Jeff Knupp

PYTHON PROGRAMMER

Everything I know about Python...

Learn to Write Pythonic Code!

Check out the book *Writing Idiomatic Python*!

Looking for Python Tutoring? Remote and local (NYC) slots still available! Email me at jeff@jeffknupp.com for more info.

Writing Idiomatic Python

As someone who evangelizes Python at work, I read a lot of code written by professional programmers new to Python. I've written a good amount of Python code in my time, but I've certainly *read* far more. The single quickest way to increase maintainability and decrease 'simple' bugs is to strive to write *idiomatic* Python. Whereas some dynamic languages embrace the idea there being no 'right' way to solve a problem, the Python community generally appreciates the liberal use of 'Pythonic' solutions to problems. 'Pythonic' refers to the principles laid out in 'The Zen of Python' (try typing 'import this' in an interpreter...). One of those principles is

```
'There should be one-- and preferably only one --obvious way to do it'
                                        -from 'The Zen of Python' by Tim Peters
```

In that vein, I've begun compiling a list of Python idioms that programmers coming from other languages may find helpful. I know there are a ton of things not on here; it's merely a skeleton list that I'll add to over time. If you have a specific idiom you think should be added, let me know in the comments and I'll add it with attribution to the name you use in your comment.

This list will temporarily live here as a blog post, but I have an interesting idea for its final home. More on that next week.

Update: The 'Writing Idiomatic Python' e-Book is here!

See here for details!

Update 10/05/12: Add context managers, PEP8, itertools, string join(), dict.get() default values

Idioms

Formatting

Python has a language-defined standard set of formatting rules known as PEP8. If you're browsing commit messages on Python projects, you'll likely find them littered with references to PEP8 cleanup. The reason is simple: if we all agree on a common set of naming and formatting conventions, Python code as a whole becomes instantly more accessible to both novice and experienced developers. PEP8 is perhaps the most explicit example of idioms within the Python community. Read

the PEP, install a PEP8 style-checking plugin for your editor (they all have one), and start writing your code in a way that other Python developers will appreciate. Listed below are a few examples.

Identifier Type	Format	Example
Class	Camel case	class StringManipulator:
Variable	Words joined by underscore	words_joined_by_underscore = True
Function	Words joined by underscore	def are_words_joined_by_underscore(words):
'Internal' class members/functions	Prefixed by single underscore	def _update_statistics(self):

Unless wildly unreasonable, abbreviations should not be used (acronyms are fine if in common use, like 'HTTP')

Working With Data

Avoid using a temporary variable when swapping two variables

There is no reason to swap using a temporary variable in Python. We can use tuples to make our intention more clear.

Harmful

```
temp = foo
foo = bar
bar = temp
```

Idiomatic

```
(foo, bar) = (bar, foo)
```

Use tuples to unpack data

In Python, it is possible to 'unpack' data for multiple assignment. Those familiar with LISP may know this as 'desctructuring bind'.

Harmful

```
list_from_comma_separated_value_file = ['dog', 'Fido', 10]
animal = list_from_comma_separated_value_file[0]
```

```
name = list_from_comma_separated_value_file[1]
age = list_from_comma_separated_value_file[2]
```

Idiomatic

```
list_from_comma_separated_value_file = ['dog', 'Fido', 10]
(animal, name, age) = list_from_comma_separated_value_file
```

Use ".join when creating a single string for list elements

It's faster, uses less memory, and you'll see it everywhere anyway. Note that the two quotes represent the delimiter between list elements in the string we're creating. `''` just means we mean to concatenate the elements with no characters between them.

Harmful

```
result_list = ['True', 'False', 'File not found']
result_string = ''
for result in result_list:
    result_string += result
```

Idiomatic

```
result_list = ['True', 'False', 'File not found']
result_string = ''.join(result_list)
```

Use the 'default' parameter of dict.get() to provide default values

Often overlooked in the `get()` definition is the `default` parameter. Without using `default` (or the `collections.defaultdict` class), your code will be littered with confusing if statements. Remember, strive for clarity.

Harmful

```
log_severity = None
if 'severity' in configuration:
    log_severity = configuration['severity']
```

```
else:
    log_severity = log.Info
```

Idiomatic

```
log_severity = configuration.get('severity', log.Info)
```

Use Context Managers to ensure resources are properly managed

Similar to the RAII principle in languages like C++ and D, context managers (objects meant to be used with the `with` statement) can make resource management both safer and more explicit. The canonical example is file IO.

Harmful

```
file_handle = open(path_to_file, 'r')
for line in file_handle.readlines():
    if some_function_that_throws_exceptions(line):
        # do something
file_handle.close()
```

Idiomatic

```
with open(path_to_file, 'r') as file_handle:
    for line in file_handle:
        if some_function_that_throws_exceptions(line):
            # do something
# No need to explicitly call 'close'. Handled by the File context manager
```

In the Harmful code above, what happens if `some_function_that_throws_exceptions` does, in fact, throw an exception? Since we haven't caught it in the code listed, it will propagate up the stack. We've hit an exit point in our code that might have been overlooked, and we now have no way to close the opened file. In addition to those in the standard libraries (for working with things like file IO, synchronization, managing mutable state) developers are free to create their own.

Learn the contents of the itertools module

If you frequent sites like StackOverflow, you may notice that the answer to questions of the form "Why doesn't Python have the following obviously useful library function?" almost always references the itertools module. The functional programming stalwarts that itertools provides should be seen as fundamental building blocks. What's more, the documentation for itertools has a 'Recipes' section that provides idiomatic implementations of common functional programming constructs, all created using the itertools module. For some reason, a vanishingly small number of Python developers seem to be aware of the 'Recipes' section and, indeed, the itertools module in general (hidden gems in the Python documentation is actually a recurring theme). Part of writing idiomatic code is knowing when you're reinventing the wheel.

Control Structures

If Statement

Avoid placing conditional branch on the same line as the colon

Using indentation to indicate scope (like you already do everywhere else in Python) makes it easy to determine what will be executed as part of a conditional statement.

Harmful

```
if name: print (name)
print address
```

Idiomatic

```
if name:
    print (name)
print address
```

Avoid having multiple statements on a single line

Though the language definition allows one to use a semi-colon to delineate statements, doing so without reason makes one's code harder to read. Typically violated with the previous rule.

Harmful

```
if this_is_bad_code: rewrite_code(); make_it_more_readable();
```

Idiomatic

```
if this_is_bad_code:
    rewrite_code()
    make_it_more_readable()
```

Avoid repeating variable name in compound if Statement

When one wants to check against a number of values, repeatedly listing the variable whose value is being checked is unnecessarily verbose. Using a temporary collection makes the intention clear.

Harmful

```
if name == 'Tom' or name == 'Dick' or name == 'Harry':
    is_generic_name = True
```

Idiomatic

```
if name in ('Tom', 'Dick', 'Harry'):
    is_generic_name = True
```

Use list comprehensions to create lists that are subsets of existing data

List comprehensions, when used judiciously, increase clarity in code that builds a list from existing data. Especially when data is both checked for some condition *and* transformed in some way, list comprehensions make it clear what's happening. There are also (usually) performance benefits to using list comprehensions (or alternately, set comprehensions) due to optimizations in the CPython interpreter.

Harmful

```
some_other_list = range(1, 100)
my_weird_list_of_numbers = list()
for element in some_other_list:
```

```
    if is_prime(element):
        my_weird_list_of_numbers.append(element+5)
```

Idiomatic

```
some_other_list = range(1, 100)
my_weird_list_of_numbers = [element + 5 for element in some_other_list if is_prime(element)]
```

Loops

Use the *in* keyword to iterate over an Iterable

Programmers coming languages lacking a for_each style construct are used to iterating over a container by accessing elements via index. Python's `in` keyword handles this gracefully.

Harmful

```
my_list = ['Larry', 'Moe', 'Curly']
index = 0
while index < len(my_list):
    print (my_list[index])
    index+=1
```

Idiomatic

```
my_list = ['Larry', 'Moe', 'Curly']
for element in my_list:
    print element
```

Use the *enumerate* function in loops instead of creating an 'index' variable

Programmers coming from other languages are used to explicitly declaring a variable to track the index of a container in a loop. For example, in C++:

```
for (int i=0; i < container.size(); ++i)
{
```

```
    // Do stuff
}
```

In Python, the `enumerate` built-in function handles this role.

Harmful

```
index = 0
for element in my_container:
    print (index, element)
    index+=1
```

Idiomatic

```
for index, element in enumerate(my_container):
    print (index, element)
```

Posted on Oct 04, 2012 by Jeff Knupp

« Software Optimization: A Systematic Approach, Part Two

Subscribe