Code Like a Pythonista: Idiomatic Python

David Goodger
goodger@python.org
http://python.net/~goodger

In this interactive tutorial, we'll cover many essential Python idioms and techniques in depth, adding immediately useful tools to your belt.

My credentials: I am

- a resident of Montreal,
- father of two great kids, husband of one special woman,
- a full-time Python programmer,
- author of the Docutils project and reStructuredText,
- an editor of the Python Enhancement Proposals (or PEPs),
- an organizer of PyCon 2007, and chair of PyCon 2008,
- a member of the Python Software Foundation,
- a Director of the Foundation for the past year, and its Secretary.

In the tutorial I presented at PyCon 2006 (called Text & Data Processing), I was surprised at the reaction to some techniques I used that I had thought were common knowledge. But many of the attendees were unaware of these tools that experienced Python programmers use without thinking.

Many of you will have seen some of these techniques and idioms before. Hopefully you'll learn a few techniques that you haven't seen before and maybe something new about the ones you have already seen.

The Zen of Python (1)

These are the guiding principles of Python, but are open to interpretation. A sense of humor is required for their proper interpretation.

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.

There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!

—Tim Peters

This particular "poem" began as a kind of a joke, but it really embeds a lot of truth about the philosophy behind Python. The Zen of Python has been formalized in PEP 20, where the abstract reads:

Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

—http://www.python.org/dev/peps/pep-0020/

You can decide for yourself if you're a "Pythoneer" or a "Pythonista". The terms have somewhat different connotations.

When in doubt:

```
import this
```

Try it in a Python interactive interpreter:

```
>>> import this
```

Here's another easter egg:

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

Coding Style: Readability Counts

Programs must be written for people to read, and only incidentally for machines to execute.

—Abelson & Sussman, *Structure and Interpretation of Computer Programs*

Try to make your programs easy to read and obvious.

PEP 8: Style Guide for Python Code

Worthwhile reading:

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

PEP = Python Enhancement Proposal

A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment.

The Python community has its own standards for what source code should look like, codified in PEP 8. These standards are different from those of other communities, like C, C++, C#, Java, VisualBasic, etc.

Because indentation and whitespace are so important in Python, the Style Guide for Python Code approaches a standard. It would be wise to adhere to the guide! Most open-source projects and (hopefully) in-house projects follow the style guide quite closely.

Whitespace 1

- 4 spaces per indentation level.
- No hard tabs.
- **Never** mix tabs and spaces.

  This is exactly what IDLE and the Emacs Python mode support. Other editors may also provide this support.
- One blank line between functions.
- Two blank lines between classes.

Whitespace 2

- Add a space after "," in dicts, lists, tuples, & argument lists, and after ":" in dicts, but not before.
- Put spaces around assignments & comparisons (except in argument lists).
- No spaces just inside parentheses or just before argument lists.
- No spaces just inside docstrings.

```
def make_squares(key, value=0):
    """Return a dictionary and a list..."""
    d = {key: value}
    l = [key, value]
    return d, l
```

Naming

- `joined_lower` for functions, methods, attributes
- `joined_lower` or `ALL_CAPS` for constants
- `StudlyCaps` for classes
- `camelCase` **only** to conform to pre-existing conventions
- Attributes: `interface`, `_internal`, `__private`

  But try to avoid the `__private` form. I never use it. Trust me. If you use it, you **WILL** regret it later.

Long Lines & Continuations

Keep lines below 80 characters in length.

Use implied line continuation inside parentheses/brackets/braces:

```
def __init__(self, first, second, third,
             fourth, fifth, sixth):
    output = (first + second + third
              + fourth + fifth + sixth)
```

Use backslashes as a last resort:

```
VeryLong.left_hand_side \
    = even_longer.right_hand_side()
```

Backslashes are fragile; they must end the line they're on. If you add a space after the backslash, it won't work any more. Also, they're ugly.

Long Strings

Adjacent literal strings are concatenated by the parser:

```
>>> print 'o' 'n' "e"
one
```

The spaces between literals are not required, but help with readability. Any type of quoting can be used:

```
>>> print 't' r'\/\/' """o"""
t\/\/o
```

The string prefixed with an "r" is a "raw" string. Backslashes are not evaluated as escapes in raw strings. They're useful for regular expressions and Windows filesystem paths.

Note named string objects are **not** concatenated:

```
>>> a = 'three'
>>> b = 'four'
>>> a b
  File "<stdin>", line 1
    a b
      ^
SyntaxError: invalid syntax
```

That's because this automatic concatenation is a feature of the Python parser/compiler, not the interpreter. You must use the "+" operator to concatenate strings at run time.

```
text = ('Long strings can be made up '
        'of several shorter strings.')
```

The parentheses allow implicit line continuation.

Multiline strings use triple quotes:

```
"""Triple
double
quotes"""

'''\
Triple
single
quotes\
'''
```

In the last example above (triple single quotes), note how the backslashes are used to escape the newlines. This eliminates extra newlines, while keeping the text and quotes nicely left-justified. The backslashes must be at the end of their lines.

Compound Statements

Good:

```
if foo == 'blah':
    do_something()
do_one()
do_two()
do_three()
```

Bad:

```
if foo == 'blah': do_something()
do_one(); do_two(); do_three()
```

Whitespace & indentations are useful visual indicators of the program flow. The indentation of the second "Good" line above shows the reader that something's going on, whereas the lack of indentation in "Bad" hides the "if" statement.

Multiple statements on one line are a cardinal sin. In Python, *readability counts*.

Docstrings & Comments

Docstrings = **How to use** code

Comments = **Why** (rationale) & **how code works**

Docstrings explain **how** to use code, and are for the **users** of your code.

Comments explain **why**, and are for the **maintainers** of your code.

Both of these groups include **you**, so write good docstrings and comments!

Docstrings are useful in interactive use (`help()`) and for auto-documentation systems.

False comments & docstrings are worse than none at all. So keep them up to date! When you make changes, make sure the comments & docstrings are consistent with the code, and don't contradict it.

There's an entire PEP about docstrings, PEP 257, "Docstring Conventions":

> http://www.python.org/dev/peps/pep-0257/

Practicality Beats Purity

> A foolish consistency is the hobgoblin of little minds.
>
> —Ralph Waldo Emerson

(*hobgoblin*: Something causing superstitious fear; a bogy.)

There are always exceptions. From PEP 8:

> But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgment. 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).

Idiom Potpourri

A selection of small, useful idioms.

Now we move on to the meat of the tutorial: lots of idioms.

We'll start with some easy ones and work our way up.

Swap Values

In other languages:

```
temp = a
a = b
b = temp
```

In Python:

```
b, a = a, b
```

Perhaps you've seen this before. But do you know how it works?

- The **comma** is the tuple constructor syntax.
- A tuple is created on the right (tuple packing).
- A tuple is the target on the left (tuple unpacking).

The right-hand side is **unpacked** into the names in the tuple on the left-hand side.

Further examples of unpacking:

```
>>> l =['David', 'Pythonista', '+1-514-555-1234']
>>> name, title, phone = l
>>> name
'David'
>>> title
'Pythonista'
>>> phone
'+1-514-555-1234'
```

Useful in loops over structured data:

`l` (L) above is the list we just made (David's info). So `people` is a list containing two items, each a 3-item list.

```
>>> people = [l, ['Guido', 'BDFL', 'unlisted']]
>>> for (name, title, phone) in people:
...     print name, phone
...
David +1-514-555-1234
Guido unlisted
```

Each item in `people` is being unpacked into the `(name, title, phone)` tuple.

Arbitrarily nestable (just be sure to match the structure on the left & right!):

```
>>> david, (gname, gtitle, gphone) = people
>>> gname
'Guido'
>>> gtitle
'BDFL'
>>> gphone
'unlisted'
>>> david
['David', 'Pythonista', '+1-514-555-1234']
```

More About Tuples

We saw that the **comma** is the tuple constructor, not the parentheses. Example:

```
>>> 1,
(1,)
```

The Python interpreter shows the parentheses for clarity, and I recommend you use parentheses too:

```
>>> (1,)
(1,)
```

Don't forget the comma!

```
>>> (1)
1
```

In a one-tuple, the trailing comma is required; in 2+-tuples, the trailing comma is optional. In 0-tuples, or empty tuples, a pair of parentheses is the shortcut syntax:

```
>>> ()
()

>>> tuple()
()
```

A common typo is to leave a comma even though you don't want a tuple. It can be easy to miss in your code:

```
>>> value = 1,
>>> value
(1,)
```

So if you see a tuple where you don't expect one, look for a comma!

Interactive "_"

This is a really useful feature that surprisingly few people know.

In the interactive interpreter, whenever you evaluate an expression or call a function, the result is bound to a temporary name, _ (an underscore):

```
>>> 1 + 1
2
>>> _
2
```

_ stores the last *printed* expression.

When a result is `None`, nothing is printed, so _ doesn't change. That's convenient!

This only works in the interactive interpreter, not within a module.

It is especially useful when you're working out a problem interactively, and you want to store the result for a later step:

```
>>> import math
>>> math.pi / 3
1.0471975511965976
>>> angle = _
>>> math.cos(angle)
0.50000000000000011
>>> _
0.50000000000000011
```

Building Strings from Substrings

Start with a list of strings:

```
colors = ['red', 'blue', 'green', 'yellow']
```

We want to join all the strings together into one large string. Especially when the number of substrings is large...

Don't do this:

```
result = ''
for s in colors:
    result += s
```

This is very inefficient.

It has terrible memory usage and performance patterns. The "summation" will compute, store, and then throw away each intermediate step.

Instead, do this:

```
result = ''.join(colors)
```

The `join()` string method does all the copying in one pass.

When you're only dealing with a few dozen or hundred strings, it won't make much difference. But get in the habit of building strings efficiently, because with thousands or with loops, it **will** make a difference.

Building Strings, Variations 1

Here are some techniques to use the `join()` string method.

If you want spaces between your substrings:

```
result = ' '.join(colors)
```

Or commas and spaces:

```
result = ', '.join(colors)
```

Here's a common case:

```
colors = ['red', 'blue', 'green', 'yellow']
print 'Choose', ', '.join(colors[:-1]), \
      'or', colors[-1]
```

To make a nicely grammatical sentence, we want commas between all but the last pair of values, where we want the word "or". The slice syntax does the job. The "slice until -1" (`[:-1]`) gives all but the last value, which we join with comma-space.

Of course, this code wouldn't work with corner cases, lists of length 0 or 1.

Output:

```
Choose red, blue, green or yellow.
```

Building Strings, Variations 2

If you need to apply a function to generate the substrings:

```
result = ''.join(fn(i) for i in items)
```

This involves a *generator expression*, which we'll cover later.

If you need to compute the substrings incrementally, accumulate them in a list first:

```
items = []
...
items.append(item)  # many times
...
# items is now complete
result = ''.join(fn(i) for i in items)
```

We accumulate the parts in a list so that we can apply the `join` string method, for efficiency.

Use `in` where possible (1)

Good:

```
for key in d:
    print key
```

- `in` is generally faster.
- This pattern also works for items in arbitrary containers (such as lists, tuples, and sets).
- `in` is also an operator (as we'll see).

Bad:

```
for key in d.keys():
    print key
```

This is limited to objects with a `keys()` method.

Use `in` where possible (2)

But `.keys()` is **necessary** when mutating the dictionary:

```
for key in d.keys():
    d[str(key)] = d[key]
```

`d.keys()` creates a static list of the dictionary keys. Otherwise, you'll get an exception "RuntimeError: dictionary changed size during iteration".

For consistency, use `key in dict`, not `dict.has_key()`:

```
# do this:
if key in d:
    ...do something with d[key]

# not this:
if d.has_key(key):
    ...do something with d[key]
```

This usage of `in` is as an operator.

Dictionary `get` Method

We often have to initialize dictionary entries before use:

This is the naïve way to do it:

```
navs = {}
for (portfolio, equity, position) in data:
    if portfolio not in navs:
        navs[portfolio] = 0
    navs[portfolio] += position * prices[equity]
```

`dict.get(key, default)` removes the need for the test:

```
navs = {}
for (portfolio, equity, position) in data:
    navs[portfolio] = (navs.get(portfolio, 0)
                       + position * prices[equity])
```

Much more direct.

Dictionary `setdefault` Method (1)

Here we have to initialize mutable dictionary values. Each dictionary value will be a list. This is the naïve way:

Initializing mutable dictionary values:

```
equities = {}
for (portfolio, equity) in data:
    if portfolio in equities:
        equities[portfolio].append(equity)
    else:
        equities[portfolio] = [equity]
```

`dict.setdefault(key, default)` does the job much more efficiently:

```
equities = {}
for (portfolio, equity) in data:
    equities.setdefault(portfolio, []).append(
                                         equity)
```

`dict.setdefault()` is equivalent to "get, or set & get". Or "set if necessary, then get". It's especially efficient if your dictionary key is expensive to compute or long to type.

The only problem with `dict.setdefault()` is that the default value is always evaluated, whether needed or not. That only matters if the default value is expensive to compute.

If the default value **is** expensive to compute, you may want to use the `defaultdict` class, which we'll cover shortly.

Dictionary `setdefault` Method (2)

Here we see that the `setdefault` dictionary method can also be used as a stand-alone statement:

`setdefault` can also be used as a stand-alone statement:

```
navs = {}
for (portfolio, equity, position) in data:
    navs.setdefault(portfolio, 0)
    navs[portfolio] += position * prices[equity]
```

The `setdefault` dictionary method returns the default value, but we ignore it here. We're taking advantage of `setdefault`'s side effect, that it sets the dictionary value only if there is no value already.

`defaultdict`

`defaultdict` is new in Python 2.5, part of the `collections` module. `defaultdict` is identical to regular dictionaries, except for two things:

- it takes an extra first argument: a default factory function; and
- when a dictionary key is encountered for the first time, the default factory function is called and the result used to initialize the dictionary value.

There are two ways to get `defaultdict`:

- import the `collections` module and reference it via the module,
- or import the `defaultdict` name directly:

```
import collections
```

```
d = collections.defaultdict(...)

from collections import defaultdict
d = defaultdict(...)
```

Here's the example from earlier, where each dictionary value must be initialized to an empty list, rewritten as with `defaultdict`:

```
from collections import defaultdict

equities = defaultdict(list)
for (portfolio, equity) in data:
    equities[portfolio].append(equity)
```

There's no fumbling around at all now. In this case, the default factory function is `list`, which returns an empty list.

This is how to get a dictionary with default values of 0: use `int` as a default factory function:

```
navs = defaultdict(int)
for (portfolio, equity, position) in data:
    navs[portfolio] += position * prices[equity]
```

You should be careful with `defaultdict` though. You cannot get `KeyError` exceptions from properly initialized `defaultdict` instances. You have to use a "key in dict" conditional if you need to check for the existence of a specific key.

Testing for Truth Values

```
# do this:        # not this:
if x:             if x == True:
    pass              pass
```

It's elegant and efficient to take advantage of the intrinsic truth values (or Boolean values) of Python objects.

Testing a list:

```
# do this:        # not this:
if items:         if len(items) == 0:
    pass              pass

                  # and definitely not this:
                  if items == []:
                      pass
```

Truth Values

The `True` and `False` names are built-in instances of type `bool`, Boolean values. Like `None`, there is only one instance of each.

False	**True**
`False` (== 0)	`True` (== 1)
`""` (empty string)	any string but `""` (`" "`, `"anything"`)
`0`, `0.0`	any number but `0` (1, 0.1, -1, 3.14)
`[]`, `()`, `{}`, `set()`	any non-empty container (`[0]`, `(None,)`, `['']`)
`None`	almost any object that's not explicitly False

Example of an object's truth value:

```
>>> class C:
...  pass
...
>>> o = C()
>>> bool(o)
True
>>> bool(C)
True
```

(Examples: execute truth.py.)

To control the truth value of instances of a user-defined class, use the `__nonzero__` or `__len__` special methods. Use `__len__` if your class is a container which has a length:

```
class MyContainer(object):

    def __init__(self, data):
        self.data = data

    def __len__(self):
        """Return my length."""
        return len(self.data)
```

If your class is not a container, use `__nonzero__`:

```
class MyClass(object):

    def __init__(self, value):
        self.value = value

    def __nonzero__(self):
        """Return my truth value (True or False)."""
        # This could be arbitrarily complex:
        return bool(self.value)
```

In Python 3.0, `__nonzero__` has been renamed to `__bool__` for consistency with the `bool` built-in type. For compatibility, add this to the class definition:

```
__bool__ = __nonzero__
```

Index & Item (1)

Here's a cute way to save some typing if you need a list of words:

```
>>> items = 'zero one two three'.split()
>>> print items
['zero', 'one', 'two', 'three']
```

Say we want to iterate over the items, and we need both the item's index and the item itself:

```
                  - or -
i = 0
for item in items:      for i in range(len(items)):
    print i, item               print i, items[i]
    i += 1
```

Index & Item (2): `enumerate`

The `enumerate` function takes a list and returns (index, item) pairs:

```
>>> print list(enumerate(items))
[(0, 'zero'), (1, 'one'), (2, 'two'), (3, 'three')]
```

We need use a `list` wrapper to print the result because `enumerate` is a lazy function: it generates one item, a pair, at a time, only when required. A `for` loop is one place that requires one result at a time. `enumerate` is an example of a *generator*, which we'll cover in greater detail later. `print` does not take one result at a time -- we want the entire result, so we have to explicitly convert the generator into a list when we print it.

Our loop becomes much simpler:

```
for (index, item) in enumerate(items):
    print index, item

# compare:                # compare:
index = 0                 for i in range(len(items)):
for item in items:                print i, items[i]
    print index, item
    index += 1
```

The `enumerate` version is much shorter and simpler than the version on the left, and much easier to read and understand than either.

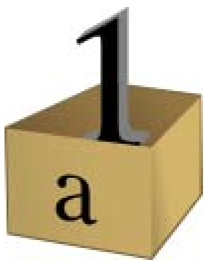
An example showing how the `enumerate` function actually returns an iterator (a generator is a kind of iterator):

```
>>> enumerate(items)
<enumerate object at 0x011EA1C0>
>>> e = enumerate(items)
>>> e.next()
(0, 'zero')
>>> e.next()
(1, 'one')
>>> e.next()
(2, 'two')
>>> e.next()
(3, 'three')
>>> e.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

Other languages have "variables"

In many other languages, assigning to a variable puts a value into a box.

```
int a = 1;
```



Box "a" now contains an integer 1.

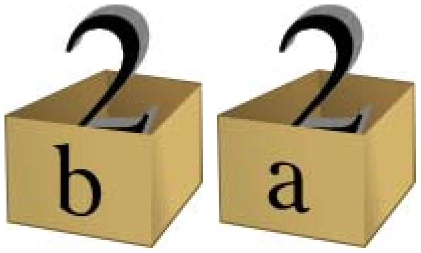
Assigning another value to the same variable replaces the contents of the box:

```
a = 2;
```

Now box "a" contains an integer 2.

Assigning one variable to another makes a copy of the value and puts it in the new box:

```
int b = a;
```

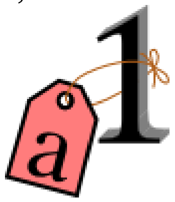


"b" is a second box, with a copy of integer 2. Box "a" has a separate copy.

Python has "names"

In Python, a "name" or "identifier" is like a parcel tag (or nametag) attached to an object.

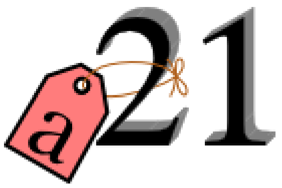
```
a = 1
```



Here, an integer 1 object has a tag labelled "a".

If we reassign to "a", we just move the tag to another object:

```
a = 2
```

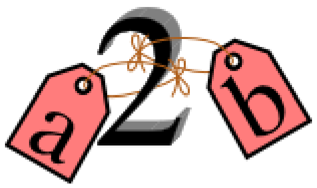


Now the name "a" is attached to an integer 2 object.

The original integer 1 object no longer has a tag "a". It may live on, but we can't get to it through the name "a". (When an object has no more references or tags, it is removed from memory.)

If we assign one name to another, we're just attaching another nametag to an existing object:

```
b = a
```



The name "b" is just a second tag bound to the same object as "a".

Although we commonly refer to "variables" even in Python (because it's common terminology), we really mean "names" or "identifiers". In Python, "variables" are nametags for values, not labelled boxes.

If you get nothing else out of this tutorial, I hope you understand how Python names work. A good understanding is certain to pay dividends, helping you to avoid cases like this:

Default Parameter Values

This is a common mistake that beginners often make. Even more advanced programmers make this mistake if they don't understand Python names.

```
def bad_append(new_item, a_list=[]):
    a_list.append(new_item)
    return a_list
```

The problem here is that the default value of `a_list`, an empty list, is evaluated at function definition time. So every time you call the function, you get the **same** default value. Try it several times:

```
>>> print bad_append('one')
['one']

>>> print bad_append('two')
['one', 'two']
```

Lists are a mutable objects; you can change their contents. The correct way to get a default list (or dictionary, or set) is to create it at run time instead, **inside the function**:

```
def good_append(new_item, a_list=None):
    if a_list is None:
        a_list = []
    a_list.append(new_item)
    return a_list
```

% String Formatting

Python's `%` operator works like C's `sprintf` function. Although if you don't know C, that's not very helpful. Basically, you provide a template or format and interpolation values.

In this example, the template contains two conversion specifications: "%s" means "insert a string here", and "%i" means "convert an integer to a string and insert here". "%s" is particularly useful because it uses Python's built-in `str()` function to to convert any object to a string.

The interpolation values must match the template; we have two values here, a tuple.

```
name = 'David'
messages = '3'
text = ('Hello %s, you have %i messages'
        % (name, messages))
print text
```

Output:

```
Hello David, you have 3 messages
```

Details are in the *Python Library Reference*, section 2.3.6.2, "String Formatting Operations". Bookmark this one!
If you haven't done it already, go to python.org, download the HTML documentation (in a .zip file or a tarball), and install it on your machine. There's nothing like having the definitive resource at your fingertips.

Advanced % String Formatting

What many people don't realize is that there are other, more flexible ways to do string formatting:

By name with a dictionary:

```
values = {'name': name, 'messages': messages}
print ('Hello %(name)s, you have %(messages)i '
       'messages' % values)
```

Here we specify the names of interpolation values, which are looked up in the supplied dictionary.

Notice any redundancy? The names "name" and "messages" are already defined in the local namespace. We can take advantage of this.

By name using the local namespace:

```
print ('Hello %(name)s, you have %(messages)i '
       'messages' % locals())
```

The `locals()` function returns a dictionary of all locally-available names.

This is very powerful. With this, you can do all the string formatting you want without having to worry about matching the interpolation values to the template.

But power can be dangerous. ("With great power comes great responsibility.") If you use the `locals()` form with an externally-supplied template string, you expose your entire local namespace to the caller. This is just something to keep in mind.

To examine your local namespace:

```
>>> from pprint import pprint
>>> pprint(locals())
```

`pprint` is a very useful module. If you don't know it already, try playing with it. It makes debugging your data structures much easier!

List Comprehensions

List comprehensions ("listcomps" for short) are syntax shortcuts for this general pattern:

The traditional way, with `for` and `if` statements:

```
new_list = []
for item in a_list:
    if condition(item):
        new_list.append(fn(item))
```

As a list comprehension:

```
new_list = [fn(item) for item in a_list
            if condition(item)]
```

Listcomps are clear & concise, up to a point. You can have multiple `for`-loops and `if`-conditions in a listcomp, but beyond two or three total, or if the conditions are complex, I suggest that regular `for` loops should be used. Applying the Zen of Python, choose the more readable way.

For example, a list of the squares of 0–9:

```
>>> [n ** 2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A list of the squares of odd 0–9:

```
>>> [n ** 2 for n in range(10) if n % 2]
[1, 9, 25, 49, 81]
```

Generator Expressions (1)

Let's sum the squares of the numbers up to 100:

As a loop:

```
total = 0
for num in range(1, 101):
    total += num * num
```

We can use the `sum` function to quickly do the work for us, by building the appropriate sequence.

As a list comprehension:

```
total = sum([num * num for num in range(1, 101)])
```

As a generator expression:

```
total = sum(num * num for num in xrange(1, 101))
```

Generator expressions ("genexps") are just like list comprehensions, except that where listcomps are greedy, generator expressions are lazy. Listcomps compute the entire result list all at once, as a list. Generator expressions compute one value at a time, when needed, as individual values. This is especially useful for long sequences where the computed list is just an intermediate step and not the final result.

In this case, we're only interested in the sum; we don't need the intermediate list of squares. We use `xrange` for the same reason: it lazily produces values, one at a time.

Generator Expressions (2)

For example, if we were summing the squares of several billion integers, we'd run out of memory with list comprehensions, but generator expressions have no problem. This does take time, though!

```
total = sum(num * num
            for num in xrange(1, 1000000000))
```

The difference in syntax is that listcomps have square brackets, but generator expressions don't. Generator expressions sometimes do require enclosing parentheses though, so you should always use them.

Rule of thumb:

- Use a list comprehension when a computed list is the desired end result.
- Use a generator expression when the computed list is just an intermediate step.

Here's a recent example I saw at work.

We needed a dictionary mapping month numbers (both as string and as integers) to month codes for futures contracts. It can be done in one logical line of code.

The way this works is as follows:

- The `dict()` built-in takes a list of key/value pairs (2-tuples).
- We have a list of month codes (each month code is a single letter, and a string is also just a list of letters). We enumerate over this list to get both the month code and the index.
- The month numbers start at 1, but Python starts indexing at 0, so the month number is one more than the index.
- We want to look up months both as strings and as integers. We can use the `int()` and `str()` functions to do this for us, and loop over them.

Recent example:

```
month_codes = dict((fn(i+1), code)
    for i, code in enumerate('FGHJKMNQUVXZ')
    for fn in (int, str))
```

`month_codes` result:

```
{ 1:  'F',  2:  'G',  3:  'H',  4:  'J', ...
 '1': 'F', '2': 'G', '3': 'H', '4': 'J', ...}
```

Sorting

It's easy to sort a list in Python:

```
a_list.sort()
```

(Note that the list is sorted in-place: the original list is sorted, and the `sort` method does **not** return the list or a copy.)

But what if you have a list of data that you need to sort, but it doesn't sort naturally (i.e., sort on the first column, then the second column, etc.)? You may need to sort on the second column first, then the fourth column.

We can use list's built-in `sort` method with a custom function:

```
def custom_cmp(item1, item2):
    return cmp((item1[1], item1[3]),
               (item2[1], item2[3]))

a_list.sort(custom_cmp)
```

This works, but it's extremely slow for large lists.

Sorting with DSU

DSU = Decorate-Sort-Undecorate

Instead of creating a custom comparison function, we create an auxiliary list that *will* sort naturally:

```
# Decorate:
to_sort = [(item[1], item[3], item)
           for item in a_list]

# Sort:
to_sort.sort()

# Undecorate:
a_list = [item[-1] for item in to_sort]
```

The first line creates a list containing tuples: copies of the sort terms in priority order, followed by the complete data record.

The second line does a native Python sort, which is very fast and efficient.

The third line retrieves the **last** value from the sorted list. Remember, this last value is the complete data record. We're throwing away the sort terms, which have done their job and are no longer needed.

This is a tradeoff of space and complexity against time. Much simpler and faster, but we do need to duplicate the original list.

Generators

We've already seen generator expressions. We can devise our own arbitrarily complex generators, as functions:

```
def my_range_generator(stop):
    value = 0
    while value < stop:
        yield value
        value += 1

for i in my_range_generator(10):
    do_something(i)
```

The `yield` keyword turns a function into a generator. When you call a generator function, instead of running the code immediately Python returns a generator object, wich is an iterator; it has a `next` method. `for` loops just call the `next` method on the iterator, until a `StopIteration` exception is raised. You can raise `StopIteration` explicitly, or implicitly by falling off the end of the generator code as above.

Generators can simplify sequence/iterator handling, because we don't need to build concrete lists; just compute one value at ta time. The generator function maintains state.

This is how a `for` loop really works. Python looks at the sequence supplied after the `in` keyword. If it's a simple container (such as a list, tuple, dictionary, set, or user-defined container) Python converts it into an iterator. If it's already an iterator, Python does nothing.

Then Python repeatedly calls the iterator's `next` method, assigns the return value to the loop counter (`i` in this case), and executes the indented code. This is repeated over and over, until `StopIteration` is raised, or a `break` statement is executed in the code.

A `for` loop can have an `else` clause, whose code is executed after the iterator runs dry, but **not** after a `break` statement is executed. This distinction allows for some elegant uses. `else` clauses are not always or often used on `for` loops, but they can come in handy. Sometimes an `else` clause perfectly expresses the logic you need.

For example, if we need to check that a condition holds on some item, any item, in a sequence:

```
for item in sequence:
    if condition(item):
        break
else:
    raise Exception('Condition not satisfied.')
```

Example Generator

Filter out blank rows from a CSV reader (or items from a list):

```
def filter_rows(row_iterator):
    for row in row_iterator:
        if row:
            yield row

data_file = open(path, 'rb')
irows = filter_rows(csv.reader(data_file))
```

Reading Lines From Text/Data Files

```
datafile = open('datafile')
for line in datafile:
    do_something(line)
```

This is possible because files support a `next` method, as do other iterators: lists, tuples, dictionaries (for their keys), generators.

There is a caveat here: because of the way the buffering is done, you cannot mix `.next` & `.read*` methods unless you're using Python 2.5+.

EAFP vs. LBYL

It's easier to ask forgiveness than permission

Look before you leap

Generally EAFP is preferred, but not always.

- Duck typing

  If it walks like a duck, and talks like a duck, and looks like a duck: it's a duck. (Goose? Close enough.)

- Exceptions

  Use coercion if an object must be a particular type. If `x` must be a string for your code to work, why not call

  ```
  str(x)
  ```

  instead of trying something like

  ```
  isinstance(x, str)
  ```

EAFP `try/except` Example

You can wrap exception-prone code in a `try/except` block to catch the errors, and you will probably end up with a solution that's much more general than if you had tried to anticipate every possibility.

```
try:
    return str(x)
except TypeError:
    ...
```

Note: Always specify the exceptions to catch. Never use bare `except` clauses. Bare `except` clauses will catch unexpected exceptions, making your code exceedingly difficult to debug.

Importing

```
from module import *
```

You've probably seen this "wild card" form of the import statement. You may even like it. **Don't use it.**

Never!

Moral: **don't use wild-card imports!**

The `from module import *` wild-card style leads to namespace pollution. You'll get thing in your local namespace that you didn't expect to get. You may see imported names obscuring module-defined local names. You won't be able to figure out where certain names come from. Although a convenient shortcut, this should not be in production code.

It's much better to:

- reference names through their module (fully qualified identifiers),
- import a long module using a shorter name (alias),
- or explicitly import just the names you need.

Namespace pollution alert!

Instead,

Reference names through their module (fully qualified identifiers):

```
import module
module.name
```

Or import a long module using a shorter name (alias):

```
import long_module_name as mod
mod.name
```

Or explicitly import just the names you need:

```
from module import name
name
```

Modules & Scripts

To make a simultaneously importable module and executable script:

```
if __name__ == '__main__':
    # script code here
```

When imported, a module's `__name__` attribute is set to the module's file name, without ".py". So the code guarded by the `if` statement above will not run when imported. When executed as a script though, the `__name__` attribute is set to "__main__", and the script code *will* run.

Except for special cases, you shouldn't put any major executable code at the top-level. Put code in functions, classes, methods, and guard it with `if __name__ == '__main__'`.

Module Structure

```
"""module docstring"""

# imports
# constants
# exception classes
# interface functions
# classes
# internal functions & classes

def main(...):
    ...

if __name__ == '__main__':
    status = main()
    sys.exit(status)
```

This is how a module should be structured.

Command-Line Processing

Example: `cmdline.py`:

```
#!/usr/bin/env python

"""
Module docstring.
"""

import sys
import optparse

def process_command_line(argv):
    """
    Return a 2-tuple: (settings object, args list).
    `argv` is a list of arguments, or `None` for ``sys.argv[1:]``.
    """
    if argv is None:
        argv = sys.argv[1:]

    # initialize the parser object:
    parser = optparse.OptionParser(
        formatter=optparse.TitledHelpFormatter(width=78),
        add_help_option=None)

    # define options here:
    parser.add_option(      # customized description; put --help last
        '-h', '--help', action='help',
        help='Show this help message and exit.')

    settings, args = parser.parse_args(argv)

    # check number of arguments, verify values, etc.:
    if args:
        parser.error('program takes no command-line arguments; '
                     '"%s" ignored.' % (args,))

    # further process settings & args if necessary

    return settings, args

def main(argv=None):
    settings, args = process_command_line(argv)
    # application code here, like:
    # run(settings, args)
    return 0        # success

if __name__ == '__main__':
    status = main()
    sys.exit(status)
```

Packages

```
package/
    __init__.py
    module1.py
    subpackage/
        __init__.py
        module2.py
```

- Used to organize your project.
- Reduces entries in load-path.
- Reduces import name conflicts.

Example:

```
import package.module1
from packages.subpackage import module2
from packages.subpackage.module2 import name
```

In Python 2.5 we now have absolute and relative imports via a future import:

```
from __future__ import absolute_import
```

I haven't delved into these myself yet, so we'll conveniently cut this discussion short.

Simple is Better Than Complex

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
>
> —Brian W. Kernighan, co-author of *The C Programming Language* and the "K" in "AWK"

In other words, keep your programs simple!

Don't reinvent the wheel

Before writing any code,

- Check Python's standard library.
- Check the Python Package Index (the "Cheese Shop"):

  http://cheeseshop.python.org/pypi
- Search the web. Google is your friend.

References

- "Python Objects", Fredrik Lundh, http://www.effbot.org/zone/python-objects.htm
- "How to think like a Pythonista", Mark Hammond, http://python.net/crew/mwh/hacks/objectthink.html
- "Python main() functions", Guido van Rossum, http://www.artima.com/weblogs/viewpost.jsp?thread=4829
- "Python Idioms and Efficiency", http://jaynes.colorado.edu/PythonIdioms.html
- "Python track: python idioms", http://www.cs.caltech.edu/courses/cs11/material/python/misc/python_idioms.html
- "Be Pythonic", Shalabh Chaturvedi, http://shalabh.infogami.com/Be_Pythonic2
- "Python Is Not Java", Phillip J. Eby, http://dirtsimple.org/2004/12/python-is-not-java.html
- "What is Pythonic?", Martijn Faassen, http://faassen.n--tree.net/blog/view/weblog/2005/08/06/0
- "Sorting Mini-HOWTO", Andrew Dalke, http://wiki.python.org/moin/HowTo/Sorting
- "Python Idioms", http://www.gungfu.de/facts/wiki/Main/PythonIdioms
- "Python FAQs", http://www.python.org/doc/faq/