Algorithmique et programmation

Mathématiques / Seconde

Introduction

Ce cours présente toutes les notions nécessaires à la programmation (en langage Python) ou la compréhension des algorithmes mentionnés dans les programmes de *mathématiques* de la classe de seconde générale.

Les parties I à IV 2. peuvent être utiles dès le début de la classe de seconde.

Bien qu'aucune connaissance sur les listes ne soit exigible en classe de seconde, celles-ci sont néanmoins nécessaires pour les algorithmes de la partie « statistiques et probabilités » et elles deviennent exigibles à partir de la classe de première. Les graphiques ne sont pas mentionnés explicitement dans les programmes, mais le document « ressources » de seconde en fait un usage abondant et il nous a semblé profitable d'en présenter les bases.

Table des matières

I Variables et affectations

1. Algorithmes

Le mot français « algorithme » provient de Al-Khwarizmi (780 env. - 850 env.), savant perse. Les algorithmes ne sont pas nés avec l'informatique. Par exemple, l'algorithme d'Euclide (recherche du PGCD de deux entiers) est vieux de plus de 2 000 ans ! On trouve également des descriptions précises d'algorithmes dans la Chine ancienne.

Définition. Un algorithme est la séquence détaillée de toutes les opérations ou instructions à effectuer pour résoudre un problème (ou simplement fournir un résultat).

Exercice 1. On considère l'algorithme ci-dessous. Compléter à chaque étape le contenu de la variable a (2^{e} colonne) quand l'utilisateur choisit 4 comme nombre.

Instructions	État de la variable a
choisir une valeur pour a	4
a prend la valeur $a+1$	
a prend la valeur $a(a-2)$	
a prend la valeur $a+1$	

Définition. En algorithmique, une *variable* :
- est repérée par une étiquette qui représente son *nom*,
- contient une *valeur* : nombre, mot, liste de nombres, de mots, etc.

L'affectation consiste à donner (ou modifier) une valeur à une variable.

Remarque. L'instruction « b prend la valeur $a+2$ » (ou « affecter $a+2$ à b ») est généralement notée $b \leftarrow a+2$ en algorithmique (on parle de *pseudo-code*). Cette instruction :
- déclenche le calcul de « *la valeur de* a » $+2$;
- puis affecte le résultat à la variable b.

Exercice 2. Compléter les tableaux d'états des variables.

Instructions	a	b
$a \leftarrow 2$		
$b \leftarrow 5$		
$a \leftarrow b$		

Instructions	a	b
$a \leftarrow 2$		
$b \leftarrow 5$		
$b \leftarrow a$		

2. Programmation en Python

L'intérêt d'un algorithme est d'être codé dans un langage informatique afin qu'une machine (ordinateur, calculatrice, etc.) puisse l'exécuter rapidement et efficacement. Le langage *Python*[1] est un des langages aujourd'hui les plus utilisés (après Java et la famille des langages C), en particulier pour le calcul scientifique.

Pour programmer sur un ordinateur, il faut installer le langage proprement dit (on appelle cela une *distribution* Python) et un logiciel appelé EDI (environnement de développement intégré) ou simplement *éditeur* comprenant :

1. Nous utiliserons Python 3 qui, sur certains points, se distingue de Python 2, toujours utilisé.

- un interpréteur de commandes ou *console* (*shell* en anglais) ; l'utilisateur « a la main » pour saisir une commande lorsque l'invite de commande (*prompt* en anglais) apparaît : `>>>`
- un éditeur de programmes dans lequel on tape le programme à exécuter (on dit aussi *script*) qui n'est autre qu'une succession de commandes ;
- des outils et fonctionnalités avancées.

3. Opérations et affectations

- Les 4 opérations de base s'écrivent respectivement : `+ - * /`
- La puissance s'écrit `**` (contrairement à la plupart des langages qui utilisent `^`)
- Le reste de la division euclidienne s'écrit `%` et le quotient `//`
- On peut taper plusieurs instructions sur la même ligne en les séparant par `;`

Exemple. Dans la console taper :
`>>> 1 + 3*2**2` ➢ ..... (Python reconnaît la priorité des opérations)
`>>> 47//10 ; 47%10` ➢ .....

- L'affectation $a \leftarrow a + 4$ s'écrit en Python `a = a + 4`
- Python permet également de faire des affectations *multiples* ou *parallèles* en une seule instruction :
  - affectation multiple : `a = b = 0`
  - affectation parallèle : `a, b = 1.2, -4`

Remarque.
- Attention, le signe d'égalité n'a pas le même sens qu'en mathématiques !
- Les noms de variables peuvent être des mots entiers (sans espace) comprenant éventuellement le caractère de soulignement, par exemple `somme_valeurs = 5`, mais jamais de `-` interprété comme une soustraction.
- L'explorateur de variables appelé aussi espace de noms ou espace de travail (*workspace*) permet d'observer la valeur courante des variables après chaque commande.

Exercice 3. Compléter les tableaux d'états des variables.

Instructions	a
`a = 3`	
`a = 5*a**2 - 1`	
`a = a%2`	

Instructions	a	b
`a, b = 5, 2`		
`b = a + b`		
`a = a + 1`		

Exercice 4. Pour échanger le contenu de deux variables `a` et `b`, des élèves proposent les deux solutions suivantes.

Proposition 1	a	b
`a, b =`		
`a = b`		
`b = a`		

Proposition 2	a	b	c
`a, b =`			
`c = a`			
`a = b`			
`b = c`			

1) En complétant les tableaux d'état, dire si les propositions ci-dessus sont valides :

   .....

2) Comment peut-on réaliser cet échange plus simplement en langage Python ?

   .....

4. Types de variables

Définition. Le codage informatique d'une variable (dans la machine) dépend de son *type*. Parmi les types de base, mentionnons :

- les *entiers* (type `int`, *integer* en anglais),
- les « nombres à virgule flottante » appelés simplement *flottants* (type `float`) : par exemple `c = 2.5` ;
- les *booléens* (type `bool`) ne peuvent prendre que 2 valeurs : `True` ou `False` ;
- les *chaînes de caractères* (type `str`, chaîne = *string* en anglais) sont encadrées par des guillemets ou des apostrophes : par exemple `a = "bonjour"` ou `a = 'bonjour'` ;
- les *listes* (type `list`), entre crochets : par exemple `L = [-1, 3.2, 1/3, 1e-3]`.

Remarque.

- Le type d'une variable est visible dans l'espace de travail ou avec la fonction `type`.
- Dans Python, le type est défini au moment de l'affectation et peut-être changé par une nouvelle affectation, contrairement à de nombreux langages de programmation où le type des variables doit être *déclaré* au début du programme.
- Les fonctions `int`, `float`, `str`, `list` permettent de réaliser des conversions de type : par exemple `>>> int(-3.85)` ➢ `-3` (troncature).

5. Entrées et sorties, commentaires

Définition. Dans un algorithme ou un programme, on appelle *entrées* les saisies que fait l'utilisateur pour communiquer avec le programme et *sorties* les affichages que le programme réalise pour donner des résultats.

En Python, les *sorties* (texte) s'obtiennent avec la fonction `print`. Les valeurs à afficher sont séparées par des virgules. L'affichage est réalisé dans la console.

Exemple. `print("la longueur est", x, "cm")`

Les *entrées* s'obtiennent par la fonction `input` *et une affectation* :
`texte = input("message")`

- `"message"` est affiché dans la console,
- le programme attend une saisie de l'utilisateur terminée par « Entrée » ;
- cette saisie est alors stockée dans la variable appelée ici `texte`.

Remarque. `input` renvoie toujours du texte (type `str`), même si on saisit un nombre. Pour qu'une saisie de l'utilisateur soit convertie en nombre (type `int` ou `float`) avec lequel on pourra faire des calculs, on utilise la fonction `eval` (ou éventuellement `int` ou `float`).

```
nombre = eval(input("message"))
```

- `eval` évalue la chaîne de caractères fournie par `input` en effectuant si besoin les calculs nécessaires (1/3 par exemple) et en convertissant le résultat dans le type approprié ;
- attention à la double parenthèse `))` !
- avec `eval`, on peut aussi utiliser une affectation parallèle pour ne pas répéter plusieurs instructions d'entrées successives.

Exemple. `a, b = eval(input("saisir a, b : "))`
On peut ici saisir deux entrées de types quelconques mais séparées par une virgule.

Définition. Dans un programme ou un algorithme, un *commentaire* est du texte qui n'est pas destiné à être exécuté, mais qui sert à expliquer le code.

- Sur une ligne de programme Python, tout texte qui suit le caractère `#` est un *commentaire* et sera ignoré lors de l'exécution.
- On peut aussi insérer des commentaires multi-lignes entre des triples guillemets ou triples apostrophes : `""" ... """` ou `''' ... '''`.
- En langage algorithmique (pseudo-code), les commentaires sont souvent signalés par `//`.

II Structure alternative

1. Si alors (sinon)

La structure alternative (appelée aussi structure conditionnelle) fait partie des *structures de contrôle* qui permettent de faire varier l'ordre dans lequel les instructions d'un programme sont effectuées. On donne ci-dessous une écriture algorithmique en pseudo-code et sa traduction en langage Python.

si condition **alors**
| instructions
sinon
| instructions
fin si

```
if condition:
    instructions
else:
    instructions
```

Lorsque la condition est vraie, on effectue le premier bloc d'instructions (après *alors*) et si elle est fausse, on effectue le second (après *sinon*).

Remarque.

- Le *sinon* (`else`) et le second bloc d'instructions sont facultatifs.
- En Python, la condition et le `else` doivent être suivis d'un *double-points*.
- Les instructions sont *indentées* c'est-à-dire décalées à droite par rapport au début de la ligne (par défaut de 4 espaces).
- Il n'y a pas de « *fin si* » en Python (ni de « *alors* »), c'est le double-points et l'indentation qui indiquent la structure : pour sortir de la structure on revient en début de ligne.
- Lorsqu'il n'y a qu'une seule ligne d'instructions, celle-ci peut être placée sur la même ligne après le double-points.

Exercice 5. On considère l'algorithme suivant et sa traduction en Python :

si $n > 2$ **alors**
| $m \leftarrow 2n$
sinon
| $m \leftarrow 3n + 1$
fin si

```
if n > 2:
    m = 2*n
else:
    m = 3*n + 1
```

Compléter le tableau des valeurs obtenues pour m en fonction des valeurs de n.

n	5	2	0	-1	3
m					

Remarque. Pour éviter d'imbriquer plusieurs structures `if` l'une dans l'autre, en créant, à chaque fois, un niveau d'indentation supplémentaire, on peut utiliser la clause `elif` (contraction de *else if*) ; on peut mettre autant de `elif` que l'on veut.

si condition1 **alors**
| instructions
sinon si condition2 **alors**
| instructions
sinon
| instructions
fin si

```
if condition1:
    instructions
elif condition2:
    instructions
else:
    instructions
```

2. Les tests

Syntaxe Python des tests simples utilisables pour l'écriture d'une « condition » :

mathématiques	$=$	$\neq$	$<$	$\leqslant$	$>$	$\geqslant$
Python	`==`	`!=`	`<`	`<=`	`>`	`>=`

Remarque.

- Le test d'égalité s'écrit `==` et non `=` pour ne pas être confondu avec l'affectation !
- On peut créer des tests combinés avec les mots-clés `and`, `or` ou `not` ainsi qu'en enchaînant des inégalités ; on peut y compris tester le type, par exemple `type(a) == int` ;
- Un test *renvoie* un booléen `True` ou `False` (que l'on peut utiliser dans une affectation).

Exemple.

si $1 \leqslant x \leqslant 3$ et $x \neq 2$ **alors**
| $y \leftarrow x + 1$
sinon
| $y \leftarrow x - 1$
fin si

```
if (1 <= x <= 3) and (x != 2):
    y = x + 1
else:
    y = x - 1
```

Exercice 6.

1) Compléter ce programme Python demandant à l'utilisateur deux nombres et affichant en sortie le plus grand des deux.

```
a, b = eval(input("a, b = "))
if ............
    print(a)
else:
    ........
```

2) Compléter ce programme Python affichant si un entier saisi par l'utilisateur est pair ou impair.

```
a = eval(input("a ? "))
if .........
    print(a, "est pair")
else:
    ......................
```

III Les boucles

1. Généralités

Définition. Dans un algorithme, lorsque l'on doit répéter un certain nombre de fois un bloc d'instructions, on dit que l'on réalise une *boucle* ou *structure itérative*. Il y a deux types de structures itératives selon que le nombre de répétitions dépend d'une condition (*boucle conditionnelle*) ou non (*boucle inconditionnelle* : le nombre de répétitions est fixé *a priori*).

Remarque. Les deux types de boucles forment, avec la structure alternative vue précédemment, les trois *structures de contrôle* fondamentales en algorithmique.

2. Boucle while (tant que)

On donne ci-dessous une écriture algorithmique d'une boucle conditionnelle et sa traduction en langage Python.

tant que condition **faire**
| instructions
fin tant que

```
while condition:
    instructions
```

Le bloc d'instructions est exécuté tant que la condition est vraie.

Remarque. La syntaxe Python suit les mêmes règles que celle de la structure alternative.

- La condition doit être suivie d'un *double-points*.
- Les instructions sont *indentées* c'est-à-dire décalées à droite par rapport au début de la ligne (par défaut de 4 espaces) ou placées après le double-points si une seule ligne.
- Il n'y a pas de « *fin tant que* » (ni de « *faire* »), c'est le double-points et l'indentation qui indiquent la structure : pour sortir de la structure on revient en début de ligne.
- La condition peut s'écrire avec les opérateurs de test vus précédemment.

Exercice 7. On considère l'algorithme ci-dessous où n est un entier naturel.

tant que $n \geqslant 5$ **faire**
| $n \leftarrow n - 5$
fin tant que

1) Quelle est la *condition* qui doit être réalisée pour que l'instruction de la boucle *tant que* soit exécutée ? .....
2) Quel est le contraire de cette condition ? .....
3) Peut-on obtenir 17 comme valeur pour la variable n après l'exécution de la boucle ?
   Justifier. .....

4) La valeur de n à l'entrée de la boucle étant 23, compléter le tableau suivant et en déduire la valeur de n à la sortie de la boucle : $n =$ .....

Étape	n	$n \geqslant 5$?
init	23	Vrai
1	18	Vrai
2		
3		
4		

5) Quel est le rôle de cet algorithme ?

.....

6) Quelles instructions faut-il ajouter à cet algorithme pour qu'il fournisse également le *quotient* de la division euclidienne de n par 5 ?

.....

Définition. Une variable initialisée à 0 et qui augmente de 1 à chaque tour de boucle (on dit aussi *incrémentée*) s'appelle un *compteur* de boucle.

Exercice 8. L'algorithme ci-dessous détermine le plus petit entier naturel n tel que la somme des entiers de 0 à n soit strictement supérieure à 1 000.

```
n ← 0
somme ← 0
tant que somme ⩽ 1000 faire
    n ← n + 1
    somme ← somme + n
fin tant que
```

1) Traduire cet algorithme en langage Python.
2) Le programmer et donner la réponse cherchée : $n =$ .....
3) Quel est le rôle de la variable n ? .....

3. Boucle for (pour)

On donne ci-dessous une écriture algorithmique d'une boucle *pour* et sa traduction en langage Python.

```
pour i de 1 à n faire
    instructions
fin pour
```

```
for i in range(1, n+1):
    instructions
```

Le bloc d'instructions est répété n fois, la variable compteur i étant *automatiquement* incrémentée (de 1 à n) à chaque tour de boucle.

Remarque.

- La syntaxe Python suit les mêmes règles que celles des autres structures : double-points, indentation.

- `range` (*range* = plage ou intervalle en anglais) peut s'utiliser avec un, deux ou trois arguments qui doivent obligatoirement être entiers (éventuellement négatifs) :
  - `range(n)` : de 0 à `n-1` (le 1er argument peut-être omis s'il vaut 0),
  - `range(m, n)` : `m` inclus → `n` exclu,
  - `range(m, n, p)` : `m` inclus → `n` exclu, `p` = pas (le pas vaut 1 s'il est omis).
- Si la valeur du compteur `i` n'est pas utilisée explicitement dans la boucle, on peut utiliser `range(n)` plutôt que `range(1, n+1)` pour répéter `n` fois.

Exercice 9. L'algorithme suivant calcule la somme $S = 1 + 2 + 3 + \ldots + n$.

```
S ← 0
pour i de 1 à n faire
|  S ← S + i
fin pour
```

1) Compléter le tableau suivant qui permet de détailler le calcul de S pour $n = 10$.

État de la variable i		1	2	3	4	5	6	7	8	9	10
État de la variable S	0										

2) Traduire cet algorithme en Python et le tester pour vérifier le résultat précédent.

IV Les fonctions

1. Définition de fonction

Définition. Une fonction est une portion de code, effectuant une tâche spécifique, qui a reçu un *nom* et que l'on peut *appeler* dans un programme ou dans la console.

- Elle *renvoie* un unique résultat (qui peut être composé de plusieurs valeurs) ou aucun et son fonctionnement dépend d'un certain nombre de *paramètres*.
- Les valeurs ou expressions que l'on donne aux paramètres au moment de l'appel de la fonction s'appellent des *arguments*.
- La première ligne, spécifiant nom et paramètres, s'appelle *l'en-tête* de la fonction, les autres lignes, contenant les instructions, forment le *corps* de la fonction.

On donne ci-dessous une écriture algorithmique d'une fonction à deux paramètres et sa traduction en langage Python.

```
fonction ma_fonction(x, y)
|  instructions
|  renvoyer valeur
fin fonction
```

```
def ma_fonction(x, y):
    instructions
    return valeur
```

Remarque.

- La syntaxe Python suit les mêmes règles que celles des structures de contrôle : double-points, indentation.
- Il y a une analogie avec les fonctions en mathématiques mais ces notions ne se recouvrent pas exactement.

- Une fonction peut n'avoir aucun paramètre ou ne renvoyer aucune valeur.
- Les arguments peuvent être des objets de n'importe quel type y compris des fonctions, de même pour la « valeur » renvoyée.
- Une fonction peut également avoir une action (appelée *effet de bord*), par exemple réaliser un affichage (`print`).
- Lorsque l'on exécute un script contenant une définition de fonction, celle-ci apparaît dans l'espace de travail et peut alors être appelée dans la console où dans un programme. À chaque modification de la fonction, il faut re-exécuter le script qui la contient, pour prendre en compte les modifications.

Exemple. Fonction appelée dans la console.

```
def volume_pyramide(côté, hauteur): # double-points obligatoire ici
    aire = côté**2 # la variable aire est "locale" à la fonction
    return 1/3*aire*hauteur
```

Après avoir exécuté le script ci-dessus, on peut, dans la console, faire l'appel :
`>>> volume_pyramide(2, 3)` ➢ `4.0` (dans ce cas 2 et 3 sont les arguments de la fonction).

Exemple. Fonction appelée dans un programme : tableau de valeurs de la fonction f, définie par $f(x) = 3x + 1$, pour x variant de -5 à 5 avec un pas de 2.

```
# définition de la fonction f
def f(x):
    return 3*x + 1
# début du programme principal
print("x : f(x)") # légende
for x in range(-5, 6, 2):
    print(x, ":", f(x))
```

Exercice 10. On considère la fonction ci-dessous.

```
def affine(a, b, x):
    return a*x + b
```

1) Quels sont les paramètres de la fonction `affine` ? .....
2) À l'appel `affine(-1, 2, 4)`, quelle est la valeur renvoyée ? .....
3) Quels arguments donner pour calculer l'image de 5 par la fonction $x \mapsto -2x + 7$?

   .....

2. Fonctions mathématiques prédéfinies

Quelques (rares) fonctions mathématiques sont automatiquement chargées au démarrage de Python :

- `max(a, b, c)` → le maximum des arguments,
- `min(a, b)` → le minimum des arguments,
- `abs(a)` → la valeur absolue de `a` (= `max(a, -a)`),
- `round(x, n)` → arrondit `x` avec `n` chiffres après la virgule.

La plupart des fonctions (et constantes) mathématiques usuelles se trouvent dans le *module* `math` (fichier contenant le code de ces fonctions), mais ne sont pas chargées au démarrage : `sqrt` (*square root* = racine carrée), `sin, cos, tan, exp, log, floor, gcd, pi, e, ...`

Le module doit être chargé en mémoire pour que ses fonctions soient accessibles. Pour cela on utilise l'instruction `import` avec plusieurs syntaxes possibles :

- `import math` : importe le module « globalement », mais les fonctions qu'il contient doivent alors être *préfixées* : `math.sqrt(2)` ;
- `from math import sin, cos` : on n'importe que les fonctions spécifiées, sans préfixe ;
- `from math import sqrt as racine` : on utilise l'*alias* `racine` à la place de `sqrt`.

3. Nombres aléatoires

Cette partie ainsi que les suivantes ne sont pas indispensables au début de la classe de seconde ; les notions présentées peuvent être abordées pour les chapitres de statistiques et probabilités.

Le module `random` (non chargé au démarrage) contient des fonctions nécessaires pour générer des nombres aléatoires, en particulier :

- `random()` → nombre aléatoire (flottant) dans $[0, 1[$,
- `uniform(a, b)` → nombre aléatoire (flottant) dans $[a, b]$,
- `randint(a, b)` → entier aléatoire compris entre `a` et `b` (inclus).

Exemple. Programme simulant `n` lancers de 2 dés et calculant la fréquence du résultat `r`.

```
from random import randint
def simul2d(n, r):
    nbr = 0
    for i in range(n):
        dé1 = randint(1, 6)
        dé2 = randint(1, 6)
        somme = dé1 + dé2
        if somme == r:
            nbr = nbr + 1
    return nbr/n
```

En simulant avec `n=1000` et pour différentes valeurs de `r`, on observe que le résultat le plus fréquent est `r` = ...

V Listes et chaînes de caractères

1. Listes

Définition. Une liste est une structure de données qui peut contenir un nombre arbitraire de valeurs repérées par un *indice* de position.

Exemple. Python : `L = [-1, 3.2, 1/3, 1e-3]`

En Python, une liste est de *type* `list` et les valeurs peuvent être de type quelconque : entiers, flottants, chaînes, listes, ...

- on accède à un élément de la liste par son indice qui *commence à 0* : `L[1]` ➢ `3.2` ;
- un élément d'une liste est modifiable s'il existe déjà : dans l'exemple ci-dessus, on peut faire `L[2] = 0` mais pas `L[4] = 0` ;
- fonctions de liste : `len` (longueur = *length* en anglais), `max`, `min`, `sum` ;
- supprimer un élément d'une liste : `del L[i]` ;
- ajouter un élément à une liste : `L.append(2)` (*append* = joindre en anglais)
  → modifie la liste mais ne renvoie rien (ne pas utiliser d'affectation !)
- ajouter une liste à une liste : `L.extend([5, -1])` ;
- renvoyer le nombre de `x` : `n = L.count(x)` ;
- multiplication de liste : `L = [0]*5` équivaut à `L = [0, 0, 0, 0, 0]` ;
- liste vide : `L = [ ]`.

Remarque. `append`, `extend` et `count` s'appellent des *méthodes* = fonctions attachées à un objet. Pour les listes, il en existe d'autres : `sort` (trier), `insert`, `reverse`, `index`, ...
`>>> list.index?` ➢ aide.

Python fournit également d'autres syntaxes très utiles avec les listes :

- `if` permet de tester si un élément est dans une liste : `if x in liste` ;
- `for` permet de parcourir tous les éléments d'une liste : `for x in liste` ;
- on peut générer des listes dites *en compréhension* (voir ci-dessous).

Exemple.

- `Xlist = [k/10 for k in range(-50, 51)]` ➢ de -5 à 5 avec un pas de 0,1 ;
- `Lsup = [x for x in liste if x > 0]` ➢ tous les éléments positifs de `liste`.

Exercice 11. On considère le programme ci-dessous.

```
from random import randint
def freq2d(n):
    L = [0]*13 # initialisation de la liste
    for i in range(n):
        r = randint(1, 6) + randint(1, 6)
        L[r] = L[r] + 1
    L = [x/n for x in L]
    return L
```

Que fait la fonction définie dans ce programme ? .....

.....

Exercice 12.

1) Dans la console, générer la liste de tous les carrés de 1^2 à 20^2.

   `>>>` .....

2) Que calcule cette commande : `sum([n[i]*x[i] for i in range(len(x))])/sum(n)` ?

.....

3) Expliquer à quoi correspondent les 3 valeurs `m, s, p` renvoyées par la fonction ci-dessous.

```
def stat(x, n):
    m = moyenne(x, n) # fonction définie ailleurs (cf. q2)
    carré_écarts = [(xi - m)**2 for xi in x]
    V = moyenne(carré_écarts, n)
    s = sqrt(V)
    binf, bsup = m-2*s, m+2*s
    p = 0
    for i in range(len(x)):
        if binf <= x[i] <= bsup:
            p = p + n[i]
    p = p/sum(n)
    return m, s, p
```

.....

.....

.....

2. Chaînes de caractères

On peut *concaténer* (assembler) des chaînes de caractères avec l'opération `+`.

Exemple. `>>> "bonjour" + ' tout le monde'` ➢ `'bonjour tout le monde'`

- Dans une chaîne de caractères, chaque lettre est référencée par un *indice* de position qui *commence à 0* : `texte[i]` renvoie la `i+1`-ième lettre de la chaîne.
- Les lettres ne sont pas modifiables individuellement.
- `len(texte)` renvoie la longueur (*length* en anglais) de la chaîne `texte`.
- `if mot in phrase:` → teste si la chaîne `mot` se trouve dans la chaîne `phrase`.
- `for c in texte:` → `c` parcourt tous les éléments (lettres) de `texte`.

Exercice 13. On considère les deux fonctions Python ci-dessous.

```
def rev(texte) :
    newtext = "" # chaine vide
    for c in texte:
        newtext = c + newtext
    return newtext
```

```
def delvow(texte):
    newtext = ""
    for c in texte:
        if c not in "aeiouy":
            newtext = newtext + c
    return newtext
```

Quel est le rôle de ces deux fonctions ?

`rev(texte)` : .....

`delvow(texte)` : .....

Remarque. Pour manipuler les chaînes de caractères, on dispose de nombreuses *méthodes* (= fonctions attachées à un objet) : `count`, `find`, `replace`, `split`, `join`, `format`, ...

VI Graphiques

1. La bibliothèque graphique matplotlib

La bibliothèque graphique `matplotlib` est une *extension*, c'est-à-dire un *paquet* de modules optionnel (non installé par défaut avec Python), disponible au téléchargement (en anglais graphique = *plot* et bibliothèque = *library*).

Celle-ci comprend le module `pyplot` qui contient les fonctions graphiques essentielles. Ce module doit être importé « globalement », mais, pour éviter un préfixage trop long, on utilise un alias, traditionnellement `plt` :

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

2. Courbes

Pour tracer une courbe, il faut générer au préalable deux listes de même taille : une pour les x et une pour les y. On peut générer la liste des x *en compréhension*, comme vu précédemment, ou bien, plus simplement, utiliser l'une des deux commandes ci-dessous :

```
Xlist = plt.np.linspace(a, b, n) # n = nb de points
Xlist = plt.np.arange(a, b, pas) # ici b est exclu
```

Remarque. `arange` ressemble à `range` mais permet de choisir un pas non entier.

On génère ensuite la liste des $y = f(x)$ avec une liste en compréhension, puis on trace la courbe avec la commande `plot` (après avoir importé `matplotlib.pyplot as plt`) :

```
Ylist = [f(x) for x in Xlist]
plt.plot(Xlist, Ylist, options)
plt.grid() # affiche la grille, facultatif
plt.show()
```

Remarque.

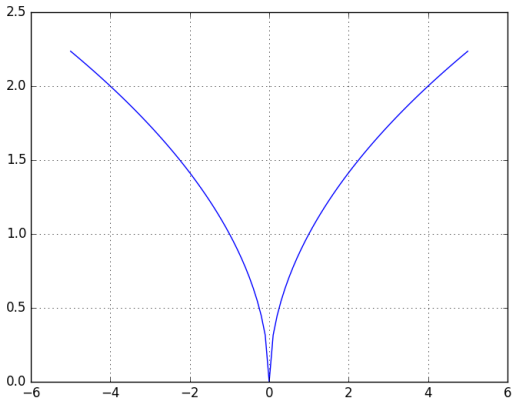
- Dans la console, on accède à une aide interactive, utile pour le détail des options, avec : `>>> plt.plot?`
- C'est la fonction `show` qui affiche le graphique, elle est indispensable !
- La fenêtre graphique propose des boutons permettant de sauver la figure, de zoomer, etc. ; la coche verte permet de régler *a posteriori* différentes options ;
- Il faut fermer la fenêtre graphique pour relancer un nouveau graphique.

Exemple. Voici un programme traçant la courbe de la fonction f telle que

$$f(x) = \begin{cases} \sqrt{x} & \text{si } x \geqslant 0 \\ \sqrt{-x} & \text{si } x < 0 \end{cases}.$$

```
from math import sqrt
import matplotlib.pyplot as plt
def f(x):
    if x >= 0:
        return sqrt(x)
    else:
        return sqrt(-x)

Xlist = plt.np.linspace(-5, 5, 101)
# avec 100 pts -> pb pour le 0
Ylist = [f(x) for x in Xlist]
plt.plot(Xlist, Ylist)
plt.grid()
plt.show()
```



Remarque. On peut prérégler la fenêtre graphique dans le programme avec les instructions :

```
plt.xlim(xmin, xmax)
plt.ylim(ymin, ymax)
```

3. Diagrammes à barres

Après avoir importé `matplotlib.pyplot as plt`, on trace un diagramme à barre avec :

```
plt.bar(Xlist, Ylist, options)
plt.show()
```

Remarque.

- Dans la console, on accède à une aide interactive, utile pour le détail des options, avec : `>>> plt.bar?`
- Parmi les options, mentionnons `tick_label=liste` qui permet de placer une `liste` d'étiquettes (textuelles) en abscisse des barres.

Exercice 14.

1) Programmer la fonction `freq2d` de l'exercice 11 (page 12) puis faire afficher le diagramme à barres correspondant aux fréquences obtenues pour 1 000 lancers de deux dés.
Avec `plt.bar(range(13), freq2d(1000))` on doit obtenir un graphique ressemblant au premier graphique ci-dessous.
2) Compléter le deuxième tableau ci-dessous pour établir la loi de probabilité de l'expérience aléatoire correspondant au lancer de deux dés.

+	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

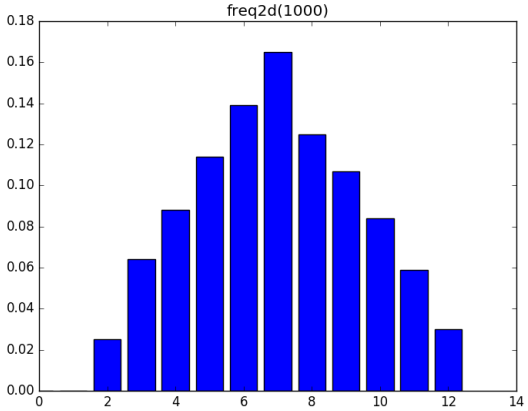
issue	2	3	4	5	6	7
proba						

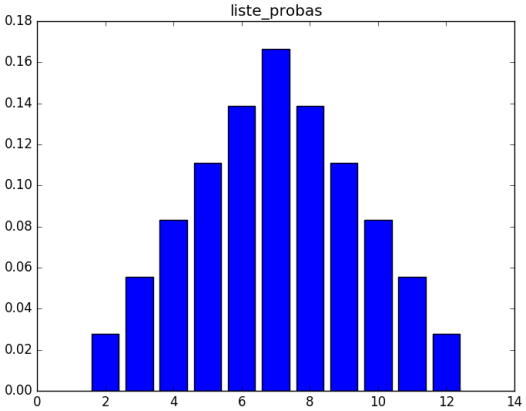
issue	8	9	10	11	12	
proba						

Générer une liste `liste_probas` avec les probabilités obtenues :

```
>>> liste_probas = ....................................................
>>> liste_probas.extend( ..............................................
```

3) Faire afficher le diagramme à barres correspondant (2e graphique ci-dessous) :
```
>>> plt.bar(range(2, 13), liste_probas) ; plt.show()
```





4. Nuages de points

Après avoir importé `matplotlib.pyplot as plt`, on trace un nuage de points avec :

```
plt.scatter(Xlist, Ylist, options)
plt.show()
```

Remarque. `>>> plt.scatter?` donne le détail des options (*scatter* = disperser en anglais).

Exercice 15.

1) Que fait la fonction `echantillon(n, p)` ci-dessous ? À quoi correspond `p` ?

```
from random import random
def echantillon(n, p):
    nbsucces = 0
    for i in range(n):
        if random() <= p:
            nbsucces = nbsucces + 1
    return nbsucces/n
```

.....

.....

.....

2) On souhaite simuler N échantillons de taille n d'une expérience aléatoire à deux issues et tracer le nuage de points correspondant. Si p la probabilité d'une issue et f sa fréquence observée dans l'échantillon, on veut également calculer la proportion des cas où l'écart entre p et f est inférieur à $\frac{1}{\sqrt{n}}$.

Compléter le programme Python ci-dessous (lignes 7 et 9).

```
1  from math import sqrt
2  import matplotlib.pyplot as plt
3
4  def simul(N, n, p):
5      compteur = 0
6      liste_freq = []
7      for
8          f = echantillon(n, p)
9          if
10             compteur = compteur + 1
11         liste_freq.append(f)
12     plt.close() # ferme le graphique précédent si resté ouvert
13     plt.scatter(range(N), liste_freq)
14     plt.grid()
15     plt.xlim(0, N) # fenêtre graphique : Xmin = 0, Xmax = N
16     plt.ylim(0, 1) # fixe Ymin = 0, Ymax = 1
17     plt.plot([0, N], [p, p], '--') # pointillés à y = p
18     plt.show()
19     return compteur/N
```

➢ Pour N = 50 échantillons avec $p = 0{,}5$ (correspondant à la simulation du pile ou face), on doit obtenir des graphiques similaires à ceux ci-dessous et une proportion d'échantillons tels que $|p - f| < \frac{1}{\sqrt{n}}$ de l'ordre de 0,95.