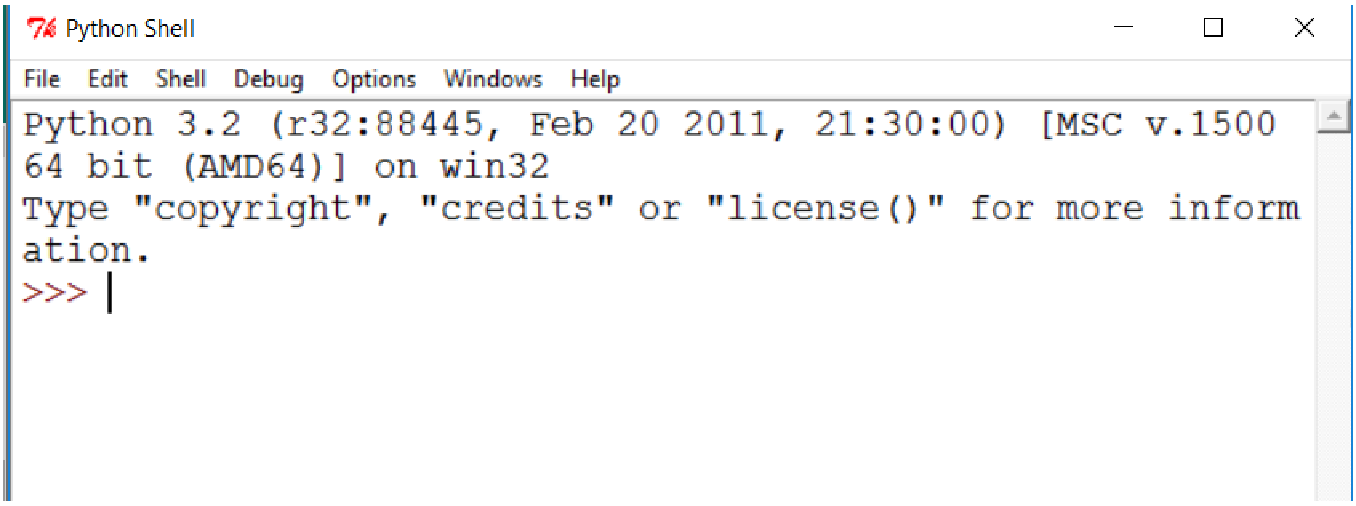
Turtles—Linear Equations

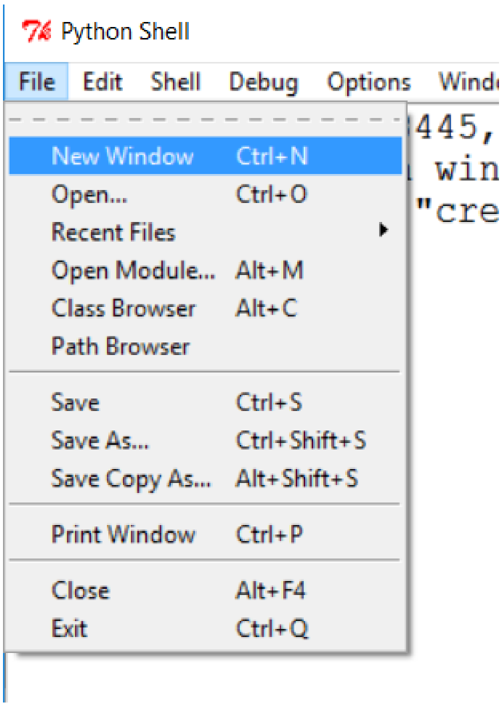
Name:	Class:	Date:

Create a new Python project as follows:

Open IDLE 3.x



Click file and create a New Window:

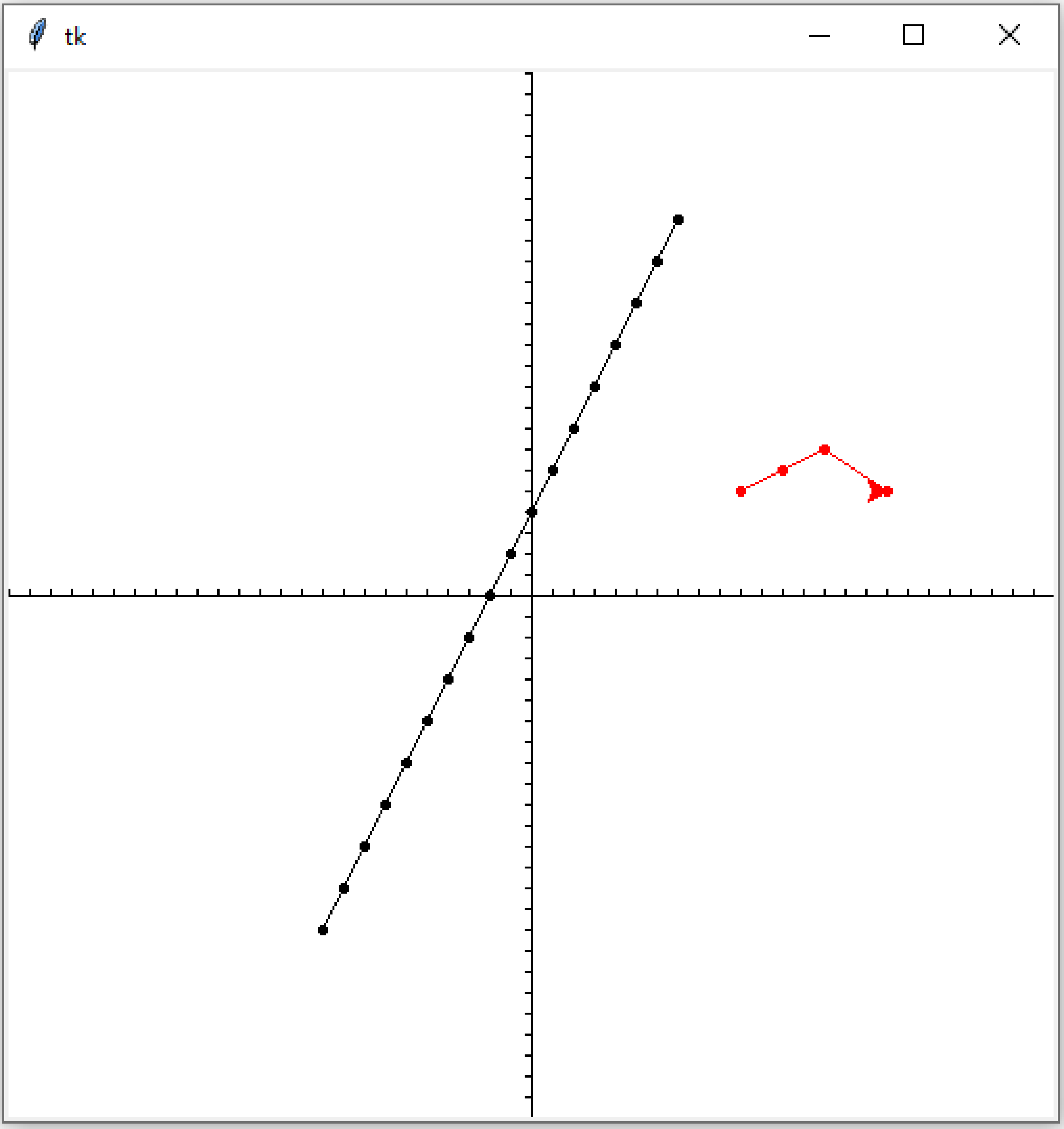


Call it LinearTurtle.py Great! Now we can start coding!

Create another file and call it Chart.py

Turtles—Linear Equations

Have you ever looked at a graph or chart on a computer and wondered how it is possible to plot lines?



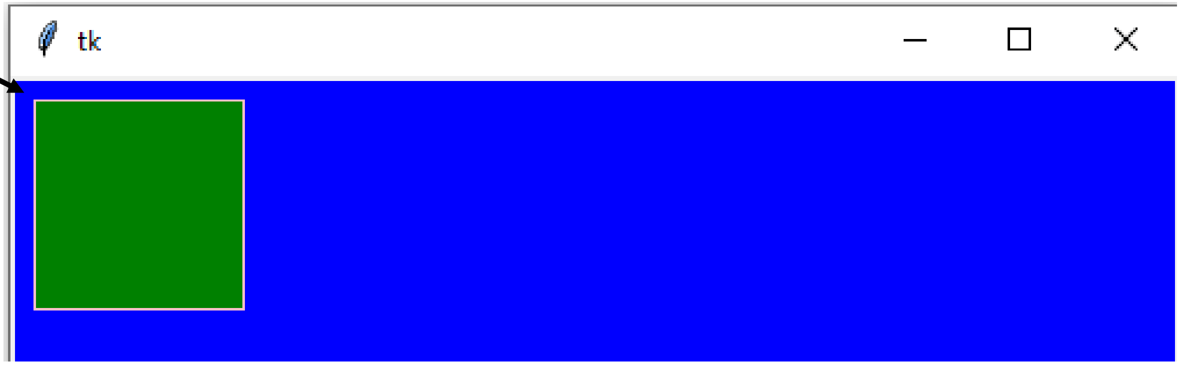
Turtles—Linear Equations

There's lots of barriers to drawing a line on a screen such as:

In many programs 0,0 is top left meaning that the y axis is upside down.

This is true in Python Tkinter and web design.

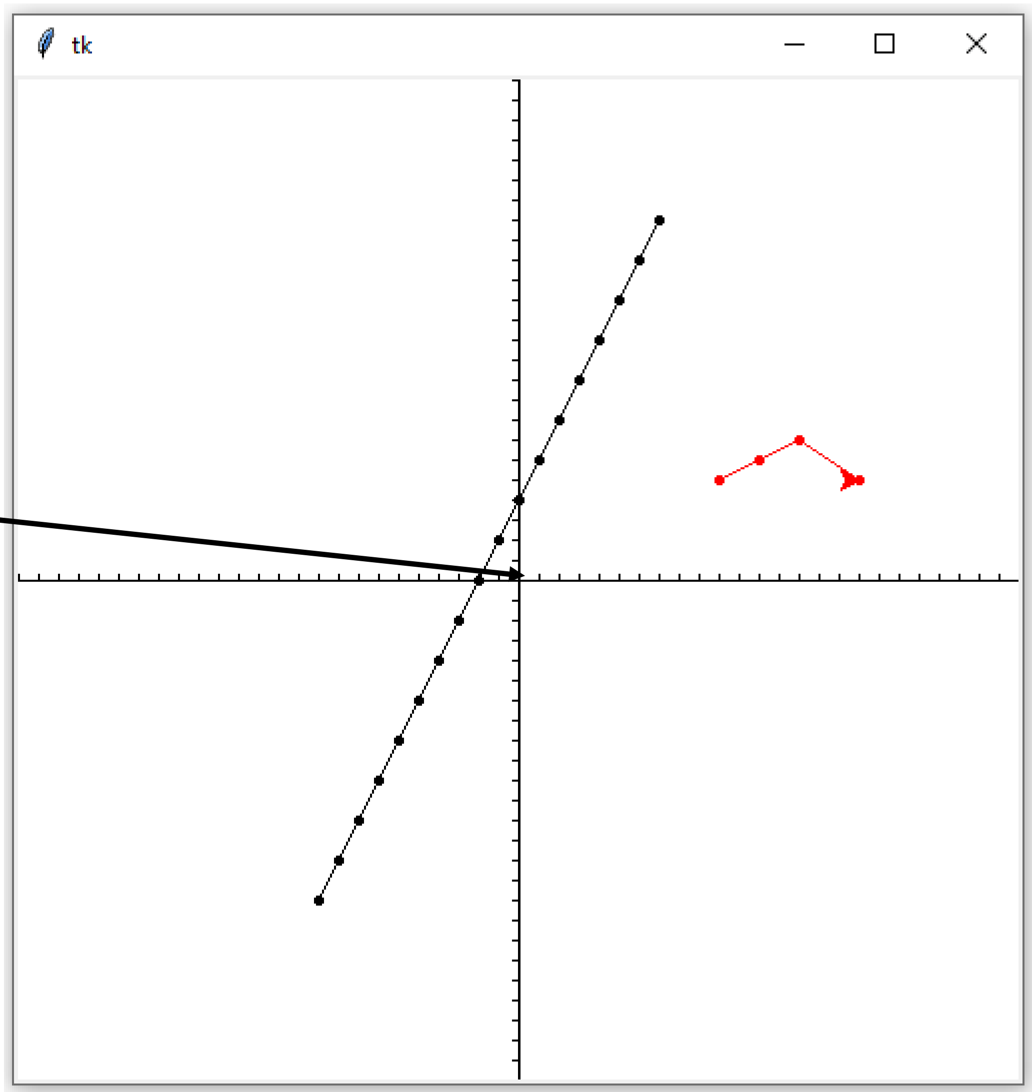
0,0



However, if you add a RawTurtle to a canvas it changes 0,0 to be the centre.

It also flips the y axis the right way up.

0,0



Turtles—Linear Equations

Let's start by creating a GUI so that the Linear Turtle we create later can be instantiated.

Part of this will also be writing a function which will have a turtle which draws the graph axis.

```
from LinearTurtle import *
from tkinter import *
from tkinter import ttk

def draw_chart(canvas):
    graphy = turtle.RawTurtle(canvas)
    graphy.ht()
    graphy.speed(0)
    graphy.penup()
    graphy.goto(250, 0)
    graphy.pendown()
    #Draws the x-axis
    for i in range(0, 50):
        graphy.back(10)
        graphy.left(90)
        graphy.forward(3)
        graphy.back(3)
        graphy.right(90)
    graphy.penup()
    graphy.goto(0, -250)
    graphy.pendown()
    graphy.left(90)
    #Draws the y-axis
    for i in range(0, 50):
        graphy.forward(10)
        graphy.left(90)
        graphy.forward(3)
        graphy.back(3)
        graphy.right(90)

class GUI():

    root = Tk()

    #This is the canvas the turtle will exist in
    canvas = Canvas(root, width = 500, height = 500)
    canvas.pack()

    draw_chart(canvas)

    root.mainloop()

GUI()
```

Turtles—Linear Equations

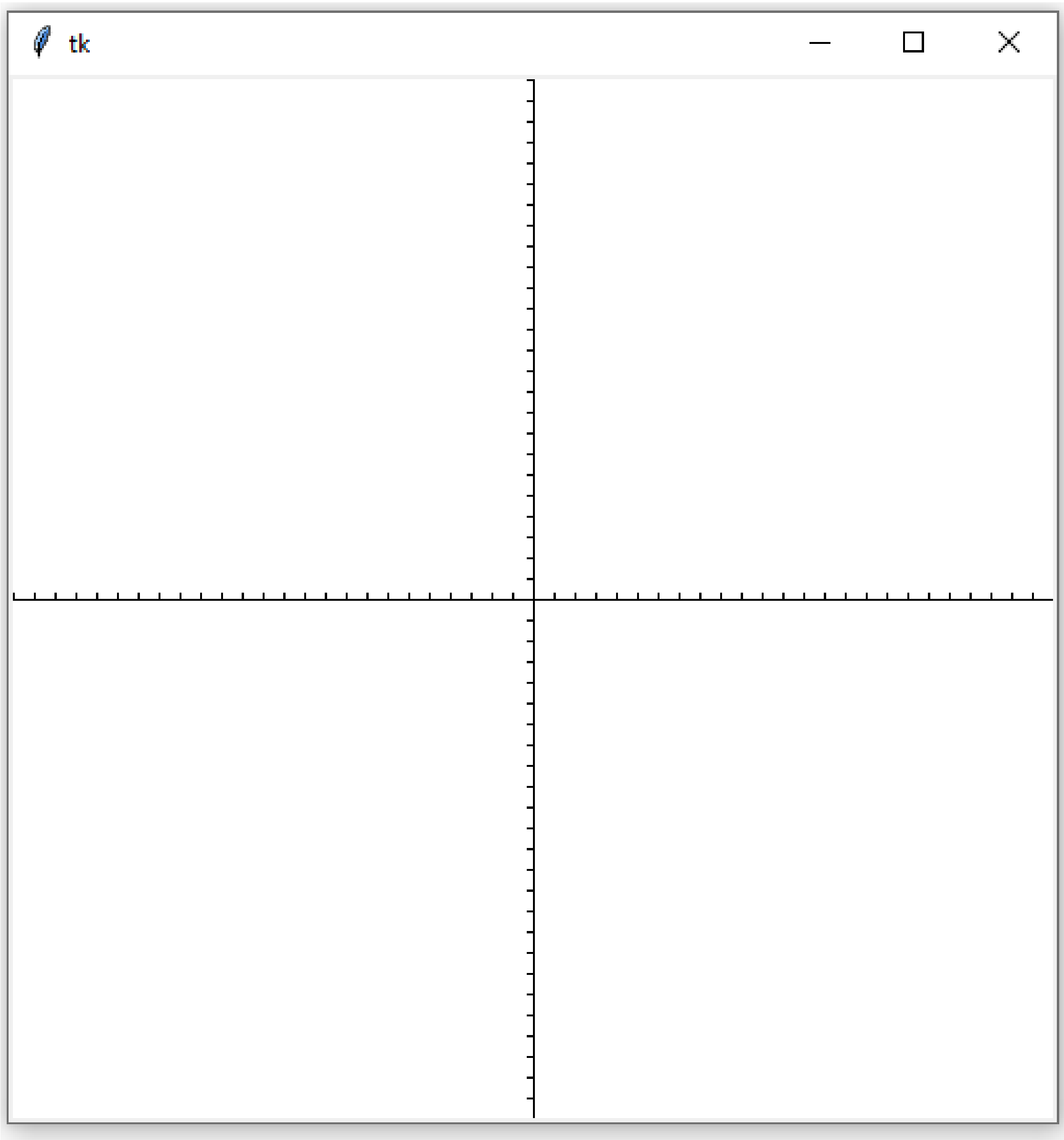
We now have a graph with 0,0 in the centre.

The turtle that did this has been hidden away.

You could have used canvas.create_line() to do this, but it would have used the top left as 0,0. It would have rendered instantly rather than drawn in, but would have required a lot of coding to get right.

The scale has been done every 10 pixels, however what these values represent will be defined later.

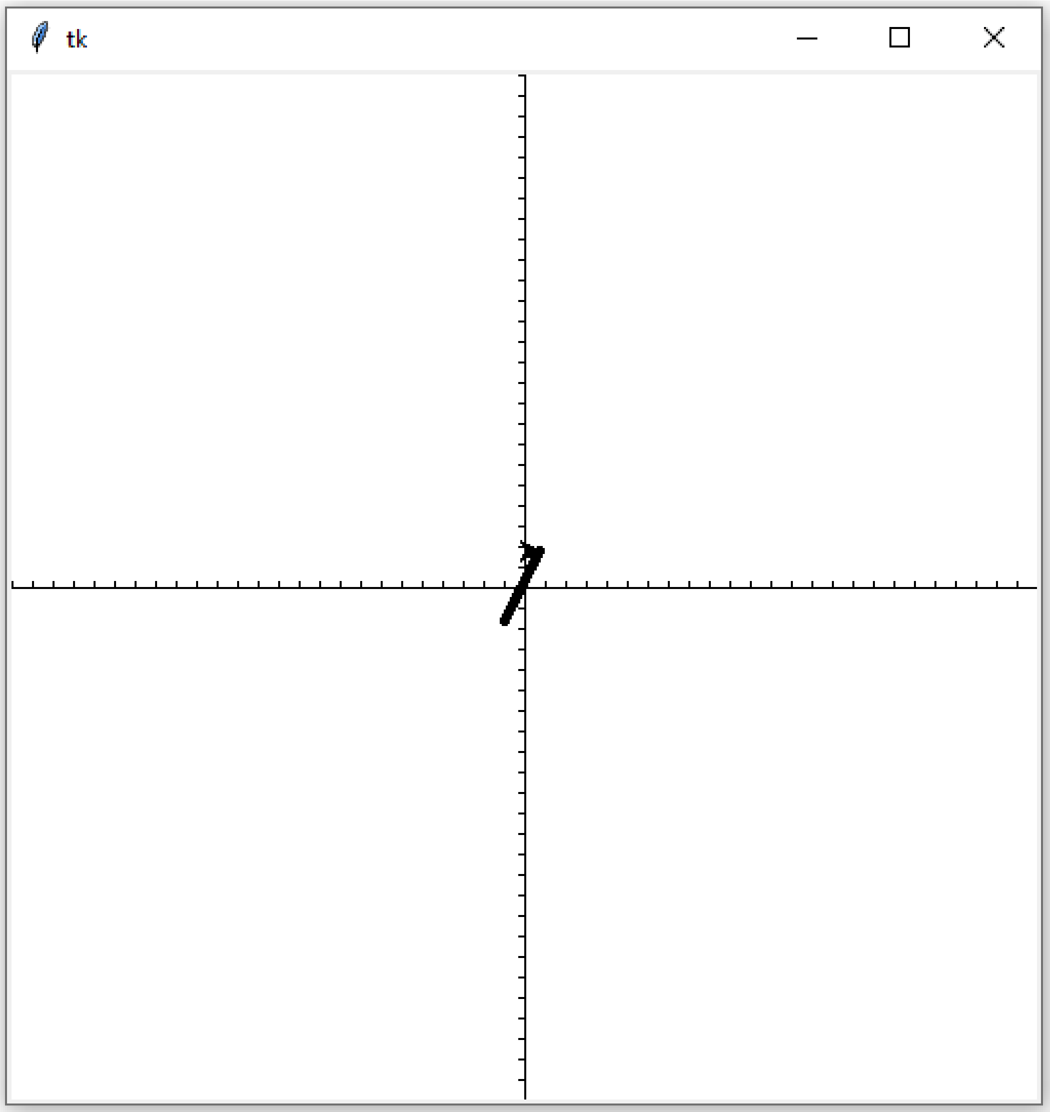
If you wish alter the code to space out the scales.



Now we move onto the next issue with drawing straight lines.

Turtles—Linear Equations

We have the issue of pixels. Suggest you want to plot a chart that goes from 0 to 100. No problem until your realise that 100 pixels is tiny. Barely 5% of a standard screen. Look at how small x - 10 to x 8 is in pixels.



This means we need to use some kind of factoring to space out the pixels, while maintaining scale.

This is where was start thinking what the points on the scale actually represent.

The canvas is 500px by 500px so we could start to consider creating a factor that increases the scale.

Turtles—Linear Equations

It is now a good time to begin building LinearTurtle.

Certain design decisions are already taken, such as it being a child of RawTurtle.

We have already decided it will need a factor variable and of course will need getter and setter methods.

So far our design can be done as below.

Class table: LinearTurtle—Inherits RawTurtle

Fields

Private factor **As** Integer *'Sets a scale to increase the distance between points.*

Properties

Public Read Write p_factor() **As** Integer

Constructor

Public New (canvas **As** Canvas) *'Use a canvas to hold the turtle.*

Methods

None

Pseudocode

'Constructor for the class.

'Preconditions: Takes one parameter, a canvas to hold the turtle.

Public Constructor New (canvas **As** canvas)

    Super Constructor New(canvas **As** canvas)

End Constructor

Turtles—Linear Equations

Now we can start coding our LinearTurtle based on this early design.

As you can see it has the variable __factor, an __init__ constructo, and the getter/setter for the variable.

```
import turtle
class LinearTurtle (turtle.RawTurtle):

    #Variables
    __factor = 10

    #Constructor
    def __init__(self,canvas):
        super(LinearTurtle,self).__init__(canvas)


    #Getter and setter methods
    def set_factor(self, change):
        self.__factor = change

    def get_factor(self):
        return self.__factor
```

It won't do much yet, but you can instantiate it in Chart.py now.

```
class GUI():

    root = Tk()

    #This is the canvas the turtle will exist in
    canvas = Canvas(root, width = 500, height = 500)
    canvas.pack()

    draw_chart(canvas)


    turtles = [LinearTurtle(canvas)]

    root.mainloop()
```

Turtles—Linear Equations

Now we need to get the turtle to actually create a point on the chart.

This is very easy after all if the canvas is just like some graph paper logic dictate the procedure to do this is as simple as plotting an x, y.

Here's the trick though, as said before we need to factor for the size of a pixel.

You may have seen __factor was defaulted to 10 in Python, you will see in the design how it will be used.

Class table: LinearTurtle—Inherits RawTurtle

Methods

plot_points(x **As** Real, y **As** Real)

Pseudocode

'Plots points for a chart, if the turtle's pen is up

'it will also draw a line.

'The position set will be x * the factor and y * the factor.

'Preconditions: Takes two parameters an x and y coordiate as real numbers.

'Postcondtions: None

Public Procedure plot_points(x **As** Real, y **As** Real)

    setposition(x * factor, y * factor)

    dot()   'Instructs the turtle top place a dot. Not required, but useful for testing.

End Procedure

Turtles—Linear Equations

Now code the plot_points() procedure.

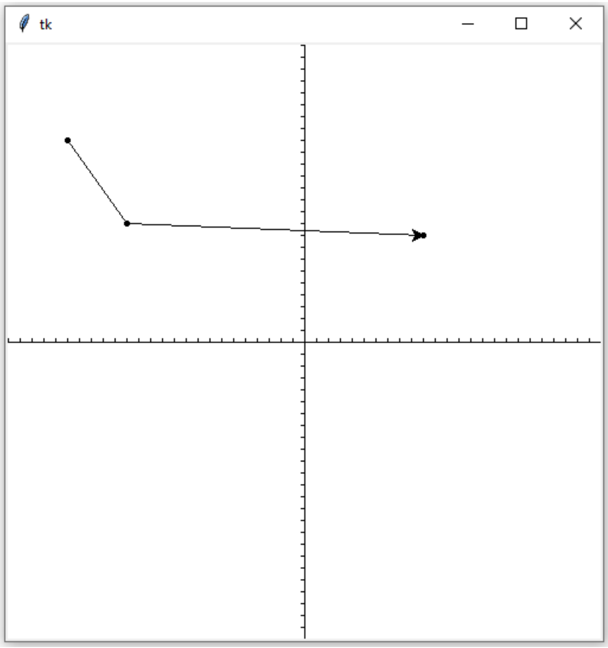
It goes straight after the properties.

```
#Plots points for a chart, if the turtle's pen is up
#it will also draw a line.
#The position set will be x * the factor and y * the factor.
#Preconditions: Takes two parameters an x and y coordiate as real numbers.
#Postcondtions: None
def plot_points(self, x, y):
    self.setpos(x * self.__factor ,y * self.__factor)
    self.dot()
```

It can be tested easily in Chart.py. Plot some points in the following way:

```
turtles = [LinearTurtle(canvas)]
turtles[0].penup()
turtles[0].plot_points(-20, 17)

turtles[0].pendown()
turtles[0].plot_points(-15, 10)
turtles[0].plot_points(10, 9)
root.mainloop()
```



This now gives us an ability to plot points one at a time. The problem is that this is a lot of effort.

We will simplify this by creating a single procedure to achieve this instead.

Turtles—Linear Equations

We now need a way to plot all the points for a given set of coordinates.

This procedure linear_plot() will make use of plot_points().

Below is an updated design.

Class table: LinearTurtle—Inherits RawTurtle

Methods

plot_points(x **As** Real, y **As** Real)

linear_plot(points **As** List of(List of **Real**))

Pseudocode

```
'Uses multiple points to draw a line graph.
'To prevent a line from 0,0 to the first point the trutle's pen is set to up if the point is index 0
'otherwise it put the pen down to draw a line.  it calls plot_point for each index in the list.
 Preconditions: Takes a list of points which are x, y coords.
 'Postconditions: None
Public Procedure linear_plot(points As List of(List of Real))
    For i In points
        If points.index(i) = 0 Then
            penup()
        Else
            pendown()
        End If
        plot_points(i[0], i[1])
    Next i
End Procedure
```

Turtles—Linear Equations

Now code the linear_plot() procedure.

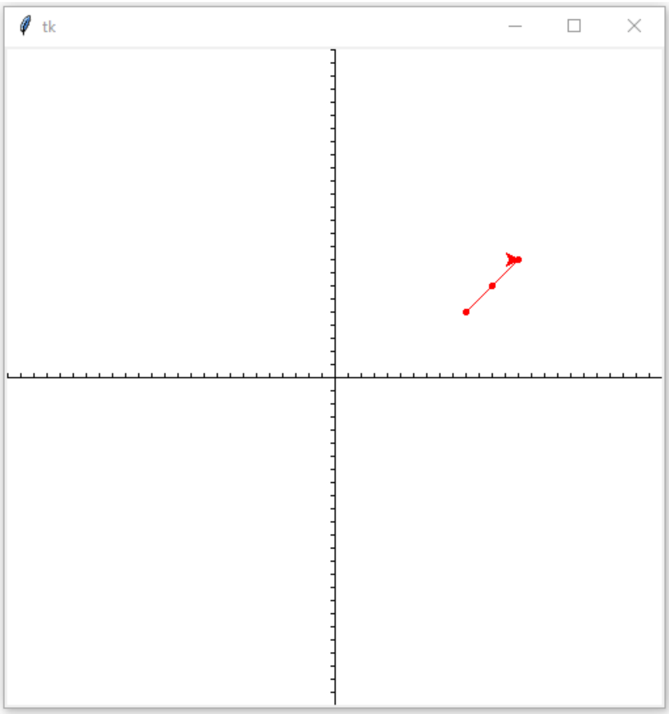
It should go after the plot_points() procedure.

```
#Uses multiple points to draw a line graph.
#To prevent a line from 0,0 to the first point the trutle's pen
#is set to up if the point is index 0
#otherwise it put the pen down to draw a line.
#It calls plot_point for each index in the list.
#Preconditions: Takes a list of points which are x, y coords.
#Postcinditions: None
def linear_plot(self, points):
    for i in points:
        if points.index(i) == 0:
            self.penup()
        else:
            self.pendown()
        self.plot_points(i[0], i[1])
```

You should be able to call the procedure in Chart.py.

```
turtles = [LinearTurtle(canvas)]

turtles[0].penup()
turtles[0].color("red")
turtles[0].linear_plot([[10, 5],[12,7],[14, 9]])
```



Due to the nature of this procedure you should be able to create graphs that have points that are not fully linear, but having this function shows how it is possible to plot graphs in Python.

The next thing to deal with is how does Python actually work out how to draw these lines anyway.

For that we need an actual linear equation.

Turtles—Linear Equations

Now we are going to create an actual linear equation that demonstrates how straight lines are created on a computer.

There are a few different equations depending on what you are trying to work out.

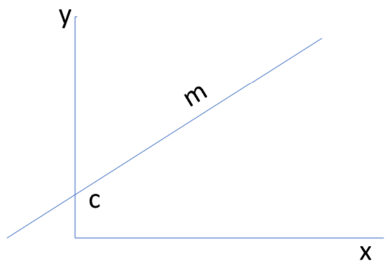
Below are the accepted variables used for these equations along with two equations. One finds the gradient (m), the other a y-coordinate.

$m =$ The slope/gradient

$y =$ Any given point on the y axis

$x =$ Any given point on the x axis

$c =$ The point that the line intercepts the y axis when x = 0



$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

$$y = mx + c$$

Here is the pseudocode for each equation.

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

```
Function findGradient(x1, x2, y1, y2)

    yChange = y2 – y1
    xChange = x2 – x1
    Return (yChange / xChange)

End Function
```

$$y = mx + c$$

```
Procedure lineDraw(x1, x2, m, c)
    For i In range(x1, x2)
        y = (m * x) + c
        plotPoints(x, round(y, 2))
End Procedure
```

There will be more to add to make them work as turtles, but it's a start.

Turtles—Linear Equations

$$y = mx + c$$

We will concentrate on the following procedure for this exercise as it serves to show how lines in computers are actually drawn.

What is of interest is this equation can draw a line between two x points and work out the y points along the way because it has two other pieces of information. *m* the gradient of the line, and *c* the point at which the line with intercept the y-axis.

```
Function lineDraw(x1, x2, m, c)
    For i In range(x1, x2)
        y = (m * x) + c
        plotPoints(x, round(y, 2))
End Function
```

This is called a brute force method of drawing lines and so that is what we will call the procedure in LinearTurtle.

Here's the pseudocode for our program.

```
Public Function brute_force(x1, x2, m, c)
    penup()
    goto(0, c * factor)
    first_point = True
    For x In range(x1, x2)
        If first_point Then    'Needed to stop the turtle drawing the first line to start point
            penup()
            first_point = False
        Else
            pendown()
        End If
        y = (m * x) + c
        plot_points(x, round(y, 2))
End Function
```

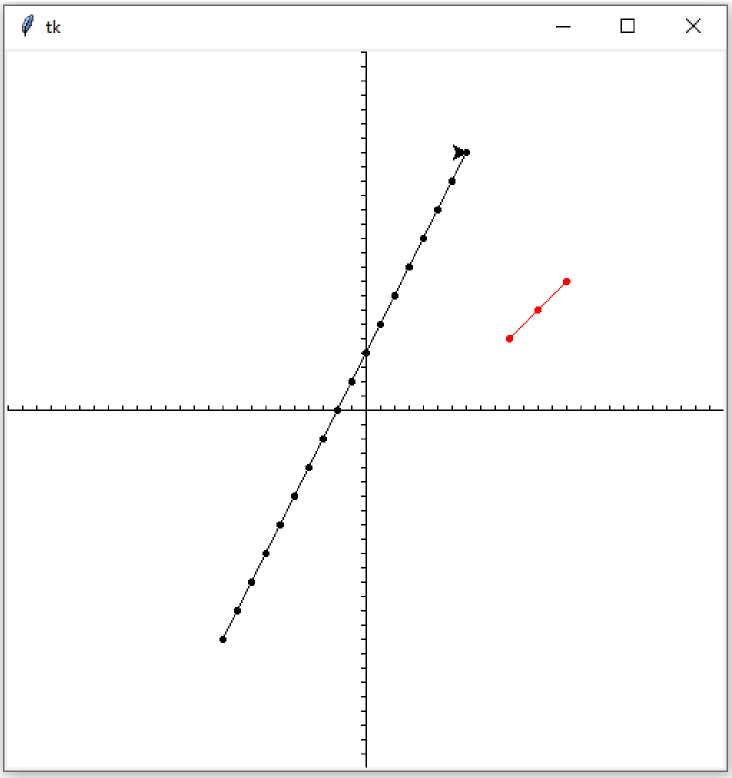
Turtles—Linear Equations

Let's now add the procedure to the LinearTurtle.

```
#A brute force  algorithm for line drawing.
#Preconditions: x1 is a decimal for the first x coord in sequence.
#x2 is a decimal for the last x coord in s a sequence.
#m is the gradient/slope
#c is the point that the point that the line intercepts the y axis when x is 0
#Postconditions: None
def brute_force(self, x1, x2, m, c):
    self.penup()
    self.goto(0 ,c * self.__factor)
    first_point = True
    for x in range(x1, x2): #For loop with the range set from the first to last
        if first_point:
            self.penup()
            first_point = False
        else:
            self.pendown()
        y = (m * x) + c      #Standard maths to find y.
        self.plot_points (x, round(y, 2))
```

Add the call in Chart.py.

```
turtles[0].penup()
turtles[0].color("black")
turtles[0].brute_force(-10, 8, 2, 4)
```



What you will see is the algorithm plotting each point on the way to the end. It's why it is a brute force algorithm, it will just use every point on the way to reach its goal.

Play around with the values given and see what happens with the way it works out the line for the graph.

This is how some programs make line graphs, by just having a start and end, then plotting it.

Very useful if you know the start and end values and need some mid-value data.

Turtles—Linear Equations

Challenge

This function can also be created to work out the gradient of a line based on to x, y coordinates. This would be a useful addition to LinearTurtle as it could potentially use this to work out a degree of turn.

The pseudocode, equation and Python are below, but it is your job to make them work in LinearTurtle.

Function findGradient(x1, x2, y1, y2)

yChange = y2 – y1

xChange = x2 – x1

Return (yChange / xChange)

End Function

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

File Edit Format Run Options Window Help

```
def findGradient(x1, x2, y1, y2):
    yChange = y2 - y1
    xChange = x2 - x1
    return yChange / xChange


gradient = findGradient(2, 3, 4, 6)
print(gradient)
```