3: Pretty pictures

Gareth McCaughan

Revision 239, July 23, 2010

Credits

Introduction

This sheet will teach you a little about doing graphics (pictures or drawings) in Python. It will also explain a very important idea in programming: “functions” or “procedures”.

It’s quite a long sheet. Don’t worry if it takes you a while to get through it all.

What you need to know

- How to edit and run a Python program: see Sheet R (*Running Python*)
- What “coordinates” are (see your maths teacher)

Points, lines and colours

In Sheet 1, you saw a rather boring example of graphics in Python. Let’s go over it again, a little more carefully. It began like this:

```
from livewires import *
begin_graphics()
```

and ended with the line `end_graphics()`.

The first line there should be familiar by now. The second tells the computer to make a new window in which you can put lines and circles and things. The last line, `end_graphics()`, waits for a moment then closes the graphics window and resets some things so that you can say `begin_graphics()` again to start over cleanly.

All the interesting stuff happens in between. (And, in fact, it’s not all that interesting.)

```
set_colour(Colour.red)
```

This means: until further notice, draw everything in red. There’s actually a very wide range of colours available, but only a few have names. All the names begin `Colour.` for reasons I shan’t go into here.

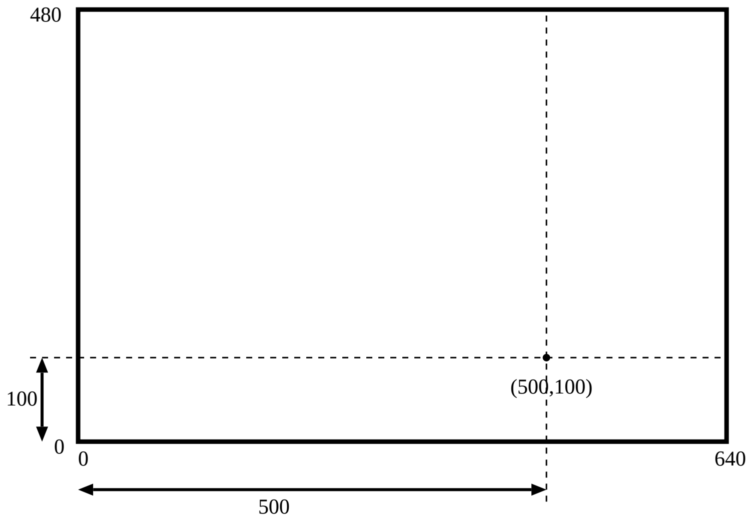
```
        move(100,100)
        draw(200,100)
```

To understand these lines, you need to know how Python (actually, this isn't Python itself so much as the `livewires` module, but never mind) thinks about graphics.

Pixels

The graphics window – like everything on the computer screen – is made up of *pixels*: little squares or rectangles of colour. If you look closely at your computer screen, you'll see what I mean. (Why the funny name "pixels"? It's short for "picture element", I think.)

You refer to places in the graphics window by their *coordinates*: first, how many pixels the place is to the right of the left-hand edge of the window (the x-coordinate), and then how many pixels it is above the bottom of the window (the y-coordinate). Here's a picture.



The "current point"

(When you read Sheet 1, I hope you actually tried typing in those commands we're talking about now. If you didn't, go back to Sheet 1 and type them in now.)

The commands beginning `move` and `draw`, between them, draw a line from the pixel called (100,100) to the pixel called (200,100). (These have the same y-coordinate (second part), so they are at the same height. The second pixel has a larger x-coordinate (first part), so it's further to the right.)

You might (or might not) find it helpful to imagine a little man, or a robot, walking around the graphics window. When you say `move(100,100)` the robot walks to (100,100). When you say `draw(200,100)` it walks to (200,100), but it draws a line underneath it as it goes.

The "last pixel it did anything with", or the "place where the robot is", is also called the "current point". Mostly because that's so much quicker to say than either of those other terms.

Facing the facts

Let's try to draw something a little more interesting. How about a face? (Later we'll extend that by drawing a person.)

Unfortunately, it's not easy to draw anything at all like a face with straight lines. Faces are sort of round, after all. So …

Going round in circles

. . . we need to be able to draw things that are sort of round. Circles, for instance. Try this little program.

```
from livewires import *
begin_graphics()
circle(300,200,100)
```

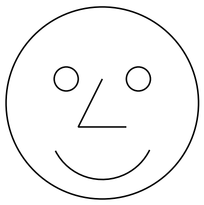
The three numbers are the two coordinates of the centre of the circle, and the radius of the circle (i.e., the distance from the centre to the edge). Try running the same program but with different values for the three numbers, to make sure you understand what they all do.

What does this do? (Try to guess before you actually run it.)

```
from livewires import *
begin_graphics()
for r in 100,110,120,130,140:
  circle(300,200,r)
```

If you still aren't sure, ask for help. . .

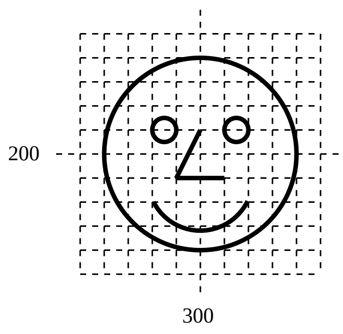
So, what about drawing that face? We'll try to draw something that looks a bit like this:



In other words, we need to draw

- One large circle (around the outside)
- Two small circles (the eyes)
- *Part* of a circle (the mouth)
- Two straight lines (the nose)

You'll find this easier if you have a bit of squared paper or graph paper. Failing that, you might find it helpful to draw some for yourself! Then, sketch the face on the squared paper:



(If you're really too lazy, you can just use the face-on-a-grid above!)

That grid is 10 squares on each side. Let's suppose that the very centre of the grid is the point (300,200) (there's nothing special about that point; it's just somewhere near the middle of the graphics window), and that each of our little grid squares is 10 pixels across.

The mouth is a little tricky, so to begin with we'll leave it out. Everything else should be pretty easy to locate. For instance, the big circle has its centre at (300,200) and its radius is 4 small squares, or 40 pixels. The left eye has its centre at (285,210) and its radius is 5 pixels (half a square). I'll let you work out all the other coordinates you need. So:

Write a program that draws all of that face except the mouth.

Mouthing off

The mouth, as I said, is harder. We need to be able to draw just *part* of a circle. We can do this by saying where the centre of the circle is, and where the ends of the line are. In the picture above, the endpoints of the mouth are at (280,180) and (320,180). It's not quite obvious where the centre should be; I reckon it's somewhere near (300,195) – just a little below the centre of the big circle.

It turns out that what you need to add to your program is:

```
        circle(300,195, endpoints = ((280,180),(320,180)))
```

That's a bit intimidating, so I'll try to explain. First you give the centre of the circle, just as if you were drawing all of it. Then, to tell the computer that you don't want a whole circle but just a bit of one, you tell it where the *endpoints* of the bit of circle are. You say where they are by giving their positions, as coordinates. You'll have noticed the radius isn't given; that's because the endpoints are on the circle so Python can figure out the radius from them.

Challenge: Modify your face-drawing program so that it draws the face in a different place. You'll need to change all the numbers that define positions of points. You *won't* need to change the numbers that say how big the circles are!

Challenge: Make some other changes to the face, to get the hang of how these graphics commands work. For instance:

- Move the eyes up a little.
- Add eyebrows (lines across the eyes, maybe sloping a bit)
- Add ears, or hair.
- Change the shape of the nose slightly.

Two heads are better than one

What if you wanted to draw two of these faces?

You could copy out everything in your program twice, and then change one copy. This would work, but it would be boring and easy to get wrong. And if you then wanted a third face, you'd have to go through all the effort again.

There's a better way.

You already know that Python lets you give names to numbers, strings, lists and other things. It also lets you give names to bits of program. A piece of Python program with a name is called a "function" (for complicated reasons that don't matter just now). Sometimes the word "procedure" is used instead. (For instance, the language Logo – which you may have used in school – calls them procedures.)

Functions

*There's much more about functions in Sheet F (*Functions*). When you've read the following stuff, you might like to turn to Sheet F. It* might *also be helpful if you get confused.*

Type in the following. (The "prompt" Python prints at the start of each line will change, like in the `for` loop examples way back in Sheet 1. Don't forget to look out for spaces at the starts of lines!)

```
>>> def shout():
...   print 'Python is wonderful!'
...   print 'And so am I.'
...                                          Just press [Enter] here.
```

What you've just done is to teach Python a new trick. If you now say

```
>>> shout()
```

what do you think will happen? Try it. What you've done is to "define" a new word that Python will understand. (`def` is short for "define".)

Challenge: Write a definition so that saying `moan()` will make the machine print

```
Python is useless!
And so are these worksheets.
```

Try it out and make sure it works.

A good argument

Functions are more than just ways of saving typing, though. Try typing in the following definition, and think about what it might mean.

```
>>> def twice(x):
...   print x, '+', x, 'is', x+x
...                                          Just press [Enter] here.
```

When you think you understand it (or when you've decided you'll never understand it), experiment with it. Ask Python for `twice(3)`, or `twice(99)`, or even `twice('ouch')`.

What's going on here is rather clever. When you say `twice(7)`, the machine does the stuff inside the definition of `twice` (just like it did for `shout` and `moan` earlier), except that "`x` is 7". So it's rather as if you'd said

```
print 7, '+', 7, 'is', 7+7
```

For some reason, things that get put inside function definitions in this way are called "arguments". So, when you say `twice('beri')`, the string `'beri'` is the argument to the function `twice`. You'll need to know this word to understand what happens if you type `twice()` instead of `twice(7)`. (Try it.)

Challenge: Write a function that behaves a bit like `twice` but, instead of telling you what `x+x` is (where `x` is the argument of the function), tells you what `x*x` is. Try it out. (Unfortunately it won't work on strings!)

A function can have more than one argument. The way this works is pretty obvious once you've seen it, so I'll just give an example.

```
def add(x,y):
  print x, '+', y, '=', x+y
```

Then you can say `add(7,5)` and the computer will tell you what happens when you add 7 and 5.

Built-in functions

Incidentally, lots of the things you've told Python to do are really functions. For instance, when you say `move(100,100)` you're just using a function with two arguments. The only difference is that you didn't have to write the definition of the `move` function yourself.

As I was saying. . .

This is all a digression. Before I started blathering about functions, we were thinking about how to draw two (or more) of those faces. Maybe you've guessed what's coming next: the Right Thing to do is to write a *function* that draws a face. The function should take arguments saying where in the graphics window the face should go. When we've done that, drawing 5 faces will just mean 5 uses of our function, instead of having to copy out the face-drawing program 5 times and make lots of changes to each copy.

So, what should our face-drawing function look like?

It needs to be able to be told where to draw the face, so let's give it two arguments saying where (like the two arguments to `move` or `draw`).

So far, we know that the function will look like this.

```
def draw_face(x,y):
  blah blah                        Whatever you need to draw a face at (x,y)
```

Obviously filling in the `blah blah` is the difficult bit.

What exactly should the arguments, `x` and `y`, mean? When we were designing the face, we thought about the positions of all the bits of the face relative to the point right in the middle of the big circle. (We took that as (300,200), remember?) So a reasonable idea would be to say that `x` and `y` are the coordinates (i.e., the two parts of the name) of the pixel right in the middle of the face. So, saying `draw_face(300,200)` ought to do exactly the same as the program we've already written.

Challenge: If you're feeling brave, go back to your face-drawing program and try to turn it into a face-drawing function. Then test the function. — If you're not feeling brave, read on and I'll try to lead you through the process.

(If you want to take up that challenge, you should stop reading right now.)

To begin with, edit your face-drawing program so that instead of saying

```
first line
second line
third line
etc
```

it says something like

```
first line
def draw_face(x,y):
  second line
  third line
  etc
```

You need to leave a couple of the earlier lines outside the function; things like the `from livewires import *` that aren't to do with drawing the face.

(If you try running your program right now it won't do anything because you need to add something else, but we'll come to that later.)

You now have a function that takes two arguments, and then ignores them and draws a face centred at (300,200) whatever the arguments were. Now we need to fix it up so that the face is centred at (`x`,`y`) instead.

The outline

The first thing we drew originally was the circle around the outside of the face. This also happens to be the easiest thing to adapt for putting into our function.

Originally, we had a line `circle(300,200,40)`. The first two arguments to `circle` say where the centre of the circle is, so they should become `x,y`. The last argument (`40`) says how big the circle is. All our faces are going to be the same size, so that should be left alone.

(Incidentally, you'll find all this *much* easier to follow if you look at the face-on-a-grid earlier in this sheet.)

So the new line should be `circle(x,y,40)` .

The eyes

We drew the eyes next. Originally the eyes were at (285,310) and (315,310): 15 pixels away from the middle horisontally (one in each direction), and both 10 pixels above the centre.

Well, that's easy enough. Here's how to draw the left eye; you can probably work out how to draw the right eye.

```
        circle(x-15, y+10, 5)
```

The nose

The nose is made up of two lines. The first one goes from 10 pixels above the very centre of the face to 10 pixels leftward and 10 pixels downward from the centre. That's easy enough.

```
        move(x, y+10)
        draw(x-10, y-10)
```

The second line goes from where the first one left off, to a point 10 pixels down and 10 pixels *to the right* from the centre of the face. I'll leave drawing that line up to you.

The mouth

You might expect this to be harder than the other bits. It isn't, really. You just need to work out what all the numbers are, relative to the centre of the face.

For instance, the left-hand endpoint of the mouth was at (280,180). That's 20 pixels left and 20 pixels down from the centre at (300,200), so it's `(x-20,y-20)`.

I'm going to be mean, and leave you to work out exactly how to adapt the line of program that draws the face's mouth. It really isn't very difficult.

Does it work?

Test your function. I said above that you need to do something else, and here it is. Add a few lines below the function:

```
        draw_face(300,200)
        draw_face(400,200)
        draw_face(350,270)
```

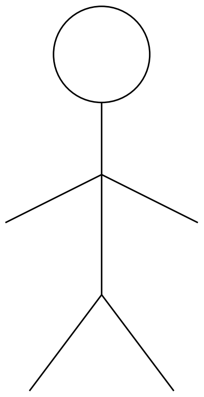
and run the program. Do you get three heads? Or a ghastly mess of detached bits of head? Or what?

It's important to put those lines below *the definition of* `draw_face`*, because until it's read that definition the computer doesn't know what* `draw_face` *is.*

The whole person

If you've had enough of graphics, you might want to skip ahead a little. But if you're still having fun, you might like to use your head-drawing function to make a person-drawing function.

Here's a simple stick figure. Sketch something like this on graph paper or squared paper, and use it to design a function that draws a stick person. Where I've just drawn a circle for the head, you should actually use the `draw_face()` function.



This may take you a while. You'll have to think carefully about what arguments you give to `draw_face` to make the head and the body fit together properly.

Make sure your functions work by making your program draw several stick figures in different places on the screen.

You may well find it much easier if you start by just drawing a single stick figure anywhere, and then turn it into a function in the same sort of way as you did for the face.

Challenge: Work out how to give one of your stick figures a hat.

Some nice patterns

This doesn't have anything to do with the stick figures. I'm just including it because I think it looks nice. Give it a try and see if you agree.

```
from livewires import *
begin_graphics()
for i in range(0,400):
  move(i,0)
  draw(0,400-i)
```

The weird thing about this is that all it does is to draw a lot of straight lines, but the result is to produce a nice smooth curve!

Ending it all

At the end of any program that draws graphics, you should actually have a line

```
end_graphics()
```

to tell the computer that you're finished with the graphics window. When it sees that line, the computer will wait for a moment then close the graphics window. (If it just went ahead and closed it right away, you wouldn't have time to see what it had drawn in the window. That would be a shame.)

What next?

- Read Sheet G (*Graphics*) for more information on doing graphics with Python
- Work out how to draw faces and people of different sizes as well as in different places. Work out how to make them face in different directions, too. Then you can draw a scene in which some of them are talking to others, or standing alone, or whatever.

You might be interested in some of the things being done in the Computer Graphics group, too.

You could also just go on to Sheet 4, in which you'll write a simple game. (It doesn't use graphics, I'm afraid.)