Retour sur le cours 3

Nous avons vu comment utiliser différentes opérations arithmétiques:

- calculer un modulo
- calculer une exponentielle modulaire
- calculer le pgcd de deux entiers
- calculer l'inverse d'un entier modulo n par Euclide étendu et les coefficients de Bézout

Nous avons appliqué ces calculs au chiffrement à clé publique proposé par RSA.

Cours 4 : Graphiques

On a déju vu plusieurs utilisations des graphiques en `Python` en utilisant l'affichage provenant de diverses librairies:

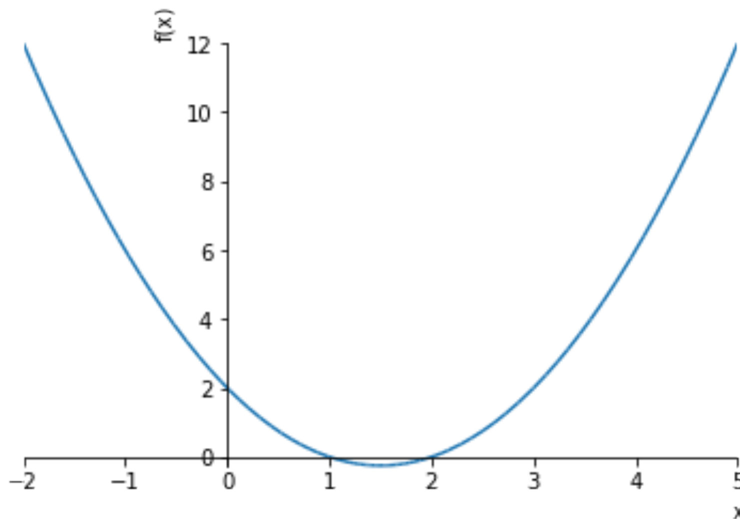
- `sympy` pour afficher une courbe
- `matplotlib` pour afficher un graphique en barres

Attention: l'utilisation des fonctions de ces deux librairies peut être assez différent, notamment pour la fonction `plot` de `sympy` et `plot` de `matplotlib.pyplot`.

Revenons sur les exemples déjà vus:

In [1]:
```
%matplotlib inline
from sympy import *
```

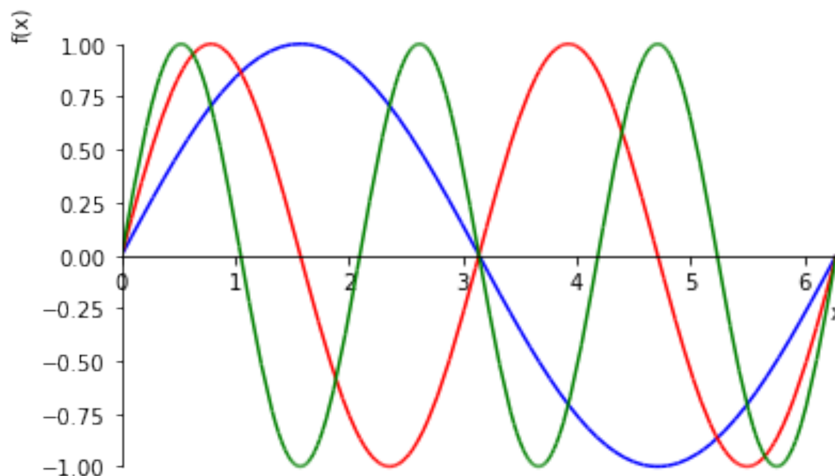
In [29]:
```
x=symbols('x')
plot(x**2-3*x+2,(x,-2,5)) #c'est le plot de sympy
```



Out[29]: <sympy.plotting.plot.Plot at 0x11198fcc0>

In [30]:
```
p1=plot(sin(x),(x,0,2*pi),show=False,line_color='b')
p2=plot(sin(2*x),(x,0,2*pi),show=False,line_color='r')
p3=plot(sin(3*x),(x,0,2*pi),show=False,line_color='g')
p1.extend(p2); p1.extend(p3) ; print(p1)
p1.show()
```

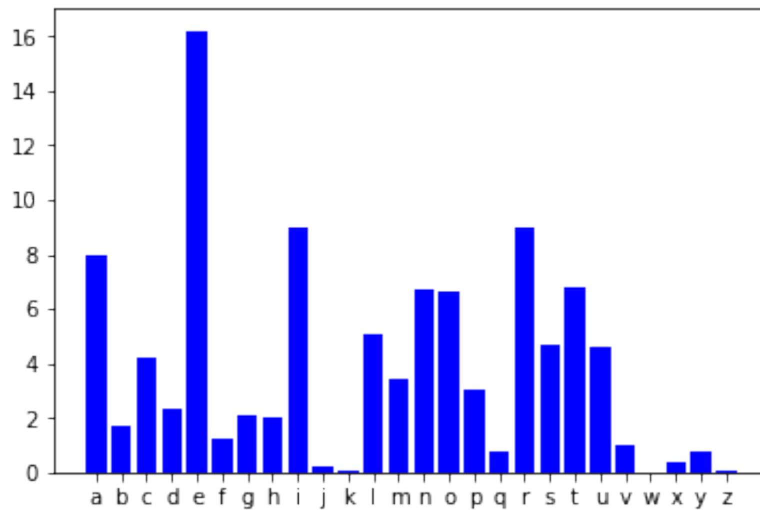
```
Plot object containing:
[0]: cartesian line: sin(x) for x over (0.0, 6.283185307179586)
[1]: cartesian line: sin(2*x) for x over (0.0, 6.283185307179586)
[2]: cartesian line: sin(3*x) for x over (0.0, 6.283185307179586)
```



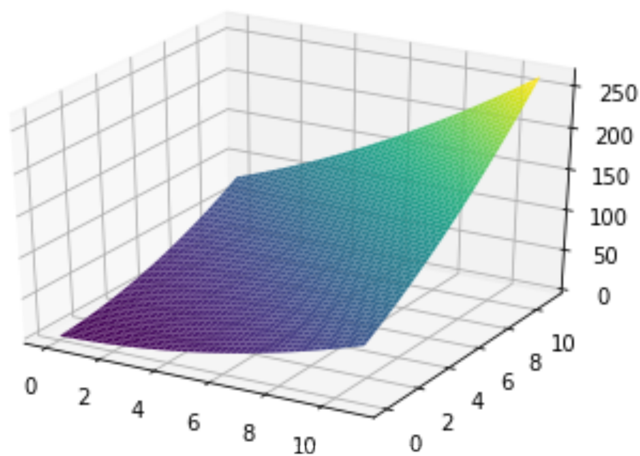
In [32]:
```
occur={'a': 8.0, 'b': 1.7, 'c': 4.2, 'd': 2.3, 'e': 16.2, 'f': 1.2,
'g': 2.1, 'h': 2.0, 'i': 9.0, 'j': 0.2, 'k': 0.1, 'l': 5.1, 'm': 3.
4, 'n': 6.7, 'o': 6.6, 'p': 3.0, 'q': 0.8, 'r': 9.0, 's': 4.7, 't':
6.8, 'u': 4.6, 'v': 1.0, 'w': 0.0, 'x': 0.4, 'y': 0.8, 'z': 0.1}
```

In [33]:
```
import matplotlib.pyplot as plt
plt.bar(list(occur.keys()),occur.values(),color='b')
```

Out[33]: <BarContainer object of 26 artists>



In [34]:
```
from sympy import *
x,y=symbols('x y')
f=Lambda([x,y],y+(x+y)*(x+y+1)/2)
f(1,3)
from sympy.plotting import plot3d
plot3d(f(x,y),(x,0,11),(y,0,11))
```



Out[34]: <sympy.plotting.plot.Plot at 0x111a9f588>

In [45]:
```
import matplotlib.pyplot as plt
```

In [ ]:
```
plt.plot(x**2)
```

nous retourne une erreur, contrairement à `plot` de `sympy`.

Nous allons tout d'abord explorer l'affichage propre à `sympy` avant de passer à celui, plus puissant de `matplotlib`.

Mais avant tout, quelques notions qui sont communes aux deux librairies

De l'interface à l'affichage

On fait appel à deux processus distincts quand on veut afficher un graphique:

- le *front-end* qui interprète les requêtes de l'utilisateur
- le *back-end* qui les affiche en résultats

Les *front-ends*

C'est l'interface utilisateur de la librairie `matplotlib`. L'une `pylab` offre à l'utilisateur une interface interactive restreinte, accessible à partir de `ipython` par la commande `ipython --pylab` qui rappelle la syntaxe d'un logiciel scientifique appelé `mathlab`. L'interface utilisateur qui nous intéressera est de l'utiliser soit au travers de `sympy`, soit au travers de `pyplot` en important la librairie par:

In [1]:
```
import matplotlib.pyplot as plt
```

La complétion automatique `plt.TAB` nous montre sa richesse: plus de 200 possibilités!

`matplotlib.pyplot` est une collection de commandes d'affichage qu'on va utiliser

Les *back-ends*

Il faut spécifier où le résultat va s'afficher. A l'écran, sur du papier, dans une application externe? Tout ceci dépend du matériel et c'est au *back-end* de le gérer. `Matplotlib` et `sympy` gèrent un grand nombre de *back-ends*. `Python` lors de son installation a reconnu votre matériel et a choisi le meilleur *back-end*. On peut voir lequel a été choisi par:

In [2]:
```
plt.get_backend()
```

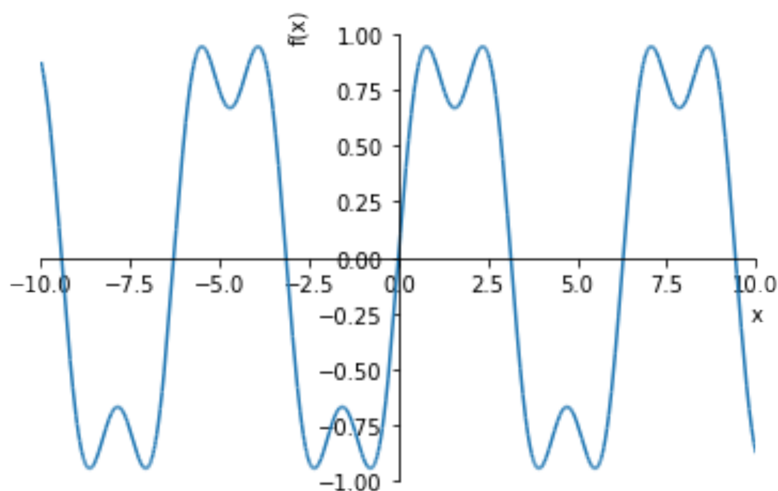
Out[2]:
```
'module://ipykernel.pylab.backend_inline'
```

Pour nous, deux options sont intéressantes qu'on fournit à l'interface de `jupyter`:

- `%matplotlib inline` qui affiche les figures directement dans l'interface
- `%matplotlib notebook` qui affiche une fenêtre unique pour toutes les figures avec des boutons de contrôle (mais qui ne fonctionne pas avec toutes les versions de `matplotlib`)

Créer un graphique simple avec `plot` de `sympy`

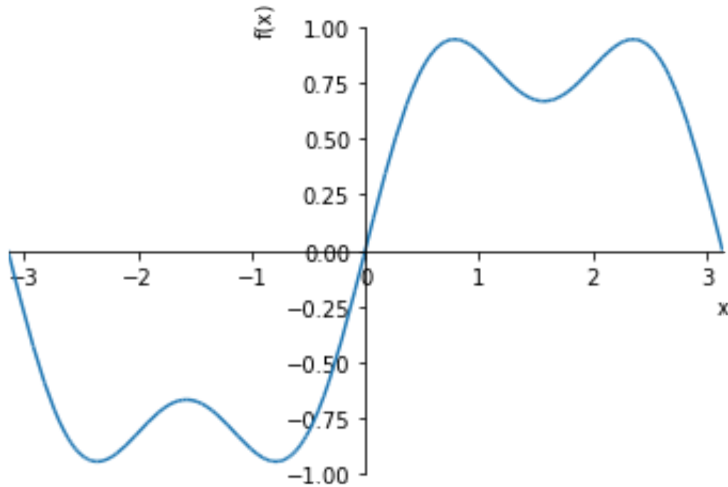
In [51]:
```
x=symbols('x')
y=sin(x)+sin(3*x)/3
plot(y)
```



Out[51]: `<sympy.plotting.plot.Plot at 0x1132e2198>`

il peut être utile de spécifier des intervalles sur les valeurs en x ou en y

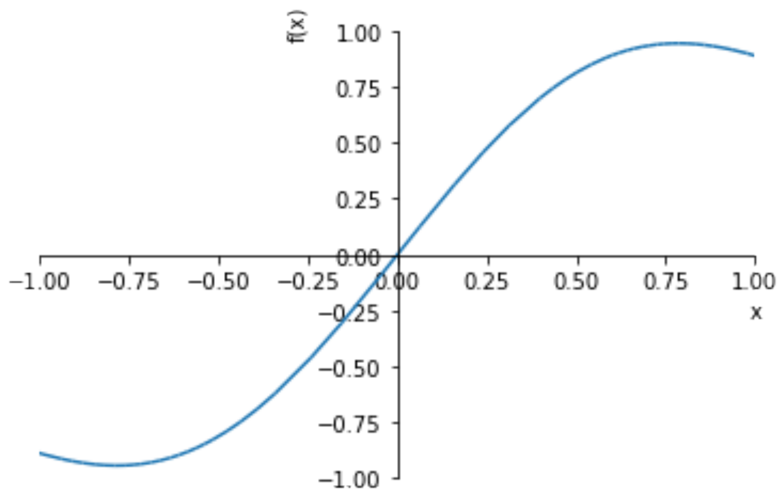
In [52]:
```
plot(y,(x,-pi,pi))
```



Out[52]: <sympy.plotting.plot.Plot at 0x1133fa3c8>

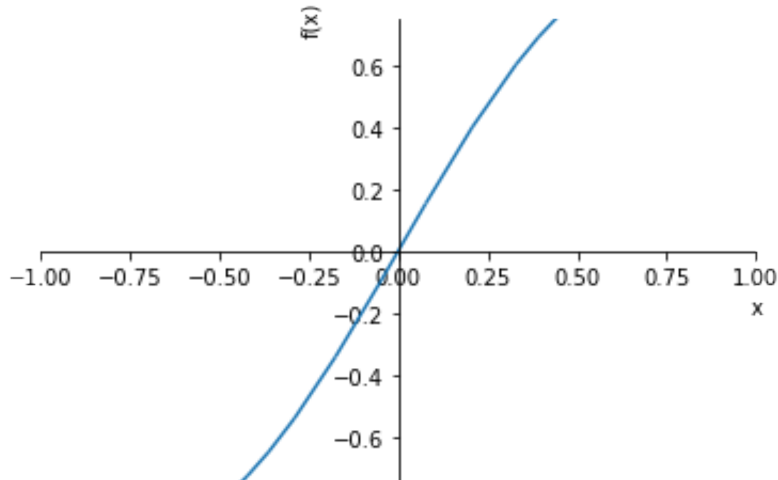
On peut spécifier une fenêtre d'affichage par `xlim`et `ylim`

In [53]:
```
plot(y,(x,-pi,pi),xlim=(-1,1))
```



Out[53]: <sympy.plotting.plot.Plot at 0x1134044e0>

In [55]:
```
plot(y,(x,-pi,pi),xlim=(-1,1),ylim=(-0.75,0.75))
```

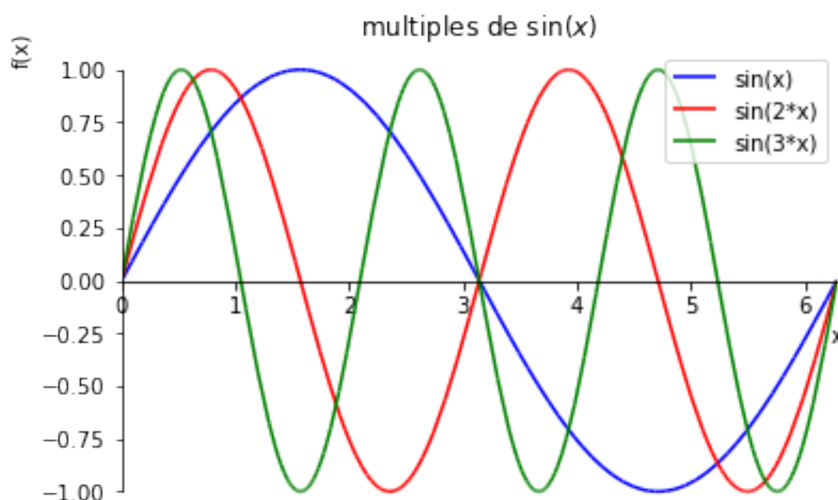


Out[55]: <sympy.plotting.plot.Plot at 0x11380ff28>

Tracer plusieurs courbes avec `plot` de `sympy`

Comme dans l'exemple de la figure de $\sin(x), \sin(2x), \sin(3x)$

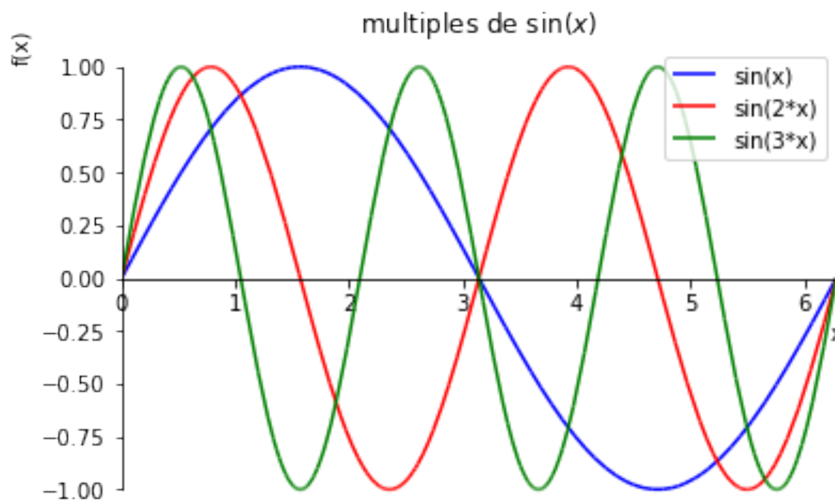
In [62]:
```
p1=plot(sin(x),(x,0,2*pi),show=False,line_color='b',title='multiple
s de $\sin(x)$',legend=('$\sin(x)$'))
p2=plot(sin(2*x),(x,0,2*pi),show=False,line_color='r')
p3=plot(sin(3*x),(x,0,2*pi),show=False,line_color='g')
p1.extend(p2); p1.extend(p3)
p1.show()
```



Ici, `p1` va contenir le tracé des différentes courbes. Pour retarder l'affichage, on lui fournit l'option `show=False`. On ajoute à `p1` le tracé de `p2` et de `p3`. La méthode `show()` permet alors d'afficher. Notez aussi le changement des couleurs pour les courbes, l'option qui permet de spécifier un titre et une légende.

On peut également sauver cette figure pour l'utiliser dans une autre application.

In [68]:
```
p1.save('./sinus.png')
```



Utiliser `matplotlib`

Si `plot` de `sympy` utilise de manière sous-jacente `matplotlib`, leurs interfaces sont différentes ; comme on l'a vu plus haut, `plot` de `sympy` et `plot` de `matplotlib` fonctionnent de façon différente. Pour certains graphiques (les graphiques en barres ou l'affichage de séries de données), on doit utiliser `matplotlib`.

`matplotlib` pour afficher des fonctions mathématiques

Pour afficher les fonctions mathématiques, on a vu que `plt.plot(x**2)` nous retourne une erreur. En effet `matplotlib` a besoin d'évaluer la fonction $x \mapsto x^2$, donc d'effectuer les calculs. Pour cela, on fait appel à la bibliothèque de calcul numérique `numpy`. `numpy` permet en particulier de gérer des tableaux de nombres (`arrays` en anglais).

In [3]:
```
import numpy as np
```

On peut convertir une liste `python`en tableau

In [5]:
```
L=[i for i in range(5)]
a=np.array(L)
a
```

Out[5]:
```
array([0, 1, 2, 3, 4])
```

Pour créer un tableau de zéros de longueur `n`:

In [8]:
```
a=np.zeros(5)
a
```

Out[8]:
```
array([0., 0., 0., 0., 0.])
```

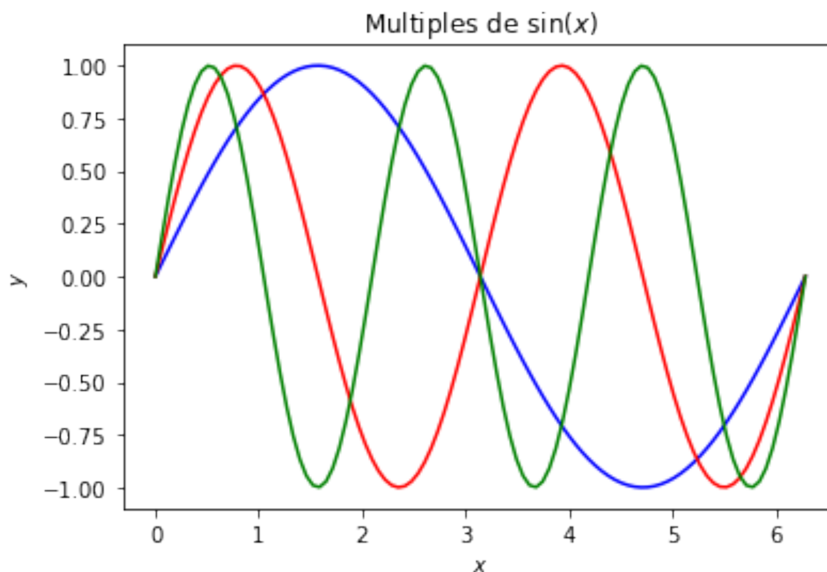
Pour tracer le graphe d'une fonction, on a besoin de donner au départ un intervalle de travail par l'instruction `np.linspace()` qui construit un tableau 1D sur cet intervalle avec un nombre donné d'éléments. Dans l'exemple, on génère 100 valeurs de l'intervalle $[0, 2\pi]$.

```
In [72]: x=np.linspace(0,2*np.pi,100)
         x
```

```
Out[72]: array([0.        , 0.06346652, 0.12693304, 0.19039955, 0.25386607,
                0.31733259, 0.38079911, 0.44426563, 0.50773215, 0.57119866,
                0.63466518, 0.6981317 , 0.76159822, 0.82506474, 0.88853126,
                0.95199777, 1.01546429, 1.07893081, 1.14239733, 1.20586385,
                1.26933037, 1.33279688, 1.3962634 , 1.45972992, 1.52319644,
                1.58666296, 1.65012947, 1.71359599, 1.77706251, 1.84052903,
                1.90399555, 1.96746207, 2.03092858, 2.0943951 , 2.15786162,
                2.22132814, 2.28479466, 2.34826118, 2.41172769, 2.47519421,
                2.53866073, 2.60212725, 2.66559377, 2.72906028, 2.7925268 ,
                2.85599332, 2.91945984, 2.98292636, 3.04639288, 3.10985939,
                3.17332591, 3.23679243, 3.30025895, 3.36372547, 3.42719199,
                3.4906585 , 3.55412502, 3.61759154, 3.68105806, 3.74452458,
                3.8079911 , 3.87145761, 3.93492413, 3.99839065, 4.06185717,
                4.12532369, 4.1887902 , 4.25225672, 4.31572324, 4.37918976,
                4.44265628, 4.5061228 , 4.56958931, 4.63305583, 4.69652235,
                4.75998887, 4.82345539, 4.88692191, 4.95038842, 5.01385494,
                5.07732146, 5.14078798, 5.2042545 , 5.26772102, 5.33118753,
                5.39465405, 5.45812057, 5.52158709, 5.58505361, 5.64852012,
                5.71198664, 5.77545316, 5.83891968, 5.9023862 , 5.96585272,
                6.02931923, 6.09278575, 6.15625227, 6.21971879, 6.28318531]
         )
```

L'intérêt de `numpy` est de pouvoir travailler directement sur le tableau, par exemple pour calculer l'image des valeurs du tableau par une fonction f sans avoir besoin d'utiliser une boucle.

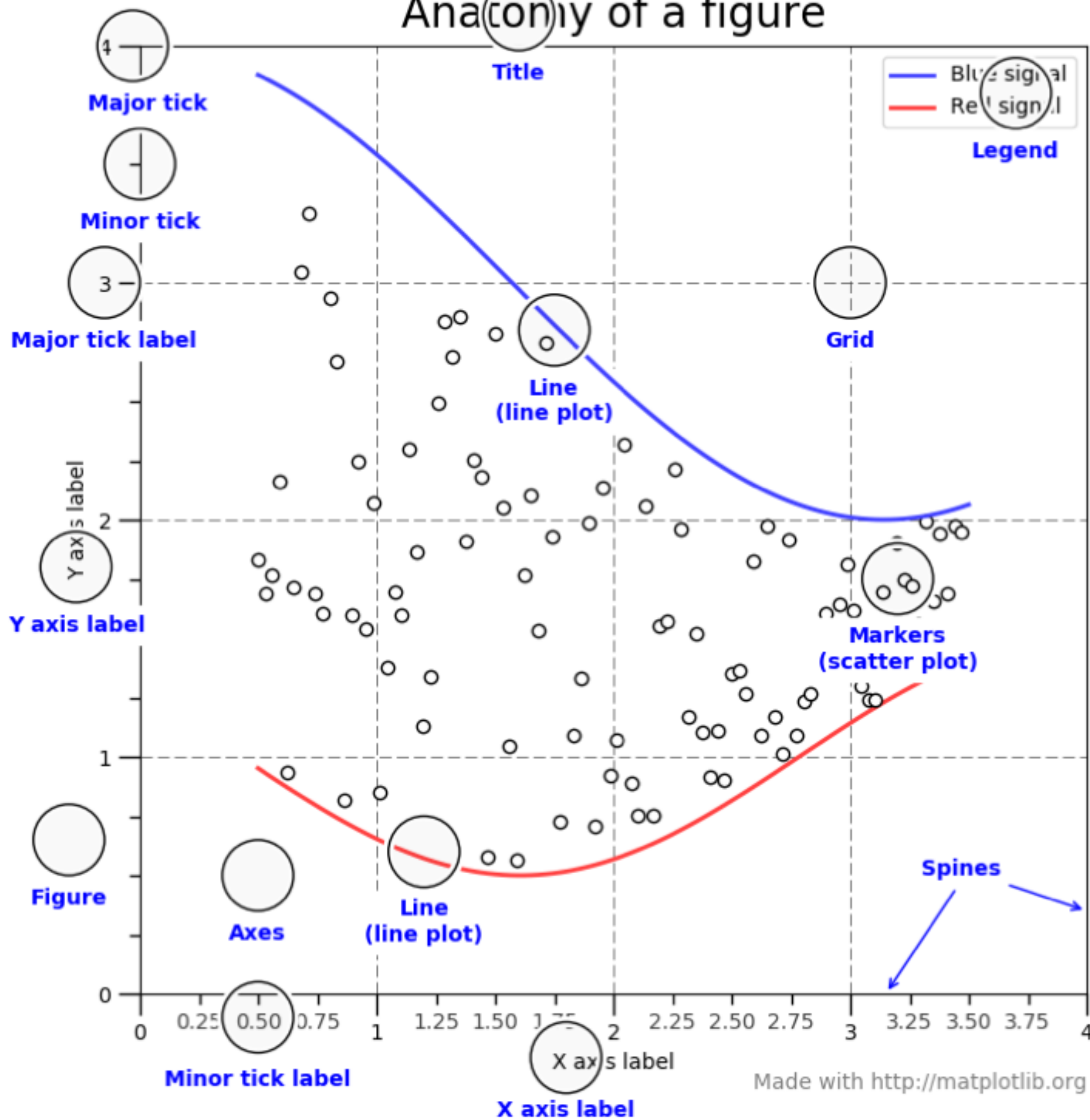
In [93]:
```
y=np.sin(x)
y2=np.sin(2*x)
y3=np.sin(3*x)
plt.plot(x,y,c='b')
plt.plot(x,y2,c='r')
plt.plot(x,y3,c='g')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.title('Multiples de $\sin(x)$')
plt.savefig('sinus.pdf')
```



`matplotlib` peut faire bien plus que d'afficher une (ou plusieurs courbes mathématiques), ce qu'on peut faire (plus facilement) au moyen de la commande `plot` de `sympy`.

Concepts généraux

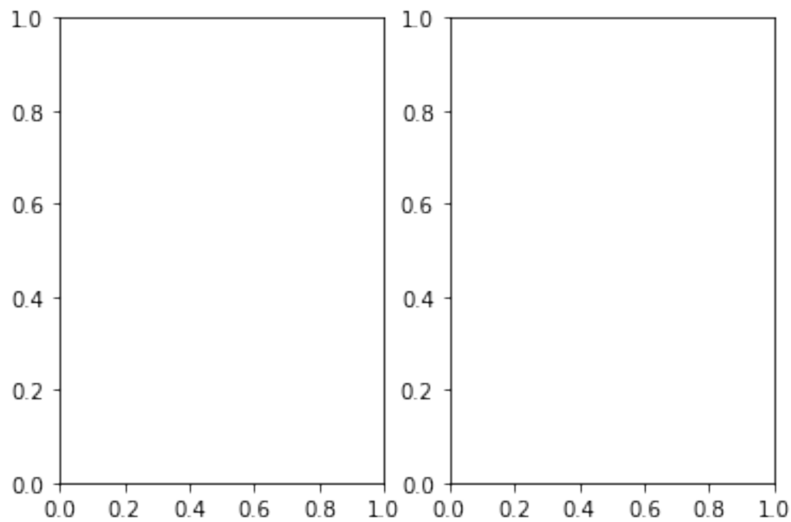
l'API de `matplotlib` est orientée objet et conçue hiérarchiquement. Au sommet de cette hiérarchie on trouve les fonctions pour afficher des éléments (lignes, images, texte,...) aux axes courants de la figure courante. Voici les différents éléments qui composent une figure:



Anatomy of a figure
Title
Major tick
Minor tick
Major tick label
Grid
Legend
Line
(line plot)
Y axis label
Markers
(scatter plot)
Figure
Axes
Spines
Minor tick label
X axis label
Made with http://matplotlib.org

In [21]:
```
fig=plt.figure()
fig.suptitle('pas d\'axes sur la figure')
fig,axes=plt.subplots(1,2) #
```

<Figure size 432x288 with 0 Axes>

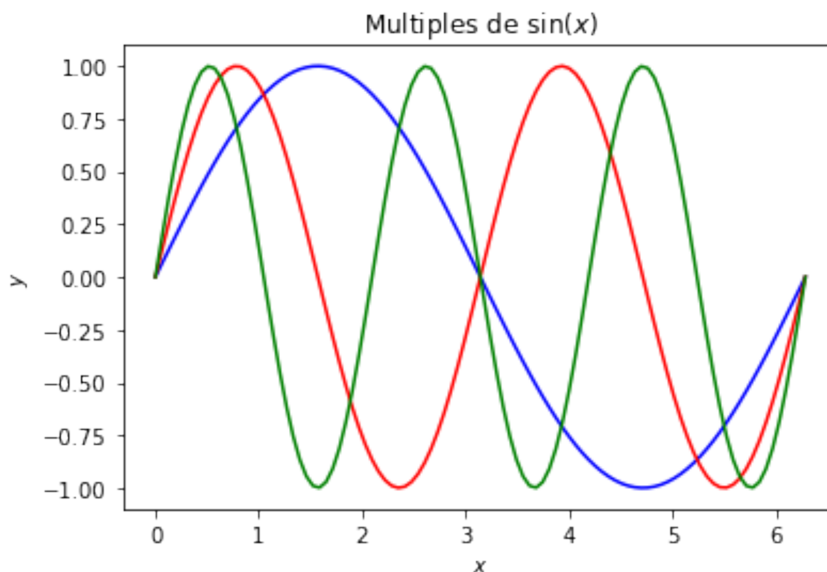


La figure conserve la trace des fils *Axes*, une connaissance superficielle des *Artistes* (titres, légendes,...) et du *canvas*. Une figure comporte au moins une instance de *Axes*. Chaque instance contient un axe en x et un en y.

Pour reconstruire la figure obtenue interactivement avec `matplotlib` mais en utilisant l'API objet:

In [104]:
```
x=np.linspace(0,2*np.pi,100)
y=np.sin(x)
y2=np.sin(2*x)
y3=np.sin(3*x)
fig=plt.figure()
ax=fig.add_subplot(111)  # ajoute une isntance de Axes à la figure fig; il en faut au moins 1
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title('Multiples de $\sin(x)$')
ax.plot(x,y,c='b')
ax.plot(x,y2,c='r')
ax.plot(x,y3,c='g')

#fig.savefig('sinus.pdf')
```

Out[104]:
```
[<matplotlib.lines.Line2D at 0x115db18d0>]
```



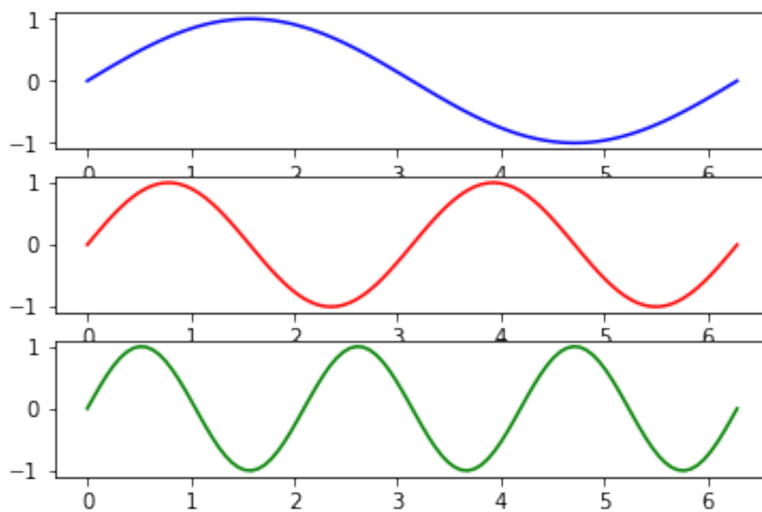
La méthode `add_subplot()` permet de gérer l'affichage de plusieurs `Axes` dans la même figure. La méthode `add_subplot()` prend une suite de 3 nombres en paramètres:

- la hauteur *height*
- la largeur *width*
- le numéro d'affichage, *plot number*

Ainsi `111` signifie 1 de hauteur, 1 de largeur et 1 de numéro d'affichage. En mettant des valeurs plus élevées, on peut obtenir plus d'axes sur la figure (et donc plusieurs courbes).

In [107]:
```
x=np.linspace(0,2*np.pi,100)
y=np.sin(x)
y2=np.sin(2*x)
y3=np.sin(3*x)
fig=plt.figure()
ax1=fig.add_subplot(311)
ax2=fig.add_subplot(312)
ax3=fig.add_subplot(313)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title('Multiples de $\sin(x)$')
ax1.plot(x,y,c='b')
ax2.plot(x,y2,c='r')
ax3.plot(x,y3,c='g')
```

Out[107]:
```
[<matplotlib.lines.Line2D at 0x1163ddeb8>]
```



Gérer les couleurs, les styles de lignes, les marqueurs de points

Choix de couleurs

Caractère	Couleur
b	bleu (par défaut)
g	vert
r	rouge
c	cyan
m	magenta
y	jaune
k	noir
w	blanc

Choix des styles de lignes

Caractère(s)	Description
-	pleine (par défaut)
--	interrompue
-.	mixte
:	pointillé

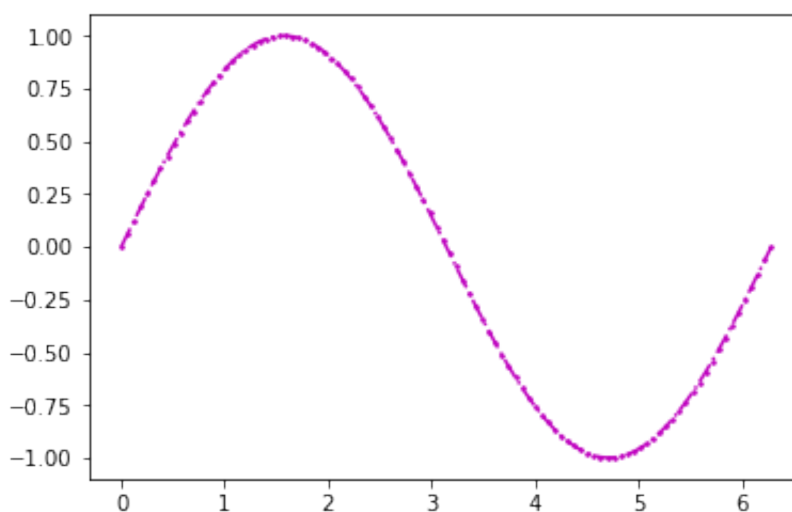
Choix des marqueurs de points

Caractère	Couleur
.	point (par défaut)
o	cercle
*	étoile
+	plus
x	x
v	triangle bas
^	triangle haut
<	triangle gauche
>	triangle droit
n	carré
p	pentagonal
h	hexagonal

On spécifie ces différents paramètres en option dans la fonction d'affichage.

In [92]:
```
plt.plot(x,y,'m-.',marker='o',markersize=1.5)
```

Out[92]:
```
[<matplotlib.lines.Line2D at 0x11517be48>]
```



On peut faire **beaucoup** de choses avec `matplotlib`. Nous l'utiliserons surtout pour sa capacité à afficher différents types de graphiques (plutôt en utilisant le mode interactif du début qu'avec l'API objet).

Les différents types de tracés avec `matplotlib`

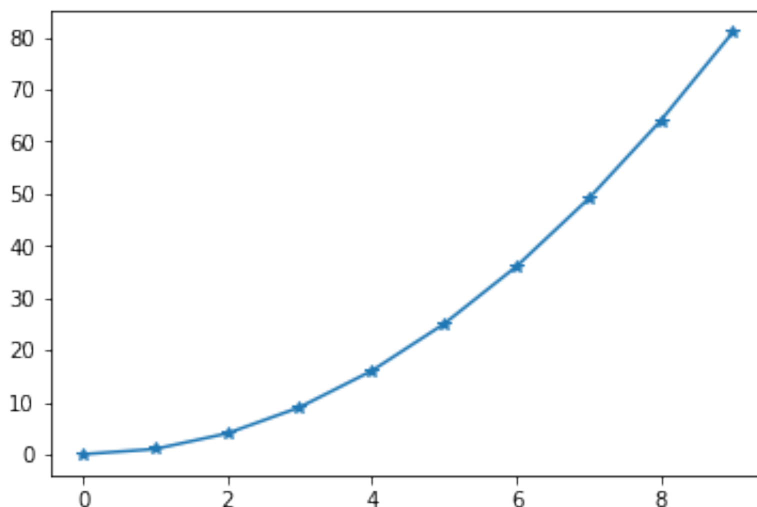
On travaillera avec l'exemple d'une suite de données en x correspondant aux entiers de 0 à 9 et en y au carré des valeurs en x:

In [108]:
```
x=[i for i in range(10)]
y=[i**2 for i in range(10)]
```

L'affichage par défaut au moyen de `plt.plot()` nous affiche les points reliés par des segments de droites. En spécifiant le type de marqueur, on met en relief les points affichés.

In [110]:
```
plt.plot(x,y,marker='*')
```

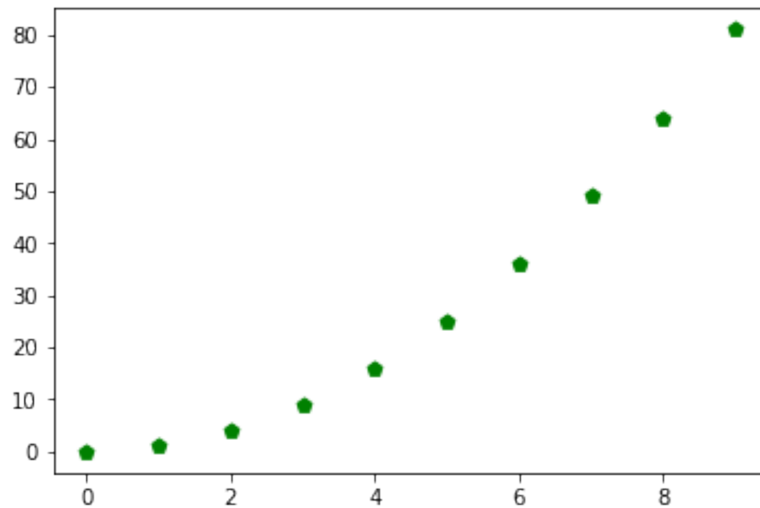
Out[110]:
```
[<matplotlib.lines.Line2D at 0x1165a4e48>]
```



Si on souhaite avoir seulement le nuage de points, on utilise la fonction `plt.scatter()`.

In [116]:
```
plt.scatter(x,y,marker='p',s=50,color='g')
```

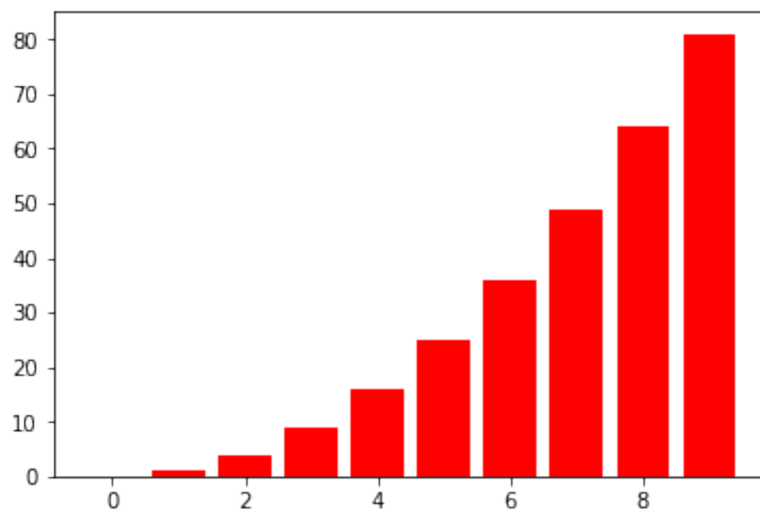
Out[116]: <matplotlib.collections.PathCollection at 0x11113b588>



Pour un graphique en barres, la fonction `plt.bar()`

In [117]:
```
plt.bar(x,y,color='r')
```

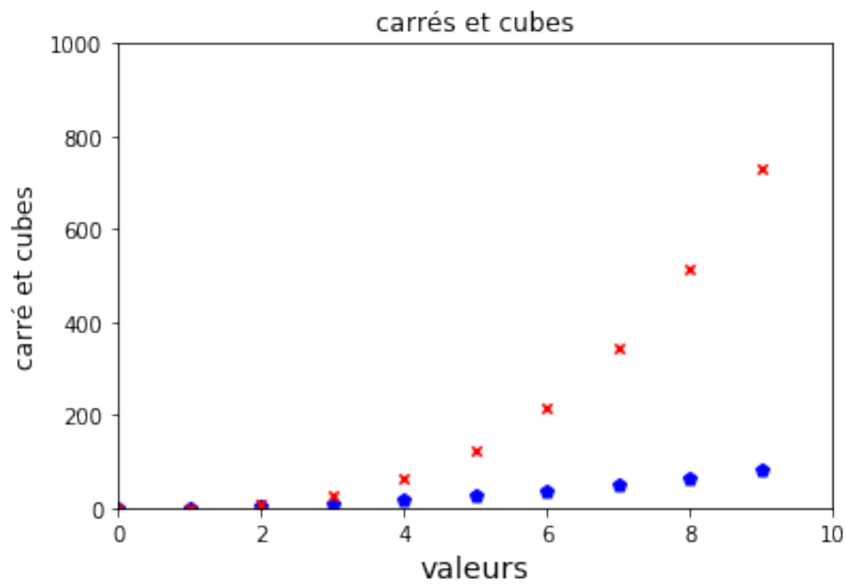
Out[117]: <BarContainer object of 10 artists>



Ajoutons comme nouvelle série de données les cubes des 10 premiers entiers pour l'afficher:

In [118]:
```
z=[i**3 for i in range(10)]
```

In [124]:
```
plt.scatter(x,y,c='blue',marker='p',s=40)
plt.scatter(x,z,c='red',marker='x',s=20)
plt.title('carrés et cubes')
plt.xlabel('valeurs',fontsize=14)
plt.ylabel('carré et cubes',fontsize=12)
plt.axis([0,10,0,1000])
plt.show()
```



Un exemple plus consistant

On considère le relevé des températures moyennes (normales saisonnière) sur Nice donné par Météo France (min et max):

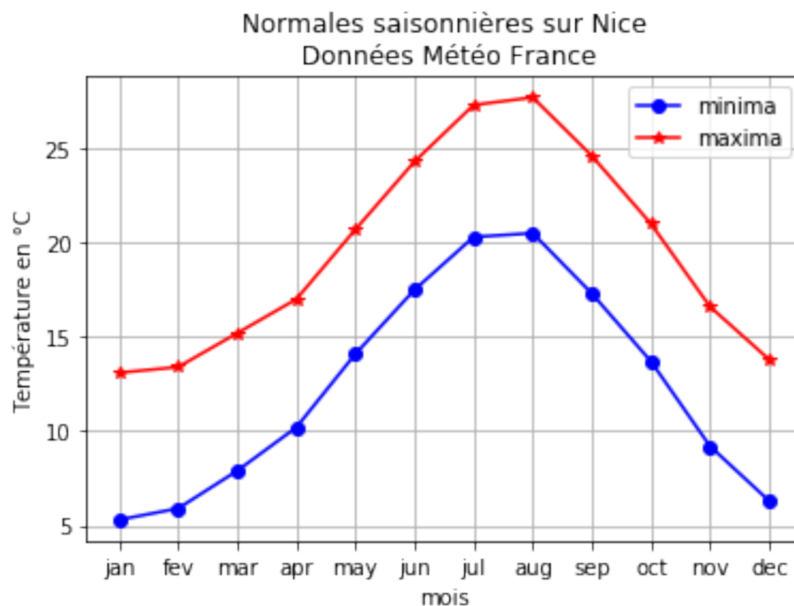
Mois	Min	Max
Jan	5.3	13.1
Fev	5.9	13.4
Mar	7.9	15.2
Avr	10.2	17
Mai	14.1	20.7
Jun	17.5	24.3
Jul	20.3	27.3
Aug	20.5	27.7
Sep	17.3	24.6
Oct	13.7	21
Nov	9.2	16.6
Dec	6.3	13.8

qu'on modélise par un dictionnaire de listes clé (le mois), valeurs (min et max)

In [125]:
```
releveNice={'jan':[5.3,13.1],
            'fev':[5.9,13.4],
            'mar':[7.9,15.2],
            'apr':[10.2,17],
            'may':[14.1,20.7],
            'jun':[17.5,24.3],
            'jul':[20.3,27.3],
            'aug':[20.5,27.7],
            'sep':[17.3,24.6],
            'oct':[13.7,21],
            'nov':[9.2,16.6],
            'dec':[6.3,13.8]}
```

In [126]:
```
mois=[mois for mois in releveNice.keys()]
mini=[min[0] for min in releveNice.values()]
maxi=[max[1] for max in releveNice.values()]
```

In [132]:
```
from pylab import legend, savefig
plt.plot(mois,mini,c='b',marker='o')
plt.plot(mois,maxi,c='r',marker='*')
plt.grid(True)
legend(['minima','maxima'])
plt.title('Normales saisonnières sur Nice\n Données Météo France')
plt.xlabel('mois')
plt.ylabel('Température en °C')
plt.savefig('meteo.png')
plt.show()
```



En résumé

`plot` de `sympy` permet d'afficher le graphe d'une fonction **symbolique**.

`plot` de `matplotlib.pyplot` permet d'afficher les points **calculés numériquement** par `numpy`.

Construire un polynôme d'interpolation

Lorsqu'on a une série de points, il peut être intéressant d'en trouver une interpolation polynomiale. Etant donné un ensemble de points, obtenus expérimentalement, on cherche un polynôme qui passe par tous ces points et qui vérifie éventuellement que le degré du polynome d'interpolation est minimal. `Python`, via sa bibliothèque `SciPy` permet de réaliser cette opération au moyen du module `scipy.interpolate`. Deux techniques sont possibles:

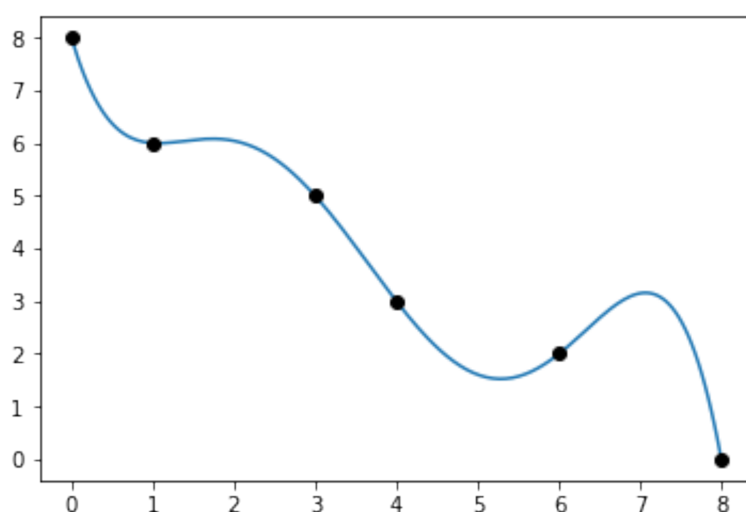
- par le polynôme d'interpolation de Lagrange (sur un échantillon d'au plus 20 points)
- par une spline d'interpolation

Par Lagrange

In [10]:
```
from scipy.interpolate import interp1d, lagrange
import matplotlib.pyplot as plt
import numpy as np
from sympy import *
data = [(0,8),(1,6),(3,5),(4,3),(6,2),(8,0)]
```

In [7]:
```
x_data = [d[0] for d in data]
y_data = [d[1] for d in data]
x_min = x_data[0]
x_max = x_data[-1]
p = lagrange(x_data,y_data)
xs = np.linspace(x_min,x_max,200)
ys = p(xs)
plt.plot(xs,ys)
plt.plot(x_data,y_data,'ok')
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x106d92c18>]
```



On peut visualiser le polynôme de Lagrange ainsi obtenu:

In [11]:
```
init_printing()
print(p)
```

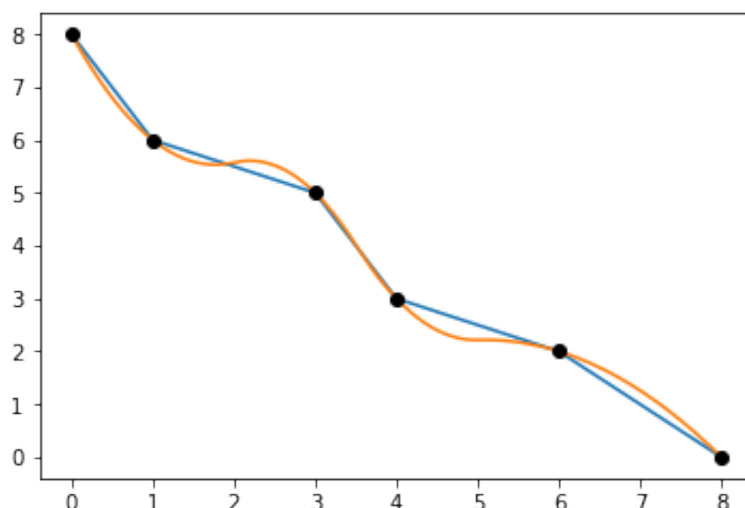
```
             5          4         3         2
-0.01518 x + 0.2875 x - 1.867 x + 4.837 x - 5.243 x + 8
```

Par une spline

Une spline est une fonction définie par morceaux au moyen de polynômes.

In [12]:
```
f1 = interp1d(x_data,y_data, kind=1)
ys1 = f1(xs)
plt.plot(xs,ys1)
f2 = interp1d(x_data,y_data, kind=2)
ys2 = f2(xs)
plt.plot(xs,ys2)
plt.plot(x_data,y_data,'ok')
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x109268358>]
```



In [13]:
```
pprint(f2)
```

```
<scipy.interpolate.interpolate.interp1d object at 0x1092694f8>
```

on a fourni une option `kind` à valeur entière pour une interpolation parmi ('linear', 'nearest', 'zero', 'slinear', 'quadratic, 'cubic' où 'slinear', 'quadratic' et 'cubic' font référence à une spline d'interpolation of 1e, 2e ou 3e ordre) ou comme une valeur entière qui spécifie l'ordre de la spline d'interpolation. La valeur par défaut est 'linear'.

Le problème est qu'on a du mal à trouver les polynôme d'interpolation fournis par `scipy`...

Exemple simple

On construit une suite quadratique de points légèrement perturbée par $0 < \epsilon < 1$ et on cherche son polynôme d'interpolation

In [14]:
```
x=[i for i in range(15)]
```

In [15]:
```
from random import random
random()
```

Out[15]: 0.7156952168484142

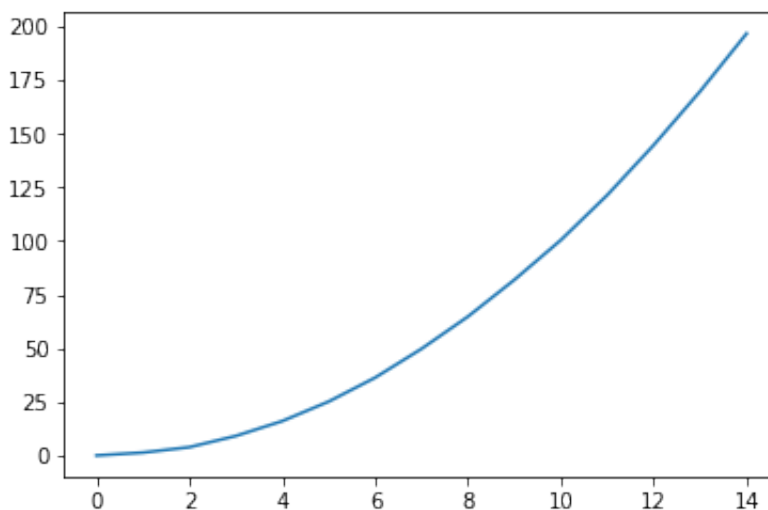
In [16]:
```
y=[i**2+(random()) for i in range(15)]
```

In [17]:
```
y
```

Out[17]: [0.16839666187724855, 1.5151878910238046, 4.030735011862667, 9.25
16.17672169559748, 25.26621184424832, 36.39712033118255, 49.84
64.86237466280743, 81.97539372036071, 100.47555411701192, 121.4
144.72016403478082, 169.76035469055077, 196.6631140090

In [18]:
```
plt.plot(x,y)
```

Out[18]: [<matplotlib.lines.Line2D at 0x109732f28>]



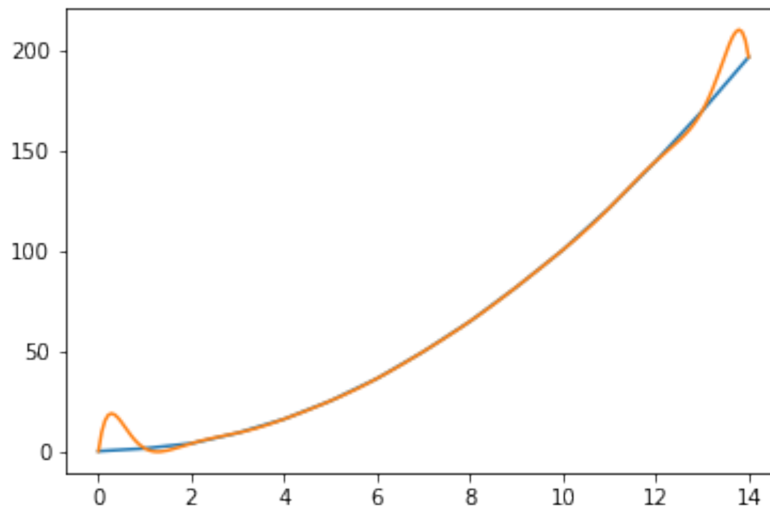
In [19]:
```
p=lagrange(x,y)
```

In [20]:
```
print(p)
```

```
              14              13              12             11
10
-1.317e-08 x  + 1.284e-06 x  - 5.636e-05 x  + 0.00147 x  - 0.02537
x
             9          8          7         6          5          4
3
 + 0.3051 x - 2.623 x + 16.26 x - 72.4 x + 227.4 x - 486.6 x + 666
.1 x
            2
 - 515.8 x + 168.7 x + 0.1684
```

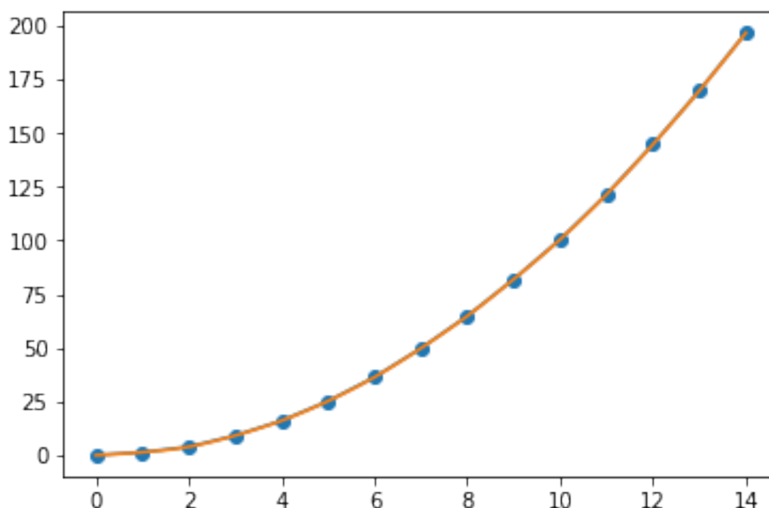
In [21]:
```
xs=np.linspace(0,14,200)
ys=p(xs)
plt.plot(x,y)
plt.plot(xs,ys)
```

Out[21]: [<matplotlib.lines.Line2D at 0x10976efd0>]



In [22]:
```
f2 = interp1d(x,y,kind=2)
xs=np.linspace(0,14,50)
ys=f2(xs)
plt.plot(x,y,marker='o')
plt.plot(xs,ys)
```

Out[22]:
```
[<matplotlib.lines.Line2D at 0x10973b7f0>]
```



bon ces méthodes ne sont pas géniales... Essayons avec `numpy.polyfit` (https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.polyfit.html)

In [23]:
```
g=np.polyfit(x,y,2)
```

In [24]:
```
g
```

Out[24]:
```
array([0.99702245, 0.08430891, 0.11474074])
```

In [25]:
```
xs=np.linspace(0,14,50)
ys=g(xs)
plt.plot(x,y,marker='o')
plt.plot(xs,ys)
```

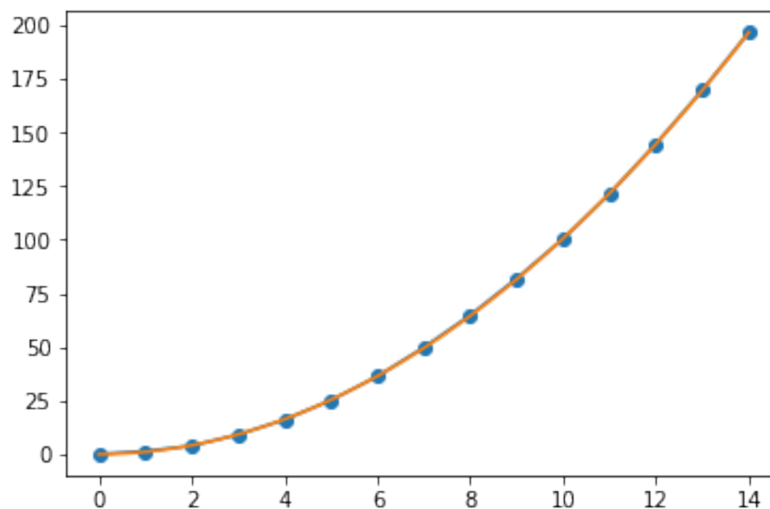
```
------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-25-85a940ae832f> in <module>()
      1 xs=np.linspace(0,14,50)
----> 2 ys=g(xs)
      3 plt.plot(x,y,marker='o')
      4 plt.plot(xs,ys)

TypeError: 'numpy.ndarray' object is not callable
```

il faut utiliser `poly1d` (https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.poly1d.html#numpy.poly1d) pour utiliser ce polynôme

In [26]:
```
p=np.poly1d(g)
```

In [27]:
```
xs=np.linspace(0,14,50)
ys=p(xs)
plt.plot(x,y,marker='o')
plt.plot(xs,ys)
```

Out[27]: `[<matplotlib.lines.Line2D at 0x1098d79e8>]`



In [28]:
```
print(p)
```

```
       2
0.997 x + 0.08431 x + 0.1147
```