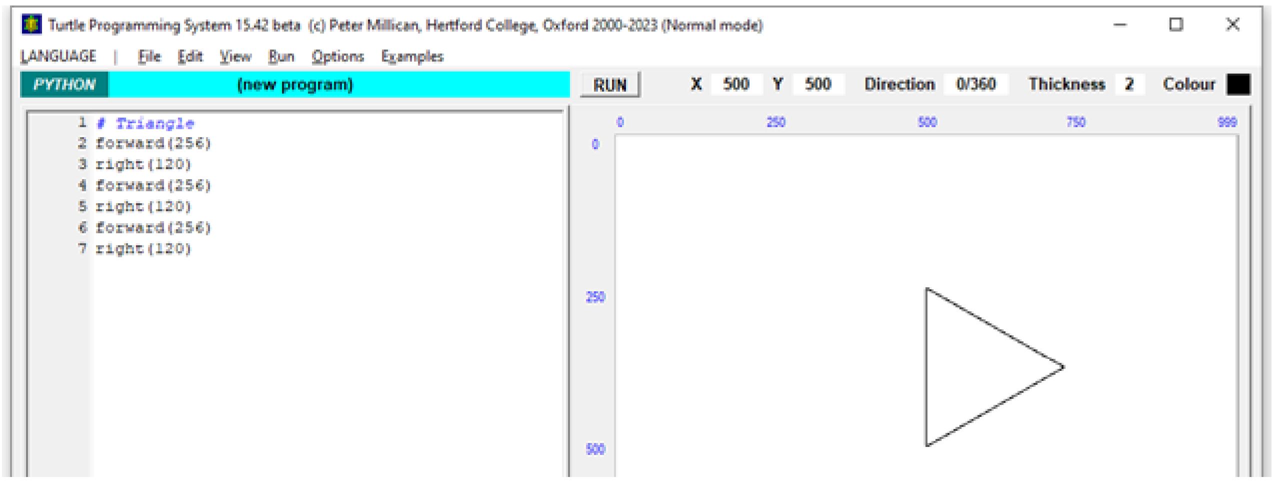
Turtle Python 3 – Introducing Recursion

1. A Triangle Program and Function

In the "Spirals and Shapes" document, we developed a program similar to the following one, which draws an equilateral triangle whose sides are length (256):

```
forward(256)
right(120)
forward(256)
right(120)
forward(256)
right(120)
```

Importantly, after this program finishes running, the *Turtle* will be back in exactly the place where it started, with both its X and Y coordinates shown as "500" just to the right of the "RUN" button:



Turtle Programming System 15.42 beta (c) Peter Millican, Hertford College, Oxford 2000-2023 (Normal mode)
LANGUAGE | File Edit View Run Options Examples
PYTHON
(new program)
RUN
Colour
0
250
500
750
999
0
250
500

X 500 Y 500 Direction 0/360 Thickness 2

```
# Triangle
forward(256)
right(120)
forward(256)
right(120)
forward(256)
right(120)
```

We also saw how such a sequence of commands can be put inside a *function*, with the length of the side as a parameter called *size*. If we convert the program above accordingly, we obtain:

```
def triangle(size):
    forward(size)
    right(120)
    forward(size)
    right(120)
    forward(size)
    right(120)

triangle(256)
```

Here the function is defined in the first 7 lines, beginning with the word `def` and including all the indented lines. Then the main program consists of the single instruction `triangle(256)`, which calls the *triangle* function with a parameter value of 256. This then produces *exactly* the same graphical effect as the previous program, as shown above.

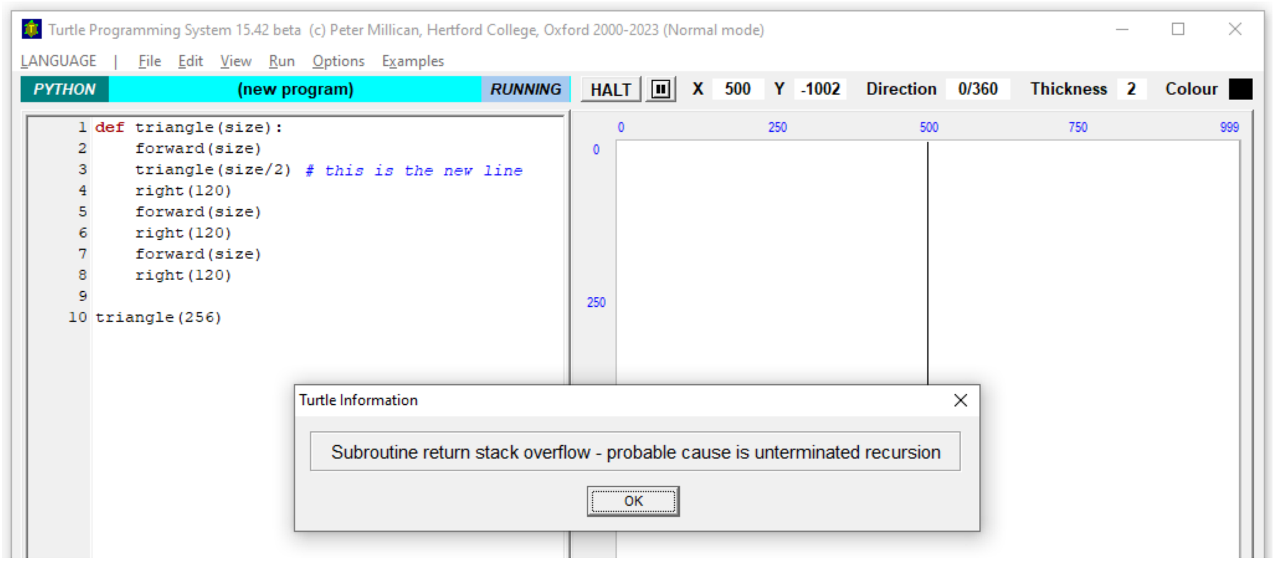
Why is changing the program like this useful? Well, one answer, as we saw in "Spirals and Shapes", is that our function is far more versatile than the instructions in the original program, because we can use it to draw triangles of lots of different sizes by varying the *size* parameter. But we're now going to explore a more exciting possibility that opens up once we have such a versatile function to play with.

2. Introducing Recursion – and an Instructive Error

Suppose, then, that we take our program and change it by adding a single line, as follows:

```
def triangle(size):
    forward(size)
    triangle(size/2) # this is the new line
    right(120)
    forward(size)
    right(120)
    forward(size)
    right(120)

triangle(256)
```

When we click on "RUN", the result will be initially disappointing:


Turtle Programming System 15.42 beta (c) Peter Millican, Hertford College, Oxford 2000-2023 (Normal mode)
LANGUAGE | File Edit View Run Options Examples
PYTHON
(new program)
RUNNING
HALT
X 500 Y -1002 Direction 0/360 Thickness 2 Colour
0 250 500 750 999
0
250
Turtle Information
OK

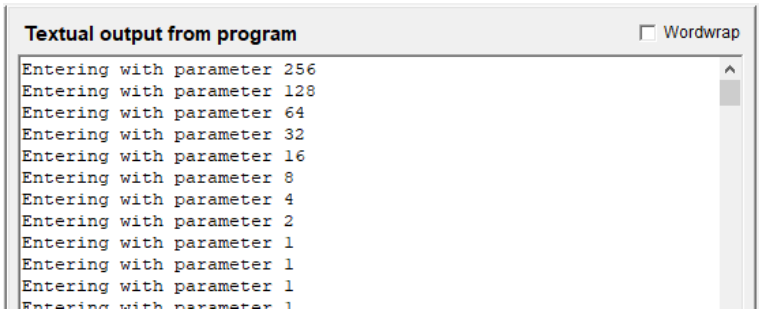
```
def triangle(size):
    forward(size)
    triangle(size/2) # this is the new line
    right(120)
    forward(size)
    right(120)
    forward(size)
    right(120)

triangle(256)
```

Subroutine return stack overflow - probable cause is unterminated recursion

To investigate what's going on, we can take advantage of the *Turtle System*'s ability to output text without affecting the graphical output of a program, by adding a line each at the beginning and end of the function:

```
def triangle(size):
    print('Entering with parameter',size)
    forward(size)
    triangle(size/2) # this is the new line
    ...
    right(120)
    print('Exiting with parameter',size)
```

This time the program will run for a much longer time before producing the error message, because it is outputting so much text to the Console. To view the entire output, click on the "Output" tab, at the bottom of the *Turtle* window just to the right of the "Canvas & Console" tab. Scrolling up to the top, you will see what is displayed here – showing that the function is being entered in turn with parameters 256, 128, 64, 32, 16, 8, 4, 2, 1, 1 again, and then repeatedly with a parameter of 1, on and on for ever. Why is this happening?



```
Entering with parameter 256
Entering with parameter 128
Entering with parameter 64
Entering with parameter 32
Entering with parameter 16
Entering with parameter 8
Entering with parameter 4
Entering with parameter 2
Entering with parameter 1
Entering with parameter 1
Entering with parameter 1
```

The explanation for the sequence 256, 128, … 2, 1 is very straightforward, because the fourth line of the function – `triangle(size/2)` – clearly calls the *triangle* function itself, with *half the current size parameter*. So when the function is entered with a value of 256, it first prints "Entering with parameter 256", then moves the *Turtle* forward by 256 units, then in effect issues the instruction `triangle(128)`. So now the function is entered with a value of 128, and it accordingly prints "Entering with parameter 128", moves the *Turtle* forward by 128, and in effect issues the instruction `triangle(64)`, and so on … Overall, the *Turtle* moves forward 256+128+64+32+16+8+4+2+1 units – a total of 511 units – which explains why we just see a vertical line disappearing off the top of the Canvas.

When we get to the *size* value of 1, we might expect that the *triangle* function will now be called with a parameter of 0.5, but this is not possible, because the *Turtle System* operates only with *whole numbers* (i.e. integers). So when *size* is equal to 1, the instruction `triangle(size/2)` instead calls the function with the *rounded* value of "1/2" – and this, being 0.5, rounds up to 1. So now we have in effect the instruction `triangle(1)` being issued again, and this then repeats on, and on, and on …

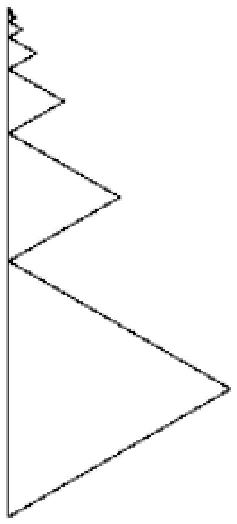
But the program does not carry on indefinitely, because although we see lots of new instances of the *triangle* function starting up, we never see any of them *finishing*: the output "Exiting with parameter …" never appears. And every new instance makes a demand on the *Turtle System*'s memory – in particular, on something called the "subroutine return stack", which keeps track of where each function call should *return to* once it has been completed. In these circumstances, it is obviously not possible for any practical system to keep an infinite number of functions running simultaneously. So eventually, the system detects an overload, and delivers the error message "Subroutine return stack overflow – probable cause is unterminated recursion". If you see this, it does not mean that the *Turtle System* is malfunctioning! What it means is that you have very likely just run a function which is *recursive* – i.e. calls itself – but without inserting an adequate *termination condition*. That is exactly the correct diagnosis in the present case.

3. A Termination Condition, and Successful Recursion

You might now be able to work out for yourself how the function needs to be changed to avoid this unterminated recursion: the most natural method is to avoid recursion when the *size* gets too small. And in fact there is no point anyway in trying to draw a triangle with a *size* parameter of 1, because at that stage we are dealing with a single pixel of the Canvas. So we change our recursive *triangle* program as follows, using a conditional (i.e. "if") instruction – and notice how all of the commands that are being made conditional on *size* being greater than 1 are indented below the "`if size>1:`" line:

```
def triangle(size):
    print('Entering with parameter',size)
    if size>1: # this avoids recursion when size is 1
        forward(size)
        triangle(size/2) # this is the recursive call
        right(120)
        forward(size)
        right(120)
        forward(size)
        right(120)
    print('Exiting with parameter',size)

movexy(-100,150) # move the starting point
triangle(256)
```

Now the recursion of the function will be limited, taking *size* parameters of 256, 128, 64, 32, 16, 8, 4, 2, and 1 in turn, but never attempting to draw anything – or to make any further recursive call – when the value of *size* is 1. Notice also that a new line has been inserted in the main program, which simply shifts the starting point of the big triangle 100 units to the left and 150 units down – this will make the pattern more central and avoid the initial vertical line extending off the top of the Canvas. When we run this new version of the program, we will see the pattern shown at the right here. And if we look at the "Output" tab, we will see that although the various *triangle* calls *started* with the parameter *size* being (in order) 256, 128, 64, 32, 16, 8, 4, 2, and finally 1, they *finished* in the opposite order, so they were effectively "last in, first out". Looking at the pattern, this indeed makes sense: the 256-triangle was started but only one-third drawn, then "put on hold" while the 128-triangle was started and one-third drawn, then that was put on hold while the 64-triangle was started and one-third drawn, and so on. Eventually the 2-triangle was started and one-third drawn, then – in effect – `triangle(1)` was called, but that did nothing more than print "Entering with parameter 1" and "Exiting with parameter 1". This exit released the hold on the 2-triangle, which thus became free to finish and exit. This likewise released the hold on the 4-triangle, and so on all the way to the 64-triangle, the 128- triangle and finally the 256-triangle.

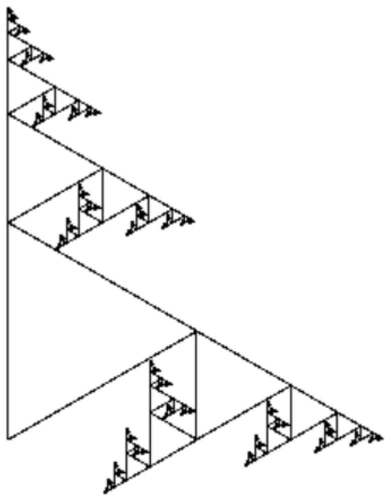


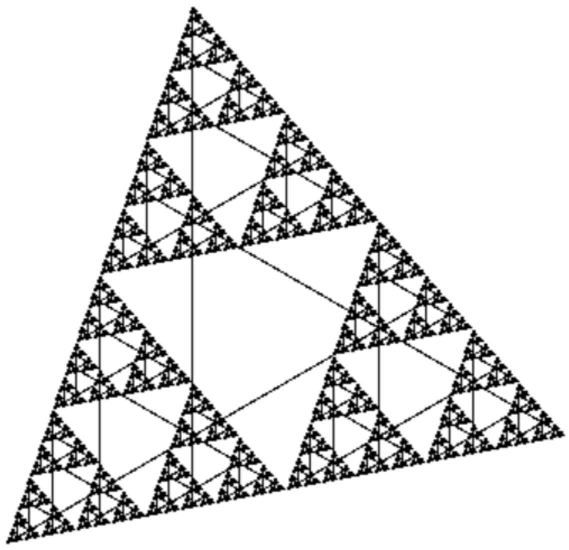
If you are familiar with Dr Seuss's character *The Cat in the Hat*, then you might find him a useful metaphor for how this all works. He wears a very big top hat on his head, under which is perched a smaller cat, *Little Cat A*. *Little Cat A* also wears a (relatively) big top hat, with *Little Cat B* perched on his head, and so on through *Little Cats C, D, E, F, G, and H*. Each *Cat* follows the instructions in the *triangle* function, using whatever parameter he has been given, with the instruction `triangle(size/2)` interpreted as meaning that he should take off his hat and wait while telling the *Little Cat* on his head to jump down and draw a triangle of half the size of his own. *The Cat in the Hat* starts off following the instructions with a parameter of 256, draws the first side of his triangle, then takes off his hat and tells *Little Cat A* to follow the same instructions with a parameter of 128. *Little Cat A* likewise draws the first side of his triangle (length 128), then takes off his hat and tells *Little Cat B* to follow the same instructions with a parameter of 64, and so on. Eventually we get to *Little Cat G* with a parameter of 2, who draws a side of length 2, takes off his hat and tells *Little Cat H* to follow the same instructions with a parameter of 1. At this point, when *Little Cat H* has just jumped off the head of *Little Cat G* to start his work, all of the other *Cats* are patiently waiting, having got only one third of the way through their respective triangles. But now *Little Cat H* looks at the instructions, and sees that because his parameter is 1 he doesn't actually have to draw anything at all. So his work is done, he jumps back onto the head of *Little Cat G*, who is now free to put his hat back on and continue drawing his triangle of side 2. One he finishes, he jumps back onto the head of *Little Cat F*, who is then free to put his hat back on and continue drawing his triangle of side 4, and so on.

We can now add a second and third recursive call (and remove the print statements, which if left in will from now on seriously slow down the program):

```
def triangle(size):
    if size>1:
        forward(size)
        triangle(size/2) # first recursive call
        right(120)
        forward(size)
        triangle(size/2) # second recursive call
        right(120)
        forward(size)
        triangle(size/2) # third recursive call
        right(120)

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

The pictures below show the result that we get with the first two recursive calls, and then with all three.





We seem to have come a very long way from the program we had at the end of §1 above, but all we have done is to add one conditional test – to see whether `size` is greater than 1 – and three recursive calls that are made if the test is positive. If you want to check out the main steps of this development again, you can follow it through by loading the four relevant programs from Examples menu 2, namely "Simple triangle", "Triangle function", "Triangle function with limit", and "Recursive triangles". The first three of these all produce an identical single triangle as output, while the last – by adding just the three recursive calls – produces the final image above, which looks remarkably like a so-called "Sierpinsky triangle" (to which we'll be returning in a future document). If we use a FOR loop within the triangle function, moreover – which we did near the end of the "Spirals and Shapes" document – we can produce this with a *single* recursive call:

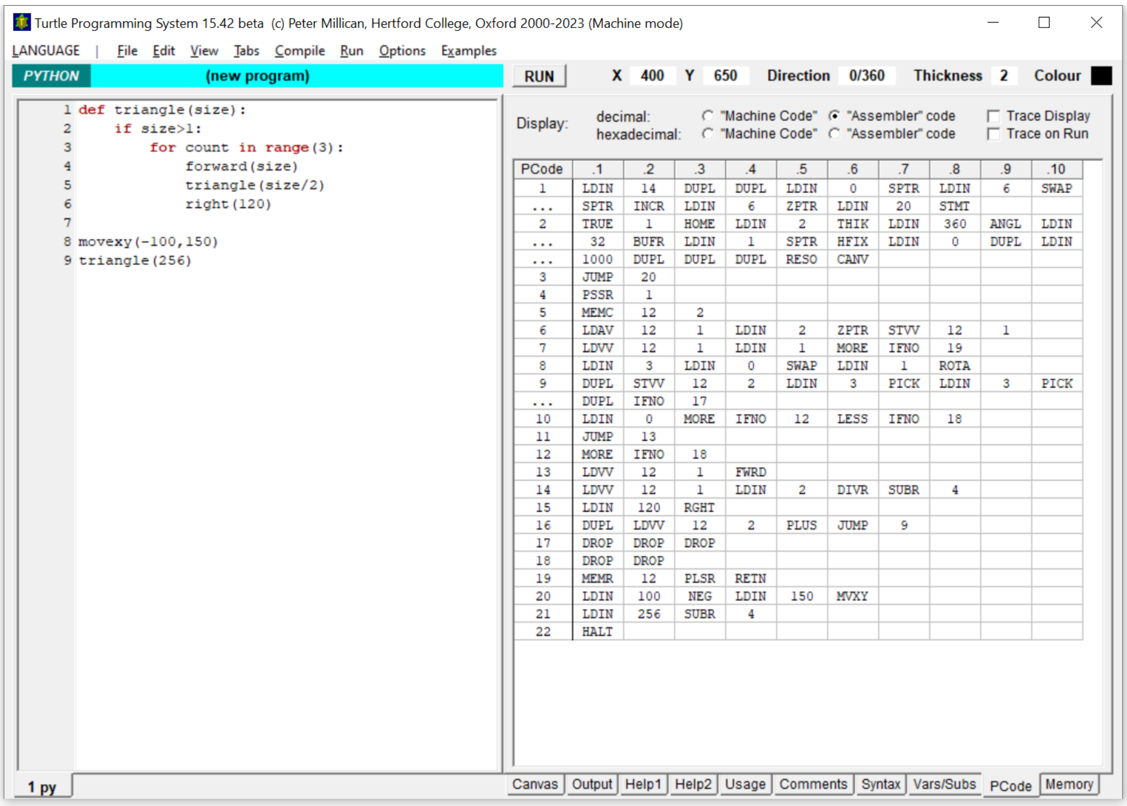
```
def triangle(size):
    if size>1:
        for count in range(3):
            forward(size)
            triangle(size/2) # just one recursive call
            right(120)

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

It seems remarkable that we can get such an intricate and beautiful pattern by adding just one line to a program that draws a simple triangle!

Appendix: A Peek at Memory

You might reasonably wonder how a computer manages to handle this sort of recursive program, because it seems that at one stage we have nine different versions of the *triangle* routine all running at the same time (in the Dr Seuss analogy, these are being executed by *The Cat in The Hat*, and *Little Cats A*, *B*, *C*, *D*, *E*, *F*, *G*, *and H*). How does the computer manage to keep track of nine different parameter values (respectively 256, 128, 64, 32, 16, 8, 4, 2, and 1), and nine different – constantly changing – *count* variables?

To gain some quick insight into this matter, and without getting seriously technical, we can take a peek "under the bonnet" of the *Turtle System*, to look at the workings of the *Turtle Machine* on which it runs. This is a *virtual machine* – in other words, a simulation of an imagined hardware chip – which is specially designed to be relatively easy to understand. And if we switch into Machine Mode (from the "View" menu), we can see both its "machine code" and memory contents:


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
1 py
Canvas Output Help1 Help2 Usage Comments Syntax Vars/Subs PCode Memory

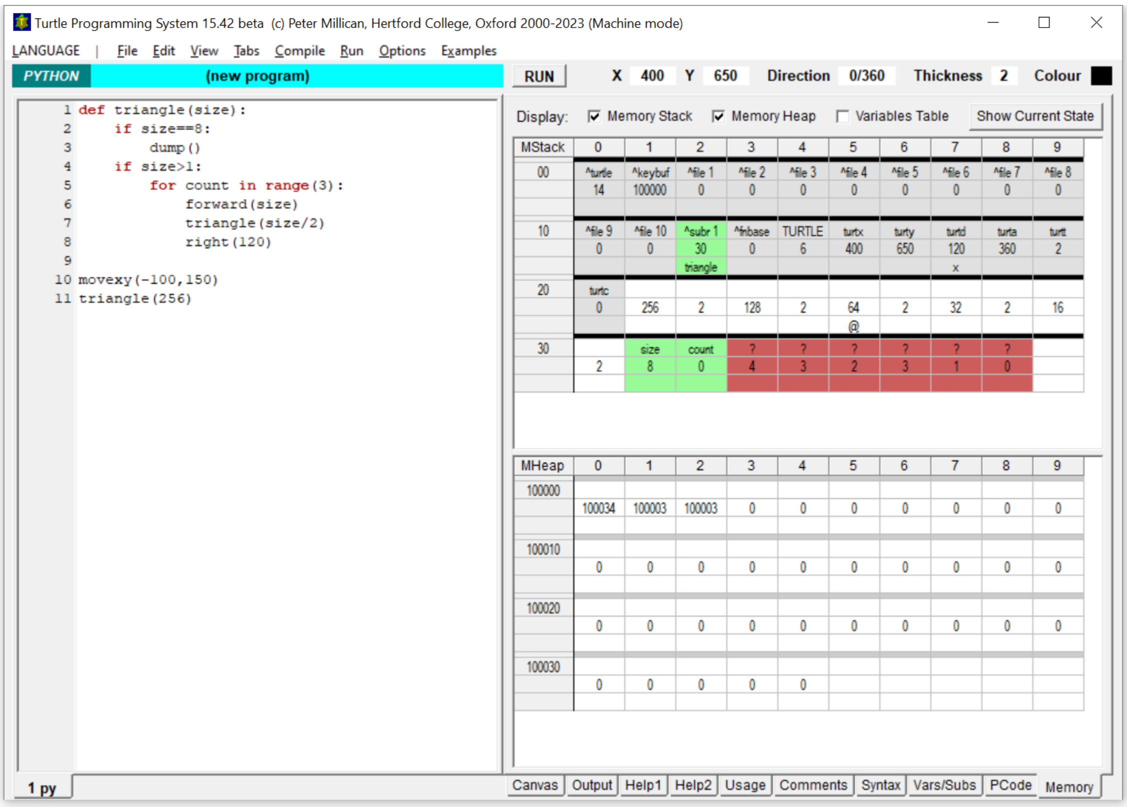
X 400 Y 650 Direction 0/360 Thickness 2 Colour

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

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

Display: decimal: "Machine Code" [✓ "Assembler" code]; hexadecimal: "Machine Code" "Assembler" code; Trace Display; Trace on Run

PCode	.1	.2	.3	.4	.5	.6	.7	.8	.9	.10
1	LDIN	14	DUPL	DUPL	LDIN	0	SPTR	LDIN	6	SWAP
...	SPTR	INCR	LDIN	6	ZPTR	LDIN	20	STMT		
2	TRUE	1	HOME	LDIN	2	THIK	LDIN	360	ANGL	LDIN
...	32	BUFR	LDIN	1	SPTR	HFIX	LDIN	0	DUPL	LDIN
...	1000	DUPL	DUPL	DUPL	RESO	CANV				
3	JUMP	20								
4	PSSR	1								
5	MEMC	12	2							
6	LDAV	12	1	LDIN	2	ZPTR	STVV	12	1	
7	LDVV	12	1	LDIN	1	MORE	IFNO	19		
8	LDIN	3	LDIN	0	SWAP	LDIN	1	ROTA		
9	DUPL	STVV	12	2	LDIN	3	PICK	LDIN	3	PICK
...	DUPL	IFNO	17							
10	LDIN	0	MORE	IFNO	12	LESS	IFNO	18		
11	JUMP	13								
12	MORE	IFNO	18							
13	LDVV	12	1	FWRD						
14	LDVV	12	1	LDIN	2	DIVR	SUBR	4		
15	LDIN	120	RGHT							
16	DUPL	LDVV	12	2	PLUS	JUMP	9			
17	DROP	DROP	DROP							
18	DROP	DROP								
19	MEMR	12	PLSR	RETN						
20	LDIN	100	NEG	LDIN	150	MVXY				
21	LDIN	256	SUBR	4						
22	HALT									

Here on the right, in the "PCode" tab, we see the machine code into which the Python program on the left has been *compiled* (i.e. translated). At the top of the tab, you'll also see that there is a "Trace" facility, which enables us to see what happens as the program is run, step by step. Without the recursive call, this program takes 139 steps in total; but with the recursive call, it takes altogether 449,413 steps! We won't try to delve into the mysteries of machine code here, though it's a very interesting topic in its own right, and the *Turtle System* is specially designed to facilitate such investigations. Instead, let us look at the *memory* of the virtual machine as it stands when a triangle of size 8 is being drawn for the very last time:


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
Show Current State
1 py
Canvas Output Help1 Help2 Usage Comments Syntax Vars/Subs PCode Memory

X 400 Y 650 Direction 0/360 Thickness 2 Colour

```
def triangle(size):
    if size==8:
        dump()
    if size>1:
        for count in range(3):
            forward(size)
            triangle(size/2)
            right(120)

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

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

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

MHeap	0	1	2	3	4	5	6	7	8	9
100000	100034	100003	100003	0	0	0	0	0	0	0
100010	0	0	0	0	0	0	0	0	0	0
100020	0	0	0	0	0	0	0	0	0	0
100030	0	0	0	0	0					

Here lines 2 and 3 of the program contain a conditional instruction to "dump" the current memory content into the "Memory" display whenever the *triangle* routine is called with *size* equal to 8. Thus when the program finishes, we are left with the very last snapshot of memory to be taken in that situation. On the right we can see two separate displays, one of the "Memory Stack" – here from locations 0 to 39 – and the "Memory Heap" – here from locations 100000 to 100039. The latter is almost empty, but the first three locations (100000 to 100002) contain the numbers 100034, 100003 and 100003 – these are the three *pointers* that keep track of the (circular) keyboard buffer discussed in the document on "Animation and User Input", and which here extends from location 100003 to 100034 inclusive (i.e. 32 locations altogether).

For the purpose of understanding how recursion works, however, we are much more interested in the Memory Stack display at the top, whose contents are listed below. (But _*don't worry at all*_ about not understanding all of this, which is inevitable – the details are for the inquisitive who want to delve deeper!)

Location 0:	14 – points to the *Turtle* attribute array
Location 1:	100000 – points to the keyboard buffer
Locations 2 to 11:	All 0, but would be used for file pointers if any files were open
Location 12:	30 – the subroutine pointer for subroutine 1 (i.e. the *triangle* function)
Location 13:	0 – the pointer for transfer for a function result (not relevant to this program)
Location 14:	6 – the number of attributes in the *Turtle* attribute array
Locations 15 to 20:	Current values of the 6 *Turtle* attributes: *turtx*, *turty*, *turtd*, *turta*, *turtt*, and *turtc*.
Locations 21 and 22:	256 and 2 – memory being used by first call of subroutine, i.e. `triangle(256)`
Locations 23 and 24:	128 and 2 – memory being used by second call of subroutine, i.e. `triangle(128)`
Locations 25 and 26:	64 and 2 – memory being used by third call of subroutine, i.e. `triangle(64)`
Locations 27 and 28:	32 and 2 – memory being used by fourth call of subroutine, i.e. `triangle(32)`
Locations 29 and 30:	16 and 2 – memory being used by fifth call of subroutine, i.e. `triangle(16)`
Locations 31 and 32:	8 and 0 – memory being used by sixth call of subroutine, i.e. `triangle(8)`
Locations 33 and 34:	4 and 3 – memory *previously used* by seventh call of subroutine, i.e. `triangle(4)`
Locations 35 and 36:	2 and 3 – memory *previously used* by eighth call of subroutine, i.e. `triangle(2)`
Locations 37 and 38:	1 and 0 – memory *previously used* by ninth call of subroutine, i.e. `triangle(1)`

Notice that the different calls of the *triangle* routine have each been allocated two memory locations, one for the *size* parameter and one for the *count* variable. The values in the odd-numbered locations from 21 to 37 are easily explained because the *size* parameters are respectively 256, 128, 64, 32, 16, 8, 4, 2 and 1. The even-numbered locations from 22 to 30 all contain 2 because this is the value of the *count* variable during the final iteration of a FOR loop that counts through the values 0, 1 and 2. Coming now to location 32, this is 0 because when our memory snapshot was taken, the *triangle* routine with *size* equal to 8 had only just been entered – the loop hadn't even started, so *count* had its default value of 0.

What of locations 34, 36, and 38? Notice that in the Memory Stack display all of the locations from 33 to 38 are shaded red and shown with a "?" above the numeric contents – this indicates that when the memory snapshot was taken, these memory locations were not "live". They therefore reflect the final values from *previous* (but now ended) calls to the *triangle* routine, in which the *size* parameter took the values 4, 2 and 1 respectively. In the first two of these cases, we can see that the *count* variable ended up equal to 3 (because in a FOR loop, termination takes place when the counting variable reaches the limit value). But in the last case, where *size* had the value 1, *count* (in location 38) remains with its default value of 0, because when *size* is not greater than 1, no loop takes place.

Finally, notice that when the snapshot takes place, the subroutine pointer at location 12 is equal to 30, so it is pointing just *before* the memory locations that are storing the *current values* of *size* and *count* (at locations 31 and 32). It is this pointer that enables the *Turtle Machine* to identify which specific copies of *size* and *count* are *currently* in use, thus keeping track of the nine different *triangle* versions.