CS 1: Introduction To Computer Programming, Fall 2015

Assignment 4: Life, the Universe, and Everything

Due: *Thursday, November 5, 02:00:00*

Coverage

This assignment covers the material up to lecture 13.

Part What to hand in

You will be handing in four files for this assignment.

You should collect the answers to section B into a file called `lab4_b.py`.

For section D, you should hand in the files `lab4_d1.py`, `lab4_d2.py` and `lab4_d3.py` containing your programs.

There is a test script for section B only. It's called `lab4_b_tests.py`. You should download it and run it to make sure your code from section B works properly.

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

Part A: Installing new Python packages

For one section of this lab, you will need to have the `Tkinter` package installed in Python. Naturally, it is already installed on the CMS cluster computers, but many of you will want to install it on your own computers. To determine if your version of Python already has this package installed, start up WingIDE and in the Python shell type:

```
>>> import Tkinter
```

If no error is reported, you are ready to go. If not, what you do depends on which operating system you are using.

Let's look at computers running Windows. The normal Python installation, as we described it in assignment 1, should have installed `Tkinter` already. If not, you will need to redo the Python installation process, following the instructions in assignment 1; this time, pay particular attention to the line that says to install all the optional packages under "Customize Python 2.7.10". This simply requires that you select the option "Tcl/Tk" and then choose "Entire feature will be installed on local hard drive". This will cause the `Tkinter` package to be installed, and after that, you should be able to use it without any problems.

If your computer runs Mac OS X, and you installed Python using the instructions in lab 1, Tkinter is probably already installed. If not, you should either use the CMS cluster computers or ask a TA to help you set up Tkinter.

Linux users should consult their distribution's package manager documentation for information on how to set up `Tkinter` on their system. Most distributions have a `python-tk` package or something similar which will install `Tkinter`.

Windows users will find some differences in using `Tkinter` compared to Linux and Mac users. The primary difference is that when using `Tkinter` on Windows (but not on Linux or Mac) no windows will show up until the `mainloop` command (described below) is entered. This shouldn't cause any serious problems.

Warning! On some computers, using `WingIDE` with `Tkinter` can result in problems if you try to execute the program from within `WingIDE` by hitting the Run button. What may happen is that the program will hang or will exit with a complicated error message. If this happens, simply use `WingIDE` as an editor only and run the program from the terminal window. So if the module you are working on is called `MyModule.py`, run it as:

```
% python MyModule.py
```

(where the `%` is the terminal prompt). If you have problems with this, ask your TA to help you. The `WingIDE` documentation claims that it works perfectly well with `Tkinter`, but we have not found this to be the case.

Part B: Exercises

In this section we will write a few simple functions which we will use in part D as part of the graphics miniproject in that section. We will also write a few functions that involve dictionaries. Every function should have a docstring explaining what it does, what its arguments (if any) mean, and what it returns. None of these functions need to be more than about a dozen lines long (not counting the docstring), so if you find that you need much more than this, you have missed something important and should talk to a TA.

1. **[20]** Write a function called `random_size` that takes two arguments (both non-negative even integers, where the first argument must be smaller than the second), and returns a random even integer which is >= the lower number and <= the upper number. Use the `randint` function from the `random` module to generate the random numbers. Use Python's `help()` function to tell you how `randint` works.

   Use `assert` statements to check the inputs to and outputs from the function; they should check that:

   1. the two input numbers are non-negative
   2. the two input numbers are both even (*hint:* use the `%` operator)
   3. the first input argument is smaller than the second argument
   4. the output number is even

   The technical term for what these assertions are checking are the *preconditions* and the *postconditions* of the function *i.e.* the conditions that must be true before the function starts computing (preconditions) and the conditions that must be true once the function is done computing (postconditions). Checking preconditions and postconditions with assertions is a common strategy to make your functions more likely to be correct.

   Note: You do not need to write a loop to solve this problem.

   Examples:

   ```
   >>> random_size(0, 10)
   8
   >>> random_size(0, 10)
   10
   >>> random_size(0, 10)
   0
   ```

```
>>> random_size(20, 102)
34
```

2. **[10]** Write a function called `random_position`. This function will take as its arguments two integers called `max_x` and `max_y`, both of which are `>= 0` (use `assert` to check this). It will return a random `(x, y)` pair (tuple), with both `x >= 0` and `y >= 0` and with `x <= max_x` and `y <= max_y`. The function is named `random_position` because the `(x, y)` pair returned from the function will represent non-negative integer pixel coordinates on a computer monitor *i.e.* a position on the monitor. Again, use the `randint` function to generate the random numbers.

   Examples:

   ```
   >>> random_position(800, 600)
   (125, 456)
   >>> random_position(800, 600)
   (234, 101)
   >>> random_position(50, 75)
   (45, 12)
   ```

3. **[20]** Write a function called `random_color` which will generate random color values in the format recognized by the `Tkinter` graphics package. These colors are represented by strings of the form:

   ```
   #RRGGBB
   ```

   where '#' is the literal character '`#`', and 'RRGGBB' are six hexadecimal digits, which are either the digits `0` to `9` or the letters `a` to `f`. The 'RR' digits represent the degree of red color in the color, the 'GG' digits represent the degree of green color, and the 'BB' digits represent the degree of blue color. The minimum value for each color is `0` and the maximum is `ff`, which is hexadecimal for 255.

   Examples:

   ```
   >>> random_color()
   '#0144a9'
   >>> random_color()
   '#ff008a'
   >>> random_color()
   '#a0a04f'
   ```

   Note that black would be the color `'#000000'`, white would be `'#ffffff'`, pure red would be `'#ff0000'`, pure green would be `'#00ff00'`, and pure blue would be `'#0000ff'`. This is not important in the function; just choose six random hexadecimal digits, put them into a string with the correct format, and return it.

   You do not need to use `assert` in this function.

   You may find the `choice` function in the `random` module to be quite useful for this problem.

4. **[20]** The keys in a dictionary are guaranteed to be unique, but the values are not. Write a function called `count_values` that takes a single dictionary as an argument and returns a count of the number of distinct values it contains. *Hint:* Remember the `values` method of dictionaries.

   Examples:

   ```
   >>> d = { 'red' :  1, 'green' : 1, 'blue' : 2 }
   >>> count_values(d)
   2
   >>> count_values({ 'a' : 97, 'b' : 98, 'c' : 99 })
   3
   >>> count_values({ 'foo' : 42, 'bar' : 42, 'baz' : 42 })
   1
   ```

5. **[25]** Write a function called `remove_value` that takes a dictionary and an arbitrary item which could be a value in the dictionary. It removes (using the `del` operator) all key/value pairs from the dictionary which have that value. If the value is not in the dictionary it does nothing. It returns nothing. You can assume that `==` will work for all values stored in any dictionary passed as an argument to `remove_value`.

   NOTE: This problem is harder than it appears! You cannot change the size of a dictionary while you are iterating through it using a Python `for` loop. *Hint:* Make a list of all the keys that need to be removed, and then remove them later.

   Examples:

```
>>> d = { 'foo' : 1, 'bar' : 2, 'baz' : 1 }
>>> remove_value(d, 1)
>>> d
{'bar' : 2}
>>> remove_value(d, 2)
>>> d
{}
```

6. **[20]** Write a function called `split_dict` that takes as its argument a dictionary which uses strings as keys and returns a tuple of two dictionaries whose key/value pairs are from the original dictionary: those whose keys start with the letters `a-m` (lower- or uppercase) and those whose keys start with the letters `n-z` (lower- or uppercase). Keys starting with other letters are not included in either dictionary. The original dictionary is not altered. You may assume that the string keys have at least one letter. *Hint:* strings can be compared using the relational operators (`<`, `<=`, `>` and `>=`).

   Examples:

```
>>> split_dict({ 'Jim' : 1, 'Tom' : 2, 'Sam' : 3, 'Bob' : 4 })
({'Bob': 4, 'Jim': 1}, {'Sam': 3, 'Tom': 2})
>>> dict = { 'rat' : 'animal', 'cup' : 'utensil', 'green' : 'color' }
>>> d1, d2 = split_dict(dict)
>>> d1
{'green': 'color', 'cup': 'utensil'}
>>> d2
{'rat' : 'animal'}
```

7. **[20]** Write a function called `count_duplicates` that takes a dictionary as an argument and returns the number of values that appear two or more times. *Hint:* The `count` method on lists may be useful to you here.

   Examples:

```
>>> count_duplicates({ 'foo' : 1, 'bar' : 2, 'baz' : 3, \
                       'xxx' : 2, 'yyy' : 1 })
2
>>> count_duplicates({ 'foo' : 'yes', 'bar' : 'no', 'baz' : 'maybe' })
0
```

Part C: Pitfalls

There is no part C this week.

Part D: Miniproject: Drawing simple pictures

Each of the following exercises will require you to write a separate Python module (program) which, when run, will draw some objects on a canvas object using `Tkinter`. All dimensions are in units of pixels. In each module, a root window called `root` should be created as well as a canvas which goes inside the root window (don't forget to use the `pack` method of the canvas object to position the canvas inside the root window!), and both it and the canvas it contains should be 800 pixels wide and 800 pixels high. The last line of the module should be this:

```
root.mainloop()
```

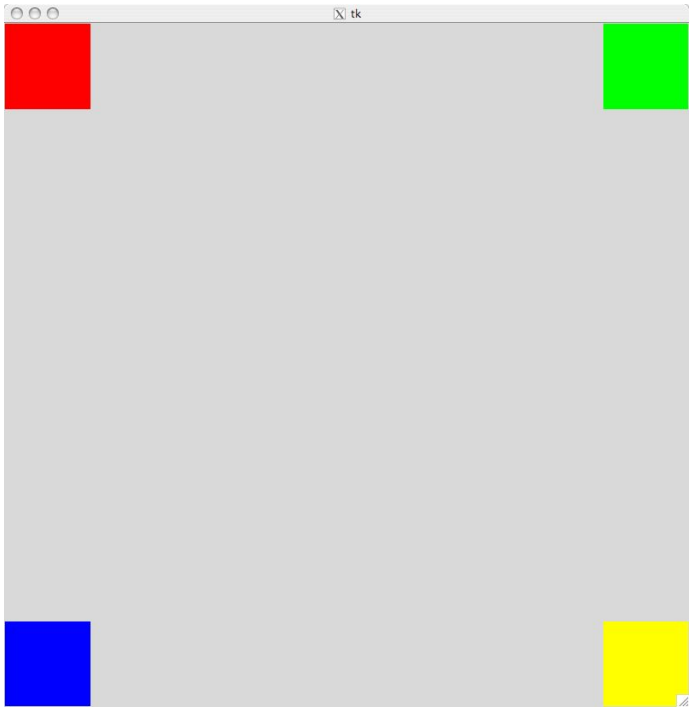
This will start the module as a separate program and display the image. For the first two programs, manually close the window and restart the Python shell to exit the program. The last program will use a different way of exiting the program.

NOTE: As mentioned above, WingIDE does not always work well with Python code that uses `Tkinter` (they interfere with one another to some extent). Therefore, when running your code, we recommend that you run it from a terminal instead of hitting the "Run" button in WingIDE. If you don't know how to do this, ask your TA. You may still use WingIDE to edit your code, of course. If you run your code from inside WingIDE, you will probably encounter bugs that will disappear when running it from the terminal. The reasons for this are beyond the scope of this class.

1. **[20]** Write a Python module (program) called `lab4_d1.py`. This module, when run, will draw four squares of size 100x100 on an 800x800 canvas using the `create_rectangle` method of the canvas object. The square in the upper-left corner should be red, the square in the upper-right corner should be green, the square in the lower-left corner should be blue, and the square in the lower-right corner should be yellow. Note that `Tkinter` knows the meaning of the strings `'red'`, `'green'`, `'yellow'`, and `'blue'` when used as color names.

   Your code shouldn't include any function definitions, just direct calls to `Tkinter` functions and methods.

   The resulting picture should look like this:

2. **[30]** Write a Python module (program) called `lab4_d2.py`. This module, when run, will draw the exact same picture as in the previous problem, but it will do so in a different way. In this module, you will write a function called `draw_square` which will take four arguments:

   1. the canvas on which the square will be drawn

2. the color of the square (which will be both the fill color and the outline color)
3. the width and height of the square (in pixels); these are obviously the same since it's a square
4. the position of the center of the square, represented as a tuple of two integers representing the horizontal and vertical position of the center in pixels

This function will both draw the square (using the `create_rectangle` method of the canvas object) and also return the handle of the square that was drawn on the canvas (note that the `create_rectangle` method itself returns the handle of the rectangle that was drawn). (Note that we won't be using these handles in this assignment, but we will in later assignments.)

The only tricky part is knowing where the center of the square should be. For a 100x100 square located in the upper-left-hand corner, the center would be at (50, 50) (height = 50, width = 50).

Make sure that your function has a docstring explaining what it does and what all the arguments to the function mean.

Once this function is written, your module will call it four times to draw the four squares in the picture. To be specific: the only place you should call the `create_rectangle` method of a canvas object is inside the `draw_square` function.

In this module, you will have some code which is part of a function (the `draw_square` function), and some code which is at the top level of the module (not in any function). To keep the module well-organized, put all of the code that isn't in the function (except for the `import` statements, which should go at the beginning of the file) into a block using the `if __name__ == '__main__': ...` idiom described in class. In other words, the module should be structured like this:

```
from Tkinter import *

def draw_square(<argument list>):
    <docstring for draw_square>
    <code for draw_square>

if __name__ == '__main__':
    <all other code>
```

3. **[40]** Write a Python module (program) called `lab4_d3.py`. This module, when run, will draw 50 colored squares of random colors, sizes, and positions whenever it's run. The module will, of course, need to import the `random` module. Use the `draw_square` function from the previous problem to draw the squares. Use a loop to generate the squares — do not have 50 lines with calls to `draw_square`, or you will get a zero grade on this section!

   Generate random colors using the `random_color` function you wrote in part B. Generate random sizes using the `random_size` function you wrote in part B. Finally, (you guessed it) generate random positions using the `random_position` function you wrote in part B. The range of sizes is from 20 to 150 pixels, and the positions can be anywhere from 0 to 800 pixels in both the horizontal and vertical directions (so they can range from (0, 0) to (800, 800)).

   There's one more thing to do before your program is done. By now, you're probably getting tired of quitting the program by manually closing the window. Instead, create a key binding so that hitting the `"q"` key anywhere inside the window will cause the program to exit. Use the `bind` method of the root object, and bind the key to a function which causes the program to exit (*e.g.* to the built-in `quit` function).

   Your file should be structured the same way it was in the previous problem:

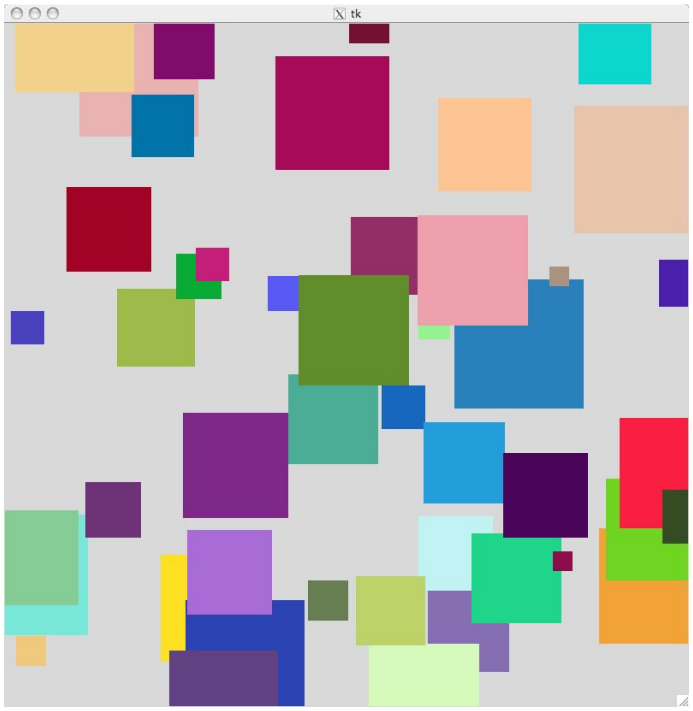
```
from Tkinter import *
import random
```

```
<code for all functions>

if __name__ == '__main__':
    <all other code>
```

You should copy the functions you're using from part B into this module.

Here is a sample picture we obtained when we ran our program:



Your images will be different from this, and will in fact be different every time you run your program.
