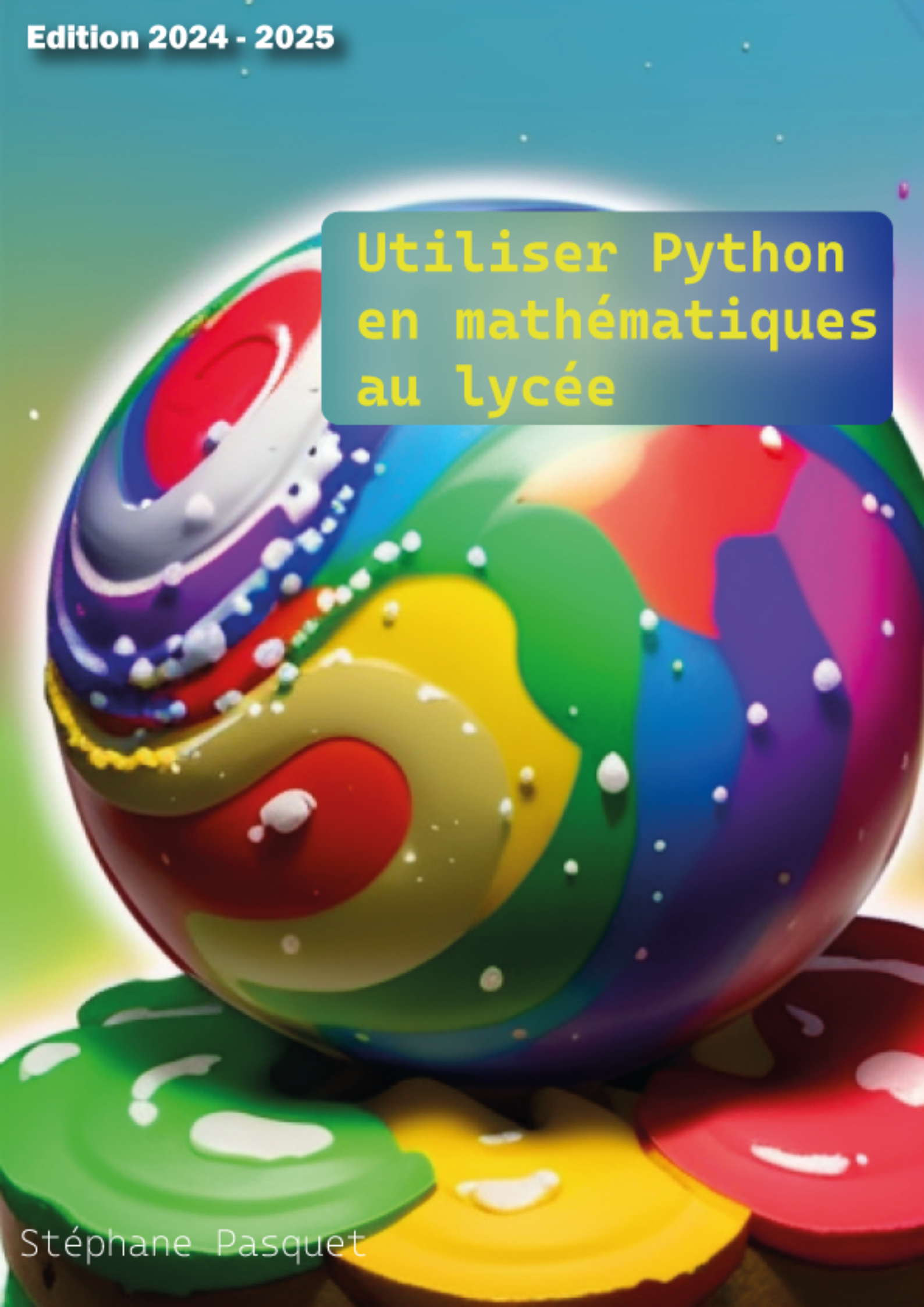
Edition 2024 - 2025

Utiliser Python en mathématiques au lycée



Stéphane Pasquet

SOMMAIRE

AVANT-PROPROS iii
AUTRES OUVRAGES iv
1 Installation 1
2 Premiers pas dans Python 4
3 Notion de variables et fonctions Python 9
4 Tests et boucles 17
5 Les listes 38
6 Les modules importants 44
7 Python en classe de Seconde 52
8 Python en classe de Première 110
9 Python en classe de Terminale 147

AVANT-PROPROS

L'objectif de cet ouvrage est d'expliquer les bases de Python pour l'utiliser au niveau Lycée, en mathématiques, puis d'aller plus loin à travers les notions que l'on peut voir en mathématiques au lycée.
Il est destiné aussi bien aux élèves qu'aux enseignant·e·s en quête d'idées d'exercices ou de codes à proposer.

Ce livre se décompose de la manière suivante :

- un chapitre pour installer Python sur son ordinateur ;
- cinq chapitres pour voir les bases de Python, offrant quelques exercices corrigés pour être plus à l'aise ;
- un chapitre par niveau (seconde, première et terminale) dans lequel sont proposés un maximum d'exercices en relation avec les programmes de mathématiques de l'enseignement secondaire général.

Dans le chapitre concernant la classe de Terminale, certains exercices sont communs aux enseignements de spécialité et complémentaire. Vous trouverez aussi dans ce même chapitre quelques programmes pour l'enseignement des mathématiques expertes.
Chacun des programmes Python ayant un titre de la forme « `Code Python xx-yy` » est disponible dans le répertoire `python` sous le nom `xx-yy.py` (il est donc facile de le retrouver).

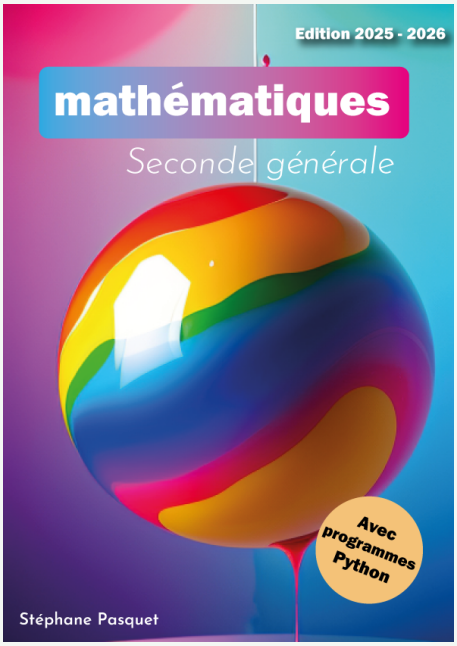
N'hésitez pas à laisser vos remarques et commentaires sur le site où vous avez acquis cet ouvrage :

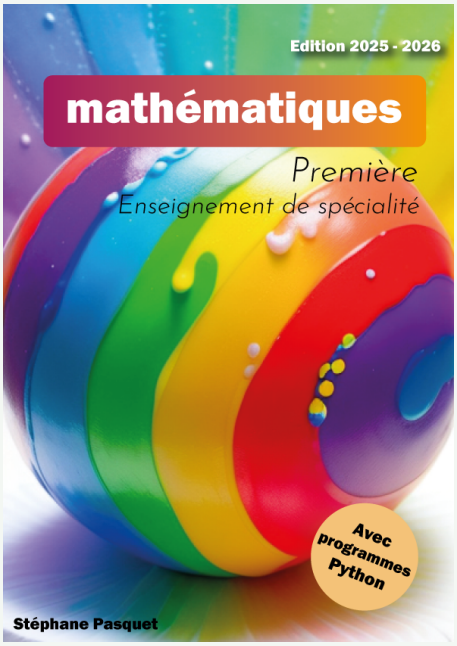
`https://mathweb.fr`.

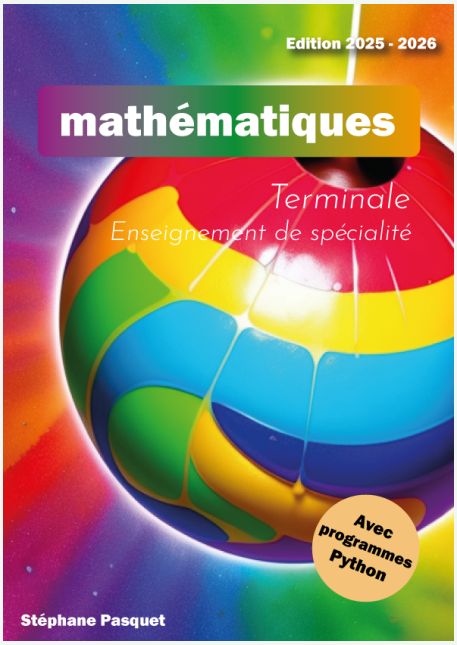
J'espère que ce livre vous sera utile.

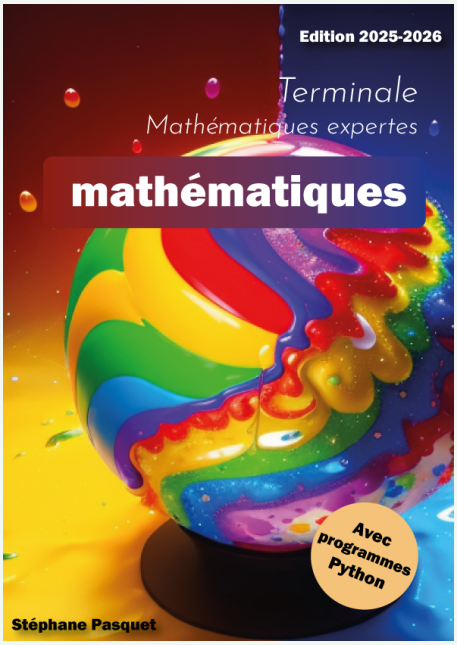
Stéphane Pasquet

AUTRES OUVRAGES









Aussi disponibles sur https://mathweb.fr

1 Installation

Plan du chapitre

I Introduction 2
II Installation de Python 2
III Installer un autre IDE 3
1 Thonny 3
2 Pyzo 3
IV Anaconda 3

Nous allons voir dans ce chapitre comment installer le nécessaire pour utiliser Python sur un ordinateur personnel.

I - Introduction

Il existe plusieurs langages de programmation, comme par exemple :

- C
- C++
- Javascript
- Pascal
- PHP
- Python
- etc.

Un *langage de programmation* est comme n'importe quel langage (comme l'anglais) : il y a des règles à respecter pour que ce que nous écrivons soit compréhensible.
La différence est que ce que nous écrivons doit être compris par une machine.

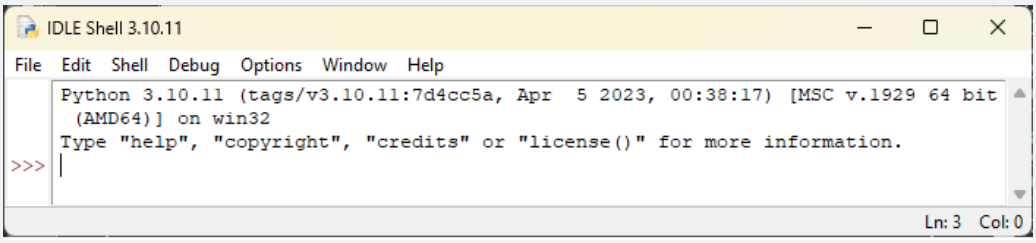
Comme beaucoup de langages de programmation, la plupart des mots importants (que l'on va qualifier de « mots-clés ») sont en anglais.

II - Installation de Python

Pour que notre ordinateur puisse comprendre le langage Python, il faut y installer un programme qui se chargera d'interpréter ce que nous écrirons plus tard.
Pour télécharger ce programme, rendez-vous sur la page internet suivante :

`https://www.python.org/downloads`

Par défaut, un IDE (*Integrated Development Environment*) – en français : EDI (Environnement De Développement) – est installé ; c'est à l'aide de cet environnement que l'on pourra écrire nos programmes.
Cet IDE par défaut est nommé IDLE (« L » pour « light »... car il est en effet très léger) et ressemble à ceci :



Bien que la version 3.13 existe à ce jour (le 9 juin 2025), j'utilise toujours la version 3.10 car mettre à jour Python, avec tous les modules utiles, est tout de même long et cela n'a que peu d'intérêt pour moi.
Les trois chevrons `>>>` présents en début d'une ligne signifient que l'on peut taper du code Python.
Cet environnement n'est pas très pratique si on code beaucoup. C'est la raison pour laquelle il est conseillé d'en installer un plus complet, tout en étant léger.

III - Installer un autre IDE

Je vais ici vous présenter deux IDE; à vous de faire votre choix en fonction de vos affinités avec chacun d'eux, sachant qu'il existe d'autres IDE.

III . 1 - Thonny

Cet IDE a le mérite d'être léger et pratique. C'est l'IDE que j'utilise personnellement au quotidien.

Adresse de téléchargement : https://thonny.org

Son interface graphique est personnalisable (sombre ou clair) et il est largement suffisant pour programmer en Python quand on débute.

III . 2 - Pyzo

Cet environnement est tout aussi léger que Thonny, et tout aussi pratique.

Adresse de téléchargement : https://pyzo.org/

Je l'utilisais avant de me rendre compte qu'il ne me convenait pas pour certains programmes plus élaborés (qui font intervenir une interface graphique). Mais il conviendra parfaitement pour les élèves de lycée qui utilisent Python dans le cadre des mathématiques.

IV - Anaconda

Certaines personnes préfèrent travailler en Python sous une distribution complète : *Anaconda*.
Le fonctionnement de cette distribution est différent du fonctionnement classique, donc je n'en dirai pas beaucoup plus dans la suite de ce document.
C'est une distribution plutôt lourde (car très complète), donc à installer sur des ordinateurs puissants. Cependant, il existe une distribution « light » : *miniconda*.
Pour installer l'environnement *Ananonda*, rendez-vous sur la page internet suivante :

https://www.anaconda.com/download

Cette distribution utilise l'IDE *Spyder*, que l'on peut aussi installer avec la distribution classique.

https://www.spyder-ide.org/

2 Premiers pas dans Python

Plan du chapitre

I Opérations mathématiques 5
1 Opérations élémentaires 5
2 Quotients et restes 5
3 Puissances 6
II Autres opérations 6
1 Arrondis 6
2 Partie entière 7
3 Valeur absolue 7
4 Racine carrée 7
III Insérer un commentaire 8
IV Récapitulatif 8

Nous allons voir dans ce chapitre les commandes natives de Python qui nous serviront pour faire des maths.

Une fonction *native* est une fonction déjà prête à être utilisée.

I - Opérations mathématiques

I . 1 - Opérations élémentaires

Les opérations élémentaires sont les additions, les soustractions, les multiplications et les divisions.

Exemple 1

1. Pour ajouter 12 et 15, on saisit :

```
>>> 12+15
37
```

2. Pour trouver la différence entre 15 et 12, on saisit :

```
>>> 15-12
3
```

3. Pour diviser 35 par 5, on saisit :

```
>>> 35/5
7.0
```

Remarque 1

Notez ici que Python retourne un nombre « à virgule » car, par défaut, quand on effectue une division, Python suppose que le résultat est un nombre *réel*, et non un entier.

4. Pour multiplier 45 par 7, on saisit :

```
>>> 45*7
315
```

I . 2 - Quotients et restes

Quand on effectue la division euclidienne de deux entiers a et b, cela peut s'écrire :

$$a = bq + r \quad , \quad 0 \leqslant r < b$$

q est le « quotient » et r, le reste.

Ces deux nombres peuvent être obtenus à l'aide de Python.

Exemple 2

1. Le quotient de 81 par 13 est obtenu en saisissant :

```
>>> 81//13
6
```

2 Le reste de la division euclidienne de 81 par 13 est obtenu en saisissant :

```
>>> 81%13
3
```

On peut ainsi écrire :

$$81 = 13 \times 6 + 3.$$

I . 3 - Puissances

Exemple 3

Pour calculer 3^7, on saisit :

```
>>> 3**7
2187
```

Remarque 2

Notez qu'il y a deux astérisques (**) pour une puissance et une astérisque (*) pour une multiplication. Ce n'est pas un hasard quand on se souvient qu'une puissance est liée à la multiplication.

II - Autres opérations

II . 1 - Arrondis

Exemple 4

1 Arrondir le nombre $\pi \approx 3,141592653589793$ à 10^{-3} près (c'est-à-dire avec 3 chiffres après la virgule), s'obtient en saisissant :

```
>>> round(3.141592653589793,3)
3.142
```

2 Arrondir le nombre $\pi \approx 3,141592653589793$ à l'unité (c'est-à-dire sans chiffres après la virgule), s'obtient en saisissant :

```
>>> round(3.141592653589793,0)
3.0
```

II . 2 - Partie entière

Exemple 5

Si on souhaite obtenir la partie entière de $\pi \approx 3,141592653589793$ au format entier (et non réel comme dans l'exemple précédent avec `round`), on fera :

```
>>> int(3.141592653589793)
3
```

II . 3 - Valeur absolue

Exemple 6

Si on souhaite calculer la différence entre deux nombres, mais que l'on ne sait pas si le résultat est positif ou négatif, on peut demander d'afficher la différence absolue, c'est-à-dire sans le signe :

```
>>> abs(3.235263-3.3258785)
0.09061549999999974
```

$3,235263 < 3,3258785$ donc la différence $3.235263 - 3.3258785$ est négative. Pour la rendre positive, on utilise donc la fonction `abs`.

Remarque 3

Notez sur cet exemple que le résultat donné par Python est mathématiquement faux. En effet, $3,235263 - 3,3258785 = -0,0906155$. Si le résultat n'est pas trop éloigné de la réalité, il n'en est pas moins faux.
Ceci est dû au fait que Python calcule de façon très particulière quand on lui donne des nombres décimaux et qu'il ne sera pas rare de voir de telles approximations.

II . 4 - Racine carrée

Exemple 7

Pour calculer la racine carrée de 2 ($\sqrt{2}$), on pourra saisir :

```
>>> 2**0.5
1.4142135623730951
```

III - Insérer un commentaire

Pour insérer un commentaire, on pourra utiliser le symbole # :

```
>>> a, b = 1, 2 # ici, on affecte à la variable "a" la valeur 1 et à la
    variable "b" la valeur 2
```

Dans un programme, on pourra aussi utiliser le triple quote, ou le triple guillemet.

Code Python 2-1

```
1 """
2 Ceci est un exemple de commentaire.
3 Nous allons ici calculer la racine carrée de 5.
4 """
5
6 print( 5**0.5 )
```

Ce dernier exemple est très utile quand on souhaite écrire un programme (ou autre chose, comme une fonction – nous le verrons plus tard) et que l'on veut mettre sa description.

IV - Récapitulatif

Opération	Syntaxe Python
Addition	+
Soustraction	-
Multiplication	*
Division	/
Quotient (division euclidienne)	//
Reste (division euclidienne)	%
Puissance	**
Arrondis	round(nombre,décimales)
Partie entière	int(nombre)
Valeur absolue	abs(nombre)
Racine carrée	nombre**0.5

3 Notion de variables et fonctions Python

Plan du chapitre

I Variables 10
1 Introduction 10
2 Casse des variables 10
3 Types de variables 11
II Fonctions 13
1 Introduction 13
2 Mode « console » et mode « fenêtre » 13
III La logique des modules 14
1 Qu'est-ce qu'un module Python ? 14
2 Les bonnes pratiques 15

Nous allons voir dans ce chapitre ce que sont des *variables* Python, ainsi que des *fonctions* Python.

I - Variables

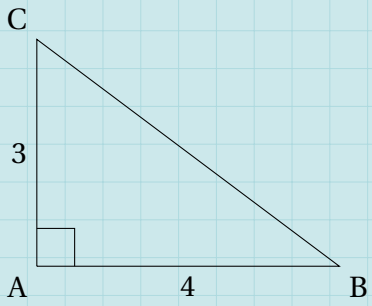
I . 1 - Introduction

En mathématiques, une variable est une lettre qui peut représenter n'importe quel nombre.
Par exemple, dans l'expression « $3x+2$ », la variable est « x ».

En informatique, c'est la même chose... ou presque ! Car on peut désigner un nombre par une lettre mais aussi un *mot*, ce qui est plus pratique pour comprendre le programme.

Exemple 8

Prenons le cas du triangle rectangle suivant :



Pour calculer la longueur du côté BC, on pourra utiliser le théorème de Pythagore qui nous dit que :

$$BC = \sqrt{AB^2 + AC^2}.$$

```
>>> AB, AC = 4, 3 # AB = 4 et AC = 3
>>> (AB**2 + AC**2)**0.5
5.0
```

On obtient BC = 5.

Dans cet exemple, vous remarquerez que l'on peut saisir la valeur de plusieurs variables, séparées par une virgule, sur une seule ligne. Ceci est plutôt pratique dans certains cas (quand il y a beaucoup de variables à définir).

I . 2 - Casse des variables

En informatique, quel que soit le langage de programmation, il faudra faire attention aux majuscules et minuscules.
Par exemple, la variable « `M` » ne désignera pas la même valeur que la variable « `m` ».

Exemple 9

```
>>> M = 3
>>> m
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'm' is not defined
```

Le message d'erreur « `NameError: name 'm' is not defined` » signifie que « `m` » n'est pas définie. En effet, c'est « `M` » qui l'est !

Les variables peuvent être nommées, en théorie, à l'aide de presque tous les caractères :

- underscore (« _ ») ;
- trait d'union (« - ») ;
- lettres accentuées
- majuscules ;
- minuscules.

Cependant, on observe que les lettres accentuées sont peu utilisées dans la pratique (c'est une habitude très ancienne) dû au fait que l'anglais ne comporte pas de telles lettres.

I . 3 - Types de variables

Il existe plusieurs types de variables :

1 **Les variables numériques**

Ce sont les variables qui représentent des nombres. Elles peuvent être de type :

- *entier* (`int`), par exemple : « `a = 3` » ;
- *flottant* (`float`), par exemple : « `a = 3.12521` »).

2 **Les chaînes de caractères**

Ce sont les variables qui représentent des mots ou des phrases. On les définit à l'aide d'*apostrophes* ou de *guillemets*.

```
>>> a = 'Bonjour.'
>>> b = "C'est génial !"
```

3 **Les booléens**

Ce sont des variables qui ne prennent que deux valeurs : « True » (vrai) ou « False » (faux). Elles sont utiles pour indiquer si une condition est vraie.

```
>>> a = ( 3**2 + 4**2 == 5**2)
>>> a
True
```

Ici, la condition est « `3**2 + 4**2 == 5**5` » : on souhaite donc savoir si $3^2 + 4^2$ est bien égal à 5^2. Quand on affiche la variable « `a` », on voit « `True` », ce qui signifie que l'égalité est vraie.

Remarque 4

Remarquez que l'on utilise le « double égal » (==) pour voir si deux variables sont égales. En effet, le simple égal (« = ») sert uniquement à affecter une valeur à une variable.

4 **Les listes (introduites à partie de la Première)**

Ces variables sont, comme leur nom l'indique, des listes de valeurs (en mathématiques, très souvent des nombres). On définit de telles variables à l'aide de crochets.

```
>>> suite = [ 1 , 5 , 10 , 15 , 20 ]
>>> suite[0]
1
>>> suite[2]
10
>>> suite[-1]
20
```

Le premier élément de la liste est *indicé* par « 0 » : c'est l'élément initial. Il y a donc un décalage entre l'indice et la position du nombre dans la liste. Quand l'indice est négatif, on part de la fin de la liste. C'est pourquoi `suite[-1]` désigne le dernier élément de la liste.

Nous reviendrons sur ce type de variables dans le chapitre 5.

5 **Les dictionnaires (non explicitement au programme en maths)**

On ne les rencontrera pas en mathématiques (car hors-programme), mais ils peuvent servir dans certains cas (en statistiques par exemple).

On définit un dictionnaire à l'aide d'accolades.

```
>>> dico = { 'Paul' : 1.70 , 'Virginie' : 1.75 }
>>> dico['Paul']
1.7
>>> dico['Virginie']
1.75
```

Les dictionnaires servent à attribuer des variables à d'autres variables.

Ici, les variables représentant les prénoms sont appelées des *clés* et celles représentant les nombres sont les *valeurs* des clés.

Je ne reviendrai pas sur ce type de variables car trop peu utilisées en mathématiques au lycée.

6 **Les tuples**

Bien que peu utilisés par les enseignants de mathématiques, ils sont très utiles dès la Seconde pour, par exemple, désigner les coordonnées de points.

```
>>> A, B = (-3,-1), (5,7)
>>> A[0]
-3
>>> B[0]
-1
>>> B[0]
5
```

On définit ici deux points $A(-3;-1)$ et $B(5;7)$ par leurs coordonnées.

L'abscisse de A est `A[0]` et l'ordonnée de A est `A[1]`.

II - Fonctions

II . 1 - Introduction

Une fonction Python a pour objectif de « faire des choses ». Elle peut effectuer une suite d'opérations et renvoyer le résultat final, ou uniquement afficher du texte ou des nombres.
On définit une fonction Python à l'aide du mot clé « `def` » suivi de nom de la fonction.

Exemple 10

```
>>> def coucou():
        print('Salut toi !')
>>> coucou()
Salut toi !
```

On remarque qu'une fonction est certes définie par le mot-clé « `def` » ainsi que par son nom, mais aussi par deux parenthèses suivies d'un « : ».

- Les parenthèses servent à y mettre des *arguments* quand la fonction en a besoin, c'est-à-dire des « variables » qui seront nécessaires aux instructions internes de la fonction ;
- les deux points servent à indiquer où commence la fonction. Par défaut, une espace horizontale s'insère quand on appuie sur la touche « Entrée » du clavier après un « : ». On appelle cela une *indentation*. Cela permet de voir ce qui est contenu dans la fonction.

  Pour sortir de la fonction, il suffit d'appuyer sur « Entrée » et sur la touche « DEL » (la touche d'effacement du clavier) pour revenir en début de la ligne.

Pour appeler la fonction, il suffit ensuite de taper son nom (sans oublier les parenthèses), et elle s'exécute.
Ici, la fonction n'a qu'une instruction : « `print('Salut toi !')` », qui affiche (`print`) une phrase.

Nous verrons bien entendu des fonctions plus élaborées dans les chapitres suivants, quand nous aurons vu les *tests* et les *boucles*.

II . 2 - Mode « console » et mode « fenêtre »

Cette terminologie n'est pas conventionnelle, mais elle me permettra de temps à autre, dans les pages suivantes, de faire la différence entre le fait d'écrire un programme :

- directement dans la console Python (quand les trois chevrons `>>>` seront présents)
- ou dans une fenêtre à part, pour sauvegarder le programme afin de l'utiliser plus tard.

Quand on est en mode « console », on peut écrire une fonction et l'appeler directement : si elle retourne un résultat, ce dernier sera affiché directement :

```
>>> def fonction(n):
        return n+3

>>> fonction(4)
7
```

Ici, la fonction retourne le nombre mis en argument augmenté de 3.

En mode « fenêtre », c'est légèrement différent. Quand on sauvegarde le programme suivant :

Code Python 3-2

```
1 def fonction(n):
2     return n + 3
```

et qu'on l'exécute, on va automatiquement en mode « console » et il ne se passe rien. Il faut appeler la fonction pour qu'elle affiche le résultat :

```
>>> fonction(7)
10
```

Si l'on souhaite que le programme affiche directement un résultat, il faut écrire par exemple :

Code Python 3-3

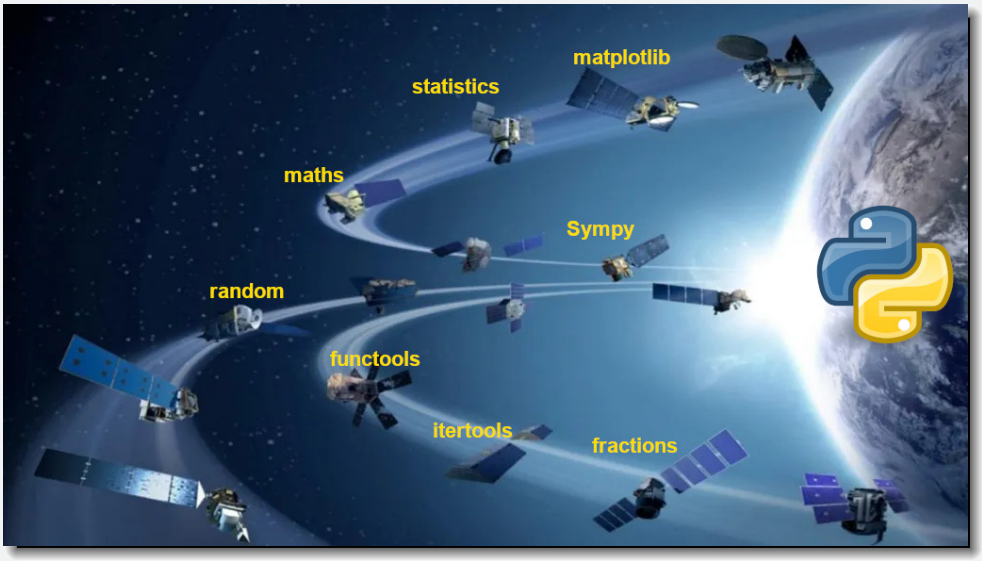
```
1 def fonction(n):
2     return n + 3
3
4 print( fonction(7) )
```

Pour qu'un programme affiche le résultat retourné par une fonction, il faut utiliser `print`.

III - La logique des modules

III . 1 - Qu'est-ce qu'un module Python ?

Les modules sont à Python ce que les propriétés sont aux mathématiques : ils servent à aller plus vite pour établir d'autres résultats.
Python peut être vu comme la planète Terre à laquelle sont rattachés plusieurs satellites :



Sans module, nous sommes capables de faire beaucoup de choses (comme nous avons pu le voir, il y a beaucoup d'outils), mais pourquoi passer du temps à définir une fonction qui existe déjà quelque part ?
C'est le but des modules : ils réunissent des fonctions qui peuvent servir à d'autres personnes. Ils peuvent même contenir des *constantes*, c'est-à-dire des nombres universellement connus.

C'est le cas par exemple du module `math` qui, comme son nom l'indique, définit plusieurs fonctions mathématiques (comme la racine carrée par exemple) et qui définit la constante π.

Chaque module possède une documentation (plus ou moins bien faite selon les auteurs des modules).

III . 2 - Les bonnes pratiques

Comme chaque module comporte plusieurs fonctions, si on a l'intention de ne faire appel qu'à quelques fonctions, il est inutile de toutes les charger.

Exemple 11

On souhaite faire appel au module `math` pour calculer la racine carrée de π.

```
>>> from math import sqrt, pi
>>> sqrt(pi)
1.7724538509055159
```

Remarque 5

1. « sqrt » est l'abréviation de « square root » (« *racine carrée* » en anglais).
2. Nous avons vu précédemment que calculer la racine carrée pouvait aussi s'écrire :

```
>>> pi**0.5
1.7724538509055159
```

On aurait donc pu n'importer que le nombre π du module `math`.

Attention 1

Beaucoup (trop) d'enseignant·e·s et d'élèves utilisent la syntaxe :

```
>>> from math import *
```

Ceci a pour conséquence que *toutes* les fonctions du module `math` sont importées, et sont stockées en mémoire pour rien. Ce n'est pas une bonne pratique car cela charge la mémoire de l'ordinateur pour rien.

D'autres utilisent la syntaxe :

```
>>> import math
```

Ceci est encore « pire » car cela impose la syntaxe suivante :

```
>>> math.sqrt( math.pi )
```

C'est une syntaxe bien trop lourde pour de petits programmes !

Cette dernière syntaxe peut s'avérer utile si un programme doit faire appel à beaucoup de modules, car dans ce cas, préfixer une fonction par le nom du module peut permettre de s'y retrouver facilement, mais dans le cadre de l'utilisation de Python en mathématiques au lycée, cela n'a aucun intérêt.

4 Tests et boucles

Plan du chapitre

I **Tests** **18**
1 Mots-clés « if », « else » et « elif » 18
2 Le mot-clé « match » 19
3 Opérateurs de comparaison 20
4 Opérateurs logiques : « and », « or » 20
5 Mot-clé « in » 22
II **Boucles** **22**
1 Boucles itératives : boucles « for » 22
a La fonction `range` 23
b Syntaxe de la boucle « for » 23
2 Boucles conditionnelles : boucles « while » 23
Enoncés **25**
Corrigés des exercices **30**

Nous abordons ici des thèmes importants à l'aide desquels il nous sera possible de créer des programmes plus complets, toujours en relation avec les mathématiques.

I - Tests

I . 1 - Mots-clés « if », « else » et « elif »

Penchons-nous dans un premier temps sur un programme élémentaire qui consiste à voir si un triangle est rectangle ou pas, à l'aide de la réciproque (ou de la contraposée) du théorème de Pythagore. Un algorithme possible est le suivant :

```
Si a*a + b*b = c*c alors:
    Retourner Vrai
Sinon:
    Retourner Faux
```

Un *algorithme* est comme une recette de cuisine : on l'exécute pas à pas.
Cet algorithme traduit en Python donne :

Code Python 4-4

```python
if a*a + b*b == c*c:
    return True
else:
    return False
```

Il arrive que certains tests ne soient pas binaires ; par exemple, on peut effectuer un test sur le reste de la division euclidienne d'un nombre entier par 3, auquel cas le test est *ternaire*. On peut alors imaginer un algorithme comme le suivant :

```
Si le reste de la division euclidienne de n par 3 est nul alors:
    Retourner "Vrai"
Sinon:
    Si le reste de la division euclidienne de n par 3 est 1 alors:
      Retourner "Pas exactement"
    Sinon:
      Retourner "Pas du tout"
```

On imbrique ici deux tests car on veut retourner trois résultats différents selon le reste de la division. En Python, cela donne :

Code Python 4-5

```python
if n % 3 == 0:
    return "Vrai"
elif n % 3 == 1:
    return "Pas exactement"
else:
    return "Pas du tout"
```

Remarque 6

`elif` est la contraction de « `else if` ».

I . 2 - Le mot-clé « match »

Depuis Python 3.10, il existe une autre façon de distinguer les divers cas possibles :

Code Python 4-6

```
1 def reste(n):
2     match n%3:
3         case 0: print('Ce nombre est divisible par 3.')
4         case 1: print('Le reste de la division euclidienne de ce nombre par 3
      vaut 1.')
5         case 2: print('Le reste de la division euclidienne de ce nombre par 3
      vaut 2.')
```

```
>>> reste(5)
Le reste de la division euclidienne de ce nombre par 3 vaut 2.
>>> reste(4)
Le reste de la division euclidienne de ce nombre par 3 vaut 1.
>>> reste(12)
Ce nombre est divisible par 3.
```

Ce mot-clé, cette instruction, s'avèrera très utile dans les situations où plus de trois valeurs pourront être prises par la variables testée.

Exemple 12

Code Python 4-7

```
1 def init(mot):
2     match mot[0]:
3         case 'A': print('Ce mot commence par la lettre A.')
4         case 'B': print('Ce mot commence par la lettre B.')
5         case 'C': print('Ce mot commence par la lettre C.')
6         case _: print('Ce mot ne commence ni par A, ni par B, ni par C.')
```

```
>>> init('Toto')
Ce mot ne commence ni par A, ni par B, ni par C.
>>> init('Abracadabra')
Ce mot commence par la lettre A.
```

Notez dans ce dernier exemple la présente de l'underscore pour désigner « tous les autres cas » : `case _`.

I . 3 - Opérateurs de comparaison

Nous avons vu qu'il était possible de voir si deux variables étaient égales à l'aide de l'opérateur « == ». On peut ainsi comparer deux variables de tout type.
Concernant les nombres, il est aussi possible de les comparer afin de voir si le premier est plus grand ou plus petit que le second.

Opérateur	Signification
`a == b`	a égal à b?
`a < b`	a strictement inférieur à b?
`a <= b`	a inférieur ou égal à b?
`a > b`	a strictement supérieur à b?
`a >= b`	a supérieur ou égal à b?

Exemple 13

```
>>> 3 <= 3
True
>>> 5 > 10
False
```

I . 4 - Opérateurs logiques : « and », « or »

1 Pour tester si deux conditions A et B sont vraies en même temps (A ET B), on utilise l'opérateur logique « `and` ».

Exemple 14

Écrivons une fonction qui retourne `True` si son argument est un nombre compris entre 3 et 10 :

Code Python 4-8

```
1 def fonction(x):
2     if (x >= 3) and (x <=10):
3         return True
4     else:
5         return False
```

que l'on peut aussi écrire :

Code Python 4-9

```
1 def fonction(x):
2     return (x >= 3) and (x <=10)
```

Cela donne par exemple :

```
>>> fonction(5.5)
True
>>> fonction(1.2)
False
```

Notez que l'on peut aussi écrire :

Code Python 4-10

```
1 def fonction(x):
2     return 3 <= x <= 10
```

Python arrive aussi à comprendre les encadrements !

2 Pour tester si l'une au moins des deux conditions est vraie, on utilise l'opérateur logique « or ».

Exemple 15

Écrivons une fonction qui retourne True si son argument est strictement inférieur à −1 ou supérieur ou égal à 7 :

Code Python 4-11

```
1 def fonction(x):
2     if (x < -1) or (x >= 7):
3         return True
4     else:
5         return False
```

que l'on peut aussi écrire :

Code Python 4-12

```
1 def fonction(x):
2     return (x < -1) or (x >= 7)
```

Cela donne par exemple :

```
>>> fonction(2)
False
>>> Fonction(8)
True
```

I . 5 - Mot-clé « in »

Le mot-clé « in » est puissant car il permet de vérifier si le contenu d'une variable est *dans* une autre variable.

Exemple 16

Testons si la chaîne de caractères « ma » est contenu dans une autre :

Code Python 4-13

```
1 def verif(chaine):
2     return "ma" in chaine
```

qui est une manière concise de voir le programme suivant :

Code Python 4-14

```
1 def verif(chaine):
2     if "ma" in chaine:
3         return True
4     else:
5         return False
```

Cela donne :

```
>>> verif("mathématiques")
True
>>> verif("cosinus")
False
```

En mathématiques, ce mot-clé ne sera pas utilisé avec des chaînes de caractères, mais plutôt avec des nombres (pour vérifier par exemple s'ils sont dans une liste).
On verra des applications ultérieurement, quand on aura un peu plus de connaissances...

II - Boucles

II . 1 - Boucles itératives : boucles « for »

Une boucle itérative est le fait d'exécuter une ou plusieurs instructions un certain nombre de fois, nombre que l'on connaît à l'avance.
Par exemple, si l'on souhaite calculer l'image de dix nombres par une fonction, on pourra utiliser une boucle itérative.
Une boucle itérative est de la forme :

« Pour <variable> prenant ses valeurs de <valeur 1> à <valeur 2> ».

II . 1 . a - La fonction `range`

C'est une fonction native de Python, c'est-à-dire une fonction incluse dans le noyau : il n'est pas nécessaire de faire appel à un module externe pour pouvoir l'utiliser.

Sa syntaxe est la suivante :

- `range(n)` représente les entiers de 0 à $n-1$: il y a donc n nombres.
- `range(a,b)` représente les entiers de a à $b-1$: il y a donc $b-a$ nombres.
- `range(a,b,p)` représente les entiers de a à $b-1$ par pas de p.
  Par exemple, `range(5,50,10)` représente les nombres 5, 15, 25, 35 et 45.

II . 1 . b - Syntaxe de la boucle « for »

Attention 2

La boucle « for » est à utiliser quand on sait à l'avance combien de fois on souhaite effectuer les instructions.

On l'utilise donc pour *parcourir* un *itérable* (c'est-à-dire une liste ou une chaîne de caractères par exemple).
L'exemple suivant nous montre comment parcourir la liste `[1,2,3,4,5,6,7,8,9,10]` pour afficher tous les carrés de 1 à 10.

Exemple 17

On souhaite faire une boucle itérative pour afficher le carrés des nombres entiers de 1 à 10 :

```
>>> for n in range(1,11):
        print(n**2)
1
4
9
16
25
36
49
64
81
100
```

II . 2 - Boucles conditionnelles : boucles « while »

Les boucles conditionnelles sont utilisées quand une ou plusieurs instructions doivent être exécutées plusieurs fois, tant qu'une condition est vraie.
Une boucle conditionnelle est de la forme :

« Tant que <condition> est vraie »

Exemple 18

Code Python 4-15

```
1 i = 0
2 while (i*i < 15):
3     i = i + 1
4
5 print(i)
```

Dans cet exemple, la condition porte sur une variable qui change de valeur à chaque itération (la variable `i`). L'instruction « `i = i+1` » est exécutée tant que la condition « $i^2 < 15$ » est vraie. Quand on exécute le programme pas à pas, on met les valeurs successives des variables dans un tableau :

Condition « $i^2 < 15$ »	\	vraie	vraie	vraie	vraie	Fausse
Valeur de i	0	1	2	3	4	\
Valeur de i^2	0	1	4	9	16	\

Attention 3

La boucle « while » est à utiliser quand on ne sait pas à l'avance combien d'itérations on souhaite effectuer.

Tests

Exercice 4.1 (fonction divisible(n))

Écrire une fonction `divisible(n)` qui admet pour argument un entier naturel n et qui affiche :

- « divisible par 2 » si n est pair,
- « divisible par 3 » si n est divisible par 3,
- « divisible par 5 » si n est divisible par 5,
- « divisible par 7 » si n est divisible par 7.

Solution page 30

Exercice 4.2 (fonction aime(n))

Écrire une fonction `aime(n)` qui admet pour argument un entier n et qui retourne :

- « Un peu » si le reste de la division euclidienne de n par 5 est égal à 1,
- « Beaucoup » si le reste de la division euclidienne de n par 5 est égal à 2,
- « Passionnément » si le reste de la division euclidienne de n par 5 est égal à 3,
- « À la folie » si le reste de la division euclidienne de n par 5 est égal à 4,
- « Pas du tout » si le reste de la division euclidienne de n par 5 est égal à 0.

Solution page 31

Exercice 4.3 (fonction inegalite_triangulaire(a,b,c))

Écrire une fonction `inegalite_triangulaire(a,b,c)`, dont les arguments a, b et c sont rangés dans l'ordre croissant, qui retourne « `True` » si un triangle de mesures a, b, c est constructible, et « `False` » dans le cas contraire.
On rappelle qu'un triangle de mesures a, b, c (avec $a < b < c$) est constructible si $c < a + b$.

Solution page 32

Boucles itératives

Exercice 4.4 (une somme)

On considère la fonction `somme(n)` définie comme suit :

Code Python 4-22

```
1 def somme(n):
2     S = 0
3     for k in range(1,n+1):
4         S = S + k
5
6     return S
```

Compléter le tableau suivant pour $n = 4$. Que fait cette fonction pour n quelconque ?

Valeurs de k	\				
Valeurs de S	0				

Solution page 32

Exercice 4.5 (somme des inverses)

Sur le modèle de l'exercice précédent, écrire une fonction `somme_inverses(n)`, avec n entier non nul, qui retourne la valeur de la somme :

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1} + \frac{1}{n}.$$

Solution page 33

Exercice 4.6 (tableau de valeurs d'une fonction)

On considère la fonction $f : x \mapsto 3x + 5$.
Proposer une fonction `tableau_valeurs(a,b)` qui affiche les images de tous les nombres entiers compris entre a et b (compris), où a et b doivent être entiers.

Solution page 33

Exercice 4.7 (signe d'une fonction)

Expliquer le programme suivant :

Code Python 4-25

```
1 def f(x):
2     return x**2 - 6*x + 5
3 for x in range(10):
4     if f(x) > 0:
5         print('f(',x,') > 0')
6     elif f(x) < 0:
7         print('f(',x,') < 0')
8     else:
9         print('f(',x,') = 0')
```

Solution page 33

Exercice 4.8 (triangle d'étoiles)

|1| Écrire une fonction Python `etoiles(n)` de sorte que l'on ait le résultat suivant :

```
>>> etoiles(5)
*
**
***
****
*****
>>> etoiles(3)
*
**
***
```

Solution page 34

Exercice 4.9 (une somme)

Écrire une fonction `somme(n)` qui renvoie, pour toute valeur de $n \geqslant 1$, la valeur de la somme :

$$S(n) = 1 + \frac{1}{1\times 2} + \frac{1}{1\times 2\times 3} + \cdots + \frac{1}{1\times 2\times 3\times \cdots \times n}.$$

Tester cette fonction et vérifier que l'on a :

```
>>> somme(100)
1.7182818284590455
```

Solution page 34

Boucles conditionnelles

Exercice 4.10 (lecture d'un programme)

On donne le programme suivant :

Code Python 4-29

```
1 a = 20
2 b = 4
3 while a*a + b*b > a * b + (a+1)*(b+1):
4     a = a - 1
5     b = b + 2
6
7 print ( a , b )
```

Compléter le tableau suivant en exécutant pas à pas ce programme :

Valeurs de A = $a^2 + b^2$:	\		…
Valeurs de B = $ab + (a+1)(b+1)$:	\		…
Test A > B :	\		…
Valeurs de a :	20		…
Valeurs de b :	4		…

Quelles sont les deux valeurs affichées par ce programme ?

Solution page 35

Exercice 4.11 (argent de poche)

Théo reçoit de sa grand-même 5 €. Celle-ci lui dit :
« Je doublerai ton argent de poche chaque mois jusqu'à ce que la somme totale que je t'aurai donnée dépasse 1 000 €. »

Écrire un programme Python, en utilisant une boucle conditionnelle, permettant de trouver le nombre de mois pendant lesquels la grand-même de Théo compte lui donner de l'argent de poche, ainsi que la somme totale obtenue.

Solution page 35

Exercice 4.12 (le retour de la mamie)

La grand-même de Théo est aussi celle de Théa, sa cousine.
Théa est un peu plus joueuse que son cousin, et sa mamie le sait. C'est la raison pour laquelle elle donne à Théa 5 € d'argent de poche tout en lui disant :
« Chaque mois, je multiplierai ce que je t'ai donné le mois précédent par 1,1 et j'ajouterai 20 €, jusqu'à ce que la somme totale que je t'aurai donnée dépasse 1 000 €. »

Écrire un programme Python, en utilisant une boucle conditionnelle, permettant de trouver le nombre de mois pendant lesquels la grand-même de Théa compte lui donner de l'argent de poche, ainsi que la somme totale obtenue.

Solution page 35

Exercice 4.13 (carré d'étoiles)

Écrire une fonction Python `carre(n)` de sorte que l'on ait :

```
>>> carre(3)
***
* *
***
>>> carre(5)
*****
*   *
*   *
*   *
*****
```

Solution page 36

Exercice 4.14 (triangle d'étoiles)

Écrire une fonction Python `triangle(n)` de sorte que l'on ait :

```
>>> triangle(4)
   *
  * *
 * * *
* * * *
>>> triangle(2)
 *
* *
```

Aide : on pourra voir la disposition des étoiles comme ci-dessous :

		*		
	*		*	
*		*		*

Solution page 36

Corrigé de l'exercice 4.1 page 25

Écrivons une fonction `divisible(n)` qui admet pour argument un entier naturel n et qui affiche :

- « divisible par 2 » si n est pair,
- « divisible par 3 » si n est divisible par 3,
- « divisible par 5 » si n est divisible par 5,
- « divisible par 7 » si n est divisible par 7.

Dans un premier temps, on serait tentés d'écrire :

Code Python 4-34

```
1 def divisible(n):
2     if n % 2 == 0:
3         return "Divisible par 2"
4     if n % 3 == 0:
5         return "Divisible par 3"
6     if n % 5 == 0:
7         return "Divisible par 5"
8     if n % 7 == 0:
9         return "Divisible par 7"
```

mais cette fonction ne convient pas. En effet, un nombre n peut être divisible par plusieurs autres nombres (par exemple, 21 est divisible par 3 et par 7). Or, quand on demande à une fonction de retourner un résultat, elle cesse de fonctionner une fois que le résultat est retourné.
Ainsi, dans ce code, pour $n = 21$, la fonction retourne « Divisible par 3 », car 3 est le premier diviseur de 21, mais n'exécute pas les tests suivants ; donc « Divisible par 7 » ne s'affiche pas.

Il faut donc préférer *afficher* le texte plutôt que le *retourner* :

Code Python 4-35

```
1 def divisible(n):
2     if n % 2 == 0:
3         print( "Divisible par 2" )
4     if n % 3 == 0:
5         print( "Divisible par 3" )
6     if n % 5 == 0:
7         print( "Divisible par 5" )
8     if n % 7 == 0:
9         print( "Divisible par 7" )
```

L'énoncé disait d'ailleurs que la fonction devait *afficher* quelque chose et non *retourner* quelque chose.

Corrigé de l'exercice 4.2 page 25

Une fonction `aime(n)` qui admet pour argument un entier n et qui retourne :

- « Un peu » si le reste de la division euclidienne de n par 5 est égal à 1,
- « Beaucoup » si le reste de la division euclidienne de n par 5 est égal à 2,
- « Passionnément » si le reste de la division euclidienne de n par 5 est égal à 3,
- « À la folie » si le reste de la division euclidienne de n par 5 est égal à 4,
- « Pas du tout » si le reste de la division euclidienne de n par 5 est égal à 0.

est par exemple la suivante :

Code Python 4-36

```
1 def aime(n):
2     if n % 5 == 1: return "Un peu"
3     elif n % 5 == 2: return "Beaucoup"
4     elif n % 5 == 3: return "Passionnément"
5     elif n % 5 == 4: return "À la folie"
6     else: return "Pas du tout"
```

Notez qu'il n'est pas nécessaire d'aller à la ligne après un « : » ; en effet, quand il n'y a qu'une instruction, on peut rester sur la même ligne. Cela rend le code plus concis.

Ici, nous ne sommes pas en mode « console » mais en mode « fenêtre », fenêtre dans laquelle nous avons écrit notre programme. Quand on lance ce programme, il ne se passe rien. C'est en mode « console » que l'on peut faire appel à cette fonction :

```
>>> aime(3)
'Passionnément'
>>> aime(16)
'Un peu'
```

Depuis Python 3.10, on peut utiliser le mot-clé « match » :

Code Python 4-37

```
1 def aime(n):
2     match n%5:
3         case 1: return "Un peu"
4         case 2: return "Beaucoup"
5         case 3: return "Passionnément"
6         case 4: return "À la folie"
7         case 0: return "Pas du tout"
```

Notez ici que cette dernière proposition n'est pas plus courte que la première.

Corrigé de l'exercice 4.3 page 25

Voici une fonction `inegalite_triangulaire(a,b,c)`, avec $a < b < c$, qui retourne « `True` » si un triangle de mesures a, b, c est constructible, et « `False` » dans le cas contraire :

Code Python 4-38

```
1 def inegalite_triangulaire(a,b,c):
2     if c < a + b:
3         return True
4     else:
5         return False
```

que l'on peut aussi écrire :

Code Python 4-39

```
1 def inegalite_triangulaire(a,b,c):
2     return c < a + b
```

Corrigé de l'exercice 4.4 page 26

Il s'agit ici d'inspecter la valeur des variables k et S à chaque itération, c'est-à-dire à chaque passage dans la boucle.

- Avant d'entrer dans la boucle, S = 0 et k n'existe pas (d'où la case barrée pour k en première colonne).
- Quand on entre dans la boucle, $k = 1$ et S vaut le nombre qu'il y avait dans S (c'est-à-dire 0) auquel on ajoute k. Donc $S = 0 + 1 = 1$.

Valeurs de k	⟍	1			
Valeurs de S	0	1			

- Au deuxième passage dans la boucle, $k = 2$ et on affecte à S la valeur 1 (ce que valait S jusqu'à présent) plus 2 (la valeur de k). Donc $S = 1 + 2 = 3$.

Valeurs de k	⟍	1	2		
Valeurs de S	0	1	3		

- Au passage suivant dans la boucle, $k = 3$ et $S = 3 + 3 = 6$.

Valeurs de k	⟍	1	2	3	
Valeurs de S	0	1	3	6	

- Nous sommes alors au niveau du dernier passage dans la boucle car maintenant, $k = 4$. S vaut alors ce qu'il y avait dans S (à savoir 6) auquel on ajoute la valeur de k : $S = 6 + 4 = 10$.

Valeurs de k	⟍	1	2	3	4
Valeurs de S	0	1	3	6	10

On constate alors que pour un n quelconque, S est la somme de tous les entiers de 1 à n :

$$S = 1 + 2 + 3 + 4 + \cdots + n.$$

Corrigé de l'exercice 4.5 page 26

La structure de la fonction demandée est identique à celle de la fonction de l'exercice précédent. Seule la formule à l'intérieure de la boucle change :

Code Python 4-40

```
1  def somme_inverses(n):
2      S = 0
3      for k in range(1,n+1):
4          S = S + 1/k
5      return S
```

À chaque itération (pour chaque passage dans la boucle), on ajoute à la valeur de S l'inverse de k quand k varie de 1 à n.
S vaut donc successivement : 0, $(0)+1$, $(0+1)+\frac{1}{2}$, $\left(0+1+\frac{1}{2}\right)+\frac{1}{3}$, etc.

Corrigé de l'exercice 4.6 page 26

Une fonction possible est la suivante :

Code Python 4-41

```
1  def tableau_valeurs(a,b):
2      for x in range(a,b+1):
3          print('f(',x,') = ',3*x+5)
```

qui donne en mode « console » :

```
>>> tableau_valeurs(-2,5)
f( -2 ) =  -1
f( -1 ) =  2
f( 0 ) =  5
f( 1 ) =  8
f( 2 ) =  11
f( 3 ) =  14
f( 4 ) =  17
f( 5 ) =  20
```

Corrigé de l'exercice 4.7 page 26

Le programme est composé de deux parties.

- La première partie (lignes 1 et 2) définit une fonction `f(x)` qui correspond à la fonction mathématique $f : x \mapsto x^2 - 6x + 5$. Cette fonction Python retourne donc l'image d'un nombre x par la fonction f.
- La seconde partie (lignes 4 à 10) est une boucle itérative sur la variable `x`, qui prend pour valeurs 0, 1, 2, 3, ... , 8 et 9.

  À chaque itération (pour chaque passage dans la boucle), on effectue un test sur le signe de $f(x)$ et on affiche le résultat.

Cela donne :

```
f( 0 ) > 0
f( 1 ) = 0
f( 2 ) < 0
```

```
f( 3 ) < 0
f( 4 ) < 0
f( 5 ) = 0
```

```
f( 6 ) > 0
f( 7 ) > 0
f( 8 ) > 0
f( 9 ) > 0
```

On peut alors déduire un tableau de signes de la fonction f sur $[0;9]$:

x	0		1		5		9
$f(x)$		$+$	0	$-$	0	$+$	

Corrigé de l'exercice 4.8 page 27

Voici une première solution :

Code Python 4-42

```
1  def etoiles(n):
2      for k in range(n+1):
3          ligne = ''
4          for _ in range(k): ligne = ligne + '*'
5          print( ligne )
```

Dans cette solution, on utilise le fait que l'on peut « ajouter » des chaînes de caractères.
Une autre solution est la suivante :

Code Python 4-43

```
1  def etoiles(n):
2      for k in range(1,n+1):
3          print( k * '*' )
```

Dans cette dernière solution, on utilise le fait que `k * '*'` affiche k fois le symbole « * ».

Corrigé de l'exercice 4.9 page 27

Voici une proposition de fonction :

Code Python 4-44

```
1  def somme(n):
2      s = 0
3      for k in range(1,n+1):
4          produit = 1
5          for p in range(1,k+1): produit = produit * p
6          s = s + 1 / produit
7
8      return s
```

Corrigé de l'exercice 4.10 page 28

Le tableau complété est le suivant :

Valeurs de A = $a^2 + b^2$:	\	416	397	388	389	400
Valeurs de B = $ab + (a+1)(b+1)$:	\	185	254	315	368	413
Test A > B :	\	vrai	vrai	vrai	vrai	faux
Valeurs de a :	20	19	18	17	16	\
Valeurs de b :	4	6	8	10	12	\

Les deux valeurs affichées sont donc : « 16 » et « 12 ».

Corrigé de l'exercice 4.11 page 28

Pour résumer la situation,

- ce mois-ci, la grand-même de Théo lui donne 5 €;
- le mois prochain, elle lui donnera le double donc 10 €. Il possèdera donc en tout 15 €;
- dans deux mois, elle lui donnera le double de ce qu'elle lui aura donné le mois précédent, donc 20 €; il possèdera donc en tout 15 + 20 = 35 €.
- etc.

Il faut donc ajouter le double de l'argent de poche reçu le mois précédent *tant que* la somme ne dépasse pas 1 000 €. On peut alors écrire le programme suivant :

Code Python 4-45

```
S = 0 # total de l'argent de poche reçu
a = 5 # argent de poche initialement reçu
n = 1 # nombre de mois

while S <= 1000:
    a = a * 2 # on double l'argent de poche
    S = S + a # on ajoute l'argent obtenu à la somme totale précédente
    n = n + 1 # on ajoute 1 mois

print(n,'mois,',S,'euros en tout.')
```

```
8 mois, 1270 euros en tout.
```

Corrigé de l'exercice 4.12 page 28

Pour résumer la situation,

- ce mois-ci, la grand-même de Théa lui donne 5 €;
- le mois prochain, elle lui donnera $1,1 \times 5 + 10 = 25,5$ €. Théa possèdera donc en tout 30,5 €;

- dans deux mois, elle lui donnera $1,1 \times 25,5 + 2 = 48,05$ €; Théa possèdera donc en tout $30,5 + 48,05 = 78,55$ €.
- etc.

Un programme possible est donc le suivant :

Code Python 4-46

```
1  S = 0 # total de l'argent de poche reçu
2  a = 5.0 # argent de poche initialement reçu
3  n = 1 # nombre de mois
4
5  while S <= 1000:
6      a = a * 1.1 + 20
7      S = S + a
8      n = n + 1
9
10 print(n,'mois,',S,'euros en tout.')
```

```
10 mois, 1262.1720432050001 euros en tout.
```

Corrigé de l'exercice 4.13 page 29

Voici une fonction possible :

Code Python 4-47

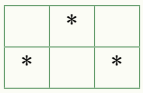
```
1  def carre(n):
2      for y in range(1,n+1):
3          if y == 1 or y == n:
4              print( '*' * n )
5          else:
6              motif = '*' + ' '*(n-2) + '*'
7              print( motif )
```

Sur la première et dernière ligne, on souhaite écrire un motif rempli d'étoiles.
Entre ces lignes, il faut que le motif soit constitué d'une étoile, puis d'une espace vide sur $(n-2)$ caractères, puis terminer avec une étoile.
La ligne 2 définit une boucle itérative sur le numéro de ligne (que j'ai désigné par la variable y) : je vais de la ligne 1 à la ligne n.
Ensuite, j'effectue un test pour savoir si la ligne est la première ou la dernière (ligne 3), auquel cas j'affiche n étoiles). Dans le cas contraire, je construis le motif comme précédemment défini, et je l'affiche.

Corrigé de l'exercice 4.14 page 29

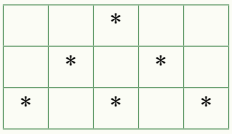
C'est un exercice plutôt compliqué car il y a du travail mathématique à faire avant le travail de programmation.
Il faut observer ce qui se passe ligne par ligne.

Si $n = 2$, il faut deux lignes comme ceci :



Le nombre de colonnes est $c = 3 = 2 \times 2 - 1$.

Si $n = 3$, il faut afficher :



Le nombre de colonnes est $c = 5 = 2 \times 3 - 1$.

On peut alors conjecturer que le nombre de colonnes nécessaires est :

$$c = 2n - 1.$$

Le motif central est « * » sur la première ligne, et pour obtenir le motif central de la ligne suivante, on y « ajoute » le motif « ␣* » (une espace vide suivie d'une étoile). On peut donc imaginer une variable `motif_central` se formant en prenant ce qu'il y avait avant et en lui ajoutant `' *'`.
Ensuite, il faut penser aux espaces vides à droite et à gauche qui tiennent sur $\frac{c-m}{2}$ colonnes, où m est la longueur du motif central.
Au final, une fonction possible est la suivante :

Code Python 4-48

```
def triangle(n):
    colonnes = 2 * n - 1
    for ligne in range(1,n+1):
        if ligne == 1:
            motif_central = '*'
        else:
            motif_central = motif_central + ' *'

        vide = (colonnes - len( motif_central)) // 2
        motif = vide * ' ' + motif_central + vide * ' '
        print( motif )
```

Bien entendu, cette solution n'est pas unique.

5 Les listes

Plan du chapitre

I **Introduction** **39**
1 Définition 39
2 Parcourir une liste 40
a Directement 40
b Par index 40
3 Connaître l'index d'un élément 41
II **Opérations diverses** **41**
1 Définir une liste vide 41
2 Ajouter un élément 41
3 Ajouter plusieurs éléments 41
4 Liste par compréhension 42
5 Liste composée d'un même nombre 42
III **Matrices** **43**
1 Introduction 43
2 Matrice vide 43

I - Introduction

I . 1 - Définition

Une *liste* est un ensemble d'objets de tout type (nombres, chaînes de caractères, booléens, ...).
Une liste est définie :

- soit par le mot-clé `list`)
- soit par des crochets et ses éléments sont séparés par une virgule.

Chaque élément est *indexé* par un nombre allant de 0 à $n-1$, où n est le nombre d'éléments de la liste.

Exemple 19

1 Une liste de nombres :

```
>>> A = list() # A est une liste vide
>>> L = [5 , 12 , 5 , -1 , 3 , 7]
>>> L[0]
5
>>> L[3]
-1
>>> L[-1]
7
```

`L[-1]` représente le dernier élément de la liste.
`L[-2]` représente l'avant-dernier.
etc.

2 Une liste de chaînes de caractères :

```
>>> noms = [ 'Luc' , 'Nathan' , 'Virginie' , 'Paul' ]
>>> noms[0]
'Luc'
>>> noms[2]
'Virginie'
```

Il sera plutôt rare d'utiliser des listes de chaînes de caractères en mathématiques.

Remarque 7

Dans la pratique, il est assez rare que l'on définisse une liste par le mot-clé `list`. Très souvent, pour déclarer une liste vide, on écrira :

```
>>> L = []
```

I . 2 - Parcourir une liste

Il existe plusieurs façons de parcourir une liste.

I . 2 . a - Directement

```
>>> L = [ 2 , 5 , 6 ]
>>> for i in L:
        print(i)
2
5
6
```

Cette méthode est utilisée pour parcourir une liste sans avoir nécessairement besoin de l'index des éléments.

I . 2 . b - Par index

```
>>> L = [ 2 , 5 , 6 ]
>>> for i in range( len(L) ):
        print( L[i] )
2
5
6
```

Cette méthode peut servir pour trouver l'index de tous les éléments égaux à un certain nombre.

Exemple 20

On souhaite afficher l'index de tous les éléments d'une liste égaux à 5.

Programme :

Code Python 5-49

```
1 L = [ 1 , 5 , 2 , 5 , 6 , 5 , 7 , 9 ]
2
3 for i in range( len(L) ):
4     if L[i] == 5:
5         print(i)
```

Résultat en mode « console » :

```
1
3
5
```

Remarque 8

Notez ici la présence du mot-clé `len`, qui désigne la longueur (*length* en anglais) de son argument. L'argument peut aussi être une chaîne de caractères.

I . 3 - Connaître l'index d'un élément

Si une liste est composée d'éléments deux à deux distincts, c'est-à-dire si aucun élément n'apparaît plus d'une seule fois, on peut connaître l'index de chaque élément à l'aide de la syntaxe suivante :

```
>>> L = [1 , 2 , 3 ]
>>> L.index(2)
1
```

Ce résultat signifie que `L[1]` contient la valeur « 2 ».

Si la liste contient plusieurs « 2 » alors `L. index(2)` retourne l'index du premier élément égal à 2.

II - Opérations diverses

II . 1 - Définir une liste vide

On peut définir une variable comme étant une liste vide de deux manières :

```
>>> L = []
>>> G = list()
```

On préfèrera très souvent la première syntaxe car plus courte et rapide.

II . 2 - Ajouter un élément

On utilise la *méthode* `append()`.

```
>>> L.append(5)
>>> L
[5]
>>> L.append(3)
>>> L
[5, 3]
```

Cette méthode ajoute *en fin de liste* l'élément mis entre parenthèses.

II . 3 - Ajouter plusieurs éléments

La méthode `append` ne permet d'ajouter qu'un élément à chaque fois. Aussi, pour ajouter plusieurs éléments en même temps, on utilise la méthode `extend` :

```
>>> L.extend([2,7])
>>> L
[5, 3, 2, 7]
```

mais on peut aussi écrire :

```
>>> L = L + [2,7]
```

II . 4 - Liste par compréhension

Il est quelques fois utile de construire une liste en s'appuyant sur une formule.

Exemple 21

On souhaite créer la liste des 10 premiers nombres entiers élevés au carré :

`L = [0 , 1 , 4 , 9 , 16 , ... , 81 ]`

On pourrait bien entendu utiliser une boucle `for` de la manière suivante :

```
>>> L = []
>>> for n in range(10):
        L.append(n**2)
```

Mais il existe une manière plus courte de faire cela : construire la liste *par compréhension* :

```
>>> L = [ n**2 for n in range(10) ]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

II . 5 - Liste composée d'un même nombre

Si l'on souhaite construire une liste de 10 « 0 » (par exemple), on pourra écrire :

```
>>> L = [0] * 10
>>> L
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Après, si un élément est changé en « 1 », par exemple le 5e élément, on pourra écrire :

```
>>> L[4] = 1
>>> L
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

III - Matrices

III . 1 - Introduction

Pour représenter une matrice, on peut utiliser une liste de listes.

Exemple 22

On souhaite représenter la matrice :

$$A = \begin{pmatrix} 1 & -1 & 2 \\ 0 & 1 & 3 \\ -1 & 1 & -2 \end{pmatrix}$$

On peut le faire ainsi :

```
>>> A = [ [1, -1, 2], [0, 1, 3], [-1, 1, -2] ]
```

III . 2 - Matrice vide

Si on souhaite définir une matrice à 5 lignes et 5 colonnes ne comportant que des « 0 », on pourra le faire ainsi :

```
>>> A = [ [0]*5 for k in range(5) ]
```

Si on souhaite insérer le nombre « 1 » dans `A[1][2]`, on écrira :

```
>>> A[1][2] = 1
>>> A
[[0, 0, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0,
   0, 0]]
```

Remarque 9

Dans `A[i][j]`, `i` désigne la ligne et `j`, la colonne.

6 Les modules importants

Plan du chapitre

I Le module `math` 45
II Le module `cmath` 47
III Le module `random` 48
IV Le module `numpy.random` 49

Nous allons voir dans ce chapitre les modules importants que l'on peut utiliser en mathématiques au lycée.

I - Le module `math`

Fonction	Exemple	Explication	Niveau
`ceil(n)`	`>>> ceil(pi)` `4`	Renvoie la partie entière supérieure de n.	Tout niveau
`comb(n,k)`	`>>> comb(10,3)` `120`	Renvoie le nombre de façons de choisir k éléments parmi n de manière non-ordonnée et sans répétition.	Terminale
`factorial(n)`	`>>> factorial(5)` `120`	Renvoie la factorielle de n sous forme d'entier, c'est-à-dire $1\times2\times3\times\cdots\times n$.	Terminale
`floor(n)`	`>>> floor(pi)` `3`	Renvoie la partie entière inférieure de n.	Tout niveau
`fsum(liste)`	`>>> fsum([0.1]*10)` `1.0`	Renvoie la somme exacte de tous les nombres de la liste. À noter que `sum([0.1]*10)` renvoie `0.99999...` et non 1.	Tout niveau
`gcd(a,b)`	`>>> gcd(45,35)` `5`	Renvoie le plus grand diviseur commun aux deux nombres a et b, donc pgcd$(a;b)$.	Seconde, Terminale (complémentaire)
`perm(n,k)`	`>>> perm(10,3)` `720`	Renvoie le nombre de façons de choisir k éléments parmi n de manière ordonnée sans répétition.	Terminale
`exp(x)`	`>>> exp(1)` `2.718281828459045`	Renvoie l'image de x par la fonction exponentielle (e^x).	Première, Terminale
`log(x)`	`>>> log(exp(1))` `1.0`	Renvoie l'image de x par la fonction logarithme népérien $(\ln(x))$.	Terminale

Fonction	Exemple	Explication	Niveau
`cos(x)`	`>>> cos(pi)` `-1.0`	Renvoie l'image de x (exprimé en radians) par la fonction cosinus ($\cos(x)$).	Tout niveau
`sin(x)`	`>>> sin(pi/2)` `1.0`	Renvoie l'image de x (exprimé en radians) par la fonction sinus ($\sin(x)$).	Tout niveau
`tan(x)`	`>>> tan(pi/4)` `0.9999999999999999`	Renvoie l'image de x (exprimé en radians) par la fonction tangente ($\tan(x)$).	Tout niveau
`degrees(a)`	`>>> degrees(pi/4)` `45.0`	Renvoie l'angle a (exprimé en radians) converti en degrés.	Tout niveau
`radians(a)`	`>>> radians(30)` `0.5235987755982988`	Renvoie l'angle a (exprimé en degrés) converti en radians.	Tout niveau
`pi`	`>>> pi` `3.141592653589793`	Renvoie la valeur approchée de π.	Tout niveau
`e`	`>>> e` `2.718281828459045`	Renvoie la valeur approchée de e[1].	Première, Terminale

II - Le module cmath

Fonction	Exemple	Explication	Niveau

`complex(a,b)`	`>>> z = complex(3,-2)`
	`>>> z`
	`(3-2j)`
		Définit un nombre complexe $z = a + bi$. Remarquez que le « i » est noté « j ».	Terminale complémentaire

`z.real, z.imag`	`>>> z = complex(3,-2)`
	`>>> z.real, z.imag`
	`(33., -2.0)`
		Renvoie respectivement la partie réelle et imaginaire d'un nombre complexe z.	Terminale complémentaire

`phase(z)`	`>>> z = complex(3,-2)` `>>> phase(z)`	Renvoie l'argument de z compris entre $]-\pi;\pi]$.	Terminale complémentaire

`polar(z)`	`>>> z = complex(3,-2)`
	`>>> polar(z)`
	`(3.6055512754639896,`
	`-0.588002603547567)`
		Renvoie le module et un argument de z, exprimé sous forme algébrique.	Terminale complémentaire

`abs(z)`	`>>> z = complex(1,1)`
	`>>> abs(z)`
	`1.4142135623730951`
		Renvoie le module de z.	Terminale complémentaire

III - Le module `random`

Ce module est indispensable pour simuler l'aléatoire.

Fonction	Exemple	Explication	Niveau
`random()`	`>>> random()` `0.6939528690800718`	Nombre réel pseudo-aléatoire dans $[0;1[$.	Tout niveau
`randint(a,b)`	`>>> randint(1,6)` `5`	Nombre entier choisi pseudo-aléatoirement dans $[a;b]$.	Tout niveau
`choice(itérable)`	`>>> choice('math')` `'a'`	Élément choisi pseudo-aléatoirement dans l'itérable (liste ou chaîne de caractères par exemple).	Tout niveau

`shuffle(liste)`	`>>> L = [5,6,9,3]`
	`>>> shuffle(L)`
	`>>> L`
	`[3, 6, 5, 9]`
		Mélange pseudo-aléatoirement les éléments de liste.	Tout niveau

`uniform(a,b)`	`>>> uniform(5,8)` `7.211930684444784`	Nombre réel pseudo-aléatoire dans $[a;b]$.	Terminale complémentaire

`sample(L,k)`	`>>> L = [1,5,8,9]`
	`>>> sample(L,k=2)`
	`[5, 1]`
		Sous-ensemble à k éléments de la liste `L` (donc sans répétition).	Tout niveau

`expovariate(λ)`	`>>>`
	`expovariate(1/5)`
	`3.2173147579260983`
		Renvoie une valeur d'une variable aléatoire suivant la loi exponentielle de paramètre `lambda` (dans l'exemple, $\lambda = 5$).	Terminale complémentaire

`gauss(m,s)`	`>>> gauss(5,1)` `4.431843559533477`	Renvoie une valeur d'une variable aléatoire suivant la loi normale de moyenne `m` et d'écart-type `s`.	Pour les profs

IV - Le module `numpy.random`

Ce module peut être utilisé pour les simulations en probabilités.
Il n'est pas à proprement parlé indispensable dans l'enseignement des mathématiques au lycée, mais il est très rapide sur des échantillons de grandes tailles.

Nombres réels aléatoires compris entre 0 et 1

Pour simuler n expériences consistant à choisir k nombres aléatoires compris entre 0 (inclus) et 1 (exclus), on pourra écrire :

```
>>> from numpy.random import rand
>>> n, k = 3, 5
>>> rand(n,k)
array([[0.60959633, 0.00612091, 0.05768334, 0.69935151, 0.90727144],
       [0.45152059, 0.52681808, 0.03733788, 0.28459394, 0.54482874],
       [0.44492722, 0.99139342, 0.62421932, 0.45552632, 0.6757296 ]])
```

Le résultat est un tableau (`array`) que l'on peut exploiter à l'aide de boucles.

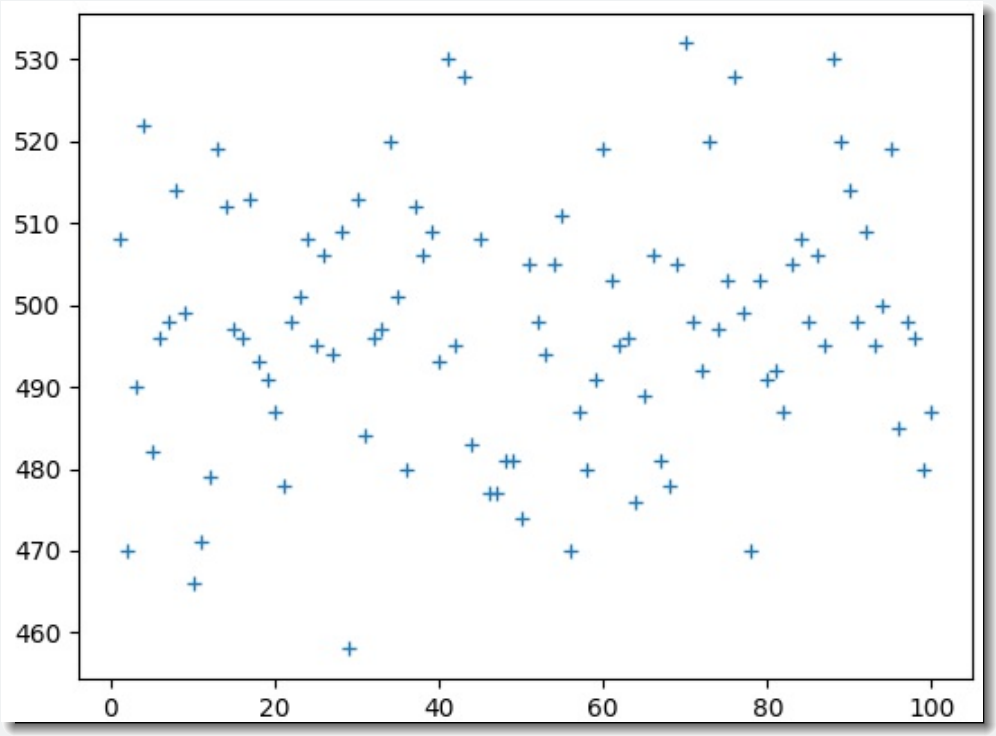
Exemple 23

On souhaite simuler 100 expériences consistant à choisir 1 000 nombres réels au hasard compris entre 0 et 1, à voir le nombre de nombres supérieurs à 0,5 pour chaque expérience puis représenter les résultats sur un graphique.
On peut alors utiliser le code suivant :

Code Python 6-50

```
1  from numpy.random import rand
2  from matplotlib.pyplot import plot, show
3
4  n, k = 100, 1000
5
6  L = rand(n,k)
7  X = range(1,n+1)
8  Y = []
9
10 for i in L:
11     y = 0
12     for j in i:
13         if j > 0.5:
14             y +=1
15     Y.append(y)
16
17 plot(X,Y,marker='+', linestyle='')
18 show()
```

Cet exemple produit un graphique semblable à celui de la page suivante.



On voit alors que le nombre de nombres supérieurs à 0,5 de chaque expérience fluctue dans un intervalle $[500-30;500+30]$ (sur l'exemple présenté ici).

Simuler une loi binomiale

Si X est une variable aléatoire suivant la loi binomiale $\mathscr{B}(n;p)$, simuler t de ces variables est possible en écrivant :

```
>>> from numpy.random import binomial
>>> L = binomial(n,p,t)
```

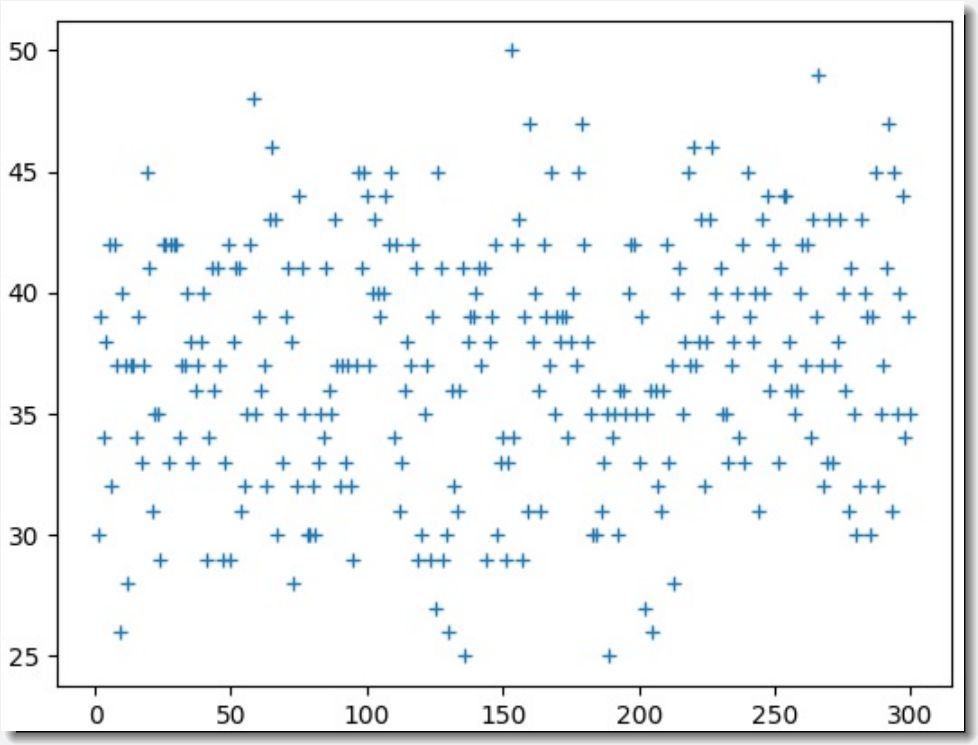
Ici aussi, le résultat est un tableau à t éléments.

Exemple 24

On souhaite simuler 300 variables aléatoires suivant la loi binomiale $\mathscr{B}(100;0,37)$ et visualiser les résultats sur un graphique.

Code Python 6-51

```
1 from numpy.random import binomial
2 from matplotlib.pyplot import plot, show
3 n, p, t = 100, 0.37, 300
4 Y = binomial(n,p,t)
5 X = range(1,t+1)
6 plot(X,Y)
7 show()
```



D'après le cours de terminale, la moyenne est $n \times p = 100 \times 0,37 = 37$ et on constate en effet que la variable fluctue autour de cette valeur (dans un intervalle qui semble être $[37-14\,;37+14]$ sur notre exemple).

Simuler une loi géométrique

Le module contient une fonction permettant de simuler t fois une variable aléatoire suivant la loi géométrique de paramètre p :

```
>>> from numpy.random import geometric
>>> L = geometric(p,t)
```

À mes yeux, cette fonction n'a pas un grand intérêt dans l'enseignement des mathématiques au lycée, mais j'ai tout de même préféré l'insérer ici au cas où certain·e·s enseignant·e·s en auraient besoin.
En mathématiques complémentaires, le programme demande de simuler une loi géométrique à partir d'une loi binomiale : c'est ce que je présente à la page VI.

7

Python en classe de Seconde

Plan du chapitre

I Les coordonnées 54
II Équations de droites 55
III Fonctions 56
1 La fonction lambda 56
2 Tracé de courbes 56
IV Statistiques 60
1 Les calculs 60
2 Les diagrammes en barres 61
3 Les histogrammes 62
V Probabilités 62
VI Échantillonnage 63
Enoncés 65
Corrigés des exercices 83

Extrait du programme de Seconde "Algorithmique et programmation"

La démarche algorithmique est, depuis les origines, une composante essentielle de l'activité mathématique. Au cycle 4, en mathématiques et en technologie, les élèves ont appris à écrire, mettre au point et exécuter un programme simple. Une consolidation des acquis du cycle 4 est proposée autour de deux idées essentielles :

- la notion de fonction ;
- la programmation comme production d'un texte dans un langage informatique.

Dans le cadre de cette activité, les élèves s'exercent à :

- décrire des algorithmes en langage naturel ou dans un langage de programmation ;
- en réaliser quelques-uns à l'aide d'un programme simple écrit dans un langage de programmation textuel ;
- interpréter, compléter ou modifier des algorithmes plus complexes.

Un langage de programmation simple d'usage est nécessaire pour l'écriture des programmes informatiques. Le langage choisi est Python, langage interprété, concis, largement répandu et pouvant fonctionner dans une diversité d'environnements. Les élèves sont entraînés à passer du langage naturel à Python et inversement.
L'algorithmique a une place naturelle dans tous les champs des mathématiques et les problèmes ainsi traités doivent être en relation avec les autres parties du programme (fonctions, géométrie, statistiques et probabilité, logique) mais aussi avec les autres disciplines ou la vie courante.
À l'occasion de l'écriture d'algorithmes et de petits programmes, il convient de transmettre aux élèves l'exigence d'exactitude et de rigueur, et de les entraîner aux pratiques systématiques de vérification et de contrôle. En programmant, les élèves revisitent les notions de variables et de fonctions sous une forme différente.

I - Les coordonnées

Un point peut être représenté, tout comme un vecteur, par un couple de nombres : ses coordonnées.
En Python, on pourra alors définir un point à l'aide d'une liste de deux nombres (définie par des crochets) ou d'un 2-uplet (défini par des parenthèses).

Exemple 25

Le point A(−3;2) peut être représenté en Python par la syntaxe suivante :

```
>>> A = [ -3 , 2 ]
```

Mais il peut aussi être représenté par :

```
>>> A = (-3, 2)
```

Dans un cas comme dans l'autre, l'abscisse est stockée dans `A[0]` et son ordonnée dans `A[1]`.

Cette façon de représenter un point ou un vecteur est bien plus pratique dans bon nombre de cas.

Exemple 26

On souhaite écrire une fonction `f` qui renvoie les coordonnées de $k\vec{u}$, connaissant les coordonnées de $\vec{u}$.
On sait que si $\vec{u}\begin{pmatrix}x\\y\end{pmatrix}$ alors le vecteur $k\vec{u}$ a pour coordonnées $\begin{pmatrix}kx\\ky\end{pmatrix}$.
Une première façon de faire est la suivante :

Code Python 7-52

```
1 def f(k,x,y):
2     return (k*x , k*y)
```

Une autre façon de faire est :

Code Python 7-53

```
1 def f(k,u):
2     return (k*u[0] , k*u[1])
```

qui donne en mode console :

```
>>> u = (-3, 2)
>>> f(-3,u)
(9, -6)
```

L'avantage ne se voit pas nécessairement dans cet exemple, mais dès qu'il faudra passer en arguments plusieurs vecteurs ou plusieurs points, la lisibilité de la fonction n'en sera que meilleure (voir les exercices portant sur le repérage dans le plan et ceux sur les vecteurs).

II - Équations de droites

Il existe deux types d'équations de droites :

- l'équation réduite, de la forme $y = mx + p$;
- les équations cartésiennes, de la forme $ax + by + c = 0$.

De même que pour les points et les vecteurs, on pourra définir une droite à partir de son équation réduite en écrivant :

```
>>> D = (m, p)
```

ou à partir d'une équation cartésienne, en écrivant :

```
>>> D = (a, b, c)
```

Exemple 27

La droite d'équation $y = -2x + 3$ pourra être représentée par :

```
>>> D = (-2, 3)
```

Cette droite admet pour équation cartésienne $-2x - y + 3 = 0$, ou encore (en multipliant tous les coefficients par -1) : $2x + y - 3 = 0$, et pourra être représentée par :

```
>>> D = (2, 1, -3)
```

Remarque 10

Dans cet exemple, on peut voir que la première variable `D` peut aussi être interprétée comme un point. Si vous souhaitez lever toute ambiguïté, vous pouvez donner un nom explicite à la variable :

```
>>> droite_reduite = (-2, 3)
>>> droite_cartesienne = (2, 1, -3)
```

III - Fonctions

III . 1 - La fonction lambda

En Python, il y a une chose bien pratique : la fonction *lambda*.
Pour définir la fonction $f : x \mapsto 3x+5$, on peut procéder ainsi :

```
>>> def f(x):
      return 3*x + 5
>>> f(3)
14
```

Mais on peut aussi écrire :

```
>>> f = lambda x: 3*x + 5
>>> f(3)
14
```

Bien que très pratique, cette syntaxe n'est que peu retenue par les élèves et les enseignants. Je ne l'utiliserai donc pas dans les exercices.

III . 2 - Tracé de courbes

Pour tracer la représentation graphique d'une fonction, il sera bien utile de faire appel au module `matplotlib`, et tout particulièrement sa bibliothèque *pyplot*.

Exemple 28

On souhaite tracer la courbe représentative de la fonction $f : x \mapsto 0,05x^3 - 3x + 1$ sur $[-5;5]$.

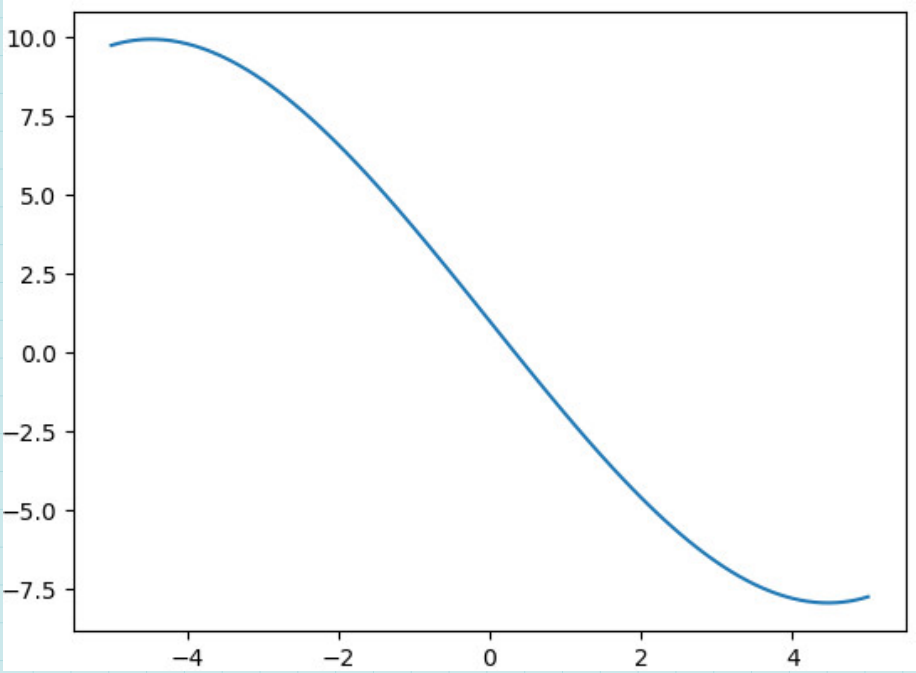
Code Python 7-54

```
1 from matplotlib.pyplot import plot, show
2
3 X = [ x/10 for x in range(-50,51) ]
4 Y = [ 0.05*x**3 - 3*x + 1 for x in X ]
5
6 plot(X, Y)
7 show()
```

- **Ligne 1 :** on commence par importer les fonctions qui nous seront utiles (ici, `plot` pour tracer et `show` pour voir le résultat dans une fenêtre).
- **Ligne 2 :** on construit une liste de valeurs de x. Dans la mesure où `range` n'accepte en aucun cas des pas décimaux, on ne peut pas aller de 0,1 en 0,1 donc j'ai décidé d'aller de $x = -50$ à $x = 51$ et de diviser toutes les valeurs par 10.
  Ainsi, je crée une liste `[-5.0, -4.9, -4.8, ... , 5.0]`.

- **Ligne 3 :** on construit une liste des images par la fonction qui nous intéresse. On calcule ici les images de toutes les valeurs contenues dans la liste `X` précédemment construite.
  → On calcule $0,05x^3 - 3x + 1$ pour tous les x contenus dans `X`.
- **Lignes 4 et 5 :** on trace la courbe et on l'affiche dans une fenêtre.

Ce code produit le graphique ci-dessous :

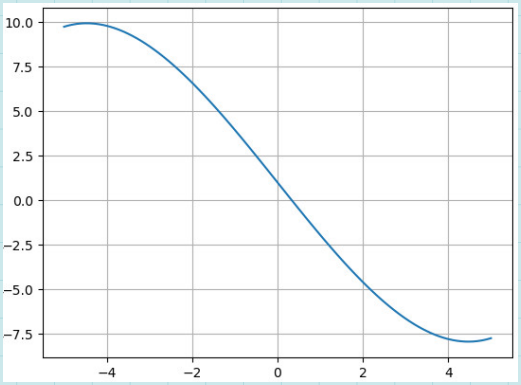


Il ne faut surtout pas s'attendre à obtenir un repère « classique » avec `matplotlib`.
Si l'exemple vous paraît trop compliqué, peut-être qu'en faisant appel au module `numpy`, les choses vous sembleront plus simples :

Exemple 29 (avec le module numpy)

Code Python 7-55

```
1 from matplotlib.pyplot import
      plot, show, grid
2 from numpy import linspace
3
4 x = linspace(-5, 5, 100)
5 y = 0.05*x**3 - 3*x + 1
6
7 plot(x,y)
8 grid()
9 show()
```



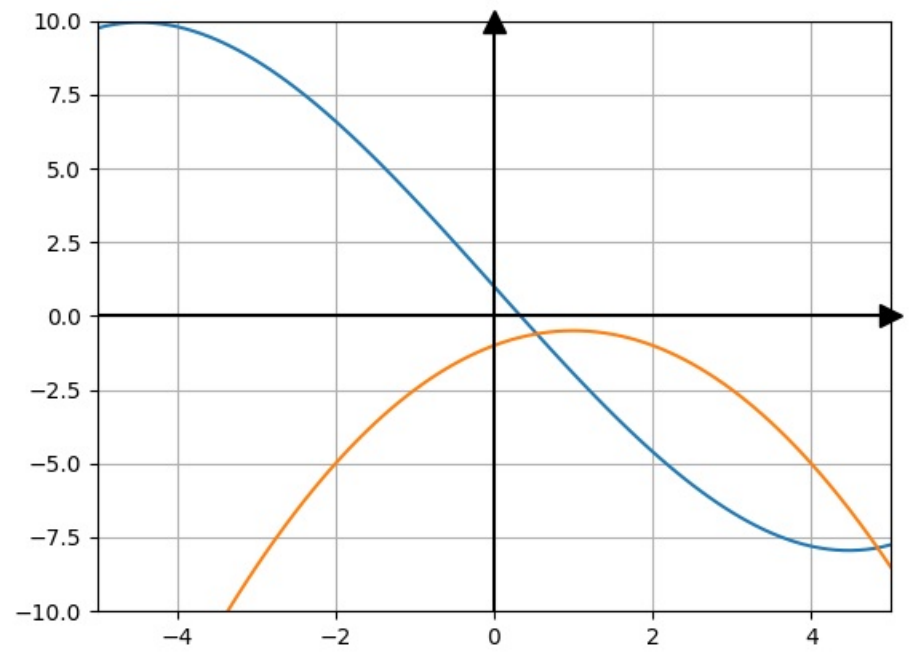
Il est vrai que la syntaxe est plus simple : le troisième argument de la fonction `linspace` est le pas des x. Ici, je suis allé de 0,01 en 0,01.

Le module `numpy` contient des fonctions mathématiques ; on peut donc tracer des fonctions plus complexes (comme les fonctions *sinus* et *cosinus*).

Exemple 30 (tracé deux courbes)

Code Python 7-56

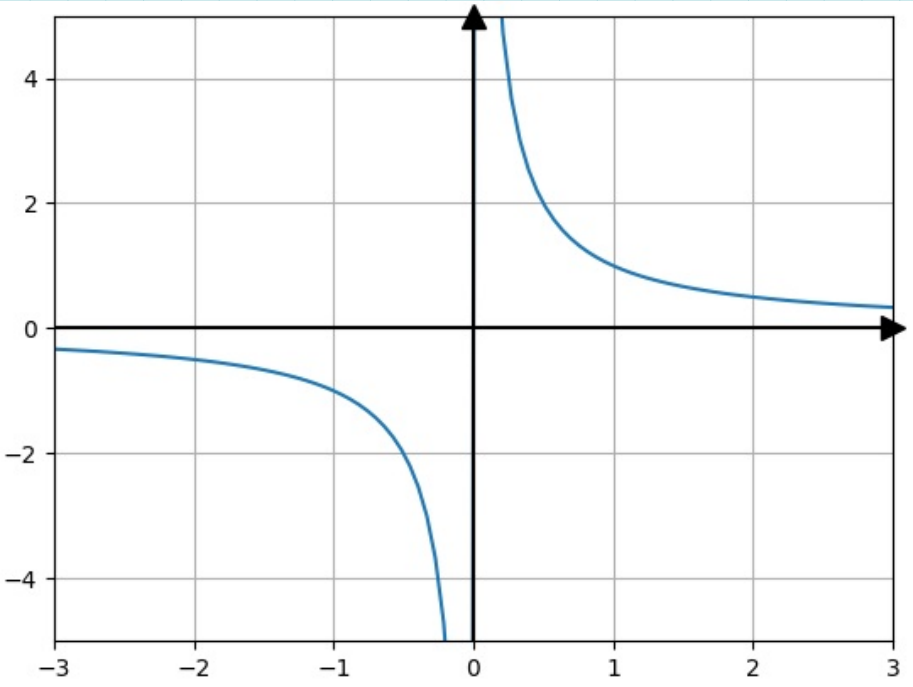
```
1  from matplotlib.pyplot import plot, show, grid, axhline, axvline, gca
2  import matplotlib.pyplot as plt
3
4  X = [x/10 for x in range(-50, 51)]
5  Y = [0.05*x**3 - 3*x + 1 for x in X]
6  Z = [-0.5*x**2 + x - 1 for x in X]
7
8  plot(X, Y)
9  plot(X, Z)
10 grid()
11 axhline(y=0, color='black', linestyle='-')
12 axvline(x=0, color='black', linestyle='-')
13
14 ax = gca()
15
16 ax.set_xlim(left=-5, right=5)
17 ax.set_ylim(bottom=-10, top=10)
18 ax.plot((1), (0), ls="", marker=">", ms=10, color="k",
       transform=ax.get_yaxis_transform(), clip_on=False)
19 ax.plot((0), (1), ls="", marker="^", ms=10, color="k",
       transform=ax.get_xaxis_transform(), clip_on=False)
20
21 show()
```



Exemple 31

Code Python 7-57

```
1  from matplotlib.pyplot import plot, show, grid, axhline, axvline, gca
2  from numpy import linspace
3
4  x = linspace(-3, 3, 100)
5  y = 1/x
6
7  plot(x, y)
8  grid()
9  axhline(y=0, color='black', linestyle='-')
10 axvline(x=0, color='black', linestyle='-')
11
12 ax = gca()
13
14 ax.set_xlim(left=-3, right=3)
15 ax.set_ylim(bottom=-5, top=5)
16 ax.plot((1), (0), ls="", marker=">", ms=10, color="k",
       transform=ax.get_yaxis_transform(), clip_on=False)
17 ax.plot((0), (1), ls="", marker="^", ms=10, color="k",
       transform=ax.get_xaxis_transform(), clip_on=False)
18
19 show()
```



IV - Statistiques

IV . 1 - Les calculs

Afin d'effectuer des calculs sur les séries statistiques, on peut faire appel au module `statistics` de Python.
Les fonctions qui pourront alors nous intéresser sont :

- `mode(L)` pour calculer le mode de la série représentée par la liste `L` ;

```
>>> from statistics import mode
>>> mode([1, 2, 2, 3, 3, 4, 4, 4, 4, 5])
4
```

- `median(L)` pour déterminer la médiane de la série représentée par la liste `L` ;

```
>>> from statistics import median
>>> median([1, 2, 2, 3, 3, 4, 4, 4, 4, 5])
3.5
```

- `quantiles(L,n=4)` pour déterminer les quartiles et la médiane de la série représentée par la liste `L` ;

```
>>> from statistics import quantiles
>>> quantiles([1, 2, 2, 3, 3, 4, 4, 4, 4, 5] , n=4)
[2.0, 3.5, 4.0]
```

- `mean(L)` pour calculer la moyenne de la série représentée par la liste `L` ;

```
>>> from statistics import mean
>>> mean([1, 2, 2, 3, 3, 4, 4, 4, 4, 5])
3.2
```

- `pvariance(L)` pour calculer la variance de la série représentée par la liste `L` ;

```
>>> from statistics import pvariance
>>> pvariance([1, 2, 2, 3, 3, 4, 4, 4, 4, 5])
1.3599999999999999
```

Attention 4

Il existe également une fonction `variance(L)`, mais elle ne retourne pas le même nombre... Sur l'exemple donné, elle retourne « `1.511111111111111` » alors que la réelle variance est bien 1,36.

- `pstdev(L)` pour calculer l'écart-type de la série représentée par la liste `L` ;

```
>>> from statistics import pstdev
>>> pstdev([1, 2, 2, 3, 3, 4, 4, 4, 4, 5])
1.16619037896906
```

Attention 5

Il existe là aussi une autre fonction `stdev`, mais il ne retourne pas la même valeur. Sur l'exemple donné, elle renvoie « `1.2292725943057183` ».

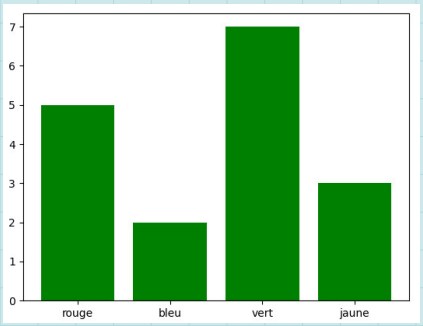
IV . 2 - Les diagrammes en barres

Les diagrammes en barres peuvent être tracés à l'aide du module `matplotlib`.

Exemple 32

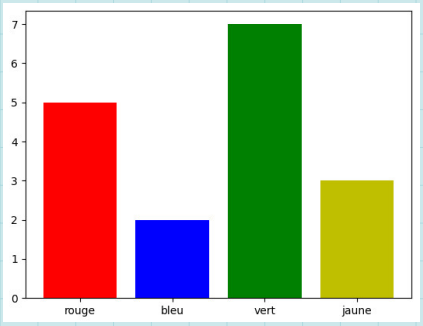
Code Python 7-58

```
1 from matplotlib.pyplot import bar,show
2
3 X = ['rouge' , 'bleu' , 'vert' , 'jaune']
4 V = [ 5 , 2 , 7 , 3 ]
5
6 bar(X, V, color='g')
7 # 'g' pour 'green'
8 show()
```



Code Python 7-59

```
1 from matplotlib.pyplot import bar,show
2
3 X = ['rouge' , 'bleu' , 'vert' , 'jaune']
4 V = [ 5 , 2 , 7 , 3 ]
5 C = [ 'r' , 'b' , 'g' , 'y' ]
6
7 bar(X, V, color=C)
8 show()
```



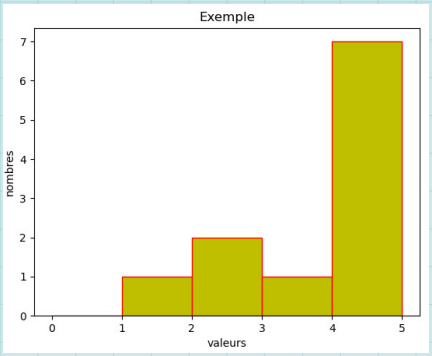
IV . 3 - Les histogrammes

Les histogrammes sont utilisés lorsque les caractères sont des *classes* (des intervalles).
`matplotlib` dispose bien d'une fonction `hist`, mais elle ne fait pas le job demandé...

Exemple 33

Code Python 7-60

```
1 from matplotlib.pyplot import hist,show,
      xlabel, ylabel, title
2
3 X = [1, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5]
4 hist(X, range = (0,5) , bins = 5, color
      = 'y' , edgecolor = 'r')
5 xlabel('valeurs')
6 ylabel('nombres')
7 title('Exemple')
8 show()
```



V - Probabilités

Il est très souvent pratique de simuler une expérience aléatoire.
Pour cela, on pourra faire appel au module `random`. Parmi les fonctions utiles, il y a :

- `randrange(a,b)` ou `randint(a,b)` qui renvoie un nombre entier aléatoire compris entre a et b (inclus).

```
>>> from random import randint
>>> randint(45,62)
52
```

- `choice(L)` qui renvoie un élément choisi au hasard dans la liste `L`.

```
>>> from random import choice
>>> choice([45,62,18,27])
62
```

- `choices(L,k)` qui renvoie k éléments choisis au hasard dans la liste `L`.

```
>>> from random import choices
>>> choices([12,36,25,84,102],k=2)
[102, 12]
```

- `shuffle(L)` qui mélange la liste L.

```
>>> from random import shuffle
>>> L = [12,36,25,84,102]
>>> shuffle(L)
>>> L
[25, 102, 12, 36, 84]
```

- `random()` qui renvoie un nombre pseudo-aléatoire compris entre 0 (inclus) et 1 (exclus).

```
>>> from random import random
0.8158474753065126
```

VI - Échantillonnage

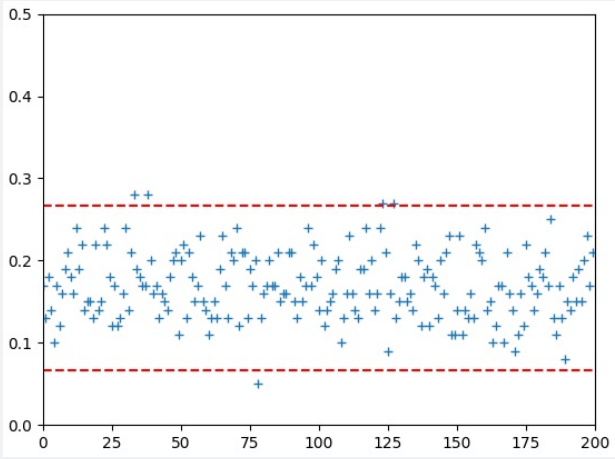
Considérons l'expérience E consistant à choisir un nombre au hasard entre 1 et 6 (inclus) : simulation d'un lancer de dé cubique équilibré.
On répète 100 fois cette expérience E et on compte le nombre de fois où l'on obtient la face « 1 ». Cette répétition constitue une expérience R. On calcule alors la fréquences d'apparition du « 1 » sur ces 100 lancers.
Répétons alors 200 fois l'expérience R et reportons sur un graphique les fréquences obtenues, à l'aide du programme suivant.

Code Python 7-61

```
from random import randint
from matplotlib.pyplot import axis, plot, show, axhline

L = [] # liste des fréquences
delta = 1/10 # 1/sqrt(100)
p = 1/6 # probabilité d'obtenir la face "1"

for _ in range(200):
    n = 0
    for _ in range(100):
        face = randint(1, 6)
        if face == 1: # si face est "1"
            n = n + 1

    L.append(n/100)

axis([0,200,0,0.5])
plot(L, marker='+', linestyle='')
axhline(y=p-delta, color='r', linestyle='--')
axhline(y=p+delta, color='r', linestyle='--')
show()
```

On obtient le graphique ci-contre.

La variable `p` joue le rôle du pourcentage de multiples de 3 obtenus lors d'une expérience R. On peut ainsi visualiser que ce pourcentage fluctue dans une sorte de « couloir » : c'est l'*intervalle de fluctuations*.



Arithmétique

Multiples et diviseurs

Exercice 7.1

Expliquer ce qu'affiche le programme suivant :

Code Python 7-62

```
1 for n in range(1,100):
2     if n % 7 == 0:
3         print(n)
```

Solution page 83

Exercice 7.2

Expliquer ce qu'affiche le programme suivant :

Code Python 7-63

```
1 n = 480
2
3 for k in range(2,n+1):
4     if n % k == 0:
5         print(k)
```

Solution page 83

Exercice 7.3

On considère le programme suivant :

Code Python 7-64

```
1  a = 450
2  b = 186
3  r = -1
4  while r !=0 :
5      q = a // b
6      r = a - b*q
7      a = b
8      b = r
9
10 print(a)
```

1. Tester ce programme « à la main » en écrivant toutes les étapes dans le tableau suivant :

Test $r \neq 0$	\	⋯
Valeurs de q	\	⋯
Valeurs de r	-1	⋯
Valeurs de a	450	⋯
Valeurs de b	186	⋯

2. À quoi correspond la valeur affichée ?

Solution page 83

Nombres premiers

Exercice 7.4

On considère la fonction Python suivante :

Code Python 7-67

```
1 def is_it(a):
2     # a est un nombre entier
3
4     for k in range(2,a):
5         if a % k == 0:
6             return False
7
8     return True
```

1. Que renvoie `is_it(18)` ?
2. Que renvoie `is_it(15)` ?
3. Que renvoie `is_it(7)` ?
4. Que fait alors ce programme ?

Solution page 84

Exercice 7.5

On considère la fonction Python suivante :

Code Python 7-68

```
def f(n):
    d = 2
    while n > 1:
        while n % d == 0:
            print(d)
            n = n // d
        d = d + 1
```

1. Qu'affiche `f(20)` ?
2. Quel rôle a cette fonction ?

Solution page 84

Exercice 7.6

Expliquer ce qu'affiche le programme suivant :

Code Python 7-69

```
for n in range(2,100):
    if n == 2:
        print(n)
    else:
        p = True
        k = 2
        while k < n and p == True:
            if n % k == 0:
                p = False
            k = k + 1

        if p == True:
            print(n)
```

Solution page 85

Divers

Exercice 7.7

Écrire une fonction Python `somme(n)` qui calcule et retourne la somme des chiffres du nombre n.

Solution page 85

Exercice 7.8

On se demande quels sont les nombres entiers n qui ont le même chiffre des unités que leur carré n^2 (comme $n = 11$ et $n^2 = 121$).
Pour cela, on considère que tous les nombres entiers n peuvent s'écrire sous la forme $n = 10x + u$, où u est un entier compris entre 0 et 9 (inclus).

1. Écrire $n = 37$ sous la forme $10x + u$.
2. Écrire $n = 146$ sous la forme $10x + u$.
3. En développant $(10x+u)^2$, énoncer une condition sur u pour que n et n^2 aient le même chiffre des unités.
4. Écrire une fonction Python `unite(n)` qui affiche les nombres compris entre 0 et n (inclus) dont le chiffre des unités est le même que celui de leur carré.

Solution page 86

Exercice 7.9

Écrire une fonction Python `fct(n)` qui affiche les nombres entiers x compris entre 0 et n (inclus) tels que la somme des chiffres de x^2 est égale à x.

Remarque 11

Si $x = 12$ alors $x^2 = 144$ et la somme des chiffres de x^2 est égale à $1 + 4 + 4 = 9$, somme différente de x donc 12 ne sera pas dans la liste.

Solution page 87

Exercice 7.10

On considère un nombre entier x. On note $d = 2x$ et $c = x^2$.
Écrire une fonction Python `fct(n)` qui affiche tous les nombres x tels que la somme des chiffres de d est égale à celle des chiffres de c (comme par exemple $x = 11$ car $d = 22$ et $c = 121$, et $2 + 2 = 1 + 2 + 1$).

Donner alors la liste des nombres compris entre 0 et 100 retournée par la fonction Python.

Solution page 88

Repérage dans le plan

Exercice 7.11

On considère la fonction Python suivante :

Code Python 7-74

```
1 def f(A,B):
2     return ( (A[0] + B[0])/2 , (A[1] + B[1])/2 )
```

1 On tape en mode console :

```
>>> A = (-3,2)
>>> B = (5,-1)
>>> f(A,B)
(1.0, 0.5)
```

Expliquer alors ce que fait la fonction Python.

2 Qu'affiche alors la séquence d'instructions suivantes :

```
>>> A = (9,1)
>>> B = (3,-5)
>>> f(A,B)
```

Solution page 88

Exercice 7.12

On considère la fonction Python suivante :

Code Python 7-75

```
1 def f(A,B):
2     return ( (B[0] - A[0])**2 + (B[1] - A[1])**2 ) ** 0.5
```

1 On tape en mode console :

```
>>> A = (-3,2)
>>> B = (5,-1)
>>> f(A,B)
8.54400374531753
```

Exécuter pas à pas la fonction Python et vérifier la valeur du résultat affiché.

2 À quoi correspond le résultat renvoyé par cette fonction ?

Solution page 89

Exercice 7.13

Le programme suivant affiche un couple de coordonnées $(x\,;y)$:

Code Python 7-76

```
1 def coord(A,B,C):
2     return ( C[0] - B[0] + A[0] , C[1] - B[1] + A[1] )
3
4 A = (-3,2)
5 B = (2,-1)
6 C = (6,3)
7
8 print( coord(A,B,C) )
```

1. Quelles sont les valeurs x et y affichée ?
2. À quoi correspondent ces coordonnées ?

Solution page 89

Exercice 7.14

On considère la fonction Python suivante :

Code Python 7-77

```
1 def zone(A,B,r):
2     return ( (B[0] - A[0])**2 + (B[1] - A[1])**2 )**0.5 <= r
```

1. On tape en mode console :

```
>>> A = (-3,5)
>>> B = (2,-1)
>>> zone(A,B,10)
True
>>> zone(A,B,5)
False
```

   Expliquer par le calcul ces deux résultats.

2. Quel est le rôle de la fonction `zone(A,B,r)` ?

Solution page 90

Exercice 7.15

On considère la fonction suivante :

Code Python 7-78

```
1 def f(A,B,C):
2     x = (B[0]-A[0])**2 + (B[1]-A[1])**2
3     y = (C[0]-A[0])**2 + (C[1]-A[1])**2
4     z = (C[0]-B[0])**2 + (C[1]-B[1])**2
5
6     return (x + y == z) or (x + z == y) or (y + z == x)
```

1 En mode console, on tape :

```
>>> A = (-3,-1)
>>> B = (4,5)
>>> C = (10,-2)
>>> f(A,B,C)
True
```

Expliquer ce résultat.

2 Quel est le rôle de cette fonction Python ?

3 Modifier cette fonction afin qu'elle renvoie la nature du triangle ABC sous forme de chaîne de caractères (par exemple, « Le triangle est rectangle isocèle ».

Solution page 91

Exercice 7.16

Écrire une fonction Python `symétrique(A,O)` renvoyant les coordonnées du symétrique d'un point A par rapport à un autre point O.

Tester cette fonction avec les points A$(-3;4)$ et O$(0;0)$.

Solution page 92

Vecteurs

Exercice 7.17

On considère la fonction Python suivante :

Code Python 7-81

```
1 def f(A,B,C):
2     AB = ( B[0] - A[0] , B[1] - A[1] )
3     AC = ( C[0] - A[0] , C[1] - A[1] )
4
5     k = AC[0] / AB[0]
6
7     return k * AB[1] == AC[1]
```

1. En mode console, on tape :

```
>>> A = (-3,-1)
>>> B = (3,2)
>>> C = (7,4)
>>> f(A,B,C)
True
```

Expliquer ce résultat.

2. Quel est le rôle de cette fonction ?

Solution page 92

Exercice 7.18

On considère le programme suivant :

Code Python 7-82

```
1  A = (0,2)   # point A, de coordonnées (0;2)
2  B = (5,-1)  # point B, de coordonnées (5;-1)
3  C = (10,-4) # point C, de coordonnées (10;-4)
4  AB = ( B[0] - A[0] , B[1] - A[1] )
5  AC = ( C[0] - A[0] , C[1] - A[1] )
6  d = AB[0] * AC[1] - AB[1] * AC[0]
7
8  if d == 0:
9      print('Les points A, B et C sont alignés.')
10 else:
11     print('Les points A, B et C ne sont pas alignés.')
```

1. Expliquer ce que représente les variables `AB` et `AC`.
2. Expliquer alors ce que représente la variable `d`.
3. Qu'affiche alors ce programme ?

Solution page 93

Exercice 7.19

On considère le programme suivant :

Code Python 7-83

```
1  def isit(A,B,C,D):
2      AB = ( B[0] - A[0] , B[1] - A[1] )
3      DC = ( C[0] - D[0] , C[1] - D[1] )
4
5      return AB == DC
```

1 Qu'obtient-on en tapant en mode console ce qui suit ?

```
>>> A = (-5,1)
>>> B = (-2,3)
>>> C = (3,3)
>>> D = (0,1)
>>> isit(A,B,C,D)
```

2 À quoi correspond le résultat affiché ?

Solution page 93

Exercice 7.20

On considère le vecteur $\vec{u}\begin{pmatrix}-3\\2\end{pmatrix}$ et le point A(5;2). On cherche à placer un point B, d'abscisse 20, de sorte que $\overrightarrow{AB}$ et $\vec{u}$ soient colinéaires. Pour cela, on considère le programme Python suivant :

Code Python 7-84

```
1  u, A = (-3,2), (5,2) # vecteur u(-3,2) et point A(5,2)
2
3  def colinear(v,w):
4      return v[0]*w[1] - v[1]*w[0] == 0
5
6  def vect(M,N):
7      return ( N[0]-M[0] , N[1]-M[1] )
8
9  y = 2
10
11 while not colinear(u,vect(A,(20,y))):
12     y = y - 1
13
14 print(y)
```

1 Expliquer ce que renvoie la fonction `colinear(v,w)`.

2 Expliquer ce que renvoie la fonction `vect(M,N)`.

3 Expliquer la boucle conditionnelle de la ligne 11.

Solution page 94

Exercice 7.21

On considère trois points $A(x_A\,;y_A)$, $B(x_B\,;y_B)$ et $C(x_C\,;y_C)$. On cherche une formule permettant de trouver les coordonnées du point D tel que ABCD est un parallélogramme. On part du principe que si tel est le cas, alors $\overrightarrow{AB} = \overrightarrow{DC}$.

1. Montrer alors que $\begin{cases} x_D &= x_C + x_A - x_B \\ y_D &= y_C + y_A - y_B \end{cases}$.
2. Écrire alors une fonction Python nommée `sommet(A,B,C)` qui renvoie les coordonnées du point D, où les arguments `A`, `B` et `C` sont définis par leurs coordonnées (par exemple, `A = (-3,2)`).
3. Tester avec les points A(−3;2), B(5;−2) et C(4;1).

Solution page 94

Exercice 7.22

Écrire une fonction Python `somme(u,v)` qui renvoie les coordonnées du vecteur $\vec{u}+\vec{v}$, où `u` et `v` sont définis par leurs coordonnées (par exemple, `u = (5,-2)` pour le vecteur $\vec{u}\begin{pmatrix}5\\-2\end{pmatrix}$).

Solution page 95

Équations de droites

Exercice 7.23

Le programme suivant retourne les valeurs de m et p de l'équation $y = mx + p$, équation de la droite (AB) :

Code Python 7-87

```
1 def reduite(A,B):
2     m = ( B[1] - A[1] ) / ( B[0] - A[0] )
3     p = A[1] - m*A[0]
4
5     return m, p
```

1. Expliquer les deux premières lignes de cette fonction.
   On suppose ici que les arguments de la fonction représentent les points définis par leurs coordonnées (par exemple, `A = (-3,2)` représente le point A(−3;2)).
2. En déduire une fonction `cartesienne(A,B)` qui renvoie a, b et c de l'équation $ax + by + c = 0$ représentant la droite (AB) (on pourra donc faire appel à la fonction `reduite(A,B)`).

Solution page 95

Exercice 7.24

On souhaite écrire une fonction Python `cartesienne(u,A)` qui renvoie une équation cartésienne de la droite (d) passant par A, représenté par une variable `A = ( A[0] , A[1] )`, et de vecteur directeur $\vec{u}$, représenté par une variable `u = ( u[0] , u[1] )`.
Pour cela, on exploite le fait que pour tout point M du plan, $\vec{u}$ et $\overrightarrow{AM}$ sont colinéaires, et donc que leur déterminant est nul.

1. Exprimer alors les nombres a, b et c dans l'équation $ax+by+c=0$ de (d) en fonction des coordonnées de A et $\vec{u}$.
2. Écrire alors la fonction Python `cartesienne(u,A)` demandée.

Solution page 96

Exercice 7.25

On donne le programme suivant :

Code Python 7-90

```
1  def det(u,v):
2      return u[0]*v[1] - u[1]*v[0]
3
4  def intersection(D,d):
5      # D = (a , b, c) représente la droite d'équation ax + by + c = 0
6      # d = (a' , b' , c') représente la droite d'équation a'x + b'y + c'
       = 0
7      u = ( d[0] , d[1] )
8      v = ( D[0] , D[1] )
9      determinant = det(u,v)
10     if determinant == 0:
11         if d[0] / D[0] == d[2] / D[2]:
12             return 'Droites confondues'
13         else:
14             return 'Droites strictement parallèles'
15     else:
16         vx = (D[0],d[0])
17         vy = (D[1],d[1])
18         vc = (-D[2] , -d[2])
19         x = det( vy , vc ) / determinant
20         y = -det( vx , vc ) / determinant
21         return x,y
```

On considère les droites (D) d'équation cartésienne $x+y+3=0$, et (d) d'équation cartésienne $6x-7y+7=0$.

1. Déterminer les coordonnées du point d'intersection de ces droites.
2. En mode console, on tape :

```
>>> D = (1,1,3)
>>> d = (6,-7,7)
>>> intersection(D,d)
(-2.1538461538461537, -0.8461538461538461)
```

Interpréter ce résultat.

Solution page 96

Exercice 7.26

1 En Python, on souhaite définir une droite dont on connaît un vecteur directeur $\vec{u}\begin{pmatrix}-3\\2\end{pmatrix}$ et passant par le point A$(1;-5)$. Pour cela, on écrit :

```
>>> u = (-3, 2)
>>> A = (1, -5)
>>> d = (u, A)
>>> d
((-3, 2), (1, -5))
>>> d[...][...]
1
```

Que doit-on écrire à la place des pointillés pour obtenir « 1 »?

2 On souhaite désormais écrire une fonction `appartient(droite,point)` qui renvoie « True » si le point mis en argument appartient à la droite mise en argument, droite définie comme précédemment.
Proposer une telle fonction.

3 Tester la fonction avec les points B$(4;-7)$ et C$(6;-8)$.

Solution page 97

Fonctions

Exercice 7.27

Écrire, à l'aide d'une boucle, un programme Python permettant d'afficher les images des nombres entiers de 0 à 10 par la fonction f définie par $f(x)=3x+5$.

Solution page 98

Exercice 7.28

Écrire, à l'aide d'une boucle, un programme Python permettant d'afficher les images des nombres entiers de -10 à 10 par la fonction f définie par $f(x)=3x+5$.

Solution page 99

Exercice 7.29

Écrire, à l'aide d'une boucle, un programme Python permettant d'afficher les images des nombres de -1 à 1, avec un pas de 0,1, par la fonction f définie par $f(x)=3x+5$.

Solution page 99

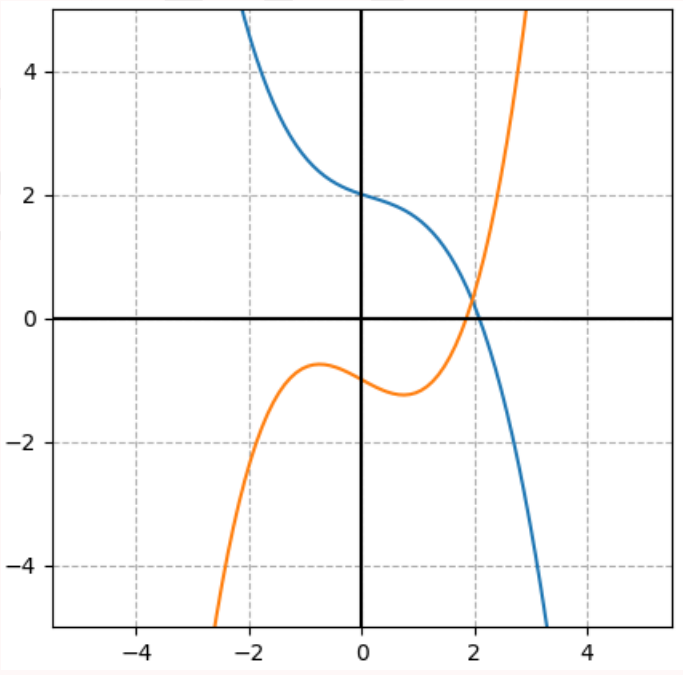
Exercice 7.30

On considère les fonctions $f : x \mapsto -0,2x^3 + 0,1x^2 - 0,3x + 2$ et $g : x \mapsto 0,3x^3 - 0,5x - 1$.
Pour tracer la représentation graphique de ces deux fonctions, on utilise le code suivant :

Code Python 7-96

```
from matplotlib.pyplot import plot, show, grid, subplots, axhline,
    axvline
from numpy import linspace

x = linspace(-5, 5, 100)
y = -0.2*x**3 + 0.1*x**2- 0.3*x + 2
z = 0.3*x**3 - 0.5*x - 1

fig, ax = subplots(figsize=(5, 5))
ax.set_ylim(-5,5)
plot(x,y)
plot(x,z)
grid(linestyle='--')

axhline(y=0, color='black', linestyle='-')
axvline(x=0, color='black', linestyle='-')

show()
```

qui produit l'image suivante :



Que faut-il changer à ce code pour produire une représentation de ces fonctions sur $[1;3]$ afin d'avoir une meilleure vue du point d'intersection des deux courbes ?

Solution page 100

Fonctions de référence

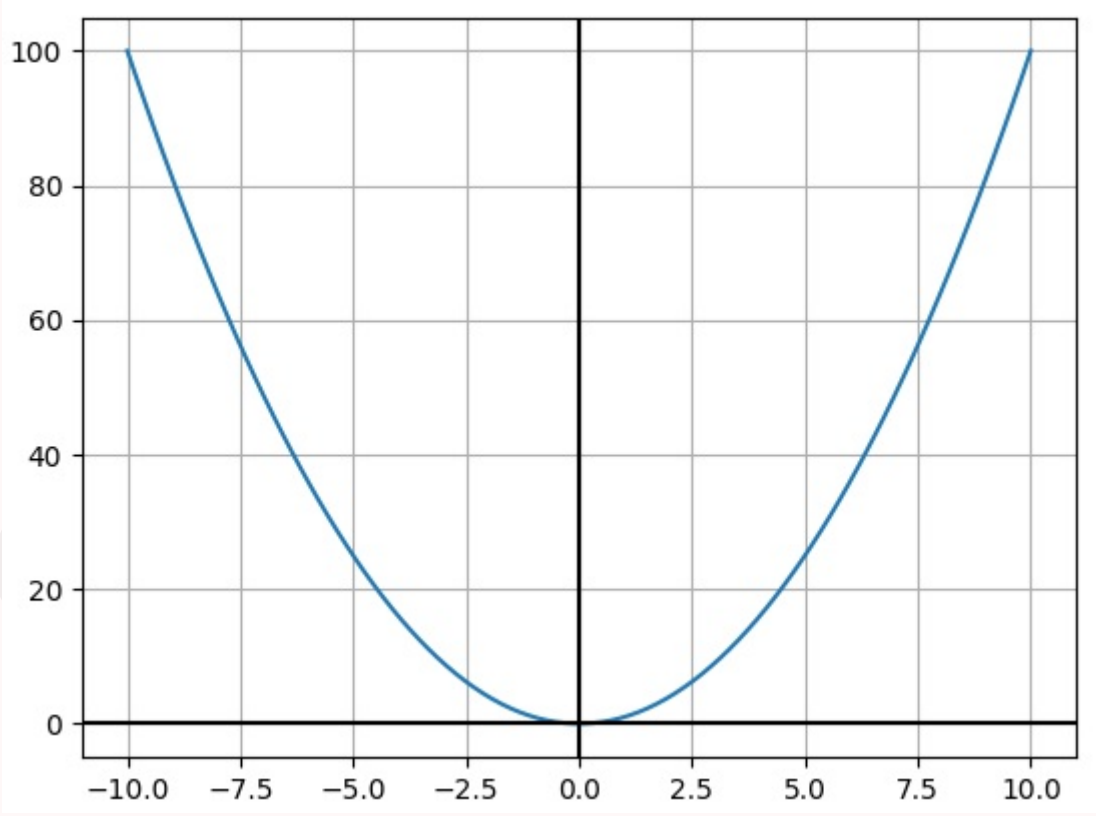
Exercice 7.31

Le code Python suivant :

Code Python 7-98

```
1 from matplotlib.pyplot import plot, show, grid, axhline, axvline
2 from numpy import linspace
3
4 x = linspace(-10, 10, 100)
5 y = x**2
6
7 plot(x,y)
8 grid()
9 show()
```

permet d'obtenir la courbe représentative de la fonction carré sur $[-10;10]$.



1. Compléter ce programme afin de visualiser la partie de la courbe telle que $x^2 \leqslant 49$ (on la représentera en rouge).
2. Compléter ce programme afin de visualiser la partie de la courbe telle que $x^2 \geqslant 25$ (on la représentera en rouge).

Solution page 101

Exercice 7.32

Modifier programme de l'exercice précédent afin qu'il affiche la courbe représentative de la fonction inverse sur $[-3;3]$, ainsi que la partie correspondant à $-2 < \frac{1}{x} < 2$.

Solution page 103

Statistiques

Remarque 14

Les listes ne sont pas officiellement au programme de maths en Seconde. Cependant, nous allons les utiliser dans cette section ainsi que dans la suivante.

Exercice 7.33

Voici un algorithme :

```
L est la liste de nombres [ 3, 5, 1 ]
N est un nombre entier
i est un nombre entier
j est un nombre entier
a est un nombre réel
m est un nombre réel
indice est un nombre entier
N ← longueur de L
Pour i variant de 0 à N-1:
    m ← L[i]
    indice ← i
    Pour j variant de i+1 à N-1:
       Si L[j]<m:
          m ← L[j]
          indice ← j
       Fin du Si
    Fin du Pour j
    a ← L[i]
    L[i] ← L[indice]
    L[indice] ← a
Fin du Pour i
Afficher L
```

1. Écrire le programme Python qui correspond à cet algorithme.
2. En exécutant ce programme, il affiche : « `[1, 3, 5]` ».
   Que semble-t-il faire ?
3. Modifier ce programme afin de déterminer et d'afficher la médiane de la série statistique entrée dans L.

Solution page 104

Exercice 7.34

On considère la fonction Python suivante :

Code Python 7-104

```
def f(L):
    # L est une liste de la forme [ (x1,n1) , (x2,n2) , ... , ]
    somme_produits = 0
    effectif_total = 0
    for x in L: # on parcourt la liste L --> x est un couple de nombres de la forme ( x[0], x[1] )
        somme_produits = somme_produits + x[0]*i[1]
        effectif_total = effectif_total + x[1]

    return somme_produits / effectif_total
```

1 Exécuter à la main et pas-à-pas cette fonction pour :

```
>>> f( [ (1,2) , (3,4) ] )
```

en complétant le tableau suivant :

x	-	(1,2)	(3,4)
somme_produits	0		
effectif_total	0		

2 À quoi correspond la valeur renvoyée par cette fonction ?

3 En s'inspirant de cette fonction, écrire une fonction `var(L)` qui renvoie la variance de la série statistique représentée par la liste `L`.

Solution page 105

Exercice 7.35

1 Compléter la fonction Python suivante de sorte qu'elle renvoie une liste correspondant aux effectifs cumulés croissants de la liste `L`, liste de la forme `[ (x1,ecc1) , (x2,ecc2) , ... ]`.

Code Python 7-106

```
def ecc(L):
    # L est une liste de la forme [ (x1,n1) , (x2,n2) , ... ] où les ni sont les effectifs
    E = []
    for i in range( len(L) ):
        if i == 0:
            E.append( ... )
        else:
            E.append( ... )

    return E
```

Remarque 15

Je rappelle que la fonction `E.append(x)` ajoute la valeur de `x` à la fin de la liste `E`. De plus, `len(L)` est le nombre d'éléments contenus dans la liste `L`.

2 Compléter la fonction Python suivante qui renvoie la médiane de la série représentée par la liste `L`.

Code Python 7-107

```
def mediane(L):
    E = ecc(L)
    N = E[-1][1] # effectif total

    if N % 2 == 0:
        moitie = ... # moitié de l'effectif total
    else:
        moitie = ...

    for i in range( len(E) ):
        if E[i][1] > moitie:
            return ...
        if E[i][1] == moitie:
            return ...
```

Solution page 106

Probabilités

Exercice 7.36

Le programme suivant effectue une simulation. Laquelle ?

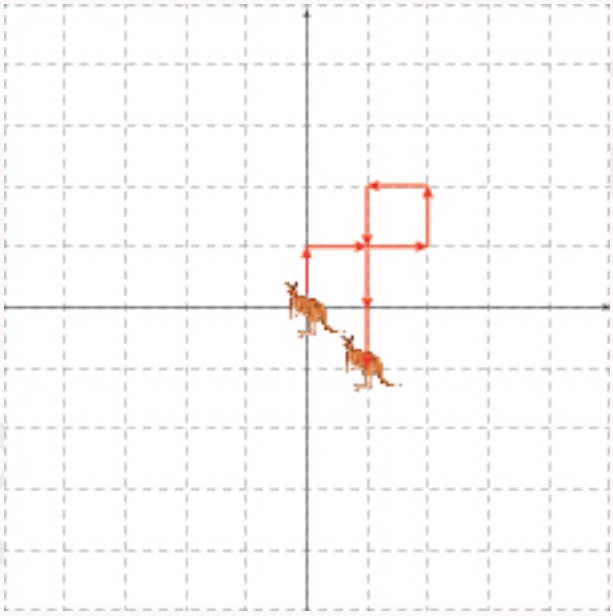
Code Python 7-110

```
from random import randint

N = [0,0,0,0,0,0]

for k in range(1000):
    n = randint(1,6)
    N[n-1] = N[n-1] + 1

print(N)
```

Solution page 107

Exercice 7.37

Un kangourou se trouve à un point donné, que l'on définira comme l'origine d'un repère orthonormé « posé au sol ».
Chaque seconde, il fait un bond. Ce bond peut être en avant, en arrière, à gauche ou à droite.
Voici un exemple de trajet pendant 8 secondes :



Écrire une fonction Python `trajet(n)` permettant de simuler un trajet de n secondes qui renvoie les coordonnées finales du kangourou.

Solution page 107

Exercice 7.38

On dispose d'un dé cubique (6 faces) et d'un dé tétraédrique (4 faces). Une expérience consiste à lancer ces deux dés et à regarder la somme des faces obtenues. Par exemple, si le dé cubique montre la face « 5 » et le dé tétraédrique repose sur la face « 1 », le résultat de l'expérience est : $5 + 1 = 6$.
Écrire une fonction Python `simulation(n)` qui simule 1 000 expériences et qui retourne la somme moyenne obtenue.

Solution page 108

Exercice 7.39

Dans les locaux d'une société, le responsable des achats sait que dans un rouleau de papier toilette de la marque « Toudou », très appréciée dans l'entreprise, il y a en moyenne 500 feuilles (il les a compté un jour où il s'ennuyait).
Selon un sondage (qu'il a effectué auprès des personnes des locaux), chaque individu utilise entre 5 et 15 feuilles par passage aux commodités.
Écrire une fonction Python `pq(n)` qui simule le passage aux toilettes de n personnes et qui renvoie le nombre de feuilles restantes. Cette fonction renverra un nombre négatif s'il manque des feuilles.

Solution page 109

Corrigé de l'exercice 7.1 page 65

La ligne 1 définit une boucle itérative qui signifie que l'on va considérer tous les nombres entiers n de 1 à 99.
Dans cette boucle, on effectue le test « `if n % 7 == 0` » c'est-à-dire que l'on vérifie si n est divisible par 7. Si tel est le cas, on affiche n (`print(n)`).

Ainsi, ce programme affiche tous les multiples de 7 compris entre 1 et 99.

Corrigé de l'exercice 7.2 page 65

On commence par définir une variable : on affecte à la variable `n` la valeur 480.
Ensuite, on définit une boucle itérative : on fait varier la variable k de 2 à n, donc ici de 2 à 480.
Dans cette boucle, on teste si le reste de la division euclidienne de n par k est nul, c'est-à-dire si n est divisible par k. Si tel est le cas, on affiche la valeur de k. k est donc un diviseur de n.
Finalement, le programme affiche tous les diviseurs de n.

Corrigé de l'exercice 7.3 page 65

1 Le tableau complété est le suivant :

Test $r \neq 0$	\	vrai	vrai	vrai	vrai	vrai	vrai	faux
Valeurs de q	\	2	2	2	1	1	2	\
Valeurs de r	-1	78	30	18	12	6	0	\
Valeurs de a	450	186	78	30	18	12	6	\
Valeurs de b	186	78	30	18	12	6	0	\

Le programme affiche donc la valeur « 6 », dernière valeur de `a` trouvée.

2 En construisant le tableau précédent, on a pu s'apercevoir que `q` et `r` étaient les valeurs successives des quotients et restes obtenus dans la division euclidienne de 450 par 186. La dernière valeur de a correspond au dernier reste non nul.

Par conséquent, la valeur affichée correspond au PGCD de 450 et 186.

Le programme proposé n'est pas optimal, mais il a été présenté ici pour être compris par le plus grand nombre de personnes. On aurait pu aussi écrire :

Code Python 7-114

```
1  a, b, r = 450, 186, -1
2  while r !=0 :
3      r = a % b
4      a = b
5      b = r
6
7  print(a)
```

ou encore, une forme dite *récursive* (bien plus compliquée pour des élèves de Seconde) :

Code Python 7-115

```
def pgcd(a,b):
    if b == 0:
        return a
    else:
        return pgcd(b , a%b)

print( pgcd(480,186) )
```

Corrigé de l'exercice 7.4 page 66

1 Ici, `a = 18`.

Au premier passage dans la boucle itérative, $k = 2$. On teste alors si le reste de la division euclidienne de 18 par 2 est nul : c'est le cas car 18 est divisible par 2. Ainsi, la fonction renvoie `False`.

2 Ici, `a = 15`.

Au premier passage dans la boucle itérative, $k = 2$. On teste alors si le reste de la division euclidienne de 15 par 2 est nul, ce qui n'est pas le cas car 15 n'est pas divisible par 2. On n'exécute donc pas l'instruction « `return False` ».

À l'itération suivante, $k = 3$. Comme 15 est divisible par 3, l'instruction `return False` est exécutée..

La fonction renvoie donc `False`.

3 Ici, `a = 7`.

On effectue alors le test « est-ce que 7 est divisible par k », pour $k = 2$, $k = 3$, $k = 4$, $k = 5$ et $k = 6$, ce qui n'est jamais le cas, donc l'instruction « `return False` » n'est jamais exécutée.

On sort alors de la boucle itérative et on effectuela dernière instruction.

Finalement, la fonction renvoie `True`.

4 On s'aperçoit alors que la fonction est chargée de voir si le nombre `a` est divisible par chacun des entiers compris entre 1 et $a-1$. S'il existe un tel entier, elle renvoie `False` ; sinon, elle renvoie `True`.

Il semblerait donc que cette fonction nous dise si le nombre `a` est un nombre premier (auquel cas, elle renvoie `True`) ou pas (auquel cas, elle renvoie `False`).

Corrigé de l'exercice 7.5 page 67

1 Regardons pas à pas ce que fait la fonction Python pour $n = 20$.

Condition $n > 1$	\	vraie	vraie	vraie	vraie	vraie	fausse
Condition `n % d == 0`	\	vraie	vraie	fausse	fausse	vraie	-
Valeur de n	\	10	5	5	5	1	-
Affichage	\	2	2	-	-	5	-
Valeur de d	2	2	3	4	5	6	-

Finalement, `f(20)` affiche :

```
>>> f(20)
2
2
5
```

2 Sur l'exemple précédent, on peut voir que la fonction affiche les facteurs qui interviennent dans la décomposition en produit de facteurs premiers de n, et que chaque facteur s'affiche autant de fois que son exposant.
Ici, $20 = 2^2 \times 5$ et « 2 » apparaît deux fois, alors que « 5 » ne s'affiche qu'une fois.

Corrigé de l'exercice 7.6 page 67

Ce programme est avant tout constitué d'une boucle itérative sur une variable `n` : cette variable prend les valeurs entières de 2 à 99. Pour chacune de ces valeurs :

- on commence par tester si elle est égale à 2 : si c'est le cas, on affiche « 2 », sinon on passe à la suite ;
- si $n \neq 2$, on initialise une variable `p` à « `True` » (variable booléenne) et une autre variable `k` à 2. On entre alors dans une boucle conditionnelle tant que $k < n$ et que `p` vaut `True`.
- si le reste de la division euclidienne de n par k vaut 0 (c'est-à-dire si k divise n), `p` vaut `False`, ce qui a pour conséquence que la boucle conditionnelle est terminée (car elle n'est exécutée que si `p` vaut `True`) ;

  La valeur de `k` est augmentée de 1 à chaque passage dans la boucle conditionnelle. Donc si `p` reste toujours à `False`, la boucle conditionnelle s'arrêtera dès lors que la condition « $k < n$ » n'est plus vérifiée, c'est-à-dire quand $k = n$;
- une fois la boucle conditionnelle terminée, si `p` vaut encore `True`, la valeur de n est affichée.

Ainsi, on n'affiche que les valeurs de n qui n'ont pas de diviseurs inférieurs à n, ce qui correspond aux nombres premiers inférieurs à 100.

Corrigé de l'exercice 7.7 page 67

Il y a plusieurs façons de calculer la somme des chiffres d'un entier.

Pour voir ces différentes façons, vous pouvez consulter la page :

https://www.mathweb.fr/euclide/2021/04/19/somme-des-chiffres-dun-nombre-en-python/

Je ne présenterai ici que la solution la plus abordable en Seconde, celle qui repose sur les mathématiques, bien qu'elle ne soit pas la plus performante.

Code Python 7-116

```
def somme(n):
    s = 0
    while n > 0:
        s = s + n % 10 # on ajoute à s le chiffre des unités
        n = n // 10 # n devient son quotient euclidien par 10

    return s
```

Regardons sur un exemple ce que fait la fonction : prenons $n = 5478$.

n > 0	\	oui	oui	oui	oui	non
n % 10	\	8	7	4	5	\
s	0	$0+8=8$	$8+7=15$	$15+4=19$	$19+5=24$	\
n	5 478	547	54	5	0	\

La variable s contient au final la somme $8+7+4+5$, qui est bien la somme demandée.

Corrigé de l'exercice 7.8 page 68

1 $n = 30+7 = 10\times 3+7$. Donc $x = 3$ et $u = 7$.

2 $n = 140+6 = 10\times 14+6$. Donc $x = 14$ et $u = 6$.

3 $(10x+u)^2 = 100x^2+20xu+u^2 = 10(10x^2+2xu)+u^2$.
Ainsi, u^2 et n^2 ont le même chiffre des unités.
Or, $u < 10$ donc nous avons 10 possibilités :

- $u = 0$: $u^2 = 0$ et donc n et n^2 ont le même chiffre des unités ;
- $u = 1$: $u^2 = 1$ et donc n et n^2 ont le même chiffre des unités ;
- $u = 2$: $u^2 = 4$ et donc n et n^2 n'ont pas le même chiffre des unités ;
- $u = 3$: $u^2 = 9$ et donc n et n^2 n'ont pas le même chiffre des unités ;
- $u = 4$: $u^2 = 16$ et donc n et n^2 n'ont pas le même chiffre des unités ;
- $u = 5$: $u^2 = 25$ et donc n et n^2 ont le même chiffre des unités ;
- $u = 6$: $u^2 = 36$ et donc n et n^2 ont le même chiffre des unités ;
- $u = 7$: $u^2 = 49$ et donc n et n^2 n'ont pas le même chiffre des unités ;
- $u = 8$: $u^2 = 64$ et donc n et n^2 n'ont pas le même chiffre des unités ;
- $u = 9$: $u^2 = 81$ et donc n et n^2 n'ont pas le même chiffre des unités.

4 Une fonction Python possible est celle proposée page suivante.

Code Python 7-117

```
def fct(n):
    for k in range(n+1):
        u = k % 10 # chiffre des unités de k
        v = k**2 % 10 # chiffre des unités du carré de k
        if u == v:
            print(k, end = ' ')
```

Explications :

- on crée une boucle itérative qui nous permet de parcourir tous les entiers de 0 à n ;
- pour chaque entier k compris entre 0 et n, on définit `u` et `v` comme étant le chiffre des unités respectivement de k et de k^2 ;
- si ces deux chiffres sont égaux (ligne 6) alors on l'affiche ; ici, l'instruction « `print(k, end=' ')` » spécifie que les nombres seront affichés en ligne (en non avec un retour à la ligne à chaque itération).

Cela donne par exemple :

```
>>> fct(20)
0 1 5 6 10 11 15 16 20
```

Corrigé de l'exercice 7.9 page 68

Code Python 7-118

```
1  def fct(n):
2      for k in range(n+1):
3          s = 0
4          x = k**2
5          while x > 0:
6              u = x % 10
7              s = s + u
8              x = x // 10
9
10         if s == k:
11             print(k, end=' ')
```

On obtient par exemple :

```
>>> fct(2000)
0 1 9
```

Si on essaie avec d'autres valeurs de n supérieures, on obtient toujours la même liste ; on peut alors conjecturer que les seuls nombres correspondant à notre recherche sont 0, 1 et 9.

Corrigé de l'exercice 7.10 page 68

Code Python 7-119

```
def fct(n):
    for k in range(n+1):
        double = 2 * k
        carre = k ** 2
        somme_d, somme_c = 0, 0
        while double > 0:
            somme_d += double % 10
            double = double // 10

        while carre > 0:
            somme_c += carre % 10
            carre = carre // 10

        if somme_c == somme_d:
            print(k, end = ' ')
```

On obtient alors :

```
>>> fct(100)
0 2 9 11 18 20 29 38 45 47 90 99
```

Corrigé de l'exercice 7.11 page 69

1 En mode console, on définit la variable `A` par un couple de nombres : `(-3,2)`. Cela peut représenter les coordonnées d'un point A.

Dans ce cas, `A[0]` représenterait son abscisse (ici, -3) et `A[1]` son ordonnée (ici, 2).

La fonction Python renvoie donc le couple $\left(\frac{x_A+x_B}{2};\frac{y_A+y_B}{2}\right)$, c'est-à-dire les coordonnées du milieu de [AB].

En l'occurrence, $\left(\frac{x_A+x_B}{2};\frac{y_A+y_B}{2}\right)=\left(\frac{-3+5}{2};\frac{2+(-1)}{2}\right)=\left(1;\frac{1}{2}\right)$.

Cela correspond bien au résultat affiché en mode console.

2 La séquence d'instructions données dans l'énoncé signifie que l'on cherche les coordonnées du milieu de [AB], avec A$(9;1)$ et B$(3;-5)$. On trouve :

$$\left(\frac{x_A+x_B}{2};\frac{y_A+y_B}{2}\right)=\left(\frac{9+3}{2};\frac{1+(-5)}{2}\right)=(6;-2).$$

Ainsi, la fonction renvoie le résultat suivant :

```
>>> A = (9,1)
>>> B = (3,-5)
>>> f(A,B)
(6.0, -2.0)
```

Corrigé de l'exercice 7.12 page 69

1 Rappelons que `**0.5` signifie « prendre la racine carrée » et que `**2` signifie « prendre le carré ».

On commence par définir la variable `A` par un couple de nombres `(-3,2)` qui peut correspondre à ses coordonnées. Il en est de même pour la variable `B`.

La fonction Python admet deux arguments `A` et `B`, et fait intervenir dans le calcul renvoyé les valeurs `A[0]` et `A[1]`, qui correspondent aux valeurs -3 et 2, ainsi que les valeurs `B[0]` et `B[1]`, qui correspondent aux valeurs 5 et -1. Le calcul fait dans la fonction est donc :

$$\sqrt{\big(5-(-3)\big)^2+(-1-2)^2}=\sqrt{8^2+(-3)^2}=\sqrt{64+9}=\sqrt{73}\approx 8{,}54400374531753.$$

Ce résultat correspond bien à celui renvoyé par la fonction en mode console.

2 Nous avons vu à la question précédente que le nombre renvoyé correspondait à :

$$\sqrt{(x_B-x_A)^2+(y_B-y_A)^2}$$

qui est la longueur AB.

Ainsi, cette fonction renvoie la longueur AB, où les coordonnées de A et B sont mis en arguments.

Corrigé de l'exercice 7.13 page 70

1 Nous avons :

`A[0]`	`A[1]`	`B[0]`	`B[1]`	`C[0]`	`C[1]`
`-3`	`2`	`2`	`-1`	`6`	`3`

La fonction calcule et affiche donc :

- `C[0] - B[0] + A[0] = 6 - 2 + (-3) = 1;`
- `C[1] - B[1] + A[1] = 3 - (-1) + 2 = 6.`

Le programme affiche donc : `(1, 6)`.

2 Appelons D$(1;6)$. En plaçant les points A, B, C et D dans un repère, on peut remarquer que ABCD semble être un parallélogramme, résultat que l'on peut démontrer en calculant les coordonnées des vecteurs :

- $\overrightarrow{AB}\begin{pmatrix}x_B-x_A\\y_B-y_A\end{pmatrix}\iff\overrightarrow{AB}\begin{pmatrix}5\\-3\end{pmatrix}$;
- $\overrightarrow{DC}\begin{pmatrix}x_C-x_D\\y_C-y_D\end{pmatrix}\iff\overrightarrow{DC}\begin{pmatrix}5\\-3\end{pmatrix}$.

L'égalité $\overrightarrow{AB}=\overrightarrow{DC}$ s'écrit aussi en coordonnées :

$$\begin{pmatrix}x_B-x_A\\y_B-y_A\end{pmatrix}=\begin{pmatrix}x_C-x_D\\y_C-y_D\end{pmatrix}$$

ou encore en égalités :

$$\begin{cases} x_B - x_A = x_C - x_D \\ y_B - y_A = y_C - y_D \end{cases}$$

ou encore :

$$\begin{cases} x_D = x_C - x_B + x_A \\ y_D = y_C - y_B + y_A \end{cases}$$

« $x_C - x_B + x_A$ » correspond justement à « `C[0] - B[0] + A[0]` » et « $y_C - y_B + y_A$ » à « `C[1] - B[1] + A[1]` ».

Ainsi, la fonction renvoie les coordonnées d'un point D, où ABCD est un parallélogramme.

Corrigé de l'exercice 7.14 page 70

1 Commençons par définir les différentes valeurs :

A[0]	A[1]	B[0]	B[1]
-3	5	2	-1

On a alors :

```
    ( (B[0] - A[0])**2 + (B[1] - A[1])**2 )**0.5
=   ( (2-(-3))**2 + (-1-5)**2 )**0.5
=   (25 + 36)**0.5
=   61**0.5
≈   7.810249675906654
```

Ainsi, le test :

« `( (B[0] - A[0])**2 + (B[1] - A[1])**2 )**0.5 <= r` »

donne le résultat « `True` » pour `r = 10` car `7.810249675906654 <= 10`.
En revanche, le test renvoie « `False` » pour `r = 5` car `7.810249675906654 > 5`.

2 La fonction renvoie le résultat du test :

`( (B[0] - A[0])**2 + (B[1] - A[1])**2 )**0.5 <= r`,

c'est-à-dire du test :

« $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2} \leqslant r$ »,

et donc du test :

$$AB \leqslant r$$

La fonction a donc pour objectif de renvoyer « `True` » si la longueur AB est inférieure ou égale au nombre r, et « `False` » dans le cas contraire.

Corrigé de l'exercice 7.15 page 71

1 En mode console, on définit trois points par leurs coordonnées.

Dans la fonction `f(A,B,C)`, on a :

- $x = (x_B - x_A)^2 + (y_B - y_A)^2 = (4-(-3))^2 + (5-(-1))^2 = 49 + 36 = 85$;
- $y = (x_C - x_A)^2 + (y_C - y_A)^2 = (10-(-3))^2 + (-2-(-1))^2 = 169 + 1 = 170$;
- $z = (x_C - x_B)^2 + (y_C - y_B)^2 = (10-4)^2 + (-2-5)^2 = 36 + 49 = 85$.

- Le test « `x + y == z` » donne « `85 + 170 == 85` » et renvoie « `False` ».
- Le test « `x + z == y` » donne « `85 + 85 == 170` » et renvoie « `True` » car l'égalité est vraie.
- Le test « `y + z == x` » donne « `170 + 85 == 85` » et renvoie « `False` » car l'égalité est fausse.

Ainsi, le test : `(x + y == z) or (x + z == y) or (y + z == x)`
est équivalent au test : `False or True or False`
et renvoie « `True` » car au moins un test est vrai.

2 Dans l'exemple précédent, on peut remarquer que x, y et z représentent respectivement les longueurs AB^2, AC^2 et BC^2.

Le test effectué vérifie donc si $AB^2 + AC^2 = BC^2$, ou si $AB^2 + BC^2 = AC^2$ ou si $AC^2 + BC^2 = AB^2$ et renvoie « `True` » si tel est le cas.

Le rôle de la fonction Python est donc de vérifier si le triangle ABC est rectangle.

3 Dans cette question, on souhaite modifier la fonction afin qu'elle affiche la nature du triangle ABC. Voici une proposition :

Code Python 7-120

```
1  def square(A,B,C):
2      nature = 'Le triangle est'
3
4      x = (B[0]-A[0])**2 + (B[1]-A[1])**2
5      y = (C[0]-A[0])**2 + (C[1]-A[1])**2
6      z = (C[0]-B[0])**2 + (C[1]-B[1])**2
7
8      if (x + y == z) or (x + z == y) or (y + z == x):
9          nature = nature + ' rectangle'
10
11     if (x == y) and (x == z) and (y == z):
12         nature  = nature + ' équilatéral'
13
14     if (x == y) or (x == z) or (y == z):
15         nature = nature + ' isocèle'
16
17     if nature == 'Le triangle est':
18         nature = nature + ' quelconque'
19
20     return nature
```

Corrigé de l'exercice 7.16 page 71

Notons A′ le symétrique de A par rapport à O. Alors, O est le milieu de [AA'] et donc :

$$x_O = \frac{x_A + x_{A'}}{2} \quad \text{soit} \quad 2x_O = x_A + x_A \quad \text{soit} \quad x_{A'} = 2x_O - x_A.$$

Il en est de même pour $y_{A'}$, d'où la fonction Python suivante :

Code Python 7-121

```
1  def symetrique(A,O):
2      return (2 * O[0] - A[0] , 2 * O[1] - A[1] )
```

On a alors par exemple, en mode console :

```
>>> A = (-3,4)
>>> O = (0,0)
>>> symetrique(A,O)
(3, -4)
```

Corrigé de l'exercice 7.17 page 72

1 Commençons par voir la valeur des variables :

A[0]	A[1]	B[0]	B[1]	C[0]	C[1]
-3	-1	3	2	7	4

Ainsi,

- La variable AB vaut :

  AB = (B[0] - A[0] , B[1] - A[1]) = (3-(-3) , 2-(-1)) = (6, 3).

- La variable AC vaut :

  AC = (C[0] - A[0] , C[1] - A[1]) = (7-(-3) , 4-(-1)) = (10, 5).

- La variable k vaut donc :

  k = AC[0] / AB[0] = 10 / 6 = 5 / 3.

Donc,

$$\texttt{k * AB[1]} = \frac{5}{3} \times 3 = 5 = \texttt{AC[1]}$$

donc le test

```
k * AB[1] == AC[1]
```

renvoie « True ».

2 À travers l'exemple précédent, on peut voir que la variable `AB` représente les coordonnées du vecteur $\overrightarrow{AB}$ et que `AC`, celles de $\overrightarrow{AC}$.
Le test « `k * AB[1] == AC[1]` » vérifie quant à lui si les coordonnées sont proportionnelles, et donc si les vecteurs sont colinéaires, auquel cas la fonction renvoie « `True` ». Et si tel est le cas, cela signifie que les points A, B et C sont alignés.

Ainsi, la fonction a pour rôle de dire si les points A, B et C sont alignés.

Corrigé de l'exercice 7.18 page 72

1 La variable `AB` est définie par un couple de nombres : d'une part `B[0] - A[0]`, d'autre part `B[1] - A[1]`, c'est-à-dire respectivement $x_B - x_A$ et $y_B - y_A$.
Ainsi, `AB` représente le vecteur $\overrightarrow{AB}$ à l'aide de ses coordonnées.
De même, `AC` représente le vecteur $\overrightarrow{AC}$ à l'aide de ses coordonnées.

2 Notons $\overrightarrow{AB}\begin{pmatrix}x\\y\end{pmatrix}$ et $\overrightarrow{AC}\begin{pmatrix}x'\\y'\end{pmatrix}$. La variable `d` prend donc la valeur $xy' - x'y$, c'est-à-dire $\det\left(\overrightarrow{AB};\overrightarrow{AC}\right)$.

3 Le programme teste donc si $\det\left(\overrightarrow{AB};\overrightarrow{AC}\right) = 0$ (ligne 10), auquel cas il affiche « Les points A, B et C sont alignés. ») ou si $\det\left(\overrightarrow{AB};\overrightarrow{AC}\right) \neq 0$ (ligne 12), auquel cas il affiche « Les points A, B et C ne sont pas alignés. ».

En l'occurrence, $\det\left(\overrightarrow{AB};\overrightarrow{AC}\right) = 5 \times (-6) - (-3) \times 10 = -30 + 30 = 0$, donc il affiche :

```
Les points A, B et C sont alignés.
```

Corrigé de l'exercice 7.19 page 73

1 Dans cette question, on définit quatre points : A(−5 ; 1), B(−2 ; 3), C(3 ; 3) et D(0 ; 1).
On a alors :

A[0]	A[1]	B[0]	B[1]	C[0]	C[1]	D[0]	D[1]
-5	1	-2	3	3	3	0	1

D'où :

AB[0]	AB[1]	DC[0]	DC[1]
3	2	3	2

Ainsi, les variables `AB` et `AD` contiennent les mêmes valeurs. Le programme affiche donc :

```
True
```

2 Le test de la fonction est basé sur la valeur des coordonnées des deux vecteurs $\overrightarrow{AB}$ et $\overrightarrow{DC}$. Si les deux vecteurs sont égaux, la fonction retourne « True », sinon elle retourne « False ». Or, $\overrightarrow{AB} = \overrightarrow{DC} \iff$ ABCD est un parallélogramme.

Ainsi, la fonction `isit(A,B,C,D)` retourne « True » si ABCD est un parallélogramme, et « False » dans le cas contraire.

Corrigé de l'exercice 7.20 page 73

1 Dans la fonction `colinear(v,w)`, la variable `v[0]*w[1] - v[1]*w[0]` représente le déterminant des vecteurs représentés par `v` et `w`.
La fonction renvoie le résultat du test « `v[0]*w[1] - v[1]*w[0] == 0` », dont retourne « True » si ce déterminant est nul, c'est-à-dire si les vecteurs sont colinéaires. Elle renvoie « False » dans le cas contraire.

Ainsi, la fonction teste si les vecteurs sont colinéaires et renvoie le résultat.

2 La fonction `vect(M,N)` renvoie un couple de nombres : `N[0]-M[0]` et `N[1]-M[1]`, qui correspondent respectivement à l'abscisse et à l'ordonnée du vecteur $\overrightarrow{MN}$.
Ainsi, cette fonction renvoie les coordonnées de $\overrightarrow{MN}$.

3 À la ligne 13, la condition de la boucle est que $\vec{u}$ et $\overrightarrow{AB}$ ne doivent pas être colinéaires.
Ainsi, les instructions de cette boucle sont exécutées tant que $\vec{u}$ et $\overrightarrow{AB}$ ne sont pas colinéaires. Dans ce cas, on enlève « 1 » à la valeur de la variable `y`.
Finalement, le programme affiche « `-8` », ce qui signifie que pour B$(20;-8)$, $\overrightarrow{AB}$ et $\vec{u}$ sont colinéaires.

Corrigé de l'exercice 7.21 page 74

1
$$\overrightarrow{AB}=\overrightarrow{DC} \iff \begin{pmatrix} x_B-x_A \\ y_B-y_A \end{pmatrix} = \begin{pmatrix} x_C-x_D \\ y_C-y_D \end{pmatrix}$$
$$\iff \begin{cases} x_B-x_A=x_C-x_D \\ y_B-y_A=y_C-y_D \end{cases}$$
$$\iff \begin{cases} x_B-x_A-x_C=-x_D \\ y_B-y_A-y_C=-y_D \end{cases}$$
$$\iff \begin{cases} x_D=x_C+x_A-x_B \\ y_D=y_C+y_A-y_B \end{cases}$$
(en multipliant par -1 de part et d'autre du signe « = »)

2 On peut alors écrire la fonction demandée en nous aidant de ce qui vient d'être obtenu :

Code Python 7-122

```
1  def sommet(A,B,C):
2      return ( C[0] + A[0] - B[0] , C[1] + A[1] - B[1] )
```

Remarque 16

Les parenthèses ne servent pas à grand-chose lors du renvoi des coordonnées du point D. On peut les mettre où ne pas les mettre.

3 Testons avec les points A(−3;2), B(5;−2) et C(4;1).

```
>>> A = (-3,2)
>>> B = (5,-2)
>>> C = (4,1)
>>> sommet(A,B,C)
(-4, 5)
```

On vérifie que ces coordonnées sont correctes car si D(−4;5) alors $\overrightarrow{DC}\begin{pmatrix}8\\-4\end{pmatrix}$ et $\overrightarrow{AB}\begin{pmatrix}8\\-4\end{pmatrix}$.

Corrigé de l'exercice 7.22 page 74

Une fonction possible est la suivante :

Code Python 7-123

```
1 def somme(u,v):
2     return u[0]+v[0] , u[1]+v[1]
```

Corrigé de l'exercice 7.23 page 74

1 Pour connaître l'équation réduite d'une droite (AB) connaissant les coordonnées des points A et B, on doit d'abord calculer sa *pente m* avec la formule du cours :

$$m = \frac{y_B - y_A}{x_B - x_A}.$$

En python, cette formule se traduit par :

```
m = ( B[1] - A[1] ) / ( B[0] - A[0] )
```

car `B[0]` et `B[1]` représentent respectivement x_B et y_B (de même pour le point A).

Une fois la pente connue, on remplace x et y par les coordonnées d'un point (A par exemple) dans l'équation réduite :

$$y_A = mx_A + p \iff p = y_A - mx_A$$

qui se traduit en Python par :

```
p = A[1] - m*A[0]
```

2 Connaissant l'équation réduite $y = mx + p$ d'une droite, il est facile d'obtenir une équation cartésienne en écrivant :

$$y = mx + p \iff mx - y + p = 0$$

Ainsi, $a = m$, $b = -1$ et $c = p$ et on peut alors avoir la fonction suivante :

Code Python 7-124

```
def cartesienne(A,B):
    m, p = reduite(A,B)
    return m, -1, p
```

Corrigé de l'exercice 7.24 page 75

1 Posons $A(x_A ; y_A)$ et $\vec{u}\begin{pmatrix}\alpha\\ \beta\end{pmatrix}$. On considère alors $M(x ; y) \in (d)$.

$$\begin{aligned}\vec{u} \text{ et } \overrightarrow{AM} \text{ colinéaires} &\iff \begin{vmatrix}\alpha & x-x_A\\ \beta & y-y_A\end{vmatrix} = 0\\ &\iff \alpha(y-y_A)-\beta(x-x_A)=0\\ &\iff \alpha y-\alpha y_A-\beta x+\beta x_A=0\\ &\iff -\beta x+\alpha y+\beta x_A-\alpha y_A=0\end{aligned}$$

Ainsi, $a=-\beta$, $b=\alpha$ et $c=\beta x_A-\alpha y_A$.

2 À l'aide de ces égalités, on peut écrire la fonction Python suivante :

Code Python 7-125

```
def cartesienne(u,A):
    a = -u[1]
    b = u[0]
    c = u[1]*A[0] - u[0]*A[1]

    return a, b, c
```

Corrigé de l'exercice 7.25 page 75

1 Pour déterminer les coordonnées du point d'intersection des droites, on peut résoudre le système formé par les deux équations cartésiennes.

- Pour trouver y :

$$\begin{aligned}\begin{cases} x+y=-3 & (E_1)\\ 6x-7y=-7 & (E_2)\end{cases} &\iff \begin{cases} -6x-6y=18 & (L_1)\leftarrow -6(E_1)\\ 6x-7y=-7 & (L_2)\leftarrow (E_2)\end{cases}\\ &\iff \begin{cases} x+y=-3 & (E_1)\\ -13y=11 & (L_1)+(L_2)\end{cases}\\ &\iff \begin{cases} x+y=-3\\ y=-\frac{11}{13}\end{cases}\end{aligned}$$

- Pour trouver x :

$$\begin{cases} x+\ \ y=-3 & (E_1) \\ 6x+7y=-7 & (E_2) \end{cases} \iff \begin{cases} 7x-7y=-21 & (L_1)\leftarrow 7(E_1) \\ 6x-7y=-7 & (L_2)\leftarrow (E_2) \end{cases}$$
$$\iff \begin{cases} x+y=-3 & (E_1) \\ 13x\ \ =-28 & (L_1)+(L_2) \end{cases}$$
$$\iff \begin{cases} x+y=-3 \\ x\ \ =-\frac{28}{13} \end{cases}$$

Finalement, le point d'intersection des droites a pour coordonnées $\left(-\frac{28}{13};-\frac{11}{13}\right)$.

2 $-\frac{28}{13}\approx -2,1538\ldots$ et $-\frac{11}{13}\approx -0,846154\ldots$, ce qui correspond aux valeurs renvoyées par la fonction.

On peut donc supposer que cette fonction renvoie les valeurs approchées des coordonnées du point d'intersection de deux droites.

Remarque 17

Cette fonction a en effet été écrite pour répondre à cette problématique. Elle permet de vérifier si deux droites sont strictement parallèles ou confondues (en calculant le déterminant de leur vecteur directeur) et , dans la négative, de calculer les coordonnées de leur point d'intersection à l'aide de formules que l'on appelle *formules de Cramer* (qui ne sont pas du tout au programme du lycée).

On peut utiliser cette fonction pour résoudre n'importe quel autre système de deux équations à deux inconnues. Par exemple, si on souhaite résoudre le système :

$$\begin{cases} 3x-5y=8 \\ 7x+3y=4 \end{cases}$$

il suffira d'écrire :

```
>>> D = (3,-5,-8)
>>> d = (7,3,-4)
>>> intersection(D,d)
(1.0, -1.0)
```

qui signifie que $(1\,;-1)$ est la solution au système.

Corrigé de l'exercice 7.26 page 76

1 La variable `d` est un 2-uplet, mais la première composante `d[0]` est aussi un 2-uplet : les coordonnées du vecteur. De même, `d[1]` est un 2-uplet : les coordonnées du point. Ainsi, l'abscisse du point A est la première composante de `d[1]` : c'est donc `d[1][0]`.
Le tableau suivant nous donne les différentes valeurs :

`d[0][0]`	`d[0][1]`	`d[1][0]`	`d[1][1]`
-3	2	1	-5

2 Avant d'écrire la fonction, il faut se demander à quelle condition le point mis en argument appartient à la droite.

Supposons, comme dans la question précédente, que l'on ait défini une droite (d) à partir d'un vecteur directeur $\vec{u}\begin{pmatrix}x\\y\end{pmatrix}$ et d'un point $A(x_A\,;y_A)$ par lequel elle passe.

Pour qu'un point $B\begin{pmatrix}x_B\\y_B\end{pmatrix}$ appartienne à (d), il faut que $\vec{u}$ et $\overrightarrow{AB}$ soient colinéaires, et donc que leur déterminant soit nul.

Cela donne lieu au code suivant :

Code Python 7-126

```
1  def appartient(d, B):
2      vecteur_AB = ( B[0]-d[1][0] , B[1]-d[1][1] )
3      vecteur_u = ( d[0][0] , d[0][1] )
4      determinant = vecteur_AB[0]*vecteur_u[1] -
       vecteur_AB[1]*vecteur_u[0]
5      return determinant == 0
```

- On commence par définir le vecteur $\overrightarrow{AB}$. `B[0]` est l'abscisse de B et `d[1][0]`, celle de A donc `B[0]-d[1][0]` représente $x_B - x_A$, c'est-à-dire l'abscisse du vecteur $\overrightarrow{AB}$. Il en est de même pour les ordonnées.
- Ensuite, on définit le vecteur directeur de la droite, en reprenant les données mises en argument de la fonction Python. On aurait pu s'en passer, mais j'ai préféré définir cette variable pour un peu plus de compréhension dans le calcul du déterminant à la ligne suivante.
- Le calcul du déterminant s'effectue à l'aide de la formule $\begin{vmatrix}x & x'\\y & y'\end{vmatrix} = xy' - yx'$.
- On finit par renvoyer le résultat du test « `determinant == 0` » : si le déterminant est nul, c'est-à-dire si l'égalité est vraie (donc si le point appartient à la droite), alors la fonction renvoie « True », sinon elle renvoie « False ».

3 Testons cette fonction avec les deux points donnés dans l'énoncé :

```
>>> B = (4, -7)
>>> appartient(d,B)
True
>>> C = (6, -8)
>>> appartient(d,C)
False
```

Le point B appartient bien à (d), mais pas le point C.

Corrigé de l'exercice 7.27 page 76

Les entiers de 0 à 10 sont représentés par `range(11)`. Pour obtenir leur image par la fonction $f(x) = 3x + 5$, on peut donc écrire :

Code Python 7-127

```
for x in range(11):
    print(3*x+5)
```

Corrigé de l'exercice 7.28 page 76

Il y a très peu de changements par rapport à la solution proposée dans l'exercice précédent :

Code Python 7-128

```
for x in range(-10,11):
    print(3*x+5)
```

On aurait aussi pu utiliser une boucle conditionnelle :

Code Python 7-129

```
x = -10
while x <= 10:
    print(3*x+5)
    x = x + 1
```

Corrigé de l'exercice 7.29 page 76

On ne peut pas ici utiliser la fonction `range` pour laquelle les nombres sont entiers.
On peut donc passer par une boucle conditionnelle :

Code Python 7-130

```
x = -1
while x <= 1:
    print(3*x+5)
    x = x + 0.1
```

Remarquons toutefois que les résultats obtenus sont quelques fois étranges :

```
2
2.3
2.5999999999999996
2.9
3.1999999999999997
3.4999999999999996
3.8
```

```
4.1
4.3999999999999995
4.699999999999999
5.0
5.3
5.6
5.8999999999999995
```

```
6.199999999999999
6.5
6.8
7.1
7.3999999999999995
7.699999999999999
7.999999999999999
```

À chaque fois qu'un résultat est affiché avec une partie décimale comportant beaucoup de « 9 », cela signifie qu'il y a une erreur d'approximation (due à la façon donc Python représente les nombres flottants, donc avec une virgule).

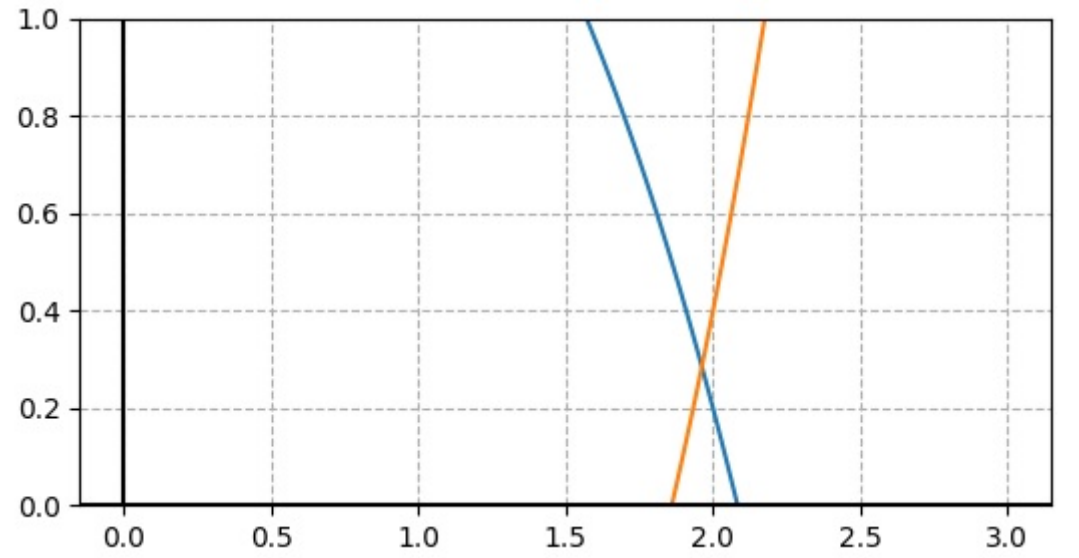
Corrigé de l'exercice 7.30 page 77

Je propose le code suivant :

Code Python 7-131

```
1  from matplotlib.pyplot import plot, show, grid, subplots, axhline,
       axvline
2  from numpy import linspace
3
4  x = linspace(1, 3, 100)
5  y = -0.2*x**3 + 0.1*x**2- 0.3*x + 2
6  z = 0.3*x**3 - 0.5*x - 1
7
8  fig, ax = subplots(figsize=(2, 1))
9  ax.set_ylim(0,1)
10 plot(x,y)
11 plot(x,z)
12 grid(linestyle='--')
13
14 axhline(y=0, color='black', linestyle='-')
15 axvline(x=0, color='black', linestyle='-')
16
17 show()
```

qui produit :



J'ai ici changé les nombres à la ligne 4 pour signifier que je ne voulais trouver les images que des valeurs comprises entre 1 et 3 (avec un pas de 1 centième), puis la ligne 9 pour spécifier que je ne voulais tracer les deux courbes que pour des y de 0 à 1.
Bien sûr, vous pouvez mettre les nombres que vous voulez.

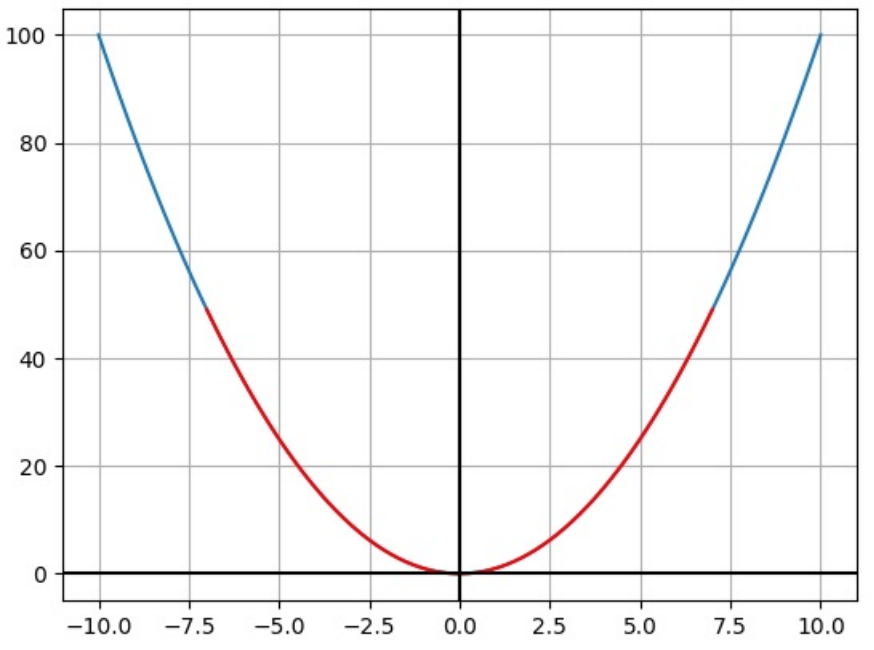
Corrigé de l'exercice 7.31 page 78

1 La partie de la courbe telle que $x^2 \leqslant 49$ est la partie de la courbe telle que $-7 \leqslant x \leqslant 7$, d'où le programme suivant :

Code Python 7-132

```
from matplotlib.pyplot import plot, show, grid, axhline, axvline
from numpy import linspace

x = linspace(-10, 10, 100)
y = x**2

plot(x,y)
grid()

# partie de la courbe telle que x*x < 49

x = linspace(-7, 7, 100)
y = x**2

plot(x,y,color='red')

# fin de la représentation de la partie de la courbe

axhline(y=0, color='black', linestyle='-')
axvline(x=0, color='black', linestyle='-')

show()
```

qui affiche :

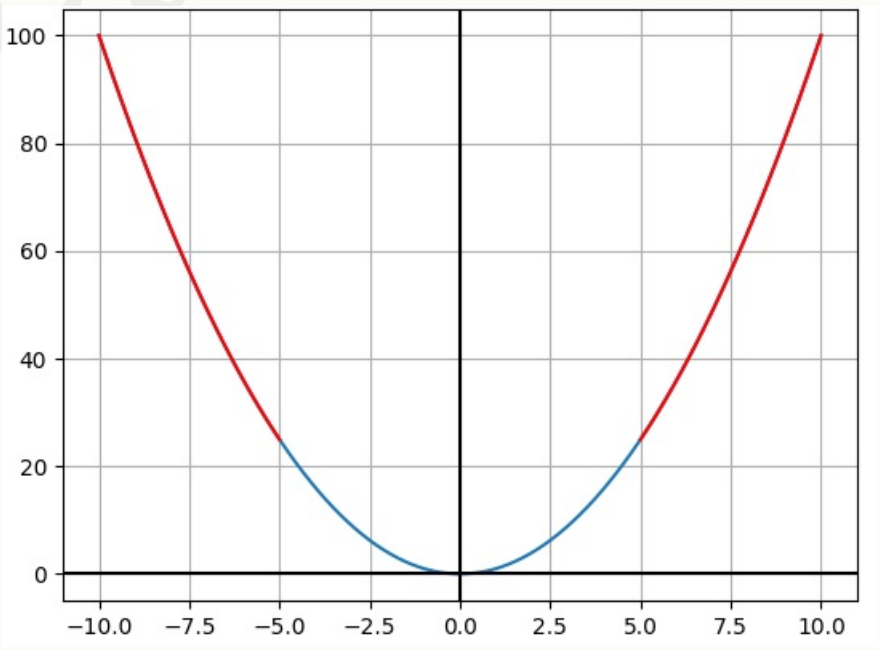


2 La partie de la courbe telle que $x^2 \geqslant 25$ est la partie de la courbe telle que $x \leqslant -5$ et $x \geqslant 5$, d'où le programme suivant :

Code Python 7-133

```
from matplotlib.pyplot import plot, show, grid, axhline, axvline
from numpy import linspace

x = linspace(-10, 10, 100)
y = x**2

plot(x,y)
grid()

# partie de la courbe telle que x*x > 25

x1 = linspace(-10, -5, 100)
y1 = x1**2
x2 = linspace(5, 10, 100)
y2 = x2**2
plot(x1,y1,color='red')
plot(x2,y2,color='red')

# fin de la représentation de la partie de la courbe

axhline(y=0, color='black', linestyle='-')
axvline(x=0, color='black', linestyle='-')
show()
```

qui affiche :



Corrigé de l'exercice 7.32 page 79

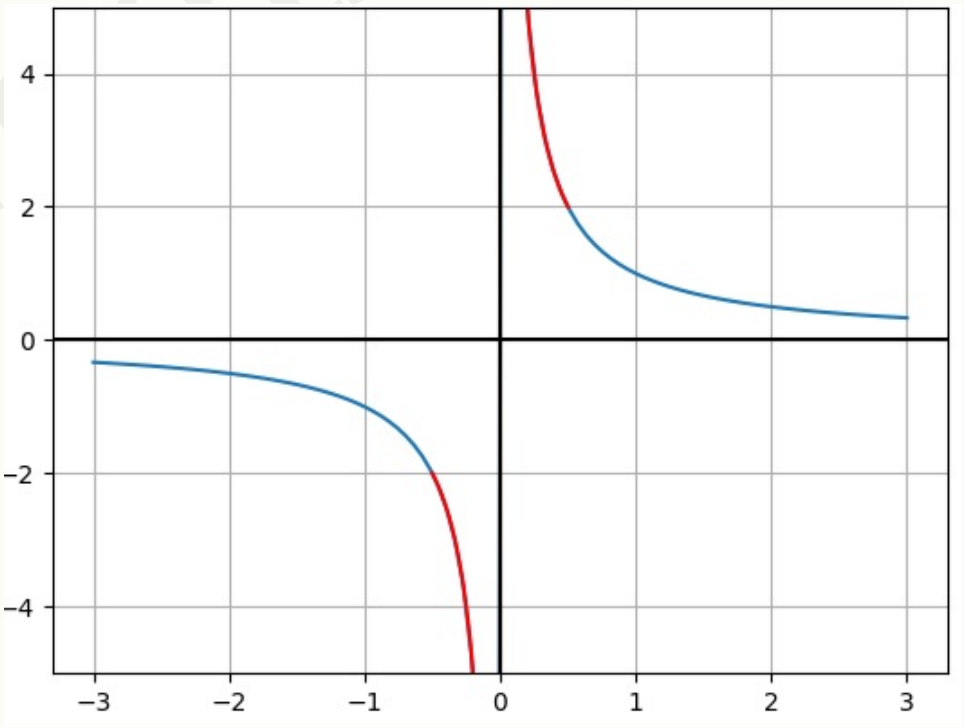
Pour le tracé de la fonction inverse, je choisis de limiter l'affichage en y (car quand on est proche de 0, $\frac{1}{x}$ est très grand dans les positifs comme dans les négatifs).
Pour cela, je fais appel à la fonction `ylim(a,b)` de sorte que les images affichées soient entre a et b.

Pour que $-2 < \dfrac{1}{x} < 2$, il faut que $-\dfrac{1}{2} < x < \frac{1}{2}$.

Code Python 7-134

```
1  from matplotlib.pyplot import plot, show, grid, axhline, axvline, ylim
2  from numpy import linspace
3
4  x = linspace(-3, 3, 100)
5  y = 1/x
6  plot(x, y)
7  grid()
8
9  x = linspace(-0.5, 0.5, 100)
10 y = 1/x
11 plot(x, y, color='red')
12
13 # Limiter l'affichage des y entre -5 et 5
14 ylim(-5, 5)
15
16 axhline(y=0, color='black', linestyle='-')
17 axvline(x=0, color='black', linestyle='-')
18 show()
```

qui affiche :



Corrigé de l'exercice 7.33 page 79

1 Le programme Python correspondant à l'algorithme est par exemple le suivant :

Code Python 7-135

```
L = [ 3,5,1 ]
for i in range(len(L)):
    m = L[i]
    indice = i
    for j in range(i+1,len(L)):
        if L[j] < m:
            m = L[j]
            indice = j
    a = L[i]
    L[i] = L[indice]
    L[indice] = a

print(L)
```

`len(L)` désigne la longueur de la liste `L`.

2 Le programme affiche la liste :

```
[1,3,5]
```

Il semble donc trier dans l'ordre croissant les données de la liste entrée.

3 Un programme possible est :

Code Python 7-136

```
L = [ 8 , 6 , 2 , 2 , 9 , 2 , 7 , 10 ]
for i in range(len(L)):
    m = L[i]
    indice = i
    for j in range(i+1,len(L)):
        if L[j] < m:
            m = L[j]
            indice = j
    a = L[i]
    L[i] = L[indice]
    L[indice] = a

if len(L) % 2 == 1:
    mediane = L[ len(L) // 2 ]
else:
    mediane = ( L[ len(L) // 2 ] + L[ len(L) // 2 + 1 ] ) / 2

print(L)
print('Médiane : ',mediane)
```

Le début reste inchangé (lignes 1 à 11). Ensuite, pour calculer la médiane, on regarde si la longueur de la liste est paire ou impaire : si elle est impaire (`if len(L)%2 == 1`) alors la médiane est l'élément central de la liste ordonnée (`mediane = L[ len(L) // 2 ]`). Notons ici que j'ai utilisé la division entière pour trouver l'indice du terme central (car un indice ne peut être qu'entier, et faire `len(L)/2` retourne un nombre réel, et non entier).

Si au contraire la liste compte un nombre pair d'éléments, alors la médiane est la moyenne des éléments centraux, ceux-ci étant `L[len(L)//2]` et `L[len(L)//2+ 1]`.

Corrigé de l'exercice 7.34 page 80

1 Le tableau complété est le suivant :

x	-	(1,2)	(3,4)
somme_produits	0	$0+1\times 2=2$	$2+3\times 4=14$
effectif_total	0	$0+2=2$	$2+4=6$

Ainsi, la valeur renvoyée est égale à $14\div 6\approx 2.333\ldots$

2 À l'aide de l'exemple précédent, on s'aperçoit que pour une série statistique $(x_i;n_i)$, on ajoute dans la boucle tous les $x_i\times n_i$ (en effet, `x[0]` correspond à x_i et `x[1]` correspond à n_i).

De plus, on ajoute aussi tous les effectifs (ligne 7).

Ainsi, lorsque la boucle est finie, on divise la somme des $x_i\times n_i$ par la somme des effectifs, ce qui correspond à la formule de la moyenne :

$$\overline{x}=\frac{1}{\text{N}}\sum_i x_i\times n_i,$$

où N est l'effectif total.

La fonction Python renvoie donc la moyenne de la série mise en argument.

3 La variance est la moyenne des $(x_i-\overline{x})^2$. Une fonction Python renvoyant la variance est donc très similaire à la fonction précédente :

Code Python 7-137

```
1  def var(L):
2      m = moyenne(L)
3      s = 0
4      effectif_total = 0
5      for x in L:
6          s = s + ( x[0] - m )**2
7          effectif_total = effectif_total + x[1]
8
9      return s / effectif_total
```

Corrigé de l'exercice 7.35 page 80

1 La fonction complétée est la suivante :

Code Python 7-138

```
1  def ecc(L):
2      # L est une liste de la forme [ (x1,n1) , (x2,n2) , ... ] où
       les ni sont les effectifs
3      E = []
4      for i in range( len( L ) ):
5          if i == 0:
6              E.append( L[i] )
7          else:
8              E.append( ( L[i][0] , E[i-1][1] + L[i][1] ) )
9
10     return E
```

Explications :

- Ligne 4 : la boucle signifie ici que l'on parcourt la liste `L` à l'aide des indices (`i`) de ses éléments.
- Ligne 5 : si l'indice est égal à 0, c'est-à-dire si l'on est sur le premier élément de la liste, ...
- Ligne 6 : on insère dans la liste `E` le premier effectif (`L[0]` représente le premier couple de nombres de la liste `L`, donc `L[0][1]` représente le deuxième nombre dans ce couple de nombres).
- Ligne 7 : si nous ne sommes pas sur le premier élément de `L`...
- Ligne 8 : on prend le deuxième nombre de l'élément précédent dans `E` (qui est un couple) et on lui ajoute l'effectif de l'élément de `L` sur lequel nous sommes.

Regardons sur l'exemple où `L = [ (1,2) , (3,4) , (5,6) ]` :

i	-	0	1	2
L[i]	-	(1,2)	(3,4)	(5,6)
L[i][1]	-	2	4	6
E[i][1]	0	$0+2=2$	$2+4=6$	$6+6=12$

Au final, `E = [ (1,2), (3,6) , (5,12)]`.

2 La fonction complétée est celle présentée page suivante.

On retrouve à la ligne 5 le test pour savoir si l'effectif total est pair (`N % 2 == 0`, auquel cas on le divise par 2 (ligne 6), ou impair (ligne 7), auquel cas on ajoute « 1 » à l'effectif total et on divise par 2.

Ensuite, on parcourt la liste `E` (qui contient les effectifs cumulés croissants, abrégés en e.c.c.) : si l'e.c.c. dépasse la moitié de l'effectif total, c'est que l'on a atteint la médiane et on la renvoie. Si, au contraire, l'e.c.c. est exactement égal à la moitié alors on retourne la moyenne entre la valeur courante (`E[i][0]`) et la suivante (`E[i+1][0]`).

Code Python 7-139

```
1  def mediane(L):
2      E = ecc(L)
3      N = E[-1][1] # effectif total
4
5      if N % 2 == 0:
6          moitie = N / 2 # moitié de l'effectif total
7      else:
8          moitie = (N +1) / 2
9
10     for i in range( len(E) ):
11         if E[i][1] > moitie:
12             return E[i][0]
13         if E[i][1] == moitie:
14             return ( E[i][0] + E[i+1][0] ) / 2
```

Corrigé de l'exercice 7.36 page 81

L'instruction `randint(1,6)` choisit un nombre entier entre 1 et 6.
La ligne 7 affecte à la valeur `N[n-1]` la valeur qu'il y avait, augmentée de 1.
Prenons un exemple : si « 3 » est choisi lors du premier passage dans la boucle, alors `N[3-1]`, c'est-à-dire `N[2]` vaudra 0 + 1 au premier passage dans la boucle.
Ainsi, à l'issue de la boucle, la liste `N` contiendra le nombre de fois que que l'on aura obtenu les nombres entiers de 1 à 6.
On peut alors imaginer que ce programme simule 1 000 lancers d'un dé à six faces.
Voici quelques exemples de simulations à l'aide de ce programme :

```
[177, 151, 159, 179, 168, 166]
[157, 176, 155, 160, 174, 178]
[189, 156, 166, 179, 159, 151]
[163, 160, 161, 182, 168, 166]
[163, 164, 152, 183, 170, 168]
[169, 163, 170, 148, 179, 171]
```

Corrigé de l'exercice 7.37 page 82

Réflexions préliminaires :

- *Comment choisir au hasard la direction du kangourou ?*
  Dès qu'il s'agit de faire un choix au hasard, il faut penser au module `random`. Ici, nous avons quatre directions possibles, que l'on peut modéliser par les nombres 1, 2, 3 et 4 (par exemple), avec la fonction `randint(1,4)`.
- *Tenir compte du nombre de secondes.*
  n représente le nombre de secondes, donc le nombre de fois que l'on devra faire un choix aléatoire de direction. Il faudra utiliser une boucle itérative « for ».

Une fonction possible est alors :

Code Python 7-140

```
from random import randint

def trajet(n):
    x, y = 0, 0 # position initiale du kangourou
    for k in range(n):
        direction = randint(1,4) # 1='droite', 2='gauche', 3='haut',
    4='bas'
        if direction == 1:
            x = x + 1
        elif direction == 2:
            x = x - 1
        elif direction == 3:
            y = y + 1
        else:
            y = y - 1

    return x, y
```

On a par exemple :

```
>>> trajet(20)
(2, -4)
```

Corrigé de l'exercice 7.38 page 82

L'issue d'un lancer de dé cubique peut être simulée à l'aide de la fonction `randint(1,6)` du module `random`.
De même, l'issue d'un lancer de dé tétraédrique peut être simulée à l'aide de la fonction `randint(1,4)`.
Pour effectuer n simulations, on doit faire une boucle itérative « for ». Cela donne par exemple :

Code Python 7-141

```
from random import randint

def simulation(n):
    somme = 0
    for k in range(n):
        a = randint(1,6)
        b = randint(1,4)
        somme = somme + a + b # somme précédente + résultat de
    l'expérience (a+b)

    return somme / n
```

L'idée ici est de définir une variable `somme` qui contiendra la somme des résultats de toutes les expériences. Pour avoir la somme moyenne, il suffit de diviser cette somme totale par le nombre de simulations.
On obtient :

```
>>> for i in range(5):
      print( simulation(1000) )

5.77
6.025
6.033
5.934
6.006
```

Sur 5 simulations de 1 000 expériences, on s'aperçoit que la somme moyenne est d'environ 6.

Corrigé de l'exercice 7.39 page 82

Une fonction possible est la suivante :

Code Python 7-142

```
1 from random import randint
2
3 def pq(n):
4     feuilles = 500
5     for k in range(n):
6         nombre = randint(5,15)
7         feuilles = feuilles - nombre
8
9     return feuilles
```

La variable `feuilles` contiendra le nombre de feuilles restantes.
Ainsi, au début, elle vaut 500 (car il y a 500 feuilles par rouleau). Ensuite, on simule le passage de n personnes à l'aide d'une boucle itérative (n passages dans la boucle : n itérations). À chaque itération, `nombre` représente le nombre de feuilles utilisées (soit un nombre aléatoire entre 5 et 15). On soustrait donc au nombre de feuilles restantes ce nombre.
À la fin de la boucle, la fonction renvoie la valeur stockée dans `feuilles` qui correspond bien au nombre de feuilles restantes après le passage de n personnes.
On a par exemple :

```
>>> pq(20)
308
>>> pq(50)
-49
>>> pq(50)
-2
```

8

Python en classe de Première

Plan du chapitre

I Suites numériques 112
1 Calcul et affichage des premiers termes 112
2 Algorithme de seuil 112
3 Méthode de Newton 114
II Fonction exponentielle : méthode d'Euler 114
III Fonctions trigonométriques 117
IV Vecteurs 118
1 Déterminant de deux vecteurs 118
2 Produit scalaire 119
Enoncés 120
Corrigés des exercices 129

Extrait du programme de Première

La démarche algorithmique est, depuis les origines, une composante essentielle de l'activité mathématique. Au collège, en mathématiques et en technologie, les élèves ont appris à écrire, mettre au point et exécuter un programme simple. La classe de seconde a permis de consolider les acquis du cycle 4 autour de deux idées essentielles :

- la notion de fonction ;
- la programmation comme production d'un texte dans un langage informatique.

L'enseignement de spécialité de mathématiques de classe de première vise la consolidation des notions de variable, d'instruction conditionnelle et de boucle ainsi que l'utilisation des fonctions. La seule notion nouvelle est celle de liste qui trouve naturellement sa place dans de nombreuses parties du programme et aide à la compréhension de notions mathématiques telles que les suites numériques, les tableaux de valeurs, les séries statistiques...
Comme en classe de seconde, les algorithmes peuvent être écrits en langage naturel ou utiliser le langage Python.
Les notions relatives aux types de variables et à l'affectation sont consolidées. Comme en classe de seconde, on utilise le symbole « ← » pour désigner l'affection dans un algorithme écrit en langage naturel.
L'accent est mis sur la programmation modulaire qui permet de découper une tâche complexe en tâches plus simples.

Attention 6

Certains aspects qui interviennent en classe de Seconde ne seront pas systématiquement repris dans ce chapitre.
Il est donc conseillé de regarder les chapitres précédents avant d'aborder celui-ci.

I - Suites numériques

I . 1 - Calcul et affichage des premiers termes

Une suite numérique peut être définie de façon explicite (à l'aide d'une fonction dont la variable est l'indice du terme à calculer) ou par une relation de récurrence (qui lie le terme à calculer à son précédent).

- Si la suite est définie de façon explicite, alors le programme Python qui permet de calculer ses premiers termes est semblable à celui qui permet de calculer des images par une fonction.

Exemple 34

Programme calculant et affichant les 20 premiers termes de la suite (u_n) définie pour tout entier naturel n par :

$$u_n = n^2 + \frac{1}{n+1}.$$

Code Python 8-143

```
1 for n in range(21):
2     print( n*n + 1/(n+1) )
```

- Si la suite est définie par récurrence, on utilise cette relation de récurrence pour calculer les termes successifs.

Exemple 35

Programme calculant et affichant les 20 premiers termes de la suite (u_n) définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 5 \\ u_{n+1} = u_n + 3 \end{cases}$$

Code Python 8-144

```
1 u = 5 # premier terme
2 print(u) # on affiche le premier terme
3 for n in range(19): # on calcule et affiche les 19 autres termes
4     u = u + 3 # relation de récurrence
5     print(u)
```

I . 2 - Algorithme de seuil

Un *algorithme de seuil* est un algorithme (ou un programme) qui permet de trouver le premier indice à partir duquel une condition sur la suite est remplie (par exemple, le premier indice n à partir duquel $u_n > 100$ ou $u_n \leqslant 0,1$).
Dans la mesure où on cherche un indice et qu'une condition est imposée, on devra utiliser une boucle conditionnelle « `while` » (car on ne sait pas le nombre d'itérations à exécuter pour y arriver).

En général, l'algorithme se présente ainsi :

```
u ← premier terme de la suite
n ← 0
Tant que < contraire de la condition imposée >:
  n ← n + 1
  u ← relation de récurrence de u
Afficher n
```

Exemple 36

Soit (u_n) la suite définie pour tout entier naturel n par : $\begin{cases} u_0 = 50 \\ u_{n+1} = 0,95u_n \end{cases}$.

On cherche le premier indice à partir duquel $u_n \leqslant 10^{-5}$.

Code Python 8-145

```
1 u = 50
2 n = 0
3 while u > 10**(-5):
4     n = n + 1
5     u = 0.95 * u
6
7 print(n)
```

```
301
```

Donc $u_{300} > 10^{-5}$ mais $u_{301} \leqslant 10^{-5}$.

Exemple 37

Soit (u_n) la suite définie pour tout entier naturel n par : $\begin{cases} u_0 = 10 \\ u_{n+1} = 1,2u_n \end{cases}$

On cherche le premier indice à partir duquel $u_n > 10^9$.

Code Python 8-146

```
1 u = 10
2 n = 0
3 while u <= 10**9:
4     n = n + 1
5     u = 1.2 * u
6
7 print(n)
```

```
102
```

Donc $u_{101} \leqslant 10^9$ mais $u_{102} > 10^9$.

I . 3 - Méthode de Newton

Cette méthode est explicitée dans le livre concernant l'enseignement de spécialité en Mathématiques. Elle définit une suite (u_n) de premier terme u_0 et dont la relation de récurrence est :

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}.$$

Il existe une façon élégante d'implémenter cette méthode à l'aide du module *sympy* :

Code Python 8-147

```
import sympy as sp

x = sp.symbols('x')

def newton(f,x0,eps=0.0001):
    f_prime = sp.diff(f, x)
    xn= x0+1
    while abs(xn-x0) > eps:
        x0, xn = xn, xn - f.subs(x,xn)/f_prime.subs(x,xn)

    return xn

# Définir la fonction
f = 0.25*x**2 + 0.4*x - 3

# On affiche la solution
print(newton(f,0))
```

Ici, on a choisi de prendre $u_0 = 0$. La suite converge rapidement vers la solution de l'équation $f(x) = 0$, où $f(x) = 0,25x^2 + 0,4x - 3$.
L'avantage de cette méthode d'implémentation réside dans le fait qu'il n'est pas nécessaire de déterminer l'expression de la dérivée.

II - Fonction exponentielle : méthode d'Euler

On recherche une fonction f telle que,

1. pour tout réel x, $f'(x) = f(x)$;
2. $f(0) = 1$.

On suppose que cette fonction existe.

Très vite, on se rend compte que cette fonction n'est pas un polynôme car le seul polynôme P qui vérifie le point 1 est $P(x) = 0$ et dans ce cas, $P(0) \neq 1$, ce qui ne satisfait pas le point 2.
Ce n'est pas non plus une fonction racine carrée ou une fraction rationnelle (quotient de deux polynômes).

Nous allons avant tout tenter de dessiner la courbe représentative de notre fonction. Pour cela, on va prendre des intervalles $[a;a+h]$, avec h très proche de 0, et on va assimiler la courbe représentative de f à sa tangente au point d'abscisse a. En effet, sur un intervalle assez petit $[a;a+h]$, la tangente est très proche de la courbe.

On sait que l'équation de la tangente à la courbe au point d'abscisse a est :

$$y = f'(a)(x-a) + f(a).$$

Posons $g(x) = f'(a)(x-a) + f(a)$. Confondre la courbe et la tangente revient à dire que $f(a+h) \approx g(a+h)$, c'est-à-dire :

$$f(a+h) \approx f'(a)(a+h-a) + f(a),$$

soit :

$$f(a+h) \approx f'(a) \times h + f(a).$$

Or, d'après le point 1, $f'(a) = f(a)$ donc :

$$f(a+h) \approx hf(a) + f(a)$$

soit :

$$f(a+h) \approx (1+h)f(a). \qquad \text{(en factorisant par } f(a))$$

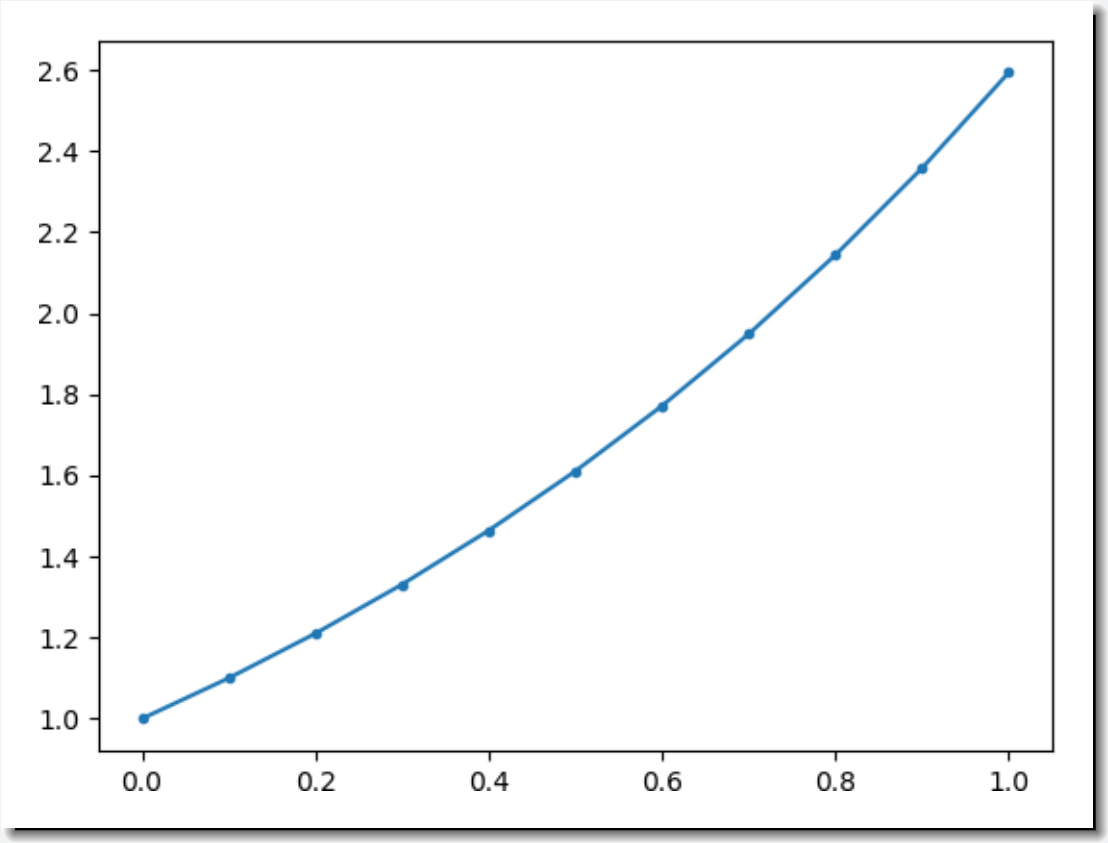
h peut être pris aussi petit qu'on le souhaite. Prenons par exemple $h = 0,1$. Alors,

- en prenant $a = 0$: $f(0+0,1) \approx (1+0,1) \times f(0) = 1,1 \times 1 = 1,1$;
- en prenant $a = 0,1$: $f(0,2) = f(0,1+0,1) \approx (1+0,1) \times f(0,1) = 1,1 \times 1,1 = 1,21$;
- en prenant $a = 0,2$: $f(0,3) = f(0,2+0,1) \approx (1+0,1) \times f(0,2) = 1,1 \times 1,21 = 1,331$; etc.

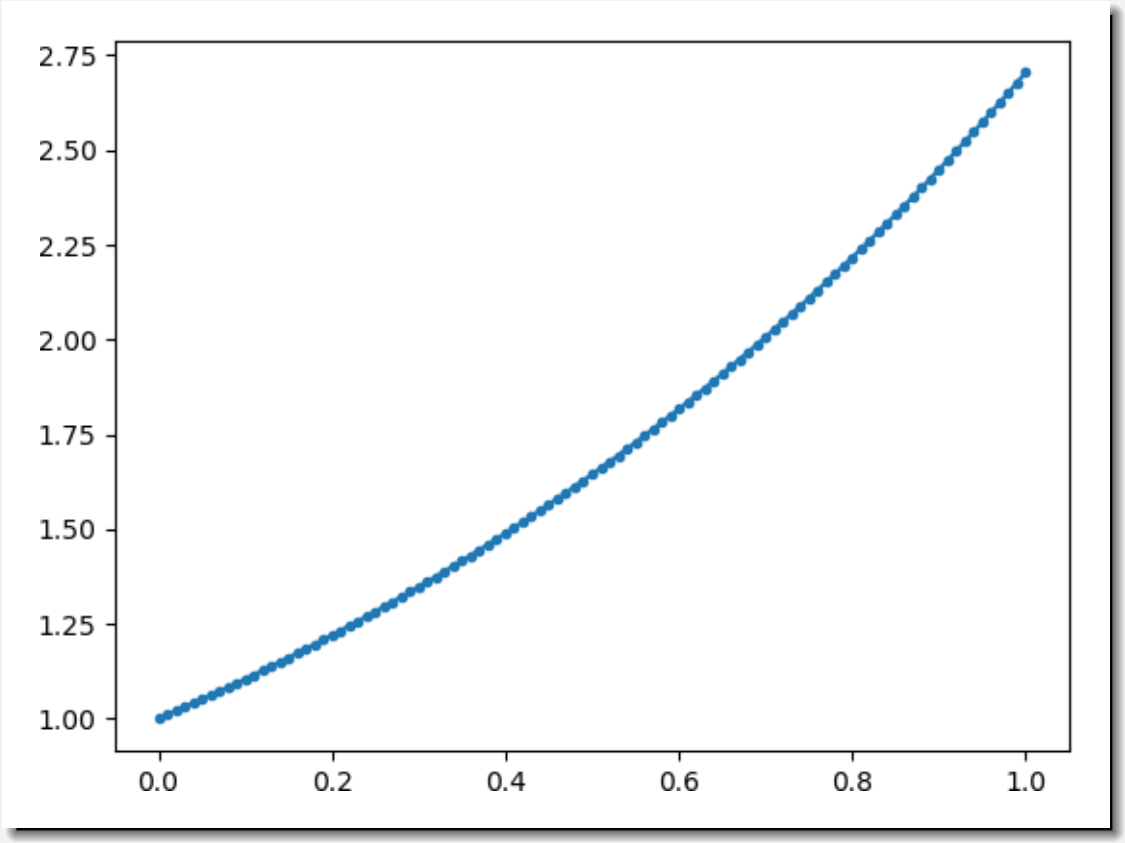
Le programme Python suivant permet d'automatiser la construction des segments :

Code Python 8-148

```
from matplotlib.pyplot import plot, show
from numpy import array

xList = [0]
yList = [1]

a = 0
h = 0.1

for i in range(10):
    x = a + h
    y = (1+h) * yList[i]
    xList.extend([x])
    yList.extend([y])
    a = x

x = array(xList)
y = array(yList)
plot(x, y, marker=".")

show()
```



Avec un pas plus petit ($h = 0,01$), on obtient la figure suivante :



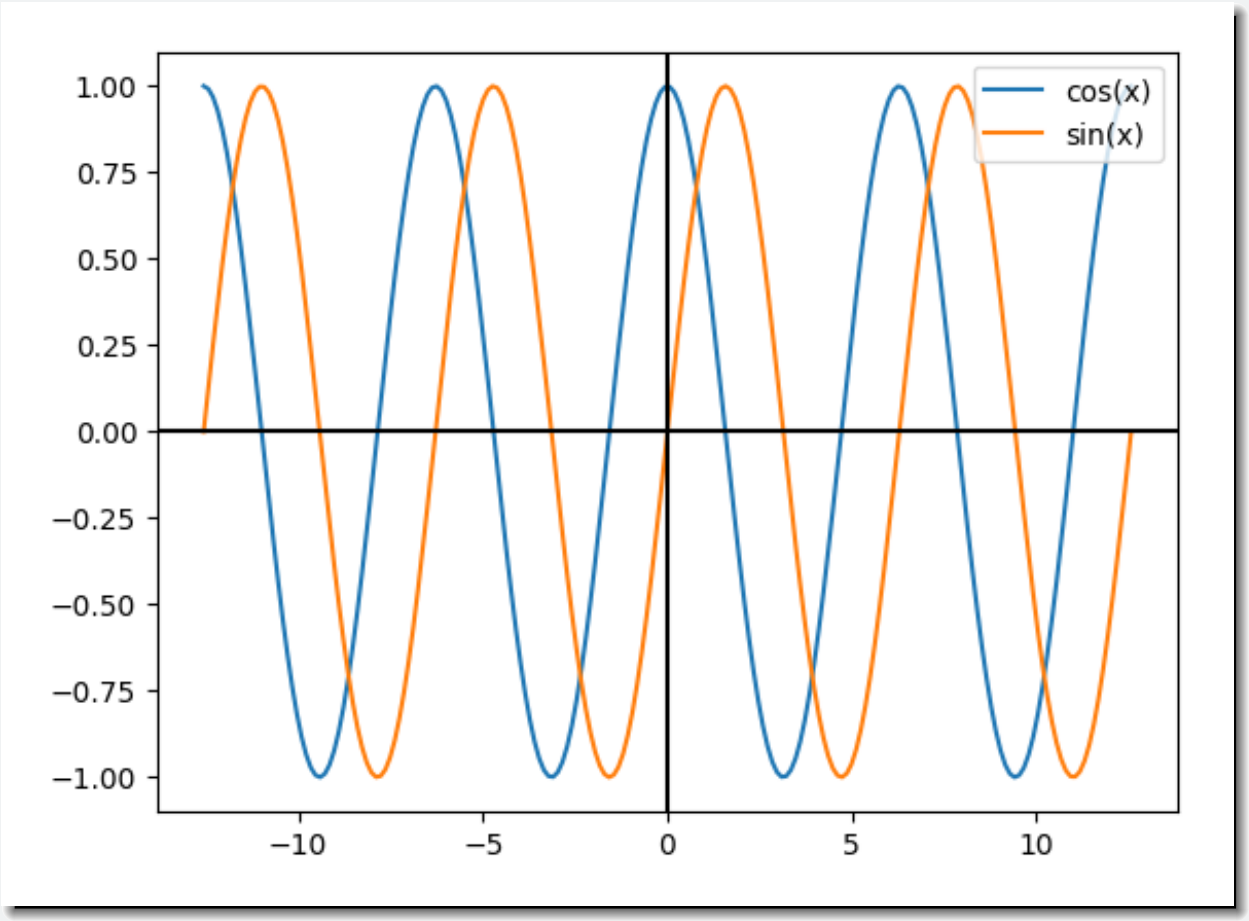
On voit alors se dessiner la courbe représentative de la fonction f cherchée.

III - Fonctions trigonométriques

Il n'y a pas grand-chose à écrire sur ce thème, si ce n'est peut-être un programme permettant de tracer les représentations des fonctions sinus et cosinus.

Code Python 8-149

```
1  from numpy import linspace, cos, sin, pi
2  from matplotlib.pyplot import plot, show, legend, axvline, axhline
3
4  x = linspace(-4*pi, 4*pi, 200)
5  y1 = cos(x)
6  y2 = sin(x)
7  plot(x, y1, label='cos(x)')
8  plot(x, y2, label='sin(x)')
9  legend()
10 axhline(y=0, color='black', linestyle='-')
11 axvline(x=0, color='black', linestyle='-')
12 show()
```



IV - Vecteurs

À l'instar des points, les vecteurs peuvent être représentés par un couple de coordonnées.

Exemple 38

Le vecteur $\vec{u}\begin{pmatrix}-1\\3\end{pmatrix}$ peut être représenté ainsi :

```
>>> u = (-1,3)
>>> u[0]
-1
>>> u[1]
3
```

C'est une sorte de tableau à une ligne et deux colonnes dans lequel sont mises les coordonnées du vecteur.
Le premier nombre est indexé par 0 (`u[0]`) et le second par 1 (`u[1]`). Mais nous avons vu cela dans le chapitre précédent.

IV . 1 - Déterminant de deux vecteurs

Le module `numpy` permet de calculer le déterminant de deux vecteurs :

Code Python 8-150

```
1  from numpy import array
2  from numpy.linalg import det
3
4  def determinant(u,v):
5      # u = (a, b) et v = (c, d)
6      vecteur1 = array([u[0], u[1]])
7      vecteur2 = array([v[0], v[1]])
8
9      matrice = array([vecteur1, vecteur2])
10
11     return det(matrice)
```

```
>>> u = (-3, 2)
>>> v = (5, -1)
determinant(u,v)
-6.999999999999999
```

IV . 2 - Produit scalaire

À l'instar du déterminant, le produit scalaire peut aussi être calculé à l'aide du module `numpy` :

Code Python 8-151

```
from numpy import array

def produit_scalaire(u,v):
    # u = (a, b) et v = (c, d)
    vecteur1 = array([u[0], u[1]])
    vecteur2 = array([v[0], v[1]])

    return vecteur1 @ vecteur2
```

```
>>> u = (-3, 2)
>>> v = (5, -1)
produit_scalaire(u,v)
-17
```

Remarque 18

On peut aussi utiliser la fonction `np.dot(vecteur1, vecteur2)` :

Code Python 8-152

```
from numpy import array, dot

def produit_scalaire(u,v):
    # u = (a, b) et v = (c, d)
    vecteur1 = array([u[0], u[1]])
    vecteur2 = array([v[0], v[1]])

    return dot(vecteur1, vecteur2)
```

Le second degré

Exercice 8.1

1. Écrire une fonction Python `delta(a,b,c)` qui renvoie le discriminant du polynôme ax^2+bx+c.
2. Écrire une fonction Python `racines(a,b,c)` qui renvoie les éventuelles racines du polynôme ax^2+bx+c.
   On pourra s'aider de la fonction `delta(a,b,c)`.

Solution page 129

Exercice 8.2

Écrire une fonction Python qui retourne les deux nombres a et b tels que $a+b=\text{S}$ et $ab=\text{P}$, où S et P sont connus.

Solution page 129

Dérivation

Exercice 8.3

On considère le programme Python suivant :

Code Python 8-155

```
1 def f(x):
2     return -0.3*x*x + 2*x + 1
3
4 def coef(a):
5     return ( f(a+10**(-10)) - f(a) ) / 10**(-10)
```

On a alors en mode console :

```
>>> coef(1)
1.4000001158365194
```

1. Expliquer ce que représente le nombre renvoyé par la fonction `coef(a)`.
2. En théorie, que devrait renvoyer `coef(1)` ?
   Comparer avec le résultat affiché.

Solution page 130

Suites

Exercice 8.4 (vérifier la nature d'une suite)

1 On considère la liste `u` définie par :

```
u = [ 5 , 8 , 11 , 14 , 17 , 20 , 23 , 26 , 29 , 31 , 34 , 37 , 40 ,
    43 ]
```

Écrire un programme qui vérifie si cette liste contient les premiers termes d'une suite arithmétique.

2 On considère une liste `v` définie par :

```
v = [ 100 , 50 , 25 , 12.5 , 6.25 , 3.125 , 1.5625 , 0.78125 ,
    0.375625 , 0.1878125 , 0.09390625 , 0.046953125 ]
```

Écrire un programme qui vérifie si cette liste contient les premiers termes d'une suite géométrique.

Solution page 130

Exercice 8.5 (calcul du n-ième terme)

Écrire une fonction Python `u(n)` qui renvoie le n-ième terme de la suite définie par :

$$\begin{cases} u_0 = 7 \\ u_{n+1} = u_n + 3,27 \end{cases}$$

Solution page 131

Exercice 8.6 (calcul du n-ième terme)

Écrire une fonction `u(n)` qui renvoie le n-ième terme de la suite définie par :

$$\begin{cases} u_0 = 7 \\ u_{n+1} = 0,3u_n \end{cases}$$

Solution page 133

Exercice 8.7 (calcul du n-ième terme)

Écrire une fonction `u(n)` qui renvoie le n-ième terme de la suite définie par :

$$\begin{cases} u_0 = 7 \\ u_{n+1} = 0.5u_n + 2 \end{cases}$$

Solution page 134

Exercice 8.8 (algorithme de seuil)

On considère la suite (u_n) définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 9 \\ u_{n+1} = 1,2u_n - 1 \end{cases}$$

Compléter le programme suivant afin qu'il affiche la première valeur de n pour laquelle $u_n \geqslant 1\,000$.

Code Python 8-167

```
u = ...
n = ...
while u < ...:
    n = ...
    u = ...

print(...)
```

Solution page 135

Exercice 8.9 (algorithme de seuil)

On considère la suite (u_n) définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 95 \\ u_{n+1} = 0.95u_n + 1 \end{cases}$$

Compléter le programme suivant afin qu'il affiche la première valeur de n pour laquelle $u_n < 20 + 10^{-9}$.

Code Python 8-169

```
u = ...
n = ...
while ...:
    n = ...
    u = ...

print(...)
```

Solution page 135

Exercice 8.10 (algorithme de seuil)

Écrire un programme Python qui détermine le premier entier n tel que $u_n < 4,0001$, où (u_n) est la suite définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 7 \\ u_{n+1} = 0.5u_n + 2 \end{cases}$$

Solution page 136

Exercice 8.11 (algorithme de seuil)

On considère la suite définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 2 \\ u_{n+1} = 1,05u_n + 1 \end{cases}$$

Écrire un programme Python permettant d'afficher le premier entier n à partir duquel $u_n \geqslant 10^9$.

Solution page 136

Exercice 8.12 (algorithme de seuil)

On considère le programme Python suivant :

Code Python 8-173

```
u = -8
s = -8
n = 0

while s < 10**9:
    n = n + 1
    u = u + 1.5
    s = s + u

print(n)
```

1. Expliquer ce que représente la valeur affichée par ce programme.
2. Retrouver cette valeur par le calcul.

Solution page 137

Exercice 8.13 (somme)

On considère la suite (u_n) définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 5 \\ u_{n+1} = 0,3u_n \end{cases}$$

Écrire un programme Python permettant de calculer la somme :

$$u_0 + u_1 + \cdots + u_{100}.$$

Solution page 138

Exercice 8.14 (somme)

Écrire un programme Python permettant de calculer la somme des 1 000 premiers termes de la suite (u_n) définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 5 \\ u_{n+1} = \frac{1}{2}u_n + 2n - 3 \end{cases}$$

Solution page 139

Exercice 8.15

On considère le programme suivant :

Code Python 8-177

```
def somme(n):
    s = 0
    for k in range(1,n+1):
        s = s + 1 / k**2

    return s
```

1. Que renvoie cette fonction ?
2. On obtient en mode console :

```
>>> somme(1000)
1.6439345666815615
>>> somme(10000)
1.6448340718480652
>>> somme(10**7)
1.6449339668472596
```

Quelle conjecture peut-on faire à la vue de ces résultats ?

Solution page 140

Exercice 8.16

1 Écrire un programme Python permettant de calculer la somme :

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\cdots+\frac{1}{2021}=\sum_{p=0}^{1010}\frac{(-1)^p}{2p+1}.$$

2 Écrire un programme en Python permettant de calculer la somme :

$$\sum_{p=1}^{2021}\frac{1}{3p-1}.$$

Solution page 140

Produit scalaire

Exercice 8.17

Dans cet exercice, on n'utilisera pas le module `numpy`.

1 Écrire une fonction Python `produit(u,v)` qui renvoie le produit scalaire de deux vecteurs $\vec{u}\begin{pmatrix}x\\y\end{pmatrix}$ et $\vec{v}\begin{pmatrix}x'\\y'\end{pmatrix}$, représentés respectivement par les variables `u = (x , y)` et `v = (x' , y')`.

2 Écrire une fonction Python `isOrth(u,v)` qui renvoie « True » dans le cas où $\vec{u}$ et $\vec{v}$ sont orthogonaux, et « False » dans le cas contraire.

Solution page 141

Exercice 8.18

Écrire une fonction Python `centre_rayon(cercle)` qui renvoie les coordonnées du centre et le rayon d'un cercle dont l'équation cartésienne est sous la forme $x^2+y^2+ax+by+c=0$. L'argument `cercle` est alors sous la forme d'un triplet `(a,b,c)`. Par exemple,

```
cercle = (-1, 2, -8)
```

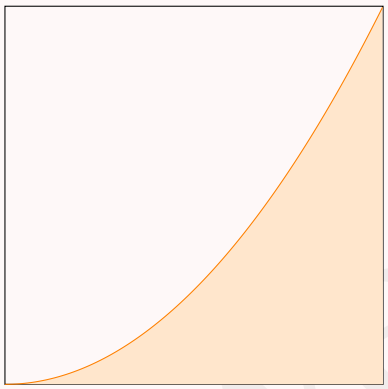
représente le cercle dont une équation cartésienne est :

$$x^2+y^2-x+2y-8=0$$

Solution page 141

Exercice 8.19 (méthode de Monte-Carlo en Python)

Dans un carré de côté 1, on trace la parabole d'équation $y = x^2$.



L'objectif de cet exercice est d'estimer l'aire sous la courbe (représentée en couleur) à l'aide d'un programme en Python.
Pour cela, nous allons utiliser la méthode de Monte-Carlo qui consiste à exécuter l'algorithme suivant :

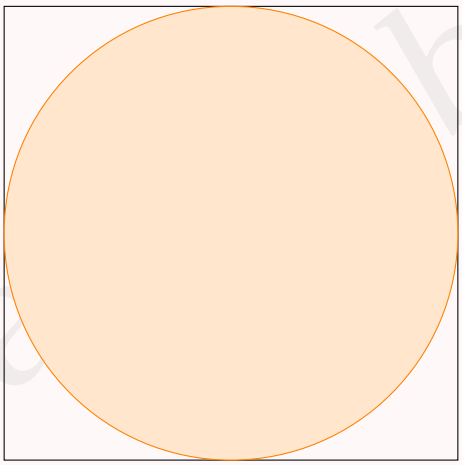
1. on choisit un point au hasard à l'intérieur du carré ;
2. s'il est sous la courbe, on incrémente un compteur ; sinon, on laisse le compteur tel qu'il est ;
3. on répète ceci un certain nombre de fois ;
4. à la fin, on calcule la fréquence de points sous la courbe : c'est l'estimation souhaitée.

Écrire un programme Python affichant une estimation de l'aire en prenant par exemple 50 000 points au hasard.

Solution page 143

Exercice 8.20

À l'intérieur d'un carré de côté 2, on trace un disque de rayon 1.



On sait que l'aire du disque est égale à π.
L'objectif de cet exercice est d'obtenir une approximation de π à l'aide d'un programme Python qui simule l'expérience suivante :

1. on choisit au hasard un point dans le carré ;
2. si ce point est à l'intérieur du disque, on incrémente un compteur ; sinon, on laisse ce compteur tel qu'il est ;
3. on répète n fois ;
4. à la fin, on calcule la fréquence de points obtenus dans le disque, et on en déduit une valeur approchée de π.

Écrire une fonction Python `approxPi(n)`, admettant en argument le nombre de points souhaités pour cette expérience, qui retourne une valeur approchée de π en suivant cette expérience.

Solution page 144

Variables aléatoires

Exercice 8.21

On considère la variable aléatoire X dont la loi de probabilité est donnée dans le tableau ci-dessous :

$X = x_i$	1	2	3	4	5	6	7
$P(X = x_i)$	0,05	0,12	0,21	0,14	0,17	0,27	0,04

On modélise en Python cette loi de probabilité par la variable `X` ci-dessous :

```
X = [ (1,0.05) , (2,0.12) , (3,0.21) , (4,0.14), (5,0.17) , (6,0.27) ,
    (7,0.04) ]
```

On considère la variable $Y = -3X + 5$.

Écrire un script Python permettant de créer la liste `Y` modélisant la loi de probabilité de Y.

Solution page 144

Exercice 8.22

On considère la variable aléatoire X dont la loi de probabilité est donnée dans le tableau ci-dessous :

$X = x_i$	1	2	3	4	5	6	7
$P(X = x_i)$	0,05	0,12	0,21	0,14	0,17	0,27	0,04

On modélise en Python cette loi de probabilité par la variable `X` ci-dessous :

```
X = [ (1,0.05) , (2,0.12) , (3,0.21) , (4,0.14), (5,0.17) , (6,0.27) ,
    (7,0.04) ]
```

1. Écrire une fonction Python `esperance(X)` qui renvoie la valeur de l'espérance de X.
2. Écrire une fonction Python `variance(X)` qui renvoie la valeur de la variance de X.

Solution page 145

Exercice 8.23 (formule de König-Huygens)

On considère la variable aléatoire X dont la loi de probabilité est donnée dans le tableau ci-dessous :

$X = x_i$	1	2	3	4	5	6	7
$P(X = x_i)$	0,05	0,12	0,21	0,14	0,17	0,27	0,04

On modélise en Python cette loi de probabilité par la variable `X` ci-dessous :

```
X = [ (1,0.05) , (2,0.12) , (3,0.21) , (4,0.14), (5,0.17) , (6,0.27) ,
    (7,0.04) ]
```

La formule de König-Huygens nous donne un moyen de calculer la variance de X en connaissance son espérance :

$$V(X) = E(X^2) - \left[E(X)\right]^2.$$

Écrire une fonction `variance(X)` calculant la variance de X en utilisant cette formule et qui renvoie cette valeur.
Pour cela, on suppose que l'on a déjà une fonction `esperance(X)` qui renvoie l'espérance de X (comme présentée dans l'exercice 8.22).

Solution page 146

Corrigé de l'exercice 8.1 page 120

1 Le discriminant d'un polynôme ax^2+bx+c est $\Delta = b^2-4ac$, d'où la fonction Python suivante :

```
def delta(a,b,c):
    return b*b - 4*a*c
```

2 Nous savons que le nombre de racines d'un polynôme de degré 2 dépend de la valeur de son discriminant Δ.

- Si $\Delta < 0$ alors il n'y a pas de racine.
- Si $\Delta = 0$ alors il n'y a qu'une racine, égale à $-\dfrac{b}{2a}$.
- Si $\Delta > 0$ alors il y a deux racines, qui sont $\dfrac{-b-\sqrt{\Delta}}{2a}$ et $\dfrac{-b+\sqrt{\Delta}}{2a}$.

On peut donc proposer la fonction Python suivante :

Code Python 8-192

```
1  def racines(a,b,c):
2      D = delta(a,b,c)
3      if D < 0: return 'Pas de racine.'
4      elif D == 0: return -b/(2*a)
5      else: return (-b-D**0.5)/(2*a) , (-b+D**0.5)/(2*a)
```

Je rappelle que `b**0.5` permet de calculer $\sqrt{b}$.

Corrigé de l'exercice 8.2 page 120

D'après le cours, on sait que si $a+b=\mathrm{S}$ et $ab=\mathrm{P}$ alors a et b sont racines de $x^2-\mathrm{S}x+\mathrm{P}$, dont le discriminant est $\Delta = \mathrm{S}^2-4\mathrm{P}$.
On peut alors proposer la fonction suivante :

Code Python 8-193

```
1  def f(S,P):
2      delta = S*S - 4*P
3      if delta > 0:
4          return (S - delta**0.5)/2 , (S + delta**0.5)/2
5      elif delta == 0:
6          return S/2
7      else:
8          return 'Pas de solution.'
```

Corrigé de l'exercice 8.3 page 120

1 Dans ce programme, il y a deux fonctions Python :

- la première renvoie l'image d'un nombre x par la fonction $f(x) = -0,3x^2+2x+1$;
- dans la seconde, on renvoie le nombre $\dfrac{f(a+h)-f(a)}{h}$, avec $h = 10^{-10}$, qui est le taux d'accroissement de la fonction f entre a et $a+h$, correspondant à une valeur approchée de $f'(a)$ car h est très proche de 0.

La fonction `coef(a)` renvoie donc une valeur approchée de $f'(a)$.

2 $f'(x) = -0,6x+2$ donc $f'(1) = -0,6+2 = 1,4$. Cette valeur correspond bien (à peu près) à celle affichée en mode console.

Corrigé de l'exercice 8.4 page 121

1 Voici une proposition de programme :

Code Python 8-194

```
1  u = [5,8,11,14,17,20,23,26,29,31,34,37,40,43]
2  r = u[1] - u[0]
3  a = True
4  for n in range(2,len(u)):
5      if u[n] - u[n-1] != r:
6          a = False
7
8  print(a)
```

Explications :

- on commence (ligne 2) par définir une variable `r` qui représente la valeur de l'éventuelle raison. Par défaut, elle vaut $u_1 - u_0$;
- ensuite, on définit (ligne 3) une variable booléenne qui vaut par défaut « True » et qui signifie que la suite est arithmétique ;
- on crée une boucle itérative (ligne 4) pour parcourir la liste `u` à partir de u_2 ;
- dans cette boucle (ligne 5), on effectue un test : si « $u_n - u_{n-1} \neq r$ », cela signifie que la différence entre le terme courant et son précédent n'est pas égal à la raison, ce qui signifie que la suite n'est pas arithmétique (donc `a` prend la valeur « False ») ;
- à la fin de la boucle, on affiche la valeur de `a` : si c'est « True », cela signifie que toutes les différences calculées dans la boucle sont égales à r et donc que la suite est arithmétique.

Quand on exécute ce programme, il s'affiche :

```
False
```

En effet, la séquence « 26 , 29 , 31 » ne correspond pas à une suite arithmétique car la différence est d'abord égale à 3, puis à 2.

2 Écrivons sur le même modèle que précédemment un programme qui permet de vérifier si v représente une suite géométrique :

Code Python 8-195

```
v = [ 100 , 50 , 25 , 12.5 , 6.25 , 3.125 , 1.5625 , 0.78125 ,
    0.375625 , 0.1878125 , 0.09390625 , 0.046953125 ]

q = v[1] / v[0]
g = True
for n in range(2,len(v)):
    if v[n] / v[n-1] != q:
        g = False

print(g)
```

On commence par calculer l'éventuelle raison $q = \frac{v_1}{v_0}$, puis on vérifie si tous les $\frac{u_n}{u_{n-1}}$ sont égaux à q.
Le résultat est sans appel :

```
False
```

La suite n'est donc pas géométrique. En effet, $\frac{50}{100} = \frac{1}{2}$ et $\frac{0,375625}{0,78125} \neq \frac{1}{2}$.

Corrigé de l'exercice 8.5 page 121

Il y a plusieurs façons d'aborder cette question :

1 On peut s'inspirer de la formule de récurrence et faire une boucle itérative pour calculer les termes successifs $u_1, u_2, \ldots, u_n$:

Code Python 8-196

```
def u(n):
    v = 7 # on ne peut pas nommer le terme 'u' car cette lettre est
    déjà prise par la fonction
    for k in range(1,n+1):
        v = v + 3.2

    return v
```

```
>>> u(8)
32.599999999999994
```

On s'aperçoit alors que le résultat n'est pas exact (car Python a une façon bien particulière de calculer et de représenter les nombres à virgules... Il fait très souvent des approximations !)

2 On peut s'aider de la formule qui donne le terme général d'une suite arithmétique (car il s'agit bien d'une suite arithmétique de premier terme $u_0 = 7$ et de raison $r = 3,2$:

$$u_n = u_0 + nr = 7 + 3,2n.$$

Dans ce cas, la fonction est très simple :

```
def u(n):
    return 7 + 3.2*n
```

```
>>> u(8)
32.6
```

3 Il y a une autre possibilité (plus compliquée au prime abord) :

Code Python 8-197

```
1 def u(n):
2     if n == 0:
3         return 7
4     else:
5         return u(n-1) + 3.2
```

```
>>> u(8)
32.599999999999994
```

Remarque 24

Cette dernière façon est la *forme récursive* du programme : dans la fonction `u(n)`, on fait appel à la fonction elle-même, mais avec un argument différent.
Pour que cette forme fonctionne (sans jeu de mot), il faut penser à « arrêter » la fonction à un moment donné : ici, c'est quand on atteint $n = 0$, auquel cas on retourne le terme initial.

4 En utilisant une boucle conditionnelle :

Code Python 8-198

```
1 def u(n):
2     v, k = 7 , 0
3     while k < n:
4         v = v + 3.2
5         k = k + 1
6     return v
```

```
>>> u(8)
32.599999999999994
```

Corrigé de l'exercice 8.6 page 121

Comme dans l'exercice précédent, il y a plusieurs façons de voir la fonction :

1 On sait que la suite est géométrique, de premier terme $u_0 = 7$ et de raison $q = 0,3$, donc son terme général est :

$$u_n = u_0 \times q^n = 7 \times 0,3^n.$$

Ce qui donne la fonction suivante :

```
def u(n):
    return 7 * 0.3**n
```

```
>>> u(8)
0.00045926999999999985
```

2 On peut s'inspirer de la relation de récurrence en créant une boucle :

Code Python 8-199

```
1  def u(n):
2      v = 7
3      for k in range(1,n+1):
4          v = v * 0.3
5      return v
```

```
>>> u(8)
0.00045926999999999996
```

3 De même, avec la forme récursive :

Code Python 8-200

```
1  def u(n):
2      if n == 0: return 7
3      else: return u(n-1) * 0.3
```

4 Avec une boucle conditionnelle :

Code Python 8-201

```
1  def u(n):
2      v, k = 7, 0
3      while k < n:
4          v = v * 0.3
5          k = k + 1
6      return v
```

Corrigé de l'exercice 8.7 page 121

La suite proposée n'est ni une suite arithmétique, ni une suite géométrique. On ne peut donc pas proposer une fonction qui retourne le terme explicite directement (à moins d'avoir étudié mathématiquement cette suite pour trouver l'expression du terme général).

1 Solution avec boucle itérative :

Code Python 8-202

```
def u(n):
    v = 7
    for k in range(1,n+1):
        v = 0.5*v + 2

    return v
```

2 Solution sous forme récursive :

Code Python 8-203

```
def u(n):
    if n == 0: return 7
    else: return 0.5 * u(n-1) + 2
```

3 Solution avec une boucle conditionnelle :

Code Python 8-204

```
def u(n):
    v = 7
    k = 0
    while k < n:
        v = 0.5 * v + 2
        k = k + 1

    return v
```

Les trois solutions renvoient le même résultat :

```
>>> u(8)
4.01171875
```

Remarque 25

La solution avec boucle conditionnelle est à mes yeux la moins performante.
En effet, elle introduit une variable `k` en plus des variables `u` et `n`. En terme informatique, cela implique un espace mémoire occupé supplémentaire.

Corrigé de l'exercice 8.8 page 122

Le programme complété est le suivant :

Code Python 8-205

```
u = 9
n = 0
while u < 1000:
    n = n + 1
    u = 1.2*u - 1

print(n)
```

Explications :

- on affecte d'abord la valeur « 9 » à la variable `u` : elle représente le premier terme de la suite (u_n) ;
- ensuite, on affecte à `n` la valeur « 0 », qui est la valeur de l'indice du premier terme de la suite ;
- on souhaite déterminer la première valeur de n pour laquelle $u_n \geqslant 1\,000$, donc on crée une boucle *conditionnelle* avec pour condition le contraire de ce que l'on veut : `u<1000`. En effet, *tant que* $u_n < 1\,000$, il faut calculer les termes successifs de la suite : u_1, u_2,... ;
- ainsi, dans la boucle, on commence par augmenter de 1 la valeur de `n`, puis on affecte à `u` la valeur `1.2*u-1`, qui représente le terme suivant (selon la relation de récurrence de la suite).

Une fois la boucle finie, cela signifie que la condition `u<1000` n'est plus remplie, et donc que `u` contient une valeur plus grande ou égale à 1 000. On affiche alors la valeur stockée dans `n`, qui correspond au rang pour lequel $u_n \geqslant 1\,000$.

Corrigé de l'exercice 8.9 page 122

Le programme complété est le suivant :

Code Python 8-206

```
u = 95
n = 0
while u >= 20 + 10**(-9):
    n = n + 1
    u = 0.95*u + 1
print(n)
```

Explications :

- on affecte d'abord la valeur « 95 » à la variable `u` : elle représente le premier terme de la suite (u_n) ;
- ensuite, on affecte à `n` la valeur « 0 », qui est la valeur de l'indice du premier terme de la suite ;

- on souhaite déterminer la première valeur de n pour laquelle $u_n < 20 + 10^{-9}$, donc on crée une boucle *conditionnelle* avec pour condition le contraire de ce que l'on veut : `u>=20+10**(-9)`. En effet, *tant que* $u_n \geqslant 20+10^{-9}$, il faut calculer les termes successifs de la suite : u_1, u_2,... ;
- ainsi, dans la boucle, on commence par augmenter de 1 la valeur de `n`, puis on affecte à `u` la valeur `0.95*u+1`, qui représente le terme suivant (selon la relation de récurrence de la suite).

Une fois la boucle finie, cela signifie que la condition `u>=20+10**(-9)` n'est plus remplie, et donc que `u` contient une valeur plus grande ou égale à $20 + 10^{-9}$. On affiche alors la valeur stockée dans `n`, qui correspond au rang pour lequel $u_n < 20 + 10^{-9}$.

Corrigé de l'exercice 8.10 page 123

Voici un programme satisfaisant la question :

Code Python 8-207

```
n, u = 0, 7 # n prend la valeur 0 et u prend la valeur 7
while u >= 4.0001:
    u = 0.5 * u + 2
    n = n + 1
print(n)
```

On commence par mettre « 0 » dans `n` (premier indice de la suite) et « 7 » dans `u` (premier terme), puis on calcule les termes successifs en multipliant la valeur stockée dans `u` par 0,5 et en ajoutant 2, ce qui donne la nouvelle valeur de `u` (et qui correspond au terme suivant), tout ceci *tant que* la valeur de `u` est plus grande ou égale au seuil imposé (4,0001) car on souhaite que ces calculs se fassent jusqu'à ce que $u_n < 4,0001$.
Ainsi, quand on sort de la boucle, cela signifie que la condition n'est plus remplie et donc que $u_n < 4,0001$, et la valeur de n affichée correspond à l'indice du premier terme terme pour lequel $u_n < 4,0001$.
On obtient :

```
15
```

ce qui signifie que $u_{14} \geqslant 4,0001$ mais que $u_{15} < 4,0001$.

Corrigé de l'exercice 8.11 page 123

Voici une proposition de programme :

Code Python 8-208

```
u, n = 1, 0 # u prend la valeur 1, n prend la valeur 0
while u < 10**9:
    u = 1.05*u + 1
    n = n + 1
print(n)
```

On commence par définir le premier n et le premier terme u_0, que l'on stocke dans `u`.
Ensuite, on calcule les termes successifs de la suite *tant que* l'on obtient une valeur plus petite que 10^9. On n'oublie pas d'ajouter 1 à n à chaque fois car on souhaite afficher le premier n pour lequel on sort de la boucle.
En exécutant ce programme, on obtient :

```
363
```

ce qui signifie que $u_{362} \leqslant 10^9$ mais $u_{363} > 10^9$.

Corrigé de l'exercice 8.12 page 123

1 Dans ce programme, on commence par initialiser trois variables : `u`, `s` et `n`.
Ensuite, on initialise une boucle *conditionnelle*; la condition est que la valeur contenue dans `s` doit être strictement inférieure à 10^9 pour que les instructions suivantes s'exécutent.
Les instructions de la boucle sont les suivantes :

- d'abord, on ajoute 1 à la valeur contenue dans `n` ;
- ensuite, on ajoute à la valeur contenue dans `s` celle contenue dans `u` ;
- pour finir, on ajoute 1,5 à la valeur contenue dans `u`.

Faisons un tableau pour représenter les valeurs des variables pas à pas.
Astuce du chef : nous allons disposer les variables dans le même ordre où elles apparaissent dans la boucle : d'abord `n`, suivie de `u`, puis de `s`.

`n`	0	1	2	3	$\cdots$
`u`	-8	$-8+1,5=-6,5$	$-6,5+1,5=-5$	$-5+1,5=-3,5$	$\cdots$
`s`	-8	$-8+(-6,5)=-14,5$	$-14,5+(-5)=-19,5$	$-19,5+(-3,5)=-23$	$\cdots$

La valeur affichée par le programme est donc la première valeur de `n` pour laquelle la condition $s < 10^9$ n'est pas remplie, à savoir la première valeur de n pour laquelle la valeur stockée dans `s` est supérieure ou égale à 10^9.

2 Si on examine de plus près le tableau précédent, on peut imaginer que la variable `u` représente les termes d'une suite arithmétique de premier terme $u_0 = -8$ et de raison $r = 1,5$, et `s` représente $s_n = u_0 + u_1 + \cdots + u_n$.
Ainsi, lorsque les variables `u`, `s` et `n` sont initialisées, elles représentent respectivement u_0, s_0 et le premier indice $n = 0$.
Vérification pour les premières valeurs :

- $u_1 = u_0 + 1,5 = -8 + 1,5 = -6,5$, qui correspond à la première valeur de `u` trouvée dans le tableau (après initialisation) ;
- $s_1 = u_0 + u_1 = -8 + (-6,5) = -14,5$, qui correspond à la première valeur de `s` trouvée dans le tableau (après initialisation) ;
- $u_2 = u_1 + 1,5 = -6,5 + 1,5 = -5$, qui correspond à la valeur suivante de `u` ;
- $s_2 = u_0 + u_1 + u_2 = -8 + (-6,5) + (-5) = -19,5$, qui correspond à la valeur suivante de `s`.

Ainsi, la valeur de n affichée par le programme est la première valeur de n pour laquelle $s_n \geqslant 10^9$.
Or, le cours nous donne une valeur de s_n :

$$\begin{aligned} s_n &= u_0 + u_1 + u_2 + \cdots + u_n \\ &= (n+1) \times \frac{u_0 + u_n}{2} \\ &= (n+1) \times \frac{u_0 + (u_0 + nr)}{2} \\ &= (n+1) \times \frac{2u_0 + nr}{2} \\ &= (n+1)\left(u_0 + \frac{nr}{2}\right) \\ &= (n+1)(0{,}75n - 8) \\ &= 0{,}75n^2 - 7{,}25n - 8. \end{aligned}$$

On souhaite donc résoudre l'inéquation :

$$0{,}75n^2 - 7{,}25n - 8 \geqslant 10^9 \iff 0{,}75n^2 - 7{,}25n - 1\,000\,000\,008 \geqslant 0.$$

Le discriminant du polynôme $0{,}75n^2 - 7{,}25n - 1\,000\,000\,008$ est :

$$\Delta = (-7{,}25)^5 - 4 \times 0{,}75 \times (-1\,000\,000\,008) = 3\,000\,000\,076{,}562\,5.$$

Donc $0{,}75n^2 - 7{,}25n - 1\,000\,000\,008 \geqslant 0$ pour $n \geqslant \dfrac{-b + \sqrt{\Delta}}{2a}$, soit pour $n \geqslant 36\,519{,}670\,966\,3$.
Le premier n est donc $n = 36\,520$, et c'est bien ce qu'affiche le programme (après vérification sur une machine).

Corrigé de l'exercice 8.13 page 124

Voici une proposition de programme :

Code Python 8-209

```
1 u = 5
2 s = 5
3 for n in range(100):
4     u = 0.3 * u # terme suivant
5     s = s + u # on ajoute le nouveau 'u' à la somme précédente
6
7 print(s)
```

En lignes 1 et 2, on définit le premier terme de la suite et le terme initial de la somme (égal à u_0).
Ensuite, on crée une boucle qui exécutera 100 fois les instructions des lignes 4 et 5 : à chaque passage dans la boucle, on calcule le terme suivant de la suite et on l'ajoute à la valeur précédente de la somme.

On obtient :

```
7.142857142857144
```

Remarque 26

On aurait aussi pu utiliser la formule du cours qui nous donne la somme en fonction de q et n :

$$u_0 + u_1 + \cdots + u_n = u_0 \times \frac{q^{n+1} - 1}{q - 1}$$

ce qui donne le programme suivant :

Code Python 8-210

```
1  u0 = 5
2  q = 0.3
3  s = u0 * ( (q**101 - 1) / (q - 1) )
4
5  print(s)
```

Mais c'est quand-même moins *fun* n'est-ce pas ?

Corrigé de l'exercice 8.14 page 124

Avant tout, je vais légèrement modifier la relation de récurrence de la suite :

$$\begin{cases} u_0 = 5 \\ u_n = \frac{1}{2}u_{n-1} + 2(n-1) - 3 = \frac{1}{2}u_{n-1} + 2n - 5 \end{cases}$$

L'avantage de cette relation de récurrence est de faire apparaître la relation entre le terme courant (u_n) et son précédent (u_{n-1}), ce qui peut nous aider pour le programme Python.

Code Python 8-211

```
1  u = 5
2  s = 5
3
4  for n in range(1,1000):
5      u = 0.5*u + 2*n - 5
6      s = s + u
7
8  print(s)
```

N'oublions pas que dans l'écriture « `u = 0.5*u + 2*n - 5` », le `u` de gauche représente u_n (le terme que l'on calcule) et le `u` de droite représente le terme précédent (donc u_{n-1}).

En exécutant le programme, on obtient :

```
1984038.0
```

Corrigé de l'exercice 8.15 page 124

1 Dans la fonction `somme(k)`, on initialise la variable `s` à 0.
Ensuite, on crée une boucle itérative (`k` prendra des valeurs de 1 à `n`) dans laquelle on stocke dans `s` la somme de la valeur qu'il y avait précédemment et de $\frac{1}{k^2}$.
Ainsi, la fonction calcule et renvoie la somme :

$$S_n = 0 + \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2}$$

c'est-à-dire :

$$S_n = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2} = \sum_{k=1}^{n} \frac{1}{k^2}.$$

2 À la vue des résultats successifs affichés pour `somme(1000)`, `somme(10000)` et `somme(10**7)`, on constate que la somme S_n se rapproche de plus en plus d'un nombre dont une valeur approchée est 1,644.
On pourrait alors conjecturer que S_n vaut à peu près 1,644 quand n tend vers l'infini.

Corrigé de l'exercice 8.16 page 125

1 Voici un programme possible :

Code Python 8-212

```
s = 0
for p in range(1011):
    s = s + (-1)**p / (2*p + 1)
print(s)
```

La formule à insérer dans la boucle est clairement dictée par celle qu'il y a dans le symbole de sommation de l'énoncé :

$$\sum_{p=0}^{1010} \frac{(-1)^p}{2p+1}.$$

Cette façon d'écrire est très utile car la fonction Python reprend les indices : on voit que la somme part de $p = 0$ et va jusqu'à $p = 1010$; comme nous utilisons la fonction `range`, il ne faut pas oublier d'indiquer le « nombre limite » auquel on ajoute 1.

2 Un programme possible est le suivant :

Code Python 8-213

```
s = 0
for p in range(1,2022):
    s = s + 1 / (3*p - 1)
print(s)
```

Je me suis là encore inspiré de la somme donnée en énoncé avec le symbole de sommation.

Corrigé de l'exercice 8.17 page 125

1 Le produit scalaire de deux vecteurs $\vec{u}\begin{pmatrix}x\\y\end{pmatrix}$ et $\vec{v}\begin{pmatrix}x'\\y'\end{pmatrix}$ est égal à $xx'+yy'$, d'où la fonction Python suivante :

```
def produit(u,v):
    return u[0] * v[0] + u[1] * v[1]
```

2 On sait que $\vec{u}$ et $\vec{v}$ sont orthogonaux si et seulement si $\vec{u}\cdot\vec{v}=$, d'où la fonction suivante :

```
def isOrth(u,v):
    return produit(u,v) == 0
```

que l'on peut aussi écrire sous la forme suivante :

```
def isOrth(u,v):
    if produit(u,v) == 0: return True
    else: return False
```

On a par exemple, en mode console :

```
>>> isOrth( (-1,3) , (3,1) )
True
>>> isOrth( (-1,3) , (3,-1) )
False
```

Corrigé de l'exercice 8.18 page 125

Il est nécessaire de faire un travail mathématique avant toute chose.

$$\begin{aligned}x^2+y^2+ax+by+c=0 &\iff (x^2+ax)+(y^2+by)+c=0\\ &\iff \left[\left(x+\frac{a}{2}\right)^2-\left(\frac{a}{2}\right)^2\right]+\left[\left(y+\frac{b}{2}\right)^2-\left(\frac{b}{2}\right)^2\right]+c=0\\ &\iff \left(x+\frac{a}{2}\right)^2+\left(y+\frac{b}{2}\right)^2=\left(\frac{a}{2}\right)^2+\left(\frac{b}{2}\right)^2-c\end{aligned}$$

On a ainsi une équation de la forme :

$$(x-x_C)^2+(x-y_C)^2=r^2$$

avec :

$$x_C=-\frac{a}{2},\qquad y_C=-\frac{b}{2}$$

et

$$r=\sqrt{\left(\frac{a}{2}\right)^2+\left(\frac{b}{2}\right)^2-c}\quad\text{si}\quad\left(\frac{a}{2}\right)^2+\left(\frac{b}{2}\right)^2-c\geqslant 0.$$

Ceci nous donne alors les coordonnées du centre C du cercle, ainsi que son éventuel rayon r.
D'où la fonction Python suivante :

Code Python 8-214

```
def centre_rayon( cercle ):
    xC = -cercle[0] / 2 # correspond à '-a/2'
    yC = -cercle[1] / 2 # correspond à '-b/2'

    R = xC**2 + yC**2 - cercle[2] # correspond à (a/2)^2 + (b/2)^2 - c

    if R >= 0:
        r = R**0.5 # racine carrée de R
        return (xC, yC) , r
    else:
        return "Ce n'est pas une équation cartésienne de cercle."
```

Testons en mode console :

```
>>> centre_rayon( (-1,2,-8) )
((0.5, -1.0), 3.0413812651491097)
```

Ceci nous indique que si on considère le cercle d'équation $x^2+y^2-x+2y-8=0$, son centre a pour coordonnées $\left(\frac{1}{2};-1\right)$ et pour rayon $r\approx 3,04$.

Pour éviter d'avoir à écrire une double paire de parenthèses lors de l'appel à la fonction `centre_rayon`, on pourrait ajouter une astérisque devant l'argument de la fonction et écrire :

Code Python 8-215

```
def centre_rayon( *cercle ):
    xC = -cercle[0] / 2 # correspond à '-a/2'
    yC = -cercle[1] / 2 # correspond à '-b/2'

    R = xC**2 + yC**2 - cercle[2] # correspond à (a/2)^2 + (b/2)^2 - c

    if R >= 0:
        r = R**0.5 # racine carrée de R
        return (xC, yC) , r
    else:
        return "Ce n'est pas une équation cartésienne de cercle."
```

Ainsi, on n'aurait plus qu'à écrire :

```
>>> centre_rayon(-1,2,-8)
((0.5, -1.0), 3.0413812651491097)
```

Mais cela dépasse largement les capacités attendues en mathématiques concernant Python.

Corrigé de l'exercice 8.19 page 126

Un programme possible est le suivant :

Code Python 8-216

```
from random import random
n = 0

for i in range(5000):
    x = random()
    y = random()
    if (y < x*x):
        n = n + 1

print(n/5000)
```

Explications :

- on commence par importer le module *random* qui permet de simuler l'aléatoire ;
- on initialise la variable `n` à 0 : c'est la variable qui va compter le nombre de points en dessous de la courbe ;
- on crée une boucle de sorte à répéter 5 000 fois l'expérience consistant à choisir au hasard deux nombres `x` et `y`, qui vont jouer le rôle des coordonnées du point ;
- on teste si $y < x^2$: si tel est le cas, cela signifie que le point est bien en dessous de la courbe et on incrémente `n` (on lui ajoute 1) ;
- une fois la boucle finie, on affiche la valeur de $\dfrac{n}{5\,000}$, qui correspond à la fréquence de points obtenus en dessous de la courbe.

Voici trois résultats successifs : 0,3358 ; 0,3326 ; 0,3308.
On peut modifier légèrement le programme afin qu'il simule plus de choix :

Code Python 8-217

```
from random import random
n = 0
c = int(input('Nombre de choix : '))
for i in range(c):
    x = random()
    y = random()
    if (y < x*x):
        n = n + 1

print(n/c)
```

On peut ainsi choisir le nombre de choix. Il semble que plus le nombre de choix augmente, plus la fréquence se rapproche de 0,333333...

Corrigé de l'exercice 8.20 page 126

Voici une proposition de fonction Python permettant de donner une approximation de π suivant l'expérience décrite dans l'énoncé :

Code Python 8-218

```
from random import random

def approxPi(n):
    c = 0
    for k in range(n):
        x = 2 * random()
        y = 2 * random()
        if ( (x-1)**2 + (y-1)**2 )**0.5 <= 1:
            c = c + 1

    f = c / n # fréquence de points à l'intérieur du disque
    print( 4*f )
```

```
>>> approxPi(10**6)
3.1428
>>> approxPi(5*10**6)
3.1418528
```

Cette fonction a tout de même ses limites... En particulier, pour des valeurs de n supérieures à 10^7, le temps d'exécution est très long, et ce pour un résultat très approximatif (uniquement 3 chiffres corrects après la virgule).
Il est donc clairement évident que l'on ne peut pas envisager une telle fonction pour avoir de très bonnes approximations de π.

Corrigé de l'exercice 8.21 page 127

Pour construire la variable `Y`, on peut procéder ainsi :

Code Python 8-219

```
X = [ (1,0.05) , (2,0.12) , (3,0.21) , (4,0.14), (5,0.17) , (6,0.27) ,
    (7,0.04) ]
Y = [ ]
for c in X:
    Y.append( ( -3*c[0]+5 , c[1] ) )
```

Explications :

- on commence par initialiser la variable `Y` comme étant une liste vide (crochets sans rien à l'intérieur) ;
- on parcourt la liste `X` avec l'instruction « `for c in X:` » : ici, chaque valeur que va prendre la variable `c` sera une valeur contenue dans la liste `X` ; donc `c` est un couple, où `c[0]` représentera la valeur de X et `c[1]` sa probabilité ;

- on insère dans la liste `Y` le couple `(-3*c[0]+5,c[1])` car $Y = -3X + 5$, ce qui signifie que chaque valeur de Y est obtenue en multipliant par -3 chaque valeur que prend X et en ajoutant 5, et que chaque probabilité reste inchangée.

Remarque 27

On peut aussi envisager de construire la liste `Y` par *compréhension*, où la syntaxe est plus concise :

Code Python 8-220

```
X = [ (1,0.05) , (2,0.12) , (3,0.21) , (4,0.14), (5,0.17) ,
    (6,0.27) , (7,0.04) ]
Y = [ ( -3*c[0]+5 , c[1] ) for c in X]
```

Corrigé de l'exercice 8.22 page 128

1 Voici une proposition de fonction pour calculer l'espérance de X :

Code Python 8-221

```
def esperance(X):
    s = 0
    for c in X:
        s = s + c[0]*c[1]

    return s
```

Pour écrire cette fonction, je me suis inspiré de la formule permettant de calculer l'espérance :

$$E(X) = \sum_{i=1}^{n} x_i \times pi$$

On doit donc ajouter tous les produits $x_i \times p_i$, représentés en Python par `c[0]*c[1]`.

2 La variance de X est l'espérance de la variable aléatoire $(X-E(X))^2$. On peut donc écrire la fonction suivante :

Code Python 8-222

```
def variance(X):
    E = esperance(X)
    Y = [ ]
    for c in X:
        Y.append( ( (c[0]-E)**2 , c[1] ) )
    return esperance(Y)
```

Si vous êtes à l'aise avec les listes par compréhension, on peut aussi considérer le programme page suivante.

Code Python 8-223

```
def variance(X):
    E = esperance(X)
    # on construit Y par compréhension
    Y = [ ((c[0]-E)**2 , c[1]) for c in X ]

    return esperance(Y)
```

```
>>> esperance(X), variance(X)
4.23, 2.6971
```

Corrigé de l'exercice 8.23 page 128

Une fonction possible est la suivante :

Code Python 8-224

```
def variance(X):
    Y = [ ]
    for c in X:
        Y.append( ( c[0]**2 , c[1] ) )
    return esperance(Y) - esperance(X)**2
```

que l'on peut aussi écrire de façon plus concise comme suit (en construisant la liste `Y` par compréhension) :

Code Python 8-225

```
def variance(X):
    Y = [ ( c[0]**2 , c[1] ) for c in X ]
    return esperance(Y) - esperance(X)**2
```

Remarque 28

Vous voyez ici l'importance de créer des fonctions pour calculer des quantités bien définies (comme l'espérance) dans un programme. En effet, on faisant ainsi, on peut se servir de ces fonctions dans d'autres fonctions pour aller plus vite et que pour le programme soit plus lisible.

9 Python en classe de Terminale

Plan du chapitre

I Factorielle 149
1 Première approche : boucle itérative 149
2 Deuxième approche : fonction récursive 149
3 Troisième approche : boucle conditionnelle 150
4 Quatrième approche : le module math 150
II Loi binomiale 150
1 L'objet « binom » et sa méthode « stats » 150
2 L'objet « binom » du module personnel probastat 151
III Dénombrement 152
1 Produit cartésien 152
2 Combinaisons 152
3 Permutations 152
4 Arrangements 153
IV Valeur approchée d'un logarithme népérien : algorithme de Briggs 153
1 Principe 153
2 L'algorithme 154
V Calcul de ln(2) par l'algorithme de Brouncker 155
1 Principe 155
2 Une première approche 157
3 Une autre approche 158
VI Maths complémentaires : simuler la loi géométrique 158
VII Math complémentaires : droite de régression linéaire 160
VIII Maths expertes : calcul du PGCD avec l'algorithme d'Euclide 161
IX Maths expertes : un couple de Bézout 162
X Maths expertes : crible d'Eratosthène 163
XI Maths expertes : décomposition en produit de facteurs premiers 163
Enoncés 165
Corrigés des exercices 174

Extrait du programme

La génération des listes en compréhension et en extension est mise en lien avec la notion d'ensemble. Les conditions apparaissant dans les listes définies en compréhension permettent de travailler la logique. Afin d'éviter des confusions, on se limite aux listes sans présenter d'autres types de collections.
Capacités attendues :

- Générer une liste (en extension, par ajouts successifs ou en compréhension).
- Manipuler des éléments d'une liste (ajouter, supprimer,...) et leurs indices.
- Parcourir une liste.
- Itérer sur les éléments d'une liste.

Attention 7

Certains aspects qui interviennent en classe de Seconde et de Première ne seront pas systématiquement repris dans ce chapitre.
Il est donc conseillé de regarder les chapitres précédents avant d'aborder celui-ci.

I - Factorielle

La notion de *factorielle* apparaît dans les dénombrements.

Définition 1

On définit la *factorielle* d'un entier n par le produit :

$$n! = 1 \times 2 \times 3 \times 4 \times 5 \times \cdots \times (n-2) \times (n-1) \times n.$$

Exemple 39

- $3! = 1 \times 2 \times 3 = 6$.
- $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

Vous avez sans doute remarqué que j'ai laissé le facteur « 1 » dans la définition de la factorielle, bien qu'inutile dans le produit. La raison est purement informatique : nous allons voir que pour implémenter une fonction Python retournant une factorielle, ce « 1 » est très important.

I . 1 - Première approche : boucle itérative

La première façon d'écrire une fonction Python `factorielle(n)` renvoyant $n!$ est d'utiliser une boucle itérative de la manière suivante :

Code Python 9-226

```
1 def factorielle(n):
2     f = 1
3     for k in range(1,n+1):
4         f = f * k
5     return f
```

```
>>> factorielle(5)
120
>>> factorielle(10)
3628800
```

I . 2 - Deuxième approche : fonction récursive

Une deuxième façon d'implémenter cette fonction est de manière récursive :

Code Python 9-227

```
1 def factorielle(n):
2     if n == 1: return 1
3     else: return n * factorielle(n-1)
```

L'implémentation récursive est basée sur le fait que :

- $2! = 2 \times 1 = 2 \times 1!$
- $3! = 3 \times (2 \times 1) = 3 \times 2!$
- $4! = 4 \times (3 \times 2 \times 1) = 4 \times 3!$
- $\vdots$
- $n! = n \times (n-1)!$

I . 3 - Troisième approche : boucle conditionnelle

De la forme récursive, on peut aussi voir la fonction de la manière suivante :

Code Python 9-228

```
1 def factorielle(n):
2     f = 1
3     while n > 1:
4         f = f * n
5         n = n - 1
6 
7     return f
```

I . 4 - Quatrième approche : le module math

Nous l'avons vu précédemment, le module « math » contient déjà une fonction factorielle (très performante puisqu'implémentée en C).

```
>>> from math import factorial
>>> factorial(435)
```

II - Loi binomiale

Le module `scipy.stats` contient des fonctions pour calculer plusieurs nombres en rapport avec la loi binomiale.

II . 1 - L'objet « binom » et sa méthode « stats »

On ne va parler de *fonction* mais d'*objet* ici.

```
>>> from scipy.stats import binom
>>> moyenne, variance, sigma = binom.stats(50, 0.3, moments='mvs')
>>> moyenne
array(15.)
>>> variance
array(10.5)
>>> sigma
array(0.12344268)
```

La deuxième ligne définit respectivement la moyenne, la variance et l'écart-type de la variable aléatoire qui suit la loi binomiale de paramètres $n = 50$ et $p = 0,3$.

Remarque 29

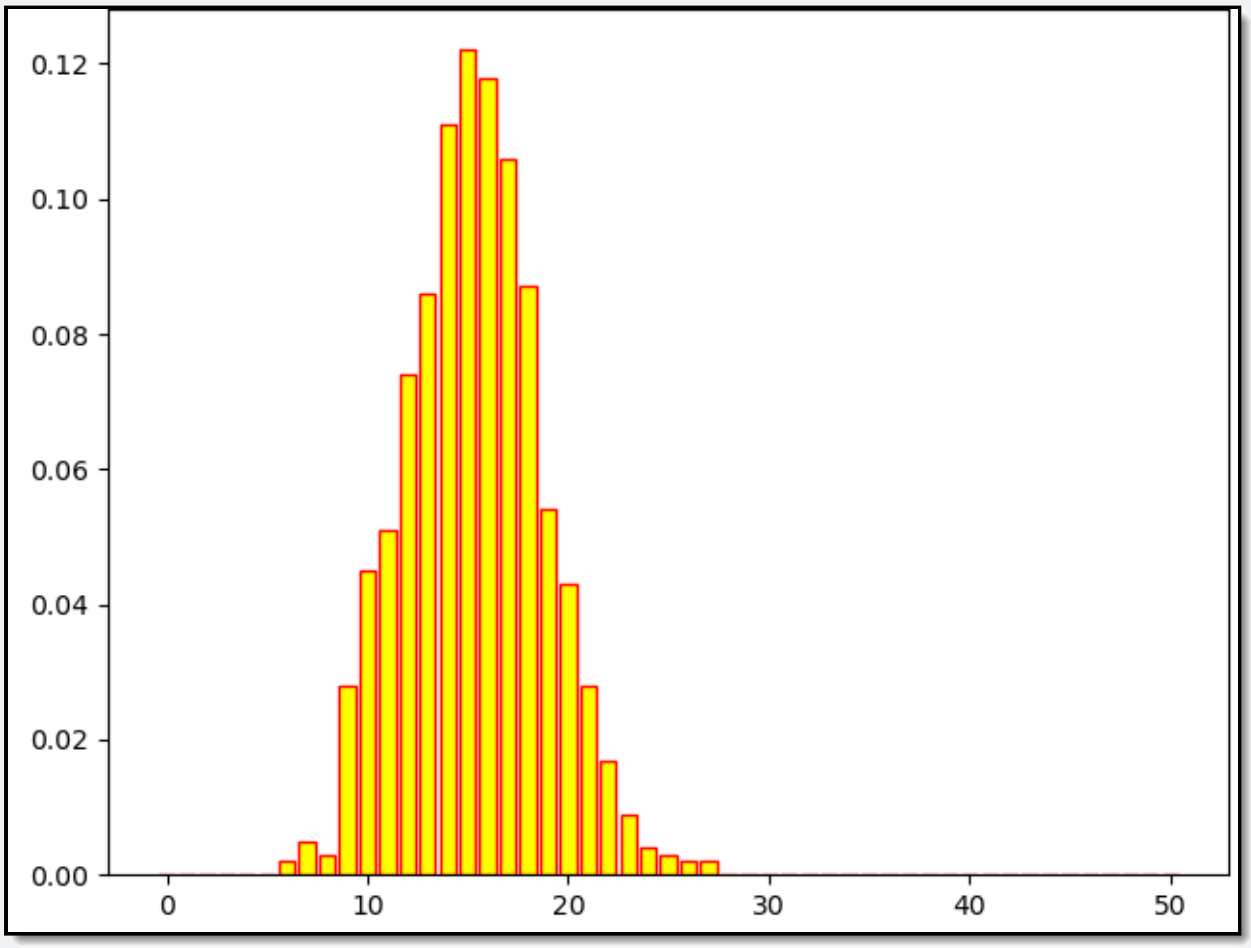
L'argument « `moments = 'mvs'` » signifie que l'on souhaite avoir un triplet de la forme `(Mean, Variance, Skew)` (*mean* pour « moyenne » et *skew* pour « écart-type »).

Remarquez aussi que les résultats sont sous forme de tableaux (*array*).

II . 2 - L'objet « binom » du module personnel probastat

N'étant pas satisfait des outils disponibles, je me suis fais mon propre module `probastat` pour obtenir ce que l'on demande le plus souvent au lycée :

```
>>> from probastat import binom
>>> X = binom(50,0.3)
>>> X.esp(2), X.var(2) , X.ecart(2)
15.0, 10.5, 3.24
>>> X.proba(10 , 2)
0.04
>>> X.proba_cdf(15 , 2)
0.57
>>> X.proba_icdf(0.95)
20
>>> X.simul(1000,'yellow','red')
```



Pour télécharger ce module, vous pouvez vous rendre sur cet article :

https://www.mathweb.fr/euclide/2021/04/09/probabilites-et-python-au-lycee-loi-binomiale-et-variables-aleatoires/

III - Dénombrement

Le module *itertools* comporte plusieurs fonctions intéressantes.

III . 1 - Produit cartésien

`product(E,F)` donne le produit cartésien des deux itérables E et F.

Exemple 40

```
>>> from itertools import product
>>> E, F = 'abc', 'mn'
>>> list( product(E,F) )
[('a', 'm'), ('a', 'n'), ('b', 'm'), ('b', 'n'), ('c', 'm'), ('c', 'n')]
```

III . 2 - Combinaisons

`combinations(iterable,k)` nous donne la liste de toutes les combinaisons de l'itérable (liste ou chaîne de caractères par exemple) à k éléments.

Exemple 41

```
>>> from itertools import combinations
>>> list( combinations('ABC',2) )
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

Si un jeu de cartes est constitué de seulement trois cartes, il n'y a que trois mains de deux cartes possibles.

Remarque 30

N'oublions pas que l'ordre des éléments ne compte pas dans une combinaison. Ainsi, la main « `('A','B')` » est identique à la main « `('B','A')` ».

III . 3 - Permutations

`permutations(iterable)` nous donne la liste de toutes les permutations des éléments de l'itérable.

Exemple 42

```
>>> from itertools import permutations
>>> E = 'ABC'
>>> list( permutations(E) )
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C',
    'A', 'B'), ('C', 'B', 'A')]
```

III . 4 - Arrangements

`permutations(iterable,k)` nous donne la liste de tous les arrangements à k éléments de l'itérable.

Exemple 43

```
>>> from itertools import permutations
>>> E = 'ABCDE'
>>> list( permutations(E,3) )
[('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'B', 'E'), ('A', 'C', 'B'), ('A',
    'C', 'D'), ('A', 'C', 'E'), ('A', 'D', 'B'), ('A', 'D', 'C'), ('A',
    'D', 'E'), ('A', 'E', 'B'), ('A', 'E', 'C'), ('A', 'E', 'D'), ('B',
    'A', 'C'), ('B', 'A', 'D'), ('B', 'A', 'E'), ('B', 'C', 'A'), ('B',
    'C', 'D'), ('B', 'C', 'E'), ('B', 'D', 'A'), ('B', 'D', 'C'), ('B',
    'D', 'E'), ('B', 'E', 'A'), ('B', 'E', 'C'), ('B', 'E', 'D'), ('C',
    'A', 'B'), ('C', 'A', 'D'), ('C', 'A', 'E'), ('C', 'B', 'A'), ('C',
    'B', 'D'), ('C', 'B', 'E'), ('C', 'D', 'A'), ('C', 'D', 'B'), ('C',
    'D', 'E'), ('C', 'E', 'A'), ('C', 'E', 'B'), ('C', 'E', 'D'), ('D',
    'A', 'B'), ('D', 'A', 'C'), ('D', 'A', 'E'), ('D', 'B', 'A'), ('D',
    'B', 'C'), ('D', 'B', 'E'), ('D', 'C', 'A'), ('D', 'C', 'B'), ('D',
    'C', 'E'), ('D', 'E', 'A'), ('D', 'E', 'B'), ('D', 'E', 'C'), ('E',
    'A', 'B'), ('E', 'A', 'C'), ('E', 'A', 'D'), ('E', 'B', 'A'), ('E',
    'B', 'C'), ('E', 'B', 'D'), ('E', 'C', 'A'), ('E', 'C', 'B'), ('E',
    'C', 'D'), ('E', 'D', 'A'), ('E', 'D', 'B'), ('E', 'D', 'C')]
```

Si l'on imagine une compétition à cinq participants, on obtient la liste de tous les podiums (trois gagnants) possibles.

IV - Valeur approchée d'un logarithme népérien : algorithme de Briggs

IV . 1 - Principe

Pour une fonction f définie en $x = a$, nous savons que l'équation réduite de la tangente en $x = a$ à sa courbe représentative est :

$$y = f'(a)(x - a) + f(a)$$

ce qui donne pour la fonction $x \mapsto \ln x$ et $a = 1$:

$$y = x - 1.$$

Par définition, cette tangente est très proche de la courbe au voisinage de $x = 1$, ce qui sous-entend que $\ln(x)$ et $x - 1$ sont très proches pour x très proche de 1 :

$$\text{pour } x \text{ très proche de } 1, \qquad \ln x \approx x - 1.$$

Considérons maintenant un nombre positif $a \neq 1$.

On démontre que la suite (u_n) définie pour tout entier naturel n par :

$$\begin{cases} u_0 = a \\ u_{n+1} = \sqrt{u_n} \end{cases}$$

converge vers 1.
Autrement dit, il existe une valeur de p telle que, pour tout $n \geqslant p$, $u_n \approx 1$. En particulier, $u_p \approx 1$.
Alors,

$$\ln(u_p) \approx u_p - 1.$$

Or, $u_p = \sqrt{u_{p-1}}$ donc :

$$\ln(u_p) = \ln\sqrt{u_{p-1}} = \frac{1}{2}\ln(u_{p-1}).$$

De même, $u_{p-1} = \sqrt{u_{p-2}}$ donc :

$$\ln(u_p) = \frac{1}{2} \times \frac{1}{2}\ln(u_{p-2}) = \frac{1}{2^2}\ln(u_{p-2}).$$

Par récurrence, on démontre alors que :

$$\ln(u_p) = \frac{1}{2^p}\ln u_0.$$

Ainsi,

$$\frac{1}{2^p}\ln u_0 \approx u_0 - 1$$

soit :

$$\ln u_0 = \ln a \approx 2^p(a-1).$$

L'algorithme de Briggs consiste à trouver la valeur de p pour déterminer la valeur de $\ln(a)$.

IV . 2 - L'algorithme

L'algorithme de Briggs consiste donc à trouver la première valeur p pour laquelle u_p est *assez proche* de 1.
Tout dépend donc de ce que l'on appelle « assez proche » : cela va être arbitraire. On peut considérer que $u_p \approx 1$ si $|u_p - 1| \leqslant 10^{-3}$ tout comme on peut considérer que $u_p \approx 1$ si $|u_p - 1| \leqslant 10^{-8}$...
D'où l'idée de définir une fonction Python admettant en argument cette marge d'erreur acceptée.

Code Python 9-229

```
1 def briggs(a,e):
2     n = 0 # indice du premier terme de la suite (u)
3     while abs(a - 1) > e:
4         a = a ** 0.5 # terme suivant dans la suite (u)
5         n = n + 1    # rang de ce terme
6 
7     # à la sortie de la boucle, n vaut p
8 
9     return (a - 1) * (2**n)
```

```
>>> briggs(45,10**(-7))
3.806662604212761
```

Si on compare cette valeur à $\ln 45 \approx 3,8066624897703\ldots$, c'est pas mal du tout !

V - Calcul de ln(2) par l'algorithme de Brouncker

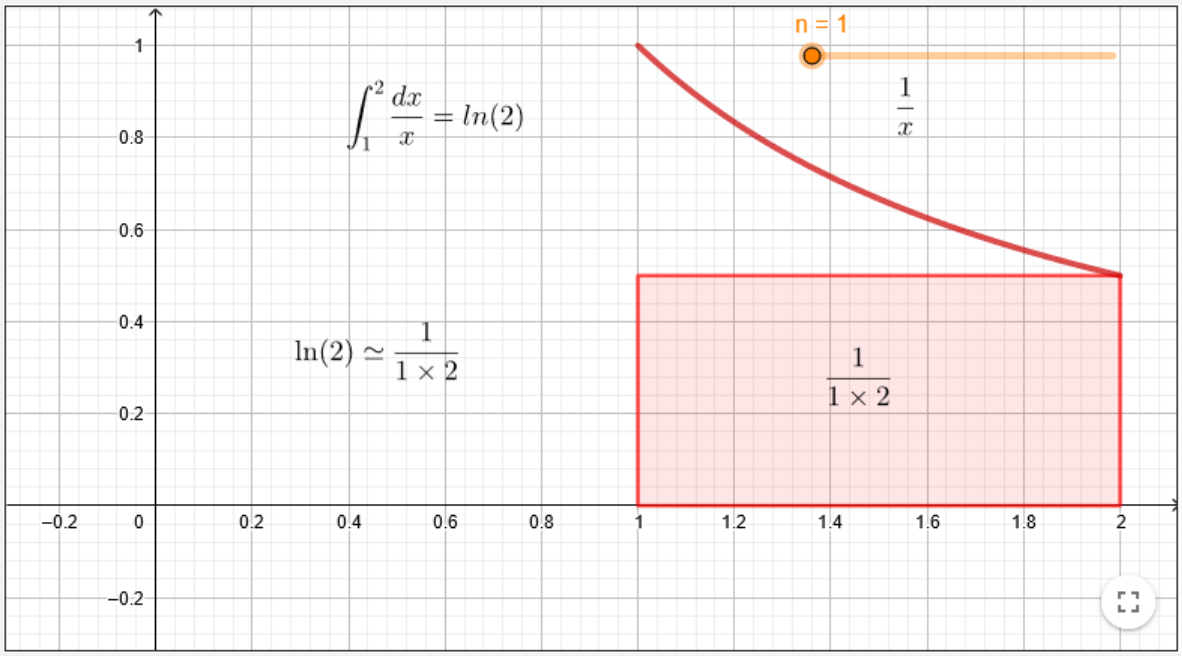
V . 1 - Principe

L'idée de cette méthode est de considérer la fonction $x \mapsto \ln x$ comme une primitive de $x \mapsto \dfrac{1}{x}$ et, en conséquence, que :

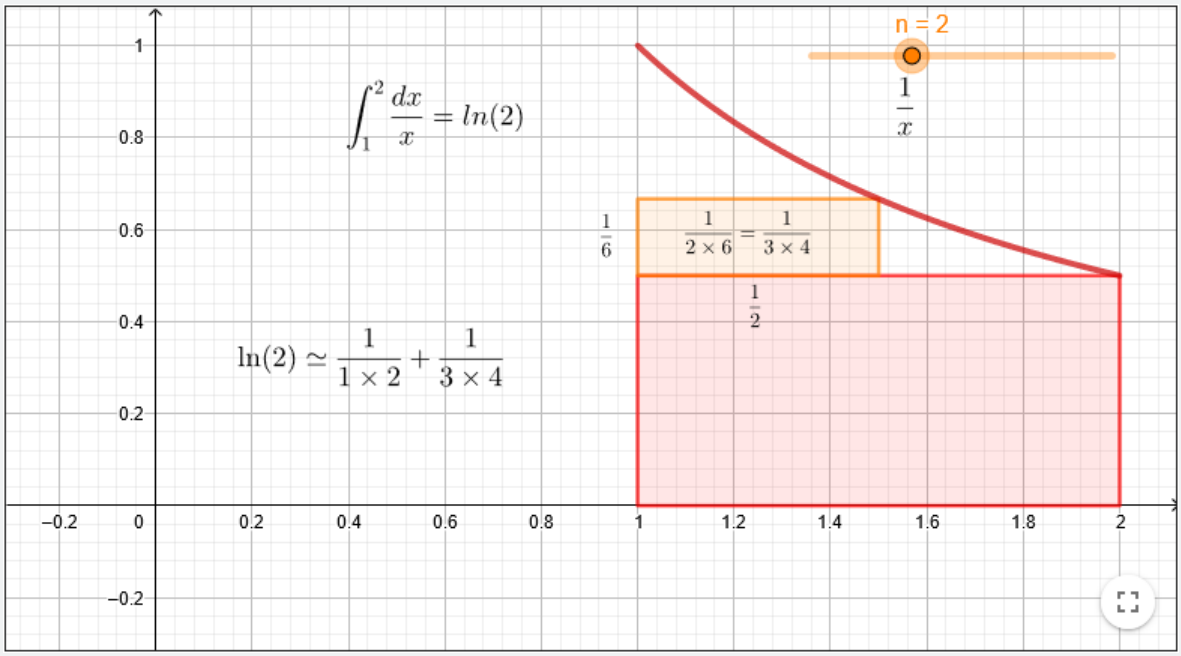
$$\int_1^2 \frac{1}{x}\,\mathrm{d}x = \ln 2 - \ln 1 = \ln 2.$$

Cette méthode consiste donc à approximer l'aire sous la courbe représentative de la fonction inverse sur $[1\,;2]$ d'une certaine manière, dont voici les explications :

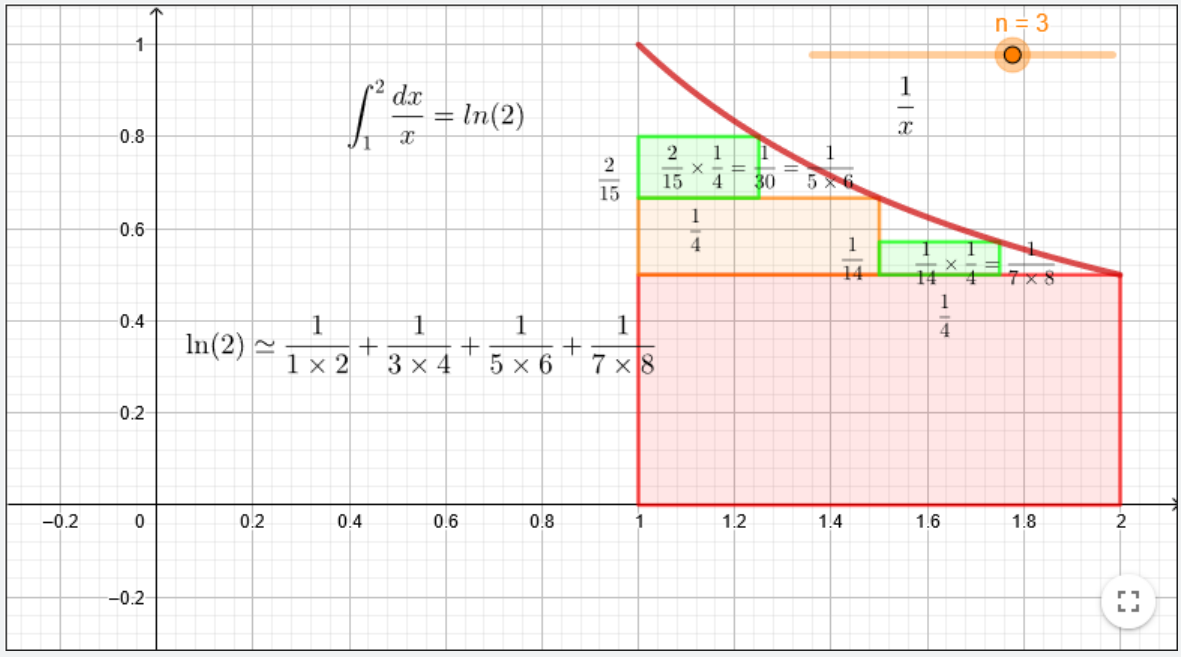
1 **Étape 1 :** on construit le rectangle de base $[1\,;2]$ et de hauteur $\dfrac{1}{2}$.



2 **Étape 2 :** on prend le milieu de $[1;2]$ et on construit le rectangle « coincé » entre le rectangle précédent et la courbe sur $\left[1;\frac{3}{2}\right]$:



3 **Étape 3 :** on prend le milieu des intervalles $\left[1;\frac{3}{2}\right]$ et $\left[\frac{3}{2};2\right]$, puis on construit un rectangle « coincé » entre les rectangles précédents et la courbe sur $\left[1;\frac{5}{4}\right]$ et $\left[\frac{3}{2};\frac{7}{4}\right]$:



En continuant ainsi, on construit de plus en plus de rectangles dont la somme des aires se rapproche de plus en plus de $\int_1^2 \frac{1}{x}\,dx$, donc de $\ln 2$.

V . 2 - Une première approche

On peut alors construire une fonction Python basée sur cette méthode, qui retourne la valeur de la somme trouvée :

Code Python 9-230

```
1  def brouncker(n):
2      aire = 0.5 # aire du premier rectangle
3      for k in range(1,n):
4          base = 1 / 2**k
5          X = [ 1 ]
6          x = 1
7          while x < 2:
8              x = x + base
9              X.append( x )
10
11         for i in range( 1 , len(X) , 2):
12             aire = aire + base * ( 1/X[i] - 1/X[i+1] )
13
14     return aire
```

`n` représente ici le nombre d'étapes souhaitées, $n \geqslant 1$, $n = 1$ étant l'étape où le premier rectangle est construit.
Je crée une boucle itérative sur `k` (donc `k` représentera le numéro de l'étape).
Pour chaque étape, je détermine la « base » des rectangles : c'est $\frac{1}{2^k}$.
Ensuite, je souhaite créer une liste `X` de toutes les subdivisions de $[1\,;2]$ par pas de $\frac{1}{2^k}$: c'est l'objet des lignes 5 à 9.
Une fois cette liste construite, je la parcours par pas de 2. En effet, chaque nouveau rectangle se construit sur une subdivision sur 2. Regardez à l'étape 3 : les subdivisions sont $\left[1\,;1+\frac{1}{4}\right]$, $\left[1+\frac{1}{4}\,;1+\frac{2}{4}\right]$, $\left[1+\frac{2}{4}\,;1+\frac{3}{4}\right]$ et $\left[1+\frac{3}{4}\,;1+\frac{4}{4}\right]$ et les rectangles verts sont sur $\left[1\,;1+\frac{1}{4}\right]$ et $\left[1+\frac{2}{4}\,;1+\frac{3}{4}\right]$.
L'expression « `base * ( 1/X[i] - 1/X[i+1] )` » sur la ligne 12 correspond à l'aire du rectangle ajouté.

On obtient alors par exemple :

```
>>> brouncker(15)
0.6931319220037134
>>> brouncker(20)
0.6931467037230511
```

Mais si d'aventure vous souhaitez mettre en argument une valeur de n supérieure à 23, le temps d'exécution sera assez long... J'ai en effet pris le parti de rendre le plus clair possible (à mes yeux) le programme, au détriment de sa complexité (qui a une incidence sur le temps d'exécution). En l'occurrence, il y a beaucoup de calculs (car beaucoup de puissances et de listes), d'où un temps d'exécution lent pour de grandes valeurs de n.

V . 3 - Une autre approche

Plutôt que de tenter de calculer la somme des aires des rectangles, penchons-nous sur la formule que l'on peut obtenir en observant le principe de la méthode.

- le premier rectangle a pour aire $1 \times \frac{1}{2}$, que l'on peut écrire aussi sous la forme $\frac{1}{1\times 2}$;
- le deuxième rectangle (orange) a pour aire $\frac{1}{2} \times \frac{1}{6}$, que l'on peut écrire aussi $\frac{1}{3\times 4}$;
- le troisième rectangle (le premier vert) a pour aire $\frac{1}{4} \times \frac{2}{15} = \frac{1}{5\times 6}$;
- le quatrième (vert) a pour aire $\frac{1}{4} \times \frac{1}{14} = \frac{1}{7\times 8}$;
- …

On peut conjecturer que l'aire totale (donc $\ln 2$) vaut :

$$\ln 2 = \frac{1}{1\times 2} + \frac{1}{3\times 4} + \frac{1}{5\times 6} + \frac{1}{7\times 8} + \cdots = \lim_{n\to+\infty} \sum_{k=0}^{n} \frac{1}{(2k+1)(2k+2)}.$$

D'où la fonction Python suivante, qui nécessite toute de même une valeur de n assez grande (supérieure à 10^5), mais pas trop (inférieure à 10^9) sinon le temps d'exécution est assez long.

Code Python 9-231

```
1 def brouncker(n):
2     somme = 0
3     for k in range(1,n+1,2):
4         somme += 1 / (k * (k+1) )
5
6     return somme
```

```
>>> brouncker(10**8)
0.6931471754736988
```

VI - Maths complémentaires : simuler la loi géométrique

On considère une variable aléatoire X suivant la loi binomiale $\mathscr{B}(n;p)$.
Si G représente le nombre de fois qu'il est nécessaire de répéter l'expérience de Bernoulli avant d'obtenir un succès, alors elle suit la loi géométrique de paramètre p.

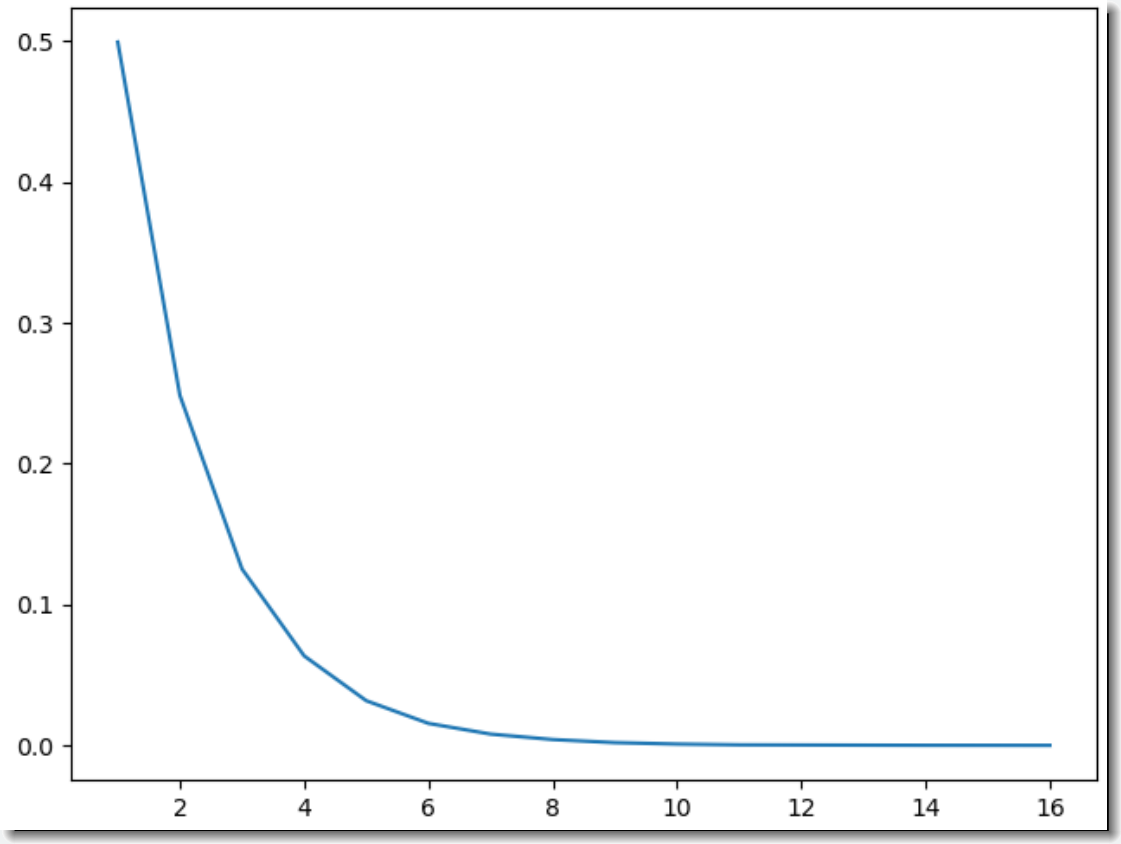
Pour simuler une loi géométrique, on va donc considérer l'expérience consistant à répéter un choix aléatoire entre 0 et 1 (échec et succès d'une expérience de Bernoulli) jusqu'à obtenir 1 (le succès) ; on note n le nombre de fois que nous avons répéter l'expérience de Bernoulli.
On répète ceci N fois en mettant les valeurs de n obtenues dans une liste L. Ensuite, on calcule la fréquence d'apparition de tous les nombres de L.

Code Python 9-232

```
1 from random import randint
2 from matplotlib.pyplot import plot, show
```

```
3
4  L = []
5  N = 10**5
6
7  for i in range(N):
8      n = 1
9      while randint(0,1) == 0:
10         n += 1
11
12     L.append(n)
13
14 X = []
15 Y = []
16
17 for k in range( 1 , max(L) + 1 ):
18     X.append(k)
19     Y.append( L.count(k) / N )
20
21 plot(X,Y)
22 show()
```

On obtient alors un graphique nous donnant la probabilité que G = k, pour k entier supérieur ou égal à 1.

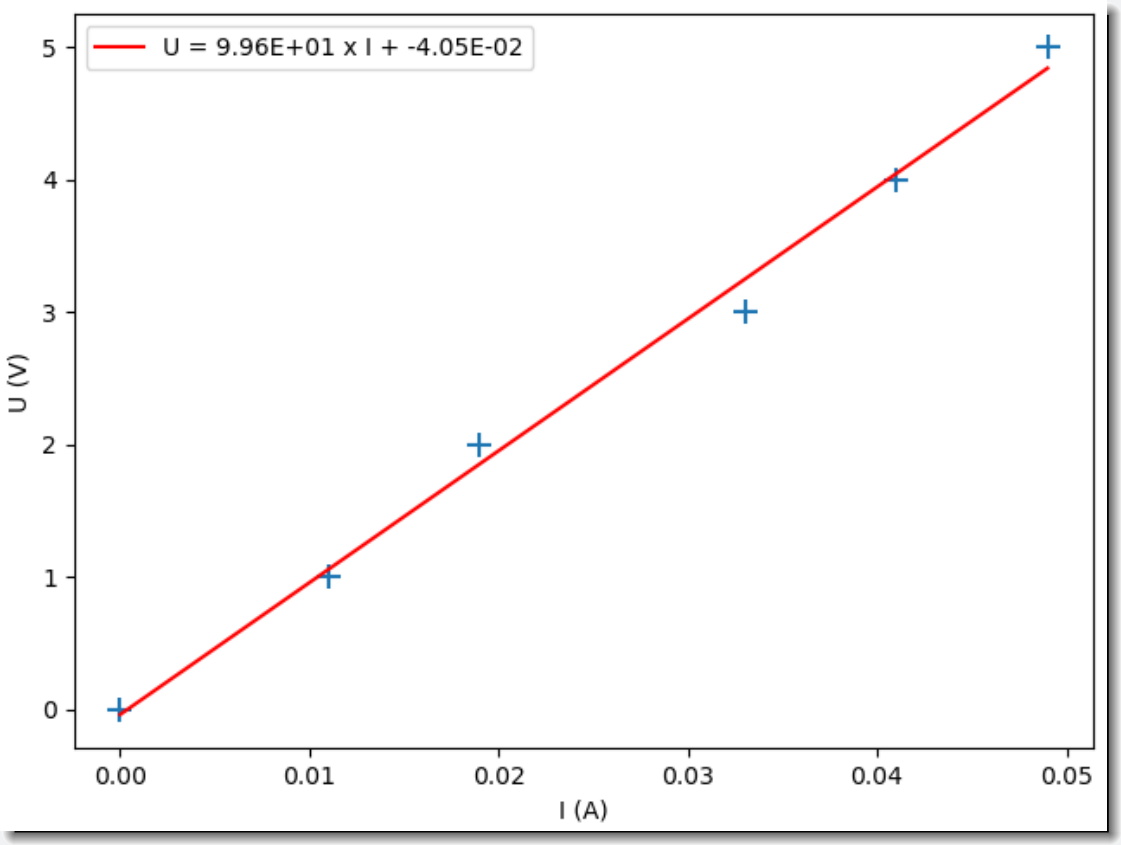


VII - Math complémentaires : droite de régression linéaire

Considérons une expérience de physique : on relève la tension et l'intensité aux bornes d'un dipôle ; on cherche à établir une équation de la droite de régression de cette série statistique.
Nous allons faire appel au module `numpy` :

Code Python 9-233

```
1  from numpy import polyfit, array
2  from matplotlib.pyplot import plot, show, scatter, legend, xlabel, ylabel
3
4  U = array( [0 , 1 , 2 , 3 , 4 , 5] )
5  I = array( [0, 0.011 , 0.019 , 0.033 , 0.041 , 0.049] )
6
7  modele = polyfit(I,U,1)
8  equation = ("U = {} x I + {}".format( format(modele[0],".2E") ,
       format(modele[1],".2E")))
9
10 scatter(I,U,marker='+',s=100) # placement des points
11 plot(I , modele[0]*I+modele[1] , color='red' , label = equation)
12
13 xlabel("I (A)")
14 ylabel("U (V)")
15 legend()
16 show()
```



VIII - Maths expertes : calcul du PGCD avec l'algorithme d'Euclide

Un programme basique

Vous trouverez sans doute beaucoup de propositions sur Internet de cet algorithme car c'est un exemple classique. En voici un :

Code Python 9-234

```
1  def pgcd(a,b) :
2      while a%b != 0 :
3          a, b = b, a%b
4      return b
```

On a par exemple :

```
>>> pgcd(1023,52)
6
```

Une autre fonction avec affichage des étapes

Je vous propose ici un algorithme permettant d'afficher les étapes :

Code Python 9-235

```
1  def euclide(a,b):
2      if b > a:
3          a,b = b,a
4
5      r = -1
6      while r !=0 :
7          q = a // b
8          r = a % b
9          print('{:3} = {:2} x {:2} + {}' . format(a,q,b,r))
10         a = b
11         b = r
```

On a par exemple :

```
>>> euclide(1023,52)

1023 = 19 x 52 + 35
 52 =  1 x 35 + 17
 35 =  2 x 17 + 1
 17 = 17 x  1 + 0
```

IX - Maths expertes : un couple de Bézout

Il s'agit ici de trouver un couple solution de l'équation $ax + by = 1$, avec $\text{pgcd}(a\,;b) = 1$.

Forme récursive

Code Python 9-236

```
def bezout(a,b):
    if b == 0:
        return 1,0
    else:
        u , v = bezout(b , a % b)
        return v , u - (a//b)*v
```

Par exemple, pour trouver une solution de l'équation $13x + 5y = 1$, on tape :

```
>>> bezout(13,5)
(2, -5)
```

Forme plus naturelle

Code Python 9-236

```
def bezout(a,b):
    u, v, x, y = 1, 0, 0, 1
    while b > 0:
        r = a % b
        q = a // b
        u1, v1 = u, v # variables temporaires
        u, v = x, y # u et v deviennent x et y
        x = u1 - q*x
        y = v1 - q*y
        a, b = b, r

    return u, v
```

Cette dernière fonction s'inspire de l'algorithme d'Euclide pris à l'envers.

X - Maths expertes : crible d'Eratosthène

Le crible d'Eratosthène consiste à prendre la liste de tous les nombres entiers de 2 à n, n étant un entier supérieur ou égal à 2, et à supprimer tous les nombres qui n'ont aucun diviseur.
On obtient ainsi la liste de tous les nombres premiers inférieurs ou égaux à n.

Code Python 9-236

```
def eratosthene(n):
    P = [ ]
    for i in range(2,n+1):
        if len(P) == 0:
            P.append(i)
        else:
            prem = True
            for k in P:
                if i % k == 0:
                    prem = False
            if prem == True:
                P.append(i)

    return P
```

On crée une liste `P` devant contenir les nombres premiers cherchés.
On parcourt tous les nombres entiers i de 2 à n : si on est sur « 2 », on l'insère dans `P` sinon, on parcourt la liste `P` (qui ne contient que des nombres premiers) et si i est divisible par l'un de ses nombres, c'est qu'il n'est pas premier donc on ne l'insère pas dans `L`.

XI - Maths expertes : décomposition en produit de facteurs premiers

L'idée ici est de créer une fonction `decompose(n)`, où l'argument n est un entier naturel, qui détermine la liste des facteurs premiers de n ainsi que le nombre de fois que n peut être divisé par ces facteurs (donc leur exposant dans la décomposition en produit de facteurs premiers de n).

On commence donc par voir si n est divisible par $i = 2$: si tel est le cas, on divise n autant de fois que possible par 2 pour déterminer l'exposant k du facteur 2 dans la décomposition.

Une fois terminé, n vaut $\frac{n}{2^k}$, et on recommence avec $i = 3$.

On continue ainsi jusqu'à ce que $n = 1$.

On a alors la fonction page suivante.

Code Python 9-237

```
def decompose (n):
    factors_list = []
    i = 2
    val = n
    while n > 1:
        exposant = 0
        while n % i == 0:
            exposant = exposant + 1
            n = n/i

        if exposant != 0:
            factors_list.append( (i,exposant) )

        i = i+1

    return factors_list
```

```
>>> decompose(420)
[(2, 2), (3, 1), (5, 1), (7, 1)]
```

Cette fonction retourne la liste de tous les facteurs premiers du nombre n avec leur exposant dans la décomposition en produit de facteurs premiers.
Sur l'exemple ci-dessus, on a donc :

$$420 = 2^2 \times 3^1 \times 5^1 \times 7^1.$$

Suites

Exercice 9.1 (relation de récurrence)

On définit la suite (u_n) pour tout entier naturel n non nul par :

$$\begin{cases} u_0 = 0 \\ u_{n+1} = \left(1 + \frac{1}{n+1}\right) u_n + 1 \end{cases}$$

Écrire un programme Python qui calcule et affiche les 100 premiers termes de cette suite après u_0.

Solution page 174

Exercice 9.2 (relation de récurrence)

On définit la suite (u_n) pour tout entier naturel n non nul par :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = n u_n^2 - \frac{u_n}{2n+3} \end{cases}$$

Écrire un programme Python qui calcule et affiche les 100 premiers termes de cette suite après u_0.

Solution page 174

Exercice 9.3 (suite de Fibonacci)

La suite de Fibonacci est définie par ses premiers termes $F_0 = F_1 = 1$ et par la relation de récurrence :

$$\forall\, n \geqslant 0, \qquad F_{n+2} = F_{n+1} + F_n.$$

1 Écrire un programme Python permettant d'afficher les 50 premiers termes de cette suite (donc de F_0 à F_{49}) sans utiliser de liste.

2 On souhaite observer le quotient $\dfrac{F_{n+1}}{F_n}$ pour $n \geqslant 0$.

Modifier la réponse à la question précédente pour afficher ces quotients, pour n variant de 0 à 49.

Solution page 175

Exercice 9.4 (les suites de Héron)

Pour un réel a strictement positif quelconque, on considère la suite $(u_n(a))_{n\geqslant 0}$ définie pour tout entier naturel n par :

$$\begin{cases} u_0(a) & >0 \\ u_{n+1}(a) & =\dfrac{1}{2}\left(u_n+\dfrac{a}{u_n}\right) \end{cases}$$

1 Compléter la fonction Python suivante afin que l'instruction :

```
>>> heron(4,7,10)
```

retourne les $n+1$ premiers termes de la suite $(u_n(4))_{n\geqslant 0}$ dans une liste.

Code Python 9-242

```
def heron(a,u,n):
    if u <=0:
        return "Le premier terme doit être strictement positif."
    elif a <= 0:
        return "La valeur de 'a' doit être strictement positive."
    else:
        L = [ ... ]
        k = 0
        while k < ...:
            u = ...
            L.append(...)
            k = ...
        return ...
```

2 L'instruction renvoie alors la liste suivante :

```
[7, 3.7857142857142856, 2.4211590296495955, 2.0366301688743387,
   2.0003294091613366, 2.0000000271231317, 2.0, 2.0, 2.0, 2.0, 2.0]
```

Faites deux conjectures à partir de ces valeurs.

On se propose d'étudier mathématiquement la suite $(u(a)_n)$ pour une valeur de $a>0$.
Pour cela, on introduit la fonction f définie pour tout réel x strictement positif par :

$$f(x)=\frac{1}{2}\left(x+\frac{a}{x}\right)$$

3 Calculer $f'(x)$, puis donner les variations de f sur $]0;+\infty[$ en fonction de a.

4 En déduire que $u_1\geqslant\sqrt{a}$, quelle que soit la valeur de $u_0>0$.

5 Montrer que pour tout réel $x\geqslant\sqrt{a}$, $f(x)\leqslant x$.

6 Montrer alors par récurrence que pour tout entier naturel $n\geqslant 1$, $\sqrt{a}\leqslant u_{n+1}\leqslant u_n$.

7 Déduire que $(u_n(a))_{n\geqslant 0}$ converge. On note ℓ sa limite.

8 On admet que $\ell=f(\ell)$. Déterminer alors la limite de$(u_n(a))_{n\geqslant 0}$.

Solution page 176

Exercice 9.5 (méthode de Newton avec le logarithme népérien)

On considère la fonction f définie sur $]-1;+\infty[$ par :

$$f(x) = 3x - 2 + \ln(x+1).$$

On considère la suite (u_n) définie pour tout entier naturel n par :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = u_n - \dfrac{f(u_n)}{f'(u_n)} \end{cases}$$

On admet que cette suite converge vers l'unique solution de l'équation $f(x) = 0$.
Écrire un programme en Python permettant d'afficher les premiers termes de cette suite jusqu'à ce que la différence entre deux termes consécutifs soit inférieure à 10^{-12}.

Solution page 177

Continuité

Exercice 9.6 (dichotomie)

On considère l'algorithme de dichotomie suivant :

```
delta ← 1
e ← 0,00001
f est une fonction continue sur [a;b]
Tant que delta > e faire:
  m = a + (b-a)/2
  delta = |b-a|
  Si f(m) = 0 alors:
    Affiche m
  Sinon:
    Si f(a)×f(m) > 0 alors:
      a ← m
    Sinon:
      b ← m
    Fin du Si
  Fin du Si
Fin du Tant que
Afficher a, b
```

En s'aidant de cet algorithme, écrire une fonction Python `dichotomie(a,b,e)` qui permet de renvoyer un encadrement d'amplitude e de la solution à l'équation $f(x) = 0$ sur $[a;b]$, en prenant $f(x) = \sqrt{(\ln x)^2 + \mathrm{e}^x} - 2x$, $e = 10^{-7}$, $(a;b) = (0,5;1)$ d'une part, $(a;b) = (4;5)$ d'autre part.

Solution page 178

Intégration

Exercice 9.7 (méthode des rectangles)

On considère l'algorithme suivant, qui permet d'obtenir un encadrement de $\int_a^b f(x)\,\mathrm{d}x$, où f est une fonction continue sur $[a;b]$:

```
longueur ← (b - a)/n
inf ← 0
sup ← 0
Pour k prenant ses valeurs de 0 à n-1 faire:
   inf ← inf + longueur*f(a+k*(longueur))
   sup ← sup + longueur*f(a+(k+1)*(longueur))
Fin du Pour
Afficher inf et sup
```

En s'inspirant de cet algorithme, écrire une fonction Python `integrale(a,b,n)` qui renvoie un encadrement de $\int_a^b f(x)\,\mathrm{d}x$, en divisant $[a;b]$ en n segments identiques.

On prendra pour exemple la fonction $f(x) = \sqrt{(\ln x)^2 + \mathrm{e}^x} - 2x$, $a = 0.739180505$, $b = 4.251385$ et $n = 100$.

Solution page 179

Dénombrement

Exercice 9.8 (fonction shuffle du module random)

La fonction `shuffle` du module `random` de Python permet de mélanger les éléments d'une liste.
Par exemple :

```
>>> from random import shuffle
>>> L = [1,2,3,4,5,6,7,8,9,10]
>>> shuffle(L)
>>> L
[5, 10, 4, 1, 9, 8, 7, 3, 2, 6]
```

Combien d'affichages distincts est-il possible d'obtenir sur cet exemple ?

Solution page 179

Exercice 9.9 (listes des noms et prénoms des élèves)

Un enseignant dispose de deux listes Python nommées `noms` et `prenoms` contenant respectivement tous les noms et prénoms distincts des élèves d'une classe.
Par exemple, le code suivant :

Code Python 9-248

```
1 from itertools import product
2
3 noms = [ 'DUPOND' , 'DUPONT' , 'DURAND' , 'DUMONT' ]
4 prenoms = [ 'Jean' , 'Anne' , 'Vincent' ]
5
6 print ( list(product( noms, prenoms) ) )
```

donne le résultat suivant :

```
[('DUPOND', 'Jean'), ('DUPOND', 'Anne'), ('DUPOND', 'Vincent'),
   ('DUPONT', 'Jean'), ('DUPONT', 'Anne'), ('DUPONT', 'Vincent'),
   ('DURAND', 'Jean'), ('DURAND', 'Anne'), ('DURAND', 'Vincent'),
   ('DUMONT', 'Jean'), ('DUMONT', 'Anne'), ('DUMONT', 'Vincent')]
```

Si les listes `noms` et `prenoms` comportent respectivement 30 noms et 25 prénoms, combien d'éléments comporte le résultat affiché ?

Solution page 179

Exercice 9.10 (compétition de danse)

L'organisatrice d'une compétition de danse souhaite afficher tous les podiums possibles (3 compagnies de danse gagnantes). Elle possède la liste de toutes les compagnies de danses inscrites, qu'elle a nommé `compagnies`.
Elle utilise alors le code Python suivant :

```
>>> from itertools import permutations
>>> len(compagnies)
25
>>> print( len( list(permutations(compagnies,3) ) ) )
```

Quel est le résultat affiché ?
On rappelle que `len(L)` désigne le nombre d'éléments contenus dans la liste `L`.

Solution page 180

Exercice 9.11 (grilles de Loto©)

Monsieur Laguigne en a marre de perdre au Loto©. Il décide alors de faire un programme Python qui génère tous les résultats possibles.
Rappelons que le jeu du Loto© consiste à choisir 5 numéros entre 1 et 49, puis un numéro « chance » à mettre à la fin.

Voici par exemple un résultat fourni par la Française des Jeux, société organisatrice du jeu :

Pour cela, il voudrait utiliser le code suivant :

Code Python 9-249

```
1  from itertools import combinations, product
2
3  grille = [ n for n in range(1,50) ]
4
5  resultats_finaux = list()
6
7  for resultat in list( combinations( grille , 5 ) ):
8      grille_restante = [ n for n in range(1,50) if n not in resultat ]
9      r = list( resultat )
10     for num in grille_restante:
11         resultats_finaux.append( r + [num] )
```

Il souhaite ainsi parcourir la liste de toutes les combinaisons possibles de 5 nombres parmi 49 (ligne 7) et pour chacune d'elles, il construit une grille des numéros non encore choisis (ligne 8). Il convertit ensuite la combinaison en liste (ligne 12) puis il parcourt la grille des nombres restants : pour chacun d'eux, il considère la liste des nombres choisis à laquelle il ajoute le nombre (le numéro chance).
Par exemple, les deux premiers résultats sont :

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 7]
```

1. Calculer le nombre d'éléments de la liste `list( combinations( grille , 5 ) )`.
2. En déduire le nombre d'éléments de la liste `resultats_finaux`.
3. En lançant son programme, la console affiche le message suivant :

```
resultats_finaux.append( r + [num] )
MemoryError
```

Avancez une explication.

Solution page 180

Probabilités

Exercice 9.12

L'expérience $\mathscr{E}$ consiste à choisir un nombre au hasard sur l'intervalle $[0;1[$.
On note X la variable aléatoire représentant le nombre de fois que l'on répète $\mathscr{E}$ pour que la somme des nombres choisis soit strictement supérieure à 1.

Écrire une fonction Python `moyenne(n)` permettant de simuler n variables aléatoires X et renvoyant leur moyenne, puis tester cette fonction pour des valeurs de n de la forme $n = 10^k$ pour k entier de 2 à 7.

Aide : la fonction `random()` du module `random` donne un nombre aléatoire dans l'intervalle $[0;1[$.

Solution page 181

Exercice 9.13 (simulation de lancers de dés)

1. Compléter la fonction Python suivante afin qu'elle affiche une estimation de la probabilité d'obtenir deux fois de suite le « 6 » sur n lancers d'un dé cubique.

Code Python 9-251

```
from random import randint

def lancer(n):
    L = []
    c = 0

    for k in range(n):
      # on insère dans la liste L un nombre entier aléatoire entre 1 et 6
        L.append( randint(1,6) )

    ...
```

2. Exécuter cette fonction pour $n = 10^k$, k étant un entier de 2 à 7.
3. Calculer mathématiquement cette probabilité.

Solution page 182

Exercice 9.14 (inégalité de concentration)

On considère l'expérience $\mathscr{E}$ consistant à choisir un nombre entier compris entre 1 et 10.
On répète 10 fois de façon indépendante l'expérience $\mathscr{E}$ et on note X la variable aléatoire représentant le nombre de multiples de 3 obtenus lors de ces 10 expériences.

1. Quelle est la loi suivie par X ?
2. Donner alors $E(X)$ et $\sigma(X)$.

3 On répète n fois l'expérience $\mathscr{E}$, $n \in \mathbb{N}^*$. Compléter la fonction Python suivante qui simule ces n expériences et retourne le nombre moyen de multiples de 3 obtenus.

Code Python 9-254

```
from random import randint

def simul(n):
    total = 0
    for i in range(n):
        c = 0
        for k in range(10):
            a = randint(1,10)
            if ...:
                c = c + 1

        total = ...

    return ...
```

4 À partir de quelle valeur de n peut-on être certain·e·s que la valeur retournée par cette fonction est une approximation de E(X) à 10^{-3} près ?

Solution page 183

Exercice 9.15 (inégalité de concentration)

Une expérience $\mathscr{E}_n$ consiste à lancer un dé cubique (six faces) n fois, les lancers étant indépendants les uns des autres.
On note X la variable aléatoire représentant le nombre de chiffres pairs obtenus.

1 Quelle est la loi de probabilité de X ?

2 Quelle en est alors son espérance et son écart-type ?

3 Compléter la fonction Python suivante afin qu'elle simule l'expérience $\mathscr{E}_n$ et renvoie le nombre de chiffres pairs obtenus.

Code Python 9-256

```
from random import randint
def simul(n):
    total = ...
    for k in range(...):
        D = ...
        if D ... :
            total = ...
    return total
```

4 Écrire une fonction Python `diff(n)` qui retourne la différence entre le nombre de chiffres pairs obtenus et l'espérance de X.

Par exemple, si la simulation trouve 512 chiffres pairs sur 1 000 lancers, on devra avoir :

```
>>> diff(1000)
12
```

5 Écrire une fonction Python `echant(N,n)` qui calcule la moyenne des résultats renvoyés si on fait appel N fois à la fonction `diff(n)`.

6 Écrire une fonction Python `proportion(N)` qui simule N expériences $\mathscr{E}_{1\,000}$ et qui renvoie la proportion des cas où l'écart entre m et E(X) est inférieur ou égal à $\dfrac{2\sigma(X)}{\sqrt{1\,000}}$, où m est le nombre moyen de chiffres pairs obtenus lors d'une simulation de 1 000 lancers de dé.

Solution page 184

Corrigé de l'exercice 9.1 page 165

Attention à la formule de récurrence : quand on souhaite écrire un programme Python pour calculer les termes consécutifs, comme ici, il vaut mieux voir la relation de récurrence comme ceci :

$$\begin{cases} u_0 = 0 \\ u_{n+1} = \left(1 + \frac{1}{n+1}\right) u_n + 1 \end{cases} \iff \begin{cases} u_0 = 0 \\ u_n = \left(1 + \frac{1}{n}\right) u_{n-1} + 1 \end{cases}$$

On s'aperçoit alors que c'est l'inverse de l'indice du terme courant qui intervient dans le calcul.

On connaît le nombre de termes à calculer, donc on utilise une boucle itérative :

Code Python 9-260

```
u = 0 # terme initial

for n in range(1,101):
    # on calcule le terme d'indice n
     u = (1 + 1/n) * u + 1
     print(u)
```

Corrigé de l'exercice 9.2 page 165

Comme dans l'exercice 9.1, il est conseillé d'écrire la relation de récurrence de la suite d'une autre manière :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = nu_n^2 + \dfrac{u_n}{2n+3}, \end{cases} \quad n \geqslant 0 \iff \begin{cases} u_0 = 1 \\ u_n = (n-1)u_{n-1}^2 + \dfrac{u_{n-1}}{2(n-1)+3}, \end{cases} \quad n \geqslant 1$$

$$\iff \begin{cases} u_0 = 1 \\ u_n = (n-1)u_{n-1}^2 + \dfrac{u_{n-1}}{2n+1}, \end{cases} \quad n \geqslant 1$$

On en déduit le programme suivant :

Code Python 9-261

```
u = 1

for n in range(1,101):
    u = (n-1)*u**2 + u/(2*n+1)
    print(u)
```

Attention 9

Il me semble réellement important de transformer la relation de récurrence initiale pour ne pas se tromper dans les valeurs de n.

Corrigé de l'exercice 9.3 page 165

1 Voici un programme possible qui permet d'afficher les 50 premiers termes de la suite de Fibonacci sans liste :

Code Python 9-262

```
a, b = 1, 1
print(a)
print(b)

for n in range(48):
    a , b = b, a+b
    print(b)
```

2 Afin d'afficher les quotients $\frac{F_{n+1}}{F_n}$ pour tout entier compris entre 0 et 49, on modifie la solution précédente ainsi :

Code Python 9-263

```
a, b = 1, 1
print(b,'/',a,'=',b/a)

for n in range(48):
    a , b = b, a+b
    print(b,'/',a,'=',b/a)
```

En exécutant ce dernier programme, on s'aperçoit que les dernières lignes sont :

```
24157817 / 14930352 = 1.618033988749897
39088169 / 24157817 = 1.618033988749894
63245986 / 39088169 = 1.6180339887498951
102334155 / 63245986 = 1.6180339887498947
165580141 / 102334155 = 1.618033988749895
267914296 / 165580141 = 1.618033988749895
433494437 / 267914296 = 1.618033988749895
701408733 / 433494437 = 1.618033988749895
1134903170 / 701408733 = 1.618033988749895
1836311903 / 1134903170 = 1.618033988749895
2971215073 / 1836311903 = 1.618033988749895
4807526976 / 2971215073 = 1.618033988749895
7778742049 / 4807526976 = 1.618033988749895
12586269025 / 7778742049 = 1.618033988749895
```

On peut alors constater que les quotients se rapprochent d'une valeur limite.
En effet, on peut démontrer mathématiquement que :

$$\lim_{n\to+\infty} \frac{F_{n+1}}{F_n} = \frac{1+\sqrt{5}}{2} = \varphi$$

qui est le *nombre d'or*.

Corrigé de l'exercice 9.4 page 166

1 La fonction Python complétée est la suivante :

Code Python 9-264

```
def heron(a,u,n):
    if u <=0:
        return "Le premier terme doit être strictement positif."
    elif a <= 0:
        return "La valeur de 'a' doit être strictement positive."
    else:
        L = [ u ]
        k = 0
        while k < n:
            u = 0.5 * (u + a / u)
            L.append(u)
            k = k + 1
        return L
```

2 On peut constater :

- d'une part que la suite $(u(4)_n)$ semble être décroissante ;
- d'autre part, que la suite $(u(4)_n)$ semble converger vers 2.

3 On a : $f'(x) = \frac{1}{2}\left(1 - \frac{a}{x^2}\right) = \frac{x^2 - a}{2x^2}$.

$x^2 - a$ est un polynôme du second degré dont les racines sont $\sqrt{a}$ et $-\sqrt{a}$, d'où le tableau suivant :

x	0		$\sqrt{a}$		$+\infty$
$f'(x)$	‖	$-$	0	$+$	
f	‖	↘	$\sqrt{a}$	↗	

4 D'après les variations de f, si $x \in [0;\sqrt{a}[$ alors $f(x) \geqslant \sqrt{a}$.
Or, $u_1 = f(u_0)$ donc si $u_0 \in]0;\sqrt{a}]$ alors $u_1 \geqslant \sqrt{a}$.
De plus, si $u_0 \geqslant \sqrt{a}$, d'après les variations de f, $f(u_0) \geqslant \sqrt{a}$.
Ainsi, quelle que soit la valeur de $u_0 > 0$, $u_1 \geqslant \sqrt{a}$.

5 Montrer que pour tout réel $x \geqslant \sqrt{a}$, $f(x) \leqslant x$.
Considérons alors la fonction $g(x) = f(x) - x$:

$$g(x) = \frac{1}{2}\left(\frac{a}{x} - x\right).$$

Sa dérivée est alors :

$$g'(x) = \frac{1}{2}\left(-\frac{a}{x^2} - 1\right)$$

donc $g'(x) < 0$ sur $[\sqrt{a};+\infty[$, ce qui signifie que g est strictement décroissante sur $[\sqrt{a};+\infty[$.
De plus, $g(\sqrt{a}) = 0$ donc $g(x) \leqslant 0$ sur $[\sqrt{a};+\infty[$. On a alors $f(x) - x \leqslant 0$, soit $f(x) \leqslant x$.

6 Démontrons par récurrence que pour tout entier naturel $n \geqslant 1$, $\sqrt{a} \leqslant u_{n+1} \leqslant u_n$.

- **Initialisation.**
  $u_1 > \sqrt{a}$ et $u_2 = f(u_1) \leqslant u_1$ d'après la question précédente.
  On a donc bien $\sqrt{a} \leqslant u_2 \leqslant u_1$.
  L'initialisation est alors réalisée.
- **Hérédité.**
  Supposons que pour un entier $k > 1$, $\sqrt{a} \leqslant u_{k+1} \leqslant u_k$.
  D'après la question **3**, f est strictement croissante sur $]\sqrt{a}; +\infty[$ donc :

$$f(\sqrt{a}) \leqslant f(u_{k+1}) \leqslant f(u_k)$$

  soit :

$$\sqrt{a} \leqslant u_{k+2} \leqslant u_{k+1}.$$

  L'hérédité est alors vérifiée.

Ainsi, d'après le principe de récurrence, pour tout entier naturel $n \geqslant 1$, $\sqrt{a} \leqslant u_{n+1} \leqslant u_n$.

7 La suite $(u(a)_n)$ est, d'après la question précédente, décroissante et minorée. Donc d'après le théorème de convergence des suites monotones, elle converge.

8 On admet que $\ell = f(\ell)$, soit $f(\ell) - \ell = 0$. Donc :

$$\begin{aligned} g(\ell) = 0 &\iff \frac{a}{\ell} - \ell = 0 \\ &\iff a - \ell^2 = 0 \\ &\iff \ell = -\sqrt{a} \text{ ou } \ell = \sqrt{a}. \end{aligned}$$

Or, $\ell > 0$ car $u(a)_n > 0$. Donc $\ell = \sqrt{a}$.
Ainsi, la limite de $(u(a)_n)$ est égale $\sqrt{a}$.

Corrigé de l'exercice 9.5 page 167

Un programme possible est le suivant :

Code Python 9-265

```
from math import log

g = lambda x : x-(x+1)*(3*x-2+log(x+1))/(3*x+4)

u = 1
v = f(1)

while abs(u-v) > 10**(-12):
    u = v
    v = g(u)
    print(v)
```

Ici, la ligne « `g = lambda x : x-(x+1)*(3*x-2+log(x+1))/(3*x+4)` » permet de définir une fonction g telle que $g(x) = x - \dfrac{f(x)}{f'(x)}$ car $u_{n+1} = g(u_n)$. Mais on aurait pu aussi écrire :

Code Python 9-266

```
def g(x):
    return x-(x+1)*(3*x-2+log(x+1))/(3*x+4)
```

C'est d'ailleurs cette syntaxe que j'ai choisi d'adopter dans tous les autres programmes Python car elle est plus naturelle... Après tout, nous ne sommes pas en NSI!
On initialise ensuite deux termes consécutifs $u = u_0$ et $v = u_1$, puis on calcule les termes successifs tant que $|u - v| > 10^{-12}$.
Le programme s'arrête assez vite (seulement 4 affichages!) et donne :

```
0.5258167668648809
0.5258221966299376
0.5258221966316697
0.5258221966316697
```

Cela signifie que la convergence est très rapide (ce qui est d'ailleurs une caractéristique de la méthode de Newton).

Corrigé de l'exercice 9.6 page 167

Une fonction possible est la suivante :

Code Python 9-267

```
from math import exp, log, sqrt

def f(x):
    return sqrt( log(x)**2 + exp(x) ) - 2*x

def dichotomie(a,b,e):
    delta = 1
    while delta > e:
        m = a + (b - a) / 2
        delta = abs(b - a)
        if f(m) == 0:
            return m
        elif f(a) * f(m)  > 0:
            a = m
        else:
            b = m

    return a, b
```

Dans la fonction `dichotomie`, je n'ai fait que traduire l'algorithme donné dans l'énoncé en

langage Python.
On obtient alors :

```
>>> dichotomie(0.5, 1, 10**(-7))
(0.7391805052757263, 0.7391805350780487)
>>> dichotomie(4, 5, 10**(-7))
(4.2513850927352905, 4.251385122537613)
```

Corrigé de l'exercice 9.7 page 168

Une fonction possible est la suivante :

Code Python 9-268

```
1  from math import exp, log, sqrt
2
3  def f(x):
4      return sqrt( log(x)**2 + exp(x) ) - 2*x
5
6  def integrale(a,b,n):
7      longueur = (b - a) / n
8      aire_inf , aire_sup = 0 , 0
9
10     for k in range (n):
11         aire_inf = aire_inf + longueur * f(a + k * longueur )
12         aire_sup = aire_sup + longueur * f(a + (k+1) * longueur )
13
14     return aire_inf , aire_sup
```

```
>>> a, b = 0.739180505, 4.251385
>>> integrale(a,b,100)
(-3.3417134238835544, -3.3417134345212043)
```

Corrigé de l'exercice 9.8 page 168

Il s'agit ici de savoir combien de permutations il est possible d'obtenir sur une liste à 10 éléments.

D'après le cours de mathématiques, il y en a 10!.

Il y a donc 10! affichages distincts possibles sur l'exemple donné.

Corrigé de l'exercice 9.9 page 169

Il s'agit ici d'un produit cartésien de deux ensembles (les deux listes Python). Si les deux listes contiennent n et p éléments, alors le résultat final donnera une liste à $n \times p$ éléments.

Ainsi, pour notre exemple, le résultat contiendra $30 \times 25 = 750$ éléments.

Corrigé de l'exercice 9.10 page 169

Il s'agit ici de savoir combien de podiums de 3 gagnantes on peut obtenir sur un ensemble à 25 éléments (en effet, la console Python nous affiche pour `len(compagnies)` le nombre « 25 », ce qui signifie qu'il y a 25 éléments dans la liste `compagnies`).
D'après le cours de mathématiques, il s'agit donc du nombre d'arrangements de 3 éléments parmi 25, qui est égal à :

$$\frac{25!}{(25-3)!} = 13\,800.$$

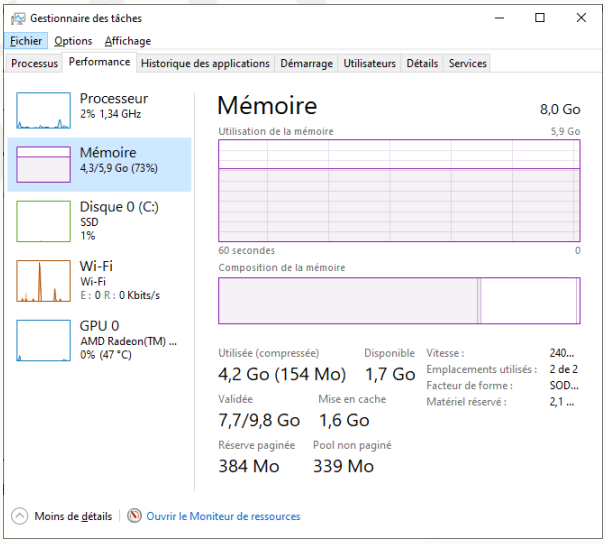
Le résultat affiché dans la console est alors `13800`.

Corrigé de l'exercice 9.11 page 169

1 Il s'agit ici de trouver le nombre de combinaisons de 5 éléments parmi 49 que l'on peut obtenir, donc de calculer $\binom{49}{5}$; on trouve 1 906 884.

2 La liste `resultats_finaux` est le résultat du produit cartésien de la liste `list( combinations( grille , 5 ) )` et de la liste `grille_restante`, qui compte $49 - 5 = 44$ éléments. Son cardinal est donc le produit de 1 906 884 par 44.
Il y a donc 83 902 896 éléments dans la liste `resultats_finaux`.

3 Chacun des éléments de `resultats_finaux` est un tuple de 6 nombres. Il faut donc stocker en mémoire $83\,902\,896 \times 6 = 503\,417\,376$ nombres.
Chaque nombre est représenté sur 1 octet (8 bits). Ainsi, il nous faut disposer de $503\,417\,376 \times 8 = 4\,027\,339\,008$ bits, ce qui correspond à un peu moins de 4 Go.
En théorie, il n'y aurait donc pas de problème de mémoire sur un ordinateur personnel classique, doté en général de 8 Go de mémoire.
Mais le système d'exploitation ainsi que les logiciels tiers (comme par exemple Python) occupent de l'espace mémoire, rendant l'espace disponible moindre, comme le montre cette capture d'écran montrant la mémoire de mon ordinateur :



Il n'est donc pas anormal que le programme de monsieur Laguigne ne fonctionne pas.

Corrigé de l'exercice 9.12 page 171

Une fonction possible est la suivante :

Code Python 9-269

```python
from random import random

def moyenne(n):
    T = 0
    for k in range(n):
        s = 0
        i = 0
        while s < 1:
            s += random()
            i += 1

        T += i

    return T / n
```

Pour des valeurs de n de la forme $n = 10^k$ pour k entier de 2 à 7, on a alors :

```
>>> for n in [ 10**k for k in range(2,8) ]:
      print( moyenne(n) )

2.83
2.719
2.7393
2.71931
2.718344
2.7186881
```

Remarque 32

Il semblerait que cette moyenne converge vers un nombre bien particulier. D'après la loi des grands nombres, ce nombre correspond à E(X), c'est-à-dire le nombre moyen de fois que l'on doit répéter l'expérience $\mathscr{E}$ pour que la somme des nombres choisis au hasard soit strictement supérieure à 1.
Avec du temps, en prenant $n = 10^8$, puis $n = 10^9$, on trouve :

```
2.71829368
2.718286834
```

Or, $e \approx 2,71828182846$. Alors... Coïncidence ?

Pour le savoir, il faudrait pouvoir calculer mathématiquement cette moyenne... Mais en terminale, les outils dont nous disposons ne sont pas assez puissants pour le faire. En effet, obtenir la loi de probabilité de X n'est pas chose aisée... mais on peut le démontrer à l'aide d'intégrales multiples (ça, c'est pour les profs !).

Corrigé de l'exercice 9.13 page 171

1 Voici une proposition :

Code Python 9-270

```
from random import randint
def lancer(n):
    L = []
    c = 0
    for k in range(n):
        L.append( randint(1,6) )

    for i in range( len(L)-1 ):
        if L[i] == 6 and L[i+1] == 6:
            c += 1

    return c/n
```

Une autre possibilité (sans liste) est la suivante :

Code Python 9-271

```
def lancer(n):
    c = 0
    for k in range(n):
        d = randint(1,6)
        if k > 0:
            if d == 6 and precedent == 6:
                c += 1
        precedent = d

    return c/n
```

Si $k = 0$, c'est-à-dire si c'est le premier lancer de dé qui est simulé, on ne fait rien et on affecte à la variable `precedent` le nombre choisi ; il servira de précédent pour le tirage suivant.

2 Pour $n = 10^k$, k étant un entier de 2 à 7, on a :

```
>>> for n in [10**k for k in range(2,8)]:
      print( lancer(n) )

0.04
0.034
0.0281
0.02832
0.02791
0.0277122
```

3 La probabilité d'avoir un 6 suivi d'un 6 est :

$$\frac{1}{6} \times \frac{1}{6} \approx 0,02777\ldots$$

Le résultat trouvé à la question précédente est donc cohérent par rapport à cette probabilité.

Corrigé de l'exercice 9.14 page 171

1 On répète 10 fois de façon indépendante l'expérience $\mathscr{E}$ donc X suit la loi binomiale de paramètres $n = 10$ et $p = \dfrac{3}{10} = 0,3$ (probabilité d'obtenir un multiple de 3 entre 1 et 10, donc obtenir « 3 », « 6 » ou « 9 » sur 10 nombres).

2 D'après le cours, $\mathrm{E}(\mathrm{X}) = np = 3$ et $\sigma(\mathrm{X}) = \sqrt{np(1-p)} \approx 1,45$.

3 La fonction dûment complétée est la suivante :

Code Python 9-272

```
from random import randint

def simul(n):
    total = 0
    for i in range(n):
        c = 0
        for k in range(10):
            a = randint(1,10)
            if a % 3 == 0:
                c = c + 1

        total = total + c

    return total / n
```

4 Notons X_k la variable aléatoire représentant le nombre de multiples de 3 obtenus lors de la simulation k, pour $1 \leqslant k \leqslant n$.
D'après l'inégalité de concentration,

$$\mathrm{P}\left(\left|\frac{\mathrm{X}_1 + \cdots + \mathrm{X}_n}{n} - \mu\right| \geqslant \delta\right) \leqslant \frac{\sigma^2}{n\delta^2}$$

En prenant $\delta = 10^{-3}$, $\mu = 3$ et $\sigma = 1,45$, cela donne :

$$\mathrm{P}\left(\left|\frac{\mathrm{X}_1 + \cdots + \mathrm{X}_n}{n} - 3\right| \geqslant 10^{-3}\right) \leqslant \frac{1,45^2}{10^{-6}n}$$

ou encore :

$$\mathrm{P}\left(\left|\frac{\mathrm{X}_1 + \cdots + \mathrm{X}_n}{n} - 3\right| \geqslant 10^{-3}\right) \leqslant \frac{1,45^2 \times 10^6}{n}.$$

Ainsi, choisir n tel que $\dfrac{1,45^2 \times 10^6}{n} \leqslant 10^{-2}$ (par exemple... le seuil 10^{-2} est à notre convenance) suffirait, soit $n \geqslant 1,45^2 \times 10^8$.

Corrigé de l'exercice 9.15 page 172

1 X suit la loi binomiale de paramètres n (on répète la même expérience n fois de manière indépendante) et $p = \dfrac{3}{6} = \dfrac{1}{2}$ (la probabilité d'obtenir un multiple de 2 à chaque lancer).

2 D'après le cours,

$$\mathrm{E}(\mathrm{X}) = np = \frac{1}{2}n$$

et

$$\sigma(\mathrm{X}) = \sqrt{np(1-p)} = \frac{1}{2}\sqrt{n}.$$

3 La fonction dûment complétée est la suivante :

Code Python 9-273

```
1  from random import randint
2
3  def simul(n):
4      total = 0
5      for k in range(n):
6          D = randint(1,6)
7          if D % 2 == 0:
8              total = total + 1
9
10     return total
```

4 Pour la fonction `diff(n)` on fait appel à la fonction `simul(n)` :

```
def diff(n):
    return abs( simul(n) - n/2 )
```

5 On peut considérer la fonction suivante :

Code Python 9-274

```
1  def echant(N,n):
2      somme = 0
3      for k in range(N):
4          somme = somme + diff(n)
5
6      return somme / N
```

On obtient par exemple :

```
>>> echant(10000,1000)
12.5797
```

6 Il s'agit ici de faire appel N fois à la fonction `simul(1000)`, qui retourne une valeur m, et à chaque fois, de vérifier si $|\,E(X) - m\,| \leqslant \dfrac{2\sigma(X)}{\sqrt{1\,000}}$, auquel cas on devra incrémenter un compteur (que l'on va nommer `somme`).

À l'issue de ces N simulations, on calculera $\dfrac{\texttt{compteur}}{N} \times 100$ qui sera le pourcentage de cas où l'on a obtenu une différence plus petite que la borne supérieure $\dfrac{2\sigma(X)}{\sqrt{1\,000}}$.

On obtient alors la fonction suivante :

Code Python 9-275

```
def proportion(N):
    n = 1000
    mu = 500
    sigma = 250
    borne_sup = 2*sigma/(n**0.5)

    compteur = 0
    for k in range(N):
        if abs( mu - simul(n) ) <= borne_sup:
            compteur = compteur + 1

    return compteur * 100 / N
```

Cela donne par exemple :

```
>>> proportion(1000)
65.8
>>> proportion(1000)
67.5
>>> proportion(1000)
68.5
>>> proportion(1000)
67.9
```

Index

Algorithme de Briggs, 153
Algorithme de Brouncker, 155
Angles
    Conversion en degrés, 46
    Conversion en radians, 46

Bézout, 162
Boucles
    conditionnelles (while), 23
    itératives (for), 22
Briggs (algorithme de), 153
Brouncker (algorithme de), 155

cmath (module), 47
Combinaisons, 45
Compréhension (liste), 42
Constantes
    e, 46
    pi, 46
Cosinus, 46
Courbes représentatives, 56

Décomposition (produit de facteurs premiers), 163
Dénombrement
    Arrangement, 153
    Combinaison, 152
    Permutation, 152
    Produit cartésien, 152
Diagrammes en barre, 61
Droite (représentation en Python), 55

e (constante), 46
Échantillonnage, 63
Encadrement, 21
Eratosthène (crible), 163
Euclide (algorithme), 161
Euler (méthode), 114
Exponentielle, 45

Factorielle, 45, 149

lambda (fonction Python), 56

Liste
    append, 41
    extend, 41
    len (longueur), 40
    Somme exacte de tous les nombres, 45
Liste par compréhension, 42
Listes
    index, 41
Logarithme népérien, 45
Loi binomiale, 50, 150
Loi géométrique, 51, 158
Loi uniforme, 49

math (module), 45
matplotlib (diagrammes en barres), 61
matplotlib (tracé de courbes), 56
Matrices, 43
Méthode d'Euler, 114
Méthode de Newton, 114
Modules, 14
Mot-clé
    and, 20
    case, 19
    def, 13
    elif, 18
    else, 18
    for, 22
    if, 18
    in, 22, 23
    len, 40
    list, 39
    match, 19
    or, 21
    while, 23

Newton (méthode), 114
Nombres complexes, 47
numpy.random (module), 49

Opérateurs
    Égalité, 20
    Inférieur, 20

Supérieur, 20

Permutations, 45
PGCD, 45
PGCD (calcul), 161
Pi (constante), 46
Point (représentation en Python), 54
Probabilités (module random), 62

random (module), 48, 62
range, 23
Régression linéaire, 160

Sinus, 46
Sinus et cosinus (courbes), 117
Statistiques, 60
Suites numériques
Algorithmes de seuil, 112
Premiers termes, 112

Tangente, 46

Vecteur (représentation en Python), 54, 118