GIIT_AMA_1819_Tutorial

Tutorial de Sage/Python

Programación en Sage

El lenguaje de programación de Sage es Python. Según Wikipedia:

"**Python** es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis muy limpia y que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License,[1] que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores."

Python, es un lenguage con una popularidad creciente, como puede verse, por ejemplo, aquí.

Comenzaremos definiendo los elementos básicos de programación. Como ya los habéis visto en las asignaturas de Programación de primero, sólo explicaremos la sintaxis de Python.

Funciones

Para definir una función en Sage/Python, usamos la palabra clave ***def***, seguida del nombre de la función, las variables de entrada entre paréntesis y dos puntos. En las siguientes líneas se escriben los comandos que queremos que evalúe la función.La identación es muy importante en Python: todo lo que esté con la misma identación será el cuerpo de la función, para terminar la función basta escribir sin identación. Por último, el valor que devuelve la función se determina por la palabra clave ***return***.

```
#Definir funcion de producto de dos numeros
def g(x,y):
    return x*y
```

Podemos usar la función igual que usamos cualquier función matemática.

```
#Ejecutamos la función definida
g(5,4)
```

```
20
```

Las funciones pueden recibir cualquier tipo de dato y devolver tantos datos como queramos, separándolos por comas (en realidad, estamos devolviendo una lista, pero llegaremos a eso más tarde).

```
def sumaProducto(a,b):
  return a+b,a*b
```

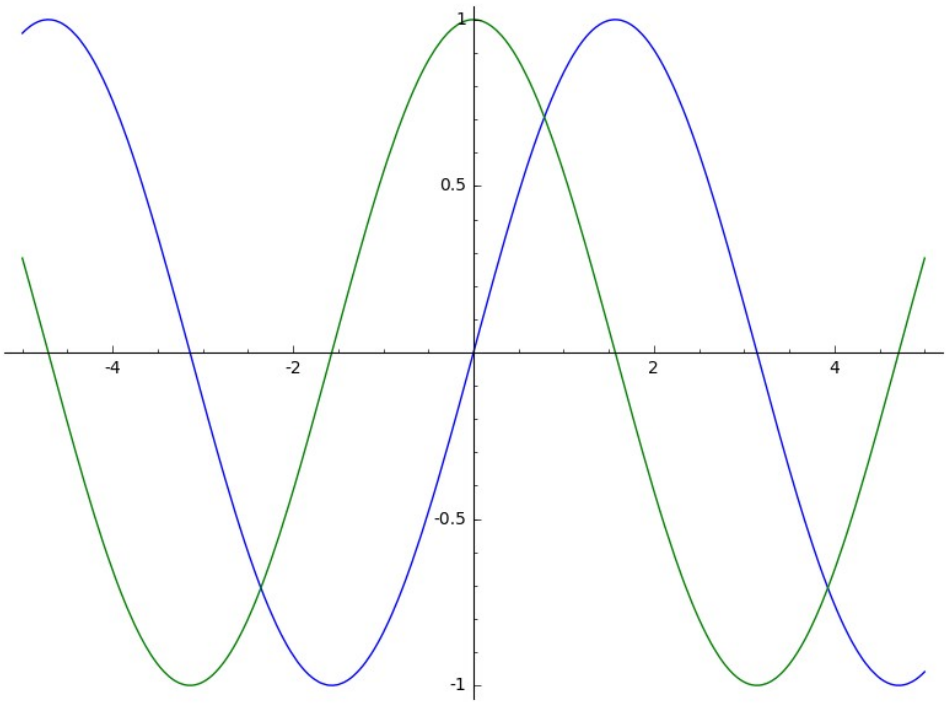
```
sumaProducto(4,3)
```

```
(7, 12)
```

Como parámetro de entrada y salida vale cualquier tipo de objeto de Sage (por ahora sólo intenta comprender qué es lo que hace, poco a poco iremos viendo los distintos elementos que hemos usado en la función).

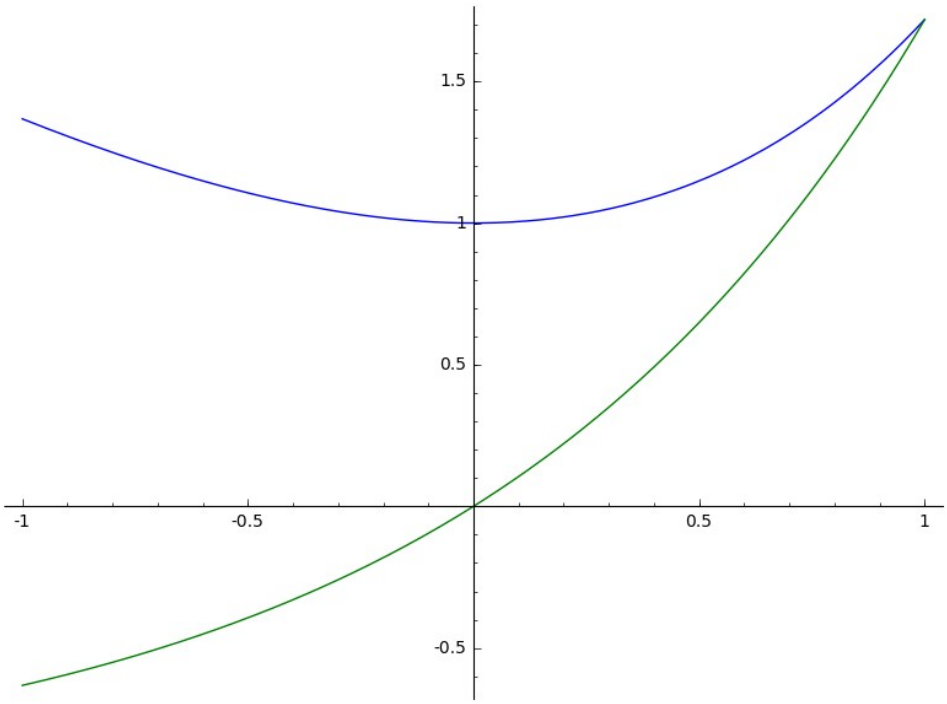
```
def graficaDerivada(f,x,a=-5,b=5):
    return plot(f(x),(x,a,b))+plot(f.diff(x)(x),(x,a,b),color="green")
```

```
f(x)=sin(x)
graficaDerivada(f,x)
```



Si te has fijado bien, en la llamada a la función anterior no hemos dicho nada de los parámetros a y b. Cuando en la definición de un parámetro de la función le asignamos un valor, ese es el valor por defecto. Para cambiarlo, hay que llamar a la función especificando otro valor del parámetro. Se ve mejor con un ejemplo:

```
g(x)=exp(x)-x
graficaDerivada(g,x,a=-1,b=1)
```



Estructuras de control

Las estructuras de control seleccionan cuál es la siguiente sentencia a ejecutar (selección -- if) o hacen que una sentencia se ejecute repetidamente (iteración -- while o for).

Selección - If

La estructura de selección condicional ***if*** *nos* permite elegir ejecutar o no un bloque de instrucciones. La sintaxis es:

if *condición***:**

instrucciones a ejecutar (1 o varias)

También se puede elegir ejecutar otras instrucciones en el caso en el que no se cumpla la condición:

if *condición*:

instrucciones a ejecutar (1 o varias)

else:

instrucciones a ejecutar (1 o varias)

Por ejemplo, vamos a calcular el valor absoluto de un número.

```
a=-5
if a<0:
  a=-a;
a
```

```
5
```

Usando ***else*** ejecutamos una instrucción cuando no se comple la condición (las funciones ***show*** y ***latex*** son para que la salida sea "más bonita") .

```
a=pi/2
if a>3:
  show( latex(a)+ latex(' es mayor que 3'))
else:
  show( latex(a)+ latex(' es menor que 3'))
```

$\frac{1}{2}\pi$ es menor que 3

Podemos combinar varias condiciones con los operadores ***and*** y ***or.***

```
a=pi/2
if a>1 and a<3:
  show( latex(a)+ latex(' está entre 1 y 5'))
else:
  show( latex(a)+ latex(' no está entre 1 y 5'))
```

$\frac{1}{2}\pi$ está entre 1 y 5

Repetición - While

Para repetir una sentencia un número indeterminado de veces, podemos usar un bucle ***while***, con sintaxis:

while *condición*:

instrucciones a ejecutar

Por ejemplo, para "calcular logaritmos", es decir, vamos a ver a cuánto hay que elevar 2 para llegar a 5000 (obviamente, no sería la forma más correcta).

```
i=0;
pot=1;

# Este bucle calcula potencias de 2 hasta llegar a 5000
while pot<5000:
   i=i+1;
   pot=2*pot;

# Hacemos que la salida sea explicativa
print "El logaritmo de 5000 en base dos es",i-1
```

```
El logaritmo de 5000 en base dos es 12
```

Repetición - For

Si queremos repetir una instrucción para cada elemento de un vector, podemos usar un bucle ***for***. La sintaxis es:

for *variable* **in** *lista*:

instrucciones a ejecutar

Vamos a mostrar por pantalla todos los cuadrados del 0 al 4.

```
for i in [0..4]:
   print i^2
```

```
0
1
4
9
16
```

Y vamos a poner en mayúscula la primera letra de cada cadena de texto de una lista.

```
for str in ["juan","mario","eva","laura"]:
        print str.capitalize()
```

```
Juan
Mario
Eva
Laura
```

Ejercicios

1. Crea una función que reciba los coeficientes a, b, c del polinomio $ax^2 + bx + c$ y devuelva las dos raíces del polinomio.

2. Crea una función que reciba dos números y devuelva el mayor de ellos.

3. Crea una función que reciba tres números y devuelva 1 si son iguales y 0 si alguno es distinto a los demás.

4. Crea una función que reciba un número y devuelva 1 si el número es primo y 0 en caso contrario.

Ejercicio 1.

```
def polinomio(a, b, c):
    if (b^2-4*a*c)>=0:
        return show(latex('El resultado es: ')), show((-b+sqrt(b^2-4*a*c))/(2*a)), show((-b-sqrt(b^2-4*a*c))/(2*a))
    else:
        print "No tiene solución entera"
```

```
#La evaluamos
polinomio(1,4,1)
```

El resultado es:

$$\sqrt{3} - 2$$

$$-\sqrt{3} - 2$$

```
(None, None, None)
```

```
polinomio(-6,6,-7)
```

```
No tiene solución entera
```

Ejercicio 2.

```
#Creamos la función
def mayor(a, b):
    if a>b:
        print a, " es mayor que ", b
    else:
        if a<b:
            print b, " es mayor que ", a
        else:
            print a, " y ", b, "son iguales"
```

```
#La evaluamos
mayor(3,6)
```

```
6  es mayor que  3
```

```
mayor(8,6)
```

```
8  es mayor que  6
```

```
mayor(6,6)
```

```
6  y  6 son iguales
```

Ejercicio 3.

```
#Creamos la función
def numeros(a, b, c):
    if a==b and a==c:
        print "1"
    else:
       print "0"
```

```
#La evaluamos
numeros(5,5,5)
```

```
1
```

```
numeros(6,5,5)
```

```
0
```

Ejercicio 4.

is_prime es una funcion booleano, si es primo devuelve true, si es falso devuelve false.
Me aseguro que si se introduce 1 que el programa no muestre nada, ya que el numero 1 no se considera ni primo ni compuesto.

```
#Creamos la función
def primo(a):
    if a>1:
        if is_prime(a):
            print "1"
        else:
            print "0"
```

```
#La evaluamos
primo(1)
```

```
# 3 es un numero primo
primo(3)
```

```
1
```

```
# 6 no es un numero primo
primo(6)
```

```
0
```

Listas

Un tipo de datos muy importante en Sage son las listas. Vamos a ver cómo crear, modificar y seleccionar elementos de ellas.

Una lista no es más que una colección ordenada de elementos. Cada elemento de la lista puede ser cualquier tipo de objeto. Para crear una lista se pone entre corchetes la lista de los elementos, separados por comas.

```
L = [2,3,5,7,11,13,17,2]
```

Para acceder a un elemento se pone el nombre de la lista, seguido de corchetes y entre corchetes el número de elemento al que queremos acceder. El primer elemento se numera con 0.

```
L[0]
```

```
2
```

```
L[0]=3;L
```

```
[3, 3, 5, 7, 11, 13, 17, 2]
```

También podemos comenzar contando desde el último elmento usando índices negativos. Así el último elemento está numerado con -1, el penúltimo con -2, etc

```
L[-2]
```

```
17
```

Podemos seleccionar un segmento de la lista con el operador **:**, siguiendo la convención nombreDeLista[primerElemento:ultimoElemento].

```
L[2:5]
```

```
[5, 7, 11]
```

Si queremos elegir hasta el último elemento o hasta el primero, se puede omitir.

```
L[2:]
```

```
[5, 7, 11, 13, 17, 2]
```

```
L[:-2]
```

```
[3, 3, 5, 7, 11, 13]
```

Si queremos saber el número de elementos, utilizamos la orden ***len***

```
len(L)
```

```
8
```

Recordemos que los elementos de una lista pueden ser de distinto tipo.

```
Lista_rara=[1,"Hola",[1,2,3]];Lista_rara
```

Podemos añadir un elemento al final de una lista con ***append***, añadir una lista con ***extend*** o símplemente unir varias listas con la operación +.

```
L.append(4)
```

```
L.extend([10,11,12])
```

```
[1,3,5]+[2,4,6]+[100]
```

Creación de listas

Sage permite crear de modo sencillo una lista de elementos equiespaciados (esta construcción no es de Python).

```
# Los números naturales del 1 al 7
[1..7]
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
# Los números pares del 2 al 10
[2,4..10]
```

```
# Dividimos el intervalo [0,1] en n trozos
n=9
[(0.0),1/n..(1.0)]
```

Nótese que la construcción anteior es correcta, sin embargo, su implementación con un blucle while no es trivial:

```
k=0.0
n=9
l=[]
while k<=1:
    l=l+[k]
    k=k+1/n
l
```

Un modo "pythonico" para crear listas de números equiespaciados es ***range***. Sólo crea listas de enteros, pero a partir de ellas se pueden construir listas de números reales.

```
# Los 5 primeros índices de una lista
range(5)
```

```
# Los puntos del apartado anterior (ahora sin errores).
l=[]
n=9.0
for k in range(n+1):
    l=l+[k/n]
l
```

Operando con listas

Sage soporta muchas operaciones directamente sobre listas. Como ejemplo, vamos a sumar los elementos de una lista, vamos a agrupar dos listas en una uniendo el elemento k de la primera con el k de la segunda, y por último ***map*** que aplica una función a cada elemento de una lista.

```
sum([1..100])
```

```
zip([1,2,3,4],['a','b','c','d'] )
```

```
map( cos, [0, pi/4, pi/2, 3*pi/4, pi] )
```

Ejercicios

a) Crea una lista con los números (naturales) del 4 al 25

b) Crea una lista con los números pares del 4 al 25

Ejercicio a).

```
#Creamos la lista
LISTA=[4..25]
print LISTA
```

```
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25]
```

Ejercicio b).

```
#Creamos la lista
PARES=[4,6..25]
```

```
print PARES
```

```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
```

Listas por comprensión

Python permite definir listas por comprensión. Es una manera rápida de crear una lista a partir de otra. La sintasis es ***[*** expresión ***for*** indice ***in*** lista***]***. Devuelve una lista resultado de evaluar la expresión (que dependerá de la variable denominada índice) aplicada a cada uno de los elementos de la lista.

```
[ 2*k for k in [0..10] ]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
[ cos(x) for x in [pi/4, pi/2..2*pi]]
```

Además se puede filtrar el resultado de la salida:

```
U =  [ k for k in [1..19] if gcd(k,20) == 1]; U
```

```
[1, 3, 7, 9, 11, 13, 17, 19]
```

Y se pueden anidar bucles:

```
[ (a,b) for a in U for b in U ]
```

Por último, la expresión inicial también puede contener condicionales:

```
[ 'prime' if x.is_prime() else 'not prime' for x in U]
```

Ejercicios

a) Crea una lista con los cuadrados de los números (naturales) del 4 al 25

b) Crea una lista con los cuadrados de los números pares del 4 al 25

c) Crea una lista con los múltiplos de 3 entre 3 y 30 cuyo seno sea positivo

Ejercicio a).

```
LISTA=[ n^2 for n in [4..25] ]
print LISTA
```

```
[16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
324, 361, 400, 441, 484, 529, 576, 625]
```

Ejercicio b).

```
LISTA=[ n^2 for n in [4,6..25] ]
print LISTA
```

```
[16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576]
```

Ejercicio c).

```
LISTA=[
    n for n in [3,6..30]
       if sin(n)>=0
    ]
print LISTA
```

```
[3, 9, 15, 21, 27]
```

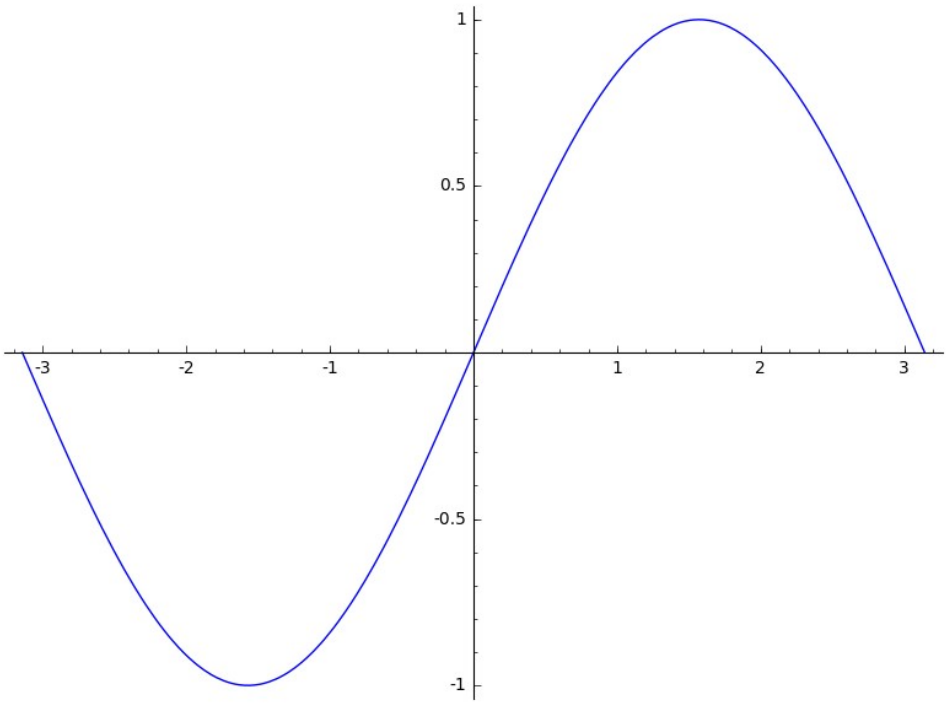
```

```

Gráficos en Sage

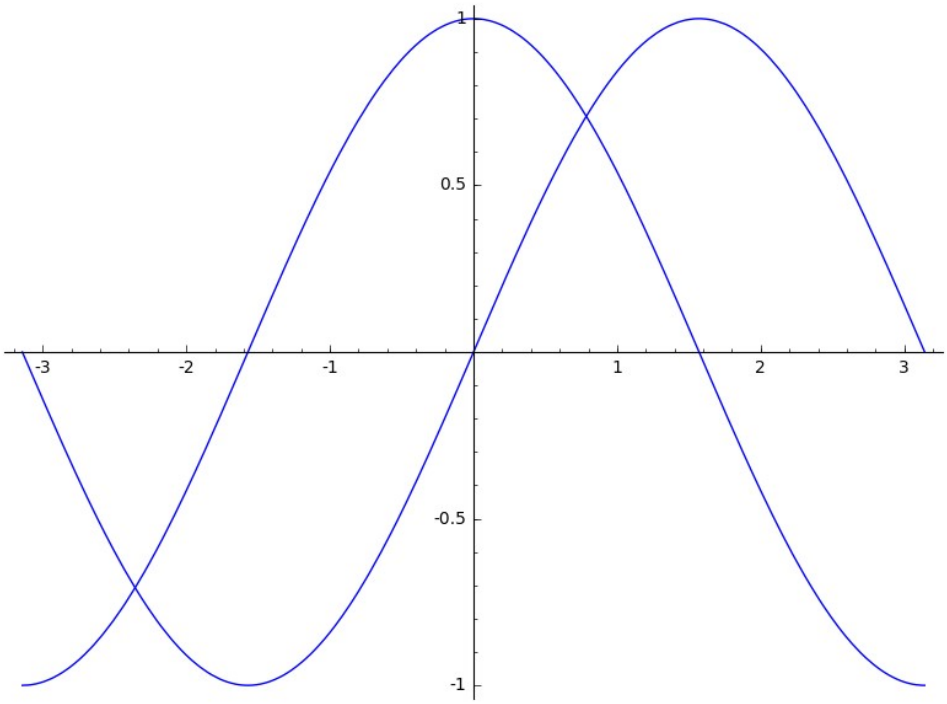
Sage nos permite representar una función matemática con el comando ***plot***. La sintaxis es *plot(f(x),(x,a,b))*, donde f(x) es la función a representar y (a,b) el intervalo donde la queremos representar. Por ejemplo:

```
plot(sin(x),(x,-pi,pi))
```



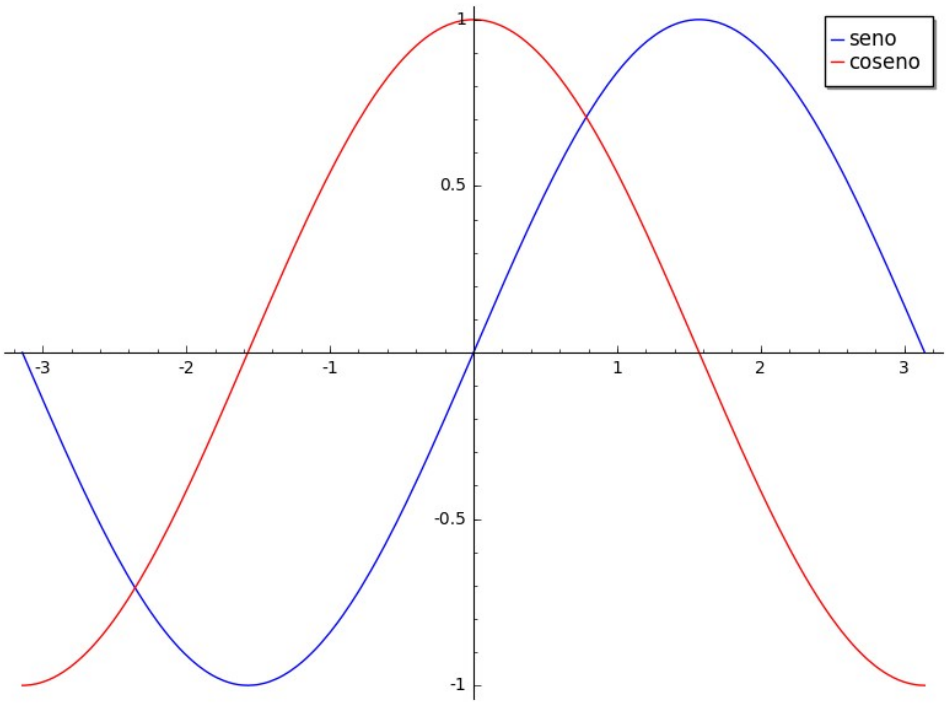
Para representar varias funciones en la misma gríafica, podemos usar el operador +, que representa dos gráficos superpuestos.

```
plot(sin(x),(x,-pi,pi))+plot(cos(x),(x,-pi,pi))
```



Las funciones anteriores no se distinguen bien. Para evitar esto, podemos cambiar el color en el que se dibuja una de ellas. Además, podemos poner una leyenda que detalle en qué color dibujamos cada función.

```
plot(sin(x),(x,-pi,pi),legend_label=u'seno')+plot(cos(x),(x,-pi,pi),color='red',legend_label=u'coseno')
```



El comando plot tiene muchas opciones. Puedes consultarlas con la ayuda:

```
help(plot)
```

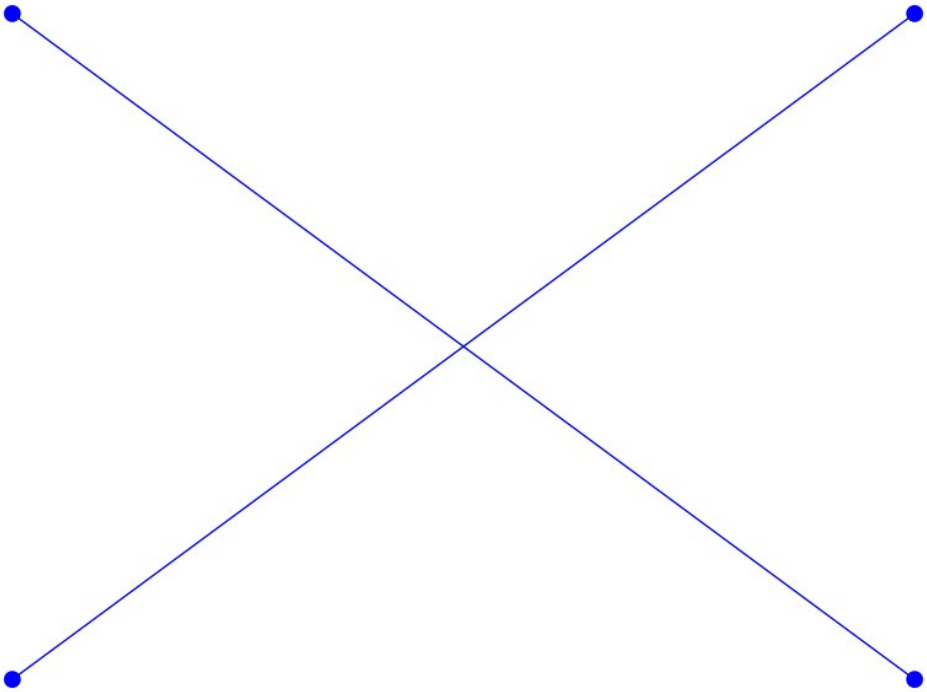
Click to open help window

En ocasiones, lo que nos interesa dibujar no son funciones sino símplemente puntos o rectas (por ejemplo, cuando tenemos una serie de datos). Para dibujar puntos, usamos ***points*** (con argumento una lista con las coordenadas) y para las rectas ***line*** (con argumento una lista con dos elementos, correspondientes a los dos puntos que limitan el segmento - cada punto es a su ver una lista con las coordenadas).

```
help(points)
```

Click to open help window

```
pA=(0,0)
pB=(1,0)
pC=(1,1)
pD=(0,1)
points(pA,size=100,axes=False)+points(pB,size=100)+points(pC,size=100)+points(pD,size=100)+line([pA,pC])+line([pB,pD])
```



El comando ***points*** también puede recibir una lista de puntos. Por otra parte, podemos utilizar listas para realizar un dibujo complejo.

```
pA=(0,0)
pB=(1,0)
pC=(1,1)
pD=(0,1)
lados=[[pA,pB],[pB,pC],[pC,pD],[pD,pA]]
points([pA,pB,pC,pD],size=100,axes=False)+sum([line(s) for s in lados])
```

Ejercicios

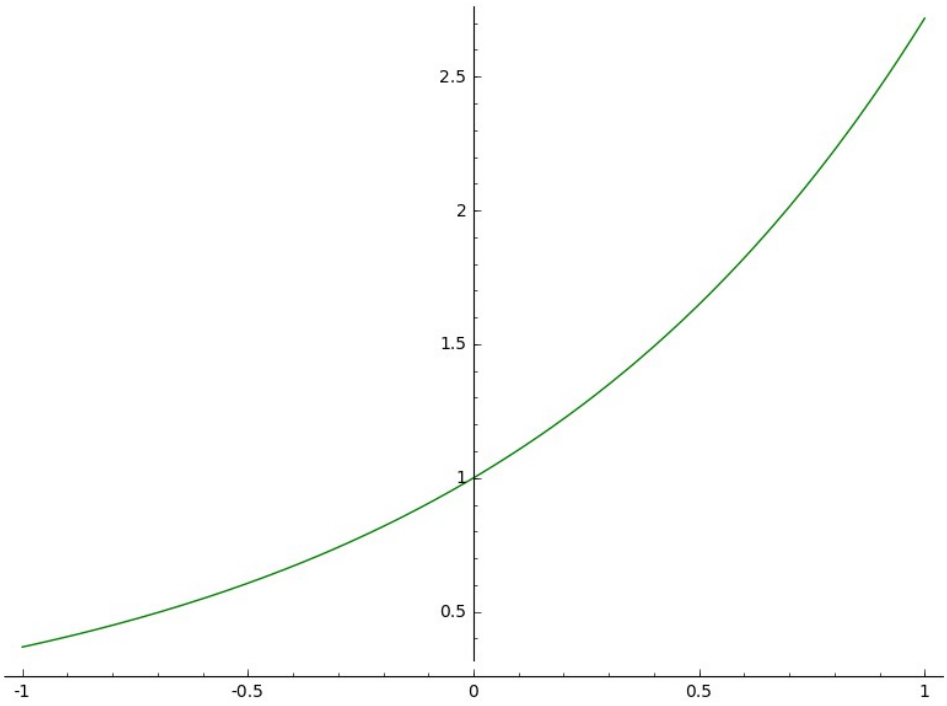
a) Dibuja la gráfica de la función e^x para x entre -1 y 1. Cambia el color del trazo.

b) Dibuja tres puntos no alineados.

c) Dibuja el triángulo definido por los tres puntos del apartado anterior.

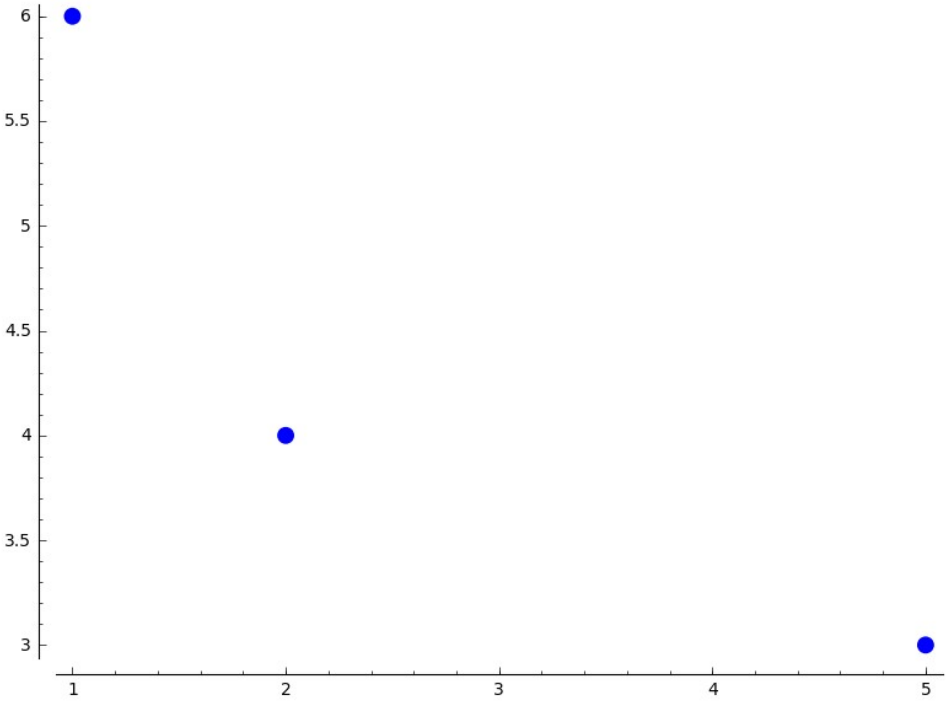
Ejercicio a).

```
f(x)=e^x
plot(f(x),(x,-1,1), color='green')
```



Ejercicio b).

```
pA=(1,6)
pB=(2,4)
pC=(5,3)
points([pA,pB,pC], size=100)
```



Ejercicio c).

```
lados=[[pA,pB],[pB,pC],[pC,pA]]
points([pA,pB,pC],size=100,axes=False)+sum([line(s) for s in lados])
```