Exercice 1. On utilise la fantastique formule des différences divisées de Newton : le polynome d'interpolation passant par les points $(x_0, y_0), ..., (x_n, y_n)$ de degré n est :

$$P(X) = \sum_{i=0}^{n} a_i \prod_{j=0}^{i-1} (X - x_j)$$

où $a_i = [y_0, y_1 ..., y_n]$, où on définit la différence divisée $[y_i, y_{i+1}, ..., y_j]$ par

$$[y_i] = y_i$$
$$[y_i, y_{i+1}, ..., y_j] = \frac{[y_{i+1}, ..., y_j] - [y_i, ..., y_{j-1}]}{x_j - x_i} .$$

Notons que pour calculer $[y_0, ..., y_n]$ on est obligé de calculer toutes les différences divisées $[y_i, y_{i+1}, ..., y_{i+j}]$, pas seulement celles commençant par y_0. On le fait efficacement, en commençant par calculer toutes les différences $[y_i, y_{i+1}]$, puis les $[y_i, y_{i+1}, y_{i+2}]$ et ainsi de suite.
Différences à deux termes :

i	0	1	2	3
$[y_i, y_{i+1}]$	-1	$-1/2$	3	-1

Différences à trois termes :

i	0	1	2
$[y_i, y_{i+1}, y_{i+2}$	1/6	7/6	-2

Différences à quatres termes :

i	0	1
$[y_i, ..., y_{i+3}]$	1/4	-19/24

Et enfin, $[y_0, ..., y_4] = -5/24$.
Pour calculer la valeur en 3, on calcule successivement les produits des $(3 - x_i)$: on a

$$3 - x_0 = 2 \quad (3 - x_0)(3 - x_1) = 2 \quad (3 - x_0)(3 - x_1)(3 - x_2) = -2 \quad (3 - x_0)...(3 - x_3) = 4$$

Ainsi, on a $P(3) = 3 + 2 \times (-1) + 2 \times \frac{1}{6} - 2 \times \frac{1}{4} + 4 \times \left(-\frac{5}{24}\right) = 0$.
Il s'agit maintenat de faire un programme, qui prend en entrée des listes x, y et un flottant a et qui renvoie $P(a)$ où P est le polynôme d'interpolation passant par les points (x_i, y_i). On implémente d'abord le calcul des différences finies :

```
def finiteDifferences(x,y):
    N=len(x);
    dif=scipy.zeros((N,N));
    for j in range(N):
        dif[0,j]=y[j];
    for i in range(1,N):
        for j in range(N-i):
            dif[i,j]=(dif[i-1,j+1]-dif[i-1,j])*1.0/(x[j+i]-x[j]);
    return dif;
```

Ceci renvoie une matrice D telle que $D[i, j] = [y_j, ..., y_{j+i}]$ pour $i + j \leq N$. On peut ensuite implémenter la formule d'interpolation de Newton :

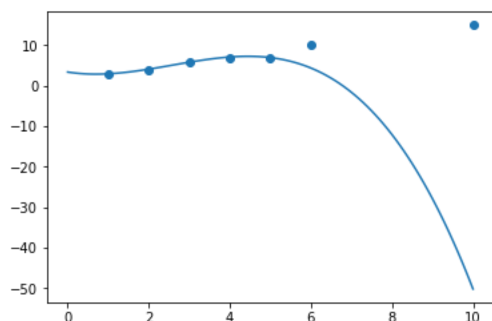
```
def newtInterpol(x,y,a):
    N=len(x);
    dif=finiteDifferences(x,y);
    p=y[0];
    prod=(a-x[0]);
    for i in range(1,N):
        p=p+newt*dif[i,0];
        prod=(a-x[i])*newt;
    return p;
```

Exercice 2. Un peu de recherche sur internet nous permet de trouver la fonction `numpy.polyfit(x, y, ord)`, avec ord=1 on obtient une régression linéaire. Pour tracer les régression d'ordre plus élevé, on définit la fonction

```
def eval(p,x):
    r=0;
    for i in range(len(p)):
        r=r*x+p[i]
    return r;
```

de sorte que $eval(p, x)$ pour $p = numpy.polyfit(x, y, ord)$ soit égal à la fonction de régression évaluée en x.
Pour la régression linéaire, avec le point $(10, 15)$ on a $P(10) = 14.867..$ alors que sans ce point on a $P(10) = 13.1000$. On s'éloigne donc, mais pas de beaucoup.
Pour la régression cubique (ord=3), avec le point $(10, 15)$ on a $P(10) = 15.024$ alors que sans ce point on a $P(10) = -50$. La régression cubique permet de s'approcher plus prêt des points dans l'intervalle entre la plus grande et la plus petite abscisse, mais il s'en éloigne beaucoups hors de cet intervalle.
On peut faire un plot pour voir :

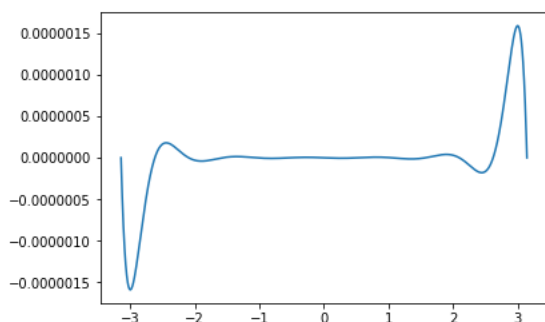
```
p=numpy.polyfit(x, y, ord=3);
z=[(10.0*i)/N for i in range(N+1)];
plt.scatter(x0,y0)
plt.plot(z,[poly(p,i) for i in z])
```



Exercice 3. 1. On se souvient des valeurs remarquables de sin :

k	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
$\sin \frac{k\pi}{6}$	0	$-\frac{\sqrt{3}}{2}$	$-\frac{1}{2}$	-1	$-\frac{1}{2}$	$-\frac{\sqrt{3}}{2}$	0	$\frac{\sqrt{3}}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{\sqrt{3}}{2}$	0

2. On utilise la fonction `newtInterpol` de l'exo 1. On obtient une erreur d'ordre 10^{-6}.



3. Les racines du polynome de Chebychev de degré 7 sont les

$$a_k = \cos\left(\frac{\pi(2k+1)}{14}\right)$$

qu'on renormalise pour obtenir une répartition sur $[-\pi, \pi]$ à la place de $[-1, 1]$:

$$x_k = \pi a_k \ .$$

On obtient une erreur maximale de 0.008, plus importante que dans le cas de 13 points equirepartis. Pour mieux comparer, on considère 13 points de Chebychev : l'erreur est alors nettement plus petite que dans le cas equirépartit, d'ordre 10^{-8}.

Exercice 4. Exercice assez mal posé.
On utilise l'interpolation de Chebychev sur $[0, 10]$, avec un nombre d de points, jusqu'à obtenir la précision souhaitée. Pour cela il faudrait déjà connaître $erf(x)$ avec une assez bonne précision au points de Chebychev ; on va supposer qu'on dispose d'une bonne estimation de ces valeurs (obtenues au préalable par une méthode de calcul d'intégrale, en prenant le temps de calcul qu'il faut) et qu'on cherche un programme pouvant donner $erf(x)$ avec une bonne précision et rapidement pour $x \in [0, 10]$.
Pour estimer l'erreur, on peut utilise la majoration

$$|P(x) - f(x)| \leq \frac{\max_{z\in[0,10]} \left|f^{(n+1)}(z)\right|}{(n+1)!} \left|\prod_{i=0}^{n}(x - x_n)\right|$$

sachant que pour les points de Chebychev on a

$$\left|\prod_{i=0}^{n}(x - x_n)\right| \leq \frac{1}{2^n}\left(\frac{10}{2}\right)^n \ .$$

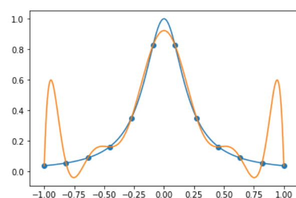
Il s'agit donc d'estimer la dérivée n-ième de f, on souhaite trouver n tel que

$$\left|f^{(n)}(x)\right| \leq \frac{n!2^n}{5^n} * 10^{-8}$$

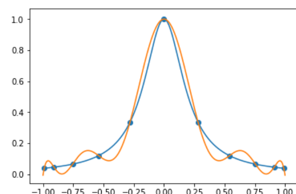
mais estimer $\left|f^{(n)}(z)\right|$ pour tout n peut être délicat.
On se contentera donc d'observer l'erreur numériquement, en comparant avec la fonction `math.erf`.
On observe qu'il faut aller jusqu'à 34 environ pour avoir une erreur strictement inférieure à 10^{-8}.

Notons que $erf(x)$ est très proche de 1 pour x grand (on a déjà $1 - erf(4) \simeq 10^{-8}$. Le résultat obtenu avec l'interpolation est donc inutile pour $x > 4$. Il vaut mieux faire l'interpolation de Chebychev sur $[0, 4]$. Il suffit alors de 29 points pour avoir une erreur inférieure à e-8 ; on pourrait aussi essayer d'autres choix de points d'interpolation que Chebychev.

Exercice 5. On observe bien le phénomène de Runge : le polynome d'interpolation devient de plus en plus différend de f, avec deux pics vers -1 et 1. Pour 11 points par exemple, avec le polynôme d'interpolation en orange :



Si on prend les points de Chebychev tout se passe mieux. Toujours avec 11 points :



Si on prend une fonction un peu moins symétrique ça marche toujours, par exemple tester $f(x) = 1/(1 + 25(x - 0.7)^2)$.
Notons que le phénomène de Runge peut toujours avoir lieu avec des points de Chebychev, si $\max_{x \in [-1,1]} \left| f^{(n)}(x) \right|$ croît plus vite que $2^n n!$ (cf l'estimation de l'erreur d'interpolation).