PYTHON

TURTLE

LET'S DRAW THE PYTHON WAY

an enhextended reimplementation of old python feature.

BY:
ARSHADUL SHAIKH
REF FROM COURSERA + UDEMY

*****NOTE**
Before starting with the amazing simple + interesting feature of python , please try to go through "the fast lane to python (part 1)" in order to understand the concepts in a better way. lot said, let's turtle :)
==========

Turtle graphics is a popular way for introducing programming to school kids and after the **New Education Policy 2020** which brings about several major reforms in education in India, i bet you would see this replacing **"LOGO"** programming language (if it is being used even today by any of the schools).

Imagine that you have a turtle that understands English. You can tell your turtle to do simple commands such as go forward and turn right. As the turtle moves around, if its tail is down touching the ground, it will draw a line (leave a trail behind) as it moves. If you tell your turtle to lift up its tail it can still move around but will not leave a trail. As you will see, you can make some pretty amazing drawings with this simple capability.

A robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the command turtle.forward(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.right(25), and it rotates in-place 25 degrees clockwise.

By combining together these kind of commands, intricate shapes and pictures can easily be drawn.The turtle module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

The turtle module provides turtle graphics primitives, in both **object-oriented**(program is divided into small parts called objects) and **procedure-oriented** (program is divided into small parts called functions) ways.

PROCEDURAL ORIENTED PROGRAMMING	OBJECT ORIENTED PROGRAMMING
In procedural programming, program is divided into small parts called ***functions*** .	In object oriented programming, program is divided into small parts called ***objects*** .
Procedural programming follows ***top down approach*** .	Object oriented programming follows ***bottom up approach*** .
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is ***less secure*** .	Object oriented programming provides data hiding so it is ***more secure*** .
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on ***unreal world*** .	Object oriented programming is based on ***real world*** .
Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc.

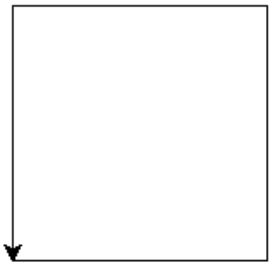
You will find working with turtle amusing as it will allow you to visualize what our code is doing, but the real purpose is to teach ourselves a little more Python and to develop our theme of computational thinking.

You'll first draw simple geometric shapes with the turtles, and then we'll summarize the concepts and syntax you've learned, in particular, classes, instances, and method invocations.

These concepts are the building blocks of object-oriented programming, a paradigm for structuring a program that is widespread in every modern programming language.

Let's run our first turtle program and then will try to understand what exactly is happening in the background (i would recommend to write and run code line by line, to see what is your turtle actually doing with every command you type) :

```
import turtle # we have imported a module which is turtle, which allow to use turtles library
newbie = turtle.Screen()      # creates a graphics window
new_turt = turtle.Turtle()    # create a turtle named new_turt
new_turt.forward(150)         # tell new_turt to move forward by 150 units
new_turt.left(90)             # turn by 90 degrees
new_turt.forward(150)          # complete the second side of a square
new_turt.left(90)
new_turt.forward(150)
new_turt.left(90)
new_turt.forward(150)    #completes the square
```



what actually happened here?

- The first line told python to load a module named turtle. That module brings us two new types that we can use: the Turtle type, and the Screen type.

- The dot notation **turtle.Turtle** means "**The Turtle type that is defined within the turtle module**".
    - Remember that Python is case sensitive, so the module name, turtle, with a lowercase t, is different from the type Turtle because of the uppercase T.

- We then created and opened what the turtle module calls a screen which we assign to variable newbie.

- The variable new_turt is made to refer to this turtle. These first three lines set us up so that we are ready to do some drawing.In lines 4-10, we have instructed the object new_turt to perform some movement actions leaving it's trace behind in order to form a perfect square.

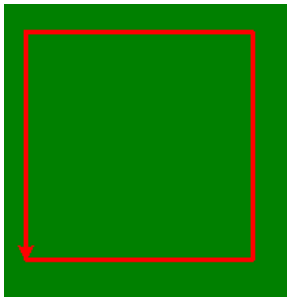
- Commands forward/right/left/backward are all methods of the object new_turt which were given to it after initializing as a turtle object, and now invoking or activating those methods we are able to perform actions

We are well aware about what are prerequisites to activate a basic turtle but some color while drawing definitely adds up the excitement and interest in what we are doing. So let us try to add and use some more **attributes (also called properties)** , if i want to add color to the background for white to consider blue, after creation the screen you can run **new_turt.bgcolor('blue')** and that's it we have a blue background screen/window for our drawing.

What else could you think of more to change?
The color of turtle itself, the width of turtle , which way it is facing , the position of the turtle within the window . These all are part of the turtle current **state,** we have a quite a number of methods to modify these attributes.

Will try to learn about these methods with one example:

```
import turtle # we have imported a module which is turtle, which allow to use turtles library
newbie = turtle.Screen()      # creates a graphics window
newbie.bgcolor('green')       # sets the background color to green
new_turt = turtle.Turtle()    # create a turtle named new_turt
new_turt.color('red')         # makes the turtle prompt color red
new_turt.pensize('3')         # set the  width of the pen
new_turt.forward(150)         # tell new_turt to move forward by 150 units
new_turt.left(90)             # turn by 90 degrees
new_turt.forward(150)          # complete the second side of a square
new_turt.left(90)
new_turt.forward(150)
new_turt.left(90)
new_turt.forward(150)    #completes the square
newbie.exitonclick()
```



- The objects stayed same but we have added new attributes like the first one **newbie.bgcolor('green')** which helped us to change the color of the screen.
- We then changed the color of the turtle itself to red using method color() as **new_turt.color('red')**
- We could have people like me who have spectacles and find it tough to see minute objects, pensize comes to your rescue to change to whatever size you want the image lines to be as we did with **new_turt.pensize('3')**

- Last line plays the most important role. The newbie variable refers to the window and invoke its **exitonclick method**, the program pauses execution and waits for the user to click the mouse somewhere in the window. When this click event occurs, the response is to close the turtle window and exit (stop execution of) the Python program.

what if you want all these attributes to be dynamic and it prompts the user to enter the desired background color , also the color of pen , the size of the pen ?

```
import turtle
newbie = turtle.Screen()
bg_color=input('what bg color do you want= ') #taking input of bgcolor from user
newbie.bgcolor(bg_color)
new_turt = turtle.Turtle()
pen_color=input('what pen color do you want= ') #taking input of pen color from user
new_turt.color(pen_color)
size=input('what pen size do you want= ')#taking input of pen size from user
size=int(size)    #most important part cause here the pen size is int and input is string
new_turt.pensize(size)
new_turt.forward(150)
new_turt.left(90)
new_turt.forward(150)
new_turt.left(90)
new_turt.forward(150)
new_turt.left(90)
new_turt.forward(150)
newbie.exitonclick()
```

```
what bg color do you want= white
what pen color do you want= red
what pen size do you want= 4
```

Questions?

What does the line “import turtle” do?

Why do we type turtle.Turtle() to get a new Turtle object?

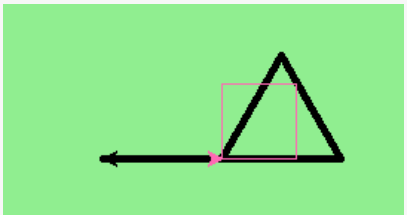
Instances: A Herd of Turtles

As we can have different variables of similar data types in a program , similarly we can have several turtles too. Each of them would be an independent object or to be specific an instance of turtle type (class).

Each instance will have its own type of state -- i.e attributes and methods new_turt that we saw might be drawing with a blue pen while the old_turt instance which we will create might be drawing with the old black and white color.

Will see an example where old_turt and new_turt both designing 2 different structures on the same window.

```
import turtle
newbie = turtle.Screen()              # Set up the window and its attributes
newbie.bgcolor("lightgreen")


new_turt = turtle.Turtle()            # create new_turt and set his pen width
new_turt.pensize(5)

old_turt = turtle.Turtle()            # create  old_turt
old_turt.color("hotpink")             # set his color

new_turt.forward(80)                  # Let new_turt draw an equilateral triangle
new_turt.left(120)
new_turt.forward(80)
new_turt.left(120)
new_turt.forward(80)
new_turt.left(120)                    # complete the triangle

new_turt.right(180)                   # turn new_turt around
new_turt.forward(80)                  # move her away from the origin so we can see old_turt

old_turt.forward(50)                  # make old_turt draw a square
old_turt.left(90)
old_turt.forward(50)
old_turt.left(90)
old_turt.forward(50)
old_turt.left(90)
old_turt.forward(50)
old_turt.left(90)

newbie.exitonclick()
```



Here are some How to think like a computer scientist observations:

- There are 360 degrees in a full circle. If you add up all the turns that a turtle makes, no matter what steps occurred between the turns, you can easily figure out if they add up to some multiple of 360.

- This should convince you that old_turt is facing in exactly the same direction as he was when he was first created. (Geometry conventions have 0 degrees facing East and that is the case here too!)

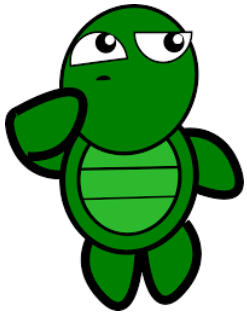
- We could have left out the last turn for old_turt, but that would not have been as satisfying. If you're asked to draw a closed shape like a square or a rectangle, it is a good idea to complete all the turns and to leave the turtle back where it started, facing the same direction as it started in.

- This makes reasoning about the program and composing chunks of code into bigger programs easier for us humans!We did the same with new_turt : it drew a triangle and turned through a full 360 degrees. Then we turned the prompt around and moved aside so that the other prompt of turtle object old_turt can be visible.

- Even the blank line 18 is a hint about how the programmer's mental chunking is working: in big terms, tess' movements were chunked as "draw the triangle" (lines 12-17) and then "move away from the origin" (lines 19 and 20)

- One of the key uses for comments is to record your mental chunking, and big ideas. They're not always explicit in the code.And, uh-huh, two turtles may not be enough for a herd, but you get the idea!

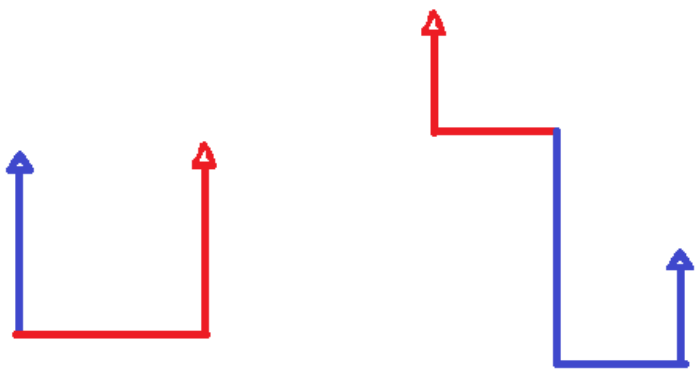
What will be your answer, if someone asks you can you have only one active turtle at a time?



Well as we just saw in above example , old_turt and new_turt both were alive and ready to execute next steps if we feed them after drawing the triangle and square.

You can create and use as many turtles as you like. As long as they have different names, you can operate them independently, and make them move in any order you like.

Try practicing below 2 examples where blue being first and red being the 2nd turtle object



OOP Concepts:

Everyone learning python after have already been through some programming languages like c++/java/c# etc would be very well aware about what is OOP(object oriented programming)

Will go through some of the important topics which we have indirectly already used while writing programs using turtle.

User Defined Classes

Given a class like **Turtle or Screen,** in all above examples we created new instances with a syntax **turtle.Turtle()** which almost looked like a function call. The old learning that you would have from any other language, ".(dot)" operator for you would be a way to access a class belonged variables/functions, but here Python interpreter is smart enough to figure out that Turtle is a class rather than a function, and so it creates a new instance of the class and returns it.

Since the Turtle class was defined in a separate module, (**confusingly, also named turtle**), we had to refer to the class as turtle.Turtle. Thus, in the programs we wrote **turtle.Turtle()** to make a new turtle. We could also write turtle.Screen() to make a new window for our turtles to paint in but as it is a good practice to initialize an object like we did with **newbie= turtle.Screen()** and then used the variable name to invoke other functions of Screen class like **exitonclick()**

Attributes

Each instance can have attributes, sometimes called instance variables.

If you have gone through java programming , you would be able to relate with what instance variable, let's try to understand it from an example (if you have never went through java don't worry these kind of programs examples would just be used as examples of too lengthy and clumsy programming :D) :

```
import java.io.*;
public class Employee {
  // this instance variable is visible for any child class.
  public String name;
  // salary variable is visible in Employee class only.
  private double salary;
  // The name variable is assigned in the constructor.
  public Employee (String empName) {
    name = empName;
  }
  // The salary variable is assigned a value.
  public void setSalary(double empSal) {
    salary = empSal;
  }
  // This method prints the employee details.
  public void printEmp() {
    System.out.println("name  : " + name );
    System.out.println("salary in $ :" + salary);
  }
  public static void main(String args[]) {
    Employee empOne = new Employee("Python");
    empOne.setSalary(1000);
    empOne.printEmp();
  }

}
```

coming back to python, from python pov instance variables are just like other variables in python. We use assignment statements, with an **=,** to assign values to them. **Thus, if new_turt and old_turt are variables bound to two instances of the class Turtle**, we can assign values to an attribute, and we can look up those attributes. For example, the following code would print out 1100.

```
new_trut.price = 500
old_trut.price = 600
print(new_trut.price + old_trut.price)
```

Methods:

Classes have associated methods, which are just a special kind of function. Considering what we wrote **new_trut.forward(150),** what exactly did interpreter do in the backend?

- Firstly it looks up for new_trut and finds that it is an instance of the class Turtle.

- Then it looks up the next attribute **forward** with the turtle class new_trut and finds that it is a method. How? because it has open round parentheses just after the attribute.
- soon after identifying the method it passes 150 as a parameter and moves 150 pixels ahead in the window.

- What is the difference between a method invocation and other function calls? The difference is that the object instance is also passed as a parameter. Thus new_trut.forward(150) moves the new_trut instance 150 pixels ahead.

- Some of the methods of the Turtle class set attributes that affect the actions of other methods. For example, the method pensize changes the width of the drawing pen, and the color method changes the pen's color. Once you change the pensize to new_trut.pensize('5') that would mean the next method call forward would move the instance 150 forward with size 5 pensize.

- **Methods return values**, just **as functions** do. However, none of the methods of the Turtle class that you have used return useful values the way the len function does. Thus, it would not make sense to build a complex expression like new_trut.forward(50) + 75. It could make sense, however to put a complex expression inside the parentheses: new_trut.forward(x + y)

Repetition with a For Loop

It is always tedious work to type similar code again and again, earlier programs that we have seen so far had so many "forward, left, right method calls, typed repetitively which i guess no one would liked.

If we want to make a repetitive pattern in your drawings, python makes the task easier. The control structure is called a loop, if you have learned other programming languages then you may have used loop in different forms, be it while/do-while/for/nested-loop, A **"for"** loop in python executes in a non-linear fashion.

Instead of evaluating the code line by line until it reaches the end, once the program reaches a for loop, it will tell the program to execute a set of lines repeatedly. After doing all that, the program will then continue to evaluate and execute whatever is below the for loop.

let's try to check the structure of for loop in python:

```
print("This will execute first")

for _ in range(3): #we have used _ you may use any other variable but
                   #as we won't be use that variable anywhere else _ is fine
    print("This line will execute first three times")
    print("This line will also execute after above line three times")

print("Now we are outside of the for loop!")
```

```
This will execute first
This line will execute first three times
This line will also execute after above line three times
This line will execute first three times
This line will also execute after above line three times
This line will execute first three times
This line will also execute after above line three times
Now we are outside of the for loop!
```

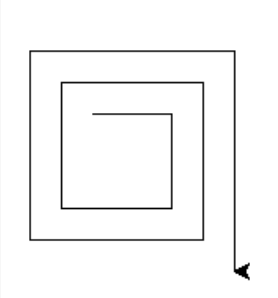
We have used **range function** in the for loop. range functions helps to specify how many times the code inside the for loop will execute.

***** KEY POINT**
The indentation is the most important thing in python. All of the statements that were printed for the number times mentioned in range function were indented under the for loop.
Once we stopped indenting those lines, then the program was outside of the for loop and it would continue to execute linearly.

Combine LOOP & TURTLE

Time to combine what we learned with turtle, we can do so many cool stuff if we can combine the control structures with it.

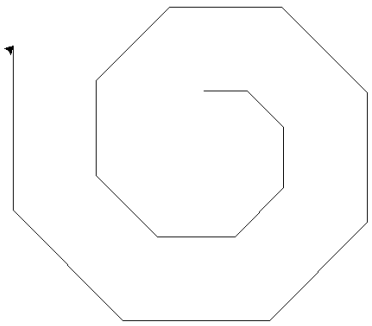
```
import turtle
wn=turtle.Screen()
maze_turtle=turtle.Turtle()
distance=50
print('lets draw our first maze')
for _ in range(5):
    maze_turtle.forward(distance)
    maze_turtle.right(90)
    distance=distance +10

print('we have succesfully drawn our first structure using for loop')
wn.exitonclick()
```



```
lets draw our first maze
we have succesfully drawn our first structure using for loop
```

We have used a variable **distance** which will keep on increasing with every loop meaning maze_turtle will cover more distance after every right turn.

```
for _ in range(15):
    maze_turtle.forward(distance)
    maze_turtle.right(45)
    distance=distance +10
```



A little change in the right turn degrees gave us entirely different structure , you can similarly play with the code creating cool such figures

More turtle Methods and Observations:

As we know that Turtle has several methods like left/right/forward/backward but luckily it also supports negative angles or distances.

- So **new_turt.forward(-100)** will move new_turt backwards, and **new_turt.left(-30)** turns it to the right.

- As there are 360 degrees in a circle, turning 30 to the left will leave you facing in the same direction as turning 330 to the right would meaning new_turt.left(30)== new_turt.right(330) (The on-screen animation will differ, though — you will be able to tell if tess is turning clockwise or counter-clockwise!)

- This means that you don't need both a left and a right turn method — you could be minimalists, and just have one method. Same is with backward method.

- A turtle's pen can be picked up or put down. This allows us to move a turtle to a different place without drawing a line. The methods are **up and down.** Note that the methods penup and pendown do the same thing.

- This commands will move the pen 50 forward but without leaving any trace or line as before running forward method up() lifted the pen , made it move forward 50 in consider air and then down dropped it down , so next command if you give forward(50) you could finally see the line.

```
new_turt.up()
new_turt.forward(50) # this moves new_turt, but no line is drawn
new_turt.down()
```

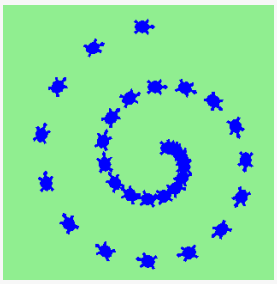
- Every turtle pen can be drawn in any shape you want. The ones available "out of the box" are arrow, blank, circle, classic, square, triangle, turtle

maze_turtle.shape('turtle')

- You can also s**peed up or slow down** the turtle's animation speed. (Animation controls how quickly the turtle turns and moves forward). Speed settings can be set between **1 (slowest) to 10 (fastest).** But if you set the **speed** to **0**, it has a special meaning — **turn off animation** and **go as fast as possible.**

- A turtle can "stamp" its footprint onto the canvas, and this will remain after the turtle has moved somewhere else. Stamping works even when the pen is up.

```
import turtle
wn = turtle.Screen()
wn.bgcolor("lightgreen")
maze_turtle = turtle.Turtle()
maze_turtle.color("blue")
maze_turtle.shape("turtle")

dist = 5
maze_turtle.up()                        # this is new
for _ in range(30):    # start with size = 5 and grow by 2
    maze_turtle.stamp()                 # leave an impression on the canvas
    maze_turtle.forward(dist)           # move maze_turtle along
    print(dist)
    maze_turtle.right(24)               # and turn
    dist = dist + 2
wn.exitonclick()
```



```
5
7
9
11
```

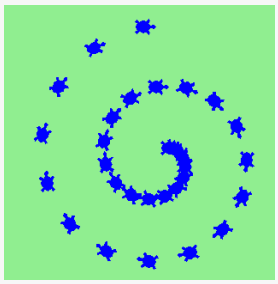
- Every turtle pen can be drawn in any shape you want. The ones available "out of the box" are arrow, blank, circle, classic, square, triangle, turtle

maze_turtle.shape('turtle')

- You can also s**peed up or slow down** the turtle's animation speed. (Animation controls how quickly the turtle turns and moves forward). Speed settings can be set between **1 (slowest) to 10 (fastest).** But if you set the **speed** to **0**, it has a special meaning — **turn off animation** and **go as fast as possible.**

- A turtle can "stamp" its footprint onto the canvas, and this will remain after the turtle has moved somewhere else. Stamping works even when the pen is up.

```
import turtle
wn = turtle.Screen()
wn.bgcolor("lightgreen")
maze_turtle = turtle.Turtle()
maze_turtle.color("blue")
maze_turtle.shape("turtle")

dist = 5
maze_turtle.up()                        # this is new
for _ in range(30):    # start with size = 5 and grow by 2
    maze_turtle.stamp()                 # leave an impression on the canvas
    maze_turtle.forward(dist)           # move maze_turtle along
    print(dist)
    maze_turtle.right(24)               # and turn
    dist = dist + 2
wn.exitonclick()
```



```
5
7
9
11
```

Glossary:

Method	Parameters	Description
Turtle	None	Creates and returns a new turtle object
forward	distance	Moves the turtle forward
backward	distance	Moves the turle backward
right	angle	Turns the turtle clockwise
left	angle	Turns the turtle counter clockwise
up	None	Picks up the turtles tail
down	None	Puts down the turtles tail
color	color name	Changes the color of the turtle's tail
fillcolor	color name	Changes the color of the turtle will use to fill a polygon
heading	None	Returns the current heading
position	None	Returns the current position
goto	x,y	Move the turtle to position x,y
begin_fill	None	Remember the starting point for a filled polygon
end_fill	None	Close the polygon and fill with the current fill color
dot	None	Leave a dot at the current position
stamp	None	Leaves an impression of a turtle shape at the current location
shape	shapename	Should be 'arrow', 'triangle', 'classic', 'turtle', 'circle', or 'square'
speed	integer	0 = no animation, fastest; 1 = slowest; 10 = very fast

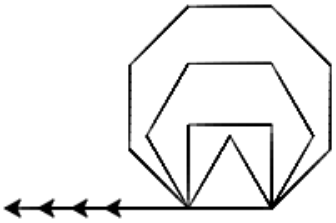
While writing a turtle program, what are the different errors you might end up in :

```
import turtle
wn = turtle.Screen()

new_turt = Turtle.turtle()   # Error 1: switched turtle and Turtle
                             #need to remember turtle is a module and
                             #Turtle is a class in turtle module


for _ in range():        # Error 2: Range() takes at least 1 arguments (0 given)

    new_turt.forward(distance) #Error 3: your o/p might look different
                               #than what you expected cause you haven't
                               #defined the variable distance

    New_turt.right(90    #Error 4: you forgot to close the round brackets, which is
                         #one of the most common errors

                       #Error 5: #capitalized the variable New_turt
                       #even though we have defined the instance variable with
                       #a all letters in lowercase

    distance=distance + 5
```

Practice

Question 1: Use for loops to make a turtle draw these regular polygons (regular means all sides the same lengths, all angles the same):
An equilateral
triangle
A square
A hexagon (six sides)
An octagon (eight sides)

Hint-> A basic output should look like this, you can change the pen size, background color, pen co ith stamps, depends all on your creativity

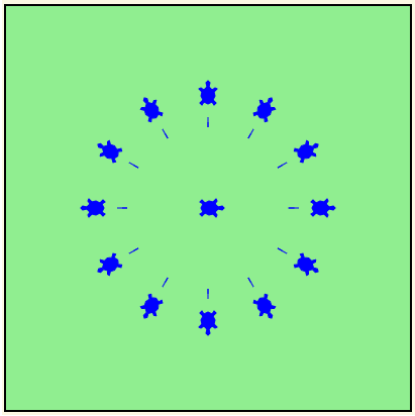


Question 2: Try writing program for a star!

Hint : after angle rotation the sum of all angles in a star pentagon is 180. considering all angles to be a+b+c+d+e=180

Question 3:

Write a program to draw a face of a clock that looks something like this:



Solutions 1:

```
import turtle
wn=turtle.Screen()
triangle_turt=turtle.Turtle()
square_turt=turtle.Turtle()
Hexagon_turt=turtle.Turtle()
Octagon_turt=turtle.Turtle()

#triangle
for _ in range(3):
    triangle_turt.forward(50)
    triangle_turt.left(360//3)

triangle_turt.left(180)
triangle_turt.forward(50)

#square
for _ in range(4):
    square_turt.forward(50)
    square_turt.left(360//4)

square_turt.left(180)
square_turt.forward(70)

#hexagon
for _ in range(6):
    Hexagon_turt.forward(50)
    Hexagon_turt.left(360//6)

Hexagon_turt.left(180)
Hexagon_turt.forward(90)

#octagon
for _ in range(8):
    Octagon_turt.forward(50)
    Octagon_turt.left(360//8)

Octagon_turt.left(180)
Octagon_turt.forward(110)
```

Solution 2:

```
import turtle
wn=turtle.Screen()
star=turtle.Turtle()

for _ in range(5):
    star.forward(100)
    star.right(144)
```

Solution 3:

```
import turtle
wn=turtle.Screen()
wn.bgcolor('lightgreen')

clock=turtle.Turtle()
clock.color('blue')
clock.shape('turtle')

clock.up()
for _ in range(12):
    clock.forward(80)
    clock.down()
    clock.forward(10)
    clock.up()
    clock.forward(20)
    clock.stamp()
    clock.forward(-110)
    clock.left(30)
```

PART 3