Self-Teach Exercises 1-12

Turtle Python

The exercises below are intended to get you started using Turtle Graphics programming, and to take you fairly systematically through some of the basic facilities that are provided. If you have difficulty getting to grips with any of the concepts as they are introduced, you might find it helpful to load and run some of the illustrative programs that are available through the "Help" menu. Note, however, that a few of these illustrative programs are quite complex, so stick to the ones that are at your current level.

Before You Start the Exercises

By default, the Turtle System opens up in the programming language *Turtle Pascal*. These exercises are for *Turtle Python*, so the first thing you need to do is go to the "LANGUAGE" menu and select "Turtle Python".

Now go to the "Help" menu, select the first of the Illustrative programs (called "Simple drawing with pauses") and see this appear in the Programming Area at the left of the screen. Click on the **RUN** button and watch what happens. Having done this, read through the section on "The Program" so that you understand what's going on, and then return back here.

Exercise 1

If a program is currently loaded into the system, select "New program" from the "File" menu to clear it. Make sure the flashing cursor is in the Programming Area at the left of the screen (click there if necessary), and type in the following program:

```
# tgpx1

def main():
  forward(200)
  right(120)
  forward(200)
```

Note that program examples from this file can be "cut and pasted" into the system if you have the file in electronic form. To try this, drag the mouse over the example above so that all six lines (within this file) are highlighted. Then press CTRL-C (i.e. hold down the "Ctrl" key and press "C") to copy the selection into the Clipboard. Now go into the Turtle System, click in the part of the Programming Area where you want the selected lines to be pasted (line 1 if no program exists yet), and then either select "Paste insert" from the Edit menu, or press CTRL-V. It's worth getting used to using CTRL-V and CTRL-C for these operations, because they're standard in nearly all Windows applications and are quicker than using the menus.

Then press on the **RUN** button to run it. You should see two sides of an equilateral triangle appear on the Canvas (the square area on the right of the screen where the drawing is done).

Now add "statements" (i.e. *Turtle Phython* commands) to the program to complete the equilateral triangle. **RUN** the new program to check that it works.

Next, edit the program (i.e. add, delete, or modify statements) so the same triangle is drawn, but this time with a horizontal base (**RUN** it to check, as usual). Then edit it again so that, in addition to the triangle, it draws a red horizontal line exactly below the base, at a distance of 100 units (note – here

you will need the `colour` command, and also the `penup`/`pendown` commands, and you may want to use `right` and/or `back`, but do not at this stage use `movexy`, `drawxy` or `setxy` etc.). If you want to know what commands are available, you might find the "QuickHelp 2" tab at the bottom of the screen useful.

Save your program, calling it **TGYX1.TGY**, within an appropriately named directory on your computer or the network that you are using (e.g. you could use Windows Explorer to create a "Turtle" directory if you don't have one already). Having saved your program, unless you've already laid it out very neatly, you might like to try selecting the "auto-format" option from the "Edit" menu. If this gives a better result, save the neatened version in place of the old one. (It's a good idea to use the "auto-format" option whenever you've done a major edit on the structure of your program, because this keeps all the indenting in order, but always save the program first in case you don't like the result for any reason.

Now choose "New program" from the "File" menu before starting the next exercise. (If you like, you could then try reloading the program you've saved, just to check out the loading and saving operations.)

Exercise 2

Write from scratch a program to draw a face, enclosed in a black circle (radius 300), with small green blots (i.e. filled circles) for eyes, a thick blue triangle for a nose, and a red smiling mouth. The simplest way of making the mouth is to draw a red blot, and then to draw a white blot slightly higher, leaving a crescent of red (but if you do it this way, you may need to think carefully about the order in which you draw the features). You will need to make use of the `circle`, `blot`, and `thickness` commands, in addition to those mentioned earlier, and you can if you wish also use `movexy` and/or `drawxy` (see if you can work out how to use these from the "QuickHelp 2" tab, with some experimentation). Save your program as **TGYX2.TGY**.

Exercise 3

Now edit the program from the previous exercise, adding a loop so that it draws a row of five or more faces across the Canvas (you will have to change the radius from 300 to 100 to fit them all in). You can do this in either of the two ways below, both of which involve a variable which is used to count the number of faces that have been drawn – here we will call this variable `facesdrawn`. Our use of this variable will tell the system that `facesdrawn` is being used as the name of a memory location that can store a single integer (i.e. whole number) at any one time. Think of this memory location as a box, into which just one integer can be put, so if a second integer is then stored in the box, the first one will be pushed out. To put the integer 0 into the box, use the statement:

```
facesdrawn=0
```

and to replace this with the number 1, use:

```
facesdrawn=1
```

Now let's look at the two methods for counting through five faces.

3(f) Using a FOR loop:

A "for loop" – often called a "counting loop" – automatically counts through the range of values you specify (e.g. 1 to 5), repeating the relevant operations each time, and incrementing your variable by the amount specified. In the example below, `facesdrawn` takes all of the values from 1 to 5 (being

incremented by 1 each time). So it is first given the value 1 (after which the statements up to **return** are executed once), then the value 2 (and the statements are executed a second time), then the value 3 (third time), then 4 (again), and finally 5 (again). Having reached 5, the loop stops.

```
# tgpx3f

def main():
  <put statements to position the first face here>
  for facesdrawn in range(1,5,1):
    <put statements to draw a face here>
    <put statements to move to the next face position here>
  return
```

3(w) Using a WHIILE loop:

A "while loop" repeatedly does the relevant operations while the specified condition remains true, and checks the condition each time before doing them. The condition does not have to involve any particular variable, but could involve some other condition (e.g. looping until some key is pressed); so this kind of loop is more versatile than a "for" loop. In the example below, however, the variable `facesdrawn` is used to count the number of faces, initially being given the value 0 and then incremented by 1 each time a face is drawn until it reaches the value 5, when the loop stops:

```
# tgpx3r

def main():
  <put statements to position the first face here>
  while facesdrawn<5:
    <put statements to draw a face here>
    <put statements to move to the next face position here>
    facesdrawn=facesdrawn+1
```

The statement:

```
facesdrawn=facesdrawn+1
```

means something like "put into the `facesdrawn` box the number that comes out of the box plus 1" – so this has the effect of adding 1 to the number in the box. Since this is a very common form of operation, the system provides a useful shorthand:

```
inc(facesdrawn)
```

Likewise `facesdrawn=facesdrawn-1` – or the shorthand `dec(facesdrawn)` – would have the effect of decrementing the variable `facesdrawn` by 1 (i.e. subtracting 1). To add 3 to the variable, use `facesdrawn=facesdrawn+3`, or to square it, `facesdrawn=facesdrawn*facesdrawn` (note that the asterisk "*" is used to signify the operation of multiplication).

Try editing your original face-drawing program in one of these ways, and when you've done it in one way, edit it further so that it works in the other way instead. Save the version with a "for" loop as **TGYX3F.TGY**, and the version with a "while" loop as **TGYX3W.TGY**. Having produced these, can you edit one of them so that it produces four rows, each of five faces? (Hint: use a variable called `rows`, which counts the number of rows just as `facesdrawn` counts the number of faces in each row – take a look at the illustrative program called "Nested FOR loops" under the first set of examples to see the structure that results in the case of "for" loops). Save whatever you produce as **TGYX3.TGY**.

Exercise 4

Write a program using at least one "for" loop and one "while" loop, to create an abstract design of your own choice. Use the command `randcol` to make it colourful (e.g. `randcol(6)` selects one of the first 6 colours). Save your program as **TGYX4.TGY**.

Exercise 5

The following example program illustrates the use of a simple procedure:

```
# squares

def drawsquare50():
  forward(50)
  right(90)
  forward(50)
  right(90)
  forward(50)
  right(90)
  forward(50)
  right(90)

def main():
  for count in range(1,8,1):
    randcol(6)
    drawsquare50()
    forward(50)
```

Start a new program and type this example in or paste it using the Clipboard, noting as you do the following points about the program structure, several of which should by now be fairly familiar:

a) The program begins with a hash (#) followed by the program's name (this is optional, but it is good practice to name your programs – the # indicates a comment that is intended to help people who read your program, but which the computer ignores);
b) The program then defines a procedure called `drawsquare50`;
c) An integer variable `count` is declared in a "for" loop;
d) The procedure `drawsquare50` is called within the "for" loop.

Having noted all these points, run the program you've entered and check that you understand how it works. Save it under the filename **SQUARES.TGY**.

Now, using the "Procedure with parameters" example program (from "Examples 1" under the "Help" menu) as a model, see if you can adapt the `drawsquare50` procedure to produce a more versatile `drawsquare` procedure which can draw squares of different sizes and in different colours – ask for help with this if necessary, because it needs some thinking![1]

[1] Hints: If the procedure is intended to draw squares of various sizes, then it could start with the line `def drawsquare(size):` and be called with `drawsquare(40)`. If it is intended to draw squares of various sizes and colours, it could start with `def drawsquare(size, col):` and be called with `drawsquare(40, green)`.

Adapt the program so that it produces an interesting design of your choice, preferably involving at least one loop (and maybe, if you're feeling very ambitious, recursion as illustrated by "Recursive triangles" under "Examples 2" in the "Help" menu). Save the result as **TGYX5.TGY**.

Exercise 6

Reload the final program you did for exercise 3, and reorganise it so that the statements which draw the face are enclosed within a procedure called `face`. Make sure that it works correctly, and that you fully understand what is going on. Save your revised program as **TGYX6.TGY**.

Can you adapt the procedure so that it is able to draw faces of different sizes and/or colours? If you manage this, save the resulting program as **TGYX6P.TGY**.

Exercise 7

Create a program which draws a row of three houses, and which includes *both* a procedure called `house`, which draws one house, and another procedure called `window`, which draws a window of a house (and is therefore called by the `house` procedure – this means that it must be placed in the program text *before* the `house` procedure). The structure of your program might be like this:

```
# tgpx7

def window():
  <put statements to draw a window here>

def house():
  <put statements to draw a house here, calling window>

def main():
  <put main program statements here, calling house>
```

You may find the `polygon` command useful in this exercise, since it will enable you to display a filled shape rather than just a line drawing. The related command `polyline` can also be useful, for example if you want to draw a line border around such a shape.[2]

If you haven't already done so, now adapt the window and house procedures so that each of them takes at least one parameter (i.e. a number that gets "fed into" the procedure), enabling your program – simply by editing these parameters – to produce different designs of windows and houses (e.g. different sizes or colours, or possibly even more elaborate variations). Save your program as **TGYX7.TGY**.

Exercise 8

Write a program using at least two procedures, to create an abstract design of your own choice. If you wish, make use of the new commands `blank`, `direction`, `ellblot`, `ellipse`, `fill`, `recolour`, `setx`, `sety`, and `setxy`. Save your program as **TGYX8.TGY**.

---

[2] `polygon(8)` fills in the shape made by the last 8 points visited. To draw a line around this shape, you need to return to the first of these 8 points, and then use the command `polyline(9)`, joining the last 9 points visited.

Exercise 9

This exercise asks you to produce a program containing a *moving* red ball. Each time the ball (i.e. a red blot) has to move, you simply "erase" it from the old position (by drawing a white blot over it), then draw it in the new position. So you need to keep a record of the ball's position at each stage, and also what "velocity" should be applied to the ball when moving from each stage to the next. Restricting ourselves to the X (horizontal) dimension for the moment, this involves the following steps:

a) Use a variable `x` to signify the x-coordinate of the ball.
b) Use a variable `xvel` to signify the velocity of the ball in the x-direction (i.e. the amount that the ball moves in the x-direction for each "cycle" of the program – if `xvel` is positive then the ball will move to the right; if `xvel` is negative, then the ball will move to the left).
c) Set `x` and `xvel` to appropriate initial values.
d) Repeatedly:
    1) Draw a white blot;
    2) Move the turtle to the position signified by `x` (use the `setx` command for this);
    3) Draw a red blot;
    4) To make the motion appear smooth, add the commands `update` and `noupdate` at this point (as explained below);
    5) Add `xvel` to `x`, so as to "move" to a new position.

To make this movement appear smooth, without the ball flashing "on" and "off", the drawing of the white blot over the previous position of the red ball – at stage (d1) – should be virtually simultaneous with the drawing of the red ball in its new position – at stage (d3). This is achieved by ensuring that the screen is not updated at stages (d1) to (d3), but only at stage (d4), by inserting an `update` command there (and then immediately turning off updating for the next time round the loop, with `noupdate`).

If you need further help, simply copy the following program (without the two indented lines), run it, and then work out what's going on – can you see how it produces the effect of a moving ball?

```
# tgpx9

def main():
  x=0
  xvel=2
  while x<1000:
    colour(white)
    blot(25)
    setx(x)
    colour(red)
    blot(25)
                    update()
                    noupdate()
    x=x+xvel
```

You'll notice that when this program is run the "ball" appears to flash a lot, because the effect of movement depends on an alternating sequence of red and white blots, with each white blot "rubbing out" the previous red blot in preparation for a new red blot to be drawn in the next position. To eliminate this flashing and give an effect of smooth movement, simply add the `update` and `noupdate` instructions to the program in the space where they're shown. Now run the program again – this

time, each white blot is updated to the Canvas only when the subsequent red blot command has also been executed: hence the blots appear simultaneously, and an impression of relatively smooth movement is given. For more examples of this technique, see the built-in illustrative programs that involve movement.

Having understood all this, see if you can adapt the program, to make the ball move in *two* dimensions (hint: you will need variables called `y` and `yvel` as well as `x` and `xvel`, and you might find the command `setxy` helpful). However far you get, save your program as **TGYX9.TGY**.

Exercise 10

Take the program which you produced in Exercise 9, and modify it so that the ball "bounces" when it reaches the edge of the Canvas. To do this, you will need to use the **`if`** instruction to change the ball's velocity at an appropriate point, so that instead of moving right, it starts moving left instead. You can do this with a statement such as:

```
if x>975:
  xvel=-xvel
```

which changes the direction of horizontal motion (by changing `xvel` from positive to negative) when the ball gets within 25 units of the right-hand edge of the standard 1000x1000 Canvas. Now you will probably find that the ball bounces back across to the left-hand edge, so to make it bounce at both sides, you can adapt the statement above as follows:

```
if (x<25) or (x>975):
  xvel=-xvel
```

This switches `xvel` between positive and negative whenever the ball gets within 25 units of either the left or right edge. Note that we need to use brackets to express a complex condition like this one. When you have finished your bouncing ball program, save it as **TGYX10.TGY**.

Quite generally, the **`if`** structure is extremely useful whenever you want to perform some statement (or set of statements) subject to some condition – e.g. depending on the position or velocity of some object, or whether or not the mouse has been clicked or a key pressed (we'll come to these later). The structure can also be extended with … **`else`** …, as in this example:

```
if radius<100:
  circle(radius)
else:
  radius:=100
  blot(radius)
```

This will draw a circle using the current value of the variable `radius` *if* that value is less than 100, but *otherwise* will set the value of `radius` to 100 before drawing a blot using that value.

Exercise 11

Variables can store string values as well as integers. A string is a sequence of letters and other characters, possibly a word or a sentence. String variables are used in much the same way as integer variables, and their values are set by putting a sequence of characters inside single quotation marks:

```
sometext='put your text here'
```

Strings can be printed to the Canvas at the current turtle location using the `print` command. This command takes three parameters – the string to be printed, a number representing the font family and style, and another number representing the font size. There are 16 font families in the Turtle System, numbered from 0 to 15; to see them, go to the "QuickHelp 1" tab, and then the "Fonts/Cursors" tab on the right.

The following program prints the string `'Hello World'` to the Canvas, in Arial (font 0), 28pt:

```
# helloworld

def main():
  s1='Hello'
  s2='World'
  print(s1+' '+s2, 0, 28)
```

Note the first parameter passed to the `print` command: `s1+' '+s2`. When applied to strings, the "+" symbol doesn't mean numerical addition (as it does with integers), but *concatenation*. Thus the result of `s1+s2` is a string of whatever characters are in the variable `s1` followed immediately by whatever characters are in the variable `s2`. The example shown also inserts a space between the two strings.

Write a program that prints a variety of text (e.g. headings and short paragraphs), using at least two string variables. Experiment with different font numbers, using the "Fonts" table as a guide. Save your finished program as **TGYX11.TGY**.

Exercise 12

Write a program illustrating all the main structures and types of command that you have learned so far, saving it as **TGYX12.TGY**.