours.keys(): print colr
...
yellow
red
black
white
blue
```

If you want to loop over the contents in some sorted order, you'll have to retrieve the list of keys and sort it before looping through it.

```
>>> keylist = fr_colours.keys() ; keylist.sort()
>>> for colr in keylist: print colr
...
black
blue
red
white
yellow
```

Remember that the `sort()` list method sorts a list in-place and returns `None`. `for colr in fr_colours.keys().sort()`will not work, because the value of `fr_colours.keys().sort()` is `None`.

**Discussion:**

The list returned by `keys()` is arranged randomly, because the key/value pairs are scattered across random bins in a hash table. Dictionaries therefore don't impose any ordering on their contents. For example, if you assemble a dictionary element-by-element, and print the dictionary after each new key/value pair is added, there's no order to how the contents are displayed.

```
>>> d = {} ; d['red'] = 'rouge' ; print d
{'red': 'rouge'}
>>> d['blue'] = 'bleu' ; print d
{'red': 'rouge', 'blue': 'bleu'}
>>> d['yellow'] = 'jaune' ; print d
{'yellow': 'jaune', 'red': 'rouge', 'blue': 'bleu'}
>>> d['green'] = 'vert' ; print d
{'yellow': 'jaune', 'red': 'rouge', 'green': 'vert', 'blue': 'bleu'}
```

There are two other methods available for retrieving the contents of a dictionary. If your loop will require each key and its corresponding value, you can use the `items()` method, which returns a list of `(key, value)` 2-tuples. Again, the list contents are arranged in a random order.

```
>>> for english, french in fr_colours.items():
...     print english, '\t', french
...
yellow  jaune
red     rouge
black   noir
white   blanc
blue    bleu
```

Least commonly used of all, the `values()` method returns a list containing only the values from a dictionary.

```
>>> for colr in fr_colours.values(): print colr
...
jaune
rouge
noir
blanc
bleu
```

The way to keep these method names straight is to remember that dictionaries map from keys to values; the `keys()` and `values()` methods return a list of the corresponding elements of the dictionary. That leaves `items()` to return a list of 2-tuples.

## 5.4 Merging Dictionaries

You have two dictionaries `d1` and `d2`. You wish to add the key/value pairs from `d2` to `d1`.

The `update` method of a dictionary adds the contents of another dictionary. To solve this problem, you can simply do: `d1.update(d2)`.

`d1.update(d2)` is equivalent to the following code:

```
for key, value in d2.items():
    d1[ key ] = value
```

If the same key is associated with a value in both `d1` and `d2`, `update()` will replace the value in `d1` with the value from `d2`. `d2` is never modified by this method.

## 5.5 Multiple-valued Dictionaries

You need a dictionary that can store the key and more than one value. For example, after `d[key] = value1` and `d[key] = value2`, you want `d[key]` to return a list containing `[value1, value2]`. This differs from an ordinary dictionary, where assigning to the same key twice causes the first value to be deleted and replaced by the second.

The following class stores lists of values; setting a new value causes it to be appended to the list.

```
import UserDict
class MultipleDict(UserDict.UserDict):
    def __setitem__(self, key, item):
        if self.data.has_key( key ):
            self.data[key].append( item )
        else:
            self.data[key] = [item]
```

Using the class is simple:

```
d = MultipleDict()
```

```
d[1] = 'a'
d[1] = 'b'
d[1] = 'c'
```

`d[1]` will return `['a', 'b', 'c']`. To remove the key, you'll have to delete the entire list of values with `del d[1]`.

This class subclasses the `UserDict` class from the `UserDict` module, which implements all the methods of dictionaries in a Python class, and stores its contents in the `data` attribute, which is a real dictionary. To implement the desired change in behaviour, we only have to override the `__setitem__` method, and store the right result in `self.data`.

# 6 Functions

## 6.1 Introduction

In Python, functions are defined using the `def` statement. The following example defines a function called `get_word()`. `get_word()` will take two parameters: a string *sentence* and a number *N*, and returns the N'th word of *sentence*.

```
import string

def get_word(sentence, N):
    words = string.split(sentence)
    return words[N]
```

The `return` statement, when followed by an expression, causes the value of the expression to be returned as the result of the function. In the above example, `words[N]` is returned as the result.

Python functions always produce a result, whether the code contains a `return` statement or not. If the path of execution falls off the end of the function, `None` is returned. A lone `return` statement, not followed by an expression, also returns *None*.

Default values for parameters can be specified by placing a "=" and an expression or a literal value after the parameter name. (For a discussion of using expressions to specify a default value, see the next section.)

In the following example, the function's *filename* parameter has no default value, the *compresslevel* argument has a default value of 9, and the *fileobj* parameter has a default value of `None`.

```
def open_file(filename,
              compresslevel = 9,
              fileobj = None):
    if fileobj is None:
        fileobj = open(filename, 'w')
    compressor = zlib.compressobj( compresslevel )
```

To call a function with keyword arguments that can be specified in any order, no special function definition is needed, though default arguments are often used in concert with keywords. The function call can simply give the names of function parameters, followed by an "=" and the desired value for the parameter. Given a function definition like `f(a, b = 'empty', c=3.5)`, the following function calls are all legal:

```
f(1, 'new', 7.2)
f(1, c=7.2)           # Equivalent to f(1, 'empty', 7.2)
f(c=1, a=4, b='new')  # Equivalent to f(4, 'new', 1)
```

The following calls are all illegal:

```
f(b = 'new')    # No value provided for a
f(a = 4, 4)     # Non-keyword argument after keyword argument
f(4, b=1, c=3, b=1) # duplicate keyword argument b
```

## 6.2 Using an Expression for a Default Parameter Value

If a default value is some expression, that expression is evaluated at the time the `def` statement is executed, not when the function is called. Consider the following example:

```
default_level = 9
def new_compressor(compresslevel = default_level):
    print 'Compression level:', compresslevel

new_compressor()   # Prints the default value
default_level = 2  # Will this change the default value?
new_compressor()
```

When this code is run, it will print:

```
Compression level: 9
Compression level: 9
```

Remember, `def` is an executable statement, binding the function's name "`new_compressor`" to a function object. At that time, `default_level` has a value of 9, so that's used as the default value of *compresslevel*. Subsequently changing the value of `default_level` has no effect on the default value of the function. It's not possible to change the object that's been defined as the default value. However, if an object is mutable and can be modified in place, as a list or dictionary can be, then any modification made to that object will change the default. Another example will make this clearer, I hope:

```
def modify_list(List = []):
    print 'Before:', id(List), List
    List.append( 1 )
    print 'After: ', id(List), List
```

Calling this function with `modify_list()` will output:

```
Before: 135048144 []
After:  135048144 [1]
```

The first number, the value of `id(List)`, is an integer guaranteed to be unique for this object while the object exists, and is usually derived from the machine address at which the object resides. The value printed here will vary from run to run of the Python interpreter, but will remain the same in each execution. On the first call of `modify_list()`, the list object assigned as the default value is changed. A second, identical call will produce:

```
Before: 135048144 [1]
After:  135048144 [1, 1]
```

`id(List)` is the same on both function calls, so the same list object is assigned as the default value, but the contents of that object have been changed, which doesn't affect the value returned by `id(List)`. New users are often surprised by this behaviour, and wonder if it's a bug in Python; it's not, and follows consistently from Python treating everything as a reference.

## 6.3 Returning a Variable Number of Values

You want to have a function that returns more than one value as its result. For example, you may have a function that takes a colour name, and returns three numbers, the red, green, and blue values for that colour.

The simplest and most common solution is to return a tuple containing the different values:

```
def get_rgb_colour(colour_name):
    # ... set R, G, B, to the values
    return R, G, B
```

An alternative solution, using a class instance to hold return values, is useful if you want to return very many values, or differing numbers of values in different cases. You would set instance values of the class instance, and then return the instance to the caller.

```
class RGBSetting: pass

def get_rgb_colour(colour_name):
    retval = RGBSetting()
    # ... set retval.R, retval.G, retval.B, to the values
    return retval
```

## 6.4 Ignoring Return Values from a Function

Some programming languages, such as the scripting language in MATLAB, allow calling a function and ignoring some of its return values. You need to do this in Python, ignoring some value returned from a function.

If the function returns a tuple, as done by most functions that return multiple results, treat the return value as a tuple and slice it to obtain the results that you need.

```
def get_rgb_colour(colour_name):
    # ... set R, G, B, to the values
    return R, G, B

# Get just the red component of a colour
tup = get_rgb_colour('teal')
R = tup[0]
```

This can be expressed more compactly, though less clearly, without a tuple variable as:

```
R = get_rgb_colour('teal')[0]
```

Yet another solution is to use a dummy variable name for the variables that aren't of interest. _ is suggested as a short and rarely used name for this purpose.

```
R, _, _ = get_rgb_colour('methyl yellow')
```

For the sake of completeness, it should be mentioned that Perl supports an idiom for ignoring return values by assigning them to `undef`, the Perl counterpart of Python's `None`. (This idiom is rarely actually used in Perl programs.) Don't try the same thing in Python, which would be:

```
    None, None, B = get_rgb_colour('da Vinci green')
```

This will run without errors, and `B` will get the expected value, but it also adds a new name to the local scope: `None` is now bound to some value. (Maybe that last sentence should end with a '!') Later uses of `None` in the same code will retrieve the assigned value, not the Python `None` object. In short, if you use the above idiom in Perl, don't attempt to use it in Python.

## 6.5 Passing Functions as Parameters

You wish to pass a function as an argument to another function.

Python is flexible enough to treat functions and methods in the same way as ordinary variables.

```
F = f.write
F(s)
```

is the same thing as:

```
f.write(s)
```

As a computer scientist would describe it, functions are first-class objects in Python. So, you can simply pass a function as one of the arguments:

```
def repeat_func(func, N, data):
    "Repeat a function N times"
    for i in range(N):
        data = func(data)
    return data

repeat_func(differentiate, 3, data)
```

It's also possible to store functions in Python data structures, such as variables, lists, tuples, and class instances. For example, the following function takes a list of functions, and calls them all, from left to right.

```
def compose(func_list, data):
    for F in func_list:
        data = F(data)
    return data
```

A common application of this is to keep a dictionary of handler functions, retrieving the function to call. The following toy example has a dictionary mapping command names to the functions that should be called:

```
def output_status():
    ...

def set_bf_mode():
    ...

... other function definitions ...

cmd_dict = {'status': output_status, 'brightfield', set_bf_mode,
            'darkfield': set_df_mode, 'icr': set_icr_mode}

def call_command(cmd_name):
    function = cmd_dict[cmd_name]
    function()
```

It's not necessary to actually have a `def` statement in order to create a function. The `lambda` keyword is used to create small unnamed functions. The function is limited to a single expression, but that's enough to be useful in many cases. A common use for `lambda` is in concert with built-in functions like `map()`, which loops over the elements of a list, performs a function on each list element, and returns a list containing the results.

```
>>> map(string.upper,
...     ["Here's", "a", "list", "of", "words"] )
["HERE'S", 'A', 'LIST', 'OF', 'WORDS']
```

`lambda` would be used if there's no built-in function corresponding to what you need. For example, the following code would take the first character of every word in the list:

```
>>> map(lambda word: word[0],
...     ["Here's", "a", "list", "of", "words"] )
['H', 'a', 'l', 'o', 'w']
```

Here's another example. In this case, a default argument is used to bind the `f.write()` method to the local variable `wr`, and added a newline before calling the write method.

```
ftp.retrlines("RETR sightings.txt",
              lambda s, wr = f.write: wr( s+"\n" ) )
```

## 6.6 Calling a Function with an Argument Tuple

You have a function in a variable `F`, and a tuple `T`containing its arguments. You wish to call `F` with the arguments `T`.

Use the built-in function `apply()`:

```
result = apply(F, T)
```

Functions that don't require any arguments can be called using an empty tuple. For example, `time.time()` doesn't take any arguments, so it can be called like this:

```
curtime = apply( time.time, () )
```

If the function takes keyword arguments, you can also pass in a dictionary whose keys and values are the keyword arguments and their values.

```
# Equivalent to
#  menu.add_command(label = 'Open Help', command = window.show_help)
kw = {'label': 'Open Help',
      'command': window.show_help }

apply(menu.add_command, (), kw)
```

**Discussion:**

One subtle point is that `apply(F, T)` is not the same as `F(T)`. `F(T)` calls `F` with exactly one argument, the tuple `T`, while `apply(F, T)` calls `F`F with `T[0]` as the first argument, `T[1]` as the second, and so forth. An empty tuple corresponds to calling the function with no arguments.

## 6.7 How do I use different functionality on different platforms with the same program?

Remember that Python is extremely dynamic and that you can use this dynamism to configure a program at run-time to use available functionality on different platforms. For example you can test the sys.platform and import different modules based on its value.

```
import sys
if sys.platform == "win32":
    import win32pipe
    popen = win32pipe.popen
else:
    import os
    popen = os.popen
```

Also, you can try to import a module and use a fallback if the import fails:

```
try:
    import really_fast_implementation
    choice = really_fast_implementation
except ImportError:
    import slower_implementation
    choice = slower_implementation
```

This is commonly used to write code which will use a faster C extension module if it's installed, but will still work without the extension.

# 7 File Input and Output

## 7.1 How do I specify a file's name, so that I can open it?

There is a problem that routinely bedevils new Python users who come from a Windows or DOS background. The problem is that on Windows, a backslash is a separator in filenames, whereas on Unix (where Python originated) the backslash has an entirely different function (as an escape character) and the *forward* slash is the filename separator character.

When a person from a Windows background starts having problems with filenames, and goes looking for a solution, the first thing that he (or she) usually finds is Python's "raw" strings. However, from the perspective of a Windows user, raw strings aren't truly raw — they are only semi-raw, as you will rudely discover the first time that you use a raw string to specify a filename (or filename part) that ends with a single backslash.

A better strategy is to routinely code your filenames using forward slashes, and then use a Python function such as os.path.normcase to change the separator to whatever is appropriate to the local operating system. A nice bonus of ths strategy is that it is platform independent

```
import os.path
myFilename = "c:/mydir/myfile.txt"
myFilename = os.path.normcase(myFilename)
```

## 7.2 How can I list files in a directory (aka, folder)?

You want to make a list of the files in a directory, in order to do something for every file.

The `os.listdir(`*`path`*`)` functions returns a list of strings containing all the contents of the directory specified by *path*, in no particular order. Note that the list will also include directory names in the resulting list.

```
>>> os.listdir('/usr')
['X11R6', 'bin', 'dict', 'doc', 'etc', 'games', ... ]
```

If you want to list all the files matching a wildcard specification such as `*.py`, you can implement it by checking each filename in the list returned by `os.listdir()`. The `glob()` function in the `glob` module can do this for you:

```
>>> import glob
>>> glob.glob('/usr/lib/python1.5/B*.py')
['/usr/lib/python1.5/BaseHTTPServer.py', '/usr/lib/python1.5/Bastion.py']
```

`glob.glob()` will also list directories that match the wildcard specification.

Once you've gotten a list of filenames, you'll often want to limit it to those names that correspond to files and not directories. The `os.path` module contains functions called `isfile(`*`path`*`)`, `isdir(`*`path`*`)`, and `islink(`*`path`*`)` which return true if the path corresponds to a file, directory, or symbolic link.

These functions can be combined with the built-in `filter()` to easily select those paths that are actually files:

```
filelist = os.listdir('.')
filelist = filter(os.path.isfile, filelist)
```

## 7.3 How do I loop over the lines of a file?

You want to read through a file line by line. This is the most common way of processing text files. For example, if you're writing a Python program that searches for text in a file, you'll have to

You can code the loop explicitly, like this:

```
file = open('/tmp/filename', 'r')
while True:
    line = file.readline()
    if line == "": break    # Check for end-of-file
    do_something(line)
file.close()
```

Or like this, which reads all of the lines of the file into a list:

```
file = open('/tmp/filename', 'r')
lines = file.readlines()
file.close()
for line in lines:
    do_something(line)
```

The `fileinput` module makes this even simpler by handling the loop for you:

```
import fileinput
for line in fileinput.input("/tmp/filename"):
    do_something(line)
```

For sheer simplicity, it's hard to top reading all the lines from the file into a list, like this:

```
file = open('/tmp/filename', 'r')
lines = file.readlines()
file.close()
for line in lines:
    do_something(line)
```

Or like this:

```
file = open('/tmp/filename', 'r')
for line in file.readlines()
    do_something(line)
```

This is easy to code, but it does require reading the entire file into memory. These days, most systems will have enough memory to effortlessly handle files a few hundreds of kilobytes long. A 10 megabyte file will cause problems - swapping to disk, if not an actual crash - for many systems. Use your common sense; if you're pretty sure you won't need to handle large files, use `file.readlines()`; otherwise, use either of the two suggested solutions.

To loop over multiple files using fileinput, use a sequence of filenames (a list, tuple, etc) instead of just a single file.

```
for filename in ("file1", "file2", "file3"):

    file  = open(filename, 'r')
    lines = file.readlines()
    file.close()

    for line in lines:
        do_something(line)
```

If you omit the names, they default to `sys.argv[1:]`, or to standard input if no arguments were given.

## 7.4 How do I read a file backwards?

If you can read the entire file into memory, you can simply read all the data into a list and then reverse the list.

```
L = open('myFile').readlines()
L.reverse()
```

On a character-by-character basis, a similarly memory-bound solution is:

```
import string
L = string.split(open('thefile', 'r').read(), '')
L.reverse()
```

## 7.5 How do I emulate 'getch()' under UNIX-like systems?

The terminal has to be put into cbreak mode, in order to be able to read single characters from the terminal.

```
import sys

def getch():
    import os, tty

    fd = sys.stdin.fileno()
    tty_mode = tty.tcgetattr(fd)
    tty.setcbreak(fd)
    try:
            ch = os.read(fd, 1)
    finally:
            tty.tcsetattr(fd, tty.TCSAFLUSH, tty_mode)

    return ch
```

## 7.6 How do I read (or write) binary data?

Use the struct module. It allows you to take a string read from a file containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 4-byte integers in big-endian format from a file:

```
import struct
f = open(filename, "rb")  # Open in binary mode for portability
s = f.read(8)
x, y = struct.unpack(">ll", s)
```

The '>' in the format string forces big-endian data; each letter 'l' reads one "long integer" (4 bytes) from the string.

You can refer to the Library Reference for more details about the struct module.

## 7.7 How do I recursively walk through a directory tree?

Import the os module, and use os.path.walk(path, function, arg). That recursively walks through the tree rooted at _path_, and calls function() in each directory that's visited. function() must accept 3 arguments: (arg, dirname, name_list). arg is the same as the value passed to os.path.walk; dirname is the current directory name; name_list is a list of the names of the directories and filenames in the directory. You can modify name_list in place to avoid traversing certain subdirectories.

Here's a simple program to find files:

```
import os, sys

def find(arg, dirname, names):
    if arg in names:
        print os.path.join(dirname, arg)

os.path.walk(sys.argv[1], find, sys.argv[2])
```

Sample usage:

```
amarok gui>python ~/t.py /tmp/ README
/tmp/xml-0.4/README
/tmp/xml-0.4/demo/README
/tmp/xml-0.4/dom/README
```

## 7.8 How do I create/delete/truncate a file or directory?

To delete (remove, unlink) a file, use `os.remove(filename)` or `os.unlink(filename)`

For documentation, see the posix section of the library manual. They are the same: `unlink()` is simply the Unix name for this function.

To remove a directory — `os.rmdir()`

To create a directory — `os.mkdir()`

To rename a file — `os.rename()`

To truncate a file, open it using `f = open(filename, "r+")`, and use `f.truncate(offset)`

(The offset defaults to the current seek position. The `"r+"` mode opens the file for reading and writing.)

There's also `os.ftruncate(fd, offset)` for files opened with `os.open()` — for advanced Unix hacks only.

## 7.9 How do I copy a file?

Most of the time this will do:

```
infile = open("file.in", "rb")
outfile = open("file.out", "wb")
outfile.write(infile.read())
```

However for huge files you may want to do the reads/writes in pieces (or you may have to), and if you dig deeper you may find other technical problems.

Unfortunately, there's no totally platform independent answer. On Unix, you can use `os.system()` to invoke the "cp" command (see your Unix manual for how it's invoked). On DOS or Windows, use `os.system()` to invoke the "COPY" command. On the Mac, use `macostools.copy(srcpath, dstpath)`. It will also copy the resource fork and Finder info.

There's also the shutil module which contains a `copyfile()` function that implements the copy loop; but in Python 1.4 and earlier it opens files in text mode, and even in Python 1.5 it still isn't good enough for the Macintosh: it doesn't copy the resource fork and Finder info.

## 7.10 How do I rename a file?

```
#Rename file with filename f1 to filename f2
        os.rename(f1,f2)
```

## 7.11 How can I persistent an object?

(Persistent == automatically saved to and restored from disk.)

The library module "pickle" solves this in a very general way (though you still can't store things like open files, sockets or windows), and the library module "shelve" uses pickle and (g)dbm to create persistent mappings containing arbitrary Python objects. For possibly better performance, use the cPickle module.

A more awkward way of doing things is to use pickle's little sister, marshal. The marshal module provides very fast ways to store noncircular basic Python types to files and strings, and back again. Although marshal does not do fancy things like store instances or handle shared references properly, it does run extremely fast. For example loading a half megabyte of data may take less than a third of a second (on some machines). This often beats doing something more complex and general such as using gdbm with pickle/shelve.

## 7.12 How can I randomizing the lines in a file?

Similar to generating a random list of numbers, produce a list containing all the lines, and then choose items at random.

```
lines = sys.stdin.readlines()
        # or file.readlines(), where file is a file object
while lines:
    line = random.choice(lines)
    lines.remove(line)
    print line,
```

## 7.13 What is the fastest way to read a file line-by-line?

You can use the 'sizehint' parameter to readlines() to get some of the efficiency of readlines() without reading in the whole file. The following code isn't optimized, but it shows the idea:

```
class BufferedFileReader:
        def __init__ (self, file):
                self.file = file
                self.lines = []
                self.numlines = 0
                self.index = 0

        def readline (self):
                if (self.index >= self.numlines):
                        self.lines = self.file.readlines(65536)
                self.numlines = len(self.lines)
                self.index = 0

                if (self.numlines == 0):
                        return ""

                str = self.lines[self.index]
                self.index = self.index + 1
                return str
```

## 7.14 How can I get a list of files in a directory?

There are a few ways of doing this.

- `os.listdir(path)` returns a list containing the names of all the files in the directory specified by path.

  ```
  >>> import os
  >>> os.listdir('.')
  ['Group.java', 'Group.class', 'User.java', ... ]
  ```

  So you can do `os.listdir('C:/whatever/sub/dir/ectory')`, and then loop over each filename with a for loop.

- The glob module returns a list of filenames matching a wildcard pattern.

  ```
  >>> import glob
  >>> glob.glob('*.java')
  ['Group.java', 'User.java', 'GroupWrapper.java', ... ]
  ```

## 7.15 How can I recursively find files with a given extension?

Suppose you want to use`os.path.walk` to look for filename and directory of any files with a given file extension, starting with the top level directory. As a warm-up exercise, try this little test program:

```
import os

# A simple function just to print its arguments
def myprinter(*args):
    print args

os.path.walk('.', myprinter, 'data')

('data', './gsrvdir1174', ['gsrv'])
('data', './.pcmcia', ['pcram'])
('data', './ZTemplates-2.3.0', ['CHANGES.txt', 'README.txt', 'LICENSE.txt'])
```

You can see that the `myprinter` function is passed three arguments: (1) the third argument of `os.path.walk`, which is intended for your own application-specific info, (2) the directory path, and (3) a list containing the filenames in the directory.

So to do what you want, you would have to do something like this:

```
file_list = []

def myfilter(file_list, dirname, names):
    # Find the database files.  For use with os.path.walk()
    for name in names:
                match = regexp.match(name)
            if match:
            fullpath = os.join(dirname, name)
            if os.isfile(fullpath):
            file_list.append((dirname, match.group(1)))

dbfiles = []
os.path.walk('.', myfilter, dbfiles)
```

## 7.16 How can I lock a file?

When a file will be accessed and modified by several programs at the same time, you need to ensure that the programs can't make conflicting changes at the same time, which can result in the file being corrupted. This is accomplished by locking the file.

The `posixfile` module provides an object that acts like Python's standard file objects, but adds some extra methods. One of the extra methods is `lock(`*`mode, len, start, whence`*`)`. *mode* is a string specifying whether you want a read or write lock, or to give up an already-acquired lock.

For example, to gain a write lock for the entire file:

```
file.lock('w')
```

Locks can be for reading or writing. Multiple read locks can be held by different processes, because several processes can read the same data at the same time without harm. Only a single write lock can be held at a given time, and read locks won't be granted while a write lock is being held.

When requesting a lock, the default is to lock the entire file. It's also possible to lock a part of the file, by specifying the start and length of a region in the file. This allows multiple processes to modify different parts of a file at the same time. Consult the Library Reference documentation for `posixfile` for more information.

## 7.17 How do I read (and write) a compressed file?

Suppose you want to read a file that's been compressed with the GNU `gzip` program, or want to write a compressed file that `gzip` can read.

The `gzip` module provides a `GzipFile` class for reading and writing `gzip`'ped files. `GzipFile` instances imitate the methods of Python's standard file objects. To read a file:

```
import gzip

# Open the file in 'r' mode for reading
file = GzipFile('data.gz', 'r')
line = file.readline()          # Read a single line
data = file.read(1024)          # Read 1K of data
```

```
file.close()
```

Writing compressed data is much the same:

```
import gzip

# Open the file in 'w' mode for writing
file = GzipFile('output.gz', 'w')
file.write( 'First line of output\n')
file.close()
```

It's legal to use the `'a'` mode to append data to a compressed file; the `gzip` file format can handle a file with several chunks of compressed data. When reading such a file with `GzipFile`, you won't be able to tell where one chunk leaves off and the next begins because `GzipFile` seamlessly handles the transition between them.

The `gzip` module is built on top of the `zlib` module, which compresses strings of data. The `zlib` module can be useful on its own to save disk space or network bandwidth. For example, you can compress data before sending it over a TCP/IP socket or storing it in a DBM file. Here

```
>>> import zlib
>>> s = "This is a test of the emergency broadcast chicken."
>>> comp = zlib.compress(s)
>>> comp
'x\234\013\311\310,V\000...'
>>> print zlib.decompress( comp )
This is a test of the emergency broadcast chicken.
```

The `zlib` module also provides compressor and decompressor objects that can be used to compress large amounts of data without having to squeeze all of it into memory. Consult the Library Reference documentation for more information.

# 8 Network Programming: Internet Protocols

## 8.1 Introduction

The rise of the Internet has greatly eased the life of network programmers, by reducing the importance of proprietary standards and replacing them with open protocols whose specifications are readily available. Various standard Internet protocols, such as the Simple Mail Transport Protocol (SMTP) for electronic mail, or the Web's HyperText Transfer Protocol (HTTP), now dominate their respective niches. Python's standard library includes many modules that implement different Internet protocols.

## 8.2 Sending E-Mail

You want to send an e-mail message that has been generated by Python code. Many applications require sending e-mail automatically. Most Web sites have a feedback form where submissions will be sent to maintenance staff. Python programs that are executed automatically may send their output to the person responsible for them, or may send an e-mail if the program can't run successfully because of an error condition, such as a full disk partition.

The way to do it is to use the `smtplib` module:

```
import smtplib

message = """From: sender@example.com
Subject: Sample E-mail

Hi!
This message is just to say hello, and has been automatically generated.
"""

S = smtplib.SMTP('mail.example.com')
errdict = S.sendmail( from_addr = 'sender@example.com',
                      to_addrs = ['receiver@example.com'],
                      msg = message)
if len(errdict) != 0:
    print 'Not all addresses accepted'
    print 'Failed addresses:', errdict.keys()
```

The Simple Mail Transfer Protocol (SMTP) is the most common protocol used for sending mail. On Unix systems, SMTP mail is most commonly handled by the sendmail daemon, though there are several alternative mail transport agents, such as Exim and qmail, that also implement SMTP.

The format of SMTP e-mail messages is defined in RFC 822. A message consists of a header followed by the actual content of the message. The end of the header is indicated by a blank line. A sample RFC 822 message looks like this:

```
Return-Path: <amk>
Received: (from amk@localhost)
        by 207-172-99-98.s98.tnt13.ann.erols.com (8.8.7/8.8.7) id QAA00866
        for amk; Sun, 14 Feb 1999 16:55:57 -0500
Date: Sun, 14 Feb 1999 16:55:57 -0500
From: "A.M. Kuchling" <amk@mira.example.com>
CC: other-person@example.com
Message-Id: <199902142155.QAA00866@mira.example.com>
To: amk@mira.example.com
Subject: Hi!

Hello!  Long time no hear...
```

Each header line contains a header name, such as "`From`" or "`CC`". Header names are case-insensitive, so it doesn't matter whether the name is "`FROM`" or "`From`". A colon separates the header name from the header's contents. RFC 822 defines various standard header names for specifying the recipient's address, the sender's name, addresses to which carbon copies should be sent, etc.

You can add your own headers by following a naming convention; header names beginning with "`X-`" are considered extensions, and won't be mangled or dropped by the programs transporting the mail. For example,if you're writing a set of scripts which will send an e-mail on completion, you might want to add a header giving the name of the script sending the mail. This header could be called "`X-Script-Name`". Don't invent headers whose names don't begin with "`X-`", because a mail transport agent might drop them along the path from the sender to the recipient. To create headers, simply add them to the header of your message:

```
message = """From: sender@example.com
Cc: testing@example.com
X-Script-Name: mailnotify.py
Subject: Sample E-mail

Hi!  ... """
```

Proprietary mail systems, such as cc:Mail from Lotus, or platform-specific interfaces such as Microsoft's MAPI, don't use SMTP, so the `smtplib` module won't help you use such systems. If you're trying to use such a system, you'll have to either access them through some platform-specific mechanism such as Microsoft's COM, or write a C extension module specifically for the mail interface.

On Unix-like systems, it's also possible to send mail by running a mail program (such as `/usr/lib/sendmail`) in a subprocess and sending it the message over a pipe; this can be done by the `os.popen()` function. Using a mail program has some drawbacks; it assumes that mail service is correctly configured on the machine, and the location of the sendmail program may vary between systems; `/usr/lib/sendmail` is most common, but alternative locations such as `/usr/sbin/sendmail` have been seen. (Refer to the manual page for Sendmail.) Worst of all, if an error occurs, the subprocess may just print an error message which will then be ignored by your program. Using `smtplib` is often the better course.

Here's an example, in case you still need to send mail this way:

```
SENDMAIL = "/usr/sbin/sendmail" # sendmail location
import os
p = os.popen("%s -t" % SENDMAIL, "w")
p.write("To: recipient@example.com\n")
p.write("Subject: test\n")
p.write("\n") # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print "Sendmail exit status", sts
```

## 8.3 Downloading Files Through FTP

You want to retrieve a file from an FTP site by using a Python program.

Use the `ftplib` module.

```
import ftplib

# Open the connection
ftp = ftplib.FTP("ftp.fbi.example.gov")
ftp.login("mulder", "trustno1")

# Change directory
ftp.cwd('pub/x-files')

# Write the output to a local file.
# It's a binary file, so be sure to open the file in
# binary mode.
f = open("trixie.jpg", "wb")
ftp.retrbinary("RETR trixie.jpg", f.write)
f.close()

# Close connection
ftp.quit()
```

The `ftplib` module provides an `FTP` class that represents a connection to an FTP server. To open the connection, you must provide a username and password.

Retrieving the file isn't completely straightforward, because the `FTP` class provides a highly flexible interface. There's no method which will retrieve a file named "x" and write it to a local file named "x"; instead, you have to handle such details yourself. When a file is retrieved via FTP using the `retr` and `retrbinary` methods, you have to provide two pieces of information: the filename, and a Python function. The Python function will be called for each chunk of the file's contents, receiving a a string containing the data; the function then has the job of writing the data to a file. Luckily, you can often find a built-in function or method that does what you need, as in the example:

```
file = open("trixie.jpg", "wb")
ftp.retrbinary("RETR trixie.jpg", file.write)
file.close()
```

`file` is a file object, whose `write(`*S*`)` method writes the string *S* to the file. In Python, you can use bound methods as callbacks; a bound method remembers the object that it belonged to. `file.write` is a bound method in this example. You can provide a custom function that does additional work beyond just writing out the data.

When retrieving a file in text mode, each line is returned without any trailing newline characters, so you can add the required line endings for your platform when writing each line out to a file.

```
f = open("sightings.txt", "w")
ftp.retrlines("RETR sightings.txt",
              lambda s, w = f.write: w(s+"\n") )
f.close()
```

The following function prints a progress counter as the file is retrieved, by maintaining a count of the bytes read so far. On every call, the current byte count is padded out to 9 characters, printed, and followed by 9 backspace characters.

```
import sys

byte_count = 0
output_file = open('...', 'w')

def write_block(data):
    global byte_count
    byte_count = byte_count + len(data)

    output_file.write( data )

    # Output the current byte count and backspaces
    sys.stdout.write('%9i%s' % (byte_count, 9*'\010') )
sys.stdout.flush()
```

`sys.stdout` has to be flushed after each `write()` call, in order to ensure that the output is actually sent to the screen as it's generated.

See also the section on "Passing Functions as Parameters". It discusses how to pass a function as a parameter to another function, and will help you to understand callbacks.

## 8.4 Uploading Files Through FTP

This is the opposite of the previous problem. You now want to upload a file to an FTP server.

The `ftplib` module comes to the rescue, again.

```
import ftplib

# Open the connection
ftp = ftplib.FTP("ftp.fbi.example.gov")
ftp.login("mulder", "trustno1")

# Change directory
ftp.cwd('pub/x-files/bigfoot')

# Read from a local text file, and send it to the FTP server
# The local filename is 'sightings', but it'll be stored on the remote FTP
# server under the name 'sightings-1999-01'.
```

```
f = open("sightings", "r")
ftp.storlines("STOR sightings-1999-01", f)
f.close()

# Upload a binary file.
# Local filename: 'footprint.jpg', remote filename: 'footprint-1.jpg'
f = open("footprint.jpg", "rb")
ftp.storbinary("STOR footprint-1.jpg", f, 1024)
f.close()

# Close connection
ftp.quit()
```

When uploading a binary file, as opposed to a text file, a third numeric argument has to be provided to the `storbinary()` method.

```
ftp.storbinary("STOR footprint-1.jpg", f, 1024)
```

The file will be read and sent over the network in chunks of the specified block size, so it shouldn't be set to a very small value like 1; this would result in too much overhead. It would be possible to spend a good deal of effort trying different values of the block size and measure the performance in order to figure out the optimal block size, but it's not that critical a parameter. 1024 is a reasonable number, so you can simply set the value and forget about it.

## 8.5 Retrieving Files Through HTTP

You want to retrieve a file through HTTP. The data returned might be the HTML text of a Web page, an image such as a GIF or JPEG, or a binary file such as a JAR or TGZ file.

The easiest course is to use the `urllib` module, which will try to open an arbitrary URL and return a file object which can be read from to retrieve the data.

```
import urllib

# Open the URL
input = urllib.urlopen('http://www.example.com/info/index.html')

# Read all the data from the file as a single string
data = input.read()
input.close()
```

An alternative solution that operates at a lower level is to use the `httplib` module to perform the HTTP transaction yourself.

```
import httplib
h = httplib.HTTP('www.example.com')
h.putrequest('GET', '/info/index.html')
h.putheader('Accept', 'text/html')
h.endheaders()
errcode, errmsg, headers = h.getreply()
if errcode == 200:
    f = h.getfile()
    data = f.read() # Print the raw HTML
```

Some pages require a user name and password; Web browsers will prompt you for them when required. The `urlopen()` function will prompt the user and request the user name and password on standard input. Such user intervention is unsuitable for fully automatic Web page retrievals, so you should specify the user name and password when retrieving such a password-protected page automatically. The user name and password can be specified in the URL, when it's written in the form `http://`*`user`*`:`*`password`*`@host/...`.

Fredrik Lundh suggests an alternative approach that's more flexible, but is also more complicated. The `urllib` module contains several classes for opening URLs using different protocols, one such class being `FancyURLopener`. The `urlopen()` function simply creates an instance of `FancyURLopener` and uses it to access URLs. When input from the user is required, the `prompt_user_passwd()` method is called. Fredrik suggests the following approach: subclass `FancyURLopener` and add attributes to hold the username and password. `prompt_user_passwd()` is then overridden to simply return the user name and password without user intervention.

```
import urllib

class myURLOpener(urllib.FancyURLopener):
    def setpasswd(self, user, passwd):
        self.__user = user
        self.__passwd = passwd

    def prompt_user_passwd(self, host, realm):
        return self.__user, self.__passwd

urlopener = myURLOpener()
urlopener.setpasswd("mulder", "trustno1")

fp = urlopener.open("http://www.secretlabs.com")
print fp.read()
```