M110: Python Programming

Self-Study#1

Turtle Graphics

AOU
الجامعة العربية المفتوحة
Arab Open University
Lebanon

Prepared by Dr. Ahmad Mikati

Content

- Introduction to Turtle Graphics
- Determining the State of the Turtle
- Problem solving using decomposition
- Using Loops to Draw Designs
- Drawing a line graph
- Programming the turtle using nested loops

Introduction

- Turtle graphics is an interesting and easy way to learn basic programming concepts. The Python turtle graphics system simulates a “turtle” that obeys commands to draw simple graphics.
- This turtle can be instructed to move around a two-dimensional space, leaving a trace of its movement using a coloured pen.
- This provides a lot of opportunities for drawing interesting shapes, diagrams and indeed pictures.

Turtle Graphics

- The first step in using Python’s turtle graphics system is to write the following statement:

import turtle

This statement is necessary because the turtle graphics system is not built into the Python interpreter.
Instead, it is stored in a file known as the **turtle module**. The *import turtle* statement loads the turtle module into memory so the Python interpreter can use it.
If you are writing a Python program that uses turtle graphics, you will write the import statement at the top of the program.

Drawing Lines with the Turtle

- The Python turtle is initially positioned in the center of a graphics window that serves as its canvas. If we instruct the turtle to move forward, it will move in the direction that the arrowhead is pointing.
- You can use the *turtle.forward(n)* command to move the turtle forward n pixels.
- **N.B**: in order to avoid writing **turtle.** before each command, you can use **from turtle import *** instead of **import turtle** at the beginning of your program. Also, you can use an alias name; for example: **import turtle as t** . The same applies to other packages/modules.

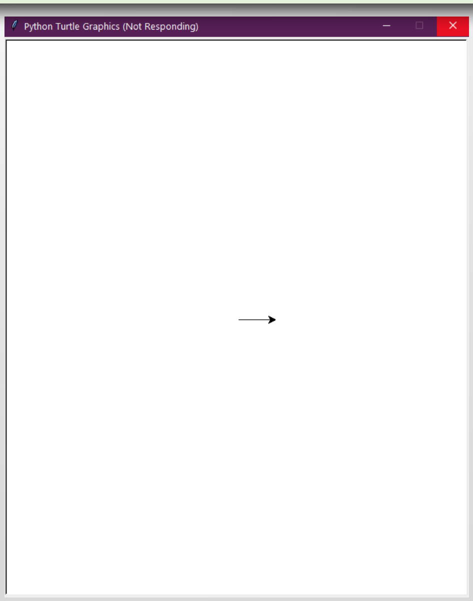
```
import turtle
turtle.forward(50)
```

OR

```
from turtle import *
forward(50)
```

OR

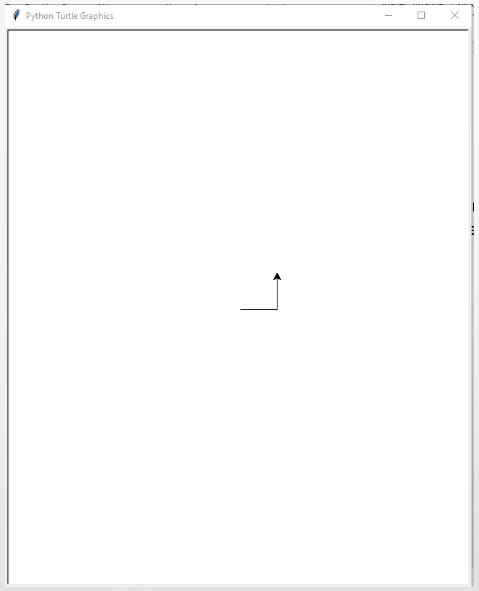
```
import turtle as t
t.forward(50)
```



Turning the Turtle

- When the turtle first appears, its default heading is 0 degrees (heading to the right).
- You can turn the turtle, so it faces a different direction by using either the **right(angle)** command, or the **left(angle)** command.

```
from turtle
import *
forward(50)
left(90)
forward(50)
```



Programming using turtle

- Here is our simple turtle program. (You can try these commands in Python and see what happens.)

```
# Draw start of staircase
from turtle import *
forward(40)
left(90)
forward(40)
right(90)
forward(40)
```

Comment “no execution“

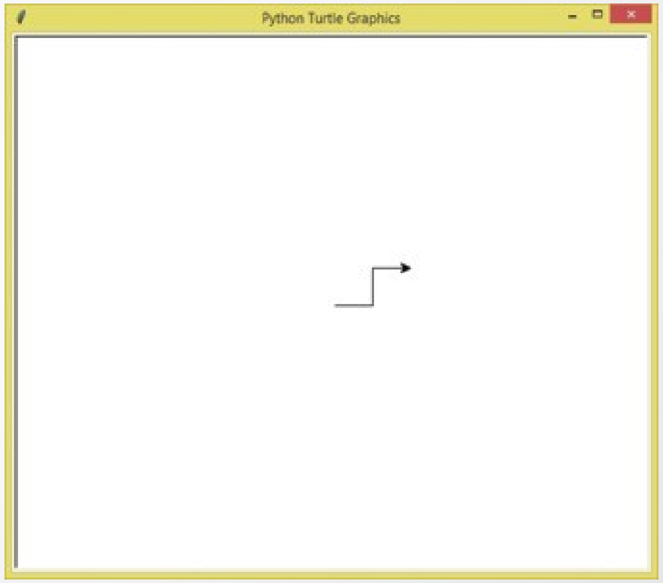
the line is stating that we will be working with commands that apply to a turtle

The operation that have been available for us to use on turtle

forward(40) : The operation works in terms of pixels. As the turtle always starts pointing horizontally to the right, so this line results in the turtle drawing a horizontal line from left to right of length 40 units.

Programming using turtle

- The program draws the start of a staircase consisting of three lines.
- Note that the figure includes an arrowhead representing the turtle.
- For our purposes it is sometimes useful to see where the turtle is and where it is pointing. This is partly because some of our solutions will be used as part of larger solutions.



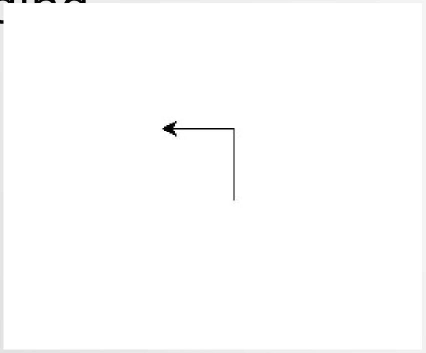
Hiding and Displaying the Turtle

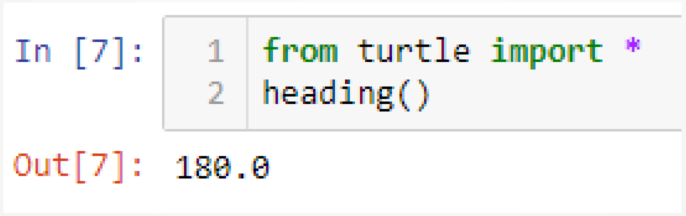
- If you don't want the turtle to be displayed, you can use the **hideturtle()** or **ht()** command to hide it.
- If you want the turtle to be displayed, you can use the **showturtle()** or **st()** command to display it.

Programming using turtle

Getting the Turtle’s Current Heading

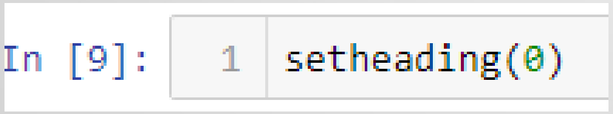
You can use the **heading()** command to display the turtle’s current head



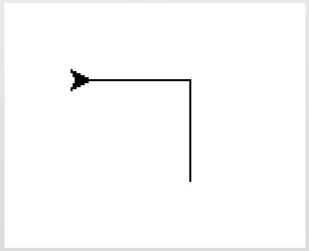


```
In [7]: from turtle import *
        heading()

Out[7]: 180.0
```

You can use the **setheading(angle)** command to set the turtle’s current heading to that angle.



```
In [9]: setheading(0)
```

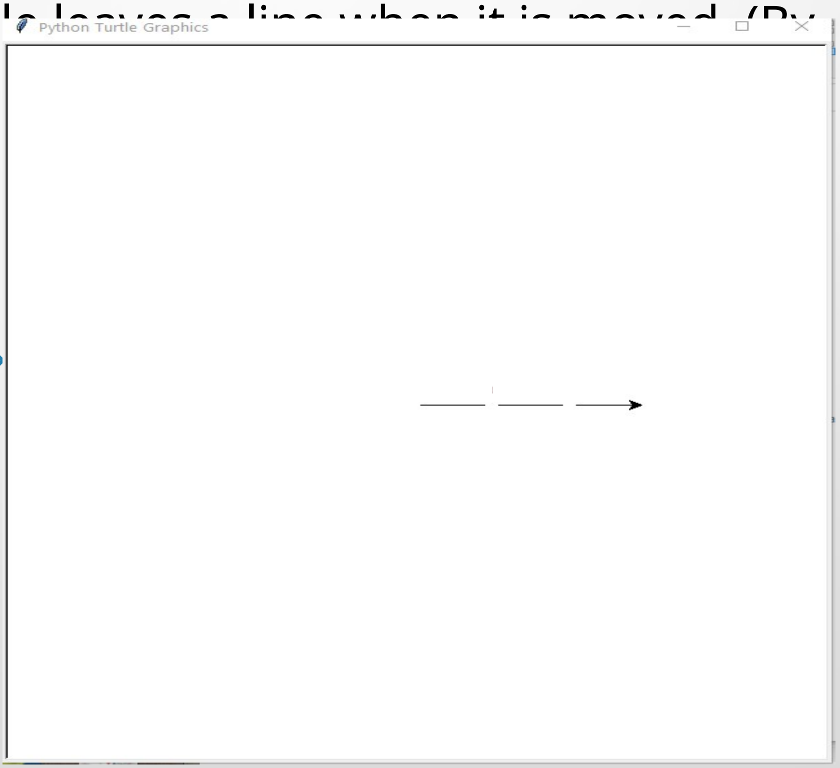


Programming using turtle

Moving the Pen Up and Down

In Python, you can use the **penup()** or **pu()** command to raise the pen, and the
pendown() or **pd()** command to lower the pen.
When the pen is up, you can move the turtle without drawing a line.
When the pen is down, the turtle leaves a line when it is moved. (By default, the pen is down.)
Consider the below program:

```
from turtle import *
forward(50)
pu()
forward(10)
pd()
forward(50)
pu()
forward(10)
pd()
forward(50)
```

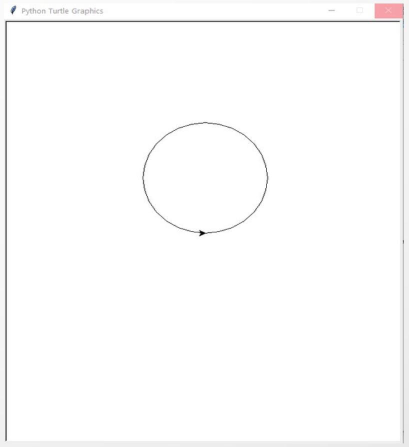


Programming using turtle

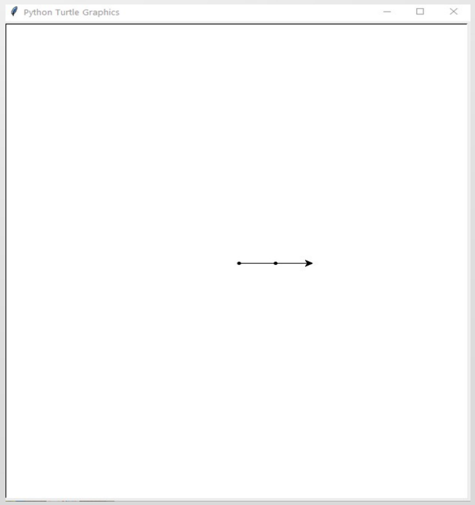
Drawing Circles and Dots

In Python, you can use the **circle(*radius value*)** command to make the turtle draw a circle with a radius pixels.
You can use the **dot()** command to make the turtle draw a simple dot.
Consider the below programs:

```
clear() # to clear the canvas
pu()
setpos(0,0) # to set the starting point
pd()
circle(100)
```



```
clear() # to clear the canvas
pu()
setpos(0,0) # to set the starting point
pd()
dot()
forward(50)
dot()
forward(50)
```



Programming using turtle

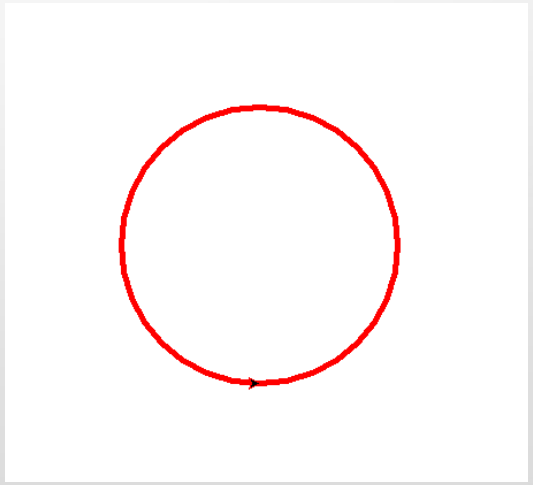
Changing the Pen Size

In Python, you can use the **pensize(*width*)** command to change the width of the turtle's pen, in pixels. The width argument is an integer specifying the pen's width.

Changing the Drawing Color

In Python, you can use the **pencolor(*color*)** command to change the turtle's drawing color. The *color* argument is the name of a color, as a string.
For example, **pencolor(*'red'*)**

```
import turtle as t
t.pensize(5)
t.pencolor('red')
t.circle(120)
```

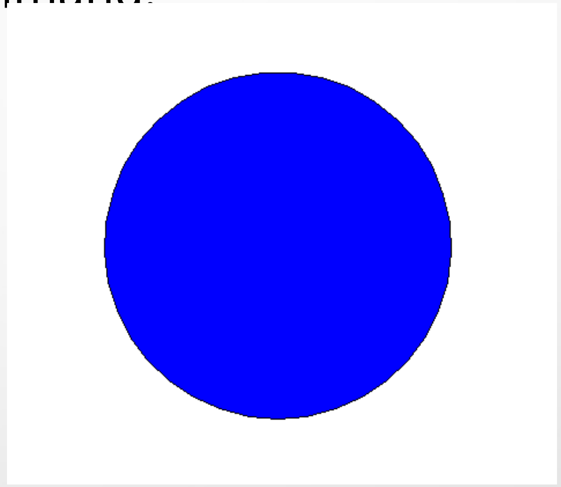


Programming using turtle

Filling Shapes

In Python, if you want to fill a shape with a color, you use the **begin_fill()** command before drawing the shape, then you use the **end_fill()** command after the shape is drawn. You can change the fill color with the **fillcolor(*color*)** command.

```
from turtle import *
reset()
ht()
fillcolor('blue')
begin_fill()
circle(150)
end_fill()
```



Setting the window's background color

To set the window's background color, you use the **turtle.bgcolor(color)** command.

Programming using turtle

Clearing the Screen

In Python, you can use the **clear()** command to erase all drawings that currently appear in the graphics window but neither change the drawing color nor change the turtle position in the center of the screen.

Resetting the Screen

In Python, you can use the **reset()** command to erase all drawings that currently appear in the graphics window, and reset the drawing color to black, and reset the turtle to its original position in the center of the screen.

```
clear() # to clear the canvas
pu()
setpos(0,0) # to set the starting point
pd()
```

=

```
reset() # to clear the canvas, and
        # to reset to the original position
```

Note that both commands will not reset the graphics window's background color to white. To do that, you should use **clearscreen()**, which does the same as **reset()** plus clearing the window's background.

Programming using turtle

Specifying the Size of the Graphics Window

In Python, you can use the **setup(***width, height***)** command to specify a size for the graphics window. The *width* and *height* arguments are the width and height, in pixels.

Displaying Text in the Graphics Window

In Python, you can use the **write(**text**)** command to display text in the graphics window. The text argument is a string that you want to display. When the string is displayed, the lower-left corner of the first character will be positioned at the turtle's X and Y coordinates.

Animation Speed

In Python, you can use the **speed**(*val*) command to change the speed at which the turtle moves. The *val* argument is a number in the range of 0 through 10. 10 is the fastest, 1 is the slowest, 0 means that the animation is disabled.

To know more about the previously explained turtle commands and many others, you can refer to the pdf file posted on your Moodle page.

Problem solving using decomposition
Programming using turtle

- If we had written the solution to the previous staircase problem in natural language (English, in our case), we might have had something like:

  > Draw start of staircase

  move forward by 40 units

  turn left by 90 degrees

  …

- Our first line uses a '>' symbol, which shows that the first line is a heading: '> Draw start of staircase', which tells us what we want to do. It describes the problem we are solving.
- The next two lines are a decomposition of the heading line above. These two lines achieve the task set out in the heading.

Problem solving using decomposition

Drawing some simple shapes through decomposition

Natural Language

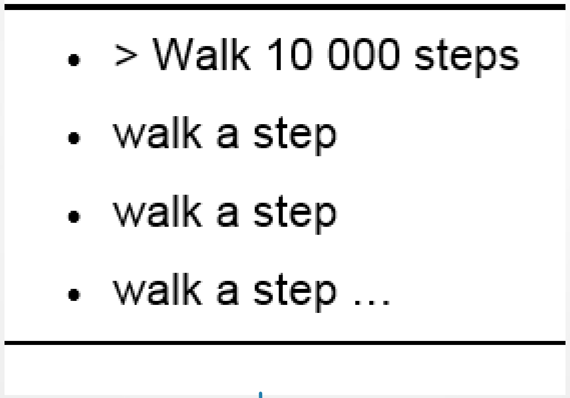
- > Draw L
- point turtle downwards
- move forward by 50 units
- turn left by 90 degrees
- move forward by 30 units

Python

```
# Draw L
from turtle import *
right (90)
forward (50)
left (90)
forward (30)
```

Using Loops to Draw Designs

- Imagine you have decided that you need to walk 10 000 steps per day to stay fit. We start to write the algorithm:

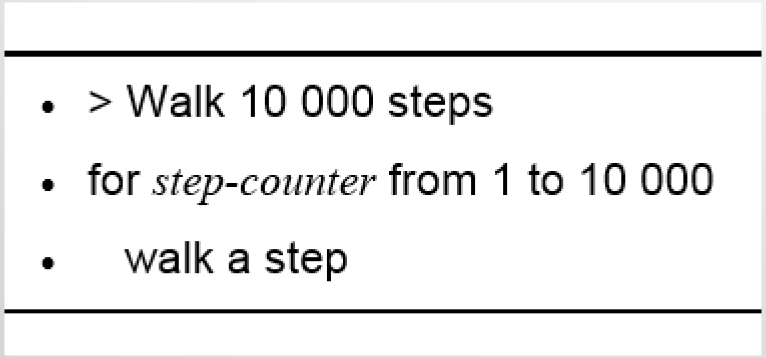


- > Walk 10 000 steps
- walk a step
- walk a step
- walk a step …

Having realized that writing the algorithm out this way is going to be very tedious, we want a way of writing down something that repeats a number of times.

Using Loops to Draw Designs

- We will have a **variable** called 'step-counter', and we need a way of giving it a value, doing something, and increasing the value. We also need to stop after 10 000.
- We will use the keyword '**for**' to show that we want to do something for a number of times. We will use it as follows:

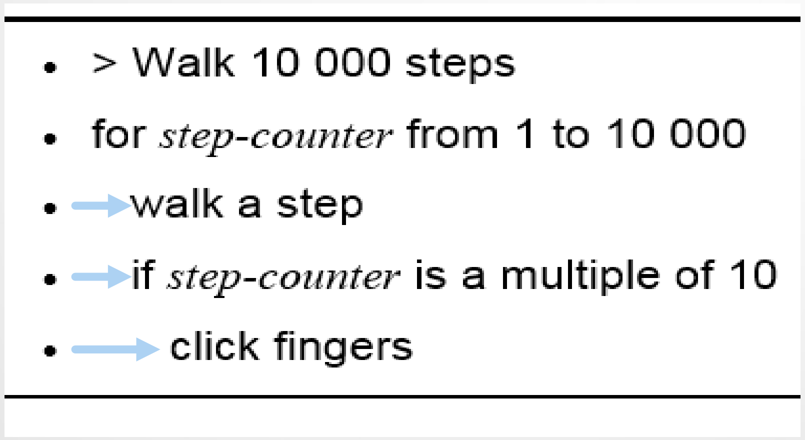


```
• > Walk 10 000 steps
• for step-counter from 1 to 10 000
•    walk a step
```

- This is an example of **iteration**: something gets carried out a number of times (Loop).
- The idea is that we do something repeating it a number of times.

Using Loops to Draw Designs (Making choices)

Now, suppose that I want to click my fingers only when a multiple of 10 is reached. Then, I can modify my algorithm as follows:

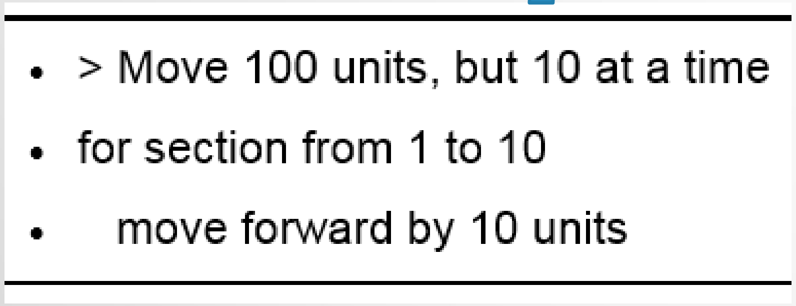


```
• > Walk 10 000 steps
• for step-counter from 1 to 10 000
•     walk a step
•     if step-counter is a multiple of 10
•         click fingers
```

- Notice that the **if** keyword is indented at the same level as ‘Walk a step’ – since they are two parts of the same sequence. The click step is indented further, to show it is conditional on the condition expressed by the if statement.

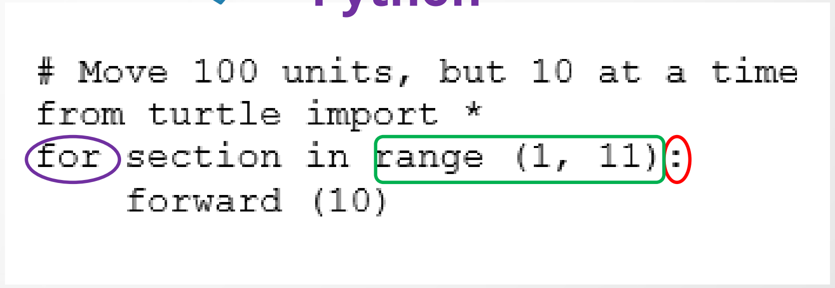
Using Loops to Draw Designs

In this case, **for** denotes iterating (looping) around a statement or sequence of statements.



```
• > Move 100 units, but 10 at a time
• for section from 1 to 10
•    move forward by 10 units
```

Python



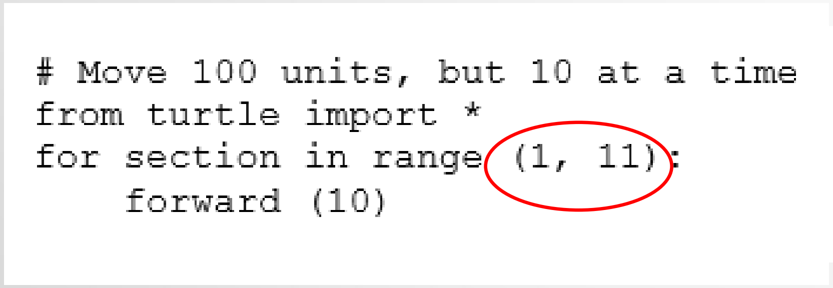
```python
# Move 100 units, but 10 at a time
from turtle import *
for section in range (1, 11):
    forward (10)
```

- The program will move the turtle forward by 100 units. It does this by moving forward 10 units and repeating this movement ten times.
- **range(1, 11)** means that the range of numbers starts at 1, counts upwards by 1 and stops just before 11.

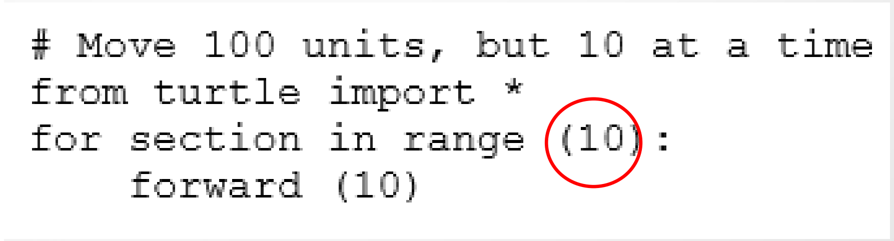
Using Loops to Draw Designs

Python

Alternative Solution



```
# Move 100 units, but 10 at a time
from turtle import *
for section in range (1, 11):
    forward (10)
```

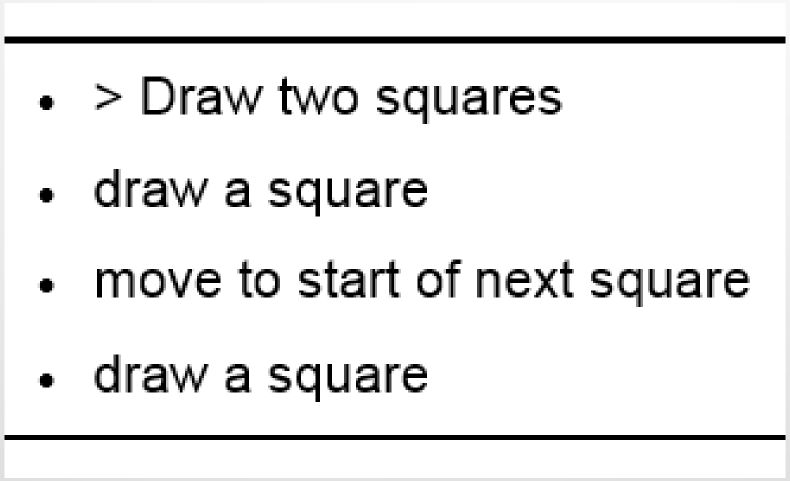


```
# Move 100 units, but 10 at a time
from turtle import *
for section in range (10):
    forward (10)
```

- Recall that In Python, when we specify a range using just one number, then the range starts at 0.
- In this case, when using range(10), we get a range of whole numbers from 0 to 9.

Using Loops to Draw Designs

- Let's think about a slightly more complicated problem. We will design and implement a program to draw two squares, one below the other, with a gap in between.

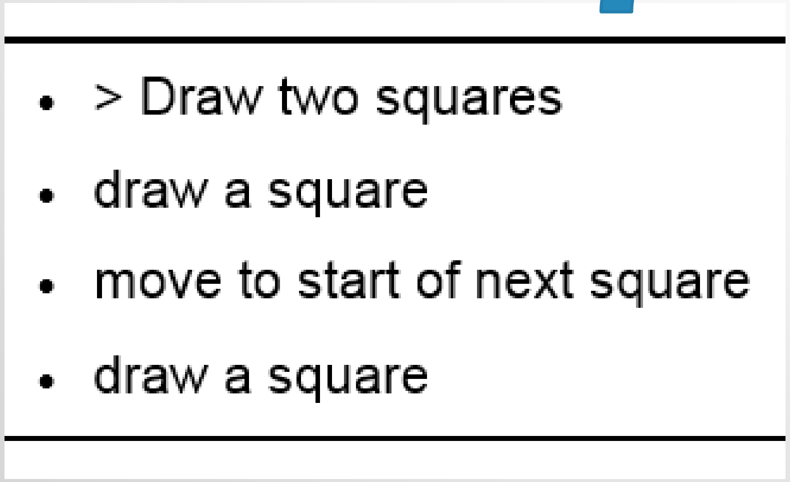


- > Draw two squares
- draw a square
- move to start of next square
- draw a square

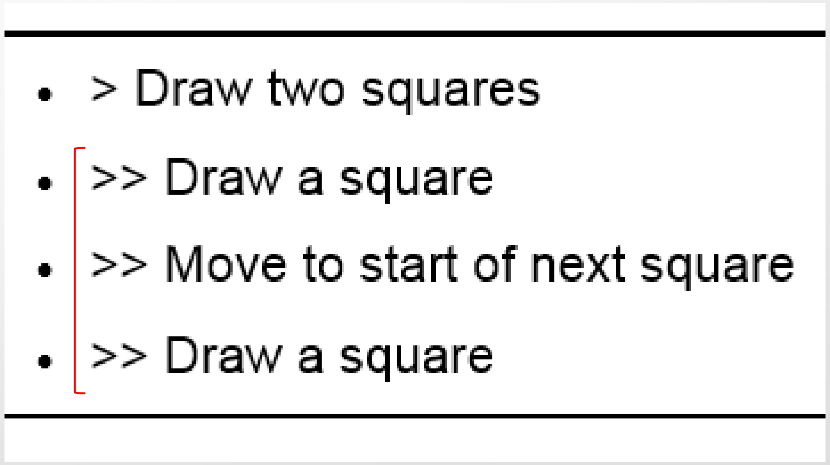
Problem decomposition

Using Loops to Draw Designs

- We have now decomposed the problem into sub-problems, rather than steps.



- > Draw two squares
- draw a square
- move to start of next square
- draw a square

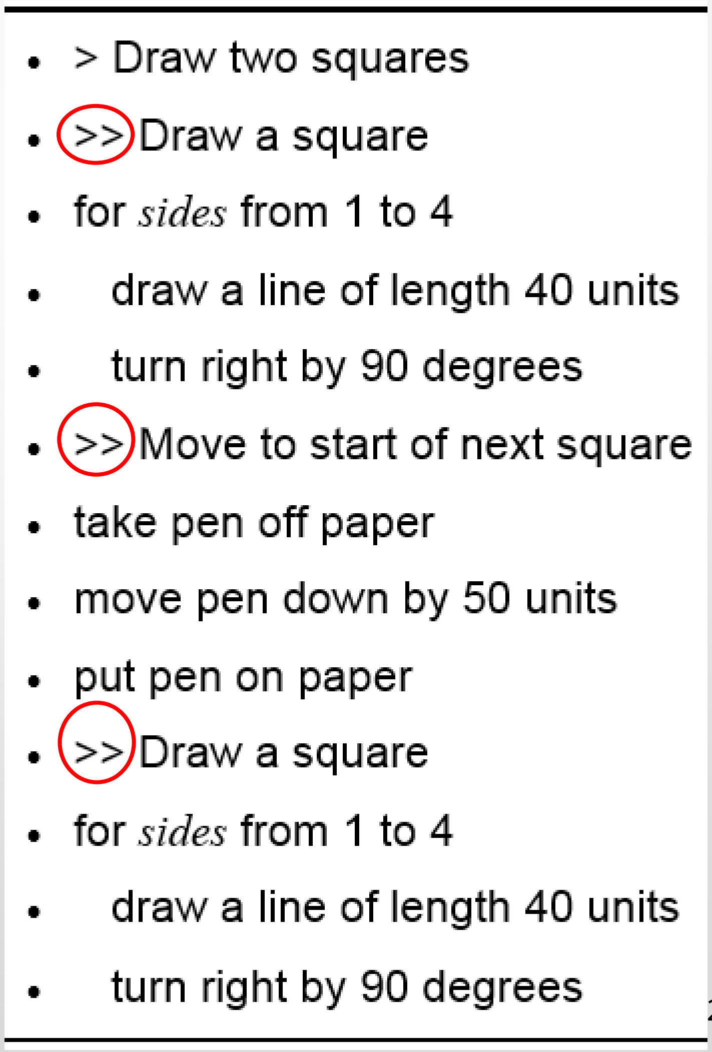


- > Draw two squares
- >> Draw a square
- >> Move to start of next square
- >> Draw a square

<u>The symbol >> refers to sub-problems</u>

Using Loops to Draw Designs

- Now we can decompose the sub-problems into steps.
- We did copy and paste from the original square algorithm! This is known as reusing solutions.
- Reuse is a mature approach to program design for saving time, and avoid introducing errors



```
• > Draw two squares
• >> Draw a square
• for sides from 1 to 4
•     draw a line of length 40 units
•     turn right by 90 degrees
• >> Move to start of next square
• take pen off paper
• move pen down by 50 units
• put pen on paper
• >> Draw a square
• for sides from 1 to 4
•     draw a line of length 40 units
•     turn right by 90 degrees
```

Drawing a graph of a fixed number of points

- In this subsection, we will consider how to draw a line graph.
- We will assume we are plotting the sales of gloves, by a given company, over the four quarters of the year.
- Now, we have seen how a variable could be used to control a loop, and that it is essentially a named box that can hold a value.
- Here we will use variables to hold the sales of gloves for each quarter.
- Variables g1, g2, g3 and g4 to hold the number, in millions, of gloves sold in the first, second, third and fourth quarters.

Drawing a graph of a fixed number of points

- The turtle has considerably more operations than we have seen so far.

  In particular, it has an operation ***goto*** that allows us to move the pen to a position, drawing a line as we go.

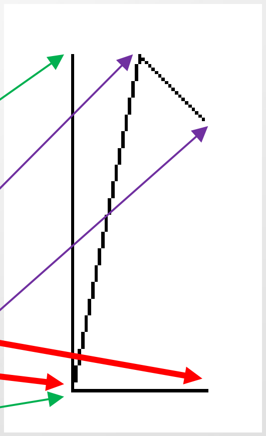
- Assume we are starting at position (0,0), the python code **goto(40,0)** moves the turtle horizontally to position (40,0), drawing a line if the pen is down. The first value in the brackets represents the horizontal displacement (from 0) and the second value represents the vertical displacement.

Drawing a graph of a fixed number of points

- > Produce graph for glove sales
- >> Set up gloves variables
- >> Produce x-axis
- >> Produce y-axis
- >> Plot data

Python

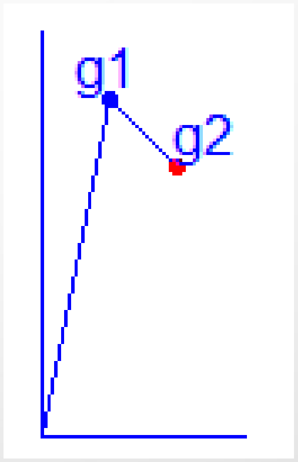
```
# Produce graph  for gloves
sales
from turtle import *
# set up the variables
g1=10
g2=8
# produce the x axis
goto(40,0)
goto(0,0)
# produce the y axis
goto(0,100)
goto(0,0)
# Plot data
goto(20, g1*10)
goto(40, g2*10)
ht()  # to hide the turtle
prompt
```



Drawing a graph of a fixed number of points

Python

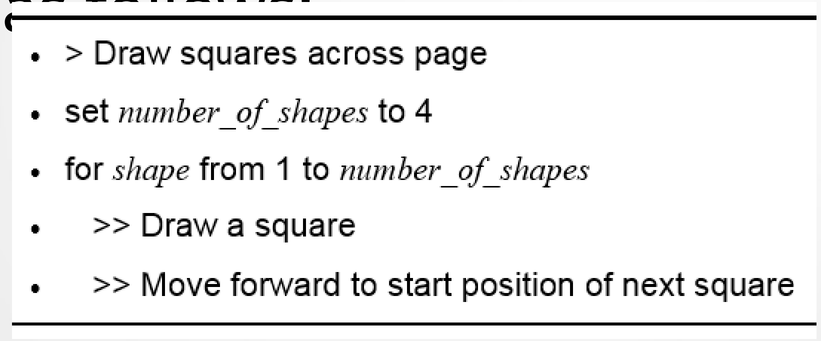
```
# Produce graph  for gloves sales
from turtle import *
# set up the variables
reset()
g1=10
g2=8
# produce the x axis
goto(40,0)
goto(0,0)
# produce the y axis
goto(0,120)
goto(0,0)
# Plot data for 2 values
goto(20, g1*10)
dot( 5, "blue")
write("g1",
False,"center","bold")
goto(40, g2*10)
dot( 5, "red")
write("g2", False,"left","bold")
ht()  # to hide the turtle prompt
```



Nested iteration

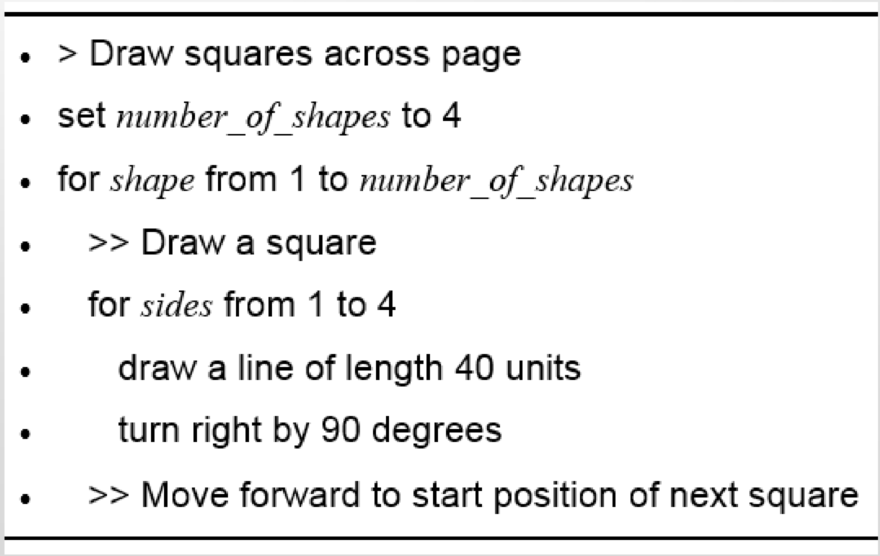
(Programming the turtle using nested loops)

- Consider a program to produce a number of squares across the page. A decomposition of this problem is a



```
> Draw squares across page
set $number\_of\_shapes$ to 4
for $shape$ from 1 to $number\_of\_shapes$
    >> Draw a square
    >> Move forward to start position of next square
```

- If we now copy the decomposition from subsection for ‘Draw a square’ into the above decomposition, we get:



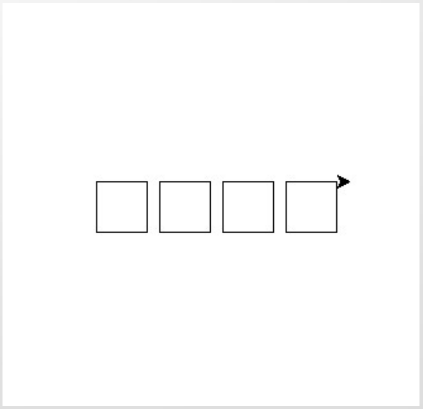
```
> Draw squares across page
set $number\_of\_shapes$ to 4
for $shape$ from 1 to $number\_of\_shapes$
    >> Draw a square
    for $sides$ from 1 to 4
        draw a line of length 40 units
        turn right by 90 degrees
    >> Move forward to start position of next square
```

Nested iteration (nested loops)

```
• > Draw squares across page
• set number_of_shapes to 4
• for shape from 1 to number_of_shapes
•     >> Draw a square
•     for sides from 1 to 4
•         draw a line of length 40 units
•         turn right by 90 degrees
•     >> Move forward to start position of next square
```

Problem Decomposition

```
# Draw squares across page
from turtle import *
number_of_shapes=4
for shape in range (1, number_of_shapes +1):
    for shapes in range(1,5):
        forward(40)
        right(90)
     # Move forward to start position of next square
    pu()
    forward(50)
    pd()
```



Translated Python Code

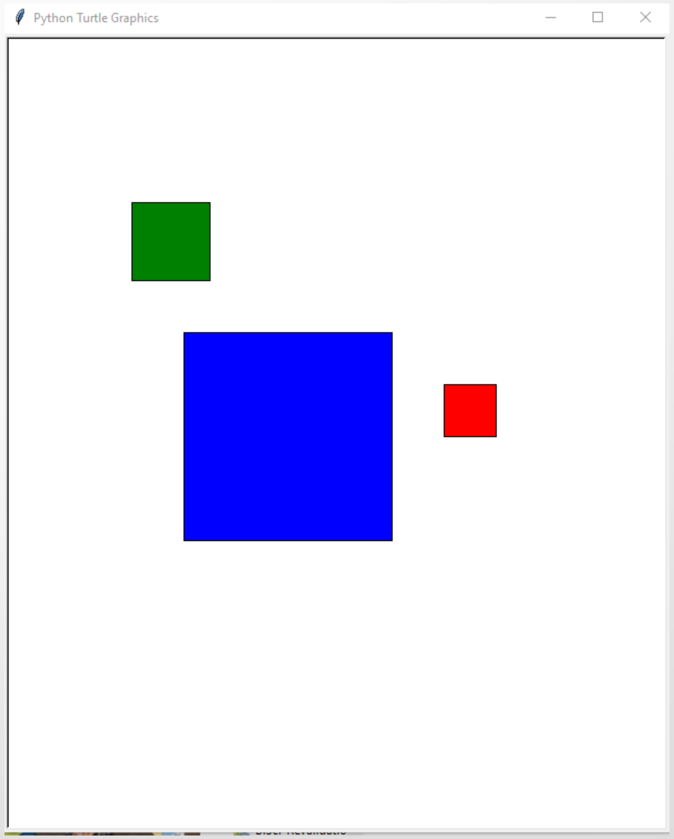
Modularizing Turtle Graphics Code with Functions

Commonly needed turtle graphics operations can be stored in functions
and then called whenever needed.
For example, if we need to draw a lot of blue squares, in different locations on the screen? Suddenly, we find ourselves writing similar lines of code, over and over.
We can simplify our program by writing a function that draws a square at a specified location, and then calling that function anytime we need it.

The following program demonstrates modularizing turtle graphics code with functions

Modularizing Turtle Graphics Code with Functions

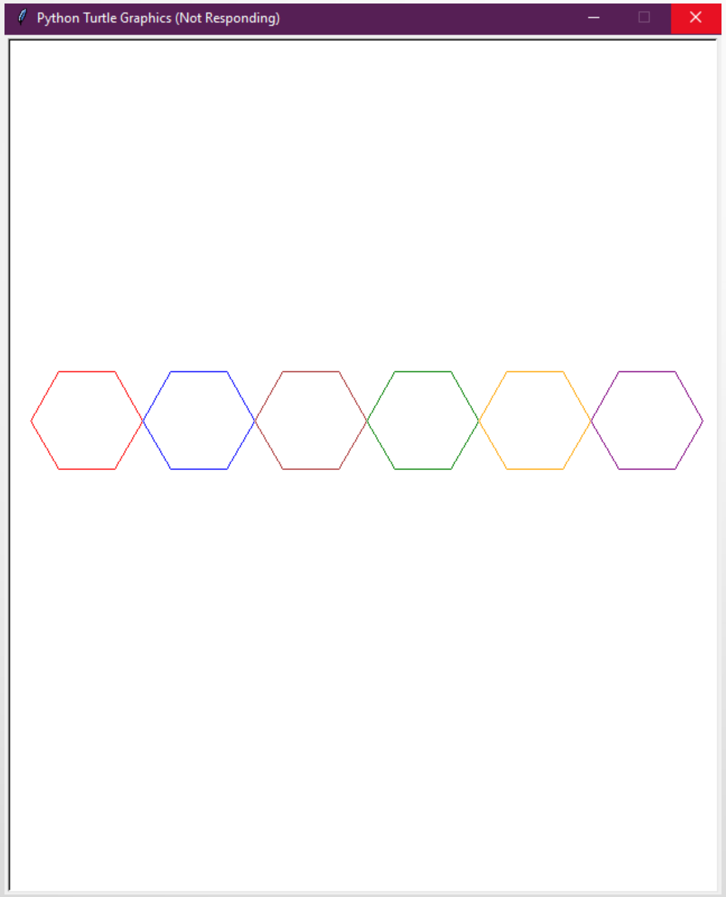
```
from turtle import *
def main():
    reset()
    ht()
    square(100, 0, 50, 'red')
    square(-150,-100, 200, 'blue')
    square(-200,150, 75, 'green')
    # The square function draws a square. The x and y
parameters
    # are the coordinates of the lower-left corner. The
width
    # parameter is the width of each side. The color
parameter
    # is the fill color, as a string.
def square(x, y, width, color):
    pu() # Raise the pen
    goto(x, y) # Move to the specified location
    fillcolor(color) # Set the fill color
    pd() # Lower the pen
    begin_fill()
    for count in range(4): # Draw a square
        forward(width)
        left(90)
    end_fill() # End filling
# Call the main function.
main()
```



Extra Exercises

Exercise 1

Draw six Hexagons with different colors. The output should be like the below figure:

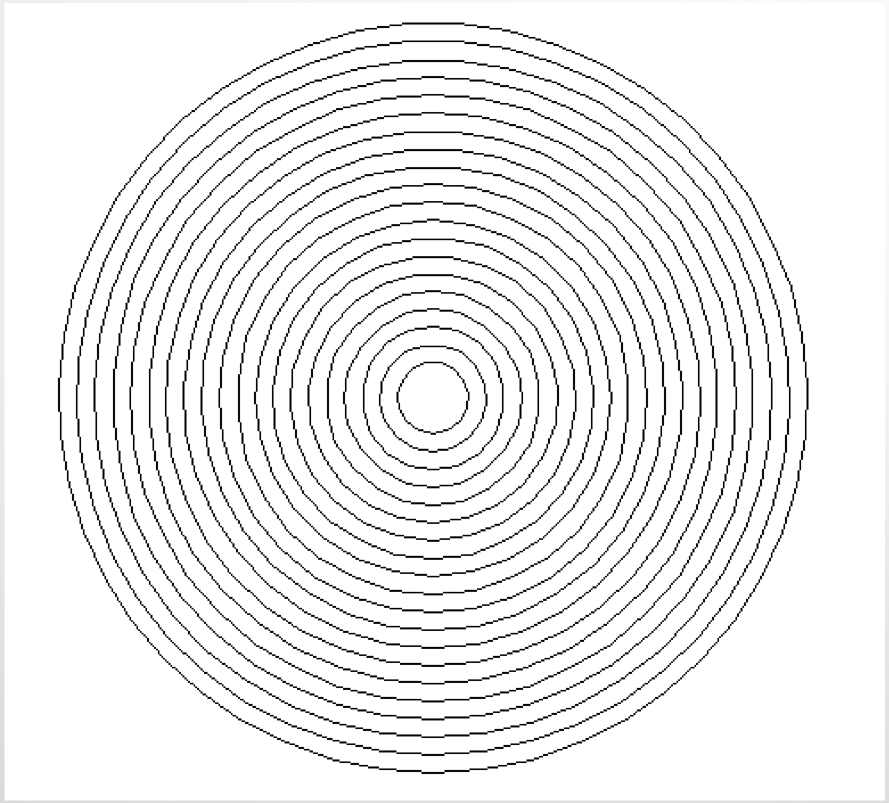


Exercise 1- Solution

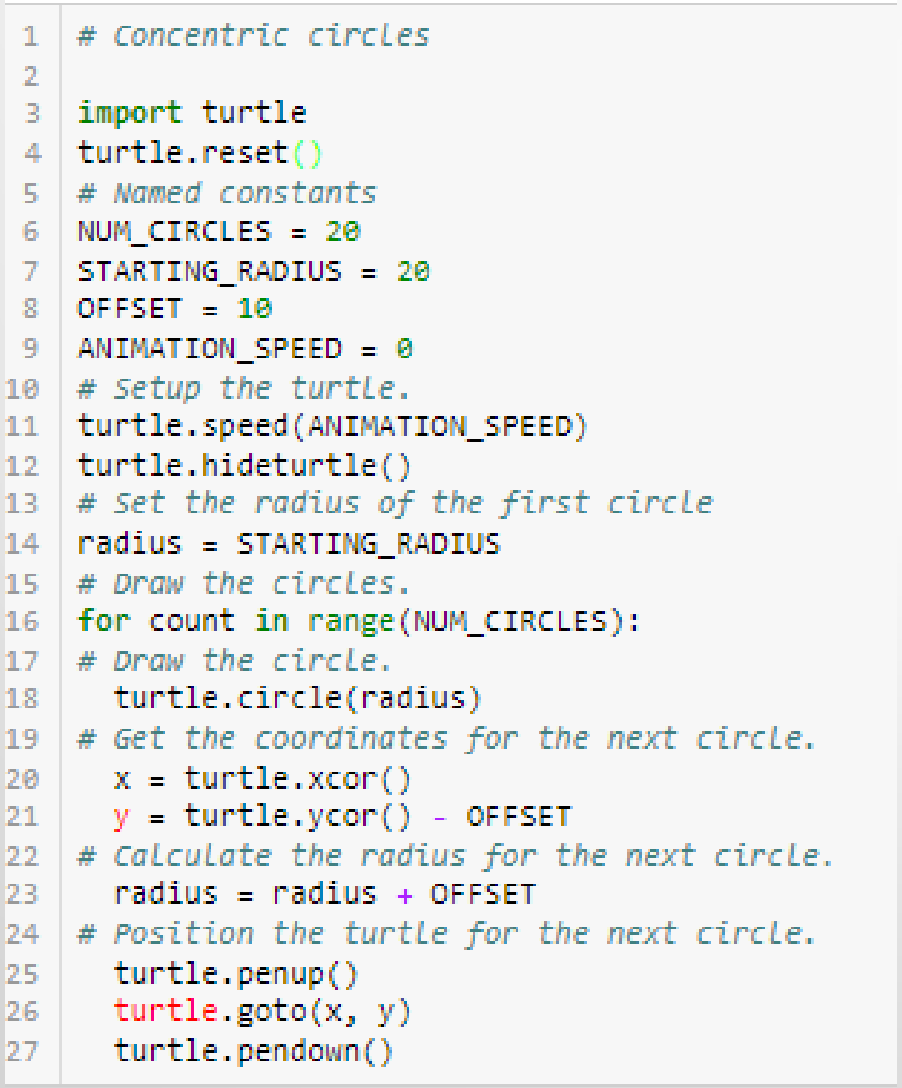
```
# Draw 6 Hexagons with different colors
from turtle import *
colors=['red', 'blue', 'brown', 'green', 'orange’,
'purple']
clear()
penup()
setpos(-225,0)
pendown()
speed(10)
for i in range(6):
 pencolor(colors[i])
 for sides in range(6):
   left(60)
   forward(50)
 penup()
 forward (100)
 pendown()
ht()
```

Exercise 2

Using a loop, draw the below shape. The output should be like the below figure:



Exercise 2- Solution



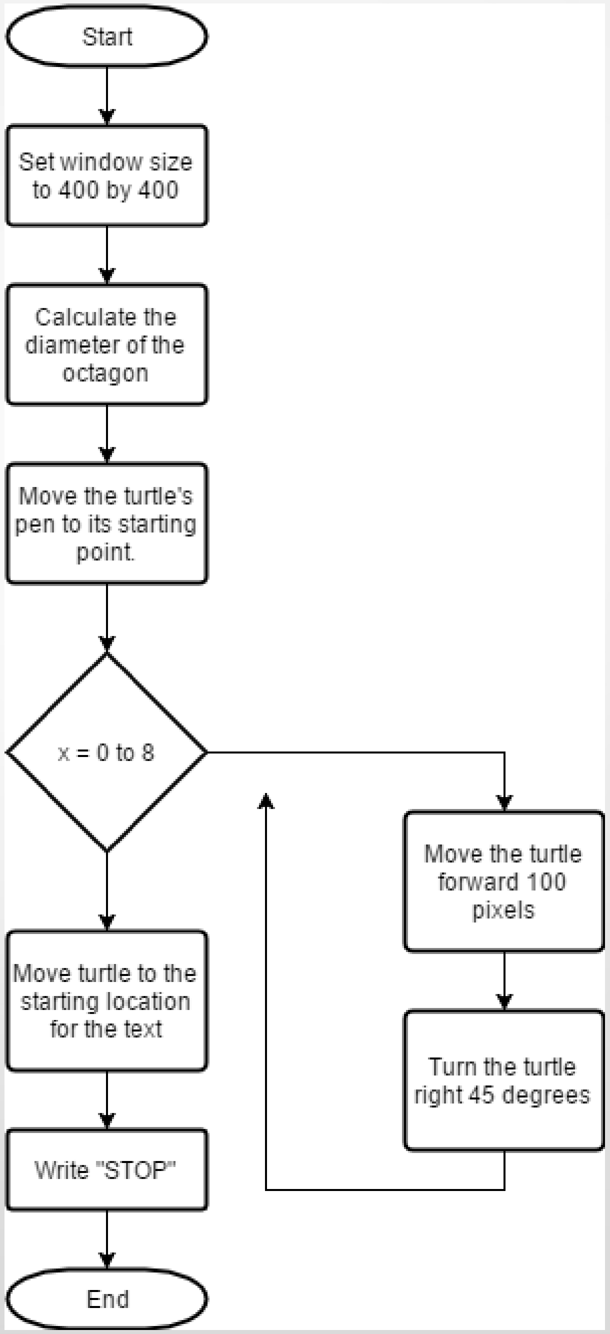
```python
# Concentric circles

import turtle
turtle.reset()
# Named constants
NUM_CIRCLES = 20
STARTING_RADIUS = 20
OFFSET = 10
ANIMATION_SPEED = 0
# Setup the turtle.
turtle.speed(ANIMATION_SPEED)
turtle.hideturtle()
# Set the radius of the first circle
radius = STARTING_RADIUS
# Draw the circles.
for count in range(NUM_CIRCLES):
# Draw the circle.
  turtle.circle(radius)
# Get the coordinates for the next circle.
  x = turtle.xcor()
  y = turtle.ycor() - OFFSET
# Calculate the radius for the next circle.
  radius = radius + OFFSET
# Position the turtle for the next circle.
  turtle.penup()
  turtle.goto(x, y)
  turtle.pendown()
```

Try to write a simpler code that displays the same output

Exercise 3

Write a Python program that implements the following flowchart.

Write your own code


Start
Set window size to 400 by 400
Calculate the diameter of the octagon
Move the turtle's pen to its starting point.
x = 0 to 8
Move the turtle forward 100 pixels
Turn the turtle right 45 degrees
Move turtle to the starting location for the text
Write "STOP"
End