An Introduction to Programming using Python

Mark Hepple

January 2014

1 Introduction

The purpose of this session is to introduce you to a set of basic programming concepts that are present in all programming languages. In particular we will be looking at **loops**, **functions**, **variables** and **decision making**. In isolation these concepts are all extremely simple – they could be explained to a 5 year old – but by putting them together we can produce computer programs of great sophistication and complexity.

In order to illustrate how complexity can arise from simplicity the lab class is going to use the example of **a robot painter that can move about on a board and change the colour of the board squares**. We will start out by just making the robot draw simple lines and shapes but by the end of the class we will see how we can simulate the seemingly intelligent behaviour of a rat searching for food in a maze.

The session will be using **a programming language called Python**. You may or may not have done some Python programming before. It is possible that you haven't done any programming before. If you haven't then don't panic! We will start out by taking small steps. The session is less about learning Python and more about learning to think *algorithmically* so I have tried to keep the amount of Python syntax to a minimum. However, **if you get stuck don't be shy about asking for help – just raise a hand**. Also, feel free to help each other and to discuss the problems amongst yourselves. Work in pairs at one machine if you like. Most importantly – experiment and have fun!

2 First steps

1. **Get a copy of the example programs.** Open the NetSurf web browser and search for my name ('mark hepple') on Google, to find my homepage. Under the Ambassador Scheme heading, follow the link to access a directory where the code is stored as a ZIP file. `Right-Click` on the file, and select `Save link`, and then choose `Save As`, and save the file to your top-level directory (`/home/pi`). Locate the file in a File Manager window, and extract it by right-clicking and selecting `Extract here`.

2. **Using IDLE.** You can open a Python interpreter shell by right-clicking on the desktop icon for `IDLE` (*not* the one for `IDLE 3`). Alternatively, if there's a particular Python file you want to edit, which is visible in a File Manager window, you can right-click on the file and select IDLE there.

3. **Write your first program!** Under the `Edit` menu of the Python shell window, select `New Window` to create a new edit window in which your can write your first Python program. The code of this window needs to be saved to a named file before you can run it. Python code file names always end in "`.py`". You might call this one `test1.py`, for example. Also, the file needs to be saved so that it's in the same directory as the code that you've downloaded. Select `Save As` under the `File` menu to proceed.

In the new editor window, carefully type in the following code:

```
from robot import Robot
robby = Robot()
robby.waitQuit()
```

Don't worry about the details of the code above but roughly speaking it works as follows. The first line tells Python that you are going to be using code that describes a Robot that is stored in a file `robot.py` that we have written for you. The second line tells Python to make a Robot called robby that will be displayed in a window. The third line makes the robot wait until you click the `Quit` icon (×) in the top-right corner of the display window, causing the window to close.

4. **Run the program.** To run the program, simply press the `F5` function key.

If all has gone well a window should appear showing a black-backgrounded painting area with your robot positioned in the top-left hand corner. If you now press the `Quit` icon (×) in the top-right corner of the display window, the window should disappear and the Python shell should display a fresh prompt (">>>"), to show that it is waiting for more input.

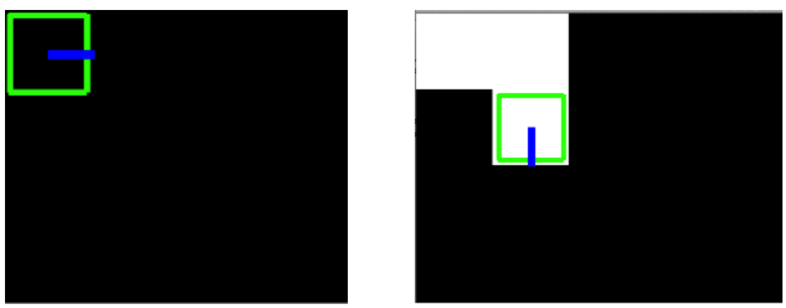

Figure 1: **Left**: Robot in initial position. **Right**: Position after code in Section 3.

3 Controlling the Robot

The Robot has a number of actions that it can perform. These can be performed by issuing the following instructions,

- `robby.turnLeft()` — turn left by 90 degrees.
- `robby.turnRight()` — turn right by 90 degrees.
- `robby.moveForward()` — move forward by one square.
- `robby.paint(colour)` – paint the square a given colour. Note, `colour` can have the value 0, 1, 2 or 3 for black, white, blue or red respectively, e.g. `robby.paint(2)` will paint the square blue.

Let's use these instructions to make the robot draw a simple pattern.

1. Edit the code that you wrote in the previous section so that it now reads as follows,

```
from robot import Robot
robby = Robot()
robby.paint(1)
robby.moveForward()
robby.paint(1)
robby.turnRight()
robby.moveForward()
robby.paint(1)
robby.waitQuit()
```

2. Run the program again by pressing F5.

If things have gone well then you should see the robot paint a white 'L' shape and end up in the position shown in the right panel of Figure 1. Check that you understand what each line of the program is doing.

End of Section Challenge:

- Edit the program so that Robby draws a horizontal red line of length 5.
- Edit the program so that Robby draws a vertical blue line of length 4.

4 Introducing loops

In the last section you wrote a program to draw a long line. You probably did this by simply repeating the instruction `moveForward()` and `paint()`. This takes a lot of typing and is rather inflexible – later we will want to be able to draw lines of different lengths. A more powerful way to repeat instructions is to use a programming construct called a *loop*.

The following code uses a loop to print `Hello` followed by `Goodbye` ten times in a row,

```
for i in range(10):
    print ''Hello''
    print ''Goodbye''
```

The line starting '`for`' tells Python that this is a loop and indicates how many times the loop should be repeated (the number of *iterations*). The instructions that are indented below the line starting '`for`' are the loop's *body*. These are the instructions that will be repeated. **Warning: the indentation is important** – Python uses the indentation to see where the body starts and ends.

- Rewrite your program for drawing a horizontal or vertical line but this time use a loop. Change the number of iterations to produce lines of different lengths. What happens if you try to draw a line that is longer than 10?

The body of a loop can be any sequence of Python instructions. So the following bit of code is perfectly valid,

```
for i in range (3):
    print ‘‘outer loop’’
    for j in range(5):
        print ‘‘inner loop’’
```

Study the code carefully and try and work out what output it will produce.

Here, a new ‘inner’ loop is appearing inside the body of an ‘outer’ loop: a loop inside a loop. This structure is called a *nested loop*. Loops can be nested to any depth, i.e. we can have a loop that’s inside a loop that’s inside a loop, etc.

Note that the outer loop starts ‘`for i ...`’ and the inner loop starts ‘`for j ...`’ . The `i` and `j` are called the *loop variables*. `i` and `j` are arbitrary names – I could have called them anything else. Variables are used for storing values that the program needs to remember. The loop variables will store how many times we have been around the loop. We will discuss variables in more detail in Section 6.

Challenge

- Using a nested loop, extend your line-drawing program so that it draws a blue box around the edge of the painting area (as in Figure 2).
  - **Tip:** The edges of the box are drawn by an inner loop that sits inside an outer loop that repeats 4 times. Remember to turn the robot after drawing each edge.

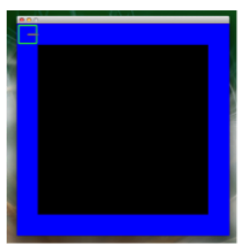

Figure 2: Board and robot position after drawing a blue border.

- What change do you need to make to your code to make it draw a smaller square, e.g. one whose sides are of length 6?

5 Functions

In the previous section we ended by writing some Python code that drew a blue border. This is a useful thing that might need to be done several times at different places in a more complex program. We don’t want to have to retype the code for doing something every time we want to do it. We can solve this problem by wrapping up these bits of code in a structure called a *function* to which we can assign a *function name*. Once a function has been defined it can be performed by calling its name.

To define a new function in Python we write,

```
    def functionName():
```

where '`functionName`' can be any name that we chose. We then write the *body* of the function (i.e. the instructions that it performs) on the following lines. Each line in the body must be indented. To call the function we then just need to write `functionName()` at some point in the program after the function has been defined. This is made clear by the example program shown in Figure 3 (next page).

```python
from robot import Robot

# define a function called drawLine
def drawLine():
    for j in range(10):
        robby.paint(2)
        robby.moveForward()

# body of the main program
robby = Robot()

# draw lines by calling my drawLine function
drawLine()
robby.turnRight()
drawLine()

robby.waitQuit()
```

Figure 3: A Python program illustrating the definition and use of a function.

In this example the lines starting with '`#`' are *comments*. Comments are ignored by Python. So why are they there? Well-chosen comments can make the code more understandable to other readers (and ourselves!)

Challenge

- Start by typing in the code above that defines and uses the function `drawLine()`. Check that it works.
- Now, add a new function called `drawSquare()` that uses the `drawLine()` function to paint a blue border. (The new function should be added before the main program body, where the function is called.)

6 Introducing variables

In the previous section we wrote a function that drew a blue square of size 10. This might be useful, but what if later we want to draw a *red* square or a *smaller* square? It would be much more useful if our function could draw a square of arbitrary size and colour. We can achieve this by using the idea of a variable.

A variable is like a labeled bucket that we can use to store a value. We can put a value in the bucket (*assignment*) or we can look in the bucket and see what it holds. For example, we can make a bucket called *length* and put the number 10 inside it,

```
length = 10
```

Then we can use the value in the bucket by referring to the bucket's name,

```
print length+5
```

This will print '15'. (Try typing these lines directly into the Python shell.)

Or, let's say that we want to change the value stored in the bucket. Say we want to increase it by 2. This can be done by typing,

```
length = length + 2
print length
```

Again, you can test this by typing the lines directly into the Python shell. Now consider the following bit of code,

```
length = 7
colour = 3
for j in range(length):
    robby.paint(colour)
    robby.moveForward()
```

This will draw a line of length 7 and colour 3.

We can now use the idea of a variable to define a *parameterized function*. Look carefully at the code below,

```
def drawLine(length, colour):
    for j in range(length):
        robby.paint(colour)
        robby.moveForward()
```

This defines a function called `drawLine` that has two *parameters*, i.e. it has two inputs. To use this function we pass it values that it will use to store in the variable `length` and `colour`. For example,

```
drawLine(10,3)
```

will draw a line of length 10 and colour 3, but,

```
drawLine(5,2)
```

will draw a line of length 5 and colour 2.

Challenge:

- Edit the `drawLine` and `drawSquare` functions that you wrote in the previous section so that they now take a length and colour parameter. Use your new function to write a program that produces the pattern shown in Figure 4.

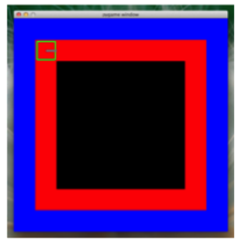

Figure 4: A red frame inside a blue frame (`borders`).

7 Using the loop variable

Remember that the loop statement includes a variable name, the *loop variable*. For example,

```
for i in range(10):
```

The loop variable is used by Python to keep track of how many times it has been around the loop. To see how it behaves, type the following bit of code directly into the Python shell:

```
for i in range(10):
    print i
```

Look carefully at what is printed. Can you see what is happening? Each time around the loop that value of i is increased, starting at 0 and ending at 9. This is very useful. For example, look at the code below – try guessing what it is going to do and then enter it into the shell to see if you are right.

```
for i in range(10):
    print i , " squared is ", i*i
```

We can now combine this idea with the parameterized functions from the previous section. For example, try adding the following lines to the end of your program in the editor,

```
for i in range(10):
    drawSqauare(i, i)
```

If you run the program the robot should generate the pattern shown in the left panel of Figure 5.

Figure 5: Patterns created by drawing squares of decreasing size (`chevrons` and `bullseye`).

Challenge

- Try and modify the code you have just written so that it produce the pattern shown on the right hand side of Figure 5.
- The patterns shown in Figure 6 can all be generated by programs that use simple (non-nested) loops that call `drawLine()`, `drawSquare()` and the robot move functions. See if you can reproduce them.

Figure 6: Patterns created by simple programs using loops (`stairs`, `stripes` and `rings`).

8 Making decisions

We have seen how by writing simple programs we can instruct the robot to produce surprisingly complex patterns. However, although the robot is moving in complicated patterns it is not responding to its environment. It always behaves the same way. We will now do something a bit more interesting: we will write a program in which the robot interacts with its environment – its behaviour will depend on how the board is initially painted.

In order to allow the robot to interact with the board pattern we will introduce two more robot functions,

- `robby.lookDown()` – returns the colour of the square where the robot is positioned.
- `robby.lookAhead()` – returns the colour of the square in front of the robot, or -1 if the robot is facing the edge of the area.

To see how these work try the following,

```
robby.paint(3);
colourBelow = robby.lookDown()
colourInfront = robby.lookAhead()
print colourBelow, colourInfront
```

We've also provided you with a function that will initialize the painting area with an interesting pattern,

- `robby.loadBoard(patternNumber)` – where `patternNumber` can be 1 to 6.

We will now write a program in which the robot looks at the colour of the board and does something that depends on the colour. To do this we will have to introduce a new Python programming concept: something called an *if-statement*. If-statements look like this,

```
if condition:
    do-something
```

Or this,

```
if condition:
    do-something
else:
    do-something-different
```

The bit called '`condition`' is a *logical test* that can be evaluated as either true or false. There are many different tests we can use. Some examples include,

- `if x==0:` – true if the value of x equals 0
- `if x!=0:` – true if the value of x is not equal to 0
- `if x<2:` – true if the value of the variable x is less than 2
- `if x>=1:` – true if the value of x is greater than or equal to 1

Look at the code below and try to work out what it does. (Note, the code would usually have comments to make it easier to understand, but I've deliberately left them out this time!)

```
from robot import Robot
robby = Robot()
robby.loadBoard(2)  # load checkerboard pattern

for i in range(10):
    for i in range(10):
        colour = robby.lookDown()
        if colour==0:
            robby.paint(1)
        else:
            robby.paint(0)
        robby.moveForward()
    robby.turnRight()
    robby.turnRight()
    for i in range(10):
        robby.moveForward()
    robby.turnLeft()
    robby.moveForward()
    robby.turnLeft()

robby.waitQuit()
```

To save you some typing, this code has been provided for you, as the file `checkerboard.py`. Open and run this code to check that it behaves as you expected.

9 Following a path

In this final section we are going to see how, with a surprisingly simple program, we can generate *seemingly* intelligent behaviour. We will program Robby to follow a path until she finds some treasure.

In this task the paths are defined by white squares on a black background. At some point on the path there will be a red square (the treasure). Robby must follow the path without moving into any of the black areas (the walls). Robby should stop when she is facing the red square. To help her remember where she has been she will paint the path blue as she moves.

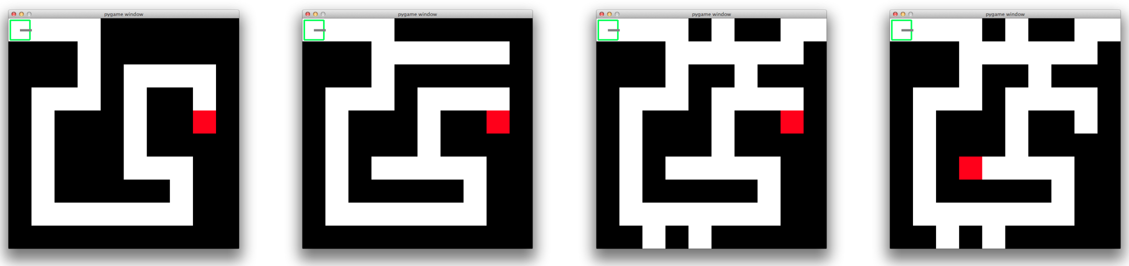

Figure 7: Initial positions after loading boards 3, 4, 5 and 6. The challenge is to write a program that navigates the robot to the red square without moving onto the black areas.

The first path we are going to use is a simple path with no branches (leftmost image in Figure 7).

```
robby.loadBoard(3)  # load path pattern
```

Below is a simple algorithm that will allow Robby to follow this path and reach the treasure,

```
for lots of time
    Look Forward
    if the square is red
        Quit
    else if the square is not white
        Turn Right
    else
        Paint the square blue
        Move Forward
```

Note that this algorithm is not written in Python. It is a semi-formal 'sketch' of a Python program written in something that is between English and formal Python code. This is called *pseudocode*. Programmers often write and discuss algorithms using pseudocode before converting the pseudocode into real code.

Challenge

- Try and turn the pseudocode above into a real Python program.

Tip: *If-statements* can be nested. For example,

```
if x<6:
    print ``x is less than 6''
else:
    if x<11:
        print ``x is in the range 6 to 10''
    else:
        print ``x is greater than 10''
```

The nested if statements above can be written in a simpler form using an *if-elif-else* structure,

```
if x<6:
    print ``x is less than 6''
elif x<11:
    print ``x is in the range 6 to 10''
else:
    print ``x is greater than 10''
```

If you get stuck you can load a correct solution (`mazeSolution1.py`) from the solutions folder.

Super challenge

Once you have solved the first maze, change the code so that the program starts by loading board 4 instead of 3. Try running it again. What goes wrong. How might you improve the algorithm so that it works for this new maze?

hint: What path would the robot would take if it were following a wall?

If you implement a wall-following strategy the robot will find the solution to board 4. The same approach should work equally well for the more complicated path in board 5. But what happens when you try board 6.

Think about how you might program the robot to systematically explore every passageway of an arbitrary maze and always find treasure.

10 Programming with Objects

It might interest you to know that, within this lab class, you've been taking your first steps at programming with **objects**. This is because `robby` in our code is an *example of an object*. Simply put, an object is just a *bundle of data*, stored as a *collection of variables*, which carries with it a *collection of functions* for operating on that data. For example, the object `robby` will have within it variables to store its `position` on the grid, its `orientation`, its `speed` of movement, and so on. It will have functions, such as `moveForward`, to change its position, depending on its orientation, and to make this change visible the screen.

How do we define what goes into an object? We do this by defining a **class**, which specifies the variables allowed, and defines the functions to operate over them. In our case, `robby` is an instance of the `Robot` class. The relationship here is like that between a general concept (the class) and an *instance* of that class (the object). For example, for the general concept of `Person`, we know that any person will have a name, an age, a gender, and so on. But a specific *instance* of `Person`, e.g. **you**, has a specific name, and a specific age and gender.

The `Robot` class we have used today doesn't allow you to have multiple robots on the same display screen, but we could modify it to do so. In that case, we might create *multiple instances* of the Robot class, perhaps called `robby` and `bobby`, which would need to store their *own* `position`, `orientation` and `speed`. Their associated functions could then be called to move them independently around the screen.

This approach to programming, with *classes* and *objects*, is known as `object-oriented programming`. It has been found to be very useful as a way to package up program functionality into usable (and reusable) blocks, making it easier to create large pieces of software that work correctly. For this reason, it has become the *dominant* approach to writing software.

11 Next steps

This session has provided the briefest introduction to programming and Python. We have focused on a small subset of concepts that lie at the heart of Python – and other programming language. We have seen that from these simple roots we can grow sophisticated behaviours. However, there is *a lot* of important stuff that we simply haven't had time to cover. If you have enjoyed this session you may wish to take your Python programming further. Fortunately Python is free to install on your computer and there are plenty of good books and online tutorials to help you learn the rest of the language. Visit the official Python web page to find out more, `http://www.python.org`.