Introduction to programming

Lecture 7:

Lecturers: Giovanni Casini (giovanni.casini@uni.lu) and Xavier Parent (xavier.parent@uni.lu)

Revised version of material from Clément Guérin

Turtle

The **turtle module** is an introductory graphic tool.

You can use it to make nice figures and diagrams. It is a module, you need to call it with the usual command:

from turtle **import** *

or

import turtle

Working with turtle, a graphic console will appear. On this console there will a an arrow.

The arrow, that is the ***turtle***, can be moved through the console, drawing lines.

You should think of the turtle as a pencil. You can move the pencil using two methods :

- Either you move **forward** in the direction of the turtle by a distance d, calling $\textbf{forward}(d)$; you can change the direction of the turtle $\textbf{left}(\theta)$ or $\textbf{right}(\theta)$, where θ is the number of degrees;
- or you move to given (x, y) coordinates by calling $\textbf{setpos}(x, y)$. The visible coordinates go from -350 to 350.

Note: The command $\textbf{setpos}(x, y)$ does not affect the direction of the turtle.

If you don't need to keep your drawing on screen, use the command $\textbf{exitonclick}()$ each time you draw something. It will allow you to smoothly exit the drawing screen. You can also keep your drawing open and use the command $\textbf{reset}()$ to start a drawing anew.

$\textbf{clearscreen}()$ will clean the grahical console.

In [3]:

```
#square
import turtle

turtle.clearscreen()
my_turtle=turtle.Turtle()
my_turtle.forward(100)
my_turtle.left(90)
my_turtle.forward(100)
my_turtle.left(90)
my_turtle.forward(100)
my_turtle.left(90)
my_turtle.forward(100)
```

In [4]:

```
#hexagon
import turtle

turtle.clearscreen()
my_turtle=turtle.Turtle()
my_turtle.forward(100)
my_turtle.left(60)
my_turtle.forward(100)
my_turtle.left(60)
my_turtle.forward(100)
my_turtle.left(60)
my_turtle.forward(100)
my_turtle.left(60)
my_turtle.forward(100)
my_turtle.left(60)
my_turtle.forward(100)
```

In [1]:

```
#hexagon usinf a For loop
import turtle

my_turtle=turtle.Turtle()
for x in range(0,6):
    my_turtle.forward(100)
    my_turtle.left(60)
```

In [2]:

```
#setpos method
import turtle

turtle.clearscreen()
turtle.setpos(100,0)
turtle.setpos(100,100)
turtle.setpos(0,100)
turtle.setpos(0,0)
```

Exercise:

write a program that draws any regular polygon, after asking from the user the number of sides.

In [22]:

```
#Exercise

import turtle

n=int(input('how many sides?' ))
d=360/n
turtle.clearscreen()
my_turtle=turtle.Turtle()
for x in range(0,n):
    my_turtle.forward(100)
    my_turtle.left(d)
```

```
how many sides?20
```

If you want to move the turtle without drawing a line, you need to call the command

penup()

After this, you should write the command

pendown()

in order to be able to write again.

You can also set a speed of the turtle with **speed( )**,

and the colour of the lines with **color('*nameofcolour')**

In [1]:

```
#Equilateral Triangle

from turtle import *

clearscreen()
speed(1)
penup() # If you don't do this you will write behind the turtle
setpos(-100,-100)
setpos(222,120)# The turtle just moves to (-100,-100) without drawing any line
setpos(-100,-100)
color('forestgreen')
pendown() # Now you will write
forward(300) #Move forward in the direction of the turtle
left(120)  #Turn left by 120 degrees
forward(300)
left(120)
forward(300)
```

Exercise:

Write a function that given the arguments: **(lx,ly,x,y)** draw the rectangle with bottom-left corner at (x,y) of horizontal length lx and vertical length ly

In [26]:

```
#Exercise : write a function that given the arguments : (lx,ly,x=0,y=0)
#draw the rectangle with bottom-left corner at (x,y)  by defaut (0,0)
#of horizontal length lx and vertical length ly

#Solution

from turtle import *

clearscreen()
def rect(lx,ly,x,y):
    penup()
    setpos(x,y) #Make sure you don't draw any unwanted lines
    pendown()
    forward(lx)
    left(90)
    forward(ly)
    left(90)
    forward(lx)
    left(90)
    forward(ly)
    left(90)
reset()

#rect(200,100,-22,-100)

a=int(input('give the horizontal lenght:' ))
b=int(input('give the vertical lenght:' ))
c=int(input('give the initial x-coordinate:' ))
d=int(input('give the initial y-coordinate:' ))

rect(a,b,c,d)
```

```
give the horizontal lenght:200
give the vertical lenght:100
give the initial x-coordinate:10
give the initial y-coordinate:20
```

More commands

Let's see a few commands that are useful tools to draw.

- **It is not meant to be exhaustive**. You should have a look on the documentation there for more commands : https://docs.python.org/3.6/library/turtle.html#turtle.reset (https://docs.python.org/3.6/library/turtle.html#turtle.reset).

Circle

We can draw a circle specifying a radius

circle(*d*)

The cicle will be drawn tangent to the direction and position of the turtle.

If the argument d is positive, the turtle will move counterclockwise,

clockwise otherwise.

Fill and colors

You can fill a figure with a color.

To do this you need to call

fillcolor$('red')$ (or any color you want to fill with)

begin_fill()

then draw with the turtle a closed figure and then call

end_fill().

In [3]:

```
# Radioactivity sign
from turtle import *

clearscreen()
reset()
speed(6)
fillcolor('yellow')
penup()
setpos(300,0)
left(90)
pendown()
begin_fill()
circle(300)# Draw the circle of given radius tangent to turtle (direction and po
sition)
end_fill()

right(90)

penup()
fillcolor('black')
setpos(0,0) # The turtle just moves to (0,0) without drawing any line
begin_fill()
pendown() # Now you will write
forward(300) #Move forward in the direction of the turtle
left(120)  #Turn left by 120 degrees
forward(300)
left(120)
forward(300)
end_fill()
begin_fill()
pendown() # Now you will write
forward(300) #Move forward in the direction of the turtle
left(120)  #Turn left by 120 degrees
forward(300)
left(120)
forward(300)
end_fill()
begin_fill()
pendown() # Now you will write
forward(300) #Move forward in the direction of the turtle
left(120)  #Turn left by 120 degrees
forward(300)
left(120)
forward(300)
end_fill()
```

It should be remarked that :

- You can draw with a color and fill with another color.
- The "inside" of the turtle is colored by the color used to fill figures, the "perimeter" of the turtle is colored with the color used to draw the lines.

Dealing with the pace of the drawing

By default the turtle is quite **slow**. We could have guessed it because of its name.

The reason for this is that turtle is supposed to help the learning student to see what the turtle is doing, if the turtle is moving too fast, you don't really see the process happening.

It is possible to temper with the speed of the turtle.

As mentioned above, we can use the command $\mathbf{speed}(x)$ where x is an integer between 0 and 10 (the defaut speed is 6). $x = 1$ is slow and $x = 10$ is fast. $x = 0$ means "no animation".

In [6]:

```
from turtle import *

clearscreen()

speed(1)# slow

#Spiral
N=28
L=10
for x in range(0,N):
    forward(L)
    left(90)
    forward(L)
    left(90)
    L*=1.15
```

In [7]:

```
from turtle import *

clearscreen()

speed(10)# fast

#Spiral
N=28
L=10
for x in range(0,N):
    forward(L)
    left(90)
    forward(L)
    left(90)
    L*=1.15
```

In [34]:

```
reset()

speed(0)# fast

#Spiral
N=28
L=10
for x in range(0,N):
    forward(L)
    left(90)
    forward(L)
    left(90)
    L*=1.15
```

In [9]:

```
# Radioactivity sign
from turtle import *

clearscreen()
reset()
speed(0)
fillcolor('yellow')
penup()
setpos(300,0)
left(90)
pendown()
begin_fill()
circle(300)# Draw the circle of given radius tangent to turtle (direction and po
sition)
end_fill()

right(90)

penup()
fillcolor('black')
setpos(0,0) # The turtle just moves to (0,0) without drawing any line
begin_fill()
pendown() # Now you will write
forward(300) #Move forward in the direction of the turtle
left(120)  #Turn left by 120 degrees
forward(300)
left(120)
forward(300)
end_fill()
begin_fill()
pendown() # Now you will write
forward(300) #Move forward in the direction of the turtle
left(120)  #Turn left by 120 degrees
forward(300)
left(120)
forward(300)
end_fill()
begin_fill()
pendown() # Now you will write
forward(300) #Move forward in the direction of the turtle
left(120)  #Turn left by 120 degrees
forward(300)
left(120)
forward(300)
end_fill()
```

Even though there is no animation, you can see that with $\mathbf{speed}(0)$ it still takes some time to draw.

By calling $\mathbf{speed}(0)$ you are just saying that you don't want to animate the move of your turtle, but it still stops at each step. The duration comes from the small stops that it makes between moves.

Of course, the more steps you have the more time it takes (even though the turtle moves instantaneously).

When you get one million steps to finish a drawing, you don't really need to see all steps of the drawing.

To do this you need to call $\mathbf{tracer}(N)$. Once you call this command, you will only make a drawing every N steps and draw the N steps in one time. Remark that when you call $\mathbf{tracer}(N)$ at some point, you might want to call $\mathbf{tracer}(1)$ at the end (indeed by calling it at the end, you make sure that every step get drown).

Alternatively, you can call $\mathbf{update}()$ at the end.

In [1]:

```
from turtle import *

from random import randrange as rrange

 #Here we draw some random walk on the plane at normal speed
scale=10
def RW():
     x=rrange(0,2)
     y=rrange(0,2)
     return (scale*(2*x-1),scale*(2*y-1))


Nstep=500
x=2
y=3
clearscreen()
tracer(30)
for  u in range(0,Nstep):
    (a,b)=RW()
    x=x+a
    y=y+b
    setpos(x,y)
tracer(1) # After you used tracer(N) do not forget to call tracer(1) because you
          # you may have some steps remaining (if you do not reach N steps you d
on't draw)
```

The $\mathbf{tracer}$ function can also take another argument which is the delay (in milisecond).

By calling $\mathbf{tracer}(10000000)$ you basically make sure that you draw everything in one single step. Remark that it might take some time to draw the figure if the figure is complicated.

I just want to add that even though the animation of the turtle was meant for kids to understand "what the turtle is doing", it really helps with debugging your code. Make sure that when you draw complicated objects (fractals for instance, see the next section), you begin with the animation (for small values of the parameters of course). By doing this, you will be able to debug your code. Once your code works, you can of course increase the speed of your script by using the $\mathbf{tracer}$ function.

A more complete example : the Von Koch snowflake

The Von Koch snowflake is a well known fractal figure. It can be described easily by an iterative process.

First, we define a transformation VK to be applied to any segment. Given a segment $[A, B]$, divides it into 3 identical segments of length ℓ (i.e. $\ell = d(A, B)/3$). Erase the middle subsegment of $[A, B]$ and draw instead a segment of length ℓ leaving from the end of the first subsegment with an angle of 60 degrees and from there a second segment going to the beginning of the third subsegment of $[A, B]$ (use a drawing). It is straightforward to see that both new segments with the erased segment form an equilateral triangle.

The N-th Von Koch snowflake for $N \geq 1$ can be constructed following these steps :

- Step 1. Draw an equilateral triangle.
- Step 2. For each segment in your figure, apply the transformation VK.
- Step 3. If you did not do Step 2 N times, go to Step 2, else you are done.

In [6]:

```
from turtle import *

# Von Koch
#Our beginning equilateral triangle will have a side of length 400
clearscreen()
speed(0)
N=int(input("At which level do you want to construct the Von Koch snowflake?"))



#Here is the base of the algorithm.
#We create the data that we will use to generate our moves
#We have two different moves :
#- The move forward
#- The turning around
#For a given N, the move forward will always be of Length 400/3^N
#The turning around needs to be recorded with their angles.




#Here is what we get at level 0
AuxList=[0,'120',0,'120',0]
# This list will later be interpreted as
# We make a move forward of 400 (400/3^0)
# We turn by 120 degree
# We make a move forward of 400 (400/3^0)
# We turn by 120 degree
# We make a move forward of 400 (400/3^0)
# We turn by 120 degree




for x in range(0,N): #For a given x, we go from the level x to level x+1
    AuxAuxList=[]  # Will be AuxList at level x+1
    for ind in range(0,len(AuxList)):
        if AuxList[ind]==x:          #Every time we come across a move forward a
t level x
                                     #We divide it in 7 moves
            AuxAuxList.append(x+1)# move forward at level x+1
            AuxAuxList.append('-60')# Turn by -60 degrees
            AuxAuxList.append(x+1)#move forward at level x+1
            AuxAuxList.append('120')# Turn by 120 degrees
            AuxAuxList.append(x+1)#move forward at level x+1
            AuxAuxList.append('-60')# Turn by -60 degrees
            AuxAuxList.append(x+1)#move forward at level x+1
        else:
            AuxAuxList.append(AuxList[ind]) #If not a move forward then keep the
record.
    AuxList=AuxAuxList #Replace the x-th level by the x+1-th.


#Now we draw the figure
penup()
setpos(-100,-100) # We begin at (-100,-100)


Length=400/3**N # We compute the caracteristic length of the drawing.
```

```
pendown()
speed(0)
for x in AuxList:
    if type(x)==type(3): #Check if x is an integer.
        forward(Length)#move forward
    else:
        left(int(x))#Else it is a turn around
print(AuxList)
```

```
At which level do you want to construct the Von Koch snowflake?2
[2, '-60', 2, '120', 2, '-60', 2, '-60', 2, '-60', 2, '120', 2, '-6
0', 2, '120', 2, '-60', 2, '120', 2, '-60', 2, '-60', 2, '-60', 2,
'120', 2, '-60', 2, '120', 2, '-60', 2, '120', 2, '-60', 2, '-60',
2, '-60', 2, '120', 2, '-60', 2, '120', 2, '-60', 2, '120', 2, '-6
0', 2, '-60', 2, '-60', 2, '120', 2, '-60', 2, '120', 2, '-60', 2,
'120', 2, '-60', 2, '-60', 2, '-60', 2, '120', 2, '-60', 2, '120',
2, '-60', 2, '120', 2, '-60', 2, '-60', 2, '-60', 2, '120', 2, '-6
0', 2]
```

You can have even be more creative with the animation see the following script.

In [51]:

```
from turtle import *
reset()
speed(0)
penup() # If you don't do this you will write behind the turtle
setpos(-100,-100)
hideturtle()
#This line of code hide "the turtle" it increases the speed of drawing
N=0
def VKsnowFlake():
    tracer(1000000000) #Hold it
    global N
    #Construction of the list
    AuxList=[0,'120',0,'120',0,'120']
    for x in range(0,N):
        AuxAuxList=[]
        for ind in range(0,len(AuxList)):
            if AuxList[ind]==x:
                AuxAuxList.append(x+1)
                AuxAuxList.append('-60')
                AuxAuxList.append(x+1)
                AuxAuxList.append('120')
                AuxAuxList.append(x+1)
                AuxAuxList.append('-60')
                AuxAuxList.append(x+1)
            else:
                AuxAuxList.append(AuxList[ind])
        AuxList=AuxAuxList
    Length=400/3**N
    clear()
    pendown()
    for x in AuxList:
        if type(x)==type(3):
            forward(Length)
        else:
            left(int(x))
    N+=1
    penup()
    setpos(80,-250)
    write("Von Koch snowflake at level {}".format(N), \
          move=False, align="left", font=("Arial", 16, "normal"))
    setpos(-100,-100)
    tracer(1) #Draw
for x in range(0,8):
    speed(0)          #
    VKsnowFlake()     # Draw the x-th snowflake
    speed(5)          #
    right(360)
```

Pyplot in matplotlib

Instead of drawing, you might want to **plot data**.

To do this, you should use the module **matplotlib.pyplot**.

In the sequel we will always denote it **plt**.

It is strongly recommanded to use the **numpy** (as **np**) module as a tool to build data to use **pyplot**.

The documentation can be read here : https://matplotlib.org/api/pyplot_summary.html# (https://matplotlib.org/api/pyplot_summary.html#).

In [52]:

```
import numpy as np
from matplotlib import pyplot as plt
```

How to plot a few points ?

Here we just see how to draw points with the **plot** command. This command can have as many arguments as you want and has keyword arguments as well. The basic call is given by :

$$\textbf{plot}(Lx, Ly)$$

where Lx and Ly are lists of points with **the same number of elements**. You will draw the points $(Lx[i], Ly[i])$ linked by a blue line. You will automatically have a frame with a x-label and a y-label, properly scaled.

If you only gives one list L as an argument, plot will consider it as a y-list and will draw the points $(i, L[i])$.

In [3]:

```
import numpy as np
from matplotlib import pyplot as plt

plt.plot([0,4,3,2],[1, 2, 3, 4]) # Wait there is no graphic there.
```

Out[3]:

```
[<matplotlib.lines.Line2D at 0x11f7d1890>]
```

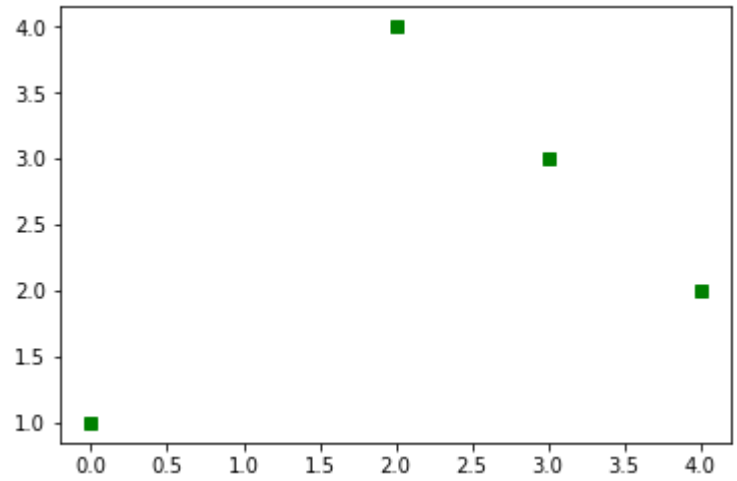
In [5]:

```
plt.show() #Ok, you need to ask to python to show the plot
```

You can change the style of drawing with a string. A priori, the style is given by $'b-'$ meaning that you draw in blue with a link between the points. You should you want to draw orange square without link between them, you should enter 'gs' (for **g**reen and **s**quare).

In [58]:

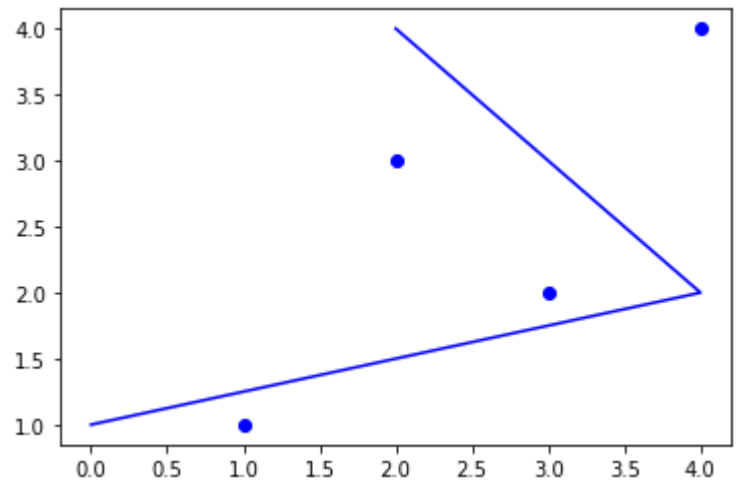
```
plt.plot([0,4,3,2],[1, 2, 3, 4],'gs')
plt.show()
```



You can build different datas on the same plot.

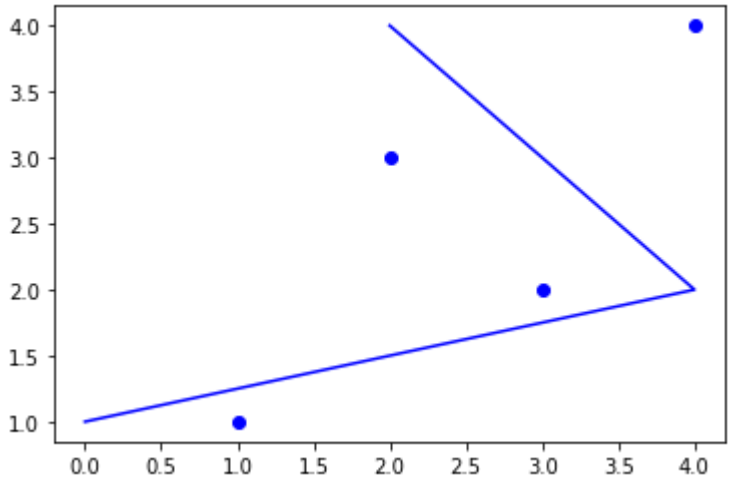
In [6]:

```
plt.plot([0,4,3,2],[1, 2, 3, 4],'b-',[1,3,2,4],[1, 2, 3, 4],'bo') # Here I show
 the points
#The first three arguments are understood as one single set of points
plt.show()
```



In [7]:

```
plt.plot([0,4,3,2],[1, 2, 3, 4],'b-')
plt.plot([1,3,2,4],[1, 2, 3, 4],'bo') # You can add the data  calling twice the
 plot function
plt.show()
```

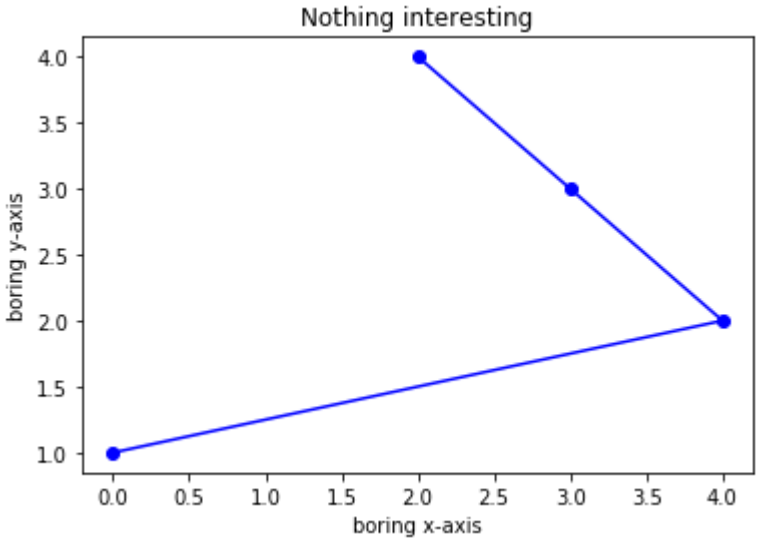


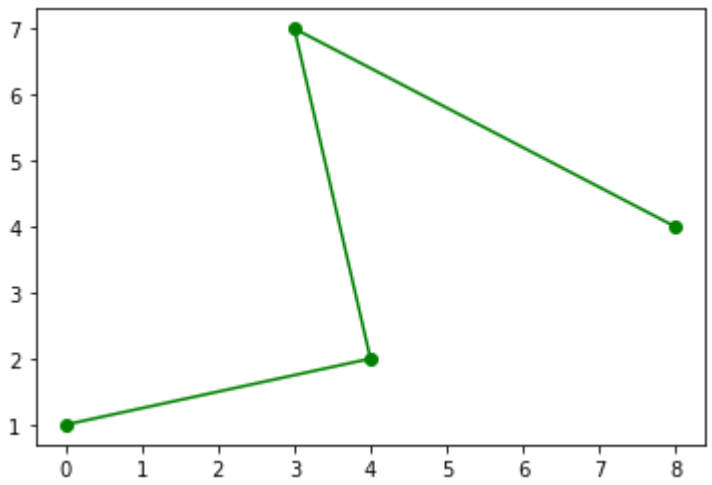
You can add some information to the plot such as a title, a label to the x-axis, the y-axis.

You can also impose the size of the graph.

In [8]:

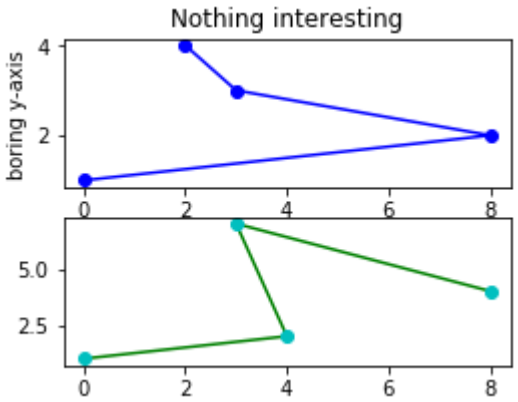
```
plt.plot([0,4,3,2],[1, 2, 3, 4],'b-o')
plt.title("Nothing interesting")
plt.xlabel("boring x-axis")
plt.ylabel("boring y-axis")
plt.show()
plt.plot([0,4,3,8],[1, 2, 7, 4],'g-o')
plt.show() # Once you call show(), it erases all the data in your plot.
```





In [62]:

```
plt.figure(1, figsize=(4, 3)) # Merge everything in one figure of a given size
plt.subplot(211) # This command allows you to draw multiple plots in the figure

plt.plot([0,8,3,2],[1, 2, 3, 4],'b-')
plt.plot([0,8,3,2],[1, 2, 3, 4],'bo')
plt.title("Nothing interesting")
plt.xlabel("boring x-axis")
plt.ylabel("boring y-axis")

plt.subplot(212)

plt.plot([0,4,3,8],[1, 2, 7, 4],'g-')
plt.plot([0,4,3,8],[1, 2,7, 4],'co')

plt.show()
```



How to construct colors ?

In all these drawings, it is nice to be able to draw different colors.

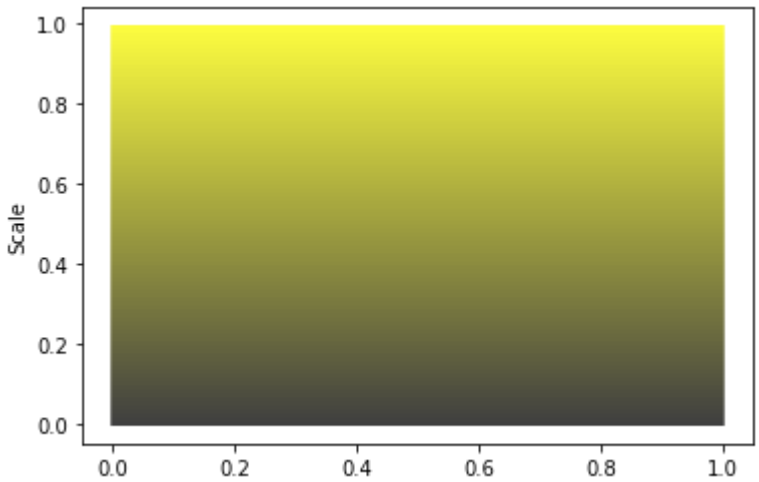
Python allows you to really draw your own colors. Go to https://matplotlib.org/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py (https://matplotlib.org/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py) to see the whole spectrum of built-in colors.

In general, a color can be constructed out of a RGB tuple. Namely you enter the level (as a float between 0 and 1) of red, green and blue.

In [9]:

```
R=float(input('red : '))   #  Enter a float between 0 and 1 (red)
G=float(input('green : ')) #  Enter a float between 0 and 1 (green)
B=float(input('blue : '))  #  Enter a float between 0 and 1 (blue)
N=100
for y in range(0,N):
    scale=y/N
    plt.plot([0,1],[scale,scale],color=(scale*R,scale*G,scale*B)) # Draw a line
 with a given color as a RGB tuple
    #Remark the keyword argument color to draw with the color you want
plt.ylabel("Scale")
plt.show()
```

```
red : 1
green : 1
blue : 0
```



How to plot functions?

In Python, you don't really plot functions $per\ se$, you draw a list of (a big numbers of) linked points instead. In order to do this in an efficient way, you should use the **numpy** module that we have imported as **np**.

Indeed, by calling $x = \mathbf{np}.linspace(m, M, N, endpoint = True)$, you create an array called x which contains N floating numbers evenly spaced between m and M ($endpoint$ is a boolean keyword argument that says if m and M should be contained in it or not).

Of course, by asking a big number of points (namely a big N) we increase the accuracy of our drawing but we also increase the time of computation.

Then if you want to plot some function f, you only need to call $y = f(x)$ and then $\mathbf{plot}(x, y)$. Indeed the call $y = f(x)$ will create an array whose elements are $f(a)$ for a in x.

In the next example, we draw $x \mapsto 3\exp(-x) + x^3$.

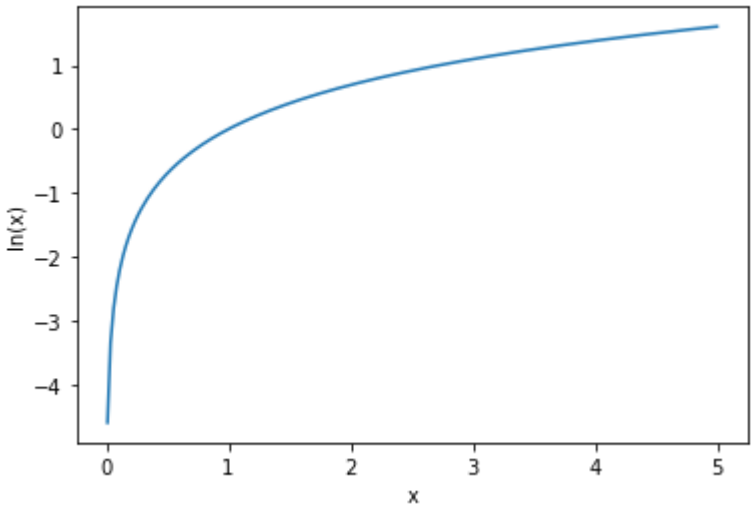
In [69]:

```
x=np.linspace(0,1,3)
y=np.exp(x)
print(x)
print(y)
```

```
[ 0.   0.5  1. ]
[ 1.          1.64872127  2.71828183]
```

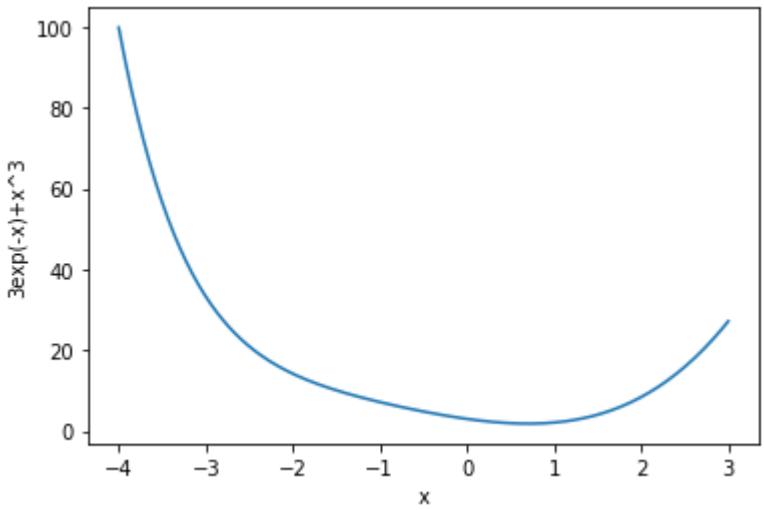
In [14]:

```
import numpy as np
from matplotlib import pyplot as plt

x = np.linspace(0.01, 5, 200, endpoint=True)
y=np.log(x)
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("ln(x)")
plt.show()
```



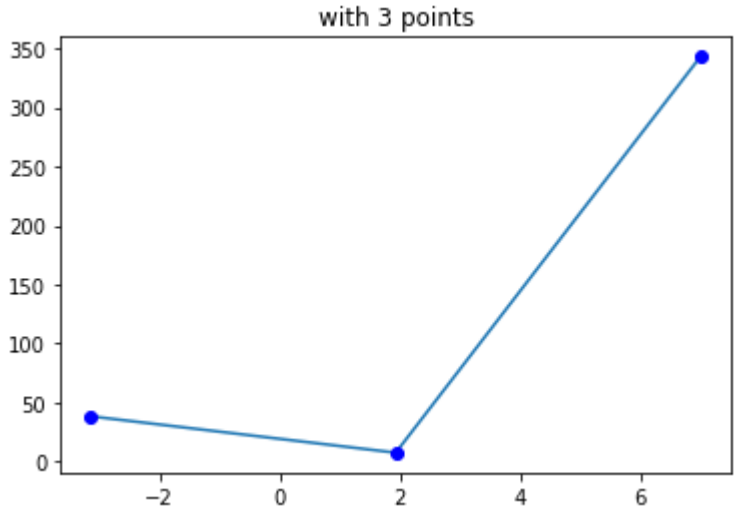
In [15]:

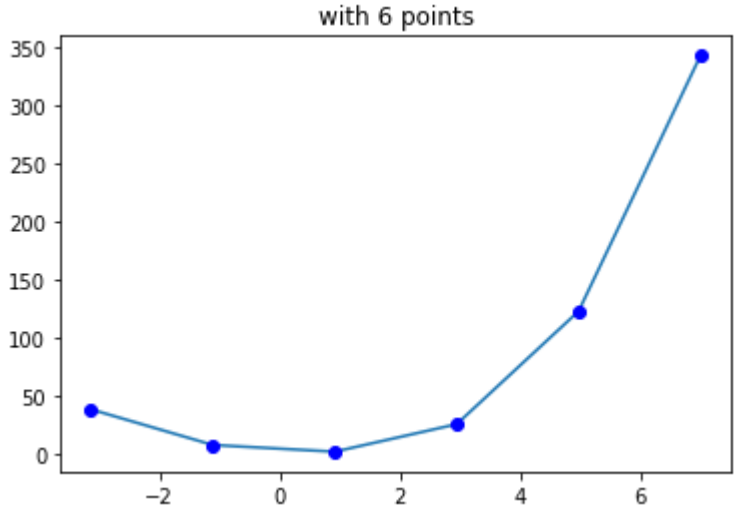
```
import numpy as np
from matplotlib import pyplot as plt

x = np.linspace(-4, 3, 200, endpoint=True)
y=3*np.exp(-x)+x**3
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("3exp(-x)+x^3")
plt.show()
```

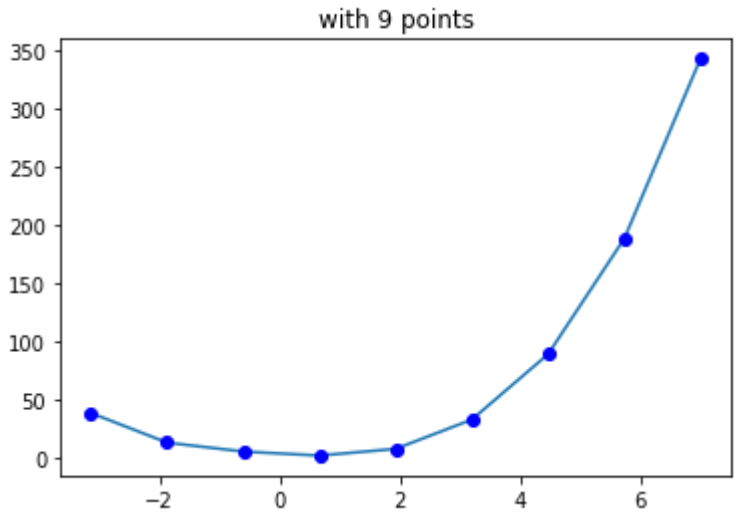


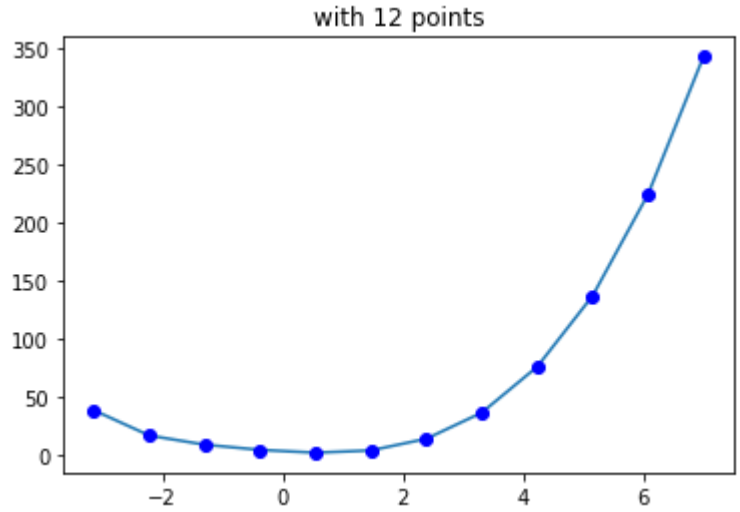
In [16]:

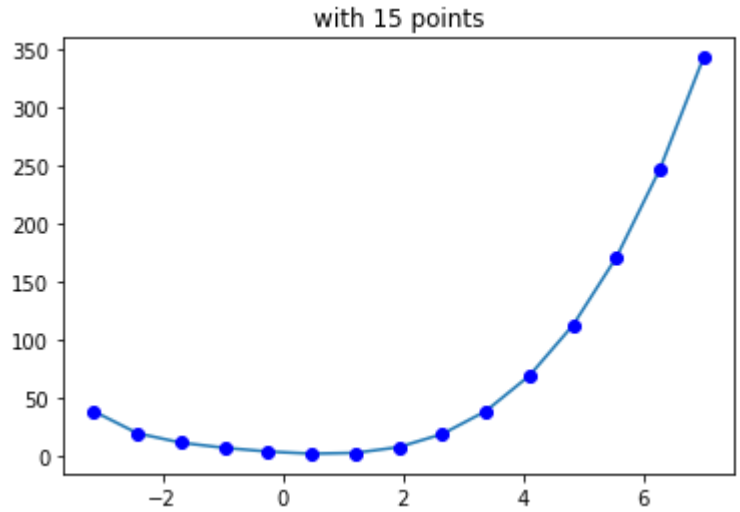
```
#Here are some examples where we ask for a few points only
for u in range(1,6):
    x = np.linspace(-np.pi, 7, 3*u, endpoint=True) # linspace is a very convenie
nt tool to build data set.
    y = 3*np.exp(-x)+x**3
    plt.plot(x, y)
    plt.plot(x, y,'bo') #We also draw the points
    plt.title("with {} points".format(3*u))
    plt.show()
```


with 3 points
350
300
250
200
150
100
50
0
−2
0
2
4
6


with 6 points
350
300
250
200
150
100
50
0
−2
0
2
4
6


with 9 points
350
300
250
200
150
100
50
0
−2
0
2
4
6





How to plot histograms?

A histogram is a way to display some data. Let's say you have a sample of measures that you want to study.

In [21]:

```
import matplotlib.pyplot as plt

x = [1,1,2,3,3,5,7,8,9,10,
     10,11,11,13,13,15,16,17,18,18,
     18,19,20,21,21,23,24,24,25,25,
     25,25,26,26,26,27,27,27,27,27,
     29,30,30,31,33,34,34,34,35,36,
     36,37,37,38,38,39,40,41,41,42,
     43,44,45,45,46,47,48,48,49,50,
     51,52,53,54,55,55,56,57,58,60,
     61,63,64,65,66,68,70,71,72,74,
     75,77,81,83,84,87,89,90,90,91
     ]

plt.hist(x, bins=10)
plt.show()
```