

Notes from Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

*<http://www.wellho.net/net/whcotnl.html>
for details.*

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

graham@wellho.net

technical contact

lisa@wellho.net

administration contact

Our full postal address is

404 The Spa

Melksham

Wiltshire

UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

Best Programming Practice

You can write good and bad programs in any programming language, and that includes Python. What makes for good and bad code? What guidelines should you follow to make your code quick to develop, be robust, easy to follow later, and flexible enough to be amendable to meet future requirements that you hadn't even dreamed of when you wrote it?

<i>Isn't it enough to be able to write a working program?</i>	<i>4</i>
<i>Analysing the requirement</i>	<i>4</i>
<i>Designing the solution</i>	<i>4</i>
<i>Reusing code.</i>	<i>4</i>
<i>Official style guide for Python code</i>	<i>5</i>

2.1 Isn't it enough to be able to write a working program?

No, it isn't!

A far higher proportion of the life costs of a piece of software are in its maintenance rather than its original writing, so it pays to spend a little more time to make a piece of code a lot more maintainable.

Writing and maintaining a program usually occupies a lot less time (and costs a lot less) than the investment that users will put into it in entering data and generating outputs. It pays to spend a little more time as you write a program ensuring that it has an excellent user interface that provides the user with what he needs to use it efficiently.

Requirements change over time, and it's usually far cheaper to adopt and adapt the existing system than keep coming up with a completely new one at each change. Sure, in time you may get to the point of doing a re-write but better to have a four-year cycle than a two-year cycle, and better to have a 10-year cycle than a five-year one.

2.2 Analysing the requirement

These paragraphs could be written for ANY language; it just happens to be part of a Python course in this case. Listen to the user's requirements, question the user, learn as much as you can about what the application is to do.

You may try and listen all at once (and it's a good idea to do so in broad overview) and/or you may listen to details and partial requirements. Techniques such as extreme programming suggest a series of requirements, each of a few sentences and implemented and tested and integrated into the whole in a relatively short timescale, and with the whole project consisting of 50 to 100 such steps.

2.3 Designing the solution

These paragraphs are written for a language that supports the Object Oriented Mantra, of which Python is one of the most ardent adherents.

For huge projects, formalised design systems such as UML, implemented using Rational Rose or other software, may be appropriate for you to use. For projects that are just large, that's probably overkill, but you want to look for a good design solution and framework.

Even if you're not going to use a full UML system, learn the principles and how the views are derived from the model and think of how each of the diagrams would look for your system. Remember:

- Use Case diagram
- Class and Object diagrams
- State diagram
- Sequence diagram
- Activity diagram
- Component diagram
- Deployment diagram

No need, probably, to use all the fancy symbols, simple boxes and arrows will be fine, although you might want to come up with company standards if there's a team of you working on a project.

2.4 Reusing code

Write your code to be re-usable. You're using an Object Oriented language and so you should naturally be thinking of objects that your whole organisation can use within all of their applications and not just in your own little area!

Figure 1 Example of a UML symbol that you really need to draw if you're just using the principles of UML ;-)

See if others have written re-usable code. If you're working for a university, has someone else already written a "student" and a "lecturer" class, and can you simply call their classes? If you're working for a pharmaceutical company, has someone already written an amino acid class, perhaps with subclasses for Alanine, Glycine and the rest?

If you've got more than a handful of Python projects within your organisation, it may be worth someone's while setting up a central repository or web site or discussion forum as appropriate. Perhaps you can even persuade your management to sponsor an annual meeting or event away from the office for cross-fertilisation of ideas and even a lecture or two from someone who's using Python in another organisation.

Search the Internet, too. There may already be classes out there that are freely available and will give you an excellent start. Have a look at the vaults of Parnassus. You'll probably find a lot of things that are not useful – that's the nature of searching – but you'll find some that are.

Here's the web site <http://biopython.org> as one Python-source example:

Figure 2 A source of Python code for Bioinformatics applications

2.5 Official style guide for Python code

The following is from the official style guide for Python code, written by Guido van Rossum, the author of the Python language, and placed in the public domain (which is why we're able to reproduce it here). It's available online at

<http://www.python.org/peps/pep-0008.html>

Why has Guido chosen to make this available not just "open source", but public domain? Because it is SO IMPORTANT that you write your Python code so that it's easy to follow and easy to maintain, and he wants the document to have the widest possible circulation.

Title: *Style Guide for Python Code*
Version: Revision: 1.25
Author: Guido van Rossum <guido at python.org>
Barry Warsaw <barry at python.org>
Status: Active
Type: Informational
Created: 05-Jul-2001
Post-History: 05-Jul-2001

Introduction

This document gives coding conventions for the Python code comprising the standard library for the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python[1].

This document was adapted from Guido's original Python Style Guide essay[2], with some additions from Barry's style guide[5]. Where there's conflict, Guido's style rules for the purposes of this PEP. This PEP may still be incomplete (in fact, it may never be finished <wink>).

A Foolish Consistency is the Hobgoblin of Little Minds

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgement. Look at other examples and decide what looks best. And don't hesitate to ask!

Two good reasons to break a particular rule:

- (1) When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.
- (2) To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).

Code lay-out

Indentation

Use the default of Emacs' Python-mode: 4 spaces for one indentation level. For really old code that you don't want to mess up, you can continue to use 8-space tabs. Emacs Python-mode auto-detects the prevailing indentation level used in a file and sets its indentation parameters accordingly.

Tabs or Spaces?

Never mix tabs and spaces. The most popular way of indenting Python is with spaces only. The second-most popular way is with tabs only. Code indented with a mixture of tabs and spaces should be converted to using spaces exclusively. (In Emacs, select the whole buffer and hit **ESC-x** untabify.) When invoking the python command line interpreter with the **-t** option, it issues warnings about code that illegally mixes tabs and spaces. When using **-tt** these warnings become errors. These options are highly recommended!

For new projects, spaces-only are strongly recommended over tabs. Most editors have features that make this easy to do. (In Emacs, make sure indent-tabs-mode is nil).

Maximum Line Length

There are still many devices around that are limited to 80 character lines; plus, limiting windows to 80 characters makes it possible to have several windows side-by-side. The default wrapping on such devices looks ugly. Therefore, please limit all lines

to a maximum of 79 characters (Emacs wraps lines that are exactly 80 characters long). For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but sometimes using a backslash looks better. Make sure to indent the continued line appropriately. Emacs Python-mode does this right. Some examples:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
            color == 'red' and emphasis == 'strong' or \
            highlight > 100:
            raise ValueError, "sorry, you lose"
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError, "I don't think so"
        Blob.__init__(self, width, height,
                     color, emphasis, highlight)
```

Blank Lines

Separate top-level function and class definitions with two blank lines. Method definitions inside a class are separated by a single blank line. Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

When blank lines are used to separate method definitions, there is also a blank line between the `class' line and the first method definition.

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. `^L`) form feed character as whitespace; Emacs (and some printing tools) treat these characters as page separators, so you may use them to separate pages of related sections of your file.

Encodings (PEP 263)

Code in the core Python distribution should always use the ASCII or Latin-1 encoding (a.k.a. ISO-8859-1). Files using ASCII should not have a coding cookie. Latin-1 should only be used when a comment or docstring needs to mention an author name that requires Latin-1; otherwise, using `\x` escapes is the preferred way to include non-ASCII data in string literals. An exception is made for those files that are part of the test suite for the code implementing PEP 263.

Imports

Imports should usually be on separate lines, e.g.:

```
No: import sys, os
Yes: import sys
     import os
```

It's okay to say this though:

```
from types import StringType, ListType
```

Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants. Imports should be grouped, with the order being

1. standard library imports
2. related major package imports (i.e. all email package imports next)
3. application specific imports

You should put a blank line between each group of imports.

Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports.

When importing a class from a class-containing module, it's usually okay to spell this

```
from MyClass import MyClass
from foo.bar.YourClass import YourClass
```

If this spelling causes local name clashes, then spell them

```
import MyClass
import foo.bar.YourClass
```

and use "MyClass.MyClass" and "foo.bar.YourClass.YourClass"

Whitespace in Expressions and Statements

Pet Peeves

Guido hates whitespace in the following places:

- Immediately inside parentheses, brackets or braces, as in:

```
spam( ham[ 1 ], { eggs: 2 } )
```

Always write this as

```
spam(ham[1], {eggs: 2})
```

- Immediately before a comma, semicolon, or colon, as in:

```
if x == 4 : print x , y ; x , y = y , x
```

Always write this as

```
if x == 4: print x, y; x, y = y, x
```

- Immediately before the open parenthesis that starts the argument list of a function call, as in `spam (1)`

Always write this as `spam(1)`

- Immediately before the open parenthesis that starts an indexing or slicing, as in:

```
dict ['key'] = list [index]
```

Always write this as

```
dict['key'] = list[index]
```

- More than one space around an assignment (or other) operator to align it with another, as in:

```
x           = 1
y           = 2
long_variable = 3
```

Always write this as:

```
x = 1
y = 2
long_variable = 3
```

(Don't bother to argue with him on any of the above – Guido's grown accustomed to this style over 20 years.)

Other recommendations

- Always surround these binary operators with a single space on either side:

assignment (=)

comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not)

Booleans (and, or, not).

- Use your better judgment for the insertion of spaces around arithmetic operators. Always be consistent about whitespace on either side of a binary operator. Some examples:

```
i = i+1
submitted = submitted + 1
x = x*2 - 1
hypot2 = x*x + y*y
```

```
c = (a+b) * (a-b)
c = (a + b) * (a - b)
```

- Don't use spaces around the '=' sign when used to indicate a keyword argument or a default parameter value. For instance:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

- Compound statements (multiple statements on the same line) are generally discouraged.

```
No: if foo == 'blah': do_blah_thing()
Yes: if foo == 'blah':
    do_blah_thing()
```

```
No: do_one(); do_two(); do_three()
Yes: do_one()
    do_two()
    do_three()
```

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end is best omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period, since it makes Emacs wrapping and filling work consistently.

When writing English, Strunk and White apply.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line containing a single #. Block comments are best surrounded by a blank line above and below them (or two lines above and a single line below for a block comment at the start of a new section of function definitions).

Inline Comments

An inline comment is a comment on the same line as a statement. Inline comments should be used sparingly. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x+1 # Increment x
```

But sometimes, this is useful:

```
x = x+1 # Compensate for border
```

Documentation Strings

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257 [3].

Write docstrings for all public modules, functions, classes, and methods.

Docstrings are not necessary for non-public methods but you should have a comment that describes what the method does. This comment should appear after the "def" line.

PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself, e.g.:

```
"""Return a foobang

    Optional plotz says to frobnicate the bizbaz first.
    """
```

For one liner docstrings, it's okay to keep the closing """ on the same line.

Version Bookkeeping

If you have to have RCS or CVS crud in your source file, do it as follows.

```
__version__ = "$Revision: 1.25 $"
# $Source: /cvsroot/python/python/nondist/peps/pep-0008.txt,v $
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent - nevertheless, here are the currently recommended naming standards. New modules and packages (including 3rd party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (or CapWords, or CamelCase - so named because of the bumpy look of its letters[4]). This is also sometimes known as StudlyCaps.
- mixedCase (differs from CapitalizedWords by initial lowercase character!)
- Capitalized_Words_With_Underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the os.stat() function returns a tuple whose items traditionally have names like st_mode, st_size, st_mtime and so on. The X11 library uses a leading X for all its public functions. (In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.)

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

- `_single_leading_underscore`: weak "internal use" indicator (e.g. "from M import *" does not import objects whose name starts with an underscore).
- `_single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, e.g. "Tkinter.Toplevel(master, class_='ClassName')."
- `__double_leading_underscore`: class-private names as of Python 1.4.
- `__double_leading_and_trailing_underscore__`: "magic" objects or attributes that

live in user-controlled namespaces, e.g. `__init__`, `__import__` or `__file__`. Sometimes these are defined by the user to trigger certain magic behavior (e.g. operator overloading); sometimes these are inserted by the infrastructure for its own use or for debugging purposes. Since the infrastructure (loosely defined as the Python interpreter and the standard library) may decide to grow its list of magic attributes in future versions, user code should generally refrain from using this convention for its own use. User code that aspires to become part of the infrastructure could combine this with a short prefix inside the underscores, e.g. `__bobo_magic_attr__`.

Prescriptive: Naming Conventions

Names to Avoid

Never use the characters ``l`` (lowercase letter el), ``O`` (uppercase letter oh), or ``I`` (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use ``l`` use ``L`` instead.

Module Names

Modules should have short, lowercase names, without underscores.

Since module names are mapped to file names, and some file systems are case insensitive and truncate long names, it is important that module names be chosen to be fairly short – this won't be a problem on Unix, but it may be a problem when the code is transported to Mac or Windows.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. `_socket`).

Python packages should have short, all-lowercase names, without underscores.

Class Names

Almost without exception, class names use the CapWords convention. Classes for internal use have a leading underscore in addition.

Exception Names

If a module defines a single exception raised for all sorts of conditions, it is generally called "error" or "Error". It seems that built-in (extension) modules use "error" (e.g. `os.error`), while Python modules generally use "Error" (e.g. `xdr.lib.Error`). The trend seems to be toward CapWords exception names.

Global Variable Names

(Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions. Modules that are designed for use via "from M import *" should prefix their globals (and internal functions and classes) with an underscore to prevent exporting them.

Function Names

Function names should be lowercase, possibly with words separated by underscores to improve readability. `mixedCase` is allowed only in contexts where that's already the prevailing style (e.g. `threading.py`), to retain backwards compatibility.

Method Names and Instance Variables

The story is largely the same as with functions: in general, use lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for internal methods and instance variables which are not intended to be part of the class's public interface. Python does not enforce this; it is up to programmers to respect the convention.

Use two leading underscores to denote class-private names. Python "mangles" these names with the class name: if class `Foo` has an attribute named `__a`, it cannot be accessed by `Foo.__a`. (An insistent user could still gain access by calling `Foo._Foo__a`.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Designing for inheritance

Always decide whether a class's methods and instance variables should be public or non-public. In general, never make data variables public unless you're implementing essentially a record. It's almost always preferable to give a functional interface to your class instead (and some Python 2.2 developments will make this much nicer).

Also decide whether your attributes should be private or not. The difference between private and non-public is that the former will never be useful for a derived class, while the latter might be. Yes, you should design your classes with inheritance in mind!

Private attributes should have two leading underscores, no trailing underscores.

Non-public attributes should have a single leading underscore, no trailing underscores.

Public attributes should have no leading or trailing underscores, unless they conflict with reserved words, in which case, a single trailing underscore is preferable to a leading one, or a corrupted spelling, e.g. `class_` rather than `klass`. (This last point is a bit controversial; if you prefer `klass` over `class_` then just be consistent. :).

Programming Recommendations

Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Pyrex, Psyco, and such). For example, do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form `a+=b` or `a=a+b`. Those statements run more slowly in Jython. In performance sensitive parts of the library, the `".join()"` form should be used instead. This will assure that concatenation occurs in linear time across various implementations.

Comparisons to singletons like `None` should always be done with `'is'` or `'is not'`. Also, beware of writing `"if x"` when you really mean `"if x is not None"` - e.g. when testing whether a variable or argument that defaults to `None` was set to some other value. The other value might be a value that's false in a Boolean context!

Class-based exceptions are always preferred over string-based exceptions. Modules or packages should define their own domain-specific base exception class, which should be subclassed from the built-in `Exception` class. Always include a class docstring. E.g.:

```
class MessageError(Exception):
    """Base class for errors in the email package."""
```

Use string methods instead of the string module unless backward-compatibility with versions earlier than Python 2.0 is important. String methods are always much faster and share the same API with unicode strings.

Avoid slicing strings when checking for prefixes or suffixes. Use `startswith()` and `endswith()` instead, since they are cleaner and less error prone. For example:

```
No: if foo[:3] == 'bar':
Yes: if foo.startswith('bar'):
```

The exception is if your code must work with Python 1.5.2 (but let's hope not!).

Object type comparisons should always use `isinstance()` instead of comparing types directly. E.g.

```
No: if type(obj) is type(1):
Yes: if isinstance(obj, int):
```

When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2.3, `str` and `unicode` have a common base class, `basestring`, so you can do:

```
if isinstance(obj, basestring):
```

In Python 2.2, the `types` module has the `StringTypes` type defined for that purpose,

e.g.:

```
from types import StringType
if isinstance(obj, StringType):
```

In Python 2.0 and 2.1, you should do:

```
from types import StringType, UnicodeType
if isinstance(obj, StringType) or \
    isinstance(obj, UnicodeType) :
```

For sequences, (strings, lists, tuples), use the fact that empty sequences are false, so "if not seq" or "if seq" is preferable to "if len(seq)" or "if not len(seq)".

Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, *reindent.py*) will trim them.

Don't compare boolean values to True or False using == (bool types are new in Python 2.3):

```
No:  if greeting == True:
Yes: if greeting:
```

References

- [1] PEP 7, Style Guide for C Code, van Rossum
- [2] <http://www.python.org/doc/essays/styleguide.html>
- [3] PEP 257, Docstring Conventions, Goodger, van Rossum
- [4] <http://www.wikipedia.com/wiki/CamelCase>
- [5] Barry's GNU Mailman style guide
<http://barry.warsaw.us/software/STYLEGUIDE.txt>

Copyright

This [The Style Guide for Python code] document has been placed in the public domain.

The Style Guide finishes here.

Copyright of the rest of this module is retained by Well House Consultants and is subject to the full copyright statement that is reproduced elsewhere and covers this set of training notes as a whole.

License

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log
Original Version, Well House Consultants, 2004

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

License Ends.