Task One

# Introduction

1. t.forward(100) # Go forwards 100 units
2. t.left(45) # Turn left 45 degrees
3. t.forward(30) # Go forwards 30 units

Click on the “Task 1” button to load in the above program, and then click the “Run” button.

Here you are creating a new turtle, called **t**, then telling it to go forward 100 units, go left 45 degrees, and then go forward again by 30 units.

The text after the **#** is called a **comment**. This is English text that is ignored by Python. This is used to explain how the code works. It has no effect on the behaviour of the turtle.

Modify the above code to make the turtle go forward by 200 units, turn 90 degrees left, then go forward by another 100 units.

Other commands you can use to move the **t** are t.right and t.backward. These are used in exactly the same way as t.forward and t.left.

Change the code to make the turtle draw a square with sides of length 100.

# Numbers and variables

You are using a language called Python to tell the turtle what to do.

Python can also do other things, such as basic maths.

1. print 3
2. print 8+11
3. print 9-10
4. print ((87+3)/30\*(51-3\*9))/6

Type in the above code and click “Run”. You should see the results of these arithmetic operations in the results box.

+represents addition, - represents subtraction, \* represent multiplication, and / represents divide. These can be combined to make very long arithmetic expressions, just like in maths.

1. x = 5
2. print x
3. x = 8
4. print x
5. x = x + 1
6. print x
7. print 2\*x

Type in the above code and click “Run”.

Here, you are using a **variable**. Python is creating some storage space called **x** which is being used to hold a number. Notice how the print x statements all result in a different number being output.

Line 5 is

1. x = x + 1

This can be thought of as **the new value of x** is set to be **the old value of x** plus 1. This has the result of increasing the stored value of **x** by 1.

# Using numbers and variables

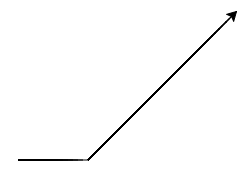
1. x = 70 # Set a variable ‘x’ to have value 100
2. t.forward(x) # Go forward ‘x’ units
3. t.left(45)
4. t.forward(x+40) # Go forward ‘x+40’ units again.

Click on “Task 1” and modify the code so it looks like the above. (Changes to be made have been highlighted). You do not need to copy the text after the #. Try modifying the value of x on line 4 and see how this affects the shape drawn by the turtle.

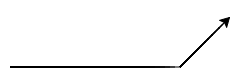
Notice how the second line is longer than the first

Try changing the code to make:

* The second line three times as long as the first line



* The first line 100 units longer than the second line.



Task Two

# A loop

1. for i in range(6): # Repeat the following lines 6 times.
2. t.forward(40)
3. t.left(45)

Click on the “Task 2” button to load in the above program, and then click the “Run” button.

Here we are using something called a **for loop**. Line 4 is saying “execute the following lines 6 times”. This results in the turtle going forward and left 6 times. At the moment it is drawing ¾ of an octagon.

Tweak the above code to make the turtle draw the whole octagon. Then modify the code to draw a square.

|  |  |
| --- | --- |
|  |  |

A for loop works by telling Python to run the next **block** of code some number of times. Any lines in this block are said to be **inside the loop**. Indentation is used in python to describe where blocks of code start and end. To indent a line of code, use either TAB or 4 spaces.

1. for i in range(4):
2. t.forward(40) # ‘inside’ the loop
3. t.left(45) # ‘inside’ the loop

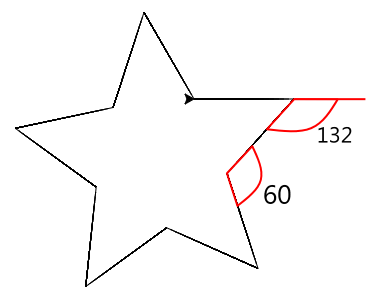
Here, lines 5 and 6 are part of the same block; they are both ‘inside’ the for loop. This means that both lines will be executed in turn as part of the for loop on line 4.

1. for i in range(4):
2. t.forward(50) # ‘inside’ the loop
3. t.left(45) # ‘outside’ the loop

Here, lines 5 and 6 are not part of the same block; only line 5 in ‘inside’ the for loop. This means only line 5 is executed as part of the for loop on line 4.

# Drawing a star

Using a for loop, write code to draw a star.



Angles have been marked on the diagram to help you.

# Drawing a spiral

This type of for loop must be written using the following **syntax:**

for *variable* in range(*number*):

*code to be repeated*

All italic text must be replaced with something that “fits” in that slot. In the example above, the slots were filled accordingly:

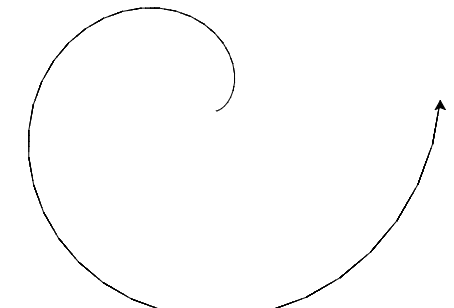
* *Variable* – **i**
* *Number*– **6**
* *Code to be repeated* – Lines 2-3. These lines must be indented, as explained above.

Every time the body of the loop is run, the value of *variable* will increase by 1. The value of *variable* starts at 0. The loop will stop after running *number* of times, meaning the maximum value of *variable* will be *number – 1****.***

To illustrate this, try the following code:

1. for i in range(10):
2. print i

As the loop is run more times, the value of *variable* increases. This can be used to draw a spiral.



In order to draw a spiral, the turtle will need to go forward by at increasing amount, then turn left by a small amount.

Using a for loop, write a program to draw a spiral, similar to above.

Hints:

* **t.forward(i)**, where **i** is the loop variable, will cause the turtle to move forward by an increasing amount.
* The turtle will need to turn left by a small amount after drawing each line. The more it turns each time, the tighter the spiral will become.

Task Three

# Procedures

At the moment, Python runs code line-by-line.

1. t.pencolor('red')
2. t.forward(60)
3. t.pencolor('blue')
4. t.forward(60)
5. t.pencolor('yellow')
6. t.forward(60)

This code will draw a red, blue, and then yellow line.

Sometimes we want to modify the order in which Python runs the lines.

1. def drawSquare(): # Define a new procedure, drawSquare
2. for i in range(4):
3. t.forward(40)
4. t.left(90)
6. drawSquare() # Call drawSquare

Click on the “Task 3” button to load in the above program, and then click the “Run” button.

Here, the code is **defining** a **procedure** called drawSquare on line 1. The following block of code (lines 2-4) then ‘belong’ to that procedure. It is then **calling** that procedure on line 6. This means that the block of code ‘belonging’ to drawSquare (the highlighted lines) is executed.

If you did not do task two, line 2 is saying ‘execute the following block (lines 3-4) of code 4 times’.

Defining a procedure tells Python **how** to do something (in this case, we are telling it how to draw a square). When a procedure is called it is telling Python **when** to do something. A procedure must be defined before it is used.

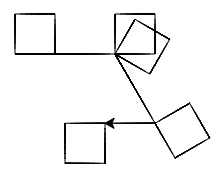
When defining a procedure, we use the keyword **def**, then give it a name (**drawSquare** in this case). To call the procedure, you need to type the produce’s name exactly as it appears in the definition (including any capitals), followed by parentheses.

The main benefits of using procedures are:

* Abstraction – You can now tell Python to ‘draw a square’ rather than ‘go forward and left 4 times’. You are telling Python **what** to do rather than **how** to do it.
* Code re-use – At any point in the program you can just write ‘drawSquare()’ and Python will draw a square. There is no need to type out the loop again.

Using drawSquare() and the t.forward, t.left, and t.right commands, draw 5 squares at various places.

The results of your drawing may look something like this:



# Writing your own procedure

Write a procedure to draw a letter of your choice. Use the skeleton code below if you are unsure how to structure it.

For simplicity, it would be a good idea to avoid letters with closed loops.

# define a procedure called ‘drawLetter’

# code to draw the letter goes here

drawLetter()

You can use t.setheading(x) to set the turtle’s heading to x.

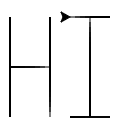
* t.setheading(0) corresponds to east
* t.setheading(90) corresponds to north



If you have time, write another procedure to draw another letter. Make sure to test each letter-drawing procedure separately. (i.e. call only drawA or drawB, do not call drawA then drawB)

Now that we have procedures to draw one or two letters, we can use them both to write a word!

By using your procedures, write two letters on the screen.



Make sure that each letter is oriented correctly. Use t.setheading(0) in between your letters to make sure the turtle is facing due east before drawing each letter.

In order to prevent the turtle from drawing a black line between each letter, t.penup() and t.pendown() can be used. As the name suggests, t.penup() will cause the turtle to ‘lift its pen’ meaning later commands will not draw lines. t.pendown() will do the opposite.

You can make each letter a different colour using t.pencolor commands, as demonstrated at the beginning of this task.

Task Four

# Random numbers

We can use python to generate random numbers for us.

1. from random import randint
2. print randint(0,10)

Type in this code and run it a number of times.

**randint** is a procedure that will evaluate to a random number. The first line is simply telling python where to find this procedure. We do not have to define it ourselves, as we are importing it.

**For this task, make sure you always have ‘**from random import randint**’ as the first line.**

What do the 0 and 10 represent?

# Infinite loops

In task two, you were using for loops, which tell python to ‘do this some amount of times’. By using an infinite loop, we can tell python to ‘do this forever’.

1. from random import randint
2. while True: # Infinite loop
3. t.forward(20)
4. angle = randint(-45,45) # ‘angle’ has a random value…
5. t.left(angle)

Click on the “Task 4” button to load in the above program, and then run it.

This turtle will draw wiggly lines forever. It moves forward a fixed amount each time, then turns by a random amount.

**WARNING: Never use ‘**write True:**’ without having at least one turtle movement command inside the loop. If you do this your browser will freeze.**

# If statements

The turtle will quickly go off screen. Ideally, we want the turtle to turn around if it is off screen. More precisely:

“If the turtle is off screen, turn it around 180 degrees. Else draw wiggly lines.”

We need to do this with an if statement. For this example, it will be structured as follows

while True:

if t.offscreen():

# turn around 180 degrees

# go forward 30 units

else:

# draw wiggly lines, as in lines 4-6 above

Replace the loop in lines 3-6 of the original code with the loop above. Replace the lines starting with # with the correct code. Make sure your indentation is correct.

t.offscreen() is an **expression** that evaluates to yes or no, depending on whether the turtle is off the screen or not. The if statement runs lines conditionally:

if t.offscreen():

# This block is run if t.offscreen() evaluates to yes

else:

# This block is run if t.offscreen() evaluates to no

If you have got the code correct, it might draw something like this.



# A wiggle-filled circle

At the moment, we are restraining the turtle to being on screen. By changing the if statement, the turtle can be restrained to a different area of the screen.

dist = t.distance(0,0) will get the turtle’s distance from the point (0,0) (the centre of the screen) and store it in the **variable** dist. dist will be a number.

By using a comparison operator (< or >) you can check whether it is greater or less than a value. For example, dist < 40 will be an **expression** that evaluates to yes or no, depending on whether dist is less than 40 or not.

The line t.setheading(t.towards(0,0)) will turn the turtle to face towards the point (0,0).

Modify your code above to constrain the turtle to a circle on the canvas. Use the skeleton code below if you are unsure how to structure it.

while True:

# Put distance of turtle from (0,0) into variable ‘dist’

# If ‘dist’ > 100

# turn towards (0,0)

# go forward some amount

else:

# as before

