Turtle Machine 2 – More Looping Structures

This document explains how three looping structures are handled by the "Turtle Machine", starting with the Pascal REPEAT and FOR loops, before considering the more complex FOR … in RANGE structure of Python. The previous document, "Turtle Machine 1 – Pcode and the Stack", has already explained how the Python WHILE loop operates, and the Pascal WHILE loop is compiled in exactly the same way.

1. Pascal REPEAT Loop

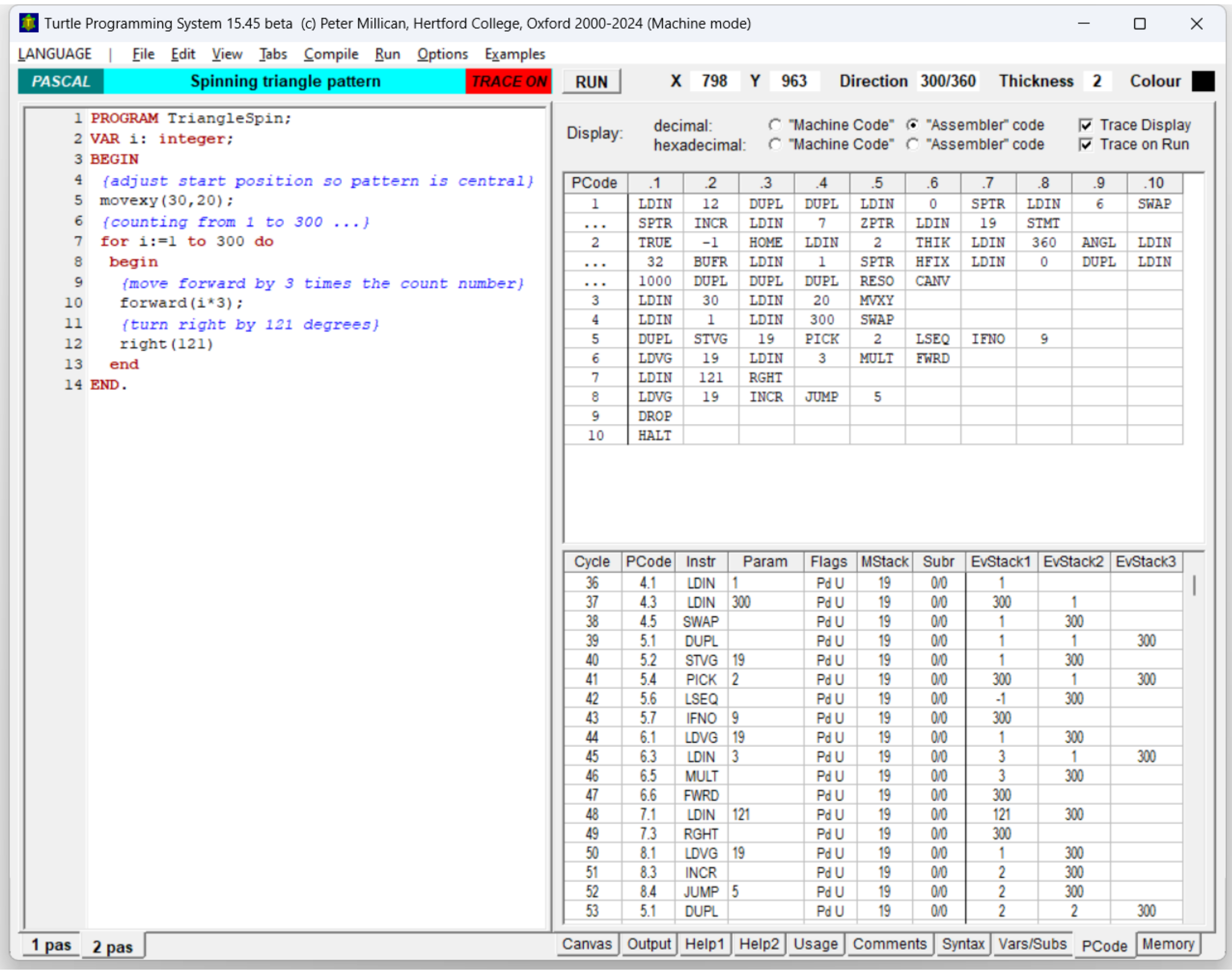
In Pascal, the "Spiral of colours" program is implemented using a REPEAT rather than WHILE loop, giving rise to a different PCode structure, as follows (and shown as usual without PCode lines 1 and 2). One other – very minor – point is that here memory location 20 is used for the variable "lineLength", instead of 21 – this reflects a subtle difference in how Python and Pascal handle subroutines (since Python standardly expects a function to return a value – and needs a location for this – whereas Pascal procedures do not).

PROGRAM ColourSpiral;	3 JUMP 8	Jump to start of main program – line 8
VAR lineLength: integer;		(lineLength will be stored at location 20)
Procedure lineTurn;	4 PSSR 1	"Push" subroutine number = 1
Begin		
forward(lineLength);	5 LDVG 20 FWRD	Load global variable 20, move forward
right(60);	6 LDIN 60 RGHT	Load integer 60, turn right
End;	7 PLSR RETN	"Pull" subroutine number, and return
BEGIN		
blank(black);	8 LDIN 0 BLNK	Load integer 0 (Black), and blank Canvas
forward(15);	9 LDIN 15 FWRD	Load integer 15, move forward
thickness(27);	10 LDIN 27 THIK	Load integer 27, thickness
lineLength := 20;	11 LDIN 20 STVG 20	Load integer 20, store in location 20
repeat	12 LDIN 40 RAND	Load 40, replace with random number 0-39
randcol(40);	INCR	Increment number (to 1-40)
	RGB COLR	Convert to native RGB code, set *Turtle* colour
lineTurn;	13 SUBR 4	Call subroutine at line 4
pause(50);	14 LDIN 50 WAIT	Load integer 50, pause 50 milliseconds
lineLength := lineLength+10	15 LDVG 20 LDIN 10	Load global location 20, load integer 10
	PLUS STVG 20	Add the two values, store in location 20
until lineLength>500	16 LDVG 20 LDIN 500	Load global location 20, load integer 500 …
	MORE	Is the former more than the latter?
	IFNO 12	If not, jump back to line 12
END.	17 HALT	But if so, halt

Note that there is no specific PCode instruction corresponding to the Pascal instruction "repeat" – this simply marks the point where the loop starts, and to which control returns after the test at PCode line 16, assuming that the "until" condition is currently false. This is therefore a simpler structure than a "while" loop, requiring just one jump, and one test *which occurs after the loop*, so that the loop is always executed at least once (unlike in the case of "while", where the test occurs *before the loop*).

2. Pascal FOR Loop

Here is the "Spinning triangle pattern" program from the Pascal menu "Examples 1 – drawing and counting loops", showing the Trace display as the program runs round the first loop:



The FOR loop is set up by loading the first and terminal values onto the Evaluation Stack, and then swapping them round so that the terminal value (i.e. 300) is below the first value (i.e. 1) on the Stack:

36	4.1	LDIN 1	| 1
37	4.3	LDIN 300	| 300 1
38	4.5	SWAP	| 1 300

From now on, the current value of the loop variable (called "i" in this case, and currently equal to 1) will be at the top of the Stack whenever we get to the start of line 5 of the PCode, which is the beginning of the loop section. Likewise the terminal value (300 in this case) will be the second value on the Stack. Note also that the terminal value will be preserved – unchanged – on the Stack throughout the loop.

The loop section begins with the current value of the loop variable *i* being duplicated and stored into the relevant memory location (here 19):

39	5.1	DUPL	| 1 1 300
40	5.2	STVG 19	| 1 300

This updates the loop variable value in memory. Now the terminal value (i.e. 300) – which is in Stack position 2 – is copied to the top of the Stack using the "PICK 2" instruction:

41	5.4	PICK 2	| 300 1 300

"LSEQ" then tests whether the second Stack value (here 1) is less than or equal to the first Stack value (i.e. 300). This leaves 0 on the Stack if it is not, or –1 on the Stack if it is. Notice here that Pascal and BASIC represent *True* with the value –1, unlike Python which represents *True* as 1. (Python's choice seems more natural, but the advantage of representing *True* as –1 is to enable bitwise operators to double as Boolean operators, because the bitwise complement of 0 is –1.)

If the result of the LSEQ comparison is 0 (i.e. the loop variable value has become greater than 300), then control branches to PCode line 9:

```
42    5.6   LSEQ        | –1  300
43    5.7   IFNO 9      | 300
```

When this eventually happens, line 9 contains the single instruction "DROP" and line 10 is "HALT". "DROP" has the effect of removing the value of 300 from the Stack before halting – it is important to "clean up" in this way after the loop has completed, because the loop might be part of a larger program in which preserving Stack structures is crucial to their operation.

The first Pascal command within the body of the loop is "forward(i*3)", which involves loading the loop variable *i*, then multiplying it by 3 and executing the FWRD command:

```
44    6.1   LDVG 19     | 1  300
45    6.3   LDIN  3     | 3  1  300
46    6.5   MULT        | 3  300
47    6.6   FWRD        | 300
```

The second Pascal command within the body of the loop is "right(121)":

```
48    7.1   LDIN  121   | 121  300
49    7.3   RGHT        | 300
```

The loop now ends by loading the current value of the loop variable *i* and incrementing it (to 2), then jumping back to line 5 with the current and terminal values correctly placed on the Stack:

```
50    8.1   LDVG 19     | 1  300
51    8.3   INCR        | 2  300
52    8.4   JUMP 5      | 2  300
```

Note that line 5 will then duplicate and store the current value of the loop variable, as follows:

```
53    5.1   DUPL        | 2  2  300
54    5.2   STVG 19     | 2  300
```

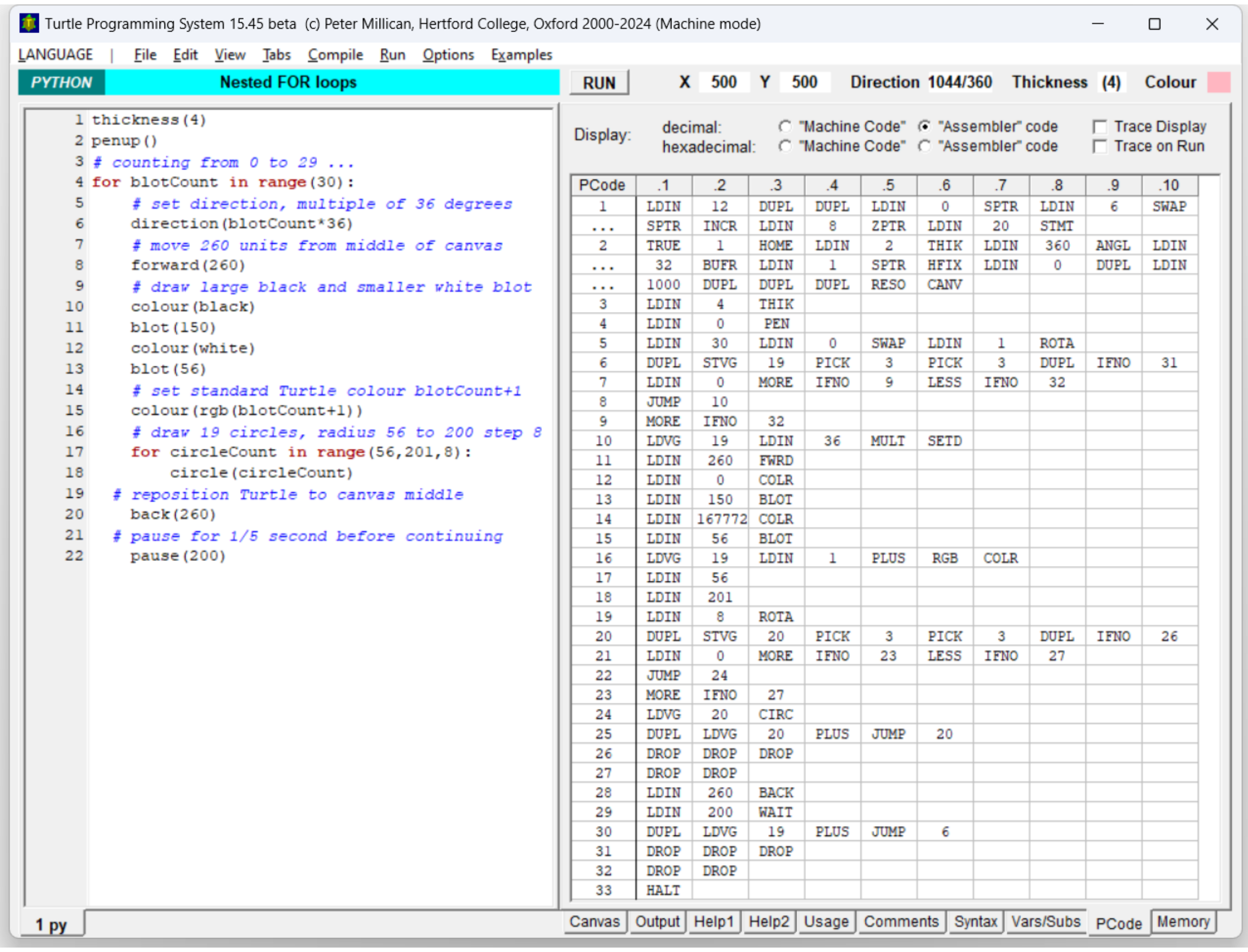
The loop will continue until eventually the current value of the loop variable reaches 301. At this point, execution of the program will terminate like this, after a jump to PCode line 9:

```
4239 5.1    DUPL        | 301  301 300
4240 5.2    STVG 19     | 301  300
4241 5.4    PICK  2     | 300  301 300
4242 5.6    LSEQ        | 0  300
4243 5.7    IFNO 9      | 300
4244 9.1    DROP
4245 10.1   HALT
```

As noted earlier, "DROP" has the effect of "cleaning up" before termination, by removing the now redundant terminal value of 300 from the Stack.

3. Python FOR … in RANGE Loop

The Python FOR loop is significantly more complicated than the Pascal FOR loop, because it is more versatile – in particular, as well as a *first* value *f* and a *terminal* value *t*, it can also involve an arbitrary *step* value *s* (which could be specified by a variable), whereas Pascal can only either increment its loop variable by 1 each time (e.g. **for** *n* := 1 **to** 100) or decrement it by 1 each time (e.g. **for** *n* := 23 **downto** 0). The "Nested FOR loops" program in the first "Examples" menu provides an illustration of this complexity.



Notice that here we have one FOR loop "nested" inside another, with the relevant structure being as follows (and omitting most of the early instructions within the outer loop):

```
for blotCount in range(30):
    direction(blotCount*36)
    ...
    for circleCount in range(56,201,8):
        circle(circleCount)
    back(260)
    pause(200)
```

The first loop – "for blotCount in range(30):" – gives blotCount in turn each value from 0 to 29 inclusive. The second – "for circleCount in range(56, 201, 8)" – starts circleCount with the value 56, then increments it repeatedly by 8 until it reaches or exceeds 201 (thus its last loop will be with 200). Note that in these range specifications, "range(30)" is short for "range(0, 30, 1)". So the general form is:

```
for v in range(first,terminal,step):
```

Each of the three parameter values may be specified numerically (as here) or using a variable, but that value will remain constant during the loop, even if has been specified using a variable which changes in value during the loop. This helps to explain why *Turtle Python* implements these loops in such a way that the specified values are preserved on the Stack.

Another significant complication in Python's FOR … in RANGE loop is that its behaviour can vary significantly depending on the nature of the *step* value, which can be negative, e.g.:

```
for v in range(365,0,-7):
    print(v)
```

which counts down 365, 358, 351, … , 15, 8, 1. Notice that here the loop terminates when the loop variable v becomes *less than or equal to* the terminal value of 0, whereas a loop with a positive step value terminates when the loop variable becomes *greater than or equal to* the terminal value. But since the step value might be specified using a variable, we also have to face the possibility that the step value will be 0 – what then? In that case, the loop is terminated immediately.

To see how all this works in detail, let's look here at a program that involves just a single loop similar in structure to the inner (more complex) loop of the "Nested FOR loops" example:

```
for v in range(56,201,8):
    print(v)
```

When run, this program simply outputs the sequence of numbers 56, 64, 72, 80, 88, 96, 104, 112, 120, 128, 136, 144, 152, 160, 168, 176, 184, 192, and 200, each on a new line – click on the "Output" tab to see them. Afterwards, *v* will be equal to 208, which exceeds the terminal value 201 and thus terminates the loop. As with the Pascal FOR loop, we can explore what's going on using the Trace display (but notice that the Trace display only shows up to three values on the Evaluation Stack – here we see all of them).

33	3.1	LDIN 56	56	Loads begin value b = 56
34	4.1	LDIN 201	201 56	Loads finish value f = 201
35	5.1	LDIN 8	8 201 56	Loads step value s = 8
36	5.3	ROTA	56 8 201	Rotates third value (56) up to the top
37	6.1	DUPL	56 56 8 201	Duplicates top value on the Stack
38	6.2	STVG 19	56 8 201	Stores top value into variable v
39	6.4	PICK 3	201 56 8 201	Copies third value to the top
40	6.6	PICK 3	8 201 56 8 201	Copies third value to the top
41	6.8	DUPL	8 8 201 56 8 201	Duplicates top value on the Stack
42	6.9	IFNO 12	8 201 56 8 201	If step value s=0, jump out of loop
43	7.1	LDIN 0	0 8 201 56 8 201	Load 0 for comparison with s
44	7.3	MORE	1 201 56 8 201	Is the step value s>0?
45	7.4	IFNO 9	201 56 8 201	If not, jump to line 9
46	7.6	LESS	1 8 201	Is v (currently 56) less than t (201)?
47	7.7	IFNO 13	8 201	If not, jump out of loop
48	8.1	JUMP 10	8 201	Jump over line 9

Line 9 simply takes the form "MORE IFNO 13", and this replaces the conditional branch "LESS IFNO 13" (in cycles 46 and 47) for the case where the step value in negative. Either way, this tests the appropriate condition for whether the loop should continue, and if so, the code continues from line 10 whether the step value is positive or negative:

49	10.1	LDVG 19	56 8 201	Load the global location 19 (i.e. variable v)
50	10.3	ITOS	200035 8 201	Convert integer to string at Heap address

51	10.4	WRIT	8 201	Print the value of v to the Console
52	10.5	LSTR #13#10	200038 8 201	Load end of line and new line string
53	10.9	WRIT	8 201	Print new line on Console
54	10.1	HCLR	8 201	Clear the Heap of string leftovers
55	11.1	DUPL	8 8 201	Duplicate the step value s
56	11.2	LDVG 19	56 8 8	Load the loop variable v
57	11.4	PLUS	64 8 201	Add step value s to loop variable v
58	11.5	JUMP 6	64 8 201	Jump back to continue the loop
59	6.1	DUPL	64 64 8 201	(compare with cycle 37 above)

Thus the loop continues onto the next iteration, and it carries on this way until it is terminated by the conditional branch at PCode 7.7 (see cycle 47 above).

Termination at that point happens when v reaches 201 or more, at which point a jump is made to line 13. Notice that we also saw a potential exit from the loop at PCode 6.9 (see cycle 42 above), which results in a jump to line 12 if the step value s is zero. The final three lines of PCode – 12 to 14 – contain six rather monotonous instructions:

```
12.1   DROP
12.2   DROP
12.3   DROP
13.1   DROP
13.2   DROP
14.1   HALT
```

In the case of the Pascal FOR loop, we saw a single "DROP" instruction being needed at the end of the loop, to "clean up" the Stack once the terminal value had become redundant. With a Python FOR … in RANGE loop more cleaning up is needed, because if the loop is exited from PCode 6.9, there are no fewer than five redundant values on the Stack, while if it is exited from PCode 7.7 (or 9.2 if the step value is negative), there are two redundant values on the Stack. The necessary cleaning up is therefore achieved by jumping to line 12 in the former case (so that five "DROP"s occur before "HALT"), and to line 13 in the latter (so that two "DROP"s occur before "HALT").

As noted earlier, this sort of "cleaning up" is essential when a loop occurs as part of a larger program. In the "Nested FOR loops" program which we saw above, for example, one FOR loop occurs within another, and *both* of these loops store their step and terminal values on the Stack. Thus if the inner loop terminated without cleaning up and removing its own redundant values from the Stack, those values would remain there and wreck the operation of the outer loop by being mistaken for its own step and terminal values.