CS 1: Introduction To Computer Programming, Fall 2015

Assignment 6: The Cheese Shop

Due: *Tuesday, December 1, 02:00:00*

Coverage

This assignment covers the material up to lecture 18.

Note on Thanksgiving

According to the usual schedule, this assignment would be due on Thanksgiving day at 2 AM, which won't work for obvious reasons. We also realize that many (most?) of you will be flying out for Thanksgiving early and/or will be otherwise occupied during this time. Therefore, the actual due date is 2 AM on Tuesday (December 1). This does not mean that assignment 7 will be pushed back until the following Tuesday, instead, it will be due on Friday, December 4th at 2 AM. So we very strongly suggest that you get started early on this assignment! In recognition of the holiday, this assignment is a bit shorter than most.

What to hand in

You will be handing in two files for this assignment.

You should collect the answers to section B into a file called `lab6_b.py`.

For section D, you should hand in the file `lab6_d.py` containing your program.

Both files of your code should be handed in to csman as usual.

Part A: Installing new Python packages

There are no new Python packages that you need to install for this week's assignment.

Part B: Exercises: Code Golf

There is a game that programmers sometimes play called "code golf". The idea of code golf is to come up with a solution to a programming problem in the smallest number of characters in the source code. In addition to being fun, it also forces you to learn more concise ways to write code than you might already know.

When grading your assignments and the midterm, we often notice that many students write code that is correct but which is far from the most concise or natural way to do it. Therefore, in this section, we will be challenging you to write some extremely simple Python expressions or functions in a very small amount of code. The idea is not to remove all whitespace (spaces, tabs, or newlines) or comments (those don't count), but to write the answers using the tools that Python already gives you in a concise way.

None of these questions should take you more than a minute, not counting the time you might need to look up some things in the Python documentation!

We will use the expression "limit X characters" to mean "you can use at most X non-whitespace characters in your solution". Remember, whitespace characters (blanks, tabs, newlines) don't count towards the total character count!

1. Write a single line of Python code that will print a blank line to the terminal. Limit 5 characters.

2. You have a variable `x` which contains a number. Write a line of Python code which will add 10 to it. Limit 5 characters.

3. You have a variable `los` which contains a list of strings. You want to join them (*hint, hint*) into a single string, with each string separated from the next in the list by a single space character. Write a Python expression that will do this by calling a single method on strings. Limit 15 characters (you actually need less than this). One character will have to be a space, of course.

4. Same as the previous question, but now the strings aren't separated by spaces (or anything else).

5. Same as the previous question, but now the strings are separated by newlines (one newline character between each string and the next).

6. You have a list `lst`. Use it to create a list `lst2` which is equal to the first 5 elements of the list `lst`. Limit 12 characters. Assume that `lst` has at least 5 elements.

7. Write a boolean expression (*i.e.* an expression evaluating to `True` or `False`) that tests whether a variable containing a python string is a vowel *i.e.* one of the letters 'a', 'e', 'i', 'o', or 'u'. Assume that the variable's name is `x` and that the string it contains is one character in length. Limit 10 characters. *Hint:* use the `in` operator.

8. You have a variable `grid` that contains a list of lists of integers (also known as a two-dimensional list). In other words, `grid` is a list whose elements are also lists (of integers). Write a Python expression that will retrieve the first element of the first list in `grid` (which you can assume exists) and assign it to the variable `first`. Limit 20 characters (you actually need less than this).

9. Following up from the last question, write a Python expression that will assign the number `42` to the first element of the first list in `grid` (which you can assume exists). Limit 15 characters (you actually need less than this).

10. Assume you have an open file object called `f` which corresponds to a text file on your computer. Write two very short lines of Python code to read all the lines of the file, capitalize them, and print them out to the terminal. Make sure you don't add any extra newlines to the output! Limit 20 characters per line. Do not use the `readlines` method on files. You don't have to close the file when you're done (at least, not for this problem; in reality you would want to do that).

11. Assume you have a variable called `s` which contains a lowercase string. Write Python code which, when executed, changes `s` so that it's now uppercase and in reversed order. This can be done in one line, but we will allow you three lines, as long as each line is no more than 20 non-whitespace characters long. Do not use any `if` statements or loops; just function calls, method calls, and assignments. *Hint:* method calls can be chained.

BONUS HONOR ROLL CREDIT! If you can do it in a single line of no more than 40 non-whitespace characters, you get 2 Honor Roll points. To do this you will probably need to use Python functions/methods that we haven't covered in class, which is OK, but they must be built-in ones (*i.e.* you don't have to `import` any modules to solve this problem). Again, no loops or `if` statements are allowed. If you do it in more than 40 non-whitespace characters but less than 64, you get 1 Honor Roll point.

Part C: Pitfalls

There is no pitfalls section in this assignment.

Part D: Miniproject: Bouncing ball animations

For this week's project, you're going to write a program which represents a very simple physical simulation: balls bouncing around inside a box. This will not be a completely realistic simulation; specifically:

- the simulated world will be two-dimensional
- there will be no gravity
- balls will not interact with each other

What the simulation will do is represent circular objects (which we'll call "balls", though "circles" would be more accurate) which move around inside a two-dimensional `Tkinter` canvas. When they hit a "wall" (one of the edges of the canvas), they will bounce back in an appropriate way. It will also be possible to create new balls and destroy existing ones, as well as to speed up or slow down the entire set of bouncing balls. If you think of the balls as gas molecules, this can be viewed as changing the temperature of the gas. (Physicists and chemists describe this kind of simulation as an "ideal gas", which means that the individual gas molecules don't interact with each other.) Mostly, though, this miniproject is a good example of a graphical program which can simultaneously do something to graphical objects without user interaction as well as handle user events. In order to do this, we will have to have a different kind of event loop than we have been using so far.

Before you begin, please read this entire section carefully. This project does not require that you write a lot of code, but there are a number of interesting ideas that will take a while to explain. Don't let the length of this section scare you! Once you understand what you need to do, writing the code won't be too hard, and the resulting program will be very cool. Also note that there is a template code file which contains much of the code for the project; you just have to add a few parts (the interesting parts!). There's more on the template file below.

Background information

In order to do this miniproject you need to know a bit more about how `Tkinter` works. This section will give you the necessary information.

More on event loops

In the lectures we have seen that to start the event loop in a Python program that uses `Tkinter`, we use the `mainloop` method of the root window. So in our code there will usually be a line like:

```
root.mainloop()
```

This will start up the event loop. This is perfectly adequate for a program which is purely event-driven *i.e.* a program which does nothing except in response to user input. However, programs that simulate a virtual world (which includes many simulations and most video games) must not only listen for user events and handle them once they occur, but must also simulate the evolution through time of the virtual world even if no events occur.

This kind of program needs a more powerful way of handling events than just calling `root.mainloop()`. Specifically, the event loop will have to be broken down into two components:

- evolve the virtual world over some time period
- handle any user events (*e.g.* key presses, mouse clicks) that occurred during that time period.

Fortunately, `Tkinter` allows us to create such an event loop quite easily. To see how this works, let's look at the line:

```
root.mainloop()
```

This can also be written as:

```
while True:
    root.update()
```

The `update` method on the `root` object handles all events that have happened since the last time the `update` method was called on that object. Some of these events may be bound to callback functions using the `bind` method we've seen before, in which case the callback functions will be called during the `update`.

OK, so we can write `root.mainloop()` in a different way with an explicit `while` loop. So what? The advantage of this is that we can put other function calls inside the `while` loop. If we have a simulation that needs to evolve at regular intervals, we can do that evolving inside the `while` loop and also listen for user events at the same time. If one step of our simulation happens when we call a function called `step()`, we can write:

```
while True:
    step()          # evolve the simulation one step
    root.update()   # handle user events, if any
```

This is a much more versatile way to write graphical programs, as we will see. Typically, one "step" corresponds to a very small time interval, where the objects being simulated move only a very small amount. When a bunch of steps occur one after another, the objects will appear to move on the canvas; this is how animations are programmed.

Not only that, but it's also very easy to halt the simulation. All we need to do is define a variable called `done` (this might be a global variable), and rewrite the event loop as follows:

```
done = False
while not done:
    step()          # evolve the simulation one step
    root.update()   # handle user events, if any
```

This simulation will continue until `done` is `True`, at which point it will halt (because `not done` will be `False`, which ends the `while` loop). For this to work, one of the callback functions must change the value of `done` from `False` to `True`. When you write the event loop this way, there is no need to call `quit()` to terminate the simulation.

New canvas methods

You will need to know about the canvas `move`, `delete` and `cget` methods. The `move` and `delete` methods were described in the lectures, but we'll describe them again here.

For a given canvas object handle `h`, a call to the `move` method would look like this:

```
canvas.move(h, 10, 20)
```

What this will do is to change the location of the object on the canvas represented by the handle `h`. It will increase the `x` position by 10 pixels and it will increase the `y` position by 20 pixels. So if the center of the object was at `(100, 150)` before this method call, afterwards it will be centered at `(110, 170)`. The `move` method can also be given negative `x` and/or `y` values.

The `delete` method takes a handle as its argument and removes the corresponding object from the canvas.

The `cget` method takes a string which describes some feature of the canvas and returns its value. For instance, you can get the height of a canvas as:

```
canvas.cget('height')
```

and the width as:

```
canvas.cget('width')
```

Note that this is another way in which `Tkinter` is not particularly object-oriented; a more object-oriented way of getting this would be to have attributes called `canvas.height` or `canvas.width`, but those attributes unfortunately don't exist.

The program

1. **[120]** Write a bouncing ball simulator. This program will put some number of colored circles ("balls") on a `Tkinter` canvas, each one moving in a particular direction with a particular speed. When one of the balls hits one of the edges of the canvas, it will bounce off the edge and continue moving at the same speed but in a different direction (reflecting off the edge). Balls do not interact with one another (they can pass through each other). In addition, the program will respond to certain key press events as follows:

    1. When the user presses the `q` key, the program will exit (`q` stands for "quit").
    2. When the user presses the `x` key, all the circles are cleared from the screen. This means that every time a circle is created, the handle that is returned must be stored somewhere.
    3. When the user presses the `+` key, an additional ball with a random color and size will be created and placed on the canvas in a random location. The entire ball must fit onto the canvas; no part of the ball is allowed to run off the edge of the canvas at any point.
    4. When the user presses the `-` key, one of the balls on the canvas will be removed (it doesn't matter which one).
    5. When the user presses the `u` key, all the balls will speed up (u for "up") by a factor of 1.2.
    6. When the user presses the `d` key, all the balls will slow down (d for "down") by a factor of 1.2.

    Each ball will be an instance of a class called `BouncingBall`. Each ball object will store:

    - the canvas it was created on
    - its handle on the canvas
    - its center coordinates (in pixels)
    - its radius (in pixels)
    - its color (as a color string like in assignment 5)
    - the direction it is moving in (see below)
    - its speed (see below)

    `BouncingBall` objects must implement the following methods in addition to the constructor method:

    1. `step`: this moves the ball the distance it would travel during one step of the simulation, bouncing it off the edges if necessary
    2. `scale_speed`: this scales the speed of the ball by some factor
    3. `delete`: this just deletes the ball from the canvas

In addition, a method called `displacement`, though not technically essential, will be very convenient. See the following section for details.

Notes and hints

Template code

We are providing you with a template file containing some of the code you'll need for this problem. It's located here. You should edit it by filling in your code at the locations marked by `# TODO` comments. (The `# TODO` comments should be removed; they are just placeholders and should not be in your final submission.)

Notice that the `BouncingBall` constructor is provided for you. The `create_oval` method of canvases is used to create the circle which represents the bouncing ball. The `delete` method is also provided for you. The other methods need to be written, and the `random_ball` and `key_handler` functions need to be completed.

Although this miniproject requires a lot of explanation, there really isn't that much code to write (about 50 extra lines in our version).

Velocity, speed and direction

Conceptually, the velocity of the ball is an example of a mathematical *vector*, which means that it has a magnitude and a direction. The magnitude is called the *speed* and is represented as a floating point number which (when rounded to an integer) is the number of pixels the ball moves in the direction of motion each step of the simulation. This speed value should be positive and is stored as an attribute of the object. The direction a ball is moving is given to the constructor as the angle between the ball's direction of movement and the direction of the positive `x` axis (going counterclockwise from the positive `x` direction). This angle is expressed in degrees, and should be between 0.0 and 360.0 degrees. Inside the constructor, this angle is converted into radians, which are more convenient for calculations than degrees. The equation relating them is:

```
angle_radians = angle_degrees * pi / 180.0
```

Once we have the angle in radians, we can decompose the velocity vector into its `x` and `y` components using these equations:

```
vx = speed * cos(angle_radians)
vy = -speed * sin(angle_radians)
```

Note that Python assumes that arguments to the `sin` and `cos` functions are in radians. The equations for `vx` and `vy` are the standard formulas for converting between polar coordinates and rectangular coordinates (we saw this last assignment), with one exception. The `vy` value must be negated because the `y` coordinate on the screen grows downward, not upward as it does in standard mathematical coordinate systems.

Since the speed of the ball will be changeable, it's more convenient to store direction coordinates as follows:

```
dirx = cos(angle_radians)
diry = -sin(angle_radians)
```

and then define:

```
vx = speed * dirx
vy = speed * diry
```

as needed. These velocity coordinates represent the distance the ball moves in the `x` and `y` directions in a single step of the simulation, assuming that the ball doesn't collide with one of the edges of the canvas.

To give you some intuition for what directions mean, here are some direction angles along with their meanings:

- 0 degrees means that the ball is moving in the positive `x` direction *i.e.* to the right.
- 180 degrees means that the ball is moving to the left.
- 90 degrees means that the ball is moving in the negative `y` direction *i.e.* upwards (towards the top of the computer screen).
- 270 degrees means that the ball is moving downwards.
- 45 degrees means that the ball is moving diagonally towards the upper right at a 45 degree angle.
- 315 degrees (360 - 45) means that the ball is moving diagonally towards the lower right at a 45 degree angle.

Moving the ball

Every time the `step` method of a `BouncingBall` object is called, the ball has to be moved on the canvas. This will be done in the `step` method of the ball. The key to accomplishing this is to compute the distance in pixels the ball has to move in the `x` and `y` directions. In order to compute this, you need to know:

1. the position of the center of the ball (an (`x`, `y`) pair)
2. the radius of the ball
3. the direction the ball is moving
4. the speed of the ball
5. the dimensions of the canvas

Moving a ball is easy if it doesn't collide with the edges of the canvas. All you need to do is compute `vx` and `vy` as described above, which will give you the pixels moved in the `x` and `y` directions in one step (note that pixel coordinates should always be rounded off to the nearest integer value, as fractional pixels are meaningless). Then you just call the `move` method on the canvas to move the ball, updating the `center` attribute of the ball in the process.

The tricky part is handling collisions with the edges correctly. If the outer edge of the ball touches the edge of the canvas, that's a collision. Furthermore, a collision can occur with one edge only or with two edges simultaneously (a corner collision). The easiest way to deal with this is to treat the `x` and `y` coordinates separately. A collision in the `x` direction occurs when either:

1. the `x` position of the center of the ball before this step, plus the change in the `x` position during this step, plus the radius of the ball is greater than the width of the canvas.
2. the `x` position of the center of the ball before this step, plus the change in the `x` position during this step, minus the radius of the ball is less than zero.

A similar situation occurs in the `y` direction, except that instead of the width of the canvas, the ball must not go past the height of the canvas in the `y` direction.

If you know that a collision in a particular direction is going to happen this step, instead of moving the ball as far in that direction as you were planning to, you must instead:

1. figure out how far in that direction you would overshoot the edge if you moved it as far as you were going to
2. move the ball up to the point where it's just touching the edge
3. reverse the direction of the ball in that coordinate
4. move it the rest of the way with the new direction
5. adjust the velocity of the ball to take the collision into account

This sounds complicated, but it's actually quite simple, and an example will show you what we mean. Let's assume that the canvas has the dimensions 800x600 (800 pixels wide, 600 pixels high). There is a red ball centered at (740, 300) with a radius of 50 pixels, moving diagonally to the lower right at a speed of 20 pixels/step. The direction angle is 315 degrees (equivalent to -45 degrees). We want to compute where it will be after a single step. We can compute the `vx` and `vy` values as follows:

```
vx = speed * cos(315 degrees) = 20 * 0.7071 = 14.14 pixels
  (round to 14 pixels)
vy = -speed * sin(315 degrees) = -20 * (-0.7071) = 14.14 pixels
  (round to 14 pixels)
# N.B. Recall that Python uses radians, not degrees, as arguments to sin and cos.
```

So if we ignored collisions, the center of the ball would move to (754, 314) in this time step. Unfortunately, this would put the right edge of the ball at 804 pixels, 4 pixels over the edge. This means that a collision with the right edge of the canvas will occur in this time step, and the ball has to bounce off of the edge. What we need to do is figure out where it will be after bouncing off the edge, and what its new velocity will be. Specifically, we need to figure out how far to move the ball in the `x` and `y` directions. The good news is that there is no collision in the `y` direction. Because of this, the ball will move the full 14 pixels in that direction. In the `x` direction, the ball will first move as far as it can (10 pixels until the ball is just touching the edge). This leaves 4 pixels, so the ball will reverse direction with respect to the `x` coordinate and will move the remaining 4 pixels in the opposite direction, giving a total movement in the `x` direction of 10 - 4 = 6 pixels. So the center of the ball will move to (740 + 6, 300 + 14) = (746, 314), and the ball will now be moving in a different direction (with the `x` coordinate of the direction reversed but the `y` coordinate the same, which leaves the ball heading towards the lower left corner).

If we use `dirx` and `diry` to store the coordinates of the direction as described above, the effect of a collision on the `x` direction is to negate the value of `dirx`, and the effect of a collision on the `y` direction is to negate the value of `diry`. Don't forget to do this!

Here's a tip: the displacement of the ball can be computed independently for the `x` and `y` dimensions, including the effects of collisions. In the template code, this is done in the `displacement` method, which takes the center coordinate (`x` or `y`, but not both), the change in that direction (assuming no collisions), and the limit in that direction (the width of the canvas for the `x` direction and the height for the `y` direction), and computes and returns the actual displacement in that direction, taking collisions into account. The radius is also needed in this method but it can be accessed through the `self` argument, since it's an attribute of the object. The `displacement` method can be called from the `step` method to compute the correct displacement in either dimension, and if it's different from the expected displacement, a collision must have occurred, which means that the direction of movement must be adjusted accordingly.

An example of how `displacement` could be used in the previous example:

```
# center of ball: (740, 300)
# velocity: 20 pixels/second to the lower right
# vx = 14.14 pixels/second, round to 14
# vy = 14.14 pixels/second, round to 14
# dimensions of canvas: 800x600

# In the step() method, we'd compute:
dx = self.displacement(740, 14, 800)
dy = self.displacement(300, 14, 600)

# Note: numbers given (740, 300, 800, 600, 14) would in the real code
# be replaced by variable names or object attribute names.

# Results:
# dx = 6  (bounces off wall, different from vy)
# dy = 14 (no bounce, same as vy)
```

Whew! This took a long time to explain, but we hope that you find the process of writing the program interesting, and the results entertaining.