In [1]:

```
# IMPORT TOOLS AND LIBRARIES
import numpy as np
import matplotlib.pyplot as plt #visualization
import pandas as pd
# 1st method which is finding best fit line using mathematical formula
```

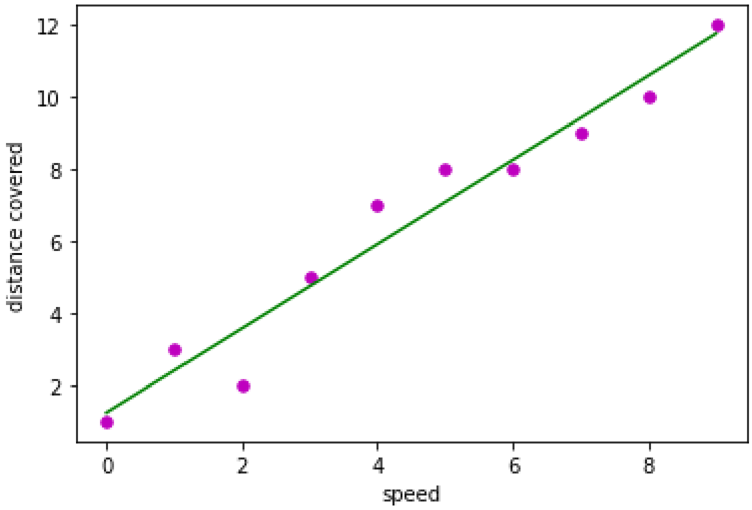
In [2]:

```
# Without going into the mathematical details, we present the result here:
# \beta_1 = \frac{SS_{xy}}{SS_{xx}}
# \beta_0 = \bar{y} - \beta_1\bar{x}
# where SS_xy is the sum of cross-deviations of y and x:
# SS_{xy} = \sum_{i=1}^{n} (x_i-\bar{x})(y_i-\bar{y}) = \sum_{i=1}^{n} y_ix_i - n\bar{x}\ba
# and SS_xx is the sum of squared deviations of x:
# SS_{xx} = \sum_{i=1}^{n} (x_i-\bar{x})^2 = \sum_{i=1}^{n}x_i^2 - n(\bar{x})^2
```

In [4]:

```
def estimate_coff(x,y):
    n=np.size(x) # size =10

#     numpy provides mean()
    mean_x=np.mean(x)
    mean_y = np.mean(y)
#     ss_xy
    SS_xy=np.sum(y*x)-n*(mean_x)*(mean_y)
    SS_xx=np.sum(x*x)-n*(mean_x)*(mean_x)

# finding coff
    b_1=SS_xy/SS_xx
    b_0= mean_y - b_1*mean_x
    return (b_0,b_1)

# function for best fit line
def plot_reg_line(x,y,b):
#     plot a scatter plot
    plt.scatter(x,y,color="m",marker="o",s=30)
# predicted response vector
    y_pred = b[0] + b[1]*x
# plotting the regression line
    plt.plot(x, y_pred, color = "g")
#     put labels
    plt.xlabel('speed')
    plt.ylabel('distance covered')

#     show
    plt.show()




def main():
#     observations
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])
# estimating coefficients
    b= estimate_coff(x,y)
    print("Estimated coefficients:\nb_0 = {} \
          \nb_1 = {}".format(b[0], b[1]))

#     plot a best fit line
    plot_reg_line(x,y,b)

if __name__== "__main__":
    main()
```

```
Estimated coefficients:
b_0 = 1.2363636363636363
b_1 = 1.1696969696969697
```


distance covered
speed
0
2
4
6
8
2
4
6
8
10
12

In [1]:

```
# importing library and tools
import matplotlib.pyplot as plt #visualisation
```

In [2]:

```
import numpy as np #numerical python which is used for matrix calculations
aman= np.genfromtxt("LinearReg_Univariate.txt",delimiter=',')#read the text file # o
print(aman)
print(len(aman))#length of dataset here m=97
```

```
[[ 6.1101  17.592  ]
 [ 5.5277   9.1302 ]
 [ 8.5186  13.662  ]
 [ 7.0032  11.854  ]
 [ 5.8598   6.8233 ]
 [ 8.3829  11.886  ]
 [ 7.4764   4.3483 ]
 [ 8.5781  12.      ]
 [ 6.4862   6.5987 ]
 [ 5.0546   3.8166 ]
 [ 5.7107   3.2522 ]
 [14.164   15.505  ]
 [ 5.734    3.1551 ]
 [ 8.4084   7.2258 ]
 [ 5.6407   0.71618]
 [ 5.3794   3.5129 ]
 [ 6.3654   5.3048 ]
 [ 5.1301   0.56077]
 [ 6.4296   3.6518 ]
 [ 7.0708   5.3893 ]
 [ 6.1891   3.1386 ]
 [20.27    21.767  ]
 [ 5.4901   4.263  ]
 [ 6.3261   5.1875 ]
 [ 5.5649   3.0825 ]
 [18.945   22.638  ]
 [12.828   13.501  ]
 [10.957    7.0467 ]
 [13.176   14.692  ]
 [22.203   24.147  ]
 [ 5.2524  -1.22    ]
 [ 6.5894   5.9966 ]
 [ 9.2482  12.134  ]
 [ 5.8918   1.8495 ]
 [ 8.2111   6.5426 ]
 [ 7.9334   4.5623 ]
 [ 8.0959   4.1164 ]
 [ 5.6063   3.3928 ]
 [12.836   10.117  ]
 [ 6.3534   5.4974 ]
 [ 5.4069   0.55657]
 [ 6.8825   3.9115 ]
 [11.708    5.3854 ]
 [ 5.7737   2.4406 ]
 [ 7.8247   6.7318 ]
 [ 7.0931   1.0463 ]
 [ 5.0702   5.1337 ]
 [ 5.8014   1.844  ]
 [11.7      8.0043 ]
 [ 5.5416   1.0179 ]
 [ 7.5402   6.7504 ]
 [ 5.3077   1.8396 ]
 [ 7.4239   4.2885 ]
 [ 7.6031   4.9981 ]
 [ 6.3328   1.4233 ]
 [ 6.3589  -1.4211 ]
 [ 6.2742   2.4756 ]
```

```
 [ 5.6397   4.6042 ]
 [ 9.3102   3.9624 ]
 [ 9.4536   5.4141 ]
 [ 8.8254   5.1694 ]
 [ 5.1793  -0.74279]
 [21.279   17.929  ]
 [14.908   12.054  ]
 [18.959   17.054  ]
 [ 7.2182   4.8852 ]
 [ 8.2951   5.7442 ]
 [10.236    7.7754 ]
 [ 5.4994   1.0173 ]
 [20.341   20.992  ]
 [10.136    6.6799 ]
 [ 7.3345   4.0259 ]
 [ 6.0062   1.2784 ]
 [ 7.2259   3.3411 ]
 [ 5.0269  -2.6807 ]
 [ 6.5479   0.29678]
 [ 7.5386   3.8845 ]
 [ 5.0365   5.7014 ]
 [10.274    6.7526 ]
 [ 5.1077   2.0576 ]
 [ 5.7292   0.47953]
 [ 5.1884   0.20421]
 [ 6.3557   0.67861]
 [ 9.7687   7.5435 ]
 [ 6.5159   5.3436 ]
 [ 8.5172   4.2415 ]
 [ 9.1802   6.7981 ]
 [ 6.002    0.92695]
 [ 5.5204   0.152  ]
 [ 5.0594   2.8214 ]
 [ 5.7077   1.8451 ]
 [ 7.6366   4.2959 ]
 [ 5.8707   7.2029 ]
 [ 5.3054   1.9869 ]
 [ 8.2934   0.14454]
 [13.394    9.0551 ]
 [ 5.4369   0.61705]]
97
```

In [4]:

```
x=aman[:,0].reshape(-1,1)# here first we have used slicing and then reshaped we use
print(x)# gives x

print(x.shape) # gives the shape
```

```
[[ 6.1101]
 [ 5.5277]
 [ 8.5186]
 [ 7.0032]
 [ 5.8598]
 [ 8.3829]
 [ 7.4764]
 [ 8.5781]
 [ 6.4862]
 [ 5.0546]
 [ 5.7107]
 [14.164 ]
 [ 5.734 ]
 [ 8.4084]
 [ 5.6407]
 [ 5.3794]
 [ 6.3654]
 [ 5.1301]
 [ 6.4296]
 [ 7.0708]
 [ 6.1891]
```

```
[20.27  ]
[ 5.4901]
[ 6.3261]
[ 5.5649]
[18.945 ]
[12.828 ]
[10.957 ]
[13.176 ]
[22.203 ]
[ 5.2524]
[ 6.5894]
[ 9.2482]
[ 5.8918]
[ 8.2111]
[ 7.9334]
[ 8.0959]
[ 5.6063]
[12.836 ]
[ 6.3534]
[ 5.4069]
[ 6.8825]
[11.708 ]
[ 5.7737]
[ 7.8247]
[ 7.0931]
[ 5.0702]
[ 5.8014]
[11.7   ]
[ 5.5416]
[ 7.5402]
[ 5.3077]
[ 7.4239]
[ 7.6031]
[ 6.3328]
[ 6.3589]
[ 6.2742]
[ 5.6397]
[ 9.3102]
[ 9.4536]
[ 8.8254]
[ 5.1793]
[21.279 ]
[14.908 ]
[18.959 ]
[ 7.2182]
[ 8.2951]
[10.236 ]
[ 5.4994]
[20.341 ]
[10.136 ]
[ 7.3345]
[ 6.0062]
[ 7.2259]
[ 5.0269]
[ 6.5479]
[ 7.5386]
[ 5.0365]
[10.274 ]
[ 5.1077]
[ 5.7292]
[ 5.1884]
[ 6.3557]
[ 9.7687]
[ 6.5159]
[ 8.5172]
[ 9.1802]
[ 6.002 ]
[ 5.5204]
[ 5.0594]
```

```
 [ 5.7077]
 [ 7.6366]
 [ 5.8707]
 [ 5.3054]
 [ 8.2934]
 [13.394 ]
 [ 5.4369]]
(97, 1)
```

In [5]:

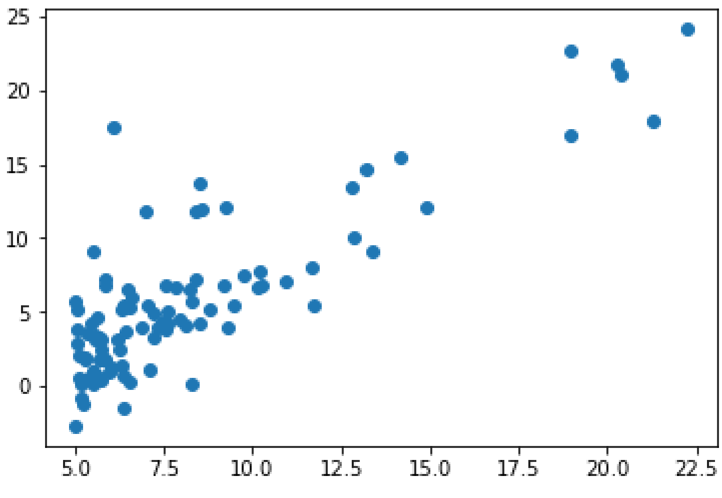
```
y=aman[:,1].reshape(-1,1)# slicing about column y or col[1]
print(y)# gives y
print(y.shape)
```

```
[[17.592  ]
 [ 9.1302 ]
 [13.662  ]
 [11.854  ]
 [ 6.8233 ]
 [11.886  ]
 [ 4.3483 ]
 [12.     ]
 [ 6.5987 ]
 [ 3.8166 ]
 [ 3.2522 ]
 [15.505  ]
 [ 3.1551 ]
 [ 7.2258 ]
 [ 0.71618]
 [ 3.5129 ]
 [ 5.3048 ]
 [ 0.56077]
 [ 3.6518 ]
 [ 5.3893 ]
 [ 3.1386 ]
 [21.767  ]
 [ 4.263  ]
 [ 5.1875 ]
 [ 3.0825 ]
 [22.638  ]
 [13.501  ]
 [ 7.0467 ]
 [14.692  ]
 [24.147  ]
 [-1.22   ]
 [ 5.9966 ]
 [12.134  ]
 [ 1.8495 ]
 [ 6.5426 ]
 [ 4.5623 ]
 [ 4.1164 ]
 [ 3.3928 ]
 [10.117  ]
 [ 5.4974 ]
 [ 0.55657]
 [ 3.9115 ]
 [ 5.3854 ]
 [ 2.4406 ]
 [ 6.7318 ]
 [ 1.0463 ]
 [ 5.1337 ]
 [ 1.844  ]
 [ 8.0043 ]
 [ 1.0179 ]
 [ 6.7504 ]
 [ 1.8396 ]
 [ 4.2885 ]
 [ 4.9981 ]
 [ 1.4233 ]
```

```
 [-1.4211 ]
 [ 2.4756 ]
 [ 4.6042 ]
 [ 3.9624 ]
 [ 5.4141 ]
 [ 5.1694 ]
 [-0.74279]
 [17.929  ]
 [12.054  ]
 [17.054  ]
 [ 4.8852 ]
 [ 5.7442 ]
 [ 7.7754 ]
 [ 1.0173 ]
 [20.992  ]
 [ 6.6799 ]
 [ 4.0259 ]
 [ 1.2784 ]
 [ 3.3411 ]
 [-2.6807 ]
 [ 0.29678]
 [ 3.8845 ]
 [ 5.7014 ]
 [ 6.7526 ]
 [ 2.0576 ]
 [ 0.47953]
 [ 0.20421]
 [ 0.67861]
 [ 7.5435 ]
 [ 5.3436 ]
 [ 4.2415 ]
 [ 6.7981 ]
 [ 0.92695]
 [ 0.152  ]
 [ 2.8214 ]
 [ 1.8451 ]
 [ 4.2959 ]
 [ 7.2029 ]
 [ 1.9869 ]
 [ 0.14454]
 [ 9.0551 ]
 [ 0.61705]]
(97, 1)
```

In [6]:

```
plt.scatter(x,y)# we are plotting the scatter plot between x and y
plt.show()# shows the plot
```



In [9]:

```
s=np.ones(97)# it creates the ones of the matrix creates x0
s
```

```
a=s.reshape(-1,1) # it is reshaped in order to get in concatenation with x1
print(a)
```

```
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
```

```
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

In [10]:

```
z=np.concatenate([a,x],1)
print(z)# creates a matrix of [x0 x1]
print(z.shape) # creating the shape (97,2)
```

```
[[ 1.      6.1101]
 [ 1.      5.5277]
 [ 1.      8.5186]
 [ 1.      7.0032]
 [ 1.      5.8598]
 [ 1.      8.3829]
 [ 1.      7.4764]
 [ 1.      8.5781]
 [ 1.      6.4862]
 [ 1.      5.0546]
 [ 1.      5.7107]
 [ 1.     14.164 ]
 [ 1.      5.734 ]
 [ 1.      8.4084]
 [ 1.      5.6407]
 [ 1.      5.3794]
 [ 1.      6.3654]
 [ 1.      5.1301]
 [ 1.      6.4296]
 [ 1.      7.0708]
 [ 1.      6.1891]
 [ 1.     20.27  ]
 [ 1.      5.4901]
 [ 1.      6.3261]
 [ 1.      5.5649]
 [ 1.     18.945 ]
 [ 1.     12.828 ]
 [ 1.     10.957 ]
 [ 1.     13.176 ]
 [ 1.     22.203 ]
 [ 1.      5.2524]
 [ 1.      6.5894]
```

```
 [ 1.      9.2482]
 [ 1.      5.8918]
 [ 1.      8.2111]
 [ 1.      7.9334]
 [ 1.      8.0959]
 [ 1.      5.6063]
 [ 1.     12.836 ]
 [ 1.      6.3534]
 [ 1.      5.4069]
 [ 1.      6.8825]
 [ 1.     11.708 ]
 [ 1.      5.7737]
 [ 1.      7.8247]
 [ 1.      7.0931]
 [ 1.      5.0702]
 [ 1.      5.8014]
 [ 1.     11.7   ]
 [ 1.      5.5416]
 [ 1.      7.5402]
 [ 1.      5.3077]
 [ 1.      7.4239]
 [ 1.      7.6031]
 [ 1.      6.3328]
 [ 1.      6.3589]
 [ 1.      6.2742]
 [ 1.      5.6397]
 [ 1.      9.3102]
 [ 1.      9.4536]
 [ 1.      8.8254]
 [ 1.      5.1793]
 [ 1.     21.279 ]
 [ 1.     14.908 ]
 [ 1.     18.959 ]
 [ 1.      7.2182]
 [ 1.      8.2951]
 [ 1.     10.236 ]
 [ 1.      5.4994]
 [ 1.     20.341 ]
 [ 1.     10.136 ]
 [ 1.      7.3345]
 [ 1.      6.0062]
 [ 1.      7.2259]
 [ 1.      5.0269]
 [ 1.      6.5479]
 [ 1.      7.5386]
 [ 1.      5.0365]
 [ 1.     10.274 ]
 [ 1.      5.1077]
 [ 1.      5.7292]
 [ 1.      5.1884]
 [ 1.      6.3557]
 [ 1.      9.7687]
 [ 1.      6.5159]
 [ 1.      8.5172]
 [ 1.      9.1802]
 [ 1.      6.002 ]
 [ 1.      5.5204]
 [ 1.      5.0594]
 [ 1.      5.7077]
 [ 1.      7.6366]
 [ 1.      5.8707]
 [ 1.      5.3054]
 [ 1.      8.2934]
 [ 1.     13.394 ]
 [ 1.      5.4369]]
(97, 2)
```

In [22]:

```
# defining the parameters and hyperparameters
```

```
alpha =0.0001 # hyperparameter
iters=10000 #epochs
theta=np.array([[1.0,1.0]])# array of thetas 1x2 dimensions
```

In [20]:

```
def computecost(z,y,theta):
    dot=np.power(((z@theta.T)-y),2)# squared error
    return np.sum(dot)/(2*len(z))# mean squared error or cost function
print(computecost(z,y,theta))

# theta has to be a n x 1 vector then when you do Matrix-Vector Multiplication (X*th
# Matrix multiplication will create the vector h(x) row by row making the correspond
```

```
10.266520491383504
```

In [23]:

```
# Step 5. Create the Gradient Descent function:

def gradientDescent(z, y, theta, alpha, iters):
    for i in range(iters):
        theta = theta - (alpha/len(z)) * np.sum((z @ theta.T - y) * z, axis=0)# simu
        cost = computecost(z, y, theta)
        # if i % 10 == 0: # just look at cost every ten loops for debugging
        #     print(cost)
    return (theta, cost)

print(gradientDescent(z, y, theta, alpha, iters)) # calling Gradient Descent functio
```

```
(array([[0.16763509, 0.78481943]]), 5.980154966664071)
```

In [11]:

```
# Step 6. Another plot:

plt.scatter(aman[:, 0].reshape(-1,1), y)
g=aman[:, 0].reshape(-1,1)
axes = plt.gca() #get current axes
x_vals = np.array(axes.get_xlim())
y_vals = y[0][0] + g[0][1]* x_vals #the line equation
plt.plot(x_vals, y_vals, '--')
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-4b7c1caf629b> in <module>
      5 axes = plt.gca() #get current axes
      6 x_vals = np.array(axes.get_xlim())
----> 7 y_vals = y[0][0] + g[0][1]* x_vals #the line equation
      8 plt.plot(x_vals, y_vals, '--')

IndexError: index 1 is out of bounds for axis 0 with size 1
```