8.14: DEBUGGING

Contributed by Chuck Severance
Clinical Associate Professor (School of Information) at University of Michigan

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

   If you are used to writing string code like this:

   ```
   word = word.strip()
   ```

   It is tempting to write list code like this: ~~~~ {.python} t = t.sort() # WRONG! ~~~~

   Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

   Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at https://docs.python.org/3.5/library/stdtypes.html#common-sequence-operations. The methods and operators that only apply to mutable sequences are documented at https://docs.python.org/3.5/library/stdtypes.html#mutable-sequence-types.

2. Pick an idiom and stick with it.

   Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

   To add an element, you can use the `append` method or the `+` operator. But don't forget that these are right:

   ```
   t.append(x)
   t = t + [x]
   ```

   And these are wrong:

   ```
   t.append([x])          # WRONG!
   t = t.append(x)        # WRONG!
   t + [x]                # WRONG!
   t = t + x              # WRONG!
   ```

   Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

   If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

   ```
   orig = t[:]
   t.sort()
   ```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

4. Lists, `split`, and files

   When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the *guardian* pattern when it comes writing programs that read through a file and look for a "needle in the haystack".

Let's revisit our program that is looking for the day of the week on the from lines of our file:

```
From stephen.marquard@uct.ac.za  Sat  Jan 5 09:14:16 2008
```

Since we are breaking this line into words, we could dispense with the use of `startswith` and simply look at the first word of the line to determine if we are interested in the line at all. We can use `continue` to skip lines that don't have "From" as the first word as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print(words[2])
```

This looks much simpler and we don't even need to do the `rstrip` to remove the newline at the end of the file. But is it better?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

It kind of works and we see the day from the first line (Sat), but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever, and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a `print` statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output, but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable `words` right before line five. We even add a prefix "Debug:" to the line so we can keep our regular output separate from our debug output.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From' : continue
    print(words[2])
```

When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Each debug line is printing the list of words which we get when we `split` the line into words. When the program fails, the list of words is empty `[]`. If we open the file in a text editor and look at the file, at that point it looks as follows:

```
        X-DSPAM-Result: Innocent
        X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
        X-DSPAM-Confidence: 0.8475
        X-DSPAM-Probability: 0.0000

        Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

The error occurs when our program encounters a blank line! Of course there are "zero words" on a blank line. Why didn't we think of that when we were writing the code? When the code looks for the first word (`word[0]`) to check to see if it matches "From", we get an "index out of range" error.

This of course is the perfect place to add some *guardian* code to avoid checking the first word if the first word is not there. There are many ways to protect this code; we will choose to check the number of words we have before we look at the first word:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print 'Debug:', words
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

First we commented out the debug print statement instead of removing it, in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use `continue` to skip to the next line in the file.

We can think of the two `continue` statements as helping us refine the set of lines which are "interesting" to us and which we want to process some more. A line which has no words is "uninteresting" to us so we skip to the next line. A line which does not have "From" as its first word is uninteresting to us so we skip it.

The program as modified runs successfully, so perhaps it is correct. Our guardian statement does make sure that the `words[0]` will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, "What might go wrong?"

Exercise 2: Figure out which line of the above program is still not properly guarded. See if you can construct a text file which causes the program to fail and then modify the program so that the line is properly guarded and test it to make sure it handles your new text file.

Exercise 3: Rewrite the guardian code in the above example without two `if` statements. Instead, use a compound logical expression using the `and` logical operator with a single `if` statement.