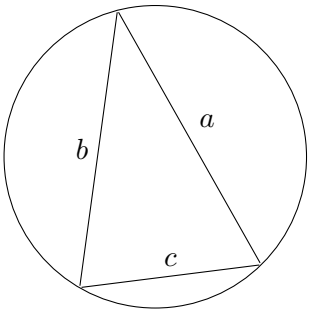
Programming for Science
Workshop 4

This workshop gives you some more practice with functions.

1. Recall the circumcircle problem from the first workshop. The diameter d of a circle that passes through the vertices of a triangle whose edges have lengths a, b and c is given by:



$$d = \frac{abc}{2\sqrt{s(s-a)(s-b)(s-c)}}$$

where s is called the semiperimeter of the triangle and is defined as

$$s = (a + b + c)/2$$

Either write a new function or modify your old program to define a function `circumcircle(a, b, c)` that calculates the radius of circumcircle of a circle passing through the vertices of triangle that has sides of length a, b and c.

Note that if the sum of the lengths of the two smaller edges is less than the length of the longest edge, then there is no triangle with edges of the given length. Make your function test for this error condition and, when it occurs print an informative message and return `None`.

Make sure your function has a useful docstring.

Write a second function (in the same file) that tests your first function. This function, `test_circumcircle` it should invoke `circumcircle` in one or two cases where you know the answer and should check that `circumcirlce` returns the correct answer. It should also check the case where the given sides don't make a triangle.

2. Write a function to print a string right justified in a line of a given width. The signature of the function should be `right_justify(s, width=70)` where `s` is the string to be justified and `width` is the width of the line to justify it in. So for example, `right_justify("Hello", 50)` would print (without the column numbers):

```
         1         2         3         4         5         6
12345678901234567890123456789012345678901234567890123456789012345678890
                                                 Hello
```

(Remember that `'X'*3` yields a string of 3 `X`'s: `'XXX'`.)

Use your function to write a new function to right justify a list of strings, such as:

```
rhyme = [
   "The Owl and the Pussycat went to sea",
   "In a beautiful pea-green boat,",
   "They took some honey, and plenty of money,",
```

```
    "Wrapped up in a five pound note."
]
```

Use your new function to right justify the "The Owl and the Pussycat" poem by Edward Lear, which you can find in the file `owl_and_pussycat.txt` on the ELE page.

3. In this exercise we will use the TurtleWorld programs in the `swampy` suite from the *Python for Software Design* book, which allows you to draw "turtle graphics". The idea is that you have a "turtle" which walks around a window, leaving a trail of paint where it has walked.

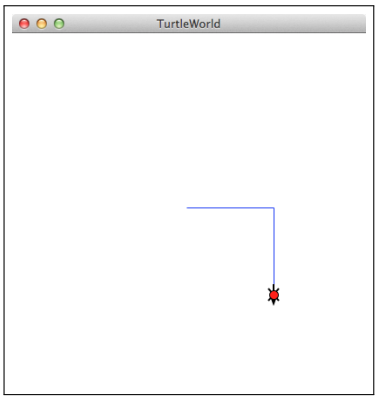
Before you can use these routines, you'll need to download them:

- Download the zipped `exswampy.zip` file from the ELE. (This is a slightly modified version of the `swampy-2.0` file from `http://greenteapress.com/thinkpython/swampy`.)
- Unzip the file, by double clicking on it. This should create a new folder/directory called `exswampy` with various Python files in it. No need to do anything to them, but you will need to change directory to the `exswampy` (using the `cd` command in the command window) before you can run any programs using `exswampy`.

Edit the following example program, perhaps named `swampytest.py` in your `exswampy` directory:

```
from TurtleWorld import *    # Import the turtle functions

# Set up the turtle world
world = TurtleWorld()

bob = Turtle()               # Your turtle
print bob


# Now some drawing
fd(bob, 100)                 # Forward 100
rt(bob)                      # Right turn
fd(bob, 100)                 # Forward another 100

wait_for_user()              # Display the result
```

Of course, you don't need to type the comments. Running the program (`python swampytest.py`), should produce a picture like the one on the right. The program ends when you close the window.



The initial few lines of the program set up the turtle world and get a turtle, here assigned to a variable, `bob`. Then the `fd(bob, 100)` and `rt(bob)` are functions that instruct the turtle (the first argument) to either move forward 100 units or to turn right. You could substitute other drawing command here. Finally the `wait_for_user()` function displays the graphics and waits until you close the graphics window.

The drawing commands (with their default arguments) that are available are:

`fd(t, dist=1)`	Move turtle `t` forwards `dist`.
`bk(t, dist=1)`	Move turtle `t` backwards `dist`.
`rt(t, angle=90)`	Turn turtle `t` right by `angle` (degrees).
`lt(t, angle=90)`	Turn turtle `t` left by `angle` (degrees).
`pu(t)`	Lift the pen up, so that the turtle can move, but doesn't draw.
`pd(t)`	Put the pen down, so that the turtle draws as it moves.

Write and test a function `square(t, length)` that uses turtle `t` to draw a square with sides of the given `length`. A neat way to do this would be to use a `for` loop.

Generalise your function to a new function `polygon(t, n, length)` that uses turtle `t` to draw an `n`-sided polygon with sides with the given `length`. Make sure that your function has a useful docstring and give it default arguments that will draw a square with sides length 100.

4. You can change the colour and width of the line that the turtle draws with:

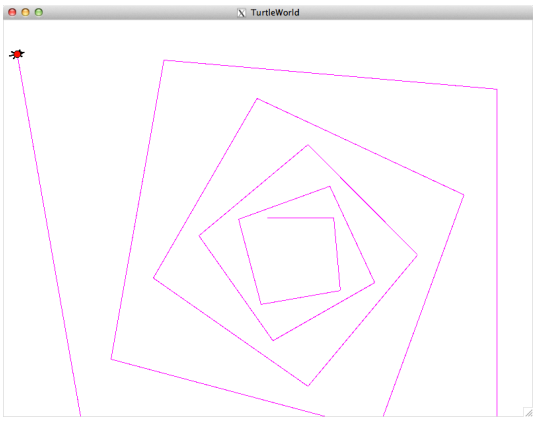
`set_pen_colour(t, color)`	Set the line colour to the given colour named in the string `color`. Legal colours are: 'red', 'orange', 'yellow', 'green', 'blue', 'violet', 'white', 'magenta'.
`set_width(t, width)`	Set the line width to the given width (a positive integer).

Modify your polygon function so that it takes arguments specifying the colour and line width. Make sure they have defaults and test your function.

Modify your function so that it returns the length of the perimeter of the polygon.

5. You can draw a spiral with the turtle with the following algorithm:

- repeat:
- draw forward a length $D = 100$
- turn right by an angle θ
- increase D by a factor α



The picture on the right was generated with an initial $D = 100$ and $\theta = 85°$ and $\alpha = 1.1$.

Write a function `spiral` that has arguments, `t`, the turtle to draw with, D, θ and α and N, the number of iterations. Experiment with different values of the arguments. What values of these parameters produce smooth looking spirals?

Stopping after a given number of steps is not so satisfactory because it's hard to judge or calculate how far from the start points the turtle will be after `N` steps. However, it's possible

to find out where the turtle is by calling the functions `where_x(t)` and `where_y(t)` which return the coordinates of the turtle `t`. Modify your function or write another to draw a spiral out to a radius `R` from the centre of the spiral. `R` should be an argument to your function.

If tuples have been discussed in lectures, you may find it more convenient to use the function `where(turtle)` which returns a tuple `(x, y)` giving the turtle's current coordinates. For example: `x, y = where(fred)` gives the coordinates of the turtle `fred`.