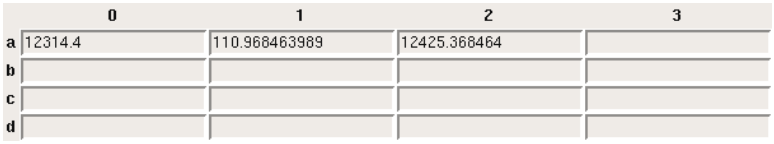
Homework #7 **Due: 12/4**

1. [100 points] In a module `spreadsheet.py`, create a new Tkinter-based class called `Spreadsheet` that is a `Frame` that contains a labelled array of cells that looks something like this:



   The prototype for a `Spreadsheet` is:

   ```
   Spreadsheet(parent, nr=4, nc=4)
   ```

   where `nr` is the number of rows and `nc` is the number of columns. The columns are numbered left to right from 0 to `nc-1`. The `nr` rows are lettered in lower case starting with "a" in on the top. (Don't worry about having more than 26 rows for now!)

   The interaction model for this spreadsheet differs from that of normal Excel^{TM}-like (actually, VisiCalc^{TM}-like) spreadsheets, and may actually be easier for a programmer (especially a Python programmer) to understand. In "normal" spreadsheets, cells have text values or numeric values. In this case, all cells contain strings which can be evaluated like any Python expression. Strings evaluate to themselves and can therefore be used as both data and labels.

   These cells allow the user to type in any Python expression, which will be maintained as a string associated with the cell. In fact, the suggested implementation is to give each cell two strings: an "expression" and a "value". The expression is what the user types into the cell. The value is a string representation of the result of evaluating the expression and is what the users sees when they are not editing a cell. When the user finishes editing the cell (usually by pressing `Return` or `Tab`), that string will be evaluated (with `eval()`), converted to a string (using `str()`), and the result will be displayed in the cell.

   Be sure to control the scope of evaluation by maintaining and passing a dictionary to `eval()`. (We'll call this the *symbol table*.)

   Initially, all cells start with a expression (and value) of `""`, an empty string, so all cells initially appear blank.

   Clicking on a cell with a mouse button brings up the cell's expression, which the user may then modify and reenter with `Return` or `Tab`.

Evaluating Cell Expressions

When `eval()` is called, the cell string may contain any Python expression[1], string, integer, or float[2]. It may also contain variables which correspond to the case-insensitive names of defined cells (e.g. `A0`, `c17`). In addition, the contents of the Python `math` module should be made available (with no "`math.`" prefix required).

One thing about `eval()`: It takes three arguments: the expression (a string in this case), a "global" symbol table (a dictionary) and a "local" symbol table (also a dictionary). `eval()` adds a `"__builtin__"` entry to the first argument before evaluating the expression, which can cause confusion and possible complications. One way to avoid this is to pass your symbol table as the third argument: the local symbol table, (e.g. "`eval(expr, {}, symbolTable)`").

Updating the Spreadsheet

A change of one cell's value can, in principle, lead to the modification of all other cell values. A really clever spreadsheet implementation keeps track of these dependencies with some kind of graph algorithm that uses an expression parser and can minimize the number of updates required, but that's beyond the scope of this project, so we're going to simplify things at the cost of some efficiency.

You might think it would work to update all cells once whenever one cell is modified, but a single update of all cells may not be enough. Updated values can propagate indirectly: If `a2` depends on `a1` and `a1` depends on `a0`, a changed `a0` will update `a2` will only work if `a1` is recalculated before `a2`. If `a2` is updated before `a1`, it won't pick up the modified `a1`.

In addition, we need to prevent cycles: `a0` cannot depend on itself or on any other cell that depends, ultimately, on `a0`.

Algorithm 1 (below) solves these problems. Note that it's in (IEEE standard, more or less) pseudocode, so you'll have to translate that to Python. (That's part of the homework.)

Dealing With Errors

Expectations for error handling are minimal:

- Nothing the user can do should crash any program using the widget.
- Do not permit cyclic expressions (see above).

[1] Within reason. My prototype doesn't work with lambda expressions, for instance! 10 points extra credit if yours does, but be sure to say so in a comment in a `README.txt` file.

[2] Don't worry about complex for now.

- An illegal expression (including one that raises a `NameError` exception) should not be allowed to be entered – when the exception is raised, revert to the former expression and value.

The instructor's prototype (compiled for 32-bit cpython) is available from the class web page.

Your module should include a self-test and document itself with `pydoc3`. (Intentionally, the prototype does not do the latter.)

Algorithm 1 Update cell p to a new expression e.

```
e' ← p's old expression
v' ← p's old value
{First, create a symbol table that makes no reference to p.}
s ← copy of the symbol table
remove p's entry from s
repeat
  clean ← true
  for all cells q except p do
    evaluate q's expression using s {Fails if q depends on p.}
    if there are errors (exceptions) and q is in s then
      remove q's entry from s
      clean ← false
    end if
  end for
until clean
{Then, try to evaluate the new value for p with the pruned symbol table.}
v ← evaluation of e using s
if there are errors (exceptions) then
  return false
end if
{Then, (tentatively) update p's expression and value.}
set p's expression to e
set p's value to v
{Finally, either update all cells to reflect the new
value of p or, if there are any errors, undo everything.}
okay ← true
repeat
  done ← true
  for all cells q except p do
    v ← evaluation of q's expression
    if there are errors (exceptions) then
      set p's expression back to e'
      set p's value back to v'
      done ← false
      okay ← false
    else if q's value ≠ v then
      set q's value to v
      done ← false{keep propagating}
    end if
  end for
until done
return okay
```