Lab 2

CS161

For each of the problems below be sure to design your solution before you write any Python code. Follow the software development stages we discussed in class:

1. Analyze the problem
2. Determine the specifications for your solution
3. Create a design for your solution in pseudocode
4. Implement your design in Python
5. Test/debug your solution

We'll forego the maintenance of the programs for now. Each Python module you turn in must begin with a block, called a *docstring*, that looks like the example below (see PEP 257 for more details). The opening triple quotes must be on line one of your file.

```
"""
Brief, prescriptive, one sentence description of what the program does.

A more detailed explanation of what the program does. Use complete
sentences and proper grammar. Treat this as if it is documentation
someone else will be reading (because it is). Use this paragraph to
also discuss your analysis of the problem; any assumptions you've made
regarding the program, the user, or the system on which it will run.

Firstname Lastname
Firstname Lastname (of partner, if applicable)

Design (pseudo code): write this before writing your code.

Testing: list the inputs/data you used when testing your program.
"""
<two blank lines>
<your code>
```

In your module docstring show the work you did following the software development stages above. Make notes about the problem as you analyze it; write specifications for what your program must do to find a solution; write your program out in pseudocode *before* you start to write Python; and design a set of test data that you can use to prove your program is producing correct results. Only after you've done all of this should you try to translate your pseudocode to Python.

Every function or method you write must also include its own docstring. That header will also use triple quotes and begin with a prescriptive, one sentence description of what the function or method does. If the function or method takes arguments, they should be described as well as any non-obvious return value(s). See examples below or PEP 257 for more details).

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...


def complex(real=0.0, imag=0.0):
    """
    Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Exercise 1: Conversions

Part 1

We are going to do some conversions, from integer to binary and then from binary back to integer. It will give us a chance to play with `if-elif-else` and `while` statements, as well as a little string slicing.

Prompt the user for an integer, convert the integer to a binary number string (there is no type for actual binary numbers so we just represent it as a string). We then take the string and turn it back into a regular integer. Things to remember:

1. If the integer is 0, then we are done since conversion back and forth of 0 is still 0. The program simply prints a note saying it is 0 and quits.
2. If the integer is negative, then we probably don't know how to do it, so the program prints a message saying it is negative and quits.
3. Otherwise, we do the conversion of the integer to a binary string (a string of 1's and 0's) and then convert that same string back to an integer to make sure we did it right.

Hints

How do we get a binary string from an integer? The easiest method uses integer division by 2 on successive quotients and then collects the remainders. It is best illustrated by an example. Consider the decimal number 156.

- 156 divided by 2 gives the quotient 78 and remainder 0.
- 78 divided by 2 gives the quotient 39 and remainder 0.
- 39 divided by 2 gives the quotient 19 and remainder 1.
- 19 divided by 2 gives the quotient 9 and remainder 1.
- 9 divided by 2 gives the quotient 4 and remainder 1.
- 4 divided by 2 gives the quotient 2 and remainder 0.
- 2 divided by 2 gives the quotient 1 and remainder 0.

- 1 divided by 2 gives the quotient 0 and remainder 1.

Stop at reaching a quotient of 0. The binary equivalent is given by concatenating the remainders, in reverse (so the last remainder is the most significant bit and the first is the least). In this example: `10011100'

How do we get an integer from a binary string? First, we know it is a string, so the elements are '1' and '0'. For every 1 in this string, we add a power of two to the overall decimal value. The power of 2 that we add depends on the position of the 1 in the binary string. A 1 in the far right (last) position of the string adds 2**0, in the next to last position adds 2**1, in the next to the next to the last position adds 2**2, and so on. If the bit is a '1', then we add that power of 2 to the overall sum; if it is 0 we do nothing.

For example, starting the last (right most) position of '10011100'

- last bit is '0', so it contributes nothing to the sum.
- next bit is '0', so it contributes nothing to the sum.
- next bit is '1', so it contributes 2**2 to the sum.
- next bit is '1', so it contributes 2**3 to the sum.
- next bit '1', so it contributes 2**4 to the sum.
- next bit is '0', so it contributes nothing to the sum.
- next bit is '0', so it contributes nothing to the sum.
- next bit is '1', so it contributes 2**7 to the sum.

The decimal equivalent is therefore 2**2 + 2**3 + 2**4 + 2**7, which equals 156.

Write and save your code in a file named `int_to_bin.py` for full credit.

Part 2

Write another program that takes an integer and converts it to hexadecimal and back again.

Write and save your code in a file named `int_to_hex.py` for full credit.

Exercise 2: More Turtle Graphics

You will use Turtle graphics to draw a picture containing multiple shapes of multiple colors and arranged to be visually pleasing. Although you are free to decide the precise shape(s) and layout for your picture, some aspect of the picture must depend on a numeric value input by the user. For example, the input might determine the size of the shapes, the number of shapes, or their spacing.

Your program must:

1. Output a brief descriptive message of what the program does.
2. Repeatedly prompt for the input until the user supplies values of the correct form (discard incorrect inputs). Your prompt should say what form of input is needed.
3. Draw a picture containing multiple shapes of multiple colors, where the input value(s) is (are) used to determine some aspect of the shapes and/or their layout.

In programming your solution, you must:

1. Use at least two repetition (while or for) statements.

2. Use at least one selection (if) statement.

You will find some example output produced by two different programs that meet these requirements at the end of this write-up. You may be creative and create your own program, or you may choose to mimic one of these two examples. The second example shows error checking being tested.

Notes and Hints

1. Do error checking on input last, i.e. after the rest of the program is working.
2. There is a method to check if a string is a number: isdigit(). It returns True, if the string is made up of only digits. You can use it like this:

```
num_str = input("Enter a number: ")
    if num_str.isdigit():
        num_int = int(num_str)
```

Creating Colors:

There are many ways to create a color but a common one used in computer graphics is the process of additive color (see http://en.wikipedia.org/wiki/Additive_color). Combining different amounts of red, green, and blue can create most (but not all) colors. In turtle, you can specify a color by giving three floating-point values, each in the range from 0.0 to 1.0, indicating the amount (fraction or percent) of each color. For instance, (1.0, 0.0, 0.0) is red, (0.0, 1.0, 0.0) is green, and (0.5, 0.5, 0.0) is brown. You can find the codes for many colors on a color chart (e.g, "% code" column on http://www.december.com/html/spec/colorcodes.html).

A convenient way to generate different colors is to repeatedly call the random function in the random module to obtain values for the color amounts. First, import the random module. Then, each call to `random.random( )` returns a pseudo random (floating-point) number in the range 0.0 to 1.0.

Using turtle graphics:

To use turtle graphics in Python you must first import the turtle module. You can then use the help function at the Python prompt to find out what methods this module includes and what they do. Just type `import turtle` in the Python Shell window, hit enter, and then type `help(turtle)` and scroll up through the list and information. For more details Google "Python 3 turtle."

Keeping the window up

If the drawing window has a tendency to disappear too quickly, you can "hold" the window by using the sleep function in the time module, as follows:

```
import time
time.sleep(seconds)
```

The program will wait the number of seconds indicated before it ends.

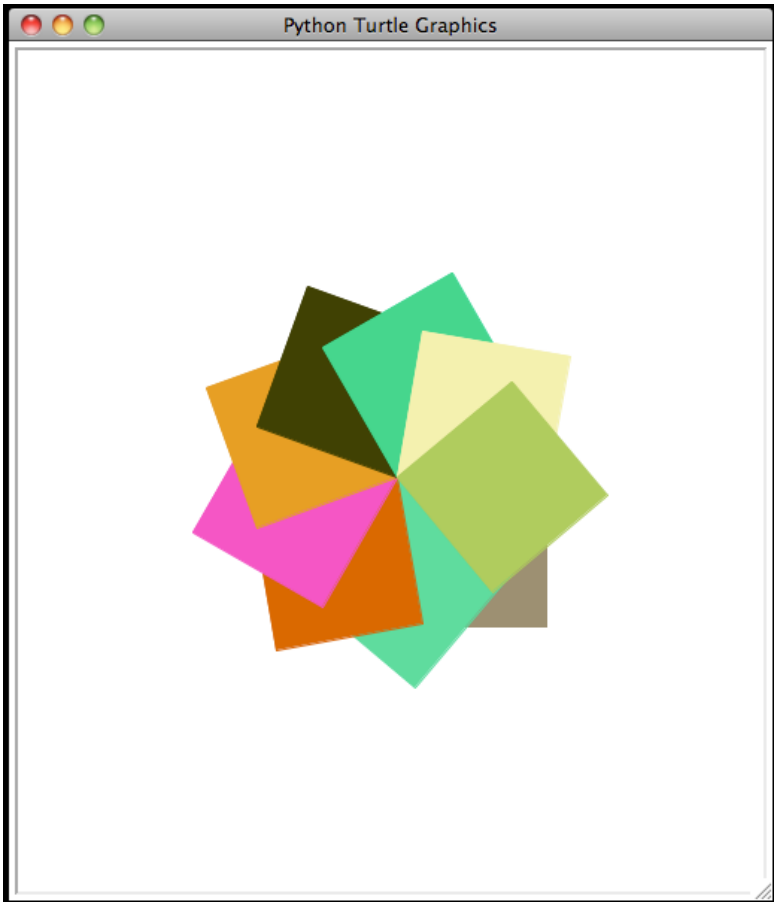
Examples

To illustrate, below are the results of executing two programs, both of which meet the requirements.

The first program draws squares of a fixed size; they all start at the origin but are arranged in a circular manner by manipulating the orientation of the turtle. An example interaction:

```
Python Shell
>>> ================================ RESTART ========
>>>
This program draws squares of many colors.

Enter the number of squares to draw: nine
The number must be an integer and at least 1.
Please try again.

Enter the number of squares to draw: 9
>>>
```

An example of the picture it draws:

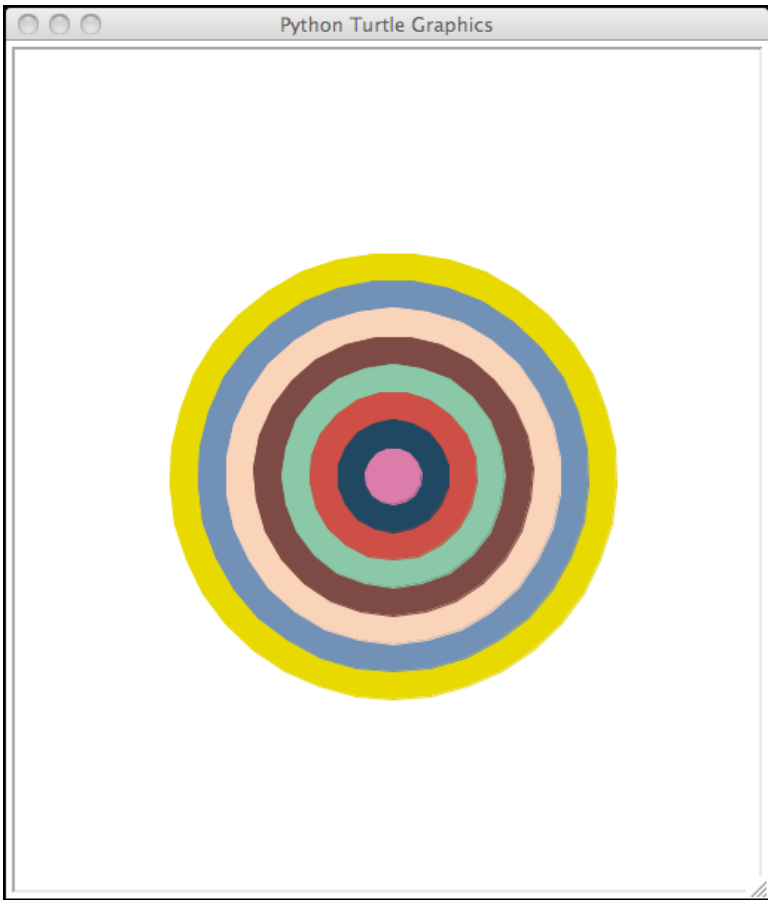


The second program draws concentric circles of many colors. The user specifies the number of circles and the radius of the largest circle.

An example interaction:

```
Python Shell
>>> ================================ RESTART ================================
>>>
This program draws concentric circles of many different colors

Enter the number of circles to draw: 8.0
The number must be an integer and at least 1.
Please try again.

Enter the number of circles to draw: 8

Enter the radius (>=50, <=200) of the largest circle: 25
The radius must be an integer between 50 and 300.
Please try again.

Enter the radius (>=50, <=200) of the largest circle: 150
>>>
```

An example of the picture it draws:



Write and save your code in a file named `turtle_3.py` for full credit.

Submitting Your Lab

Put all the Python files you wrote for this lab in a folder named `lab-2`, then copy it and all its contents into your shared folder.