**Python: Myths about Indentation**

|  |
| --- |
| Note: Lines beginning with ">>>" and "..." indicate input to Python (these are the default prompts of the interactive interpreter). Everything else is output from Python. |

There are quite some prejudices and myths about Python's indentation rules among people who don't really know Python. I'll try to address a few of these concerns on this page.

**"Whitespace is significant in Python source code."**

No, not in general. Only the indentation level of your statements is significant (i.e. the whitespace at the very left of your statements). Everywhere else, whitespace is not significant and can be used as you like, just like in any other language. You can also insert empty lines that contain nothing (or only arbitrary whitespace) anywhere.

Also, the exact amount of indentation doesn't matter at all, but only the relative indentation of nested blocks (relative to each other).

Furthermore, the indentation level is ignored when you use explicit or implicit continuation lines. For example, you can split a list across multiple lines, and the indentation is completely insignificant. So, if you want, you can do things like this:

|  |  |  |  |
| --- | --- | --- | --- |
|  | >>> foo = [ ...            'some string', ...         'another string', ...           'short string' ... ] >>> print foo   |  | | --- | | ['some string', 'another string', 'short string'] |   >>> bar = 'this is ' \ ...       'one long string ' \ ...           'that is split ' \ ...     'across multiple lines' >>> print bar   |  | | --- | | this is one long string that is split across multiple lines | |

**"Python forces me to use a certain indentation style."**

Yes and no. First of all, you can write the inner block all on one line if you like, therefore not having to care about intendation at all. The following three versions of an "if" statement are all valid and do exactly the same thing (output omitted for brevity):

|  |  |
| --- | --- |
|  | >>> if 1 + 1 == 2: ...     print "foo" ...     print "bar" ...     x = 42  >>> if 1 + 1 == 2: ...     print "foo"; print "bar"; x = 42  >>> if 1 + 1 == 2: print "foo"; print "bar"; x = 42 |

Of course, most of the time you will want to write the blocks in separate lines (like the first version above), but sometimes you have a bunch of similar "if" statements which can be conveniently written on one line each.

If you decide to write the block on separate lines, then yes, Python forces you to obey its indentation rules, which simply means: The enclosed block (that's two "print" statements and one assignment in the above example) have to be indented more than the "if" statement itself. That's it. And frankly, would you really want to indent it in any other way? I don't think so.

So the conclusion is: Python forces you to use indentation that you would have used anyway, unless you wanted to obfuscate the structure of the program. In other words: Python does not allow to obfuscate the structure of a program by using bogus indentations. In my opinion, that's a very good thing.

Have you ever seen code like this in C or C++?

|  |  |
| --- | --- |
|  | /\*  Warning:  bogus C code!  \*/  if (some condition)         if (another condition)                 do\_something(fancy); else         this\_sucks(badluck); |

Either the indentation is wrong, or the program is buggy, because an "else" always applies to the nearest "if", unless you use braces. This is an essential problem in C and C++. Of course, you could resort to always use braces, no matter what, but that's tiresome and bloats the source code, and it doesn't prevent you from accidentally obfuscating the code by still having the wrong indentation. (And that's just a very simple example. In practice, C code can be much more complex.)

In Python, the above problems can never occur, because indentation levels and logical block structure are always consistent. The program always does what you expect when you look at the indentation.

Quoting the famous book writer Bruce Eckel:

Because blocks are denoted by indentation in Python, indentation is uniform in Python programs. And indentation is meaningful to us as readers. So because we have consistent code formatting, I can read somebody else's code and I'm not constantly tripping over, "Oh, I see. They're putting their curly braces here or there." I don't have to think about that.

**"You cannot safely mix tabs and spaces in Python."**

That's right, and you don't want that. To be exact, you cannot safely mix tabs and spaces in C either: While it doesn't make a difference to the compiler, it can make a big difference to humans looking at the code. If you move a piece of C source to an editor with different tabstops, it will all look wrong (and possibly behave differently than it looks at first sight). You can easily introduce well-hidden bugs in code that has been mangled that way. That's why mixing tabs and spaces in C isn't really "safe" either. Also see the "bogus C code" example above.

Therefore, it is generally a good idea not to mix tabs and spaces for indentation. If you use tabs only or spaces only, you're fine.

Furthermore, it can be a good idea to avoid tabs alltogether, because the semantics of tabs are not very well-defined in the computer world, and they can be displayed completely differently on different types of systems and editors. Also, tabs often get destroyed or wrongly converted during copy&paste operations, or when a piece of source code is inserted into a web page or other kind of markup code.

Most good editors support transparent translation of tabs, automatic indent and dedent. That is, when you press the tab key, the editor will insert enough spaces (not actual tab characters!) to get you to the next position which is a multiple of eight (or four, or whatever you prefer), and some other key (usually Backspace) will get you back to the previous indentation level.

In other words, it's behaving like you would expect a tab key to do, but still maintaining portability by using spaces in the file only. This is convenient and safe.

Having said that -- If you know what you're doing, you can of course use tabs and spaces to your liking, and then use tools like "expand" (on UNIX machines, for example) before giving the source to others. If you use tab characters, Python assumes that tab stops are eight positions apart.

**"I just don't like it."**

That's perfectly OK; you're free to dislike it (and you're probably not alone). Granted, the fact that indentation is used to indicate the block structure might be regarded as uncommon and requiring to get used to it, but it does have a lot of advantages, and you get used to it very quickly when you seriously start programming in Python.

Having said that, you *can* use keywords to indicate the end of a block (instead of indentation), such as "endif". These are not really Python keywords, but there is a tool that comes with Python which converts code using "end" keywords to correct indentation and removes those keywords. It can be used as a pre-processor to the Python compiler. However, no real Python programmer uses it, of course.   
*[Update]* It seems this tool has been removed from recent versions of Python. Probably because nobody really used it.

**"How does the compiler parse the indentation?"**

The parsing is well-defined and quite simple. Basically, changes to the indentation level are inserted as tokens into the token stream.

The lexical analyzer (tokenizer) uses a stack to store indentation levels. At the beginning, the stack contains just the value 0, which is the leftmost position. Whenever a nested block begins, the new indentation level is pushed on the stack, and an "INDENT" token is inserted into the token stream which is passed to the parser. There can never be more than one "INDENT" token in a row.

When a line is encountered with a smaller indentation level, values are popped from the stack until a value is on top which is equal to the new indentation level (if none is found, a syntax error occurs). For each value popped, a "DEDENT" token is generated. Obviously, there can be multiple "DEDENT" tokens in a row.

At the end of the source code, "DEDENT" tokens are generated for each indentation level left on the stack, until just the 0 is left.

Look at the following piece of sample code:

|  |  |
| --- | --- |
|  | >>> if foo: ...     if bar: ...         x = 42 ... else: ...   print foo ... |

In the following table, you can see the tokens produced on the left, and the indentation stack on the right.

|  |  |
| --- | --- |
|  | <if> <foo> <:>                    [0] <INDENT> <if> <bar> <:>           [0, 4] <INDENT> <x> <=> <42>             [0, 4, 8] <DEDENT> <DEDENT> <else> <:>      [0] <INDENT> <print> <foo>            [0, 2] <DEDENT>                          [0] |

Note that after the lexical analysis (before parsing starts), there is *no* whitespace left in the list of tokens (except possibly within string literals, of course). In other words, the indentation is handled by the lexer, not by the parser.

The parser then simply handles the "INDENT" and "DEDENT" tokens as block delimiters -- exactly like curly braces are handled by a C compiler.

The above example is intentionally simple. There are more things to it, such as continuation lines. They are well-defined, too, and you can read about them in the Python Language Reference if you're interested, which includes a complete formal grammar of the language.

[[HTML 4.01]](http://validator.w3.org/check?uri=http://www.secnetix.de/olli/Python/block_indentation.hawk)