Turtle Machine 3 – Subroutines and Recursion

This document explains how the "Turtle Machine" handles *subroutines*, illustrated through three simple Python programs from the menu "Examples 2 – functions and simple recursion". We have already seen the first program from this menu, "Spiral of colours", in the document "Turtle Machine 1 – PCode and the Stack". That involved a subroutine which just consisted of two instructions, one moving *forward* and the other turning *right*, and referring only to one *global* variable. Subroutines become much more interesting and tricky when they require the use of *local* variables, and especially when they are *recursive*. Knowing how variables are treated is particularly valuable for understanding the behaviour of recursive programs, so as to achieve a realistic mental model of how the computer is processing them.

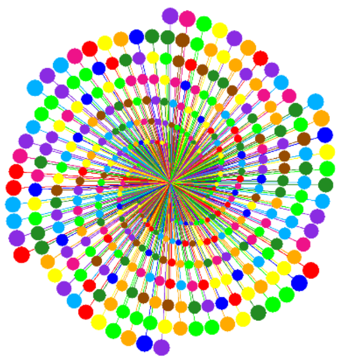
1. Function with Parameter

The "Function with parameter" program contains one short *subroutine* – commonly called a *function* in Python – which has the effect of drawing a "prong" of a given size:

```
def prong(size):
    forward(size)
    blot(size/20)
    back(size)
```

The value of "size" is passed into the function as a *parameter* when the function is called by the instruction "prong(count+100)". Here "count" is the counting variable of a FOR loop, counting down from 360 to 0, so the "size" parameter accordingly counts down from 460 to 100. For each given value of "size", the *Turtle* moves forward by that number of units, draws a "blot" (i.e. a filled circle) of radius size/20 (rounded to the nearest integer), and then moves back to the point from which it started. Each prong is given one of the first ten *Turtle* colours at random, and immediately afterwards, the *Turtle* is turned right by 61 degrees:

```
for count in range(360,0,-1):
    randcol(10)
    prong(count+100)
    right(61)
```



Since there are 360 degrees in a circle, turning right by 61 degrees each time has the effect of moving the *Turtle* 6 degrees to the right (i.e. 366 degrees) for every six prongs that are drawn. And because the prongs are getting shorter each time, this produces a spiral effect (as shown here). If you change the angle to 59 degrees, the sprial will go the other way. The result is quite different if you change it to 60 degrees – can you see why?

We won't here consider in detail the PCode that handles the FOR loop as shown above, because this was discussed in detail in the previous document "Turtle Machine 2 – More Looping Structures". But a couple of points are worth noting:

- First, you will see at PCode lines 11 to 13 that the three parameters to the "range" function are loaded in turn onto the Evaluation Stack: first 360, then 0, then –1. But you will see that –1 is not just loaded using "LDIN -1"; instead, there are two PCode instructions, "LDIN 1" and "NEG", the second of which *negates* the value on the Stack (i.e. changes –1 to 1). The reason for this is that the symbol "–" is not always part of a number, as it also serves as the subtraction operator. So when the program text is read in, "–" has to be treated initially as a distinct "lexeme" (i.e. lexical item) from whatever number may follow it. The *Turtle* compiler reflects this in the PCode accordingly (and because it is intended to be educational rather than maximally efficient, it does not optimise the PCode by combining the two PCode instructions into "LDIN -1").

- Secondly, at PCode lines 15 to 17 you will see:

  ```
  15      LDIN 0  MORE  IFNO 17  LESS  IFNO 23
  16      JUMP 18
  17      MORE  INFO 23
  ```

  As line 15 is entered, the *step* value of the loop is currently at the top of the Stack, with the *terminal* value just below it, and the current value of the counting variable just below that. (If you are not clear about this, consult the "Turtle Machine 2" document.) So "LDIN 0 MORE" checks whether the *step* value is more than zero, and if not, "IFNO 17" branches to line 17. Otherwise, "LESS" compares the current counting variable with the *terminal* value, and if it is not less, branches to line 23 and thus leaves the loop. However in the current program – unlike those we've seen previously – the *step* value is negative (i.e. –1) rather than positive, so in this case we do branch to line 17. And here, the test "MORE" (rather than "LESS") is applied to compare the current counting variable with the *terminal* value, so that we branch out of the loop if the counting variable is not *more than* the terminal value. With a terminal value of 0 (as we have here), this branch out of the loop occurs when the count gets down 0 (or less).

We'll return to the main program later, but let's now turn to the subroutine "prong".

The PCode for the subroutine is as follows, starting at line 4. As we've seen before (in both the "Turtle Machine 1" and "Turtle Machine 2" documents), the first subroutine in a program standardly starts with "PSSR 1" and ends with "PLSR RETN". The first of these *pushes* the subroutine number onto a *Subroutine Register Stack* (which is useful when tracing but plays no essential part in the processing). Then "PLSR" simply *pulls* that number off the *Subroutine Register Stack* before "RETN" returns from the subroutine.

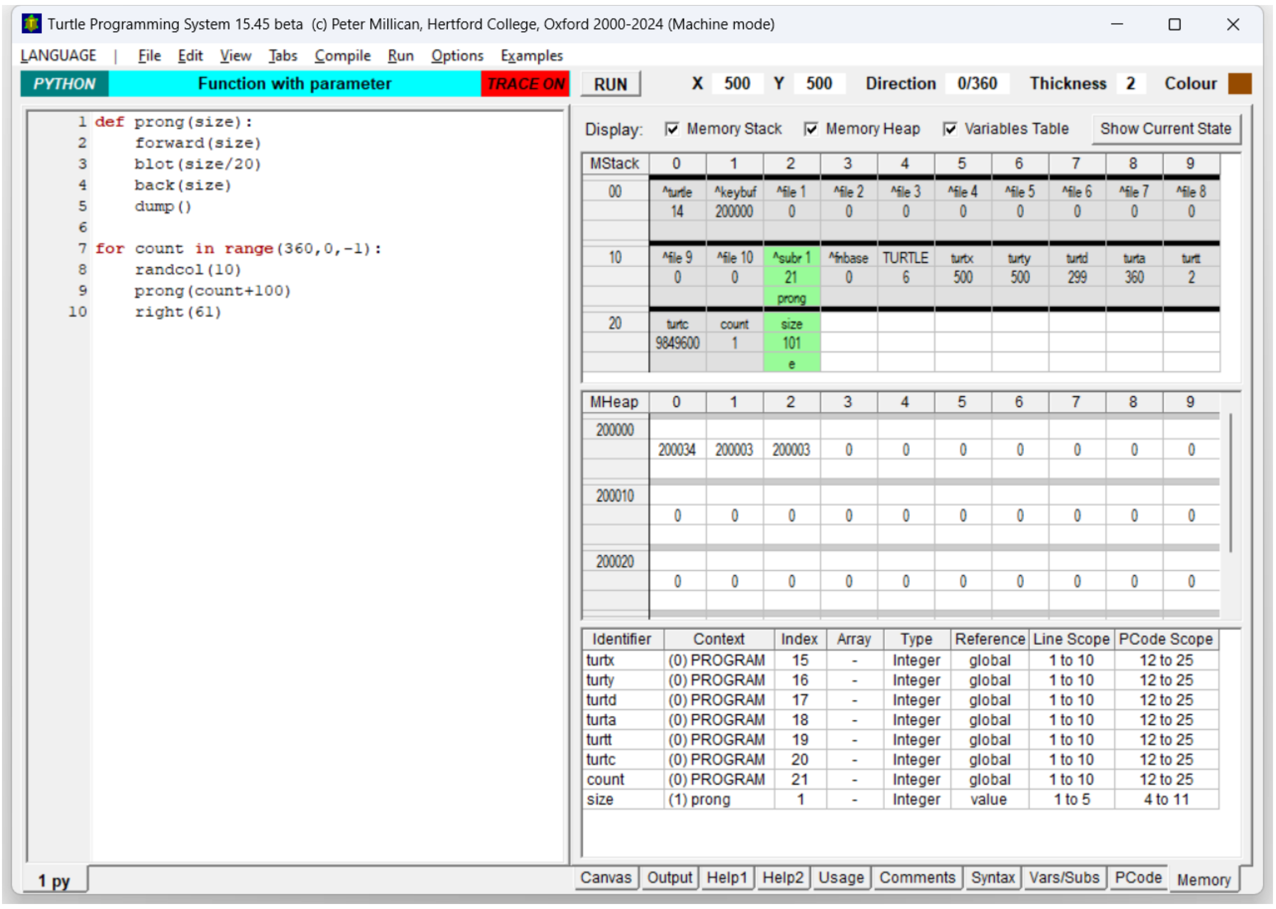
```
4       PSSR 1
5       MEMC 12 1
6       LDAV 12 1  LDIN 1  ZPTR  STVV 12 1
7       LDVV 12 1  FWRD
8       LDVV 12 1  LDIN 20  DIVR  BLOT
9       LDVV 12 1  BACK
10      MEMR 12  PLSR  RETN
```

Focusing now on the main body of the subroutine, lines 7 to 9 will turn out to be relatively straightforward, but the unfamiliar complications concern lines 5 and 6, and "MEMR 12" in line 10.

The crucial point here is that the subroutine *prong* has a variable of its own – the parameter *size* – which is passed into it from the main program and then plays a role within the subroutine. This variable is an *integer* variable, so it requires only one storage location in memory, and what line 5 does is to *claim that memory space for the subroutine*. Hence "MEMC 12 1" effectively tells the *Turtle Machine* to set aside one memory location as local storage for the subroutine. But what is the "12" doing here? To answer that, we need to look inside the *Turtle Machine*'s memory as the subroutine is running, which we do by appending one line to it, as follows:

```
def prong(size):
    forward(size)
    blot(size/20)
    back(size)
    dump()
```

What the "dump()" instruction does is to *dump* the current contents of the *Turtle Machine*'s memory into the Memory table which is displayed in the "Memory" tab (just to the right of the "PCode" tab at the bottom right of the *Turtle System* window). And because this is occurring *while the subroutine is running*, we get to see the memory as it is at that point, as pictured below.


Turtle Programming System 15.45 beta (c) Peter Millican, Hertford College, Oxford 2000-2024 (Machine mode)
LANGUAGE | File Edit View Tabs Compile Run Options Examples
PYTHON
Function with parameter
TRACE ON
RUN
X 500 Y 500 Direction 0/360 Thickness 2 Colour
Display: Memory Stack Memory Heap Variables Table
Show Current State
1 py
Canvas Output Help1 Help2 Usage Comments Syntax Vars/Subs PCode Memory

```
def prong(size):
    forward(size)
    blot(size/20)
    back(size)
    dump()

for count in range(360,0,-1):
    randcol(10)
    prong(count+100)
    right(61)
```

MStack	0	1	2	3	4	5	6	7	8	9
00	^turtle	^keybuf	^file 1	^file 2	^file 3	^file 4	^file 5	^file 6	^file 7	^file 8
	14	200000	0	0	0	0	0	0	0	0
10	^file 9	^file 10	^subr 1	^fnbase	TURTLE	turtx	turty	turtd	turta	turtt
	0	0	21	0	6	500	500	299	360	2
			prong							
20	turtc	count	size							
	9849600	1	101							
			e							

MHeap	0	1	2	3	4	5	6	7	8	9
200000										
	200034	200003	200003	0	0	0	0	0	0	0
200010										
	0	0	0	0	0	0	0	0	0	0
200020										
	0	0	0	0	0	0	0	0	0	0

Identifier	Context	Index	Array	Type	Reference	Line Scope	PCode Scope
turtx	(0) PROGRAM	15	-	Integer	global	1 to 10	12 to 25
turty	(0) PROGRAM	16	-	Integer	global	1 to 10	12 to 25
turtd	(0) PROGRAM	17	-	Integer	global	1 to 10	12 to 25
turta	(0) PROGRAM	18	-	Integer	global	1 to 10	12 to 25
turtt	(0) PROGRAM	19	-	Integer	global	1 to 10	12 to 25
turtc	(0) PROGRAM	20	-	Integer	global	1 to 10	12 to 25
count	(0) PROGRAM	21	-	Integer	global	1 to 10	12 to 25
size	(1) prong	1	-	Integer	value	1 to 5	4 to 11

You will see that memory cell 12 is highlighted, and currently contains the value "21". In the memory display it also carries two annotations, "^subr 1", which tells us that this cell contains the "memory pointer" for subroutine number 1, and "prong", which helpfully gives us the name of this subroutine. *But note that as far as the Turtle Machine is concerned, neither of these annotations makes any difference at all – it is just an ordinary memory location containing the number 21*.

When a program is compiled, the *Turtle System* puts aside one memory location for each subroutine within a program, to be used as a memory pointer. The current program has only one subroutine, so it has just one such pointer, at location 12. Then what the PCode instruction "MEMC 12 1" (at line 5) does is to *claim one memory location as storage for the subroutine whose pointer is at location 12*. This sets aside space in memory for the local variable "size", which is needed by that subroutine. Notice also that memory location 12 contains the value 21: this points to the memory location *immediately below* the relevant subroutine's variable space. In the current case, we can see that the local variable "size" – variable 1 within the subroutine – is being stored at memory location 22 (because 22 = 21 + 1). The value of "size" is shown as 101, because that is the value it had at the last "dump()" instruction. But if we now click on the "Show Current State" button at the top right of the "Memory" tab, we will get to see instead the memory state *after the program has terminated*, with a significant change to both of the highlighted cells. Now cell 12 shows as containing 0 (rather than 21). This is the effect of "MEMR 12" in PCode line 10, just before the subroutine returns. This *releases* the memory space that had earlier been *claimed* by "MEMC 12 1", restoring things to the state that had previously obtained. Hence memory location 22 is no longer being actively used by the subroutine (and accordingly the cell is shown red in the memory display rather than green, with the "size" annotation replaced by "?").

The PCode lines 7 to 9 within the subroutine should now be fairly straightforward to understand:

```
7     LDVV 12 1  FWRD
8     LDVV 12 1  LDIN 20  DIVR  BLOT
9     LDVV 12 1  BACK
```

"LDVV 12 1" loads onto the Stack *the first local variable of the subroutine whose pointer is at memory location 12* – in the current case, this means the contents of location 22 (i.e. the local "size" variable). Then "FWRD" moves the *Turtle* forward by that amount, corresponding to the Python instruction "forward(size)". "LDVV 12 1 LDIN 20" again loads the current value of "size" onto the Stack, followed by the integer 20. "DIVR" then performs rounded division on these two values, leaving the result on the Stack, after which "BLOT" draws a blot, corresponding to "blot(size/20")". Finally "LDVV 12 1 BACK" corresponds straight-forwardly to "back(size)".

All that remains is to explain how the parameter value *size* is set up when the "prong" subroutine is called. Here is the relevant code within the main program:

```
20      LDVG 21  LDIN 100  PLUS  SUBR 4
```

and here is the relevant code at the beginning of the subroutine:

```
6       LDAV 12 1  LDIN 1  ZPTR  STVV 12 1
```

PCode line 20 probably looks relatively familiar by now. "LDVG 21" loads the value of *global* memory location 21 onto the Stack – this corresponds to the *global* variable "count". Then "LDIN 100" loads the integer 100 onto the Stack, "PLUS" adds the two together (leaving "count+100" on the Stack), and "SUBR 4" calls the "prong" subroutine – all this fits precisely with the instruction "prong(count+100)". *Note that when the subroutine is called, the appropriate parameter value "count+100" is already on the Stack*.

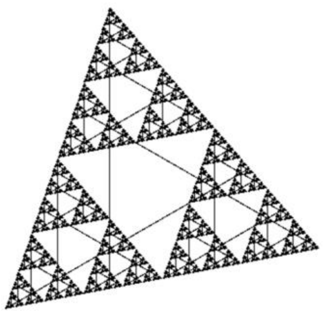
PCode line 6 – which initialises the subroutine's local variables – is far less familiar. "LDAV 12 1" loads *the memory address* of the first local variable (i.e. 22). Then "LDIN 1" loads integer 1, and "ZPTR" sets to zero a number of memory locations, taking the second Stack value as the starting address, and the top Stack value as the number of locations to be zeroed. In the current case, this just stores zero into memory location 22. Then "STVV 12 1" stores the value which is left at the top of the Stack – i.e. the parameter value "count+100" – into the subroutine's first memory location. So this is how the subroutine's parameter is set. All very well, but you might well consider that the instructions "LDAV 12 1 LDIN 1 ZPTR" have in this case been entirely pointless, setting to zero a single memory location which was then immediately filled with a parameter value. *True!* And if the compiler were attempting to optimise its PCode, these redundant instructions would be removed. But to keep things simple, the *Turtle System*'s compiler always starts any subroutine by zeroing all of the local memory locations.

2. Recursive Triangles

The "Recursive triangles" program generates the intricate pattern shown at the right here, using a function which *calls itself* (which is what we mean by calling it "recursive". For explanation of how this works from a programming point of view, see the document "Turtle Python 3 – Introducing Recursion". That document ends by briefly looking "under the bonnet" of the *Turtle Machine*, to see what is happening while an adapted version of the program (using a FOR loop) is executed. We'll do the same here, but this time looking closely at the PCode that it produces. Here is the adapted program:

```
def triangle(size):
    if (size>1):
        for count in range(3):
            forward(size)
            triangle(size/2)
            right(120)

movexy(-100,150)
triangle(256)
```



The main program starts by simply moving the *Turtle* left by 100 units and down by 150 units – this is just to ensure that the eventual pattern will be centrally placed on the Canvas. Then it calls the "triangle" function with the parameter value 256 – this becomes the initial value of local variable "size" within the program. The "triangle" function itself first tests whether "size" is greater than 1, and if not, it simply terminates (this test is to prevent an infinite recursive loop). Assuming that "size" is indeed greater than 1, however, we get a FOR loop with variable "count" taking the values 0, 1 and 2 in turn, and for each of these values, the *Turtle* moves forward by "size" units, drawing a line as it goes, then calls "triangle" with the parameter value "size/2", and finally turns right by 120 degrees.

Let us now go line by line through the entire PCode for the program, excluding as usual the first two lines which initialise the memory and set up the *Turtle* attributes and Canvas etc.

```
3      JUMP 20
```

This jumps over the code for the "triangle" subroutine, to start the main program at PCode line 20.

<u>Subroutine "triangle"</u>

```
4      PSSR 1
```

As before, this *pushes* the number of the current subroutine (i.e. 1) onto the *Subroutine Register Stack*.

```
5      MEMC 12 2
```

This claims 2 memory cells for the current *instance* of the subroutine whose memory pointer is at location 12 – one for the parameter "size", and one for the local variable "count". In this program, there can be up to nine instances of the "turtle" subroutine running simultaneously, each with its own 2 local memory cells.

```
6      LDAV 12 1  LDIN 2  ZPTR  STVV 12 1
```

When the subroutine is entered, the value of the parameter "size" is at the top of the Stack. The final instruction of this PCode line (i.e. "STVV 12 1") stores that value into the first local variable location (of the subroutine whose memory point is at location 12 – i.e. the "triangle" subroutine). But before doing this, "LDAV 12 1" loads onto the Stack the address of the first local variable, then "LDIN 2" loads the number of local variables, and finally "ZPTR" zeroes all of the relevant memory locations. This standardly happens with any *Turtle System* subroutine: the local variables are all initialised to zero, after which the parameter values are copied from the Stack into the relevant memory locations.

```
7      LDVV 12 1  LDIN 1  MORE  IFNO 19
```

This loads the first local variable (i.e. "size") onto the Stack, then loads the integer 1, after which "MORE" tests whether the former is greater than the latter (i.e. is size>1?). If not, then a jump is made to PCode line 19, which terminates the "triangle" subroutine. This corresponds to the program line "if (size>1)".

```
8      LDIN 3  LDIN 0  SWAP  LDIN 1  ROTA
```

This line sets up the Stack for the FOR loop, with 0 (the initial value) at the top, then 1 (the step value) then 3 (the terminal value). Because the shorthand syntax was used – "for count in range(3)" rather than "for count in range(0, 3, 1)" – the "3" gets onto the Stack first, and that's why "SWAP" is needed to put the three parameters in the appropriate order. The rest of the PCode structure of the FOR loop follows exactly the same pattern as in §3 of "Turtle Machine 2 – More Looping Structures", except that "STVV 12 2" replaces "STG 19" because the second local variable of the subroutine (whose memory pointer is at location 12) is being used for counting instead of a global variable.

```
9      DUPL  STVV 12 2  PICK 3  PICK 3  DUPL  IFNO 17
10     LDIN 0  MORE  IFNO 12  LESS  IFNO 18
11     JUMP 13
12     MORE  IFNO 18
```

Now comes the body of the FOR loop, with the compiled versions of "forward(size)", "triangle(size/2)" and "right(120)". Note here that "size" is the *first* local variable of the subroutines whose memory pointer is at location 12, so the command to load it onto the Stack is "LDVV 12 1":

```
13     LDVV 12 1  FWRD
14     LDVV 12 1  LDIN 2  DIVR  SUBR 4
```

Here “SUBR 4” is the recursive call to the “triangle” subroutine, which comes after (the rounded value of) size/2 has been placed on the Stack. The body of the loop ends with “right(120)”:

```
15      LDIN 120  RGHT
```

Following the loop body, we come to three lines of PCode that end the FOR loop, and again these are exactly parallel to what we saw in §3 of “Turtle Machine 2”, except that “LDVV 12 2” replaces “LDVG 12”:

```
16      DUPL  LDVV 12 2  PLUS  JUMP 9
17      DROP  DROP  DROP
18      DROP  DROP
```

With the FOR loop completed, the “triangle” subroutine terminates in the usual way, returning control to the point from which it was called:

```
19      MEMR 12  PLSR  RETN
```

Main program

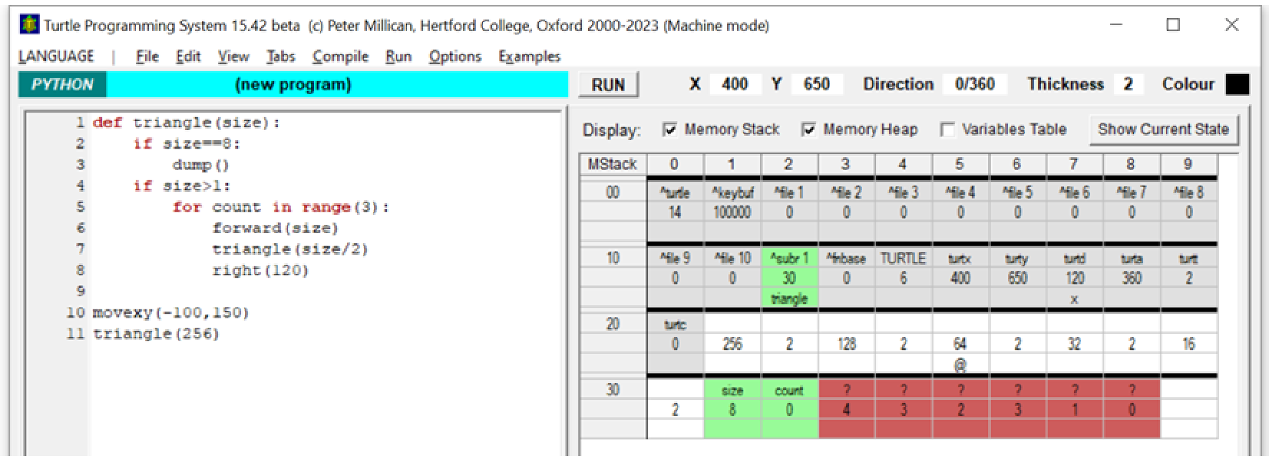
The main program follows the subroutine within the PCode (as it does within the Python text), but it runs first, because line 20 is the destination of the jump from PCode line 3. This line performs the instruction “movexy(-100,150)”:

```
20      LDIN 100  NEG  LDIN 150  MVXY
```

And finally the program ends after the subroutine call “triangle(256)”:

2.1 Claiming and Releasing Memory

The “Recursive triangles” program demonstrates the vital importance of the PCode commands “MEMC” and “MEMR”, which respectively *claim* memory for the subroutine when it starts, and *release* that memory when the subroutine finishes. We can illustrate this using an image from the document “Turtle Python 3 – Introducing Recursion”, which shows a snapshot of the memory display as it is while the “triangle” subroutine is running with the “size” parameter equal to 8 (which is achieved by performing the “dump()” instruction at that point, as can be seen in lines 2-3 of the Python code displayed in the image):


Turtle Programming System 15.42 beta (c) Peter Millican, Hertford College, Oxford 2000-2023 (Machine mode)
LANGUAGE | File Edit View Tabs Compile Run Options Examples
PYTHON
(new program)
RUN

X 400 Y 650 Direction 0/360 Thickness 2 Colour

Display: [✓] Memory Stack [✓] Memory Heap [ ] Variables Table

```
 1 def triangle(size):
 2     if size==8:
 3         dump()
 4     if size>1:
 5         for count in range(3):
 6             forward(size)
 7             triangle(size/2)
 8             right(120)
 9
10 movexy(-100,150)
11 triangle(256)
```

MStack	0	1	2	3	4	5	6	7	8	9
00	^turtle	^keybuf	^file 1	^file 2	^file 3	^file 4	^file 5	^file 6	^file 7	^file 8
	14	100000	0	0	0	0	0	0	0	0
10	^file 9	^file 10	^subr 1	^fnbase	TURTLE	turtx	turty	turtd	turta	turtt
	0	0	30	0	6	400	650	120	360	2
			triangle					x		
20	turtc									
	0	256	2	128	2	64	2	32	2	16
						@				
30		size	count	?	?	?	?	?	?	
	2	8	0	4	3	2	3	1	0	

When the “triangle” subroutine is called for the first time (from line 11 of the program as shown), with a parameter value of 256, “MEMC 12 2” claims two memory locations for this instance of the subroutine – the first of these memory locations will be used to store the parameter “size”, and the second to store the FOR loop variable “count”. Before that call, the last location in memory that is currently used is address 20 (storing the *Turtle* colour attribute “turtc”, as can be seen on the image). So “MEMC 12 2” then claims locations 21 and 22 for this first instance of the “triangle” subroutine. We can see the traces of this in the memory display, with location 21 holding 256 (the “size” parameter) and 22 holding 2 (the value of the “count” variable at the point when the snapshot was taken).

The next call of the "triangle" subroutine is at line 7 of the displayed Python code – this is the first *recursive* call, with a parameter value of 128. And this time, because memory locations 21 and 22 have already been claimed, "MEMC 12 2" serves to claim the next two available locations, 23 and 24 – again, we can see the traces of this in the memory display, with location 23 holding 128 (the value of "size") and 24 holding 2 (the value of "count" at that point). *Multiple calls to the same subroutine can work correctly without mutual interference only because each instance of the subroutine is able to claim its own "private" memory locations, which are then reserved for its own use until that instance terminates*.

But how does the *Turtle Machine* know where to find the appropriate variables for the *current* instance of the "triangle" subroutine – the one whose PCode commands are currently executing? The key is the subroutine memory pointer at location 12, which points to the memory location just below the current instance's variables (i.e. the memory location which was the last claimed cell *before* the current instance of the subroutine claimed its memory). As shown in the image, we can see that location 12 is holding the value 30, which means that the relevant memory locations are 31 and 32 – these are highlighted in green within the memory display, and labelled "size" and "count" respectively, to show that they are the currently active values. (But note that the highlighting and labelling are done by the *Turtle System* interface to make things visually clear for the user, and have nothing to do with the operation of the *Turtle Machine*, which simply takes those as the relevant locations because location 12 stores the value 30 at this point.) So at the point when this snapshot was taken, the active instance of the "triangle" subroutine is the one that has parameter value 8, using locations 31 and 32. And at this point, memory locations 21 to 32 are all reserved – they will all come into play in due course, when control is returned to the relevant instances of the subroutine. Memory locations 33 to 38, however are *not reserved*, even though we can see that they have been used by the program (because they are highlighted and displayed). Locations 33 and 34 will be claimed when the next recursive call of "triangle" is made (with "size" equal to 4). Then locations 35 and 36 will be claimed when the recursive call after that is made (with "size" equal to 2). And locations 37 and 38 will be claimed when the ultimate recursive call is made (with "size" equal to 1). But note that for the memory to be used efficiently, *it is vital that each instance of the subroutine releases its private memory space as it terminates, so that the memory space can be reused*. In the current example, this is achieved when the PCode instruction "MEMR 12" executes just before the subroutine ends. This tells the *Turtle Machine* to release the memory concerned, and to return to the memory allocation which existed just before the current instance of the subroutine started executing (i.e. before the "MEMC 12 2" instruction was executed).

To achieve this reinstatement of the previous memory allocation, the *Turtle Machine* standardly uses a "Memory Control Stack", which keeps track of the historic values of the "Memory Stack Top Pointer" (i.e. the number of the top memory location that is currently claimed) as each currently operative subroutine instance began, so that the relevant value can be restored when the subroutine instance ends. (Note that a *stack* structure is required here, because it needs to operate on a "last-in-first-out" basis: at any point, the first of the currently operative subroutines to terminate will be the one that started last.) This operation of the Memory Control Stack goes on "behind the scenes" as memory is claimed and released, and is not shown in the Trace or Memory displays. But the "Compile" menu contains an option to compile the code *without using* this separate Memory Control Stack, in which case the relevant control will be done explicitly on the main Evaluation Stack, using the PCode instructions "LDMT" and "STMT" (which respectively load and store the Memory Stack Top Pointer). This is less efficient than the default operation, but is instructive if you want to be able to see in the Trace display exactly how the memory is managed.

3. Returning a Function Value

It is common to refer to all Python subroutines as "functions". But more strictly, there is a distinction between two different kinds of subroutines. Using this terminology, a *function* is *a subroutine that returns a value*, while a *procedure* is *a subroutine which does not return a value*. So far, all of the subroutines that we have looked at have been *procedures* rather than *functions* – they performed operations, but they did not return a value to the code that called them.

Functions are very familiar in programming, but most of these are "native" within a programming language – for example Turtle Python has many native functions of which the following are just a sample:[1]

abs	returns the absolute value of a number, e.g. abs(-5) = 5
chr	returns the character corresponding to an ASCII code, e.g. chr(65) = 'A'
hex	returns the hexadecimal string of a number, e.g. hex(cyan,6) = '00FFFF'
int	returns the numerical value of a string, e.g. int("0xFFFF") = 65535
max	returns the maximum of two values, e.g. max(24,36) = 36
ord	returns the ASCII code corresponding to a character, e.g. ord("A") = 65
pixcol	returns the colour of a pixel on the Canvas, e.g. pixcol(500,500) = 65535 (if it's cyan)
randint	returns a random integer within an inclusive range, e.g. randint(1,10) = 7 (say)
read	returns a string read from the keyboard (after RETURN is pressed)
rgb	returns an RGB numeric code for a native *Turtle* colour, e.g. rgb(28) = 65535
str	returns the string representation of an integer, e.g. str(123) = '123'
time	returns the time elapsed since the program started (in milliseconds)

However, it is entirely possible to write your own functions in Python, and these become extremely important in more sophisticated programs. To illustrate, however, we take here a relatively simple function, albeit one that is *recursive*. This is taken from another program in the menu "Examples 2 – functions and simple recursion", namely "Recursive factorials", but here we simplify by calling the function with just a single value, rather than using a FOR loop to count through a range of values:

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)

print(factorial(7))
```

The function takes a single parameter "*n*", which it tests for equality with 0, returning the value 1 if *n* is equal to 0 – thus factorial(0) gives the result 1. If *n* is not equal to 0 however – and here we assume that *n* is a *positive* integer – then the factorial function itself is called to find the value of factorial(*n*–1), and this value is then is multiplied by *n* to yield the returned value of factorial(*n*).[2] After seven such recursive calls (all the way down to *n*=0), the value of factorial(7) turns out to be 7x6x5x4x3x2x1x1 = 5040. Note that for all this to work, we not only have to be able to pass a value *into* the function (as when the parameter 7 is used within the main program), but also, the function must be able to pass a value back to the code that calls it (as when the value 5040 is printed). Let's now look at the PCode which makes this possible.

```
3    JUMP 13
```

This jumps to the main program, in the (by now) familiar way. Then the next three lines start off the "factorial" subroutine:

```
4    PSSR 1
5    MEMC 12 2
6    LDAV 12 1  LDIN 2  ZPTR  STVV 12 2
```

But you might well feel that there is something odd here, because the subroutine only appears to have one local variable – the parameter "n". And yet the last two lines indicate space being reserved for *two* local variables (by "MEMC 12 2"), which are then both zeroed (by "LDAV 12 1 LDIN 2 ZPTR"), after which the

[1] Here the mathematical functions that handle "real values" (e.g. those concerned with trignonometry or logarithms) are not mentioned, because they involve various unfamiliar complications that are irrelevant here. To see these in action, take a look at the "Mathematical functions" program within the submenu "Examples 3 – further commands and structures". That submenu also contains programs illustrating string functions, both native and user-defined.

[2] It may seem unnatural that 0! is equal to 1, but this makes good sense precisely because of the rule that *n*! is equal to (*n*–1)! multiplied by *n*, and hence 1! should be equal to 0! multiplied by 1.

parameter value (from the Stack) is apparently stored in the *second* of them (by "STVV 12 2"). This puzzle is solved if we consult the *Turtle System*'s "Variables" display (in the "Vars/Subs" tab, but also copied at the bottom of the "Memory" tab). This shows that there is a local variable called "return" with index 1, and also a local variable called "n" with index 2. So the compiler has worked out that this subroutine – precisely because it is a *function* that returns a value – needs to have an additional reserved memory location in order to store that return value.[3] Continuing now with the PCode of our function:

```
    7       LDVV 12 2  LDIN 0  EQAL  IFNO 10
```

This loads the "n" variable and compares it with zero. If they are the same, …

```
    8       LDIN 1  STVV 12 1
    9       JUMP 12
```

then the value 1 is stored in the "return" variable, and we jump to line 12 which terminates the function. But if the "n" variable is greater than zero, …

```
    10      LDVV 12 2  LDVV 12 2  LDIN 1  SUBT  SUBR 4
```

The value of "n" is loaded again twice, 1 is subtracted, and the "factorial" subroutine is called (recursively – because this is from within the subroutine itself) to obtain the value of factorial(*n*–1).

```
    11      LDVV 13 1  MULT  STVV 12 1
```

Now we apparently load the first variable of the subroutine whose memory pointer is at location 13 (which seems puzzling, for there is no other subroutine!). Then we multiply that value by the value of "n" (which has been sitting on the Stack meanwhile). For this to make sense, "LDVV 13 1" has to somehow be giving us the value of factorial(*n*–1) – and indeed this is what is happening, as we shall see shortly.

```
    12      LDVG 12  STVG 13  MEMR 12  PLSR  RETN
```

This line terminates the function. But before releasing its memory, notice that *global memory location 12* – which is the memory pointer for the current subroutine – *is copied into memory location 13* – which, if we look in the "Memory" display – is annotated as "^fnbase". So now we can understand what's going on. When any Python program that involves a subroutine is run, one memory location – in this case, at address 13 – is set aside as a "function base pointer", to store temporarily the memory pointer for a terminating function. Then just before any function (i.e. a subroutine that returns a value) terminates, its own memory pointer – in this case, memory location 12 – is copied into that function base pointer. This means that even after the function has released its own memory, and after its own memory pointer has reverted to 0 (or to some other previous value, if more than one instance was running simultaneously), the "return" variable of that function can still be picked up using the function base pointer, just as though it was the first local variable of a corresponding subroutine (i.e. in this case, with "LDVV 13 1").

```
    13      LDIN 7  SUBR 4
```

This is the first line of the main program, to which we jumped at PCode line 3. This simply loads the parameter value 7 and calls the "factorial" function, just as would be expected from the Python code. Now it has to print the value, which it gets hold of exactly as we saw above, using "LDVV 13 1":

```
    14      LDVV 13 1  ITOS  WRIT  LSTR 2 #13 #10  WRIT  HCLR
    15      HALT
```

"ITOS" converts an integer – here the value 5040 – to a string, i.e. the four-character sequence "5040", which is put temporarily on the *Turtle Machine*'s "Heap" in memory, after which "WRIT" prints it to the Console and Output tab. Then "LSTR #13 #10" loads the two-character sequence consisting of control codes #13 and #10 (traditionally called "carriage return" and "line feed") onto the Heap, and this again is printed, generating a new line. Finally, "HCLR" cleans up the Heap by removing the now-redundant strings, and the program halts.

[3] Functions in the programming language Pascal automatically have a variable "result", which can be used within the function, and whose final value – when the function terminates – is returned as the value of the function.