Game: Pacman

Neil Turton

Revision 1.7, May 14, 2001

Introduction

This is a Python games worksheet. When you've finished working through it, you'll have a version of the classic arcade game called Pacman.

What you need to know

- The basics of Python (from Sheets 1 and 2)
- Functions (from Sheet 3; you might want to look at Sheet F too)
- Lists (from Sheet A)
- Classes and Objects (see Sheet O)

You will need to look over these sheets before you start. If you get stuck, you can always ask for help.

What is Pacman?

First, I'll describe the game of Pacman. No doubt some of the details described here are different from the original Pacman game, but the result is still an enjoyable game. The game takes place in a maze filled with little bits of food. Our champion, Pacman, is yellow, round and very hungry. The object of the game is to guide Pacman round the walls of the maze to eat all the bits of food which are lying around.

"Easy!" you may think, but Pacman's task is made harder by the brightly coloured ghosts which haunt the maze, drifting aimlessly about, and Pacman will faint from fright if he ever bumps into one of those. When this happens, Pacman and all the ghosts return to their starting positions. If this happens too many times (three times in fact), the fright will be too much for Pacman and he will surely die.

The only form of defence Pacman has against the ghosts is a capsule (or pill?) which gives Pacman power to send the ghosts back to where they came from. When a capsule is consumed, all the ghosts turn white (or light blue) and if Pacman captures a ghost which is white, it returns to its original starting position and colour. After this has happened, Pacman is again powerless against that ghost because it has returned to its original colour (This is the best explanation I have come up with. If you have a better one, please let me know).

After a while, however, the power of the capsule begins to wear off and the ghosts will start showing their original colour, switching rapidly between their colour and white as they do so. While this is happening, Pacman can still safely capture the ghosts. Then, when the power of the capsule wears off completely, all the ghosts return to their original colours, and again, Pacman is powerless against them.

When Pacman completes the task of eating all the food which has been left in the maze, he quickly moves on to another level which is in some way harder than the last. Perhaps the ghosts move faster, or there are more of them, or they start trying to find Pacman in the maze to give him the fright of his life.

Not only does Pacman have to avoid the ghosts, but he must also move quickly because there is only a limited time available for him to collect the food. When Pacman completes a level he is rewarded with points for the time which is left remaining. And what do points mean? Prizes! (Never mind.) Points are also given for each ghost captured.

The traditional maze also had a tunnel passing from one side of the screen to the other. Anything which entered the tunnel on one side of the screen would appear on the other side.

How the grid works

The grid is made up of imaginary vertical and horizontal lines (which I'll call *grid lines*) which make up squares. The places where vertical grid lines cross horizontal grid lines, I'll call *grid points*. Walls will lie along grid lines, and Pacman and the ghosts will travel along grid lines. The food and capsules will sit on grid points.

Each grid point is identified by a pair of numbers: Its row number and its column number. It will also have its coordinates on the screen. I'll call the first pair of numbers *grid coordinates* and the second pair *screen coordinates*.

Things on the screen

The kinds of object which take part in the game are: Maze, Pacman, Food, Walls, Ghosts and Capsules. Some of these (Pacman and ghosts) can move, but not through walls - we'll call these *movables*. Others never move (walls, food and capsules). We're going to define a class for each of these kinds of object. You may like to skim-read this section and refer back to it later.

The Maze class

The `Maze` contains all the other objects in a grid of squares. It's main job is to keep track of where all the objects are. It will deal with two types of object: those that do and those that don't. `Movable` objects will ask the `Maze` about walls which might be in the way. Pacman will keep telling the `Maze` where he is, and the `Maze` will check for collisions with other objects.

The Immovable class

This class is a superclass for objects which are stationary. When one of these objects is created it is expected to draw itself on the screen. The `Maze` object will keep a note of which object is at each location.

The Wall class

Walls are pretty boring. They don't move. They don't do anything. They just sit there on the screen, stopping movables from passing. They are a kind of `Immovable` because they don't move. The walls lie along grid lines, but it turns out to be easier if we have a `Wall` object at each grid point which the wall passes through. If a `Wall` object is next to another `Wall` object, we will draw a line on the screen between them.

The Food class

These are a kind of `Immovable`. When eaten, they tell the `Maze` object that there is one less bit of `Food` to collect.

The Capsule class

These are also a kind of `Immovable`. When eaten, they tell the `Maze` that all the `Ghost` objects should turn white. The `Maze` then tells each `Ghost` to turn white.

The Movable class

These can move about the maze, but they need to check that there aren't any walls in the way. We're only going to allow movables to travel along grid lines. The `Maze` will keep telling the movables to move, but what happens then depends on which kind of `Movable` it is.

The Pacman class

Our hero! `Pacman` is a kind of `Movable`, which moves when the player holds a key down. `Pacman` will keep telling the `Maze` where he is, and the `Maze` will check for things he might have bumped into.

The Ghost class

A `Ghost` is also a kind of `Movable`. They don't interact with anything except `Wall` objects and `Pacman`. Each `Ghost` will keep travelling in the same direction until it reaches a grid point. Then it will decide on a new direction to travel in.

The first working program

It's always satisfying to get something working early on, even if it can't even be played. We'll start with a simplified version of the game which is missing most of the features. Having done that, we can add things to it until it does everything we want it to. We'll eventually go through the following steps:

- Make a Maze with just Walls,
- Add Pacman,
- Add Food,
- Add the Ghosts,
- Add the Capsules.

After each of these steps we'll have a working program and at the end, we'll have a playable game. There will still be things we can do to improve it after that.

An outline of the program

The main program is simple.

```
from livewires import *                          As usual.

# ...                                            Class definitions go here.

the_maze = Maze()                                Make a Maze object.
while not the_maze.finished():                   Keep playing until we're done.
  the_maze.play()
the_maze.done()                                  We're finished.
```

Don't type the lines which start `# ...`. They are just there to indicate that something else belongs there.

The basic outline is that we make the `Maze`, play until we're finished and then stop. Of course this code won't work yet because we haven't defined the `Maze` class, but we'll never need to change it because all we need is the right definition of the `Maze` class. All we will do is put some definitions after the first line.

Colours and sizes

It is often useful to have definitions of things you might want to change at the top of your program. When you decide you'd like a green Pacman and tiny graphics, you can just make a couple of changes at the top of the program and it's done. We'll start with a few things and add to them later.

```
grid_size = 30
margin = grid_size

background_colour = Colour.black
wall_colour = make_colour(0.6, 0.9, 0.9)
```

This sets the size of everything.
How much space to leave round the edge.

The colours we use.

The Maze layout

One of the first challenges is to tell the computer what the Maze looks like in a way that is easy for both the computer and you to understand. One good solution to this is to have a list of strings with one string for each row (horizontal grid line) of the maze, and one character in the string for each grid point along the row. Different characters will have different meanings; we'll have "G" for Ghost and so on. We'll put all of the objects into the layout now, so that we don't have to change it. We'll then write the program so that anything it doesn't recognise is just ignored. I think it's best if I show you.

```
# The shape of the maze.  Each character
# represents a different type of object.
#   % - Wall
#   . - Food
#   o - Capsule
#   G - Ghost
#   P - Pacman
# Other characters are ignored.

the_layout = [
  "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%",
  "%.....%.................%.....%",
  "%o%%%.%.%%%.%%%%%%%.%%%.%.%%%o%",
  "%.%.....%......%......%.....%.%",
  "%...%%%.%.%%%%.%.%%%%.%.%%%...%",
  "%%%.%...%.%.........%.%...%.%%%",
  "%...%.%%%.%.%%% %%%.%.%%%.%...%",
  "%.%%%.......%GG GG%.......%%%.%",
  "%...%.%%%.%.%%%%%%%.%.%%%.%...%",
  "%%%.%...%.%.........%.%...%.%%%",
  "%...%%%.%.%%%%.%.%%%%.%.%%%...%",
  "%.%.....%......%......%.....%.%",
  "%o%%%.%.%%%.%%%%%%%.%%%.%.%%%o%",
  "%.....%........P........%.....%",
  "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"]
```

This is here to remind us what all the different characters mean.

There are 31 %s in this line.

You'll need a layout like this in your program. If you like, you can make changes to it later, but stick with this one for now.

The first few lines (which begin with #) are *comment* lines which means that Python ignores them when it sees them. They are just there for people to read. It will be useful to have a reminder in the program for what the different characters mean. Writing comments in your programs is a very good thing because it will help you to understand your program when you come back to it and it is almost impossible to have too many.

Looking at the section of program above, you can see what the maze is going to look like. The computer has an easy job as well. To find out what character is on row `Y` in column `X`, we just say `the_layout[max_y - Y][X]`. If you were expecting `the_layout[Y][X]`, consider the top line: This will have the highest row number, `max_y`, but it is entry 0 in the list. Every time we add 1 to the row number, we must subtract 1 from the entry number, so the row number and entry number always add up to `max_y`.

The Maze class

The Maze class will keep `Movable` objects separate from those which don't move. It will keep those which move in a list and it will go through the list every time it wants to find one.

This wouldn't be very efficient for the objects which don't move because the list would be very long. There are over 200 bits of food scattered around the maze and going round each one to ask "Are you touching Pacman?" would take a long time. If you pointed at each dot in the layout and said "Are you touching Pacman?" you'd get bored pretty quickly. For the `Movable` objects it's a different matter. There are only 5 of them in total and going round each of them in turn won't take long at all.

To keep track of the stationary objects we will have a list for each row in the maze, with an entry for each grid point, which will contain the object at that point. We will keep these lists in yet another list. Let me say that another way. There is a list for the `Maze` which contains a list for each row in the `Maze` which contains an object for each grid point in the row. The aim here is to be able to easily answer the question: "What is at the grid point in row `Y` and column `X`?". To answer this question, we first find the list for the row, and then find the object in that list for the column. In Python this looks like `map[Y][X]`.

There are several things the Maze class will need to do. The first is to build the map from the layout we defined earlier. In the process, it will bring up a window on the screen and create objects that will draw into it. Each character in the layout will have an object associated with it.

```
class Maze:
  def __init__(self):
    self.have_window = 0                  We haven't made the window yet.
    self.game_over = 0                    The game isn't over yet.
    self.set_layout(the_layout)           Make all the objects.
```

We have 2 variables in the `Maze` object. The first tells us if we've called `begin_graphics` yet. The second tells us when to stop playing the game. The final thing we do is to call the `set_layout` method which we've not yet defined. Having a `set_layout` method will be useful if we decide to have another level with a different maze layout.

Now we'll define the `set_layout` method. It works out the size of the layout, makes a map of the maze and adds all the objects from the layout definition into the map.

```
class Maze:                               This should already be in your program.
  # ...                                   Other definitions will be here.

  def set_layout(self, layout):
    height = len(layout)                  The length of the list
    width = len(layout[0])                The length of the first string
    self.make_window(width, height)
    self.make_map(width, height)          Start a new map

    max_y = height - 1
    for x in range(width):                Go through the whole layout
      for y in range(height):
        char = layout[max_y - y][x]       See the discussion 1 page ago.
        self.make_object((x, y), char)    Create the object
```

The first line works out the height of the layout. The `len` function is a function which is supplied by Python. If the `len` function is given a list it will return the number of items in the list. Our layout list has an item (string in this case) for each row of the maze. The second line works out the width of the layout. If the `len` function is given a string, it will return the number of characters in the string. We count the number of characters in the first line of the layout to give the width of the maze. Working out the size of the maze in this way means that we can have different layouts of different sizes.

The Maze class is also responsible for creating the window and managing the size of things. The `make_window` method makes sure the window is large enough to contain the maze.

```
class Maze:                                   This should already be in your program.
  # ...                                       Other definitions will be here.

  def make_window(self, width, height):
    grid_width = (width-1) * grid_size        Work out the size of the window.
    grid_height = (height-1) * grid_size
    screen_width = 2*margin + grid_width
    screen_height = 2*margin + grid_height
    begin_graphics(screen_width,              Create the window.
                   screen_height,
                   background_colour)
    allow_movables()                          Most objects move or disappear.
```

Working out the size of the window need a little explanation. The number of grid points across the maze is `width`, but the grid points don't really take up any space. The space is taken up by the gap between the grid points and the number of these is one less than the number of points. If there were three points, there would only be two gaps. One between points 1 and 2, and one between points 2 and 3.

The values of `grid_width` and `grid_height` use `width-1` and `height-1` because that is the number of gaps between grid points that there are. There is a margin between the edge of the screen and the first grid point on all 4 sides of the screen.

We'll be using a method for converting from grid coordinates to screen coordinates. The game will operate in grid coordinates, but whenever something needs drawing on the screen we'll need to know the screen coordinates.

```
class Maze:                                   This should already be in your program.
  # ...                                       Other definitions will be here.

  def to_screen(self, point):
    (x, y) = point
    x = x*grid_size + margin                  Work out the coordinates of the point
    y = y*grid_size + margin                  on the screen.
    return (x, y)
```

We've now got 2 more methods to define. The first is `make_map` which will create an empty map of the maze, but one thing we haven't yet considered is: What do we put in the map for an empty grid point?

One of the best things to put in an empty map entry is an object that represents an empty grid point. This may seem quite odd at first, to have something to represent nothing, but it makes writing the program a lot easier. You don't have to keep checking to see if the grid point you're dealing with is empty before doing something with object. Each grid point has an object, so you can safely use it. Empty points will have an object in the `Nothing` class. [Actually the number `0` and the word "nothing" both represent nothing. We're just doing the same thing here.]

A useful thing to know is that if you find you've got a bunch of objects in a list somewhere, they've probably got something in common. (Otherwise why did you put them in the same list?) If two different kinds of object have something in common, they should often have the same superclass. This is particularly the case if they are in the same list, because you can make sure that all the objects in the list have a particular method by putting it in the superclass. You'll see how we do this later with the `is_a_wall` method. The superclass in this case is the `Immovable` class.

```
class Immovable:
  pass                                     We have nothing to put in this class yet.

class Nothing(Immovable):
  pass

class Maze:                                This should already be in your program.
  # ...                                    Other definitions will be here.

  def make_map(self, width, height):
    self.width = width                     Store the size of the layout.
    self.height = height
    self.map = []                          Start with an empty list
    for y in range(height):
      new_row = []                         Make a new row list
      for x in range(width):
        new_row.append(Nothing())          Add an entry to the list
      self.map.append(new_row)             Put the row in the map
```

We will now make a start on the `make_object` method. This will check the character from the layout and create an object of the right class if it knows how. For the moment, we'll only check for the wall character which is "%". If it is a wall, it creates a Wall object and puts it into the map.

```
class Maze:                                This should already be in your program.
  # ...                                    Other definitions will be here.

  def make_object(self, point, character):
    (x, y) = point
    if character == '%':                   Is it a wall?
      self.map[y][x] = Wall(self, point)
```

We're nearly finished with the `Maze` class. What we need now are definitions of `finished`, `play` and `done` methods. We arrange for the maze to appear on the screen until the window is closed.

```
class Maze:                                This should already be in your program.
  # ...                                    Other definitions will be here.

  def finished(self):
    return self.game_over                  Stop if the game is over.

  def play(self):
    sleep(0.05)                            Just pass the time.

  def done(self):
    end_graphics()                         We've finished
    self.map = []                          Forget all the objects
```

The Wall class

Now we define the Wall class. The `__init__` method will be called with 2 extra parameters which are the Maze object and the grid coordinates. (Remember, the first parameter is always the object which is being created.) Since this is the first version of the program, we're going to simplify things a little and just draw a filled circle at each Wall object.

```
class Wall(Immovable):
  def __init__(self, maze, point):
    self.place = point                          Store our position.
    self.screen_point = maze.to_screen(point)
    self.maze = maze                            Keep hold of the Maze.
    self.draw()

  def draw(self):
    (screen_x, screen_y) = self.screen_point
    forbid_movables()
    dot_size = grid_size * 0.2
    circle(screen_x, screen_y, dot_size,        Just draw a circle.
           colour = wall_colour, filled = 1)
    allow_movables()
```

Trying it out

And that's it. If you've typed in all of the program sections above, you should have the first working program. Try it out! Show your friends! It's not very impressive yet, but it does something. When you have finished looking at the maze, close the window and the program will finish (rather ungracefully, but never mind about that).

Challenge: Design a maze layout which is better than the one I did.

Making nicer walls

Now we've got a working program, we can begin on our improvement campaign. We've got a maze, but it doesn't look very nice. We'll start by making the maze look a bit nicer, by drawing lines between the circles. You can either leave the circles in, or take them out.

Consider two `Wall` objects next to each other. Somehow we need to draw a line between them. The question now is: How does a Wall object know that it's next to another `Wall` object? If you look back at the `make_object` method you'll see that as soon as the `Wall` object has been created, it is put into the map of the `Maze`.

When a `Wall` object gets created we'll have it ask the `Maze` about the surrounding grid points and when it finds a `Wall` object, it will draw a line between them. We'll never draw a line twice because it's always up to the `Wall` object which was created later to draw the line.

So far the `Maze` has a map which tells it what object is at each grid point. We need to know about this from the `Wall` class, so we need to add a method to the `Maze` class so we can ask it "what is the object at this point?". One thing this will need to check for is the possibility of someone asking about a point which is outside our map. In this case, we'll return an `Nothing` object. Without further ado, here it is:

```
class Maze:                                     This should already be in your program.
  # ...                                         Other definitions will be here.

  def object_at(self, point):
    (x, y) = point

    if y < 0 or y >= self.height:               If the point is outside the maze,
      return Nothing()                          return nothing.

    if x < 0 or x >= self.width:
      return Nothing()

    return self.map[y][x]
```

Since there are `self.height` rows in the maze and they are numbered from zero, there is no row with a number `self.height` or higher. Similarly for columns.

So far, the only objects that can be at a grid point are: A `Wall` object or an `Nothing` object. When a new `Wall` object gets created it will want to know if the object next door is another `Wall` object. The simplest approach is to ask it. We'll arrange for all the objects which have `Immovable` as a superclass to have a method called `is_a_wall` which returns true if the object is a `Wall` object, and false otherwise.

Looking ahead, we'll see that the answer for most objects will want this method to return false, but a the Wall object will want to return true. Being lazy, we only want to write a method called `is_a_wall` which returns false, once. The way to do this is to put this method in the `Immovable` class, and override it in the `Wall` class.

When we come to write the Food class we can just make it a subclass of `Immovable` and not bother about having to write the `is_a_wall` method (and other methods we put in the `Immovable` class).

```
class Immovable:                                    This should already be in your program.
  def is_a_wall(self):
    return 0                                        Most objects aren't walls so say no

class Wall(Immovable):                              This should already be in your program.
  # ...                                             Other definitions will be here.

  def is_a_wall(self):
    return 1                                        This object is a wall, so say yes
```

We're now in a position for a `Wall` object to check for neighbouring `Wall` objects. We'll invent the method `check_neighbour` that will look at one of the neighbours and draw a line between them if it's another wall.

```
class Wall(Immovable):                              This should already be in your program.
  # ...                                             Other definitions will be here.

  def draw(self):
    (screen_x, screen_y) = self.screen_point
    forbid_movables()
    dot_size = grid_size * 0.2                      You can remove...
    circle(screen_x, screen_y, dot_size,            ...these three lines...
           colour = wall_colour, filled = 1)        ...if you like.
    (x, y) = self.place
    neighbours = [ (x+1, y), (x-1, y),              Make a list of our neighbours.
                   (x, y+1), (x, y-1) ]
    for neighbour in neighbours:                    Check each neighbour in turn.
      self.check_neighbour(neighbour)
    allow_movables()

  # ...                                             Other definitions will be here.
```

The `check_neighbour` method will ask the Maze for an object and then ask that object if it's a wall or not.

```
class Wall(Immovable):                              This should already be in your program.
  # ...                                             Other definitions will be here.

  def check_neighbour(self, neighbour):
    maze = self.maze
    object = maze.object_at(neighbour)              Get the object.
    if object.is_a_wall():                          Is it a wall?
      here = self.screen_point                      Draw a line from here...
      there = maze.to_screen(neighbour)             ... to there if it is.
      line(here, there, colour = wall_colour)
```

You should now be able to try the program and see the nicer maze. Next we'll add Pacman.

Introducing Pacman

Pacman is our first moving object. Lets have a look at what Pacman needs to do. We'll arrange for the Maze class to call the `move` method on all the Movables on a regular basis. When this gets called we need to check which keys are pressed and if a direction key is pressed, try to move in that direction. Pacman is only allowed to follow grid lines, so if the he's not on grid line which lies in that direction, we'll move him to move towards it. If Pacman is on a grid line which is in the right direction, we move him along it unless he's about to hit a wall.

We'll start with the Movable class. All the initializer will do is make a note of the Maze object, set the current position to the point passed in, and remember the speed.

```
class Movable:
  def __init__(self, maze, point, speed):
    self.maze = maze                               For finding other objects
    self.place = point                             Our current position
    self.speed = speed                             Remember the speed
```

Next, we'll start the Pacman class. We'll start with the `move` method. This is fairly straightforward. We just check each key in turn and if it's pressed, move in that direction. We'll invent methods like `move_left` to make things easier.

```
class Pacman(Movable):
  def __init__(self, maze, point):
    Movable.__init__(self, maze, point,            Just call the Movable initializer
                     pacman_speed)

  def move(self):
    keys = keys_pressed()
    if   ’z’ in keys: self.move_left()             Is the ‘‘z’’ key pressed?
    elif ’x’ in keys: self.move_right()            Is the ‘‘x’’ key pressed?
    elif ’;’ in keys: self.move_up()               ...
    elif ’.’ in keys: self.move_down()             ...
```

The bit which says `Movable.__init__` is the way to get at the `__init__` method in the `Movable` class. Because we haven't yet told it which object to use, we have to pass the object in as the first parameter.

Now that we've invented the 4 movement methods, we're going to have to write them. We'll invent another method called `try_move` which takes an amount to add to the X coordinate and an amount to add to the Y coordinate, and tries to move `Pacman` to the new place. To move one grid point to the right, the numbers would be (1, 0). To move one grid point down it would be (0, -1). The pair of numbers to add to the coordinates is called a *vector*.

```
class Pacman(Movable):                             This should already be in your program.
  # ...                                            Other definitions will be here.

  def move_left(self):
    self.try_move((-1, 0))

  def move_right(self):
    self.try_move((1, 0))

  def move_up(self):
    self.try_move((0, 1))

  def move_down(self):
    self.try_move((0, -1))
```

Now it's down to the `try_move` method to figure out what needs to happen. It will first move Pacman towards the nearest grid line if he's not already on one. It will then move him along it if there isn't a wall in the way.

One of the important things about movables (like Pacman) is that they aren't always at grid points. When asking the `Maze` about fixed objects like `Food` and `Walls`, it always needs to be given a grid point, and not some point between grid points.

In other words, the coordinates given must be whole numbers. The best we can do is to find the nearest grid point to where we want. We'll pretend for the moment that we've solved this problem and have a method called `nearest_grid_point`, which returns the nearest point to the current position of the `Movable`, but we must remember to write it later.

We'll also invent methods called `furthest_move` and `move_by`. The first will take a vector to move by (which must be no larger than 1 grid unit) as a parameter and will return an vector which is how much we can move by without hitting a wall or going too fast. In effect, it answers the question: "Is it possible to move in this direction? If so, how far?". If the movement passed as a parameter is OK, it will just return that. If we would hit a wall by going that far, it will return the movement that is possible before we hit the wall. If going that far would mean moving too fast, it will return a suitably adjusted vector. If we're right next to the wall and trying to move towards it, it will return (0, 0) to indicate that we can't move in that direction at all. The `move_by` method will actually move Pacman and just takes a vector as a parameter. It assumes that the vector has already been checked by `furthest_move`.

```
class Pacman(Movable):                              This should already be in your program.
  # ...                                             Other definitions will be here.

  def try_move(self, move):
    (move_x, move_y) = move
    (current_x, current_y) = self.place
    (nearest_x, nearest_y) = (
        self.nearest_grid_point() )

    if self.furthest_move(move) == (0,0):           If we can't move, do nothing.
        return

    if move_x != 0 and current_y != nearest_y:      If we're moving horizontally
      move_x = 0                                    but aren't on a grid line
      move_y = nearest_y - current_y                move towards the grid line

    elif move_y != 0 and current_x != nearest_x:    If we're moving vertically
      move_y = 0                                    but aren't on a grid line
      move_x = nearest_x - current_x                Move towards the grid line

    move = self.furthest_move((move_x, move_y))     Don't go too far.
    self.move_by(move)                              Make the move.
```

After we've figured out that we can move, we make sure we're on a grid line in the direction we're trying to move. If the player tries to move right and they aren't on a horizontal grid line, we'll move them up or down until they are. This is so that the player doesn't have to line Pacman up exactly with a gap in a wall in order to go through. In this case `move_x` will not be zero and `current_y` will not be `nearest_y`, so we will set `move_y` to the distance to the nearest horizontal grid line. Instead of trying to move in the direction the player indicates, we move towards the nearest horizontal grid line. The same thing happens for vertical grid lines.

Next we will deal with the `furthest_move` method. For this, we just work out where the next grid point we are going to encounter is and ask the `Maze` if there is a wall there. It also limits the step according to the speed of the object. As before we will use a vector to describe where we're going. We will want to use this method for Ghosts as well, so we'll put it in the `Movable` class so that it can be shared.

```
class Movable:                                   This should already be in your program.
  # ...                                          Other definitions will be here.

  def furthest_move(self, movement):
    (move_x, move_y) = movement                  How far to move.
    (current_x, current_y) = self.place          Where are we now?
    nearest = self.nearest_grid_point()          Where's the nearest grid point?
    (nearest_x, nearest_y) = nearest
    maze = self.maze

    if move_x > 0:                               Are we moving towards a wall to the right?
      next_point = (nearest_x+1, nearest_y)
      if maze.object_at(next_point).is_a_wall():
        if current_x+move_x > nearest_x:         Are we close enough?
          move_x = nearest_x - current_x         Stop just before it.

    elif move_x < 0:                             Are we moving towards a wall to the left?
      next_point = (nearest_x-1, nearest_y)
      if maze.object_at(next_point).is_a_wall():
        if current_x+move_x < nearest_x:         Are we close enough?
          move_x = nearest_x - current_x         Stop just before it.

    if move_y > 0:                               Are we moving towards a wall above us?
      next_point = (nearest_x, nearest_y+1)
      if maze.object_at(next_point).is_a_wall():
        if current_y+move_y > nearest_y:         Are we close enough?
          move_y = nearest_y - current_y         Stop just before it.

    elif move_y < 0:                             Are we moving towards a wall below us?
      next_point = (nearest_x, nearest_y-1)
      if maze.object_at(next_point).is_a_wall():
        if current_y+move_y < nearest_y:         Are we close enough?
          move_y = nearest_y - current_y         Stop just before it.

    if move_x > self.speed:                      Don't move further than our speed allows
      move_x = self.speed
    elif move_x < -self.speed:
      move_x = -self.speed

    if move_y > self.speed:
      move_y = self.speed
    elif move_y < -self.speed:
      move_y = -self.speed

    return (move_x, move_y)
```

Next comes the `nearest_grid_point` method we promised. It takes the current coordinates and returns the nearest grid point. That is, it returns the nearest point which has whole numbers (or integers) as its coordinates. The way to find the nearest whole number to a number `x` is to say `int(x+0.5)`. Armed with this knowledge our job is quite easy.

```
class Movable:                                   This should already be in your program.
  # ...                                          Other definitions will be here.

  def nearest_grid_point(self):
    (current_x, current_y) = self.place
    grid_x = int(current_x + 0.5)                Find the nearest vertical grid line.
    grid_y = int(current_y + 0.5)                Find the nearest horizontal grid line.
    return (grid_x, grid_y)                      Find where they cross.
```

Now we need to tell the computer how to draw Pacman on the screen. So far the only method involved is the `move_by` method. This will move Pacman on the screen, but so far there isn't anything to put Pacman on the screen in the first place. We'll write a method called `draw` to do this.

```
pacman_colour = Colour.yellow                     Put these at the top of your program.
pacman_size = grid_size * 0.8                     How big to make Pacman.
pacman_speed = 0.25                               How fast Pacman moves.

# ...                                             Other classes will be here

class Pacman(Movable):                            This should already be in your program.
  def __init__(self, maze, point):
    self.direction = 0                            Start off facing right
    Movable.__init__(self, maze, point,           Call the Movable initializer
                     pacman_speed)

  # ...                                           Other definitions will be here.

  def draw(self):
    maze = self.maze
    screen_point = maze.to_screen(self.place)
    angle = self.get_angle()                      Work out half the mouth angle
    endpoints = (self.direction + angle,          Rotate according to the direction
                 self.direction + 360 - angle)
    self.body = circle(screen_point, pacman_size, Draw the sector
                       colour = pacman_colour,
                       filled = 1,
                       endpoints = endpoints)

  def get_angle(self):
    (x, y) = self.place                           Work out how far away the
    (nearest_x, nearest_y) = (
        self.nearest_grid_point() )               nearest grid point is.
    distance = ( abs(x-nearest_x) +
                 abs(y-nearest_y) )               Between -1/2 and 1/2

    return 1 + 90*distance                        This is between 1 and 46
```

The `abs` function takes a number and returns its *absolute* size. If the number is positive it just returns the number. If a number `n` is negative, `abs(n)` is the same as `-n`.

The `move_by` method is a little tricky. It works out where Pacman is moving to, draws a new body on the screen and then removes the old body. Doing things this way means that when we create the new body, we can make it a different shape from the old one (perhaps the mouth is wider). It also means that there is always a body on the screen so you don't catch a glimpse of the background when the old body is removed.

```
class Pacman(Movable):                            This should already be in your program.
  # ...                                           Other definitions will be here.

  def move_by(self, move):
    self.update_position(move)
    old_body = self.body                          Get the old body for removal
    self.draw()                                   Make a new body
    remove_from_screen(old_body)                  Remove the old body

  def update_position(self, move):
    (old_x, old_y) = self.place                   Get the old coordinates
    (move_x, move_y) = move                       Unpack the vector
    (new_x, new_y) = (old_x+move_x, old_y+move_y) Get the new coordinates
    self.place = (new_x, new_y)                   Update the coordinates

    if move_x > 0:                                If we're moving right ...
      self.direction = 0                          ... turn to face right.
    elif move_y > 0:                              If we're moving up ...
      self.direction = 90                         ... turn to face up.
    elif move_x < 0:                              If we're moving left ...
      self.direction = 180                        ... turn to face left.
    elif move_y < 0:                              If we're moving down ...
      self.direction = 270                        ... turn to face down.
```

As you can see we've also defined a method called `update_position`, which updates the coordinates and points `Pacman` in the direction in which he's moving.

Plumbing it in

Next we need to arrange for `Pacman`'s `move` method to be called on a regular basis. In fact all the Ghosts will want this as well, so there are lots of objects which want their `move` methods calling. We'll keep these objects in a list called `movables` in the `Maze` object, so that the `Maze` can call them in its `play` method.

First we'll change the `make_object` method in the `Maze` class so that when it finds a letter "P" in the layout, it creates a Pacman object.

```
class Maze:                                       This should already be in your program.
  # ...                                           Other definitions will be here.

  def make_object(self, point, character):
    (x, y) = point
    if character == '%':                          Is it a wall?
      self.map[y][x] = Wall(self, point)
    elif character == 'P':                        Is it Pacman?
      pacman = Pacman(self, point)
      self.movables.append(pacman)

  # ...                                           Other definitions will be here.
```

We also need to change the `set_layout` method in the `Maze` class to create a list of movables, so that `make_object` can put `Movable` objects in it. This goes in `set_layout` because whenever we want to change the layout, we want to start off with a clean sheet and not have Ghosts hanging about from the previous one.

When all of the objects have been created it also goes through the list and asks each one to draw itself on the screen. This happens after the stationary objects have been drawn, so that the Ghosts appear on top of all the `Food`, not on top of the `Food` which was created before them and below the `Food` which was created afterwards.

```
class Maze:                                       This should already be in your program.
  # ...                                           Other definitions will be here.

  def set_layout(self, layout):
    height = len(layout)                          This is as before.
    width = len(layout[0])
    self.make_window(width, height)
    self.make_map(width, height)
    self.movables = []                            Start with no Movables.

    max_y = height - 1
    for x in range(width):                        Make the objects as before.
      for y in range(height):
        char = layout[max_y - y][x]
        self.make_object((x, y), char)

    for movable in self.movables:                 Draw all of the movables.
      movable.draw()

  # ...                                           Other definitions will be here.

  def done(self):
    end_graphics()                                We've finished
    self.map = []                                 Forget all the stationary objects
    self.movables = []                            Forget all the moving objects

  # ...                                           Other definitions will be here.
```

Now we can change the `Maze`'s `play` method to call the `move` method on each of the movables and then wait a short period of time.

```
class Maze:                                        This should already be in your program.
  # ...                                            Other definitions will be here.

  def play(self):
    for movable in self.movables:                  Move each object
      movable.move()
    sleep(0.05)                                    Pass the time.
```

You can try your program again. Guide Pacman round the maze. Try moving through walls. Does it feel natural?

Food, glorious food!

The life-cycle of a `Food` object is as follows. It gets created and draws itself on the screen. The `Maze` puts the `Food` object in its map, so that when `Pacman` arrives at that grid point the food can be consumed. When this happens, `Pacman` tells the `Food` that it's been eaten and the `Food` object removes its image from the screen. It then tells the `Maze` to remove it from the map. When all the `Food` has been eaten, `Pacman` has completed his mission. The simplest way of finding out when this happens if for the `Maze` to keep a count of the number of `Food` objects there are.

We'll start by modifying `make_object` to create `Food`. It needs to keep a count of the number of `Food` objects, so we'll have to start with a count of zero. Again, this goes in the `set_layout` method because if we change layout, we will remove all the `Food` objects first.

```
class Maze:                                        This should already be in your program.
  # ...                                            Other definitions will be here.

  def set_layout(self, layout):
    height = len(layout)                           This is as before
    width = len(layout[0])
    self.make_window(width, height)
    self.make_map(width, height)
    self.movables = []
    self.food_count = 0                            Start with no Food

    max_y = height - 1                             Make the objects as before
    for x in range(width):
      for y in range(height):
        char = layout[max_y - y][x]
        self.make_object((x, y), char)

    for movable in self.movables:
      movable.draw()
```

Then every time a `Food` object is created we need to add 1 to the count.

```
class Maze:                                        This should already be in your program.
  # ...                                            Other definitions will be here.

  def make_object(self, point, character):
    (x, y) = point                                 As before...
    if character == '%':
      self.map[y][x] = Wall(self, point)
    elif character == 'P':
      pacman = Pacman(self, point)
      self.movables.append(pacman)
    elif character == '.':
      self.food_count = self.food_count + 1        Add 1 to the count.
      self.map[y][x] = Food(self, point)           Put a new object in the map.
```

Now we can write the Food class. We'll start by defining what happens when it's created: It gets drawn on the screen.

```
class Food(Immovable):
  def __init__(self, maze, point):
    self.place = point
    self.screen_point = maze.to_screen(point)
    self.maze = maze
    self.draw()
```

We need to define the `draw` method to actually draw the food on the screen.

```
food_colour = Colour.red                        Put this at the top of your program.
food_size = grid_size * 0.15                    How big to make the food.

# ...                                           Other definitions will be here.

class Food(Immovable):                          This should already be in your program.
  # ...                                         Other definitions will be here.

  def draw(self):
    (screen_x, screen_y) = self.screen_point
    self.dot = circle(screen_x, screen_y,
                      food_size,
                      colour = food_colour,
                      filled = 1)
```

Now, we need to arrange for the food to get eaten at the right time. The way we will do this is to make `Pacman` call a method called `eat` on the `Immovable` at a grid point when he gets close enough. It is then up to the object to do the right thing. We'll put a default method in the `Immovable` class which does nothing, so that nothing happens when `Pacman` tries to eat empty space. This is just telling Python that we really want nothing to happen and haven't forgotten about what should happen. We'll then override that method in the `Food` class. It may be useful to know who's doing the eating, so we will pass Pacman as a parameter.

```
class Immovable:                                This should already be in your program.
  # ...                                         Other definitions will be here.

  def eat(self, pacman):                        The default eat method
    pass                                        Do nothing
```

In the `Food` class we want this method to remove the `Food` from the screen and the `Maze`. To tell the `Maze` that some food needs removing, we'll invent a method called `remove_food`.

```
class Food(Immovable):                          This should already be in your program.
  # ...                                         Other definitions will be here.

  def eat(self, pacman):
    remove_from_screen(self.dot)                Remove the dot from the screen
    self.maze.remove_food(self.place)           Tell the Maze
```

We now need to write the `remove_food` method in the `Maze` class. When there is no food left, Pacman wins.

```
class Maze:                                          This should already be in your program.
  # ...                                              Other definitions will be here.

  def remove_food(self, place):
    (x, y) = place
    self.map[y][x] = Nothing()                       Make the map entry empty.
    self.food_count = self.food_count - 1            There is 1 less bit of Food.
    if self.food_count == 0:                         If there is no food left...
      self.win()                                     ... Pacman wins.

  def win(self):
    print "You win!"
    self.game_over = 1
```

So far, nothing is going to call the `eat` method. We need `Pacman` to do this, when he gets close enough. We'll do this in the `move_by` method. For this purpose close enough means $3/4$ of the distance `Pacman` moves each time round the main loop, on either side of the grid point.

```
class Pacman(Movable):                               This should already be in your program.
  # ...                                              Other definitions will be here.

  def move_by(self, move):
    self.update_position(move)                       As before...
    old_body = self.body
    self.draw()
    remove_from_screen(old_body)

    (x, y) = self.place                              Get the distance to
    nearest_point = self.nearest_grid_point()        the nearest grid point.
    (nearest_x, nearest_y) = nearest_point
    distance = ( abs(x-nearest_x) +                  As before.
                 abs(y-nearest_y))

    if distance < self.speed * 3/4:                  Are we close enough to eat?
      object = self.maze.object_at(nearest_point)
      object.eat(self)                               If so, eat it.
```

If you try the program now, you'll see that the game is starting to take shape. There's no challenge to it yet. That comes next.

Ghosts

A `Ghost` is another `Movable` object. The first thing we'll do is add it to `Maze`'s `make_object` method.

```
class Maze:                                          This should already be in your program.
  # ...                                              Other definitions will be here.

  def make_object(self, point, character):           You should also have this.
    # ...                                            Checks for other characters go here as before.
    elif character == 'G':                           Is it a Ghost?
      ghost = Ghost(self, point)                     Make a new Ghost
      self.movables.append(ghost)                    Add it to the list of Movables
```

The Ghosts need to wander through the maze at random. There are many ways to do this. The way we will do it is for each `Ghost` to choose a neighbouring grid point and move towards it. When it reaches that point, it will make another choice. It may take several calls of the `move` method to reach the next grid point, so the `Ghost` will have to remember which point it decided to move towards. We'll call this `next_point`. Our choice of next point will depend on the direction the `Ghost` was travelling, so we will need to store that as well.

Now we'll start the `Ghost` class itself. The first thing we need is an initializer. It takes the `Maze` and the starting grid point as an argument, and it will need to call the initializer in the `Movable`. We sneakily set the `next_point` to be the starting point, so that the `Ghost` will immediately choose to go somewhere else. The initializer also picks a colour from the list of ghost colours and sets the colour variable in the `Ghost` object. When the ghost is draw, it will get this colour.

```
ghost_colours = []                                  A list of all the Ghost colours.
ghost_colours.append(Colour.red)
ghost_colours.append(Colour.green)
ghost_colours.append(Colour.blue)
ghost_colours.append(Colour.purple)

ghost_speed = 0.25                                  How fast the Ghosts move.

class Ghost(Movable):
  def __init__(self, maze, start):
    global ghost_colours                            Tell Python to use the global
                                                    variable with this name.

    self.next_point = start                         Don't move anywhere to start with.
    self.movement = (0, 0)                          We were going nowhere.

    self.colour = ghost_colours[0]                  Pick a colour from the list.
    ghost_colours[:1] = []                          Remove it from the start of the list.
    ghost_colours.append(self.colour)               Put it back on the end.

    Movable.__init__(self, maze,                    Just call the Movable's initializer.
                     start, ghost_speed)
```

The next thing that's going to happen is that the `Maze` will call the `Ghost`'s `draw` method, so we need to write that. We need to tell the computer what a ghost looks like. We do this by giving a list of coordinates of points for the computer to join up.

```
ghost_shape = [                                     The coordinates which define the
    ( 0,    -0.5 ),                                 Ghost's shape. The coordinates in
    ( 0.25, -0.75 ),                                this list are measured grid units.
    ( 0.5,  -0.5 ),
    ( 0.75, -0.75 ),
    ( 0.75,  0.5 ),
    ( 0.5,   0.75 ),
    (-0.5,   0.75 ),
    (-0.75,  0.5 ),
    (-0.75, -0.75 ),
    (-0.5,  -0.5 ),
    (-0.25, -0.75 )
  ]

# ...                                               Other definitions will be here.

class Ghost(Movable):                               This should already be in your program.
  # ...                                             Other definitions will be here.

  def draw(self):
    maze = self.maze
    (screen_x, screen_y) = (
        maze.to_screen(self.place) )                Get our screen coordinates
    coords = []                                     Build up a list of coordinates.
    for (x, y) in ghost_shape:
      coords.append((x*grid_size + screen_x,
                     y*grid_size + screen_y))

    self.body = polygon(coords, self.colour,        Draw the body
                        closed = 1,
                        filled = 1)
```

Making them move

Next we need to make the Ghosts move. To do this, we first try to move towards `next_point`, and if we're getting nowhere, choose another point to move towards.

```
class Ghost(Movable):                          This should already be in your program.
  # ...                                        Other definitions will be here.

  def move(self):
    (current_x, current_y) = self.place        Get the vector to the next point.
    (next_x, next_y) = self.next_point
    move = (next_x - current_x,
            next_y - current_y)
    move = self.furthest_move(move)            See how far we can go.
    if move == (0, 0):                         If we're getting nowhere...
      move = self.choose_move()                ... try another direction.
    self.move_by(move)                         Make our move.
```

Now we need to decide how to choose the next grid point to move towards. If we allow the `Ghost` to reverse its direction too easily, it usually won't get very far. For this reason, we'll check all the directions we could move in which wouldn't cause us to switch into reverse, and then choose one at random. Only as a last resort do we consider turning round. For this we define a method called `can_move_by` which returns true if the given move is possible.

```
class Ghost(Movable):                          This should already be in your program.
  # ...                                        Other definitions will be here.

  def choose_move(self):
    (move_x, move_y) = self.movement           The direction we were going in.
    (nearest_x, nearest_y) = (
        self.nearest_grid_point() )
    possible_moves = []

    if move_x >= 0 and self.can_move_by((1, 0)): Can we move right?
      possible_moves.append((1, 0))

    if move_x <= 0 and self.can_move_by((-1, 0)): Can we move left?
      possible_moves.append((-1, 0))

    if move_y >= 0 and self.can_move_by((0, 1)): Can we move up?
      possible_moves.append((0, 1))

    if move_y <= 0 and self.can_move_by((0, -1)): can we move down?
      possible_moves.append((0, -1))

    if len(possible_moves) != 0:               Is there anywhere to go?
      move = random_choice(possible_moves)     Pick a direction at random.
      (move_x, move_y) = move
    else:
      move_x = -move_x                         Turn round as a last resort.
      move_y = -move_y
      move = (move_x, move_y)

    (x, y) = self.place
    self.next_point = (x+move_x, y+move_y)     Set the next point

    self.movement = move                       Store this move for next time.
    return self.furthest_move(move)            Return the move.

  def can_move_by(self, move):
    move = self.furthest_move(move)            How far can we move in this direction?
    return move != (0, 0)                      Can we actually go anywhere?
```

We now need to define the `move_by` method which is called by the `move` method. This just takes a movement vector as a parameter, and moves the `Ghost` that far.

```
class Ghost(Movable):                                      This should already be in your program.
  # ...                                                    Other definitions will be here.

  def move_by(self, move):
    (old_x, old_y) = self.place                            Get the old coordinates
    (move_x, move_y) = move                                Unpack the vector

    (new_x, new_y) = (old_x+move_x, old_y+move_y)  Get the new coordinates
    self.place = (new_x, new_y)                            Update the coordinates

    screen_move = (move_x * grid_size,
                   move_y * grid_size)
    move_by(self.body, screen_move)                        Move the body on the screen.
```

You can now play the game! OK, so the computer doesn't notice if you hit a Ghost and there are no Capsules yet, but you can still have fun trying to avoid the ghosts.

Collision detection.

We now need to find out when `Pacman` bumps into a `Ghost`. The way we will do this is for `Pacman` to tell the `Maze` where he is and for the `Maze` to pass this on to all the `Movables`. It is then up to the object to do the checking. If `Pacman` stands still, we still need to do the checking, so we need to do it in the `move` method.

```
class Pacman(Movable):                                     This should already be in your program.
  # ...                                                    Other definitions will be here.

  def move(self):
    keys = keys_pressed()                                  Check the keys as before.
    if   'z' in keys: self.move_left()
    elif 'x' in keys: self.move_right()
    elif ';' in keys: self.move_up()
    elif '.' in keys: self.move_down()
    self.maze.pacman_is(self, self.place)                  Tell the Maze where we are.
```

The `Maze` simply passes this on to each `Movable`.

```
class Maze:                                                This should already be in your program.
  # ...                                                    Other definitions will be here.

  def pacman_is(self, pacman, point):
    for movable in self.movables:                          Go through the Movables.
      movable.pacman_is(pacman, point)                     Pass the message on to each.
```

Because `Pacman` is a `Movable`, he will also get the message. We don't want to know if `Pacman` collides with himself (whatever that means), so we just ignore that.

```
class Pacman(Movable):                                     This should already be in your program.
  # ...                                                    Other definitions will be here.

  def pacman_is(self, pacman, point):
    pass                                                   Pacman knows where Pacman is.
```

We do want to know when `Pacman` bumps into a `Ghost` though. We'll write a simple but adequate detection method. We'll say that `Pacman` and the `Ghost` have collided when their centres are less than 1.6 grid units apart. Since `Pacman` is 0.8 grid units from his middle to his outer, and so is the `Ghost`, if they are placed side by side their centre will be 1.6 grid units apart. `Pacman` can sometime overlap the `Ghost` a little before it triggers, but we could put that down to a lucky escape. We'll invent a method called `bump_into` which we'll call when the `Ghost` bumps into `Pacman`.

Before writing the method, we need to do a little maths. The question is: how do we know when the two centre points are 1.6 grid units apart? This was answered by a man called Pythagoras a few thousand years ago. He said that if the distance between two points horizontally is `X`, the distance vertically is `Y`, and the distance between the points is `D` then `X*X + Y*Y = D*D` (or words to that effect in ancient Greek).

If the distance `D` is less than 1.6 grid units then `D*D` will be less than `1.6*1.6` grid units squared, but according to Pythagoras `D*D` is the same as `X*X + Y*Y` which we can work out.

```
class Ghost(Movable):                         This should already be in your program.
  # ...                                       Other definitions will be here.

  def pacman_is(self, pacman, point):
    (my_x, my_y) = self.place
    (his_x, his_y) = point
    X = my_x - his_x
    Y = my_y - his_y
    DxD = X*X + Y*Y
    limit = 1.6*1.6
    if DxD < limit:
      self.bump_into(pacman)
```

Now we need to decide what happens when the `Ghost` and `Pacman` collide. For the moment, we just want `Pacman` to lose, so we'll call the `lose` method on the `Maze`.

```
class Maze:                                   This should already be in your program.
  # ...                                       Other definitions will be here.

  def lose(self):
    print "You lose!"
    self.game_over = 1

# ...                                         Other classes will be here

class Ghost(Movable):                         This should already be in your program.
  # ...                                       Other definitions will be here.

  def bump_into(self, pacman):
    self.maze.lose()
```

Try the new program out. Try out bumping into Ghosts, to check that the collision detection works.

Capsules

The next thing we're going to do is add the capsules which allow Pacman to capture the ghosts. These are going to be objects which sit in the `Maze` until `Pacman` eats them, just like the `Food`. The first thing we'll do is change the `make_object` method in the `Maze` class to recognise the letter "o" in the layout and create a Capsule object when it finds one.

```
class Maze:                                   This should already be in your program.
  # ...                                       Other definitions will be here.

  def make_object(self, point, character):    You should also have this.
    # ...                                     Checks for other characters go here as before.
    elif character == 'o':                    Is it a Capsule?
      self.map[y][x] = Capsule(self, point)   Put a new Capsule in the map.
```

Now we'll need to write the `Capsule` class. To start with this will look a bit like the `Food` class. First we need to tell the computer how to initialize a `Capsule` object.

```
class Capsule(Immovable):
  def __init__(self, maze, point):
    self.place = point
    self.screen_point = maze.to_screen(point)
    self.maze = maze
    self.draw()
```

Now we need to describe how to draw it on the screen. For this we'll just draw a circle on the screen.

```
capsule_colour = Colour.white              Put these at the top of your program.
capsule_size = grid_size * 0.3             How big to make the capsules.

class Capsule(Immovable):                  This should already be in your program.
  # ...                                    Other definitions will be here.

  def draw(self):
    (screen_x, screen_y) = self.screen_point
    self.dot = circle(screen_x, screen_y,
                      capsule_size,
                      colour = capsule_colour,
                      filled = 1)
```

If you try the game now, you'll see the capsules on the screen, but `Pacman` just ignores them when he walks over them. What's happening here? `Pacman` is calling the `eat` method on the `Capsule` and Python is finding the `eat` method in the `Immovable` class, so nothing happens. To make `Pacman` eat the `Capsule`, we need to write an `eat` method in the `Capsule` class telling the computer how `Pacman` should eat capsules.

```
class Capsule(Immovable):                  This should already be in your program.
  # ...                                    Other definitions will be here.

  def eat(self, pacman):
    remove_from_screen(self.dot)           Remove the dot from the screen.
    self.maze.remove_capsule(self.place)   Tell the Maze to scare the ghosts.
```

The `eat` method works in a similar way to the `eat` method in `Food` class. The difference is that instead of calling the `remove_food` method on the `Maze` object, we call the `remove_capsule` method. This method will need to remove the `Capsule` from the map and tell all the ghosts to turn white.

```
class Maze:                                This should already be in your program.
  # ...                                    Other definitions will be here.

  def remove_capsule(self, place):
    (x, y) = place
    self.map[y][x] = Nothing()             Make the map entry empty.
    for movable in self.movables:          Tell all the Movables that a capsule
      movable.capsule_eaten()              has been eaten.
```

We don't have a list of all the `Ghost` objects anywhere, but we do have a list of all the `Movable` objects. We simply tell all the movables that a capsule has been eaten. Only some of the movables (the ghosts) will want to know, so we'll write a default method in the `Movable` class to do nothing. Anything that doesn't want to know about capsules being eaten (Pacman) won't need to do anything special.

```
class Movable:                             This should already be in your program.
  # ...                                    Other definitions will be here.

  def capsule_eaten(self):                 Called when a Capsule has been eaten.
    pass                                   Normally, do nothing.
```

You may think that we could have written this in the `Pacman` class, and that is true. That would mean that if we invented another type of `Movable` object, we would have to write a `capsule_eaten` method for it even if it didn't want to know capsules being eaten. Now, we need to override the `capsule_eaten` method in the `Ghost` class. This is going to make the `Ghost` change colour for a while. We can do this by changing the `colour` variable.

After a while the ghost will need to change back to its original colour, so we will need to know two things. We will need to know how long we have left before the original colour returns, and also what the original colour was. We will store these two values in variables called `time_left` and `original_colour`. When the ghost isn't scared, we will store the value `0` in `time_left`, so that we know. We need to set the values of both of these variables in the initializer. The `change_colour` method will remove the old body from the screen, set the `colour` variable and draw a new body. We'll invent a `redraw` method to remove the old body.

```
                                              Put these at the top of your program.
scared_colour = Colour.white                  The colour the ghosts turn when
                                              Pacman eats a capsule.
scared_time = 300                             How long Ghosts stay scared.

class Ghost(Movable):                         This should already be in your program.
  def __init__(self, maze, start):            As should this.
    global ghost_colours                      As before.

    self.next_point = start
    self.movement = (0, 0)

    self.colour = ghost_colours[0]
    ghost_colours[:1] = []
    ghost_colours.append(self.colour)

    self.original_colour = self.colour        Store the original colour.
    self.time_left = 0                        We're not scared yet.

    Movable.__init__(self, maze,              As before.
                     start, ghost_speed)

  # ...                                       Other definitions will be here.

  def capsule_eaten(self):
    self.change_colour(scared_colour)         Change to the scared colour.
    self.time_left = scared_time

  def change_colour(self, new_colour):
    self.colour = new_colour                  Change the colour.
    self.redraw()                             Recreate the body.
```

Here, we've invented a `change_colour` method to change the ghost's colour. By setting the `time_left` variable, we'll remember that we're scared and that we should change the colour back later. We now need to define the `redraw` method.

```
class Ghost(Movable):                         This should already be in your program.
  # ...                                       Other definitions will be here.

  def redraw(self):
    old_body = self.body
    self.draw()
    remove_from_screen(old_body)
```

As we did in the `move_by` method in the `Pacman` class, we first draw a new body and then remove the old body to avoid catching a glimpse of the background.

At this stage we have managed to change the colours of the Ghosts. We now need to make them change back again after a while. Just before they are about to change to their original colour, they need to flicker between white and their original colour. We will update the `time_left` variable within the `move` method of the `Ghost` class.

```
warning_time = 50                           Put this at the top of your program.

# ...                                       Other classes will be here.

class Ghost(Movable):                       This should already be in your program.
  # ...                                     Other definitions will be here.

  def move(self):
    (current_x, current_y) = self.place     As before.
    (next_x, next_y) = self.next_point
    move = (next_x - current_x,
            next_y - current_y)
    move = self.furthest_move(move)
    if move == (0, 0):
      move = self.choose_move()
    self.move_by(move)
    if self.time_left > 0:                  Are we scared?
      self.update_scared()                  Update the time and colour.

  def update_scared(self):
    self.time_left = self.time_left - 1     Decrease the time left.
    time_left = self.time_left
    if time_left < warning_time:            Are we flashing?
      if time_left % 2 == 0:                Is time_left even?
        colour = self.original_colour       Return to our old colour.
      else:
        colour = scared_colour              Go to the scared colour.
      self.change_colour(colour)            Actually change colour.
```

If `time_left` is zero, we do nothing special. Otherwise, every time `move` gets called we subtract one from `time_left`. When `time_left` drops below `warning` time, we start flashing the ghost. When this is happening, we change the colour of the ghost on every call to `move`. When `time_left` is even, we return to our original colour, and when it's odd we change to the scared colour. A new thing here is the use of `n % 2` to check if a number is even. The `%` means "the remainder when divided by". All even numbers have no remainder when divided by 2 and all odd numbers have a remainder of 1 when divided by 2. When `time_left` reaches zero, the ghost will return to its original colour because zero is even.

The last thing we need to do is to allow Pacman to capture the ghost when `time_left` is zero. In other words we need to change what happens when we `bump_into` Pacman.

```
class Ghost(Movable):                       This should already be in your program.
  # ...                                     Other definitions will be here.

  def bump_into(self, pacman):              This should also be in your program.
    if self.time_left != 0:                 Are we scared?
      self.captured(pacman)                 We've been captured.
    else:
      self.maze.lose()                      Otherwise we lose as before.
```

We have invented a method called `captured` which we call when we're scared and we bump into Pacman. This should send the `Ghost` back to its starting position. If follows that we need to know what the starting position is.

```
class Movable:                              This should already be in your program.
  def __init__(self, maze, point, speed):
    self.maze = maze                        As before.
    self.place = point
    self.speed = speed
    self.start = point                      Our starting position

  # ...                                     Other definitions will be here.
```

We're now ready to write the `captured` method.

```
class Ghost(Movable):                       This should already be in your program.
  # ...                                     Other definitions will be here.

  def captured(self, pacman):
    self.place = self.start                 Return to our original place...
    self.colour = self.original_colour      ... and colour.
    self.time_left = 0                      We're not scared.
    self.redraw()                           Update the screen.
```

The basic Pacman game is now finished! Try it out and see if you can find any problems with it.

Further improvements

Now we've got a working game, there are improvement which can be made to it.

- Make Pacman look nicer.
- Make the Ghosts look nicer.
- Keep track of the score.
- Keep the time left for the level.
- Allow Pacman 3 lives.
- Do something more interesting when Pacman wins or loses.
- Add more levels.
- Make the Ghosts smarter.