Chapitre 1

Fonctions de référence

I/ Racine carrée

1°) Définition

a) Racine carrée d'un réel positif

La **racine carrée** de x est l'unique réel positif dont le carré vaut x.

b) Ensemble de définition

Seuls les réels positifs ont une racine carrée, on dit que la fonction *racine carrée* est définie sur $[0; +\infty[$.

c) En Python

Par défaut, *Python* n'a pas de fonction *racine carrée*. Mais une des méthodes de l'objet *math* en possède une, qui se note *sqrt* :

sqrt est une abréviation de **sq**uare **r**oo**t**.

```
from math import *
print(sqrt(64))
print(sqrt(-1))
```

Le texte d'erreur signifie que -1 est en-dehors du **domaine de définition** de la fonction.

d) Notation

On note $\sqrt{x}$ la racine carrée de x.

2°) Propriétés

a) Variations

La fonction $x \mapsto \sqrt{x}$ est strictement croissante sur $[0; +\infty[$.

b) Signe

Par définition !

La fonction $x \mapsto \sqrt{x}$ est positive sur $[0; +\infty[$.

II/ Valeur absolue

1°) Définition

On considère le programme de calcul suivant :

1: Prendre un nombre x ;

2: Remplacer son signe, quel qu'il soit, par un "+" (autrement dit, oublier son signe)

3: Retourner le résultat.

Ceci définit une fonction sur $\mathbb{R}$. On appelle **valeur absolue** cette fonction.

a) Fonction affine par intervalle

Si x est positif, le programme ci-dessus ne le change pas : La valeur absolue d'un nombre positif est le nombre lui-même. Par contre si x est négatif, le rendre positif le remplace par son opposé :

```
def absolue(x):
        if x>=0:
                return x
        else
                return -x
```

b) Notation

La valeur absolue de x se note $|x|$.
En *Python*, elle se note *abs* :

```
print([abs(x) for x in range(-5,6)])
```

2˚) Propriétés

a) Domaine de définition

La fonction $x \mapsto |x|$ est définie sur $\mathbb{R}$.

b) Variations

La fonction $x \mapsto |x|$ est strictement croissante sur $[0; +\infty[$ et strictement décroissante sur $]-\infty; 0]$.

c) Signe

La fonction $x \mapsto |x|$ est positive sur $\mathbb{R}$.

Par définition !

III/ Construction de fonctions

On peut construire des fonctions à partir des fonctions de référence par somme, produit etc. Alors

1: Additionner une constante à une fonction ne change pas ses variations ;

2: Multiplier une fonction parune constante positive ne change pas ses variations, la multiplier par une constante négative inverse ses variations ;

3: La racine carrée d'une fonction positive a les mêmes variations que celle-ci ;

4: L'inverse d'une fonction croissante est décroissante, l'inverse d'une fonction décroissante est croissante.

Chapitre 2

Alignement dans le plan repéré

I/ Objets géométriques en Python

1°) Point

```
class Point:
        def __init__(self,x,y):
                self.x=x
                self.y=y
        def __str__(self):
                return '('+str(self.x)+';'+str(self.y)+')'
        def vecteur(self,p):
                return Vecteur(p.x-self.x,p.y-self.y)
```

2°) Vecteur

```
from math import *

class Vecteur:
        def __init__(self,x,y):
                self.x=x
                self.y=y
        def __str__(self):
                return '('+str(self.x)+';'+str(self.y)+')'
        def __add__(self,p):
                return Vecteur(self.x+p.x,self.y+p.y)
        def norme(self):
                return hypot(self.x,self.y)
        def __rmul__(self,r):
```

```
                return Vecteur(self.x*r,self.y*r)
```

a) Norme

La distance AB s'appelle la **norme** du vecteur $\overrightarrow{AB}$ et se note $\left\|\overrightarrow{AB}\right\|$. **Si le repère est orthonormé**, on peut ajouter une méthode *norme* grâce à l'import de l'objet *math* (ci-dessus). Elle s'appelle par

```
u=Vecteur(4,3)
print(v.norme())
```

3°) Vecteurs colinéaires

a) Déterminant

Le **déterminant** de deux vecteurs $\vec{u}\begin{pmatrix} x_{\vec{u}} \\ y_{\vec{u}} \end{pmatrix}$ et $\vec{v}\begin{pmatrix} x_{\vec{v}} \\ y_{\vec{v}} \end{pmatrix}$ est le nombre $x_{\vec{u}} \times y_{\vec{v}} - x_{\vec{v}} \times y_{\vec{u}}$.

b) Colinéarité

Deux vecteurs sont colinéaires si et seulement si leur déterminant est nul.

c) Méthode

```
        def determinant(self,v):
                return self.x*v.y-self.y*v.x
        def colin(self,v):
                return self.determinant(v)==0
```

II/ Droite du plan repéré

1°) Droite comme objet Python

```
class Droite:
        def __init__(self,A,B):
                self.A=A
                self.B=B
```

La droite est définie par deux points A et B.

2°) Vecteur directeur

La vecteur $\overrightarrow{AB}$ est un **vecteur directeur** de la droite (AB). Tout vecteur non nul colinéaire à $\overrightarrow{AB}$ est aussi directeur de (AB). C'est une méthode de l'objet *droite* :

```
        def directeur(self):
                return self.A.vecteur(self.B)
```

3°) Équation cartésienne

En écrivant que le point $M(x_M; y_M)$ est aligné avec A et B, on obtient successivement (parce que les vecteurs $\overrightarrow{AM}$ et $\overrightarrow{AB}(a, b)$ doivent pour cela être colinéaires) :

$$a(y_M - y_A) - b(x_M - x_A) = 0$$
$$-bx_M + ay_M = ay_A - bx_A$$
$$-bx + ay = c$$

avec

$$c = ay_A - bx_A$$

Ce qui donne une équation cartésienne de (AB) :

```
        def __str__(self):
                a=-self.directeur().y
                b=self.directeur().x
                c=-self.directeur().y*self.A.x
                c+=self.directeur().x*self.A.y
                eq='('+str(a)+')x+('+str(b)+')y='+str(c)
                return eq
```

Chapitre 3

Statistiques descriptives

I/ Simulation

1°) Tableaux

Pour simuler 100 lancers d'un dé équilibré, on peut mettre les résultats des 100 lancers dans un tableau :

```
from random import *
donnees=[randint(1,6) for n in range(100)]
```

Le résultat du premier lancer est alors stocké dans *donnees[0]*.

2°) Tableaux triés

Une fois un tableau trié dans l'ordre croissant avec *sort()*, on repère les éléments du tableau trié qui sont au quart, au milieu ou aux trois quarts :

```
def mediane(tableau):
        tableau.sort()
        n=len(tableau)
        if n%2==1:
                return tableau[int(n/2)]
        else:
                return (tableau[int(n/2-1)]+tableau[int(n/2)])/2

def Q1(tableau):
        tableau.sort()
        n=len(tableau)
        return tableau[int(n/4)]
```

```
def Q3(tableau):
        tableau.sort()
        n=len(tableau)
        return tableau[int(3*n/4)]
```

3°) Boîtes à moustaches

II/ Moyenne

Pour éviter d'avoir un quotient euclidien, on ajoute le réel zéro à la longueur du tableau, ce qui a pour effet de la convertir en réel :

```
def moyenne(tableau):
        return sum(tableau)/len(tableau)
```

On peut maintenant calculer la moyenne de n'importe quel tableau :

III/ Écart-type

1°) Variance

```
def variance(tableau):
        m=moyenne(tableau)
        return moyenne([(x-m)**2 for x in tableau])
```

La variance est la moyenne des carrés des écarts à la moyenne.

2°) Écart-type

L'écart-type est la racine carrée de la variance :

```
from math import *

def ecartype(tableau):
        return sqrt(variance(tableau))

print(variance(donnees))
```

Chapitre 4

Nombre dérivé

I/ Définition

1°) Nombre dérivé

Lorsque le quotient $\dfrac{f(x+h)-f(x)}{h}$ se rapproche d'une limite a lorsque h tend vers 0, on dit que f est **dérivable** en a. Dans ce cas, la limite est appelée **nombre dérivé** de f en a et noté $f'(a)$

2°) Tangente

Le nombre dérivé de f en a est le coefficient directeur de la tangente en $(a; f(a))$ à la représentation graphique de f.

II/ Fonction dérivée

1°) Algorithme

On peut implémenter une valeur approchée du nombre dérivé comme ceci :

```
def NDer(f,a):
    h=1e-10
    return (f(a+h)-f(a))/h
```

Pour connaître le nombre dérivé de $x \mapsto x^2 - 2$ en 3, on peut faire

```
def f(x):
    return x**2-2
```

```
print(NDer(f,3))
```

Ceci définit une fonction :

2°) Définition

La fonction f' qui, à a, associe le nombre dérivé de f en a, est une fonction ne dépendant que de f, appelée **fonction dérivée de** f.

3°) Exemples

1: La dérivée d'une fonction affine est son soefficient directeur ;

2: La dérivée de $x \mapsto \sqrt{x}$ est $x \mapsto \dfrac{1}{2\sqrt{x}}$;

3: La dérivée de $x \mapsto \dfrac{1}{x}$ est $x \mapsto -\dfrac{1}{x^2}$;

4: La dérivée de $x \mapsto x^n$ est $x \mapsto n \times x^{n-1}$ (si $n \in \mathbb{N}$)

4°) Propriétés

a) Somme

La dérivée d'une somme est la somme des dérivées : $(u+v)' = u' + v'$.

b) Produit

$(uv)' = u'v + uv'$

c) Quotient

$\left(\dfrac{u}{v}\right)' = \dfrac{u'v - uv'}{v^2}$

III/ Variations

1°) Signe de la dérivée

Une fonction f dérivable est croissante si et seulement si sa dérivée f' est positive, et décroissante si et seulement si sa dérivée est négative. On peut donc établir le tableau de variations de f à partir du tableau de signes de sa dérivée.

2°) Extrema

Une fonction dérivable f passe par un maximum ou par un minimum en un nombre a tel que $f'(a) = 0$.

Chapitre 5

Second degré

I/ Définitions

1°) Polynômes

a) Monômes

Un **monôme** est le produit d'une constante par une puissance de x. L'exposant est le **degré** du monôme. Exemple : $x \mapsto 3x^7$ est de degré 7.

b) Polynômes

Un **polynôme** est une somme de monômes. Le plus grand des degrés des monômes s'appelle le **degré** du polynôme.

Exemples :

1: Un polynôme de degré 0 est une fonction constante.

2: Un polynôme du premier degré est une fonction affine.

3: $x^3 - 2x^2 + 7x - 5$ est de degré 3.

2°) Trinômes

Un **trinôme** est un polynôme du second degré.

Il tire son nom du fait qu'il est composé de 3 monômes : ax^2, bx et c.

3°) Le trinôme comme objet

On peut définir un trinôme comme un objet ayant pour propriétés les trois coefficients a, b et c :

```
class Trinome:
        def __init__(self,a,b,c):
                self.a=a
                self.b=b
                self.c=c
```

II/ Forme canonique

1°) Formule

En développant $a\left(x+\frac{b}{2a}\right)^2-\frac{b^2-4ac}{4a}$, on trouve ax^2+bx+c. La première forme est la **forme canonique** de ax^2+bx+c.

2°) Sommet

Il en résulte que le point de coordonnées $\left(-\frac{b}{2a};-\frac{b^2-4ac}{4a}\right)$ est le **sommet** de la parabole qui représente la fonction ax^2+bx+c.

3°) Définition

Le nombre $\Delta=b^2-4ac$ est appelé **discriminant** du trinôme. C'est une propriété de celui-ci :

```
class Trinome:
        def __init__(self,a,b,c):
                self.a=a
                self.b=b
                self.c=c
                self.Delta=b*b-4*a*c
```

4°) Forme canonique en Python

```
        def canonique(self):
                self.xs=-self.b/(2*self.a)
                self.ys=-self.Delta/(4*self.a)
                fc=str(self.a)+'(x+('+str(-self.xs)+'))**2'
                fc+='+'+str(self.ys)
```

```
                return fc
```

III/ Racines d'un trinôme

1°) Définitions

On appelle **racines**, ou **zéros** d'une fonction f, les antécédents de 0 par f, c'est-à-dire les solutions de l'équation $f(x) = 0$.

2°) Cas des trinômes

Le nombre de solutions de l'équation $ax^2 + bx + c = 0$ est déterminé par le signe de Δ :

D'où son nom : Il permet de discriminer les différents cas.

a) Discriminant négatif

Si $\Delta < 0$, l'équation $ax^2 + bx + c = 0$ n'a pas de solution : $\mathcal{S} = \emptyset$.

b) Discriminant nul

Si $\Delta = 0$, l'équation $ax^2 + bx + c = 0$ a une seule solution $-\dfrac{b}{2a}$: $\mathcal{S} = \left\{-\dfrac{b}{2a}\right\}$.

c) Discriminant positif

Si $\Delta > 0$, l'équation $ax^2 + bx + c = 0$ a deux solutions $\dfrac{-b - \sqrt{\Delta}}{2a}$ et $\dfrac{-b + \sqrt{\Delta}}{2a}$.

3°) Résolution de l'équation du second degré

```
        def racines(self):
                if self.Delta<0:
                        self.r='{}'
                else:
                        self.d=sqrt(self.Delta)
                        x1=(-self.b-self.d)/(2*self.a)
                        x2=(-self.b+self.d)/(2*self.a)
```

```
                self.r='{'+str(x1)+';'+str(x2)+'}'
            return self.r
```

Pour résoudre $x^2 - 5x + 6 = 0$, il suffit d'écrire

```
t=Trinome(1,-5,6)
print(t.racines())
```

IV/ Dérivée

1°) Résultat

La dérivée de $ax^2 + bx + c$ est $2ax + b$; c'est une fonction affine.

2°) Méthode

On peut définir la dérivée comme une méthode de l'objet *trinôme* :

```
    def derivee(self):
        return Trinome(0,2*self.a,self.b)
```

Pour calculer la dérivée de $x^2 - 5x + 6$, il suffit alors de faire

```
t=Trinome(1,-5,6)
print(t.derivee())
```

C'est avec ce genre d'algorithme que fonctionnent les logiciels de **calcul formel** comme *Maxima* ou *Xcas*.

Chapitre 6

Suites numériques

I/ Définition

1°) Suite de réels

Une suite u est une fonction définie sur $\mathbb{N}$. L'image de n est notée $u(n)$ mais aussi u_n.

2°) Exemple

a) Tri d'une liste

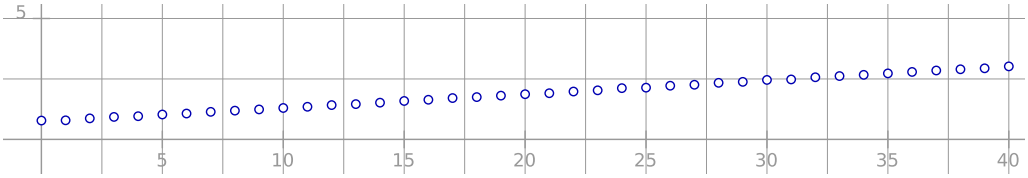
Pour trier dans l'ordre croissant une liste comprenant n nombres, il faut un certain temps qui dépend de n : Le temps mis à trier une liste de n nombres est une suite $(t_n)_{n\in\mathbb{N}}$. Pour mesurer ce temps, on peut créer deux objets de type *time* (donnant l'heure de leur création), l'un avant d'effectuer le tri, l'autre après. Leur différence sera la durée du tri. Mais pour avoir des mesures plus précises, on va faire le tri 1 000 000 fois et le temps affiché sera donc le temps d'une opération mélange-tri en micosecondes :

```
from time import *
liste=[i for i in range(101)]
begin=clock()
for i in range(1000000):
        liste.reverse()
        liste.sort()
end=clock()
print(end-begin)
```

On lit 6.23 ce qui veut dire qu'une opération de mélange suivi de tri prend 6,23 microsecondes si la taille du tableau est 100 nombres.

b) Temps de tri

La suite des mesures des temps de tri t_n d'une liste de taille n par Python, représentée graphiquement, ressemble à ceci :



II/ Suite définie par formule

1°) Exemple

La suite $u_n = \dfrac{n}{n+1}$ est simple à étudier parce qu'on peut calculer u_n par une formule :

```
def u(n):
    return n/(n+1)

for n in range(40):
    print('u('+str(n)+')='+str(u(n)))
```

2°) Variations

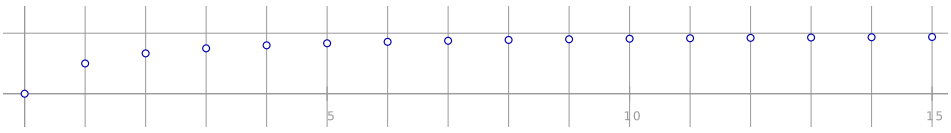
On constate que chaque terme de la suite u_n est supérieur au précédent : On dit que u_n est croissante.

3°) Limite

On constate également que plus n est grand, plus u_n est proche de 1 :

```
for n in range(10):
    print('u'+str(10**n)+')='+str(u(10**n)))
```

On dit que u_n **tend vers** 1, ou que 1 est la **limite** de u_n. On note $\lim\limits_{n\to\infty} u_n = 1$.



III/ Suites récurrentes

1°) Définition

Une suite u_n est dite récurrente si on dispose d'un moyen pour calculer u_n à partir de u_{n-1} (et éventuellement d'autres termes antérieurs).

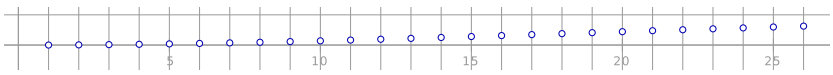
2°) Exemples

a) Anniversaires

Si on appelle p_n la probabilité que deux personnes parmi n aient leur anniversaire le même jour, on définit une suite pour $n \geqslant 1$ pour $n = 1$, la probabilité est nulle). Alors la suite $v_n = 1 - p_n$ est récurrente, parce que $v_{n+1} = \dfrac{365 - n}{365} \times v_n$:

```
v=1
for n in range(2,30):
    v*=(366-n)/365
    print(1-v)
```

La suite est croissante :



Plus surprenant, elle tend vers 1.

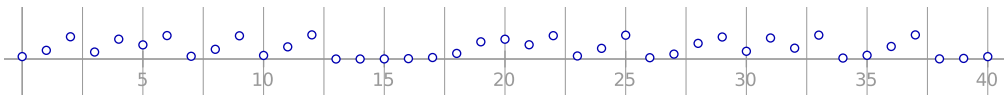
Plus on est nombreux, plus on a de chances que deux anniversaires coïncident.

b) Suite pseudo-aléatoire

La suite c_n définie par $c_0 = 0,1$ et $c_{n+1} = 4(c_n - c_n^2)$ n'est ni croissante, ni décroissante, et n'a pas de limite :

```
c=0.1
for n in range(40):
    c=4*(c-c**2)
    print(n,c)
```

Voici la représentation graphique de la suite :



C'est avec ce genre de suite récurrentes qu'on simule le hasard avec les calculatrices et les ordinateurs.

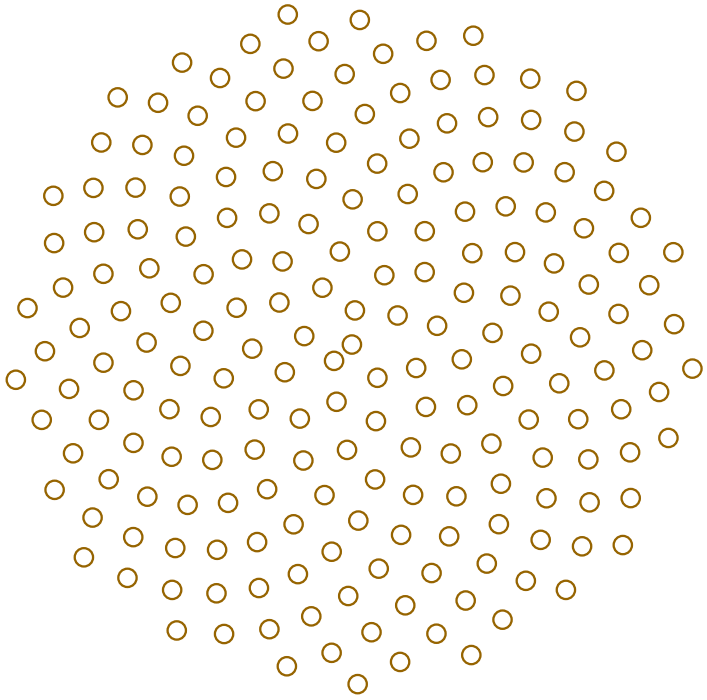
c) Suite de Fibonacci

La suite de Fibonacci F_n est définie par $F_0 = 1$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$:

```
a,b = 1,1
for n in range(2,40):
    a,b=a+b,a
    print(n,a)
```

Les termes de cette suite interviennent souvent comme nombres de spirales dans les méristèmes de certaines plantes :

Ainsi que dans l'ananas...



Chapitre 7

Radians

I/ Conversion

1°) De degrés en radians

```
from math import *
print(radians(30))
print(pi/6)
```

2°) De radians en degrés

```
from math import *
print(degrees(pi/4))
```

II/ Mesure principale

1°) Définition

Si on additionne 2π à un angle en radians, on se retrouve à la même position sur le cercle trigonométrique. On dit qu'un angle a une infinité de mesures différentes. Celle qui est entre 0 et 2π est la **mesure principale** de l'angle.

2°) Calcul

On peut calculer la mesure principale de x avec l'algorithme suivant :

```
from math import *

def principale(x):
        if x<0:
                while x<0:
                        x+=2*pi
        else:
                while x>2*pi:
                        x-=2*pi
        return x

print(principale(17*pi/6))
```

Mais c'est plus rapide de faire

```
print((17*pi/6)%(2*pi))
```

III/ Fonctions trigonométriques

1°) Cosinus

```
from math import *
print(cos(pi/6))
print(sqrt(3)/2)
```

2°) Sinus

```
from math import *
print(sin(pi/4))
print(sqrt(2)/2)
```

IV/ Angle orienté de vecteurs

1°) Notation

L'angle orienté entre $\overrightarrow{AB}$ et $\overrightarrow{AC}$ (en radians) se note $\left(\overrightarrow{AB}, \overrightarrow{AC}\right)$.

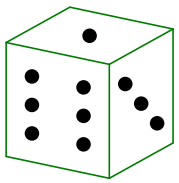
2°) Propriété

$$(\vec{u}, \vec{v}) + (\vec{v}, \vec{w}) = (\vec{u}, \vec{w})$$

Cette propriété s'appelle la relation de Chasles pour les angles.

Chapitre 8

Variables aléatoires



I/ Définitions

1°) Variable aléatoire

Lorsque les éventualités d'une expérience de probabilité sont des nombres, les évènements constituent une variable aléatoire.

2°) Exemples

a) Dé

On lance un dé pipé, tel que les probabilités des différents résultats possibles sont données dans ce tableau :

Évènements	1	2	3	4	5	6
Probabilités	0,15	0,15	0,15	0,15	0,2	0,2

Les nombres de la deuxième ligne du tableau constituent ce qu'on appelle la **loi de la variable aléatoire**.

Pour simuler cette expérience aléatoire, on peut construire la loi sous forme d'un tableau :

```
from random import *

omega=range(1,7)
loi=[0]+[0.15]*4+[0.2]*2

def proba(X):
        return sum(loi[x] for x in X)
```

Ici la variable aléatoire s'appelle *omega*. Alors si on affiche la probabilité de *omega*, on doit trouver 1 :

```
print(proba(omega))
```

b) Pile ou face

Les résultats « pile » et « face » n'étant pas des nombres, le lancer d'une pièce n'est pas une variable aléatoire. Mais on peut décider de jouer à un jeu de hasard en suivant ces règles :

1: Si la pièce tombe sur « pile », on gagne 3 €.

2: Sinon on perd 2 €.

Le gain (positif si on gagne, négatif si on perd) à ce jeu est une variable aléatoire, prenant les valeurs 3 et -2 avec les probabilités respectives d'avoir « pile » et « face ».

c) Durée d'un jeu de hasard

Au début d'une partie de « dada » (ou « petits chevaux »), on lance un dé jusqu'à ce qu'on ait un 6. La durée de cette phase du jeu (le nombre de fois qu'il a fallu lancer le dé) est une variable aléatoire entière.

II/ Espérance et Écart-type

1°) Espérance

```
def esperance(X):
        return sum(loi[x]*x for x in X)

print(esperance(omega))
```

Si on mesure un grand nombre de fois une variable aléatoire, la moyenne des mesures s'approche de ce nombre.

2°) Écart-type

a) Variance

```
def variance(X):
        m=esperance(X)
        return esperance([(x-m)**2 for x in X])
```

b) Écart-type

```
from math import *

def ecartype(X):
        return sqrt(variance(X))

print(ecartype(omega))
```

III/ Variables de Bernoulli

1°) Expérience de Bernoulli

a) Définition

Une expérience aléatoire est dite **de Bernoulli** si son univers contient deux éléments. Dans ce cas chacun est le contraire de l'autre. Les deux éventualités sont en général appelées « succès » et « échec ». La probabilité du succès est notée p.

b) Exemples

1: Pile ou face : En admettant que la pièce ne peut pas tomber sur la tranche, on n'a que deux éventualités possibles. p est la probabilité d'avoir « pile ».

2: Quand on évoque « la probabilité d'avoir le Bac », on s'intéresse à une expérience de Bernoulli, et plus précisément à p.

3: Monsieur et Madame Bonacci attendent un enfant, et se demandent si ce sera un garçon ou une fille. Dans ce cas p est proche de 0,5.

2°) Exemple de variable aléatoire

Si $p = 0,4$ (la pièce est truquée), l'espérance de la variable aléatoire précédente est $0,4 \times 3 + 0,6 \times (-2) = 1,2 - 1,2 = 0$: L'espérance est nulle, et donc le jeu est équilibré.

La variance est alors $0,4 \times (3-0)^2 + 0,6 \times (-2-0)^2 = 0,4 \times 9 + 0,6 \times 4 = 3,6 + 2,4 = 6$, donc son écart-type est $\sqrt{6} \simeq 2,45$.

3°) Variable aléatoire de Bernoulli

a) Définition

On dit que X est une variable aléatoire de Bernoulli si $P(X = 1) = p$ et $P(X = 0) = 1 - p$. Autrement dit, X est égale à 0 ou 1 selon le résultat d'un lancer à pile ou face.

b) Simulation

Pour simuler une variable aléatoire de Bernoulli de paramètre $p = 0,4$, on peut placer 400 boules dans une urne portant le numéro ① et 600 boules dans la même urne portant le numéro ⓪. On tire une boule de cette urne au hasard, le numéro obtenu est une variable aléatoire de Bernoulli de paramètre $p = 0,4$. Alternativement on peut aussi faire ainsi :

```
from random import *

X=1 if random()<0.4 else 0
```

Chapitre 9

Suites de référence

I/ Suites arithmétiques

1°) Définition

Une suite u_n est dite **arithmétique** si la différence entre deux termes consécutifs est constante. Dans ce cas, cette constante s'appelle la **raison** de la suite.

2°) Propriétés

a) Suite arithmétique vue comme suite récurrente

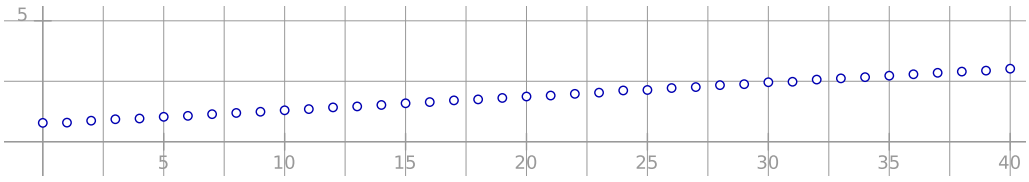
Puisque par définition, $u_{n+1} = u_n + r$, la suite est récurrente.

b) Terme général

On montre que $\forall n \in \mathbb{N}, u_n = u_0 + n \times r$. Une suite arithmétique est donc une fonction affine définie sur $\mathbb{N}$.

c) Représentation graphique

Il en résulte que les points représentant une suite arithmétique sont alignés. Par exemple, la représentation graphique des temps de tri de tableaux de n nombres suggère que la suite t_n est arithmétique :



La raison est le temps additionnel pour trier un élément de plus.

En calculant différentes valeurs de $u_{n+1}-u_n$ dans un tableur, on estime sa raison à 0,056 microsecondes (donc $u_{n+1} = u_n + 0,056$) et donc son premier terme à 0,7. Ce qui fournit un moyen rapide d'estimer le temps qu'il faudra pour trier n nombres : $u_n \simeq 0,056 \times n + 0,7$. Par exemple, pour trier 100 nombres, on estime qu'il faudra $0,056 \times 100 + 0,7 = 6,3$ microsecondes, ce qui est proche de la mesure effectuée dans le chapitre sur les suites. Et pour trier un tableau comportant un million de nombres, la même formule prévoit un temps de 56 000,7 microsecondes, soit un vingtième de seconde. En fait le script ci-dessous affiche un peu plus (0,07 seconde) :

```
from time import *
liste=[i for i in range(1000001)]
begin=clock()
liste.reverse()
liste.sort()
end=clock()
print(end-begin)
```

3°) Somme des termes

On a $u_0+u_1+u_2+...+u_n = (n+1)\dfrac{u_0+u_n}{2}$: Pour additionner les termes d'une suite arithmétique, on fait comme si chacun de ses termes était égal à la moyenne entre le premier et le dernier.

II/ Suites géométriques

1°) Définition

raison vient du latin *ratio* qui veut dire « quotient ».

Une suite u_n est dite **géométrique** si le quotient entre deux termes consécutifs est constant. Dans ce cas, cette constante s'appelle la **raison** de la suite.

2°) Propriétés

a) Suite géométrique vue comme suite récurrente

Puisque par définition, $u_{n+1} = u_n \times r$, la suite est récurrente.

b) Terme général

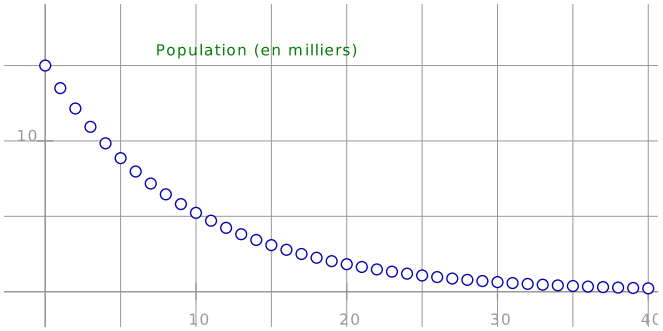
On montre que $\forall n \in \mathbb{N}, u_n = u_0 \times r^n$.

3°) Exemple

Une population de baleines, initialement estimée à 15 000 individus, fait l'objet d'une chasse telle que chaque année, elle diminue de 10 %. La suite des populations chaque année est alors géométrique de raison 0,9 et de premier terme 15 000. On peut suivre son évolution avec

```
u=15000
for n in range(40):
    u*=0.9
    print('u('+str(n)+')='+str(round(u,0)))
```

On constate que la suite est décroissante :



Les baleines sont de moins en moins nombreuses à cause de la chasse.

La population est destinée à une extinction puisque la suite tend vers 0 ($\lim\limits_{n\to\infty} u_n = 0$). Pour trouver combien de temps cela va prendre, on peut faire ceci :

```
u=15000
n=0
while u>1:
    n+=1
    u*=0.9

print(n)
```

4°) Somme des termes

On a $u_0 + u_1 + u_2 + ... + u_n = u_0 \dfrac{1 - r^{n+1}}{1 - r}$.

Chapitre 10

Produit scalaire

Dans tout ce chapitre, le repère est orthonormé, condition nécessaire pour pouvoir parler de distances et d'angles.

orthonormé parce que $\|\vec{\imath}\| = \|\vec{\jmath}\|$.

I/ Définitions

1°) Avec les coordonnées

a) Définition

Dans un repère orthonormé, la somme dont le premier terme est le produit des abscisses de $\vec{u}$ et $\vec{v}$ et dont le second terme est le produit de leurs ordonnées, est appelée **produit scalaire** des deux vecteurs.

b) Notation

Le produit scalaire de $\vec{u}$ et $\vec{v}$ est noté $\vec{u}.\vec{v}$.

2°) Méthode

On peut définir le produit scalaire comme une méthode de l'objet *vecteur* :

```
        def __mul__(self,v):
            return self.x*v.x+self.y*v.y
```

Avec ça il suffit d'écrire le symbole de multiplication entre deux vecteurs pour calculer leur produit scalaire :

```
u=Vecteur(3,1)
v=Vecteur(2,6)
print(u*v)
```

3°) Propriétés

1: $\vec{u}.\vec{v} = \vec{v}.\vec{u}$

2: $(\vec{u}+\vec{v}).\vec{w} = \vec{u}.\vec{w} + \vec{v}.\vec{w}$

3: $\vec{u}.(k \times \vec{v}) = k \times \vec{u}.\vec{v}$.

4: $\vec{u}.\vec{u} = \|\vec{u}\|$. Ce produit scalaire se note $\vec{u}^2$. Plus généralement, si l'unité du repère **orthonormé** est le centimètre, les produits scalaires se mesurent en centimètres carrés, même s'ils sont négatifs.

II/ Autres expressions

Les définitions suivantes du produit scalaire ont l'avantage de ne pas dépendre du repère orthonormé.

1°) Avec l'angle

a) Expression

$$\vec{u}.\vec{v} = \|\vec{u}\| \times \|\vec{v}\| \times \cos(\vec{u},\vec{v})$$

b) Cas particulier

Deux vecteurs sont orthogonaux si et seulement si leur produit scalaire est nul

2°) Avec les normes

a) Découverte d'un résultat expérimental

```
u=Vecteur(3,1)
v=Vecteur(2,6)
print(u*v)
print((u+v).norme()**2-u.norme()**2-v.norme()**2)
```

b) Expression

On admet qu'en général, $\|\vec{u}+\vec{v}\|^2 - \|\vec{u}\|^2 - \|\vec{v}\|^2 = 2\vec{u}.\vec{v}$, ce qui permet d'écrire le produit scalaire de $\vec{u}$ et $\vec{v}$ par $\vec{u}.\vec{v} = \dfrac{\|\vec{u}+\vec{v}\|^2 - \|\vec{u}\|^2 - \|\vec{v}\|^2}{2}$.

c) Application au triangle

En posant $\vec{u} = \overrightarrow{BA}$ et $\vec{v} = \overrightarrow{AC}$, Chasles donne $\vec{u} + \vec{v} = \overrightarrow{BC}$. L'égalité précédente s'écrit alors

$$BC^2 = AB^2 + AC^2 - 2 \times AB \times AC \times \cos \widehat{BAC}$$

Ce résultat qui généralise le théorème de $\Pi\upsilon\theta\alpha\gamma\omega\rho\alpha\sigma$, est appelé **théorème d'Al Kashi**, du nom de l'astronome iranien de l'observatoire de Samarkande.

Chapitre 11

Variables aléatoires binomiales

I/ Indépendance

1°) Définition

On dit que deux évènements A et B sont indépendants si

$$P(A \cap B) = P(A) \times P(B)$$

2°) Méthode

```
def indep(A,B):
        return proba(A&B)==proba(A)*proba(B)
```

II/ Variables binomiales

1°) Exemple

On lance 5 fois un dé, et on souhaite savoir combien de fois on aura un 6 (ou, ce qui revient au même, on lance 5 dés et on compte les "6"). Si le dé n'est pas pipé, l'expérience revient à répéter 5 fois une expérience de Bernoulli avec $p = \frac{1}{6}$.

Exercice : Calculer la probabilité d'avoir au moins une fois un 6 en lançant 5 fois un dé.

2°) Définition

Le nombre de succès dans la répétition de n expériences de Bernoulli indépendantes entre elles mais ayant la même loi, est une variable aléatoire **binomiale** de paramètres n et p. C'est la somme de n variables aléatoires de Bernoulli de paramètre p.

3°) Simulation

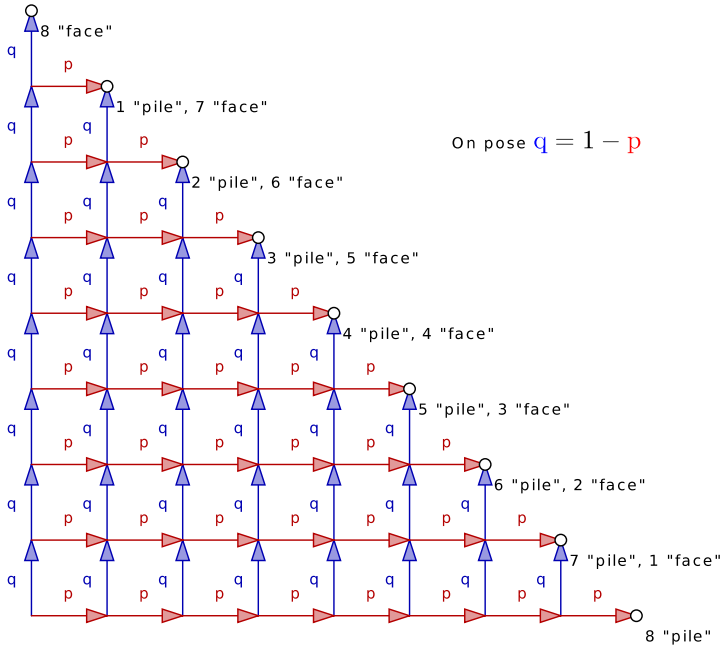
Pour simuler une variable aléatoire de paramètres 5 et $0,4$, on peut lancer 5 pièces qui ont toutes la même probabilité 0,4 de tomber sur « pile », puis compter le nombre de « pile » obtenus :

```
from random import *

pieces=[('pile' if random()<0.4 else 'face') for n in range(5)]
print(pieces.count('pile'))
```

4°) Loi d'une variable binomiale

a) Exemple

Si on lance 8 pièces, chacune ayant une probabilité p de tomber sur « pile », l'arbre des éventualités est plus petit que prévu, parce que 8 évènements donnent le même résultat « une seule pièce tombe sur pile » :



Chaque fois qu'on parcourt une flèche rouge, on multiplie par p et chaque fois qu'on parcourt une flèche bleue, on multiplie par $q = 1-p$. Donc par exemple, lorsqu'on arrive à « 5 "pile", 3 "face" », la probabilité obtenue est p^5q^3. Mais il reste à compter le nombre de chemins qui mènent à cette éventualité. Ce nombre est égal à la somme du nombre de chemins qui arrivent à « 5 "pile", 2 "face" » juste en-dessous (et qui arrivent par la flèche bleue) et du nombre de chemins qui arrivent à « 4 "pile", 3 "face" » en arrivant par la flèche rouge. Ce qui donne un algorithme pour calculer ces nombres :

Le nombre de manières d'arriver à « k "pile", $n-k$ "face" » est noté $\binom{n}{k}$ et se prononce « k parmi n ».

b) En Python

```
def Pascal(n,k):
        if n<0 or k<0:
                return float('nan')
        elif n==0 or k==0 or k==n:
                return 1
        else:
                return Pascal(n-1,k)+Pascal(n-1,k-1)
```

'nan' est une abréviation pour "not a number" qui signifie que $\binom{n}{k}$ n'existe pas si n ou k est négatif.

Le tableau des valeurs de $\binom{n}{k}$ est appelé **triangle de Pascal** . Pour l'imprimer avec *Python*, on peut l'imprimer ligne par ligne :

```
for n in range(10):
```

En hommage à Blaise Pascal qui l'a trouvé, mais ce tableau était déjà connu des chinois dès le 13e siècle.

```
        ligne=''
        for k in range(n+1):
                ligne+=str(Pascal(n,k))+' '
        print(ligne)
```

c) Loi binomiale

Si X est binomiale de paramètres n et p, la probabilité que $X = k$ est

$$P(X = k) = \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{n-k}$$

```
def proba_binomiale(n,p,k):
                return Pascal(n,k)*p**k*(1-p)**(n-k)
```

d) Binôme

On a

$$(p+q)^8 = p^8 + 8p^7q + 28p^6q^2 + 56p^5q^3 + 70p^4q^4 + 56p^3q^5 + 28p^2q^6 + 8pq^7 + q^8$$

Or les nombres 1, 8, 28 etc. sont justement les nombres $\left(\begin{array}{c} 8 \\ k \end{array} \right)$: On les appelle donc des coefficients binomiaux.

D'où le nom de loi binomiale.

e) Simulation

Le tableau théorique des probabilités binomiales peut se calculer avec les algorithmes précédents :

```
tt=[proba_binomiale(8,0.4,k) for k in range(9)]
print(tt)
```

Alors que la simulation donne ceci :

```
from random import *

def simul(n,p):
        pieces=[('pile' if random()<p else 'face') for i in range(n+1
        return(pieces.count('pile'))
tp=[simul(8,0.4) for i in range(1000)]
tp=[tp.count(k)/1000.0 for k in range(9)]
print(tp)
```

III/ Propriétés

1°) Espérance

L'espérance d'une variable aléatoire binomiale de paramètres n et p est $n \times p$.

Exemple : On lance 5 fois un dé, l'espérance du nombre de "6" est $5 \times \frac{1}{6} = \frac{5}{6}$.

2°) Écart-type

L'écart-type d'une variable aléatoire binomiale de paramètres n et p est $\sqrt{n \times p(1-p)}$.

Exemple : On lance 5 fois un dé, l'écart-type du nombre de "6" est $\sqrt{5 \times \frac{1}{6} \times \frac{5}{6}} = \frac{5}{6}$.

Chapitre 12

Équations cartésiennes

I/ Droite dans un repère orthonormé

1°) Définition

Une équation de deux inconnues x et y est appelée **équation cartésienne** d'une courbe $\mathcal{C}$ du plan si elle est équivalente à $M(x;y) \in \mathcal{C}$.

cartésienne parce que c'est René Descartes qui a eu l'idée...

2°) Vecteur normal

a) Remarque

Si $\vec{u}(a;b)$ est un vecteur directeur de la droite d et si $\vec{n}(-b;a)$ alors

$$\vec{u}.\vec{n} = a \times (-b) + b \times a = ab - ab = 0$$

b) Définition

Le vecteur $\vec{n}$ est dit **normal** à la droite d s'il est orthogonal à un vecteur directeur de d.

Dans ce cas, il est orthogonal à tous les vecteurs directeurs de d.

c) Méthode

```
class Droite:
        def __init__(self,A,B):
                self.A=A
                self.B=B
        def directeur(self):
                return self.A.vecteur(self.B)
```

```
        def normal(self):
                nx=-self.directeur().y
                ny=self.directeur().x
                return Vecteur(nx,ny)
```

3°) Équation cartésienne

Dans un repère orthonormé,

$$M(x,y) \in d \Leftrightarrow \overrightarrow{AM}.\vec{n} = 0$$

Alors si $ax + by = c$ est une équation cartésienne de d, le vecteur $\vec{n}(a,b)$ est normal à d.

```
        def __str__(self):
                a=self.normal().x
                b=self.normal().y
                c=a*self.A.x+b*self.A.y
                return str(a)+'x+('+str(b)+')y='+str(c)
```

II/ Cercles du plan

Dans un repère qui n'est pas orthonormé, on ne parle pas de cercles.

1°) Avec les distances

Si $C(x_C;y_C)$ est le centre du cercle $\mathcal{C}$ et R son rayon,

$$M(x;y) \in \mathcal{C} \Leftrightarrow CM = R$$

$$M(x;y) \in \mathcal{C} \Leftrightarrow CM^2 = R^2$$

$$M(x;y) \in \mathcal{C} \Leftrightarrow (x-x_C)^2 + (y-y_C)^2 = R^2$$

$$M(x;y) \in \mathcal{C} \Leftrightarrow x^2 + y^2 - 2xx_C - 2yy_C = R^2 - x_C^2 - y_C^2$$

2°) Avec les angles

Si $[AB]$ est un diamètre du cercle $\mathcal{C}$, alors

$$M(x;y) \in \mathcal{C} \Leftrightarrow \overrightarrow{MA}.\overrightarrow{MB} = 0$$

$$M(x;y) \in \mathcal{C} \Leftrightarrow (x-x_A)(x-x_B) + (y-y_A)(y-y_B) = 0$$

$$M(x;y) \in \mathcal{C} \Leftrightarrow x^2 + y^2 - (x_A+x_B)x - (y_A+y_B)y = -x_Ax_B - y_Ay_B$$

On retrouve l'équation précédente avec $2x_C = x_A + x_B$ et $2y_C = y_A + y_B$.

Chapitre 13

Échantillonnage

I/ Loi géométrique

1°) Exemple

Combien de fois doit-on lancer un dé au minimum, pour que la probabilité d'avoir au moins un "6" soit supérieure à 0,9 ?

Si on note A_n l'événement « le 6 est sorti au moins une fois en n lancers », le contraire de A_n a pour probabilité $\left(\frac{5}{6}\right)^n$.

Pourquoi ?

Or $P(A_n) \geqslant 0,9 \Leftrightarrow P(\overline{A_n}) \leqslant 0,1$; et comme la suite $P(\overline{A_n})$ est géométrique de raison inférieure à 1, elle tend vers 0 et l'inégalité $P(\overline{A_n}) \leqslant 0,1$ finit par arriver ; pour trouver quand, on peut écrire les termes successifs de cette suite :

```
probas=[(5.0/6)**n for n in range(20)]
petit=[probas[n]<0.1 for n in range(20)]
N=min([n for n in range(20) if petit[n]])

print(N)
```

2°) Loi géométrique

a) Exemple

En fait on ne lance le dé que tant qu'on perd, dès qu'on a un "6" on arrête. Le nombre de lancers de dés suit une loi géométrique de paramètre $\frac{1}{6}$. Le nombre de lancers de dés nécessaires pour commencer à jouer aux « petits chevaux » suit donc une loi géométrique de paramètre $\frac{1}{6}$.

géométrique, comme dans *suite géométrique*...

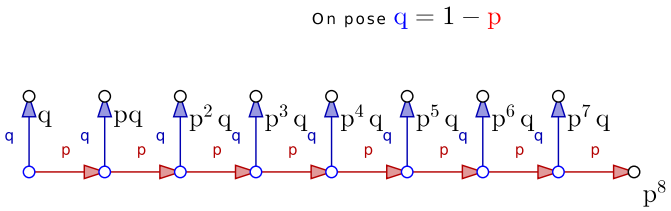
b) simulation

```
from random import *

de=0
n=0
while de!=6:
        de=randint(1,6)
        n+=1

print(n)
```

c) Suite géométrique

On fait comme avec la loi binomiale mais on arrête le dessin de l'arbre au premier succès :

$p = \frac{5}{6}$ et $q = \frac{1}{6}$, on a inversé les rôles du succès et de l'échec pour gagner de la place.



3°) Loi géométrique tronquée

a) Exemples

En fait on finit par arrêter le jeu par lassitude, ou simplement pour éviter qu'il ait une durée infinie. Par exemple, au **jeu du lièvre et de la tortue** :

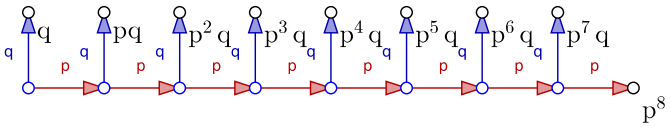
> Deux joueurs s'affrontent sur une piste à 5 cases. Ils sont tous les deux au départ, et lancent ensemble un dé. Si le dé tombe sur 6, l'un des joueurs appelé lièvre a gagné. Sinon l'autre joueur appelé tortue avance d'une case.

La tortue gagne donc si le dé ne tombe jamais sur 6 pendant la partie, donc 4 fois de suite. La probabilité de cet évènement est $\left(\frac{5}{6}\right)^4 \simeq 0,48$: Le lièvre a un peu plus de chances de gagner que la tortue.

La durée d'une partie de ce jeu suit une **loi géométrique tronquée** puisque si le 6 n'est toujours pas sorti au quatrième lancer, la partie est finie par victoire de la tortue.

Alors le nombre moyen de lancers nécessaire pour avoir un "6" est donné par l'espérance de la loi. Dans le cas de 8 lancers, on a ceci :

$$EX = q + 2pq + 3p^2\,q + 4p^3\,q + 5p^4\,q + 6p^5\,q + 7p^6\,q + 8p^7\,q$$



Exercice : Simplifier cette expression sachant que $p + q = 1$.

Écrire l'espérance sous la forme $qf(p)$, puis trouver une fonction $g(p)$ dont la dérivée est f.

II/ Échantillonnage

1°) Exemple

a) Énoncé

Dans une village de 1000 habitants, 440 ont l'intention de voter pour le maire sortant aux prochaines municipales, mais il ne le sait pas. Alors il va commanditer un sondage sur un échantillon de 25 personnes. 13 personnes parmi les 25 disent vouloir voter pour lui.

b) Simulation du sondage

On va simuler 1000 échantillons de 25 personnes chacun, et compter combien d'entre eux donnent l'impression que le maire sera réélu :

```
population=['contre' for n in range(560)]
population=population+['pour' for n in range(440)]
print(len(population))
print(sample(population,25))
```

Et pour compter les échantillons favorables au maire parmi les 1000 :

```
def favorables(ech):
        return ['vote' for vote in ech if vote=='pour']

def youpi():
        return len(favorables(sample(population,25)))>12
```

```
p=len([n for n in range(1000) if youpi()])
print(p/1000)
```

On constate qu'environ le quart des échantillons de 25 sont favorables au maire, ce qui suggère que le premier quartile de cette série est environ 12.

c) Théorie

En considérant les échantillons de 25 personnes comme aléatoires, le nombre de partisans du maire dans un de ces échantillons est binomial de paramètres 25 et 0,44. Le tableau des fréquences cumulées théoriques est obtenu avec

```
tt=[proba_binomiale(25,0.44,k) for k in range(26)]
fc=[sum(tt[0:n]) for n in range(26)]
print(fc)
```

On voit que le premier quartile est 10, mais surtout que la fréquence cumulée dépasse 0,025 au rang 7 et 0,975 avant le rang 18 : La probabilité que le pourcentage d'électeurs favorables au maire dans un échantillon de 25 soit entre 7 et 18 est de plus de 0,95. On exprime ce résultat en disant que $\left[\frac{7}{25};\frac{18}{25}\right]$ **est un intervalle de fluctuation à 95 %** pour le pourcentage d'électeurs favorables au maire. Ici on trouve $[0,28;0,72]$.

d) Rappel

En Seconde, on a vu qu'un tel intervalle était donné par $\left[0,44-\frac{1}{\sqrt{25}};0,44-\frac{1}{\sqrt{25}}\right]$, soit $[0,24;0,6]$. Ce n'est pas le même résultat, mais on démontre que pour de grandes valeurs de N (par exemple $N \geqslant 100$), les deux intervalles coïncident.