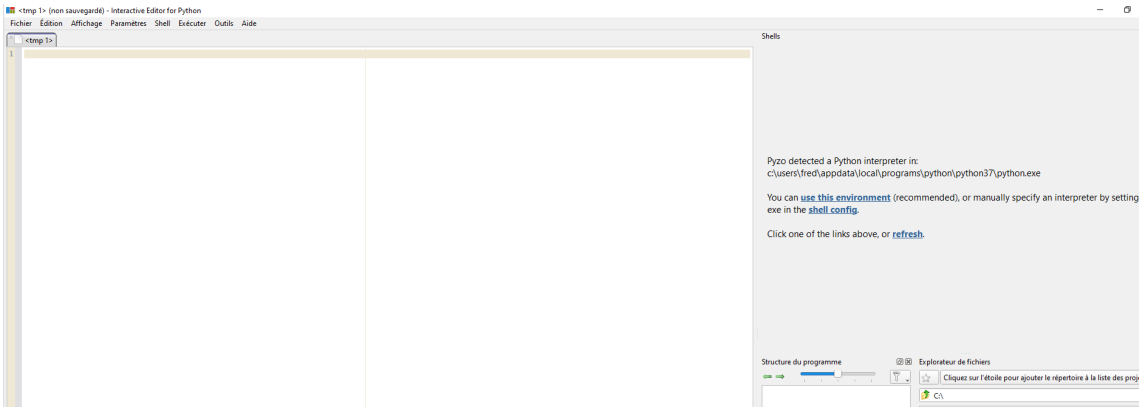
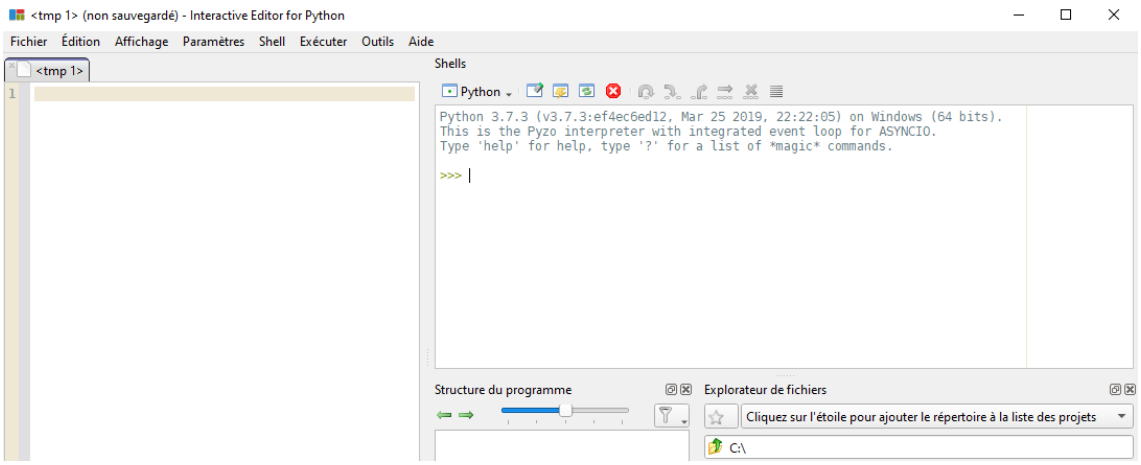
Python : tracé de fonctions

I. Python

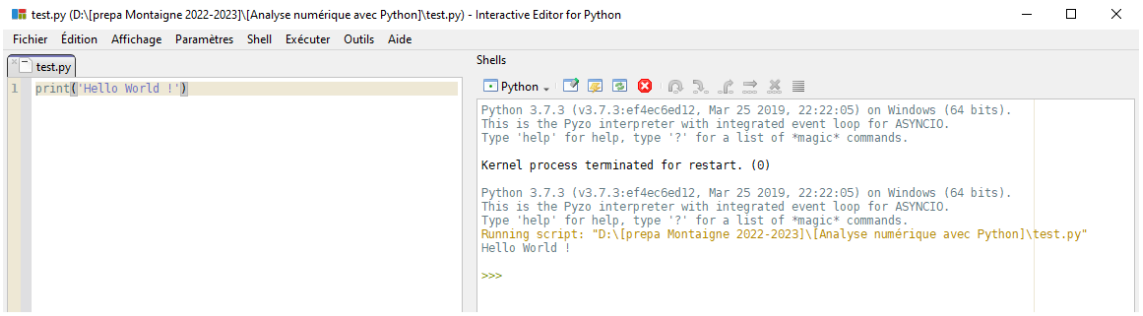
Nous utiliserons l'éditeur Pyzo lors de nos différentes sessions. Après avoir lancé Pyzo, vous devriez obtenir un écran qui ressemble à ceci :



Pour ouvrir la console, il suffit de cliquer sur « use this environment » en haut à droite et nous voilà prêt !



Nous nous servirons dans ce cours uniquement des fenêtres situées à gauche et en haut à droite. À gauche, c'est la fenêtre de l'éditeur dans laquelle nous allons taper tous nos scripts, nos fonctions, etc. C'est cette partie qui est enregistrée quand vous sauvegardez votre fichier (avec une extension .py). En haut à droite, c'est la console. Quand vous exécutez votre script, le résultat sera affiché dans la console. **Pour exécuter votre script, allez dans l'onglet « Exécuter » puis « Démarrer le script ».**



Vous pouvez aussi exécuter directement des instructions dans la console mais elles ne seront exécutées que pour la session active.

II. Fonctions

II.1. Définition

Quand on définit une fonction dans un script, c'est pour pouvoir la réutiliser ensuite. On peut après soit faire afficher le résultat dans le script (en utilisant la commande print), soit l'utiliser dans la console. Voici la syntaxe pour définir la fonction $f : x \mapsto x^2 + 3x - 4$ (attention à l'indentation !) :

```
def f(x):
    y=x**2+3*x-4
    return(y)
```

Si vous exécutez votre script, rien ne s'affichera dans la console. C'est normal, vous avez seulement défini la fonction. Pour afficher un résultat, soit vous tapez directement dans la console $f(2)$, soit dans le script, vous utilisez la commande **print** pour afficher un résultat(en tapant par exemple après voir défini votre fonction print($f(2)$)).

Une fonction est à chaque fois caractérisée par un nom (dans l'exemple précédent f) et une ou des variables muettes (dans l'exemple précédent x). Elle renvoie ce qui est en paramètre de la commande **return**. Un autre exemple :

```
def fonctionde2variables(x,y):
    z=x*y+6
    return(z)
```

III. Module numpy et tracé de fonctions

III.1. Numpy

De nombreuses fonctions sont déjà implémentées. Pour utiliser les fonctions usuelles et effectuer de l'analyse numérique (calcul d'intégrales, résolution d'équations différentielles, etc.) avec Python, on utilise le module **numpy**. On l'importe au début du script afin de le réutiliser ensuite :

```
import numpy as np
```

On peut ensuite calculer différentes valeurs, par exemple, **np.log(2))** calcule $\ln(2)$, **np.sin(np.pi/4))** donne $\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$, que l'on peut également calculer en tapant **np.sqrt(2)/2**. Voici par exemple la syntaxe pour définir la fonction $f : x \mapsto 3\cos(x) + 4\sin(x)$:

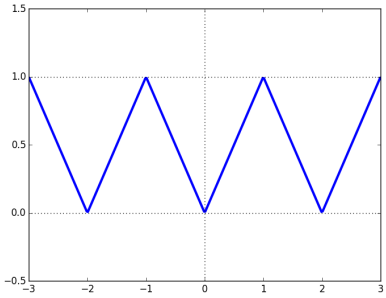
```
import numpy as np
#ceci est ecrit une seule fois en debut de script

def f(x):
    return(3*np.cos(x)+4*np.sin(x))
```

Exercice d'application 1. Définir les fonctions suivantes (et vérifier en calculant $f(1)$, $g(\sqrt{\pi})$ et $h(0)$) :

1) $f : x \mapsto x^2 + 2x + 3\ln(x)$.
2) $g : x \mapsto 2 - 3\sin(x^3)$.
3) $h : x \mapsto 2e^{-3x} + \sqrt{2x+4}$.

Exercice d'application 2. Définir une fonction **valabs** qui renvoie la valeur absolue d'un réel x. *Cette fonction est déjà implémentée sous le nom* ***abs****.*

Exercice d'application 3. Définir la fonction périodique qui a le graphe suivant (on pourra utiliser la fonction partie entière (**np.floor(x)** calcule $\lfloor x \rfloor$) et/ou la commande % qui permet de calculer a modulo b ($a\%b$)) :



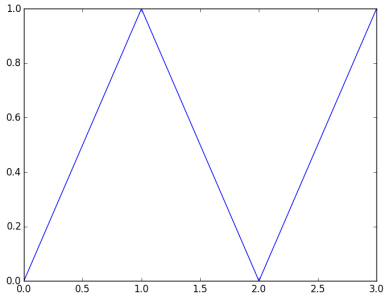
III.2. Tracé de fonctions

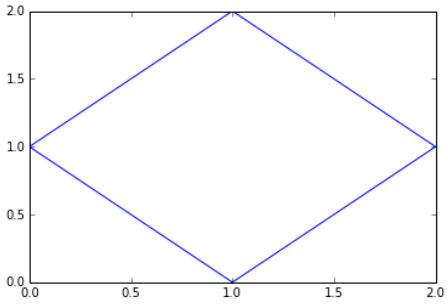
Pour tracer les fonctions, on utilise la fonction **plot** du module **matplotlib.pyplot**. On peut comme dans la partie précédente importer tout le module avant de travailler :

```
import matplotlib.pyplot as plt
```

La fonction **plot** fonctionne de la manière suivante. Elle prend en paramètres deux listes ou deux tableaux de la forme $[x_0, \ldots, x_n]$ et $[y_0, \ldots, y_n]$ et elle trace les points de coordonnées (x_k, y_k) pour k allant de 0 à n et les relie par des traits. Quelques exemples :

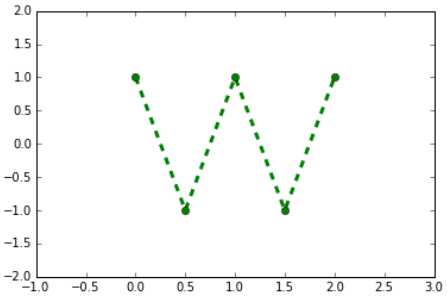
```
plt.plot([0,1,2,3],[0,1,0,1])
plt.plot([1,0,1,2,0],[0,1,2,1,0])
```





Puisque la taille de nos objets ne variera pas, il est souvent plus pratique d'utiliser des tableaux en utilisant la fonction **array** (du module numpy). Par exemple :

```
X=np.array([0,0.5,1,1.5,2])
Y=np.array([1,-1,1,-1,1])
plt.plot(X,Y,color='green',linestyle='dashed',linewidth=3,marker='o')
plt.axis([-1,3,-2,2])
```

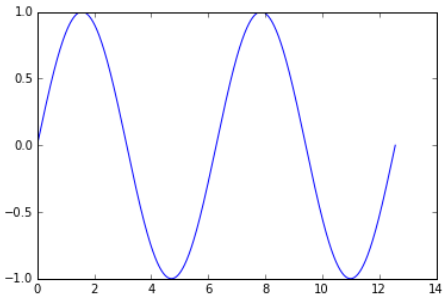


Il existe de nombreuses façons de rendre le tracé plus présentable (légende, couleur, choix de la fenêtre d'affichage, etc.), je vous renvoie à la suite pour des exemples et/ou à l'aide de Python et/ou aux différents exemples que vous pourrez trouver par vous-même. Revenons au tracé. Pour donner l'illusion d'une fonction « lisse », il faut donner en abscisse un tableau avec des valeurs très rapprochées. Il existe deux fonctions pour réaliser cela, toutes les deux présentes dans le module numpy : **linspace** et **arange**.

- $X = np.linspace(a, b, n)$ construit un tableau dont le premier élément est a, le dernier b et contenant n valeurs également espacées.
- $X = np.arange(a, b, h)$ construit un tableau dont le premier élément est a et où les éléments suivants sont espacés de h jusqu'à b (exclu).

Voici le tracé de la fonction sinus sur $[0, 4\pi]$:

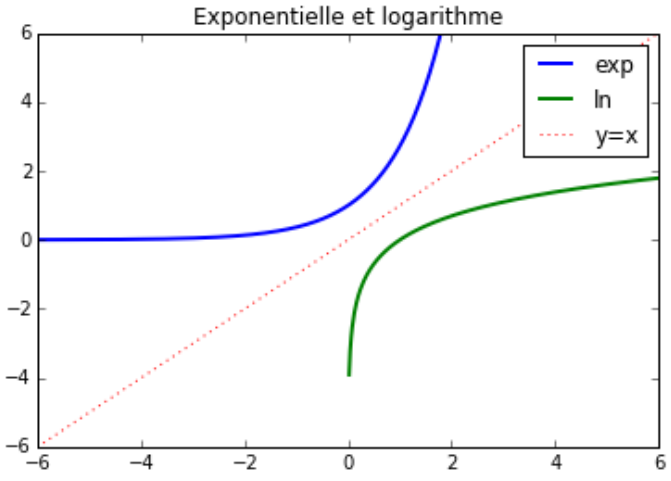
```
X=np.linspace(0,4*np.pi,1000)
Y=np.sin(X)
plt.plot(X,Y)
```



Exercice d'application 4. Définir un tableau X_1 allant de 3 à 10 contenant 200 valeurs également espacées et un tableau X_2 allant de 1 à 10 dont les valeurs sont espacées de 0.3.

On peut bien entendu afficher plusieurs fonctions sur le même graphe, donner des noms aux axes, aux fonctions, un titre, etc. Un dernier exemple :

```
X1=np.linspace(-10,10,1000)
X2=np.linspace(0,20,1000)
Y1=np.exp(X1)
Y2=np.log(X2)
plt.plot(X1,Y1,label='exp',linewidth=2)
plt.plot(X2,Y2,label='ln',linewidth=2)
plt.plot(X1,X1,label='y=x',linestyle='dotted')
plt.axis([-6,6,-6,6])
plt.title('Exponentielle et logarithme')
plt.legend()
```



Exercice d'application 5. Définir et tracer sur $[-10, 10]$ la fonction $f : x \mapsto x + \sin(x)$.

Exercice d'application 6. Définir et tracer sur $[-3, 3]$ la fonction $g : x \mapsto 3\cos(2x) + 4\sin(2x)$. D'après le cours, on a $\exists A \in \mathbb{R}_+, \exists \omega, \varphi \in \mathbb{R} \ / \ \forall x \in \mathbb{R},\ g(x) = A\cos(\omega x - \varphi)$. Déterminer graphiquement les valeurs approchées (ou pas !) de A, ω et φ.

Exercice d'application 7. Voici des mesures du temps d'exécution d'une fonction inconnue en fonction d'un paramètre n :

n	1	2	3	4	5	6	7	8
t	1.03	6.40	15.97	32.04	56.76	88.64	130.22	181.95

1) Tracer sur un graphe le temps t en fonction de n.

2) On veut savoir si le temps de calcul est polynomial en n, autrement dit, puisque $t(1) \approx 1$, on veut l'approcher par une fonction du type $t(n) = n^\alpha$. Tracer sur un graphe le temps $\ln(t)$ en fonction $\ln(n)$ et confirmer cette hypothèse (on déterminera approximativement la valeur de α).

3) Une autre fonction donne ces temps d'exécution :

n	1	2	3	4	5	6	7	8
t	10.8	35.9	115.7	345.8	1341.7	3277.0	7381.0	21678.1

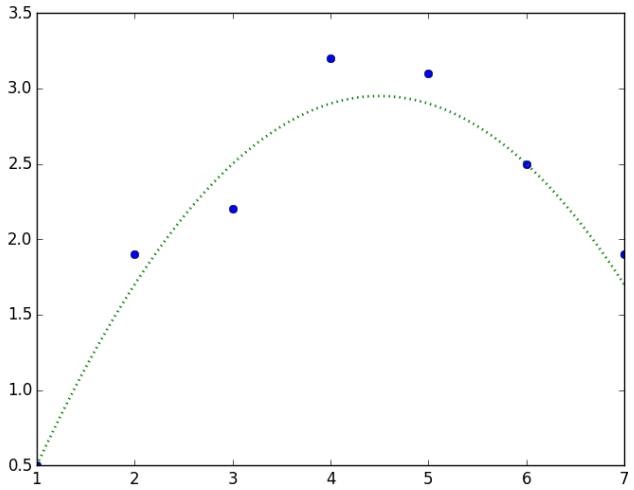
Tracer t en fonction de n. On pense ici que le temps de calcul va être exponentiel en fonction de n, autrement dit que $t(n) = \lambda K^n$. Tracer sur un graphe $\ln(t)$ en fonction de n et confirmer cette hypothèse (on déterminera des valeurs approximatives de λ et de K).

IV. Pour aller plus loin

On cherche parfois à trouver la fonction polynomiale qui approche le mieux un nuage de points. Plutôt que d'ajuster les valeurs graphiquement au hasard, on utilise plutôt la fonction **polyfit** qui prend en argument deux tableaux $X = [x_0, \ldots, x_n]$ et $Y = [y_0, \ldots, y_n]$ et un entier d (le degré du polynôme recherché) et qui renvoie une liste contenant les coefficients du polynôme de degré d qui approche le mieux les points $\{(x_k, y_k),\ k \in [\![0, n]\!]\}$. Le résultat est renvoyé sous la forme d'une liste commençant par le coefficient de X^d, puis celui de X^{d-1}, etc. Un exemple :

```
X=np.array([1,2,3,4,5,6,7])
Y=np.array([0.5,1.9,2.2,3.2,3.1,2.5,1.9])
print(np.polyfit(X,Y,2))
```

Le résultat est alors $[-0.19880952, 1.81547619, -1.1]$. En traçant sur le même graphique le nuage de points et le polynôme $-0.2X^2 + 1.8X - 1.1$, voici ce que l'on obtient :



On peut également tracer des histogrammes avec python (voir la commande **histogram**, toujours dans le module numpy), je vous laisse consulter l'aide de python pour plus d'informations.

V. Correction des exercices

Exercice d'application 1. On a $f(1) = 3$, $g(\sqrt{\pi}) = 2$ et $h(0) = 4$. *En Python, π se tape* ***np.pi*** *après avoir importé numpy sous le nom np.*

```
import numpy as np

def f(x):
    return(x**2+2*x+3*np.log(x))

def g(x):
    return(2-3*np.sin(2x**2))

def h(x):
    return(2*np.exp(-3*x)+np.sqrt(2*x+4)
```

Exercice d'application 2. On utilise un test :

```
def valabs(x):
    if x<0:
        return(-x)
    else:
        return(x)
```

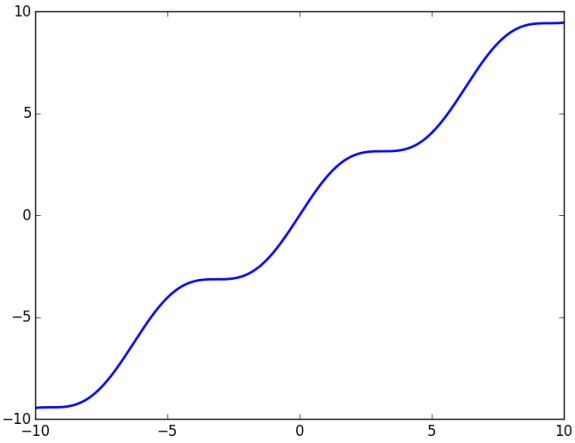
Exercice d'application 3. Encore une fois avec un test :

```
def f(x):
    y=x%2
    if y<1:
        return(y)
    else:
        return(2-y)
```

Exercice d'application 4. On a $X_1 = np.linspace(3, 10, 200)$ et $X_2 = np.arange(1, 10, 0.3)$.

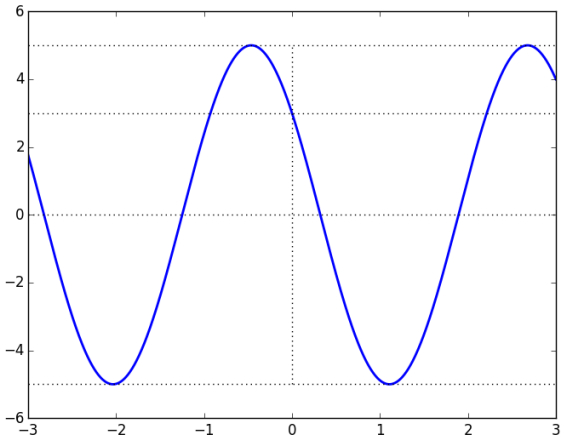
Exercice d'application 5.

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

def f(x):
    return(x+np.sin(x))
X=np.linspace(-10,10,1000)
Y=f(X)
plt.plot(X,Y)
```



Exercice d'application 6.

```
import numpy as np
import matplotlib.pyplot as plt
def g(x):
    return(3*np.cos(2*x)-4*np.sin(2*x))
X=np.linspace(-3,3,1000)
Y=g(X)
plt.plot(X,Y,linewidth=2)

Y2=[0 for k in X]
Y3=[5 for k in X]
Y4=[-5 for k in X]
plt.plot(X,Y2,color='black',linestyle='dotted')
plt.plot(X,Y3,color='black',linestyle='dotted')
plt.plot(X,Y4,color='black',linestyle='dotted')
X3=[0,0]
Y5=[-5,5]
plt.plot(X3,Y5,color='black',linestyle='dotted')
Y6=[g(0) for k in X]
plt.plot(X,Y6,color='black',linestyle='dotted')
```

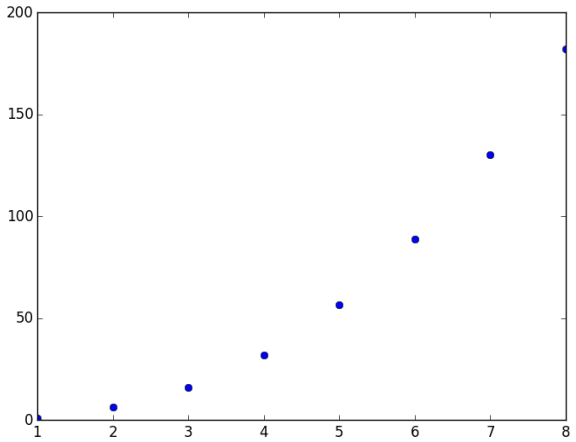


On a alors $A = 5$, $\omega = 2$ (la fonction est π-périodique) et $A\cos(\varphi) = 3$ donc $\cos(\varphi) = \dfrac{3}{5}$. Si on cherche $\varphi \in]-\pi, \pi]$, on remarque que $\varphi < 0$ donc $\varphi = -\arccos(3/5) \approx -0.92$. On observe effectivement graphiquement un maximum proche de $x \approx -0.5$, ce qui donne bien $\varphi \approx 2 \times (-0.5) \approx -1$.

Exercice d'application 7.

1) On trace avec une largeur de ligne de 0 pour n'afficher que des points.

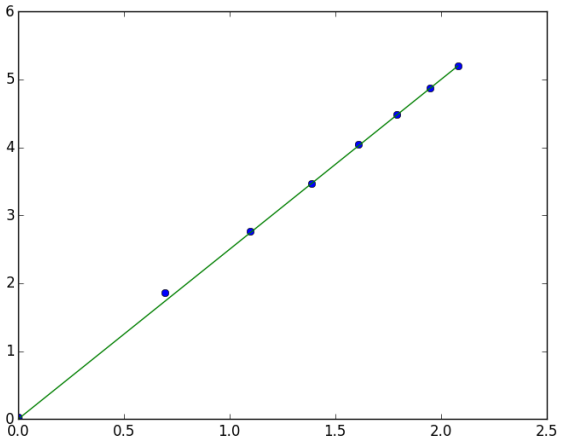
```
import numpy as np
import matplotlib.pyplot as plt
X=np.array([1,2,3,4,5,6,7,8])
T=np.array([1.03,6.4,15.97,32.04,56.76,88.64,130.22, 181.95])
plt.plot(X,T,marker='o',linewidth=0)
```



2) On procède de même. Il est normal d'obtenir une droite puisque si $t = n^\alpha$, alors $\ln(t) = \alpha \ln(n)$ et il faut donc trouver le coefficient directeur de cette droite.

```
X2=np.log(X)
T2=np.log(T)
```

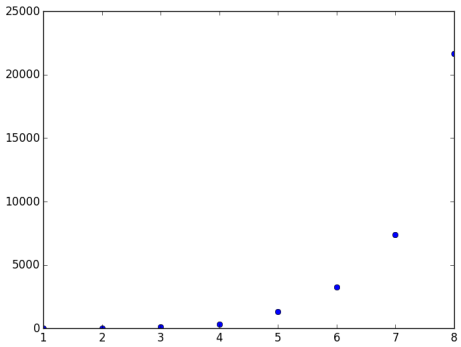
```
plt.plot(X2,T2,marker='o',linewidth=0)
plt.plot(X2,2.5*X2)
```



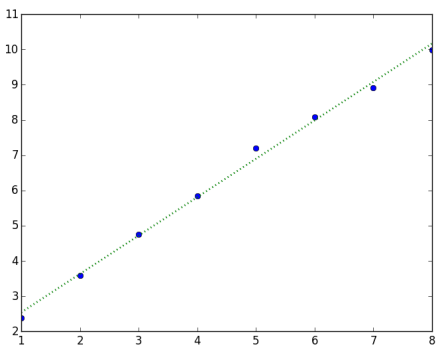
On a donc $\alpha \approx \dfrac{5}{2}$ qui semble bien convenir. *Ce que l'on peut retrouver avec polyfit.*

3) Si $t = \lambda K^n$, alors $\ln(t) = \ln(K)n + \ln(\lambda)$ et on a une équation de degré 1. On procède de la même façon en utilisant **polyfit** en fin de calcul pour trouver la droite de meilleure approximation.

```
X=np.array([1,2,3,4,5,6,7,8])
T=np.array([10.8,35.9,115.7,345.8,1341.7,3277,7381,21678])
plt.plot(X,T,marker='o',linewidth=0)
```




```
T2=np.log(T)
print(np.polyfit(X,T2,1))
plt.plot(X,T2,marker='o',linewidth=0)
plt.plot(X,1.09*X+1.45,linestyle='dotted',linewidth=2)
```



On a donc $K \approx e^{1.09} \approx 2.95$ et $\lambda \approx e^{1.45} \approx 4.29$.