Introduction To Python

Week 7: Program Dev:
Conway's Game of Life

Dr. Jim Lupo
Asst Dir Computational Enablement
LSU Center for Computation & Technology
Overview

- Look at a helpful development tool.
  - SVN (Subversion)
  - Create a project space
- Examine the rules of Life.
- Decide on data structures.
- Decide how to handle computations.
- Decide on display of results.
- Decide on user control.
Example Files

• Three Python Scripts are provide.
• Same program, different phases of development.
• Can be used as examples for the source code revision control we'll be talking about.
Revision Control with Subversion, AKA `svn`

- Source code revision control systems has been around a long time: update, sccs, pvcs, rcs, cvs, `svn`, git, mercurial
- Maintains a history of changes made to a file (best for text-based).
- Allows recovery of any previous version.
- Provides tools for simultaneous changes by multiple developers.
- Provides tools for `forked` versions with custom changes.
SVN Availability

- **Windows**: TortoiseSVN Explorer plugin
- **Linux**: typically installed by default, or readily available via package manager.
- **MAC**: Available from Apple in the open source area of the apps store, but other sources are also available.
Create Repository: Python

• A repository is a directory structure that holds the revision database.
• It can hold more than one software project.
• Use `svnadmin create` on Linux and MAC
• Will use `TortoiseSVN` on windows.

Start with an empty directory named `SVN`. 
Step 2: TortoiseSVN "Create Repository Here" in empty directory.
Step 3: "Create Folder Structure" then "Start Repobrowser."
Step 4: Check out a copy of "trunk":

![Checkout dialog box showing the URL of repository and checkout directory settings.](image)
Step 5: Create and Add a Directory in `trunk`
Step 6: Commit and Clean Up

Delete trunk/Python to finish.
Step 7: Checkout Project Directory

Checkout Python to any location, even multiple locations!

"Working copies" are checked out here.
Typical Usage Pattern

• Create a new file in project.
• Edit it.
• Add to repository.
• while not PERFECT :
  • edit
  • test
  • commit
• If accidentally deleted, do an svn update.

Commits are CHEAP! Commit EARLY and Commit OFTEN!!
The Game Of Life

- John Conway was studying "cellular automaton" and devised an interesting set of rules (not really a 'game'):
  - Consider the squares of a grid to be cells, and that a cell may be either alive or dead.
  - Each cell has 8 neighbors.
  - The boundary cells 'wrap' around to the other edge, making the grid continuous in both dimensions.

Colored means 'alive', grayed means 'dead'.
Generational Rules

• The grid evolves from one generation to the next following these rules:

For a cell that is alive:
  • If 0, 1, 4, 5, 6, 7, or 8 live neighbors, cell dies.
  • If 2 or 3 live neighbors, cell lives

For a cell that is dead:
  • If exactly 3 live neighbors, cell is born.
  • Rules applied all at once, not incrementally.
Example Generations

Generation 1

Generation 2

Generation 3
How to Handle the Grid?

**Screen Coordinates**

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Conventional Coordinates**

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
How To Handle Generations?

Grid 0

Grid 1

Gen 0 ➔ Gen 1
Gen 2 ➔ Gen 3
Gen 4 ➔ Gen 5
Manual Example

Now use Gen 1 to produce Gen 2.
Then use Gen 2 to produce Gen 3.
Then use Gen 3 to produce Gen 4.
Repeat indefinitely.
Data Structure Choice

• What to use?
  • List of lists?

• How to initialize?
  • Values for live and dead?
  • Should allow use for other things?

• How to toggle grid use?
Use List of Lists

- Could support more complex rules later.
- Syntax: grids[][]
  - Each sublist is a full set of rows and columns
  - Final structure:

\[
\text{grids}[[\text{row1}],[\text{row2}],\ldots,[\text{rowN}]], [[\text{row1}],[\text{row2}],\ldots,\text{rowN}]]
\]

- grids[1][5][42] then means:
  - the second grid (index 1)
  - the sixth row (index 5)
  - the 43 column (index 42)
Toggle Between Grids?

• Could define variable \( g_0 \) as always representing the current grid, and \( g_1 \) always representing the future grid.

• Start with \( g_0=0 \) and \( g_1=1 \), then toggle with:
  
  \[
  \begin{align*}
  \text{tmp} &= g_0 \\
  g_0 &= (g_0+1) \mod 2 \\
  g_0 &= g_1 \\
  g_1 &= \text{tmp}
  \end{align*}
  \]
  or

  \[
  \begin{align*}
  g_0 &= (g_0+1) \mod 2 \\
  g_1 &= (g_1+1) \mod 2
  \end{align*}
  \]

• Done at the end of each generation, then at start:
  • \( \text{grids}[g_0] \) is always the current grid.
  • \( \text{grids}[g_1] \) is always the future grid.
Rule Logic

- What must be calculated?
  - Think in terms of arbitrary cell coords.
- How are boundaries wrapped?
  - What does this mean to the programmer?
  - Implement without wrapped boundaries and add that feature later?
Processing The Rules

• One generation involves:
  • For each cell in the current grid:
    • Count number of live neighbors, excluding self.
    • If cell is dead, apply one rule set.
    • If cell is alive, apply other rule set.
    • Change the cell in the future grid to match rule result.
Dealing With One Cell

- Use nested loops to check on neighbors
  - Set neighbor count to 0
    - for row above, same, below
    - for column left, same, right
  - If not looking at self, and alive, count

- This is where boundary wrapping is done:
  - If row < 0, row = max row
  - If row > max row, row = 0
  - if column < 0, column = max column
  - if column > max column, column = 0
Control Generations?

● This is really the main program loop.
● What user control to allow?
  • How to present to user?
● How does this interact with the display of results?
Code: Model 0, Mark 1, Part 1

Initialize grids, assuming a 20x20 square is desired.

Add rows of proper size.

Set all cells to 0 (dead).

Loop control variables.

```
grids = [[],[]]
g0 = 0
g1 = 1
size = 20
for i in range(size) :
    grids[g0].append(range(size))
    grids[g1].append(range(size))

for i in range(size) :
    for j in range(size) :
        grids[g0][i][j] = 0
        grids[g1][i][j] = 0

running = True
generation = 0
```
while running:
    for j in range(size):
        for i in range(size):
            c = 0
            for v in [-1, 0, 1]:
                for h in [-1, 0, 1]:
                    if v == 0 and h == 0:
                        continue
                    x = i + h
                    if x < 0:
                        x = size - 1
                    if x >= size:
                        x = 0
                    y = j + v
                    if y < 0:
                        y = size - 1
                    if y >= size:
                        y = 0
                    c += grids[g0][x][y]
            if grids[g0][j][i] == 1:
                if c in [2, 3]:
                    grids[g1][j][i] = 1
                else:
                    grids[g1][j][i] = 0
        if grids[g0][j][i] == 0:
            if c in [2, 3]:
                grids[g1][j][i] = 1
            else:
                grids[g1][j][i] = 0
        generation += 1
    if generation > 100:
        running = False
    i = g0
    g0 = g1
    g1 = i

Runs. Doesn't do much exciting!
Do Some SVN Stuff

- Add the file to the repository.
- Look at the log entry.
- Delete the file 'accidentally'.
- Use SVN to check for modifications.
- Use SVN update to recover file.
Generate Some Output

- Use a list to define the characters for dead and alive, and display new grid.

```python
symbols = [' ','*']
print "Generation", generation
for j in range( size ) :
    row = ""
    for i in range ( size ) :
        row += symbol[grids[g1][j][i]]
    print row

cmd = raw_input('Enter to Continue/q to quit: ')
if cmd.strip() == 'q' :
    running = False
```
Do the SVN Thing

- Runs but shows all blank lines - no live cells in generation 0!
- But control seems to work.
- Commit changes before moving on.
- Manually insert a glider at some coordinate (the demo graphic).
Hardwire In A Glider

- Start in second column and second row:
  \[
  \text{grids}[g0][1][2] = 1 \ # \ _* \\
  \text{grids}[g0][2][3] = 1 \ # \ ___* \\
  \text{grids}[g0][3][1] = 1 \ # \ *** + next two lines. \\
  \text{grids}[g0][3][2] = 1 \\
  \text{grids}[g0][3][3] = 1
  \]
- Get fancy later.
- Test and it should work.
- Commit!
Using Tkinter Graphics

- Take a side trip into Python graphics.
- Uses Tk - a graphical interface version of TCL. Stand alone, they are the Tool Command Language / Tool Kit - Tcl/Tk
- Both can be embedded for use in other programs - Python embeds Tk.
- It's another whole set of languages!
- Will investigate next week!