COSC 301: Operating Systems
Fall 2019
Homework 2: Tab Nanny: The Python indentation checker
Due: 1 October 2019, 11pm

Python, as you are very likely well aware, uses indentation as a way to identify "code blocks" and the structure of a program. There is a built-in Python module called tabnanny that can check the indentation structure of a Python program, and for this homework you'll basically write a clone of that module. The key data structure you'll use for checking indentation structure is a *stack* of integers. You will implement this as a linked list in C[1].

The basic operation of your program, in the end, should be to accept the name of a Python file as a command-line argument. Your program should check the indentation structure of the file and report on which line it found an inconsistency, or that the "tab nanny" has been satisfied. Before exiting, your program should print the current values (integers) on the stack, from top to bottom.

As with homework 1, please work in groups of 2 for this homework (and all subsequent homeworks). To get started, click on the Github classroom link on Moodle to create a repo in your own account, then clone the repo on birds.cs.colgate.edu. (You will, of course, need to ssh onto birds.cs.colgate.edu to do the cloning step).

Learning Goals

By the end of this homework you should be able to:

1. Work more effectively with C strings and files.
2. Use C pointers to implement a stack as a linked list.
3. Gain more practice working with C control structures including for loops and if statements.

Python indentation and stacks

Two preliminaries:

1. You should assume that all Python programs are indented using plain spaces only, or with tabs only, but that they are not mixed in the same program. By the way, tabs are

[1] The Python interpreter is written in C, which is pretty fun to think about. Most of the data structures you use in Python, e.g., lists, dictionaries, strings, etc., have at least part of their implementation in C. If you're curious about some of the details for how modules written in C can get used from Python, take a look at some Python documentation on the subject: https://docs.python.org/3/extending/index.html.

evil[2].

2. In the description below, the term *column* is used to indicate the position of a character within one line of text. Columns start at 0 and can be arbitrarily large, but you can assume that a given column number is no bigger than 1022. This implies that the maximum (assumed) length of a line can fit into a buffer of size 1024, including the newline and null-termination characters.

Every Python program file *must* have its first non-blank line start at column 0. For example, the following program is invalid in Python (note that each space is denoted by the special character ␣ to facilitate the explanation):

```
␣␣␣print("Hello, bad indentation") # notice the 3 spaces beginning the line
```

This example shows correct starting indentation:

```
print("Hello, correct indentation") # starting column 0
```

The starting column (aka "indentation") of any non-blank line must be *identical* to the previous line unless the previous line ends in `:` (colon), in which case the indentation of the current line must be strictly greater than the previous line[3]. If the previous line does not end in a colon, checking whether the current line is at the correct indentation level (i.e., identical to the previous line) is straightforward: the indentation should match the value at the top of the stack.

The following example is illustrative of these situations:

```
print("I am correctly indented")
if True:
␣␣␣␣print("I always print")
```

In the example above, the first and second lines are indented at column 0 (far left). Since the first line does not end in a colon, the if statement line should be indented the same. The final print statement must have an indentation strictly greater than that of the if statement since the if statement ends with a colon.

There's one more situation, that of *dedenting*, and the stack data structure is key. Consider if we use a stack to store the current indentation level (e.g., initially 0). When we encounter a colon and the indentation level *increases*, we should *push the new indentation level onto the stack*. For every new line that comes after a line that does *not* have a colon, we check whether its indentation level is the same as the value at the top of the stack. If the indentation of the current line does *not* match the previous line, we continue to pop values off the stack until either

---

[2] https://www.python.org/dev/peps/pep-0008/\#tabs-or-spaces

[3] There is actually another situation that changes indentation processing in Python, which we will ignore in this assignment. When a line ends with \ (backslash), the next line is considered to be a *continuation* of the current line, and thus the indentation should be ignored. Again, you can ignore line continuations in this assignment.

(1) the value at the top of the stack matches the new (dedented) indentation level, or (2) the stack is empty, in which case we declare that there is an indentation error.

Implementing the stack of integers

The first task for this homework is to implement the stack functions `init`, `push`, `pop`, `peek`, and `empty`. All the code to implement these functions should go in `stack.c`. The definitions of stack structures can be found in `stack.h` and are also described here. The stack implementation uses *two* C structs: `struct stack` and `struct stack_node`. The `struct stack` is defined as follows:

```
struct stack {
    struct stack_node *top;
};
```

Notice that a `struct stack` simply contains a pointer to the top of the stack, which is implemented as a pointer to `struct stack_node`. From this, you should infer that the stack will be implemented as a linked list (i.e., each stack node points to the next stack node; the bottom of the stack just points to NULL). Each node in the stack is, of course, implemented as a `struct stack_node`:

```
struct stack_node {
    int value;
    struct stack_node *next;
};
```

You can see from the above struct that each node just has a value (an integer) and a pointer to the next element in the stack.

There is a `teststack.c` file included in the homework repo. Once you implement the stack functions, you can run `./teststack` (after compiling everything!) to test the functions of your stack. If anything goes amiss with your stack, the test program will crash --- this is intended behavior! You can feel free to modify any of the test code if you'd like, in order to test more specific aspects of your implementation.

Some notes on the different stack functions to implement:

- For `init`, consider this: the stack should be empty to begin with. An empty list of nodes on the stack should just be represented with a `NULL` linked list.
- For the `empty` function, consider the above: an empty stack will just be `NULL`.
- For `push`, you'll need to create a new `struct stack_node` and connect it with the struct stack pointer that is passed into push (you'll need to use malloc!). The new element should become the top of the stack.
- For `pop` and `peek`, your code should just assume that there is at least one element on the stack. You should not check that the stack is non-empty before attempting to pop or peek. For pop, remember to free the stack node that is popped before returning the

value that was in the node.

You may not change the type signature of any of the stack functions --- they must remain as defined in `stack.h` and `stack.c`.

If you encounter a "segmentation fault" and program crash when testing your stack functions, it means that you are "trashing memory" in some way. Your best method of debugging will be to use valgrind and read its output carefully.

Using the stack to implement TabNanny

The implementation for the tabnanny indentation checking will go in the file `main.c`. There are some functions started in that file to help get you started:

- The main function has been completely written for you. It checks that the program has been initiated with one command line argument, which should be the name of a file with Python source code to be checked. The main function also opens this file (using fopen) and passes a pointer to the file into the tabnanny function.
- The tabnanny function is where most of what you'll need to implement should go. It already contains some code to create an initialize a stack (called `intstack`). It also has a while loop that will cause the program to loop over all lines of the file. See below for more description of this function.
- The skeleton of a function called `cleanstack` has been written (you'll need to implement it). It should just pop elements off the stack until the stack is empty. This function is already called within tabnanny as the last thing that it does (which is the correct place for it to go).
- The skeleton of a function `first_nonwhitespace` has also been written for you. This function should take a C string (i.e., a line of Python code read from the input file) and find the index of the first non-whitespace character. The value returned from this function is equivalent to the indentation level of the current line of the input file, which you definitely need for implementing tabnanny's logic!

I'd recommend that you implement the two small functions described above (`cleanstack` and `first_nonwhitespace`) first, then work on the main tabnanny logic.

- You'll notice in the tabnanny function that the value 0 is pushed onto the stack as soon as the stack is created and initialized. For any Python program, the first line of the file is expected to be at the first column (column 0), so that's why 0 is already pushed onto the intstack.
- Review the basic logic for tabnanny above. It may help to look at the four example .py files that are included in the repo and trace "manually" how tabnanny should work and what should be on the stack.
- For implementing the tabnanny's basic logic, you might want to write a separate function that finds the last non-whitespace character in a line in order to determine whether a line ends with : (i.e., the colon character), since tabnanny's logic depends on it. Remember (and review the discussion above) that when a colon is found at the end of a line the indentation level of the *next* line should be pushed on the stack *only if the indentation*

level is greater than the value that is currently on the top of the stack (otherwise, tabnanny will not be happy!)

- Any blank line of the input file should just be ignored. In the while loop within tabnanny, I'd recommend that you handle this situation explicitly at the top of the loop. For the line that was just read from the file, if the line is empty then just ignore it and go back to the top of the loop. (The continue statement in C will cause execution to go to the top of the loop.) And remember that the last character on a line may just be the newline character!
- If you find "bad" indentation, you should immediately finish the program. You should print a message about the line number that had bad indentation.

Examples

In the git repo for this homework, there are four example Python (.py) files to test your tabnanny with. Below is some example output using each of the test files. Refer to the test (.py) files for the indentation structure and why one or the other should be correctly validated by tabnanny, or why it should fail tabnanny's checks. The output shown below is an example; your output does not have to look exactly like this, but should be similar.

Test0.py:

```
$ ./tabnanny test0.py
Tab nanny has been satisfied.
Stack at the end:
    4
    0
```

Test1.py:

```
$ ./tabnanny test1.py
Line 3 col 2: unexpected indentation
  2||print("hello, world")
  3||  assert()

Stack at the end:
    0
```

Test2.py:

```
$ ./tabnanny test2.py
Line 9 col 3: unexpected indentation
  8||        ok so far
  9||   but this is a bad line

Stack at the end:
```

Test3.py:

```
$ ./tabnanny test3.py
```

```
Tab nanny has been satisfied.
Stack at the end:
    2
    1
    0
```