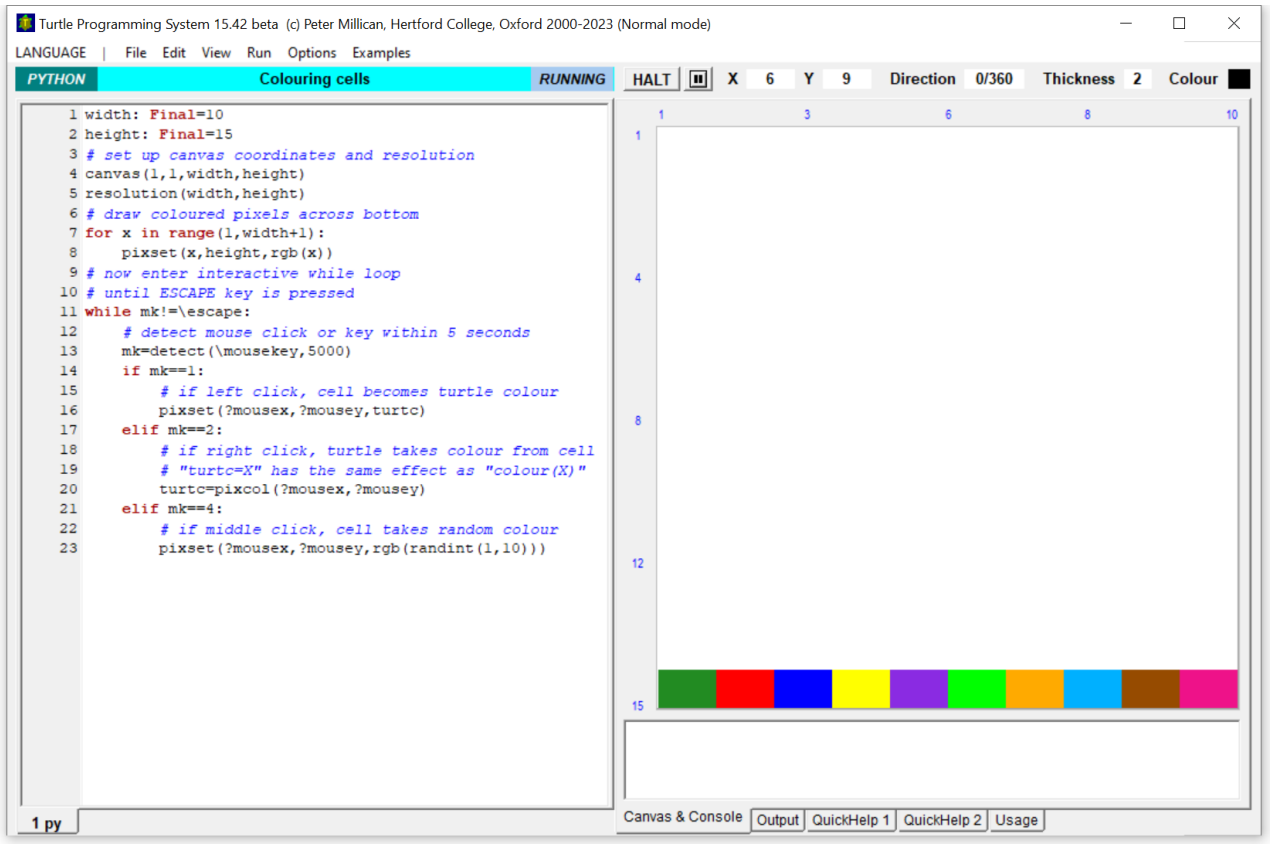
Turtle Python 5 – Cellular Models

Cellular models provide a powerful and relatively straightforward way of modelling many different phenomena, from crystal growth to biological patterning, and from the spread of disease to social interaction. They can also be fun and even quite exciting! *The Turtle System* has a number of features that make these models relatively easy to program, and if you do not know these already, they can be quickly introduced by starting with a relatively simple program which is not itself a model.

1. Optional Introduction: The "Colouring Cells" Program

Go to Examples menu 5 – which covers "User input, interaction and games", and select the "Colouring cells" program. To see what it does, click on "RUN", and you will see that 10 differently coloured rectangles appear along the bottom of the Canvas:



Each of these blocks is in fact a single coloured *pixel*, and they are coloured with the first ten "native" *Turtle* colours (as displayed on the relevant tab under "QuickHelp 1" at the bottom of the screen). Now try clicking with the *left* mouse button on some of the white area of the Canvas – each time you do, the relevant pixel will turn black, and this will enable you to see that the entire Canvas has been divided into a grid of 10x15 pixels (with coordinates running from 1 to 10 along the top, and 1 to 15 down the side, as shown). Then try clicking with the *right* mouse button on one of the coloured pixels at the bottom – this time, you will see from the "Colour" patch (at the right above the Canvas) that the *Turtle*'s colour has changed accordingly. And if you now *left-click* elsewhere on the Canvas, you'll see that the relevant pixel now turns to the new colour (rather than black), and you can go on in this way making a coloured pattern on the Canvas. Finally, try clicking with the *middle* mouse button on a number of pixels. Now, you will find that the pixel turns into one of the first 10 native colours, chosen randomly. When you've played with this enough to understand what's happening, press the "Escape" key to finish the program.

While not in itself a particularly useful *program*, this introduces a wide range of very useful techniques, in only 23 lines of code (but only 14 if comments are removed). You may have come across a number of these techniques before, in the "Animation and User Input" document.

```
width: Final=10
height: Final=15
# set up canvas coordinates and resolution
canvas(1,1,width,height)
resolution(width,height)
# draw coloured pixels across bottom
for x in range(1,width+1):
    pixset(x,height,rgb(x))
# now enter interactive while loop
# until ESCAPE key is pressed
while mk!=\escape:
    # detect mouse click or key within 5 seconds
    mk=detect(\mousekey,5000)
    if mk==1:
        # if left click, cell becomes turtle colour
        pixset(?mousex,?mousey,turtc)
    elif mk==2:
        # if right click, turtle takes colour from cell
        # "turtc=X" has the same effect as "colour(X)"
        turtc=pixcol(?mousex,?mousey)
    elif mk==4:
        # if middle click, cell takes random colour
        pixset(?mousex,?mousey,rgb(randint(1,10)))
```

Let's now take this step by step, explaining each of the program's distinctive features in turn:

```
width: Final=10
height: Final=15
```

- These two lines define *constants* named "width" and "height" using the `Final` keyword. Specifying these values at the beginning of the program makes it easy to change the number of pixels into which the Canvas will be divided. They could be specified as ordinary variables (e.g. "width = 10"), but using `Final` enables *Turtle Python* to handle them more efficiently and to declare an error if an attempt is made to change them. *Note that any such "Final" constant definitions have to come at the very beginning of the program, before any variables have been defined*.

```
canvas(1,1,width,height)
```

- This defines the Canvas coordinates as starting from the point (1,1) at the top left, with the given width and height. Note, however, that it would be more standard, when creating a cellular model, to use `canvas(0,0,width,height)`, so that the x-coordinates run from 0 to width-1, and the y-coordinates from 0 to height-1. (This makes it much easier if the model is intended to "wrap around" from bottom to top and right to left.) We'll generally do that in future..

```
resolution(width,height)
```

- This defines the Canvas *resolution* – the number of pixels actually used in the Canvas image – as *width* x *height*. In cellular models, we almost always want the *resolution* dimensions to match the *coordinate* dimensions, as indeed they do in this case.

```
for x in range(1,width+1):
    pixset(x,height,rgb(x))
```

- A FOR loop over `range(width)` – which is shorthand for `range(0,width)` – would count from 0 to width-1 inclusive. This loop instead counts from 1 to width, in keeping with the non-standard range of x-coordinates that were defined in the `canvas` instruction above.
- The command `pixset(x,y,red)` would set the individual pixel at coordinates (*x*, *y*) to the specified colour – in this case, *red*. *But note here that "red" is actually shorthand for a particular colour code number, in fact for 16711680 which is 0xFF0000 in hexadecimal*.
- The function `rgb(n)` returns the n^{th} native *Turtle* colour code: so, for example, `rgb(1)` returns the colour code for green, `rgb(2)` for red, `rgb(3)` for blue, `rgb(4)` for yellow, and so on. Thus the short loop colours the ten bottom pixels with the first ten native colours..

```
while mk!=\escape:
```

- The program now enters a `while` loop (the body of which is indented), and this will continue until the *ESCAPE* key is pressed. Initially, the variable `mk` will be equal to 0 (the default value), but if it becomes equal to 27 – which is the value of the *keycode* `\escape` – then the while condition `mk!=\escape` will become false and the loop will terminate.

```
mk:=detect(\mousekey,5000);
```

- Tells the system to wait for up to 5 seconds (i.e. 5000 milliseconds) to detect either a mouse click or a keypress. If no such event is detected, the variable `mk` will be made equal to 0. If a mouseclick is detected, however, then `mk` will be made equal to 1, 2, or 4 depending on whether it was the left, right or middle mouse button. And if a keypress is detected first, then `mk` will be made equal to a *keycode* value (the only relevant one here is `\escape`, which happens to be equal to 27). The value 5000 here is purely illustrative, and the program would work just as well with the instruction `mk:=detect(\mousekey,0)`, which waits indefinitely for either a mouse click or a keypress.

```
if mk==1:
    pixset(?mousex,?mousey,turtc)
```

- If `mk` is made equal to 1 by the `detect(\mousekey)` function, this means the left mouse button has been clicked. Then `pixset(?mousex,?mousey,turtc)` sets the pixel at the coordinates of that mouse click to the current *Turtle* colour.

```
elif mk==2:
    turtc=pixcol(?mousex,?mousey)
```

- If `mk` is made equal to 2 by the `detect(\mousekey)` function, this means the right mouse button has been clicked. Then `pixcol(?mousex,?mousey)` picks up the colour of the pixel at the coordinates of that mouse click, and `turtc` – the *Turtle* colour attribute – is set equal to that colour.

```
elif mk==4:
    pixset(?mousex,?mousey,rgb(randint(1,10)))
```

- If `mk` is made equal to 4 by the `detect(\mousekey)` function, this means the middle mouse button has been clicked. Then `randint(1,10)` generates a random integer between 1 and 10 inclusive, the `rgb` function – as we saw earlier – converts this integer into the colour code of the corresponding native colour, and the `pixset` instruction then uses that to colour the pixel on which the middle mouse click occurred. So that pixel takes on a random native colour.

2. A Cellular Model of an Epidemic "Tipping Point"

Here we'll be looking at *cellular models* that are relatively easy to implement, partly because – in contrast with some *cellular automata* discussed in a later document – they are *asynchronous* and *randomised*, with individual cells being processed in turn rather than entire neighbourhoods. As a first example, load and run the "Tipping point" program (from Examples menu 7 – "Cellular models"). This illustrates the phenomenon discussed in Malcolm Gladwell's 2000 book *The Tipping Point*, pp. 260-1:

> "The best way to understand the Tipping Point is to imagine a hypothetical outbreak of the flu. Suppose, for example, than one summer 1,000 tourists come to Manhattan from Canada carrying an untreatable strain of twenty-four-hour virus. This strain of flu has a 2 percent infection rate, which is to say that one out of every 50 people who come into close contact with someone carrying it catches the bug himself. Let's say that 50 is also exactly the number of people the average Manhattanite – in the course of riding the subways and mingling with colleagues at work – comes into contact with every day. What we have, then is a disease in equilibrium. … With those getting sick and those getting well so perfectly in balance, the flu chugs along at a steady but unspectacular clip through the rest of the summer and the fall. But then comes the Christmas season. The subways and buses get more crowded with tourists and shoppers, and instead of running into an even 50 people a day, the average Manhattanite now has close contact with, say, 55 people a day. All of a sudden, the equilibrium is disrupted. … That moment when the average flu carrier went from running into 50 people a day to running into 55 people was the Tipping Point … at which an ordinary and stable phenomenon … turned into a public health crisis."

The program starts by defining various constants:

(a) Canvas dimensions *width* and *height* (both 100 in this case).
(b) Colours to indicate cells that are *susceptible* (*lightgreen*), *infectious* (*red*), and *recovered* (*blue*).
(c) An initial number of infected individuals (10 by default).
(d) The probability of a contact being infected: *infectprob* (to be interpreted as a percentage), and the number of contacts that any infectious individual will have: *contacts*. By default, these are set to 2% and 50 respectively, just as in Gladwell's example.

The point of defining these constants (rather than using numbers directly in the code) is to make them easy to change, so we can experiment with different values, in particular to see how radically the behaviour of the epidemic changes as the number of contacts rises above the "tipping point" of 50.

The program then defines a few variables, notably *numinfected* and *numinfectious*, which will keep track respectively of the total number of individuals who have been *infected* overall, and the number who are currently *infectious*. To make it easy to ensure that these keep in step with the displayed infections, the *infect(x,y)* procedure is defined so as to increment both variables at the same time as a particular pixel changes colour to *infectious*.

When the program runs, random susceptible individuals (i.e. pixels) are selected to be infected until the chosen initial number (as in (c) above) has been reached. Then the program main loop randomly selects an individual, and if they are infectious, randomly selects in turn the appropriate number of contacts (e.g. 50 as in (d) above) and then, with an appropriate probability (e.g. 2% as in (d) above), infects them if (but only if) they are susceptible. Having gone through all of the contacts, the infected individual now recovers and becomes *recovered* (changing colour to blue). Eventually, the epidemic will subside as more and more individuals become *recovered* (and thus immune). When nobody is left infectious, the program terminates and reports how many individuals have been infected overall.

To appreciate Gladwell's message about "tipping points", try running the program with increased values of *infectprob* and/or *contacts*. You will see that a relatively small change in either of them can bring about a very significant change in the impact of the epidemic.

3. Modelling the Spread of Disease, and Its Prevention

The "Tipping point" program takes no account of the geography of the landscape on which the epidemic is playing out – it just assumes that within the city, everyone is mixing randomly. But now we'll turn to look at a more sophisticated model, in the "Spread of disease" example program (again from menu 7). This is one example of an implementation of the well-known "SIR" (Susceptible, Infected, Recovered) model of the spread of infectious disease, and conveys some very important practical lessons about disease prevention. Like the "Tipping point" program, this starts by defining various constants:

(a) Canvas dimensions *width* and *height* (again both 100).
(b) Colours to indicate cells that are *susceptible* (*lightgreen*), *infected* (*red*), and *recovered* (*blue*).
(c) An integer *startradius* (10) that defines the maximum boundary of the initial infection.
(d) Three probabilities, each of which is to be interpreted as a percentage: *infectprob* (1%), the probability that a cell within *startradius* will be initially infected; *immuneprob* (2%), the probability that a cell will be immune throughout (e.g. due to prior vaccination); and *recoverprob* (15%), the probability that an infected cell will recover in any time period.

As before, the point of defining these constants is to make them very easy to change. The whole point of this program, indeed, is to see how the behaviour then changes.

Following the constants and a few variables – including *numinfected*, which keeps count of infected cells – there is again a simple procedure *infect(x,y)*, which colours cell (*x*,*y*) with the *infected* colour (i.e. *red*) and increments *numinfected* accordingly. Then the main program begins, defining the canvas dimensions and resolution (just as we saw above), initialising *numinfected* to zero, and colouring the cells of the canvas according to the following rules:

- If a cell's distance from the centre of the canvas is less than or equal to *startradius* (10), then with probability *infectprob* (1%), it will be *infected* (*red*) from the start.
- Otherwise, with probability *immuneprob* (2%), the cell will be *recovered* (*blue*) from the start – this colour is given to cells that are immune, mostly after recovery from the infection.
- Otherwise, the cell will be *susceptible* (*lightgreen*).

The canvas is temporarily frozen – using `noupdate()` ... `update()` – while this colouring is taking place, so it can be completed much more quickly.

After this initialisation, the spread and eventual decline of the infection are modelled with a loop that continues until *numinfected* becomes zero. The loop starts by choosing a random value of *x* between 0 and (*width*-1) and a random value of *y* between 0 and (*height*-1). Then we check the colour of cell (*x*,*y*) to see whether it is *infected* (i.e. *red*) or not. If it is, then with probability *recoverprob* (15%), we change it to *recovered* (i.e. *blue*), to indicate that it has now recovered and thus become immune from further infection. (To do this with the correct probability, we use the conditional `if randrange(100)<recoverprob` to select a random number between 0 and 99 and check whether it is less than *recoverprob*.) Finally, if the cell is infected and has *not* recovered, then the following code is executed:

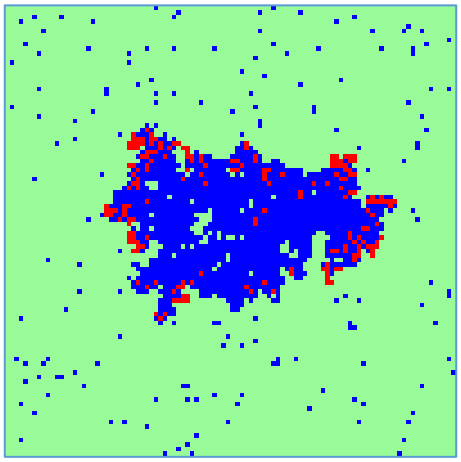
```
n = randrange(4)*2+1
x = x + n//3 - 1
y = y + n%3 - 1
if pixcol(x,y)==susceptible:
    infect(x,y)
```

The purpose of this code is to select at random one of the four closest neighbours of the infected cell and then, if that cell is *susceptible* (i.e. *lightgreen*), to infect it – this is how the infection spreads. The arithmetic here is neat but a bit tricky, going through the following steps and using the operators for integer division (often called *div*, but "//" in Python) and remainder (often called *mod*, but "%" in Python):

randrange(4)	randrange(4)*2	n	n // 3	n // 3 – 1	n % 3	n % 3 – 1
0	0	1	0	-1	1	0
1	2	3	1	0	0	-1
2	4	5	1	0	2	1
3	6	7	2	1	1	0

The shaded columns show the four possible random numbers (between 0 and 3), and the corresponding values that get added to *x* and *y* respectively. Adding (-1,0) corresponds to a move left on the canvas, (0,-1) to a move up, (0,1) to a move down, and (1,0) to a move right. So after these additions, the coordinates (*x,y*) do indeed identify one of the four neighbouring cells. Now it just remains to test whether that cell is *susceptible* (*lightgreen*), and if it is, to infect it (*red*).

> *You might have noticed that moving in these ways from a cell on the edge of the grid could take us to a pixel off the canvas. One convenient feature of pixset and pixcol – unlike corresponding operations with lists – is that they do not throw up an error message if this happens, and since it causes no trouble for the operation of our program, we are able to ignore this complication here. But can you find a way in which changing the colours used could introduce a problem?*



One practical virtue of this model of infection – which in its more sophisticated forms is highly influential and widely used – is to demonstrate very clearly the value of vaccination. If the program is run as it stands, the infection is very likely to spread from the centre of the canvas to most of the susceptible cells (the image here shows it spreading aggressively in several directions). But if the value of *immuneprob* is set higher – for example, changed from 2% to 12% – then you will find that the infection has far less impact, often dying out quickly and usually reaching only a small proportion of the canvas. Thus artificially vaccinating even 10% of the population can potentially bring a huge payoff in disease control for the population as a whole. Real diseases, of course, will vary in infectivity and other characteristics, so we cannot assume that this conclusion will apply to them. But this model does allow for variation, and enables us to explore how the critical value of *immuneprob* at which the disease can be tamed depends on the probability of recovery in each time period: if recovery typically takes a long time (because *recoverprob* is low), then more widespread prior immunity will be required to keep the disease in check. The crucial point here is that the longer recovery takes for any individual, the more opportunity the disease has to infect that individual's neighbours, and so the higher the probability that it will indeed be passed on to them.

This particular population structure – which assumes that every individual is statically located within a fixed grid, with precisely four neighbours each – is of course very crude, but more complex versions of the "SIR" model play a vitally important role in the real world, helping *epidemiologists* (those who study such things) to understand, predict and combat the spread of diseases. In the mathematics of infection, the most important parameter for any disease is its "basic reproduction number" (commonly denoted R_0) – that is, the number of individuals that could be expected, on average, to be directly infected from one newly-infected individual within a totally susceptible population. In our model (with *recoverprob* at 15%), this parameter is around 2.35 for an individual surrounded by four "susceptibles", but of course any individual thus infected will then have only three adjacent susceptibles (since the individual who infected them is no longer susceptible), and their expected direct infectivity drops accordingly, to around 1.76.

An interesting experiment is to see what happens if individuals are allowed to wander (e.g. perhaps by incorporating some sort of "diffusion", or long-range swaps – as a crude model for air travel), in which case the spread of disease is likely to be significantly greater. The "Spread of disease" program actually allows for this, by providing a constant, *movement*, which is set to `False` by default, but can be edited to `True` if you want to see the effect. Another constant, *report*, can be made `True` if you want to see updates on the progress of the epidemic.

4. Schelling's Segregation Model

Now we go on to examine in detail another built-in program from the *Turtle System* (again under Examples menu 7, "Cellular models"), but this time one of social rather than medical relevance. This implements a model of social segregation due to the Nobel prizewinning economist Thomas Schelling, which is famous for showing how relative small local preferences can lead to global segregation. If you run the program and then quickly click on "HALT" (before 2 seconds have elapsed), you will see the initial situation in which we have a "map" of a square city surrounded by a rim of *green*. Within the city, most of the cells are either *blue* or *red* – in a strictly alternating pattern – but roughly 4% (1 in 25) are *green*. The *blue* and *red* cells represent dwellings that are occupied by members of two different religious communities, the Blues and the Reds, while the *green* cells are unoccupied. Now run the program again, and see what happens, as the (initially very few) Blues and Reds who find that they are "unhappy" with their neighbourhood move around. The precise outcome depends on chance, but almost invariably, you will end up with a pattern in which the Blues and the Reds are very largely segregated rather than interspersed. You might naturally think this indicates that they are strongly biased against the other group, but in fact this result comes about even when the bias is relatively mild.

1	2	3
8		4
7	6	5

In the "Spread of disease" program, we effectively treated each cell as having at most four immediate *neighbours* that it could infect. But with this program, we consider each cell as having eight *neighbour* cells, as shown on the right here. To assess any individual's "happiness", we simply count the number of neighbours that have the same religion (variable *like*) and the number of neighbours that have the opposite religion (variable *unlike*), and check whether it is true that (*like* ≥ *unlike*–1). If it is true, then the individual is "happy"; if false, "unhappy". In other words, an individual becomes unhappy when the neighbours from the *unlike* religion outnumber the neighbours from the *like* religion by 2 or more. So, for example, a Red individual with 3 Red neighbours will be happy as long as they have a maximum of 4 Blue neighbours; but with 3 Red and 5 Blue neighbours, they would become unhappy. Any individual assessed as unhappy will immediately try to find an *empty* (i.e. *green*) cell where they would be happy, and on finding one, will move there (and the cell from which they move will become *empty*). This one simple rule governs the entire dynamics of the model! Let us now go through the program, line by line:

```
width = 50
height = 50
empty = green
```

As is usual with these models, the *width* and *height* of the grid are set at the beginning, to make the code easily adjustable, and the *empty* colour is set as *green*, which makes that easily changeable also (e.g. to *midgrey*) if desired. As we saw in §1 above, we could if we wished use the `Final` keyword to ensure that these definitively stay unchanged throughout the program, but that isn't essential, so here we go for the more standard way of doing things. (Another common convention is to use capital letters for constants.)

Next, we come to a function definition, whose purpose is to calculate whether or not an individual of a given colour *c* – whether red or blue – would be "happy" in the cell with coordinates (x,y). It starts like this:

```
def happy(x,y,c):
```

```
    like=0
    unlike=0
```

These first two lines following the function header initialise the count of *like* and *unlike* neighbours for the "current" cell at location (*x*, *y*). Both counts, obviously, start at 0.

```
    for i in range(-1,2):
        for j in range(-1,2):
            if (i!=0) or (j!=0):
```

Variables *i* and *j* are used, respectively, for the *x*-offset and *y*-offset of each neighbour from the current cell (*x*, *y*), so the neighbour in question will be cell (*x*+*i*, *y*+*j*). These three lines ensure that we count through all eight neighbours of the current cell, making each of *i* and *j* equal to -1, 0 and 1 in turn, but ignoring the case in which both *i* and *j* are 0 (since this would be the current cell itself).

```
                neighbour=pixcol(x+i,y+j)
                if neighbour!=empty:
                    if neighbour==c:
                        like+=1
                    else:
                        unlike+=1
```

These are the commands performed in turn for each of the eight neighbouring cells. First, *neighbour* is made equal to the relevant pixel colour. If this is *empty* (i.e. the relevant neighbouring cell is a green space), nothing further happens. Otherwise, *neighbour* must be either *blue* or *red*, so either *like* or *unlike* should be incremented, depending on whether it matches colour *c* (which is the colour of the individual whose happiness – or potential happiness – we are trying to assess). The upshot of all this is that after having gone through this code eight times (once for each neighbouring cell), *like* will record the number of neighbouring cells of colour *c*, and *unlike* will record the number of neighbouring cells of the "opposite" colour.

```
    return (like>=unlike-1)
```

This line specifies the *result* of the function (i.e. *true* if (*like* ≥ *unlike*–1) and *false* otherwise), and then ends the function definition. Our function *happy* has now been defined, and can be used in the remainder of the program, to which we now turn.

```
canvas(-1,-1,width+2,height+2)
resolution(width+2,height+2)
```

These specify the Canvas size (i.e. coordinate range) and resolution (i.e. number of pixels). Here the coordinates will range from -1 to *width* in the *x* direction (with *width*+2 horizontal pixels) and -1 to *height* in the *y* direction (with *height*+2 vertical pixels). So for example if *width* has been set to 50, there will be 52 horizontal pixels, and each of these will correspond to a distinct *x*-coordinate, from -1 to 50 inclusive. (Likewise for *height* and the *y*-coordinates.) The reason for the "+2" is to allow for an *empty* (*green*) border, one cell wide, around the 50x50 city map, which saves us from worrying about the edge cells within the city.

```
noupdate()
blank(empty)
for i in range(width):
    for j in range(height):
```

The `noupdate()` command prevents the Canvas display from updating until the initial disposition has been completed, so as to make the start of the program neater and quicker. Then `blank(empty)` blanks the entire Canvas to the colour *empty* (i.e. *green*), thus creating the green borders. The double loop that follows will count through every cell (*i*, *j*) within the city (*i* counting horizontally from *0* to *width*-1, and *j* counting vertically from 0 to *height*-1), setting the initial colour of each cell.

```
        if randrange(25)==0:
            pixset(i,j,empty)
```

```
            else:
                if (i+j)%2==0:
                    pixset(i,j,red)
                else:
                    pixset(i,j,blue)
    pause(2000)
```

The first command within the loop tests whether a randomly chosen number between 0 and 24 is equal to 0; if it is (i.e. with a 1-in-25 chance, or 4% probability), the pixel at location (*i*, *j*) is set to *empty* (i.e. *green* – which in fact it is already). Otherwise, the relevant pixel is set to *red* if (*i+j*) is an even number (i.e. its remainder on division by 2 is zero) or *blue* if it is an odd number. Hence the entire city is coloured in a "chequerboard" *red*/*blue* fashion, except for the 4% *green* cells. That done, the program *pauses* for two seconds (2000 milliseconds) to show this idyllic scene of (temporary) social integration!

```
while ?key!=\escape:
    noupdate()
```

The *while* loop here makes it possible to halt the program by pressing the *ESCAPE* key. The loop that follows will combine four tasks:

- Find a cell whose occupant is currently unhappy, and record their colour as *this*.
- Colour the relevant cell *empty* (i.e. *green*), to show that the occupant has left.
- Find a cell which is currently *empty*, and where an occupant of *this* colour would be happy.
- Colour the relevant cell *this* colour, to show that it is now occupied.

The loop starts with `noupdate()` – and will finish with `update()` – to ensure that the switching of colours is done smoothly, with both recolourings becoming visible at the same time.

```
        this=empty
        while (this==empty) or (happy(tryi,tryj,this)):
            tryi=randrange(width)
            tryj=randrange(height)
            this=pixcol(tryi,tryj)
        pixset(tryi,tryj,empty)
```

First *this* is set to *empty* as an initial value. Then a *while* loop is entered to try to find an unhappy individual within the grid. To achieve this, *tryi* is set to a random value between 0 and *width*-1, and *tryj* is set to a random value between 0 and *height*-1: these together mean that the coordinate pair (*tryi*, *tryj*) specifies a random cell within the city grid. Then *this* is set to the pixel colour of that cell, and this whole sequence is repeated until we have found a cell which is *not empty*, and whose occupant is *unhappy* at that position. Having found such a cell, it is made *empty*, and we move on to finding a new location for the previously unhappy occupant.

```
        while (pixcol(tryi,tryj)!=empty) or not(happy(tryi,tryj,this)):
            tryi=randrange(width)
            tryj=randrange(height)
        pixset(tryi,tryj,this)
```

Again we repeatedly find a random cell with coordinates (*tryi*, *tryj*) within the city grid, but this time the sequence is repeated until we have found a cell which is *empty*, and where the individual we are trying to move (whose colour is *this*) would be *happy*. Having found such a cell, it is coloured *this* to complete the individual's move from an unhappy to a happy location.

```
        update()
```

As remarked earlier, the last instruction in the main loop is `update()` to ensure that the recolourings (of the moved individual's old and new cells) take place at the same time. The program finishes when this main loop finishes (i.e. when the *ESCAPE* key is pressed), or the "HALT" button is clicked.

5. Some Ideas for Further Exploration

Many different variations are possible on the models we have considered here, and you can have a lot of fun – and also learn a lot about programming – by trying to find interesting ways to extend or modify them. To get you started, here are some suggestions.

5.1 Epidemic Programs

1. In the "Tipping point" program, there is no limit on the geographical distance between infectious individuals and their contacts – they are all assumed to be mixing randomly within a city's transport system. You might like to see what happens if you impose such a limit, or if – for example – you insist on the vast majority of an individual's contacts being within the same quadrant of the city.

2. The box on page 6 above asks whether changing the colours used in the "Spread of disease" program *could* introduce a problem. Can you discover what sort of problem this is hinting at?

3. Within either of these programs, you might want to try modifications inspired by the Covid epidemic – e.g. implementing partial "lockdowns", or travel bans, or tests for those who suspect that they may be infected followed by "self-isolation". You could also try adapting the programs so that you are able interactively to "infect" one of the cells by clicking on it, thus simulating the effect of someone infected arriving from abroad.

4. More ambitiously, you might like to consider developing a program that allows for virus mutation, with people who have recovered being only partially immune to new variants, depending on how much the virus has changed in the meantime. (For example, you might colour individuals with different shades of red, depending on the virus concerned.)

5.2 Schelling Segregation Model

5. Currently the program defines a "neighbourhood" as 8 cells: a 3x3 square omitting the central cell. Can you adapt it to use a neighbourhood of 24 cells (i.e. 5x5 omitting the central cell)? If you do this, what "happy" rule would give the best results?

6. The "happy" rule currently depends on the absolute difference between *like* and *unlike*, irrespective of the total number of neighbours. Can you make it depend instead on the relevant *ratios* (e.g. an individual might be happy if *unlike* is less than 150% of *like*, or equivalently, less than 60% of the total number of neighbours)?

7. Sometimes the program seems to stop even when there are still some unhappy individuals who could be made happy by moving (something you may see far more often if you change "randrange(25)" to "randrange(100)" at line 28). Can you see why this might be, and can you suggest a way of fixing it, either totally or partially?

8. Imagine that the city is being settled, from the start, by alternate Blue and Red individuals moving into random cells where they would be happy. How does this affect the dynamics?