Sanity vs. Invisible Markings

TABS VS. SPACES

Dear KV,

My team resurrected some old Python code and brought it up to version 3. The process was made worse by the new restriction of not mixing tabs and spaces in the source code. An automatic cleanup that allowed the code to execute by replacing the tabs with spaces caused a lot of havoc with the comments at the ends of lines. Why does anyone make a language in which white space matters this much?

White Out

Dear White,

Ever edited a makefile? Although there is a long tradition of the significant use of white space in programming languages, all traditions need to change. We no longer sacrifice virgins to make the printer work, or at least KV hasn't sacrificed one recently.

In Python, many people have taken issue with the choice to have white space—and not braces—to indicate the limits of blocks of code, but since the developers didn't change their minds on this with version 3 of Python, I suspect we're all stuck with it for quite a bit longer, and I am quite sure that there will be other languages, big and small, where white space remains significant.

If I could change one thing in the minds of all programming language designers, it would be to impress

upon them—forcefully—the idea that anything that is significant to the syntactic or structural meaning of a program must be easily visible to the human reader, as well as easily understood by the systems with which koders write code.

Let's deal with that last point first. Making it easy for tools to understand the structure of software is one of the keys to having tools that help programmers prepare proper programs for computers. Since the earliest days of software development, programmers have tried to build tools that show them—before the inevitable edit-compile-test-fail-edit endless loop—where there might be issues in the program text. Code editors have added colorization, syntax highlighting, folding, and a host of other features in a desperate, and some might say fruitless, attempt to improve the productivity of programmers.

When a new language comes along, it is important for these signifiers in the code to be used consistently; otherwise your editor of choice has little or no ability to deploy these helpful hints to improve productivity. Allowing any two symbols to represent the same concept, for example, is a definite no-no. Imagine if you could have two types of braces to delineate blocks of code, just because two different parts of the programming community wanted them, or if there were multiple syntactic ways to dereference a variable. The basic idea is that there must be one clear way to do each thing that a language must do, both for human understanding *and* for the sanity of editor developers. Thus, the use of invisible, or near-invisible, markings in code, especially tabs and spaces, to indicate structure or syntax.

Invisible and near-invisible markings bring us to the human part of the problem—not that code editor authors aren’t human, but most of us will not write new editors, though all of us will use editors. As we all know, once upon a time computers had small memories and the difference between a tab, which is a single byte, and a corresponding number of spaces [8] could be a significant difference between the size of source code stored on a precious disk, and also transferred, over whatever primitive and slow bus, from storage into memory.

Related articles

- [File-system Litter](#)
 Cleaning up your storage space quickly and efficiently
 Kode Vicious
<https://queue.acm.org/detail.cfm?id=2003323>
- [A Generation Lost in the Bazaar](#)
 Quality happens only when someone is responsible for it.
 Poul-Henning Kamp
<https://queue.acm.org/detail.cfm?id=2349257>
- [Demo Data as Code](#)
 Automation helps collaboration.
 Thomas A. Limoncelli
<https://queue.acm.org/detail.cfm?id=3355565>

Changing the coding standard from eight spaces to four might improve things, but let’s face it, none of this has mattered for several decades. Now, the only reason for the use of these invisible markings is to clearly represent the scope of a piece of code relative to the pieces of code around it.

In point of fact, it would be better to pick a single character that is not a tab and not a space and not normally used in a program—for example, Unicode code point U+1F4A9—and to use that as the universal indentation character. Editors would then

be free to indent code in any consistent way based on the user’s preferences. The user could have any number of

blank characters used per indent character—8, 4, 2, some prime number, whatever they like—and programmers could choose their very own personal views of the scope. On disk, this format would cost only one character (two bytes) per indent, and if you wanted to see the indent characters, a common feature of modern editors, you flip a switch, and voila, there they all are. Everyone would be happy, and we would have finally solved the age-old conundrum of tabs vs. spaces.

KV

George V. Neville-Neil *works on networking and operating-system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.*

Copyright © 2020 held by owner/author. Publication rights licensed to ACM.