éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE PROFESSIONNELLE

CAP 2DE 1RE TLE

Mathématiques

ALGORITHMIQUE ET PROGRAMMATION

Mots-clés
Transversal – algorithmique – programmation – python – boucle – variable – instruction conditionnelle – booléen – entier – flottant – liste – affectation – algorigramme

SOMMAIRE

Intentions majeures

Pourquoi travailler l'algorithmique et la programmation ?

En quelques années, le numérique et plus particulièrement la programmation ont vu leur importance croître dans de nombreux métiers. Il est donc essentiel, de l'école élémentaire au post-baccalauréat, d'accompagner ces changements.

Dans la continuité du collège, les programmes de mathématiques de la classe de seconde professionnelle et de CAP visent à consolider les notions d'algorithmique travaillées au cycle 4 à l'aide d'un logiciel de programmation visuel en mathématiques et en technologie. Par ailleurs, ce travail sur la programmation facilitera la poursuite d'études après le baccalauréat, en section de techniciens supérieurs notamment.

L'algorithmique trouve naturellement sa place dans tous les domaines des programmes de mathématiques de la voie professionnelle. Les problèmes traités en algorithmique et programmation peuvent également s'appuyer sur les autres disciplines (la physique-chimie, les enseignements professionnels, etc.) ou la vie courante.

L'écriture d'algorithmes et de programmes est également l'occasion de transmettre aux élèves l'exigence d'exactitude et de rigueur et de les entraîner à la vérification et au contrôle des démarches qu'ils mettent en œuvre. Les capacités travaillées, comme reconnaître un schéma ou décomposer un problème en sous-problèmes, sont transférables en mathématiques et l'algorithmique et la programmation fournissent un nouveau cadre permettant de les développer. En programmant, les élèves revoient, par exemple, les notions de variable et de fonction mathématiques sous une forme différente. En outre, ce module permet de développer des compétences transversales comme le raisonnement, la logique, l'argumentation et la démonstration.

De plus, le fait de procéder par essais/erreurs et d'avoir un retour immédiat sur les instructions écrites dans un programme contribue grandement à donner du sens à ce que fait l'élève et contribue à sa motivation.

Ce ne sont ni la maîtrise d'un langage de programmation ni une virtuosité technique qui sont visées, mais davantage la mise en exergue de la programmation en tant qu'outil au service de la formation des élèves à la pensée algorithmique.

Continuité et ruptures avec le cycle 4

Au cycle 4, les élèves ont appris à :

- écrire une séquence d'instructions ;
- utiliser simultanément des boucles « répéter ... fois » et « répéter jusqu'à ... » avec des instructions conditionnelles permettant de réaliser des figures, des calculs et des déplacements ;
- décomposer un problème en sous-problèmes.

En technologie, les élèves ont travaillé l'algorithmique et la programmation d'une part en étudiant des algorigrammes, et d'autre part en manipulant divers capteurs dans le cadre de projets de robotique.

En seconde, les élèves passent progressivement de l'utilisation du langage de programmation visuel qu'ils ont utilisé dans les classes antérieures au langage interprété Python. L'accent est mis sur la programmation modulaire qui consiste à découper une tâche complexe en tâches plus simples. Pour ce faire, les élèves utilisent des fonctions informatiques.

Contrairement au cycle 4, on utilise l'algorithmique et la programmation afin de répondre à des problématiques mathématiques en s'appuyant autant que possible sur des situations issues des domaines professionnels ou de la vie courante.

Liens avec d'autres disciplines

La démarche algorithmique peut être réinvestie en physique-chimie notamment dans le cadre de l'utilisation de cartes à microcontrôleurs associées à des capteurs. Ceci permet, en plus d'aborder les capacités et connaissances du module « Électricité » de manière transversale, d'éviter le côté trop protocolaire et « boîte noire » que l'utilisation de l'ExAO suscite.

En outre, la majorité des instructions d'un langage de programmation sont définies en anglais. Une démarche interdisciplinaire peut être envisagée à ce sujet.

Dans le cadre de la co-intervention, la pensée algorithmique peut intervenir dans de nombreuses situations professionnelles du domaine de la production ou des services. En plus de permettre une connexion directe entre les référentiels des activités professionnelles et les programmes de mathématiques, ces situations mettant en œuvre la pensée algorithmique apportent une réelle plus-value et une richesse aux contenus disciplinaires.

Modalités de mise en œuvre

Comme le précise le programme, il n'y a pas lieu de faire de cours spécifiques d'algorithmique et de programmation. Il s'agit de s'appuyer sur une situation à étudier, un problème à résoudre pour introduire les connaissances et développer les capacités du module.

Planification : quelle organisation annuelle ? Éléments de progressivité

Il n'y a pas lieu de morceler le travail sur les connaissances et capacités. Il est préférable de les associer dès le début de l'année plutôt que d'attendre que les élèves maîtrisent les types de variables, les instructions conditionnelles et les boucles, qui ont d'ailleurs déjà été rencontrées au collège, avant de leur demander de travailler avec les fonctions. C'est le choix d'activités pertinentes dans le contexte mathématique du cours qui permet d'introduire ou de revenir sur de nombreuses capacités et connaissances de ce module, comme l'illustre l'activité proposée du « jeu du 7 ».

Quelle institutionnalisation ? Quelle trace écrite ?

Les éléments travaillés dans une activité doivent être explicités en fin de tâche et les savoirs institutionnalisés dans un cahier ; l'élève pourra s'y référer tout au long de sa scolarité.

Différenciation

Pour faire face à l'hétérogénéité, il est important de prévoir une différenciation :

- étayage à l'aide de coups de pouce variés ;
- prolongements proposés aux élèves les plus rapides. Il peut également être intéressant de s'appuyer sur ces derniers afin de les sensibiliser à l'entraide et à la formation entre pairs.

La dynamique de projet, qu'elle soit disciplinaire ou pluridisciplinaire, peut permettre d'aborder des approches variées sur les problèmes posés. Le type de tâches proposées (avec plus ou moins d'initiative laissée aux élèves) ainsi que la diversité des supports utilisés créent de fait une différenciation.

Prolongement hors de la classe : quel travail en autonomie ?

Le langage Python est un langage multiplateformes. Dès lors, les élèves ont la possibilité de l'utiliser en ligne sur ordinateur ou sur leurs machines nomades (tablettes ou smartphones) afin de travailler en autonomie.

Au sein de l'établissement, l'existence ou la mise en place d'un club ou d'un atelier (de mathématiques, d'informatique ou de robotique) peut également participer à donner de l'appétence pour la programmation qui est dans ce contexte abordée par des biais moins scolaires et permettre ainsi l'approfondissement et/ou la remédiation des notions vues en classe ou à venir.

Éléments pour l'enseignant : débuter en algorithmique et en programmation

Cette partie a pour objectif de rappeler quelques éléments concernant l'algorithmique et la programmation. Les exemples développés figurent le plus souvent dans le programme de seconde professionnelle au sein des rubriques « exemples d'algorithmes et d'activités numériques ».

Un algorithme est une suite finie d'instructions élémentaires qui s'appliquent dans un ordre déterminé à un nombre fini de données dans le but de répondre à une problématique.

Après avoir présenté l'écriture de certains algorithmes en pseudo-code ou à l'aide d'algorigrammes, quelques éléments sont précisés à l'aide du langage Python, langage interprété retenu dans les programmes de lycée des voies générale, professionnelle et technologique.

Pseudo-code et algorigramme

Dans un premier temps, sans faire référence à un langage de programmation, mais tout en respectant certaines conventions, il peut être intéressant de demander aux élèves d'écrire un algorithme en pseudo-code. En effet, l'écriture en pseudo-code permet de développer une démarche structurée en libérant l'esprit de la syntaxe et des spécificités d'un langage de programmation donné.

À titre d'exemple, si l'on souhaite étudier un algorithme qui permet d'afficher les nombres entiers compris entre 1 et 5, voici ce que l'on pourrait écrire en pseudo-code :

i* ← *1	*Dans un premier temps, on initialise la variable i à la valeur 1.*
Tant que i* ≤ *5	*On utilise une boucle dite non bornée.*
Afficher i	*On affiche la valeur de la variable i.*
i* ← *i* + *1	*Puis on incrémente sa valeur de 1.*
Fin Tant que	*Enfin, on indique la fin de la boucle.*

La flèche ← est utilisée pour affecter une valeur à une variable. Elle permet également de mettre en avant le sens de la lecture qui se fait de droite à gauche.

Au cycle 4, en technologie, les élèves ont pu rencontrer des algorigrammes, qu'on appelle aussi organigrammes de programmation. Ce mode de représentation des algorithmes permet d'en visualiser les différentes étapes. S'il n'est pas nécessaire de systématiser leur utilisation, celle-ci peut aider certains élèves à comprendre plus facilement certains algorithmes simples. On ne s'arrête pas à une description d'une quelconque norme de représentation.

Les algorigrammes présents dans ce document comprennent divers éléments graphiques dont la légende est indiquée ci-dessous :

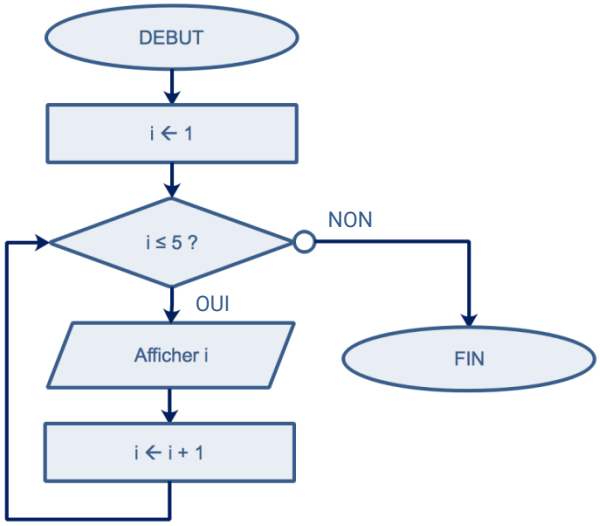
Annoncer le début, une interruption temporaire ou la fin d'une procédure.

Indiquer une phase de traitement.

Utiliser des blocs d'instruction conditionnelle et des boucles.

Traiter les instructions de type entrée / sortie.

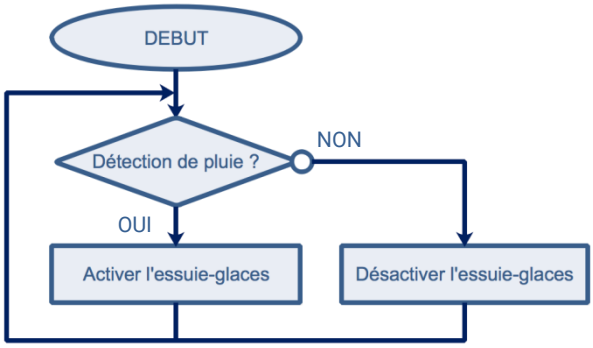
L'algorigramme ci-dessous est une transcription de l'algorithme proposé en pseudo-code précédemment.



Dans ce document, le cercle au niveau de la boucle indique la sortie de celle-ci lorsque la condition n'est pas vérifiée.

Exemple dans d'autres disciplines

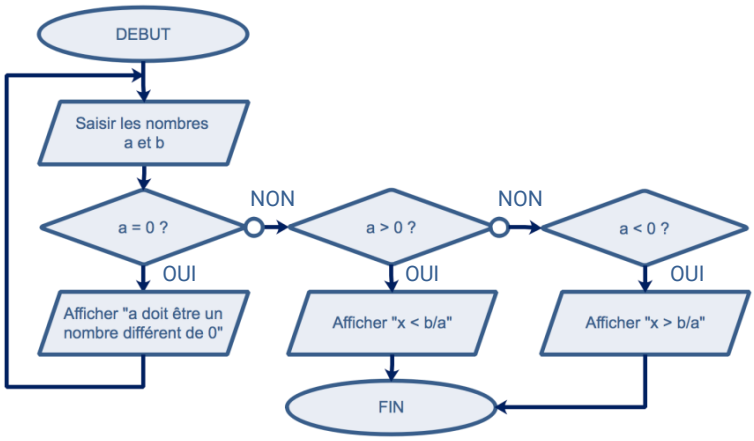
Il est possible de modéliser de façon simplifiée le fonctionnement des essuie-glaces à l'aide d'un algorigramme ci-dessous.



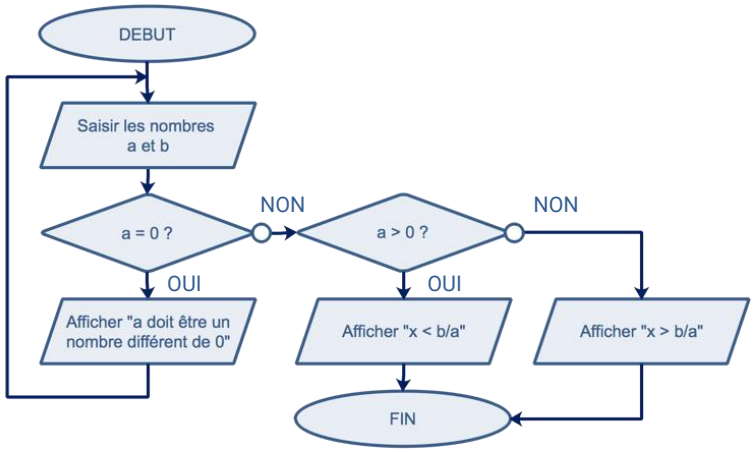
Exemple de la résolution d'une inéquation du premier degré

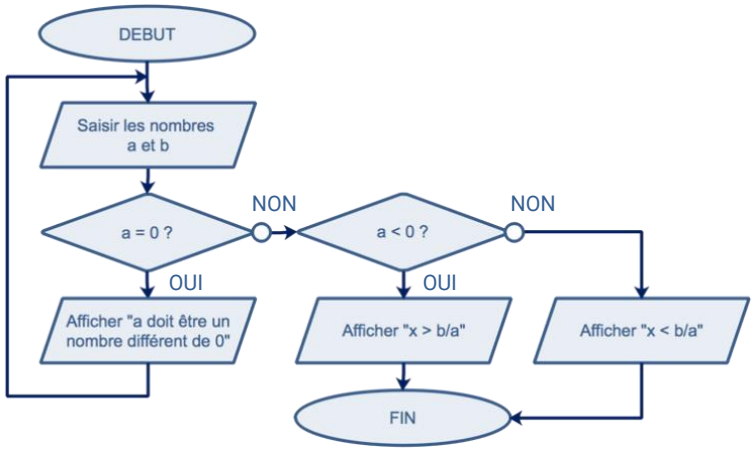
Dans le programme de seconde professionnelle, l'exemple de la résolution de l'inéquation du premier degré à une inconnue du type $ax < b$ avec $a \neq 0$ est mentionné.

Ce premier algorigramme permet de répondre à cette question :



On peut amener les élèves à réfléchir sur une simplification de l'algorigramme et montrer, en travaillant la logique, que le dernier test est superflu. Deux algorigrammes peuvent alors être construits avec eux :

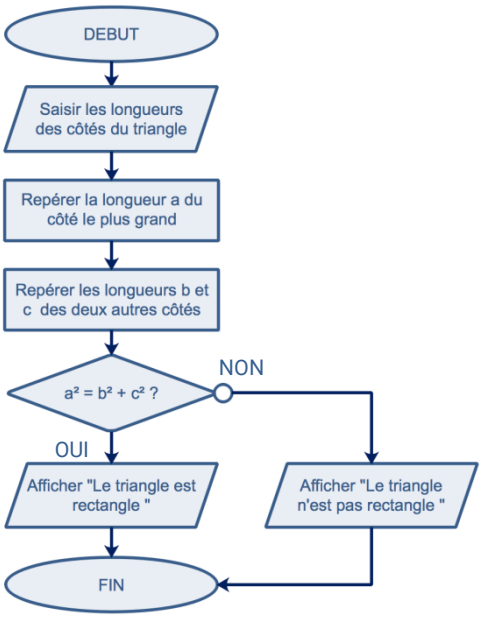




Exemple de la réciproque du théorème de Pythagore

Dans le module de géométrie du programme de CAP et de celui de seconde professionnelle figurent le théorème de Pythagore et sa réciproque.

L'algorigramme ci-dessous peut schématiser le raisonnement à utiliser pour démontrer qu'un triangle est ou non rectangle.



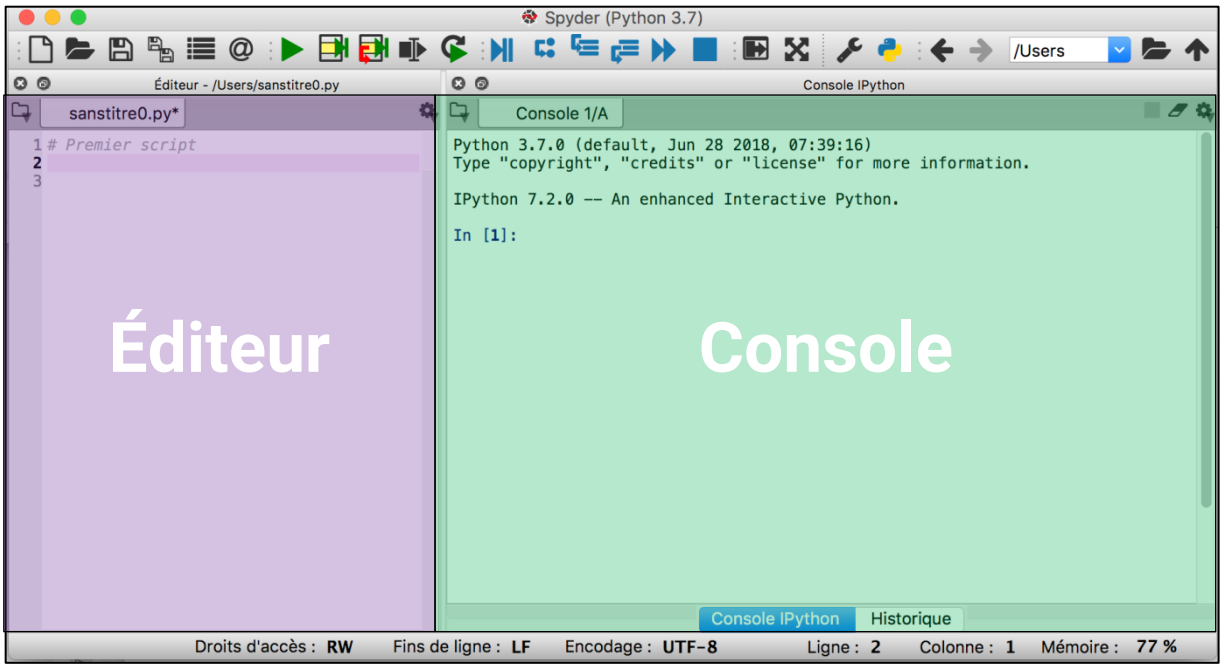
Premiers pas en Python

Le langage Python a été choisi pour sa concision, sa simplicité, son implémentation dans de multiples environnements et son utilisation dans l'enseignement supérieur. C'est un langage interprété, ce qui signifie que les lignes de code sont exécutées successivement sans tenir compte, *a priori*, des suivantes. L'avantage réside dans le fait qu'un script (on peut dire aussi un « programme ») peut être exécuté de la même manière indépendamment du système utilisé, dès lors qu'il dispose d'une application (appelée interpréteur) comprenant ce langage. De nombreux environnements de développement sont disponibles gratuitement (PyCharm, EduPython, Spyder, etc.). En annexe sont proposés des liens permettant d'installer une distribution Python sur un ordinateur ou de s'en servir en ligne sans installation préalable.

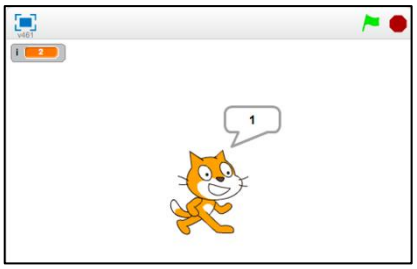
Présentation d'un environnement de développement

Quel que soit l'environnement de développement choisi, il est toujours constitué de deux parties principales :

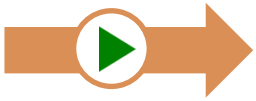
- l'éditeur (ici, la partie gauche) : il permet d'écrire le programme (également appelé script), de le modifier et de l'enregistrer ;
- la console (ici, la partie droite) : elle permet d'exécuter et par conséquent de tester (à n'importe quel moment) un script qui aura été saisi en amont dans l'éditeur.



Reprenons l'exemple de l'affichage des nombres entiers compris entre 1 et 5. Les impressions d'écran ci-dessous présentent la transcription des algorithmes présentés précédemment ainsi que le résultat de leur exécution sous Scratch et Python :




```
1 # Premier script
2
3 i = 1
4 while i <= 5:
5     print(i)
6     i = i + 1
```



```
In [1]: runfile('/Users/Documents/OK.py'
1
2
3
4
5
```

Remarque : c'est l'appui d'un bouton spécifique qui déclenche l'exécution ; l'utilisateur n'a pas à taper lui-même le code **runfile('....')**.

Les bibliothèques

Le langage Python permet l'utilisation de multiples instructions qui ne sont pas toutes disponibles par défaut dans l'éditeur. C'est pourquoi il est nécessaire, en fonction des besoins du script que l'on souhaite écrire, d'importer une instruction contenue dans une bibliothèque ou une bibliothèque entière (également appelée module), qui regroupe plusieurs instructions.

Voici quelques-unes de ces bibliothèques :

- **math** permet d'avoir accès à certains nombres (« pi », « e », etc.), mais également à des fonctions mathématiques telles que cos, sin, sqrt (la racine carrée), log pour le logarithme népérien, log10 pour le logarithme décimal, etc. ;
- **random** regroupe des instructions de gestion des nombres aléatoires ;
- **scipy** contient des fonctionnalités relatives à l'algèbre linéaire et aux statistiques ;
- **matplotlib** propose de nombreux types de représentations graphiques ;
- **turtle** permet la création de figures et de dessins ;
- **numpy** permet de manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux.

Les instructions suivantes permettent d'importer tout ou partie d'une bibliothèque.

```
import math                          # import de la bibliothèque math
from math import sqrt, cos           # import des instructions racine carrée et cosinus contenues dans la bibliothèque math

import matplotlib.pyplot as plt      # import de la bibliothèque matlplot que l'on définit comme un objet plt

import scipy as sc                   # import de la bibliothèque scipy que l'on définit comme un objet sc

import turtle as t                   # import de la bibliothèque turtle

import random                        # import de la bibliothèque random
from random import randint           # import de l'instruction randint contenue dans la bibliothèque random

import numpy as np                   # import de la bibliothèque numpy que l'on définit comme un objet np
```

Il est possible d'obtenir de l'aide sur une instruction ou une bibliothèque en saisissant **help()** dans la console puis d'indiquer l'instruction à développer. Dans l'exemple suivant, on souhaite obtenir de l'aide sur l'instruction **sqrt** contenue dans la bibliothèque math.

```
help> math.sqrt
Help on built-in function sqrt in math:

math.sqrt = sqrt(x, /)
    Return the square root of x.
```

En saisissant uniquement **math** en aval de **help()**, on obtient de l'aide sur l'ensemble des instructions contenues dans cette bibliothèque. Pour quitter la rubrique d'aide, il suffit de saisir **quit** dans la console.

Les fonctions

Les fonctions python ne sont pas exclusivement des fonctions avec des variables numériques. En programmation, une fonction permet de traiter des paramètres de type numérique (entier ou flottant), mais également des chaînes de caractères (*string*), des listes, une autre fonction. Par conséquent, une fonction pourra tout aussi bien renvoyer du texte, une liste et même une fonction ou bien, renvoyer des types d'objets différents de ceux présents en arguments, renvoyer des valeurs aléatoires (comme la fonction **de6()** ci-dessous qui renvoie un nombre entier compris entre 1 et 6), voire ne rien renvoyer du tout !

Pour définir une fonction en langage Python, on utilise la commande **def** en prenant soin de respecter sa syntaxe (nom, parenthèses, « : » et indentation à la ligne suivante) comme le présentent les exemples ci-dessous.

```
1 # Premier script
2
3 def premierscript():
4     i = 1
5     while i <= 5:
6         print(i)
7         i = i + 1
```

L'exemple ci-contre présente une autre version du premier script développé ci-dessus, cette fois en y créant une fonction **premierscript()**.

Il est possible de le tester en l'exécutant puis en tapant **premierscript()** dans la console.

Leur utilisation facilite la décomposition d'un problème complexe en sous-problèmes plus simples à résoudre.

Le tableau ci-dessous dresse une liste non exhaustive de quelques fonctions pouvant illustrer le programme de seconde professionnelle.

La commande r**eturn** permet d'indiquer le résultat que doit renvoyer la fonction. Elle interrompt l'exécution de la fonction et renvoie un résultat. Il faut noter qu'on dit bien **renvoyer** et non **retourner** un résultat.

Script	Descriptif

`from math import pi`
`def airedisque(R):`
`  return pi*R**2`
	Calcul de l'aire d'un disque de rayon R

`def volume(L,l,h):` `  return round(L*l*h)`	Calcul de la valeur arrondie à l'unité du volume d'un parallélépipède rectangle de dimensions L, l et h

`from random import randint`
`def de6():`
`  return randint(1,6)`
	Simulation d'un lancer de dé à 6 faces

`from random import randint`
`def deux_des12():`
`  return randint(1,12)+randint(1,12)`
	Simulation de la somme de deux dés à 12 faces

`def fct(x):` `  return 2*x+3`	Calcul de l'image d'un nombre réel par la fonction qui, à tout nombre réel x, associe le nombre $2x+3$
`def convert(T):` `  return T+273.15`	Conversion d'une température de degré Celsius en Kelvin

`from math import sqrt`
`def diam(h,V):`
`  return round(2*sqrt(V/(pi*h)),2)`
	Calcul de la valeur arrondie au centième du diamètre d'un cylindre de révolution de hauteur h et de volume V

`def remise(M,T):` `  return M*(1-T/100)`	Calcul du prix d'un montant M après une remise au taux T %

On remarque que, selon les cas, comme par exemple dans le cas de la fonction **airedisque()** ou **de6()**, il est nécessaire d'importer une ou plusieurs fonctions contenues dans des bibliothèques.

```
In [2]: convert(25)
Out[2]: 298.15

In [3]: fct(7.3)
Out[3]: 17.6

In [4]: volume(2.4,5.2,4.8)
Out[4]: 60

In [5]: diam(6,1050)
Out[5]: 14.93
```

Une fois la fonction définie dans le script, elle peut être appelée dans l'éditeur par une autre fonction, mais elle peut également l'être dans la console comme c'est le cas pour les quatre fonctions de l'exemple.

Une fonction peut avoir plus d'un argument, comme l'illustre la fonction **volume(L, l, h)**. Il est intéressant de dire à cette occasion aux élèves que si de telles fonctions existent en mathématiques, on ne rencontre au lycée que des fonctions d'une seule variable.

L'indentation

```
1 # Premier script
2
3 def premierscript():
4     i = 1
5     while i <= 5:
6         print(i)
7         i = i + 1
```

Tout comme dans l'algorithme sous forme de pseudo-code, on peut facilement remarquer différents niveaux de tabulation aux lignes 4 à 7 du script étudié. Cette tabulation est appelée **indentation**. En plus de hiérarchiser les instructions, le principe d'indentation est nécessaire à la saisie d'un script.

Une mauvaise utilisation de ce principe d'indentation peut engendrer des résultats erronés lors de l'exécution d'un script. Pour comprendre ce principe, la rédaction pas à pas du script étudié est détaillée.

1er cas de figure

```
1 # Premier script
2
3 def premierscript():
4 i = 1
5 while i <= 5:
6 print(i)
7 i = i + 1
```

Lors de l'exécution du script, un message d'erreur apparaît dans la console indiquant une mauvaise indentation à la ligne 4.

```
  File "/Users/ok.py", line 4
    i = 1
    ^
IndentationError: expected an indented block
```

Le niveau d'indentation des instructions de niveau 1 doit être modifié ligne par ligne.

2e cas de figure

```
1 # Premier script
2
3 def premierscript():
4     i = 1
5     while i <= 5:
6     print(i)
7     i = i + 1
```

Lors de l'exécution du script, un message d'erreur identique apparaît dans la console mais cette fois à la ligne 6.

```
  File "/Users/ok.py", line 6
    print(i)
        ^
IndentationError: expected an indented block
```

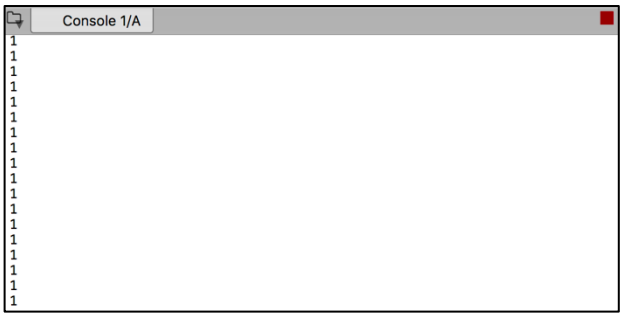
Le niveau d'indentation de l'instruction **print()** qui permet d'afficher la valeur de la variable **i** doit être modifié.

3e cas de figure

```
1 # Premier script
2
3 def premierscript():
4     i = 1
5     while i <= 5:
6         print(i)
7     i = i + 1
```

Après avoir exécuté le script, plus aucun message d'erreur n'est affiché dans la console.

Néanmoins, lorsqu'on appelle la fonction **premierscript()** dans la console, on voit apparaître une infinité d'occurrences du nombre 1, ce qui ne correspond pas au résultat escompté.



Ceci est dû au fait que l'indentation permet également de contrôler les sorties de boucles ou de blocs d'instructions conditionnelles.

Dans le 3e cas de figure, la boucle **while i <= 5** (tant que la valeur de la variable **i** est inférieure ou égale à 5) est une boucle infinie puisque la condition d'arrêt n'est jamais atteinte dans ce cas. En effet, l'instruction visant à incrémenter la valeur de la variable **i** (c'est-à-dire l'instruction d'affectation **i = i + 1**) a été saisie au même niveau que l'instruction de début de boucle. Elle ne pourra être exécutée que dans le cas où l'on sort de cette boucle, ce qui n'arrive pas dans ce 3e cas.

L'incrémentation de la variable **i** doit se faire à chaque passage dans la boucle, c'est pourquoi le niveau d'indentation doit être prévu en conséquence.

4e cas de figure

```
1 # Premier script
2
3 def premierscript():
4     i = 1
5     while i <= 5:
6         print(i)
7         i = i + 1
```

```
In [6]: premierscript()
1
2
3
4
5
```

Lorsqu'on appelle la fonction **premierscript()** dans la console (après avoir exécuté le script), on voit que le résultat obtenu est bien conforme à ce que l'on attendait.

Variables informatiques

En informatique, on peut voir une variable comme une étiquette fichée dans une boîte qui peut contenir différentes valeurs. Le contenu de chaque boîte varie au cours de l'exécution d'un programme, ce qui n'est pas le cas d'une variable mathématique.

Pour dire **affecter à** la variable **a le nombre 5**, on écrira, en pseudo-code, « **a** ← **5** » et, avec le langage Python, « **a** = 5 », qu'on ne lira pas « **a** est égal à 5 » mais « **a reçoit 5** » pour éviter toute confusion chez les élèves.

Quand l'ordinateur évalue une expression dans laquelle figure une ou plusieurs variables, comme dans l'instruction **c** = **a + b**, il commence par aller chercher les valeurs rangées dans les deux boîtes étiquetées par **a** et **b**, il calcule leur somme et range cette valeur dans la boîte étiquetée par **c**.

Dans le langage Python, une instruction telle que **a = a + 1** est possible. Ceci signifie qu'on ajoute 1 au contenu de la boîte étiquetée avec **a** avant de la ranger de nouveau dans cette boîte. L'affectation n'est pas une égalité.

Par exemple :

```
In [1]: a=5

In [2]: b=3

In [3]: c=a+b

In [4]: c
Out[4]: 8

In [5]: d=a**2

In [6]: d
Out[6]: 25

In [7]: a==b
Out[7]: False

In [8]: a==5
Out[8]: True
```

*On affecte la valeur 5 à la variable **a**.*

*On affecte la valeur 3 à la variable **b**.*

*On récupère les valeurs des variables **a** et **b**, on calcule leur somme et on l'affecte à la variable **c**.*

*On affiche la valeur de la variable **c**.*

*On récupère la valeur de la variable **a**, on l'élève au carré et on affecte le résultat à la variable **d**.*

*On affiche la valeur de la variable **d**.*

*On teste si la valeur des variables **a** et **b** sont égales.*

*On teste si la valeur de la variable **a** est 5.*

Dans les deux derniers cas, le résultat obtenu est appelé **booléen** (vrai ou faux). Ce type de variable est utilisé en seconde professionnelle tout comme les **entiers** (int), les **flottants** (float) et les **chaînes de caractères** (str). Les **listes** seront quant à elles à développer en classes de première et terminale.

Enfin, il est important de respecter la casse dans la définition ou lors de l'appel d'une variable. Autrement dit, il convient de prêter une attention particulière à la distinction entre les lettres majuscules et les lettres minuscules lors de la saisie.

Point sur les entiers et les flottants

Le langage Python permet de travailler avec des nombres entiers naturels de taille quelconque.

En revanche, on peut observer les limites d'utilisation des nombres non entiers par les ordinateurs. En effet, ils ne travaillent pas avec des nombres réels, mais avec des flottants aussi appelés nombres à virgule flottante, qui constitue un sous-ensemble des nombres décimaux dont la précision est limitée par des contraintes liées au codage en mémoire.

Pour illustrer ceci, on saisit la somme suivante dans la console.

```
In [9]: 0.1+0.1+0.1
Out[9]: 0.30000000000000004
```

Le résultat obtenu est une conséquence directe de l'utilisation du codage binaire par l'ordinateur. Les nombres non entiers sont alors pris en compte avec une précision donnée.

Dans l'exemple ci-dessus, le nombre 0,1 n'a pas une écriture finie en base 2 : son écriture dans cette base est l'écriture périodique suivante $0{,}000\mathbf{1100}1100\mathbf{1100}1100\mathbf{1100}11\ldots_2$ qui est infinie (ce n'est pas le cas de tous les nombres décimaux, car, par exemple, 0,125 s'écrit $0{,}001_2$ en base 2).

L'ordinateur doit alors, pour le nombre 0,1, utiliser une valeur approchée et les erreurs commises en ne considérant que cette approximation obtenue de 0,1 s'ajoutent lors de la somme, ce qui explique que, pour l'ordinateur, 0,1 + 0,1 + 0,1 n'est pas égal à 0,3. En revanche, si l'on saisit 0,125 + 0,125 + 0,125 dans la console, on trouve bien que la somme est égale à 0,375.

La précision des nombres à virgule flottante est comprise entre 10^{-16} et 10^{-15}. Saisir les deux calculs ci-dessous dans la console nous permet d'apprécier cette précision.

```
In [10]: 3+10**(-16)
Out[10]: 3.0

In [11]: 3+10**(-15)
Out[11]: 3.000000000000001
```

De manière générale, on retiendra qu'**il faut éviter de tester l'égalité entre deux flottants**. En revanche, bien sûr, il n'y a aucun problème à comparer deux nombres entiers. C'est la raison pour laquelle, dans le programme de la classe de seconde, on s'intéresse aux triplets pythagoriciens, c'est-à-dire aux triplets de nombres **entiers** strictement positifs $(x\,;\,y\,;z)$ tels que $x^2+y^2=z^2$ plutôt que de tester si un triangle est rectangle ou non avec des côtés de longueurs non nécessairement entières.

L'exemple ci-contre nous indique, par exemple, que le triangle de dimensions 0,5 cm, 1,2 cm et 1,3 cm n'est pas rectangle ce qui erroné, puisqu'en réalité on a bien l'égalité $0{,}5^2+1{,}2^2=1{,}3^2$.

```
In [12]: 0.5**2+1.2**2==1.3**2
Out[12]: False
```

Affichage d'un résultat

L'instruction **print()** permet d'afficher le contenu des parenthèses dans la console. Bien qu'il faille préférer la programmation fonctionnelle à une programmation de type entrée / sortie (**input / print**), cette instruction peut s'avérer être intéressante si l'on souhaite débugger un script, c'est-à-dire en trouver les éventuelles erreurs pour les corriger, en demandant l'affichage de résultats intermédiaires.

Les instructions conditionnelles et les booléens

Pour illustrer la notion d'instruction conditionnelle, considérons l'offre suivante proposée par un site internet d'impression de cartes de visite :

« Le prix d'une carte de visite est de 8 centimes d'euros TTC. À partir de 1 000 cartes de visite commandées, un rabais de 5 % est accordé sur l'ensemble de la commande. »

Ce que l'on peut traduire en pseudo-code puis par une fonction python qui calcule le prix de la commande en fonction du nombre de cartes de visite commandées :

Si nombre < 1000	*On teste si le nombre de cartes est strictement inférieur à 1000.*
Alors prix ← nombre * 0,08	*Si c'est le cas, alors on calcule le prix à payer (0,08 € par unité).*
Sinon	*Sinon*
prix ← 0,95 * nombre * 0,08	*On applique une réduction de 5 % sur le prix total.*
Fin Si	*Puis on indique la fin de l'instruction conditionnelle.*

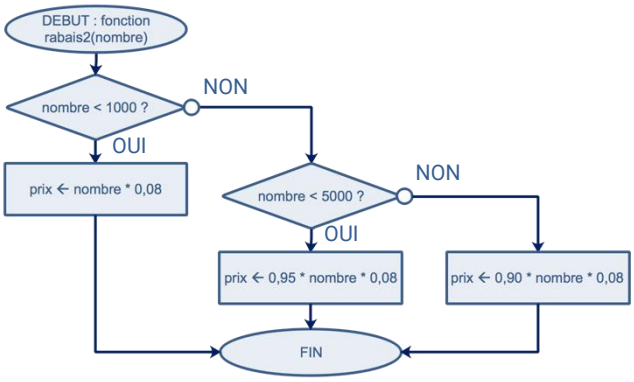
Le script ci-dessous transcrit l'algorithme ci-dessus.

```
def prix_apres_rabais(nombre):
    if nombre < 1000:
        prix = nombre * 0.8
    else:
        prix = 0.95 * nombre * 0.8
    return prix
```

Lorsqu'on réalise un test, on fait intervenir un **booléen**, qui peut prendre deux valeurs : True et False. L'instruction **return** permet quant à elle de renvoyer le résultat obtenu après l'appel d'une fonction.

Dans le cadre d'une différenciation, il est tout à fait envisageable de complexifier le problème en modifiant la consigne, comme dans l'exemple suivant.

« Le prix d'une carte est de 8 centimes d'euros TTC. Entre 1 000 et 4 999 cartes de visite commandées, un rabais de 5 % est accordé sur l'ensemble de la commande. À partir de 5 000 cartes de visite commandées, un rabais de 10 % est accordé sur l'ensemble de la commande. ».



Dans cet exemple, plusieurs tests consécutifs sont à mettre en œuvre : on teste d'abord si la valeur de la variable **nombre** est strictement inférieure à 1 000. Si ce n'est pas le cas, alors on teste si cette valeur est strictement inférieure à 5 000. Il est possible de le faire en imbriquant les **if… else** ou en utilisant l'instruction **elif** qui permet de simplifier l'écriture.

```
def prix_apres_rabais2(nombre):
    if nombre < 1000:
        prix = nombre * 0.8
    else:
        if nombre < 5000:
            prix = 0.95 * nombre * 0.8
        else:
            prix = 0.90 * nombre * 0.8
    return prix
```

```
def prix_apres_rabais3(nombre):
    if nombre < 1000:
        prix = nombre * 0.8
    elif nombre < 5000:
        prix = 0.95 * nombre * 0.8
    else:
        prix = 0.90 * nombre * 0.8
    return prix
```

Les conditions peuvent utiliser les opérateurs de comparaison : **<**, **>**, **<=**, **>=**, **==** (pour tester l'égalité), **!=** (pour tester la non-égalité) ainsi que les opérateurs **and**, **or** et **not**.

Dans le module « Géométrie », il est proposé de travailler autour de triplets pythagoriciens. Les booléens peuvent être utilisés dans ce cadre afin de tester si un triplet de nombres entiers est un triplet pythagoricien :

```
def pythagoricien(a,b,c):
    return a**2+b**2==c**2
```

Ce script renvoie directement un booléen dont la valeur est True lorsque la condition est vérifiée et False dans le cas contraire.

Cette fonction peut alors être appelée dans une autre fonction contenue dans le même script afin de déterminer, par exemple, la liste des triplets pythagoriciens constitués de nombres entiers strictement positifs inférieurs ou égaux à 1 000, les trois éléments du triplet sont ici rangés dans l'ordre croissant.

```
def liste_triplets_pythagore(seuil):
    for a in range(1,seuil+1):
        for b in range(a+1,seuil+1):
            for c in range(max(a,b), seuil+1):
                if pythagoricien(a,b,c):
                    print(a,b,c)
```

Après avoir exécuté le script, on saisit dans la console le nom de la fonction étudiée dans notre cas : liste_triplets_pythagoricien(1000).

Il faut attendre plusieurs minutes pour obtenir la réponse avec ce seuil de 1000.

Les boucles

Il en existe deux types :

- les boucles bornées pour lesquelles des instructions seront exécutées un nombre indiqué de fois ;
- les boucles non bornées qui nécessitent de spécifier une condition d'arrêt.

Afin de bien comprendre le fonctionnement de ces deux types de boucles, reprenons l'exemple de l'affichage des nombres entiers compris entre 1 et 5.

```
# Premier script

def premierscript():
    i = 1
    while i <= 5:
        print(i)
        i = i + 1
```

Comme nous l'avons souligné précédemment, la fonction **premierscript()** contient une boucle non bornée caractérisée par l'instruction **while** (**tant que** la condition est vérifiée). Il est donc primordial de faire en sorte que la condition puisse être vérifiée afin d'éviter que la boucle soit infinie.

```
def premierscript_bis():
    for i in range(5):
        print(i)
```

L'instruction for est utilisée dans le cadre d'une boucle bornée. Le script présenté ici permet de répéter 5 fois d'affilée l'affichage d'une variable que l'on incrémente d'une unité.

```
In [17]: premierscript_bis()
0
1
2
3
4
```

Le résultat obtenu après l'appel de cette fonction dans la console nous montre que l'instruction **for i in range(5)** permet d'afficher la valeur de la variable **i** pour **i** allant de 0 à 4.

```
def premierscript_bis():
    for i in range(1,6):
        print(i)
```

Pour obtenir la série de valeurs souhaitées, il est possible d'utiliser une variante de cette instruction, en précisant les bornes de l'intervalle sur lequel on veut travailler. Ici, le script permet d'afficher l'ensemble des nombres entiers de l'intervalle [1 ; 6 [.

```
def premierscript_ter(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s
```

Enfin, tout comme c'est le cas dans la fonction **premierscript()**, on peut également utiliser un compteur dans une boucle bornée. La fonction **premierscript_ter()** illustre ce cas en renvoyant la somme des n premiers nombres entiers naturels strictement positifs.

Les listes

Contrairement aux quatre types de variables évoqués précédemment (entier, flottant, chaîne de caractères et booléen), les **listes** ne figurent pas au programme de seconde professionnelle. Néanmoins, elles peuvent apporter une certaine plus-value quant à la simplification d'un script. Elles ne seront abordées qu'en classe de première.

Les listes sont des structures regroupant de manière ordonnée des variables ou des données alphanumériques en vue d'un affichage ou d'un traitement ultérieur. Elles bénéficient d'une syntaxe qui leur est propre : les crochets. Pour créer simplement une liste, il suffit donc de saisir entre crochets un certain nombre de valeurs séparées par une virgule. Le principal avantage qu'elles suscitent réside dans la possibilité d'accéder à une ou plusieurs valeurs la constituant, et ceci à tout moment.

Pour illustrer cela, nous allons considérer la liste **L** ci-après qu'on crée **en extension**, ce qui signifie qu'on en précise tous les éléments.

```
L = [7, 8, 5.2, True, 'programmation']
```

Le dernier élément de cette liste est une chaîne de caractères, qu'on signale à l'aide de guillemets, indifféremment simples ou doubles.

Comme c'est le cas pour bon nombre de langages de programmation, Python permet un adressage des listes c'est-à-dire que chaque élément d'une liste possède une adresse précise (appelée indice, *index* en anglais) qui correspond à la position de cet élément dans la liste.

```
In [8]: L
Out[8]: [7, 8, 5.2, True, 'programmation']

In [9]: L[0]
Out[9]: 7

In [10]: L[3]
Out[10]: True

In [11]: L[4]
Out[11]: 'programmation'
```

En saisissant le nom de la liste **L** dans la console, celle-ci est affichée.

On remarque que le premier élément de la liste est noté **L[0].**

La liste **L** contient ici 5 éléments notés respectivement **L[0]**, **L[1]**, **L[2]**, **L[3]**, **L[4]**.

De nombreuses instructions spécifiques à l'édition des listes existent. Le tableau suivant présente trois de ces instructions.

Ajouter la valeur 14 en dernier : **L.append(14)**	[7, 8, 5.2, True, 'programmation', 14]
Compter le nombre d'éléments de **L** : **len(L)**	`len(L)` `6`
Compte le nombre d'occurrences de 14 : **L.count(14)**	`L.count(14)` `1`

```
liste = [12, 43, 3.5, 18, 21]

liste.sort()

liste
[3.5, 12, 18, 21, 43]
```

Dans le cas d'une liste exclusivement numérique, **sort()** permet de trier par ordre croissant les valeurs contenues dans celle-ci. Ceci pourra être utilisé en première pour la détermination des médianes ou des quartiles.

Une liste peut contenir un nombre important de valeurs. La manipulation de ces valeurs peut s'avérer fastidieuse. Pour pallier cela, il existe d'autres moyens de générer une liste.

```
def sa(U1,r,n):
    L=[]
    for i in range(1,n+1):
        L.append(U1+(i-1)*r)
    return L

def sa2(U1,r,n):
    L=[U1+(i-1)*r for i in range(1,n+1)]
    return L
```

À titre d'exemple, le script présenté ici a pour objectif de générer une liste contenant les **n** termes d'une suite arithmétique de raison **r** et de premier terme **U1**.

Les fonctions **sa()** et **sa2()** proposent deux possibilités pour générer cette liste.

Dans le programme de première professionnelle figure la génération des listes **en compréhension,** ce qui a été fait dans le dernier exemple; cette notion est à rapprocher de celle des ensembles définis en compréhension.

L'exemple suivant montre comment créer la liste des 100 premiers termes de la suite de terme général $u_n = n^2 - 3$:

```
L = [n**2 - 3 for n in range(100)]
```

Représentations graphiques

Le langage Python permet de réaliser de multiples représentations graphiques : courbe représentative d'une fonction sur un intervalle donné, diagrammes statistiques, etc.

Une bibliothèque contient l'ensemble des instructions permettant de réaliser ces représentations : la bibliothèque **matplotlib**.

```
import matplotlib.pyplot as mp
```

En premier lieu, il convient d'importer cette bibliothèque puis de la définir en tant qu'objet appelé **mp** (pour faciliter la saisie ultérieure).

L'objectif des scripts ci-dessous n'est pas de répondre directement aux exigences du programme de seconde professionnelle en termes de capacités et connaissances, mais davantage de présenter succinctement quelques-unes des différentes options disponibles dans la bibliothèque **matplotlib**.

Nuages de points

On s'attarde ici à la simulation d'un lancer de dé à 6 faces et de l'évolution des fréquences d'apparition de la face 6.

```
import matplotlib.pyplot as mp
from random import randint

def de6(n):
    L=[]
    somme=0
    for i in range(1,n+1):
        if randint(1,6) == 6:
            somme += 1
        L.append(somme/i)
    return L
```

La variable **somme** comptabilise le nombre d'apparitions de la face 6 sur n lancers.

Le script renvoie une liste **L** contenant les valeurs successives des fréquences d'apparition de la face 6.

L'instruction **somme += 1** est équivalente à **somme = somme + 1**.

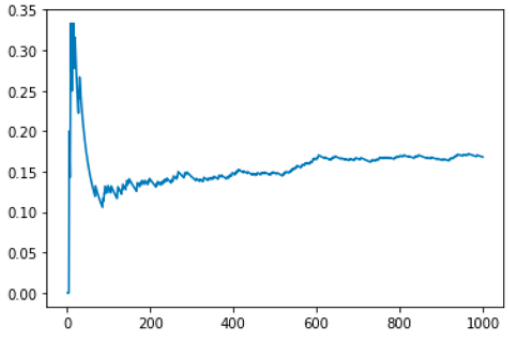
Remplaçons la dernière instruction du script par celle-ci :

```
mp.plot(list(range(1,n+1)),L)
```

Cette instruction affiche le nuage de points dont :

- les abscisses représentent les nombres entiers (numéro du lancer) appartenant à l'intervalle [1 ; n] ;
- les ordonnées représentent les fréquences d'apparition de la face 6 correspondant à chaque lancer, c'est-à-dire les valeurs de la liste **L**.

En saisissant **de6(1000)** dans la console, on obtient un graphique représentant l'évolution des fréquences d'apparition de la face 6 en fonction du nombre de lancers.



La fenêtre graphique obtenue à la page précédente est personnalisable.

```
mp.plot(list(range(1,n+1)),L,'r.')
mp.xlabel("nb lancers")
mp.ylabel("f")
mp.xlim(0,n+1)
mp.ylim(0,0.5)
mp.title("Evolution de f en fonction de n")
mp.grid()
```

On précise le **marqueur**, la « forme » des points : ici, on souhaite utiliser des points (.) rouges (r).

On ajoute une étiquette sur les axes.

On définit les valeurs limites de la fenêtre graphique.

On insère un titre et on ajoute une grille correspondant au repère.

De nombreux marqueurs, couleurs et paramètres complémentaires sont disponibles. La liste complète :

- des marqueurs est accessible à l'adresse : https://matplotlib.org/api/markers_api.html ;
- des couleurs est accessible à l'adresse : https://matplotlib.org/api/colors_api.html

Enfin, l'ensemble des options de paramétrage ainsi qu'une documentation de la bibliothèque **matplotlib** sont disponibles à l'adresse :

https://matplotlib.org/tutorials/index.html

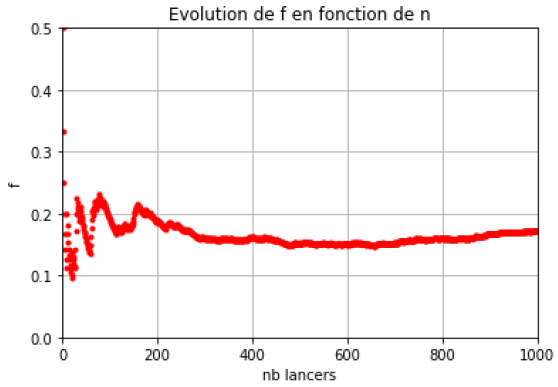


Diagramme en bâtons

Dans le même esprit que pour les nuages de points, il est possible de créer des diagrammes en bâtons.

```
import matplotlib.pyplot as mp
from random import randint

def batons():
    abscisses = list(range(1,11))
    ordonnees = [randint(0,20) for i in range(10)]
    mp.xticks(range(1,11))
    mp.grid()
    mp.bar(abscisses, ordonnees, width=0.5, color='orange')
```

Dans ce script, on crée une liste nommée **abscisses** qui permet de configurer l'axe des abscisses avec des nombres entiers appartenant à l'intervalle [1 ; 10].

On génère ensuite une liste nommée **ordonnees** contenant 10 nombres entiers aléatoires compris entre 0 et 20.

Cet exemple met en avant deux instructions supplémentaires :

- `mp.xticks(range(1,11))` qui modifie les graduations de l'axe des abscisses (de 1 à 10) ;
- `mp.bar(abscisses, ordonnees, width=0.5, color='orange')` qui permet d'afficher le diagramme en bâtons correspondant aux deux listes précédentes. Les bâtons de largeur 0.5 (c'est-à-dire 50 % de la largeur de l'unité graphique choisie pour l'axe des abscisses) sont coloriés en orange.

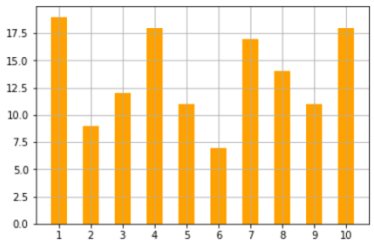
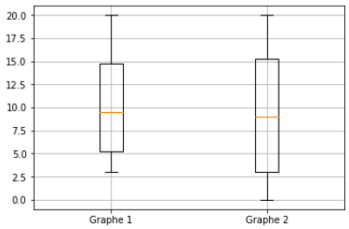


Diagramme en boîtes

Reprenons quelques éléments de l'exemple utilisé pour le diagramme en bâtons, plus particulièrement le principe permettant de générer la liste **ordonnees**.

```
def boite():
        graphe1 = [randint(0,20) for i in range(10)]
        graphe2 = [randint(0,20) for i in range(100)]
        mp.boxplot([graphe1,graphe2])
        mp.xticks([1,2],['Graphe 1','Graphe 2'])
        mp.grid()
```



Dans cet exemple, on crée deux listes **graphe1** et **graphe2**. On souhaite réaliser le diagramme en boîtes correspondant aux données aléatoires générées dans ces deux listes.

`mp.boxplot([graphe1,graphe2])` trace les deux diagrammes en boîtes

`mp.xticks([1,2],['Graphe 1','Graphe 2'])` modifie les 1re et 2e graduations de l'axe de l'abscisse afin d'y créer des étiquettes.

Courbes représentatives de fonctions

Il existe plusieurs méthodes pour réaliser la courbe représentative d'une fonction sur un intervalle donné. La première d'entre elles consiste à placer puis relier un certain nombre de points par balayage sur un intervalle donné. Prenons l'exemple de la fonction numérique réelle définie sur $[-10\,;\,10]$ qui, à tout nombre x de cet intervalle, associe le nombre x^2.

```
import matplotlib.pyplot as mp

def f(x):
    return x**2

def graphe(a,b):
    abscisses = list(range(a,b,2))
    ordonnees = [f(i) for i in abscisses]
    mp.plot(abscisses,ordonnees,'.-',color='green')
    mp.grid()
    mp.gca().xaxis.set_ticks(range(a,b,2))
```

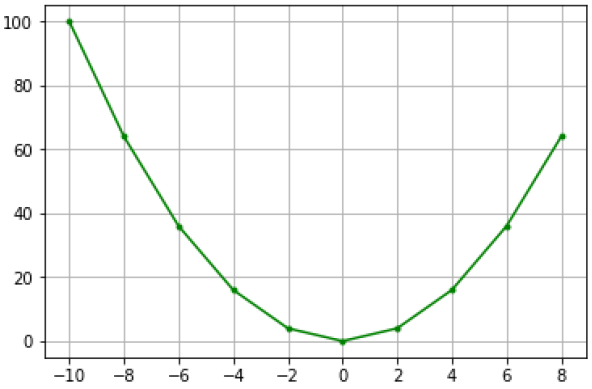
On définit la fonction à étudier.

On définit une liste des abscisses des points à tracer : sur l'intervalle [a ; b] avec un pas de 2.

On trace les points que l'on relie en vert.

On définit la graduation sur l'axe des abscisses avec un pas de 2.

La représentation graphique obtenue nous montre la limite de la méthode utilisée. En effet, le passage du discret au continu lorsqu'on choisit un pas trop important entre les abscisses ne permet pas d'obtenir une représentation fidèle de la courbe étudiée.



```
abscisses = list(range(a,b,2))
ordonnees = [f(i) for i in abscisses]
```

La boucle **for** et la fonction **range** utilisées pour générer les listes **abscisses** et **ordonnees** ne permettent de paramétrer qu'un pas exclusivement entier (en l'occurrence ici 2).

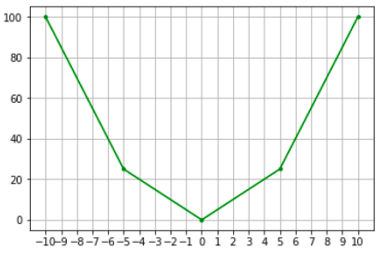
Pour contourner ce problème, il est possible d'utiliser une autre bibliothèque : **numpy**. Parmi les nombreuses possibilités qu'offre cette bibliothèque, **numpy** permet d'utiliser un certain nombre de fonctions mathématiques opérant sur des tableaux de valeurs. En outre, elle donne la possibilité de travailler avec des pas au format float, des flottants. Toutes les indications nécessaires à son utilisation devront être apportées aux élèves.

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

def f(x):
    return x**2

def graphe2(a,b,n):
    abscisses = np.linspace(a,b,n)
    ordonnees = [f(i) for i in abscisses]
    mp.plot(abscisses,ordonnees,'.-',color='green')
    mp.grid()
    mp.gca().xaxis.set_ticks(range(a,b+1))
```

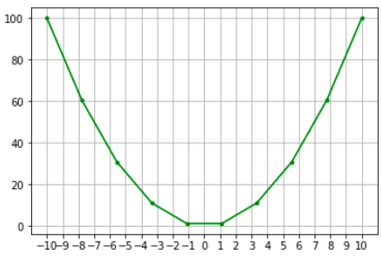
On modifie alors le script précédent en utilisant **linspace(a,b,n)**. Ceci permet de créer une liste contenant n valeurs équidistantes entre les nombres a et b.

Autrement dit, cela permet de placer sur le graphique n points d'abscisses appartenant à l'intervalle [a ; b] avec un pas de de $\frac{b-a}{n}$.

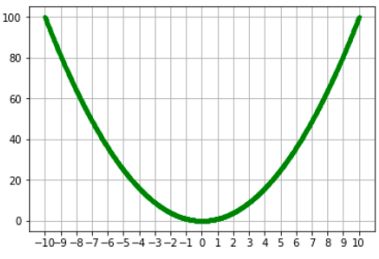
Après avoir exécuté le script, on tape **graphe2(-10,10,n)** dans la console. On obtient alors les représentations graphiques suivantes pour trois valeurs de n différentes :



n = 5



n = 10



n = 1 000

Exemples d'activités pour la classe

Tableau synthétique

Le tableau suivant, non exhaustif, met en regard les capacités travaillées en algorithmique et en programmation avec les exemples d'algorithmes proposés dans le programme de seconde professionnelle.

Domaine ou module	Algorithme	Instructions conditionnelles	Boucles bornées	Boucles non bornées	Arguments d'une fonction	Valeur(s) renvoyée(s) par une fonction	Affectation d'une variable	Entiers	Flottants	Chaînes de caractères	Booléens
Statistiques	Déterminer la fréquence d'apparition d'une lettre dans un texte.					x				x	
Probabilités	Modifier une simulation donnée (par exemple, en augmentant la taille de l'échantillon pour percevoir une version vulgarisée de la loi des grands nombres : « Lorsque *n* est grand, sauf exception, la fréquence observée est proche de la probabilité »).					x					
	Utiliser une simulation fournie pour estimer une probabilité non triviale.					x					
	Écrire des fonctions permettant de simuler une expérience aléatoire, une répétition d'expériences aléatoires indépendantes.	x	x		x	x	x	x	x		
Équations	Formaliser par un organigramme la résolution d'une inéquation du premier degré à une inconnue du type $ax < b$.										
Fonctions	Traduire un programme de calcul à l'aide d'une fonction python.					x			x		
	Calculer les images de nombres par une fonction.	x				x			x		
	Déterminer l'équation réduite d'une droite non parallèle à l'axe des ordonnées.				x	x	x		x		
	Rechercher un extremum par balayage sur un intervalle donné.	x	x		x	x	x		x		
	Rechercher un encadrement ou une valeur approchée d'une solution d'une équation du type $f(x) = 0$ par balayage sur un intervalle donné.	x		x	x	x	x		x		
Calculs commerciaux et financiers	Calculer le montant d'un intérêt simple.					x			x		
	Calculer le montant net à payer après une remise pour une facture.					x			x		
Géométrie	Chercher les triplets d'entiers pythagoriciens jusqu'à 1 000.	x	x			x		x			x
	Calculer des aires ou des volumes.					x			x		
	Calculer le diamètre d'un cylindre connaissant sa hauteur et son volume.					x			x		
	Calculer l'aire d'un carré de périmètre connu.					x			x		
	Construire une figure géométrique.		x		x		x	x			

Module « Fluctuations d'une fréquence selon les échantillons, probabilités »

Il s'agit de sensibiliser les élèves d'une part à l'expérimentation et à la simulation et d'autre part au fait que le hasard suit des lois mathématiques ; ils observent la stabilisation des fréquences et estiment la probabilité à partir de cette stabilisation.

L'activité du « jeu du 7 » permet de travailler, assez tôt dans l'année, avec les fonctions python et également avec de nombreuses autres notions d'algorithmique et de programmation sans beaucoup de prérequis.

Dans la plupart des situations proposées, la probabilité est souvent connue d'avance et les élèves retrouvent plus qu'ils n'estiment cette probabilité. Lorsque la probabilité à estimer n'est pas trivialement connue, il s'avère que la pensée algorithmique et la programmation apportent une réelle plus-value, comme il est possible de le voir sur l'exemple du franc-carreau proposé en exemple d'algorithme dans le programme de seconde professionnelle.

Le jeu du 7

Deux personnes jouent au « jeu du 7 » qui consiste, en lançant un ou deux dés, à obtenir « 7 points » pour gagner la partie.

Pour chaque lancer, une des deux personnes utilise un dé à 12 faces sur lequel il lit les points indiqués sur la face supérieure et l'autre utilise deux dés à 6 faces pour lesquels il additionne les points indiqués sur les faces supérieures des deux dés.

Problématique : les chances des deux joueurs sont-elles les mêmes ?

Voici un scénario pouvant être proposé pour répondre à cette problématique en classe de seconde professionnelle.

L'enseignant commence par demander aux élèves de reformuler cette problématique en termes de probabilités afin de s'assurer qu'ils s'approprient le contexte : « la probabilité d'obtenir 7 points en lançant deux dés à six faces est-elle égale à celle d'obtenir 7 points en lançant un dé à 12 faces ? ».

Dans un deuxième temps, l'enseignant demande aux élèves de proposer une méthode qui permettrait de répondre à la problématique posée. Un dialogue est alors amorcé afin que l'utilisation de l'algorithmique et de la programmation apparaisse comme une nécessité.

Quatre propositions de fonctions python sont proposées par l'enseignant afin de sensibiliser les élèves, d'une part, à la programmation fonctionnelle et, d'autre part, à l'utilisation de la fonction **randint()**, qui permet d'obtenir un entier aléatoire entre deux bornes. En fonction des besoins, une aide pourra être apportée à l'oral sur cette fonction et son utilisation dans le contexte proposé.

Les élèves doivent à partir de ces quatre propositions choisir deux fonctions permettant de simuler les lancers d'un dé à 12 faces et de deux dés à 6 faces.

```
def lancer1():
    return randint(1,6) + randint(1,6)
def lancer2():
    return randint(1,12)
def lancer3():
    return 2*randint(1,6)
def lancer4():
    return randint(1,12) + randint(1,12)
```

Les élèves peuvent, s'ils le souhaitent, valider leurs choix en réalisant plusieurs essais. Une attention particulière sera alors portée au fait de sensibiliser les élèves sur la nécessité d'importer la fonction **randint()** avant l'exécution du script. Un point de rappel sera éventuellement proposé sur l'import d'une bibliothèque.

```
def un_de():
    c = 0
    for i in range(10):
        if lancer2()==7:
            c += 1
    return c/10

def deux_des():
    c = 0
    for i in range(10):
        if lancer1()==7:
            c += 1
    return c/10
```

Après avoir identifié les deux fonctions qui permettent de simuler le lancer de deux dés à 6 faces et celui d'un dé à 12 faces, les élèves sont amenés à écrire un algorithme puis un script permettant de simuler 10 lancers consécutifs.

En fonction des difficultés rencontrées, les élèves pourront bénéficier d'un certain nombre de coups de pouce permettant de procéder à une différenciation pédagogique.

À titre d'exemples, voici quelques suggestions :

Coup de pouce n°1	Proposer les étapes de l'algorithme aux élèves. Ils auront alors à en déterminer la chronologie et les connexions entre les différentes étapes.
Coup de pouce n°2	Fournir à l'élève le début (ou toute autre partie) de l'algorithme.
Coup de pouce n°3	Proposer aux élèves l'algorigramme « vierge » lui indiquant ainsi sa structure qu'il aura à renseigner en aval.
Coup de pouce n°4	Fournir aux élèves l'algorithme d'une des deux fonctions (par exemple la deuxième). Ils devront alors l'associer à la fonction correspondante.

Le recours au travail de groupe permet de multiplier les approches :

- commencer par la simulation de 10 lancers de deux dés à 6 faces ;
- commencer par la simulation de 10 lancers d'un dé à 12 faces ;
- traiter les deux simulations de front.

L'algorithme associé à chacune des deux fonctions sera, le cas échéant, fourni aux élèves.

Les différents cas étudiés feront l'objet d'une trace écrite suite à la mise en commun de l'analyse des élèves.

Si certains élèves parviennent à proposer facilement l'un ou l'autre des algorithmes (ainsi que le script correspondant), l'enseignant peut leur demander de réaliser le même travail pour la seconde fonction ou de créer une fonction unique traitant simultanément les deux cas afin d'optimiser l'étude finale.

```
def un_de2(n):
    c = 0
    for i in range(n):
        if lancer2()==7:
            c += 1
    return c/n

def deux_des2(n):
    d = 0
    for i in range(n):
        if lancer1()==7:
            d += 1
    return d/n

def deux_cas(n):
    c = 0
    d = 0
    for i in range(n):
        if lancer2()==7:
            c += 1
        if lancer1()==7:
            d += 1
    return c/n,d/n
```

Pour le passage de l'algorithme au script, l'enseignant fournit aux élèves toute l'aide qu'il juge nécessaire, tant sur le plan technique (utilisation des boucles, des variables, niveaux d'indentation, etc.) que sur le plan pédagogique en insistant sur les objectifs didactiques de la séance.

En comparant leurs résultats, les élèves prennent conscience de la fluctuation de ceux-ci. L'enseignant peut alors amener les élèves à introduire un argument correspondant au nombre de lancers souhaité et leur demande de réaliser plusieurs simulations en modifiant la valeur du nombre de lancers simulés.

En leur demandant de réaliser plusieurs simulations avec différentes valeurs du nombre de lancers simulés, l'enseignant peut alors amener les élèves à comprendre l'intérêt d'ajouter comme argument de la fonction le nombre de lancers souhaité. Ceci assure une optimisation de l'expérimentation et facilite la lecture des résultats de celle-ci. L'enseignant peut profiter de ce temps pour évoquer l'intérêt d'utiliser une fonction en programmation, et notamment sur la possibilité de l'exploiter à nouveau dans d'autres cas de figure.

Dans un dernier temps, il s'agira de répondre à la problématique.

L'objectif est ici de réaliser une étude fréquentiste des probabilités en augmentant le nombre de lancers simulés. Ces derniers éléments pourront faire l'objet d'une trace écrite décontextualisée en vue d'une utilisation ultérieure dans un autre contexte, par exemple professionnel dans le cadre d'une séance de co-intervention.

La trace écrite est adaptée aux prérequis et au niveau de maîtrise des notions par les élèves.

Exemples pouvant figurer sur la trace écrite

- détermination de la probabilité étudiée à l'aide d'un tableau à double entrée ;
- définition de probabilité et de fréquence d'un événement ;
- perception d'une version vulgarisée de la loi des grands nombres ;
- stabilisation des fréquences lorsque le nombre de simulations augmente.

Prolongements possibles

- Recontextualisation : activité autour du jeu du craps simplifié (étude de la probabilité de faire un 7 ou un 11 en un lancer de deux dés à 6 faces) ;
- Utiliser une simulation fournie pour estimer une probabilité non triviale (exemple : franc-carreau) ;
- Comparer les simulations par l'algorithmique et la programmation ou en utilisant le tableur.

Analyse de l'activité au regard des modules « Fluctuations d'une fréquence selon les échantillons, probabilités » et « Algorithmique et programmation »

Cette activité permet avec peu de prérequis d'introduire ou de revoir de nombreuses connaissances et capacités de ces deux modules, comme le montre le tableau ci-après.

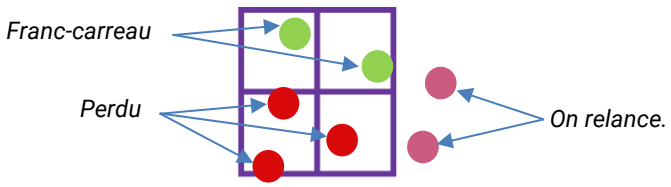
Capacités	Prérequis	Introduit dans la séquence	Connaissances	Prérequis	Introduit dans la séquence
Statistique et probabilités : fluctuations d'une fréquence selon les échantillons, probabilités					
Expérimenter pour observer la fluctuation des fréquences (jets de dés, lancers de pièces de monnaie…). Réaliser une simulation informatique, dans des cas simples, permettant la prise d'échantillons aléatoires de taille *n* fixée, extraits d'une population où la fréquence *p* relative à un caractère est connue.	X	X	Vocabulaire des probabilités : expérience aléatoire, ensemble des issues (univers) événement, probabilité. Expérience aléatoire à deux issues. Échantillon aléatoire de taille *n* pour une expérience à deux issues (avec remise).	X	X
Estimer la probabilité d'un événement à partir des fréquences.		X	Stabilisation relative des fréquences vers la probabilité de l'événement quand *n* augmente.		X
Calculer la probabilité d'un événement dans le cas d'une situation aléatoire simple. Faire preuve d'esprit critique face à une situation aléatoire simple.		X X	Dénombrements à l'aide de tableaux à double entrée ou d'arbres.		X
Algorithmique et programmation					
Analyser un problème. Décomposer un problème en sous-problèmes	X				
Repérer les enchaînements logiques et les traduire en instructions conditionnelles et en boucles		X	Séquences d'instructions, instructions conditionnelles, boucles bornées (for) et non bornées (while).		X
Choisir ou reconnaître le type d'une variable. Réaliser un calcul à l'aide d'une ou plusieurs variables	X		Types de variables : entiers, flottants, chaînes de caractères, booléens.	X	

Capacités	Prérequis	Introduit dans la séquence	Connaissances	Prérequis	Introduit dans la séquence
Modifier ou compléter un algorithme ou un programme. Concevoir un algorithme ou un programme simple pour résoudre un problème.		X X			
Comprendre et utiliser les fonctions. Compléter la définition d'une fonction. Structurer un programme en ayant recours à des fonctions pour résoudre un problème donné.		X	Arguments d'une fonction. Valeur(s) renvoyée(s) par une fonction		X X

Le jeu du franc-carreau

Le jeu du franc-carreau consiste à lancer une pièce de monnaie (ou un jeton) au sol, sur lequel un damier avec un nombre fini de carreaux est dessiné, et de regarder la position de cette pièce lorsqu'elle s'est immobilisée. La règle du jeu est la suivante :

- si le centre de la pièce se trouve en dehors du damier, le joueur relance à nouveau sa pièce ;
- si la pièce est située entièrement à l'intérieur d'une case du damier, autrement dit que la pièce n'est pas en contact avec le quadrillage, le joueur gagne la partie et on dit qu'il fait un franc-carreau ;
- si le centre de la pièce se trouve à l'intérieur du damier et que la pièce chevauche une ligne du quadrillage ou bien si elle est tangente à une des lignes du quadrillage, le joueur perd la partie.



Après quelques essais avec une pièce et un damier constitué de carrés, les élèves constatent rapidement qu'il est assez difficile de faire un franc-carreau mais ne sont pas capables d'estimer la probabilité d'en réaliser un.

Nous détaillons ci-après les étapes permettant de concevoir une simulation à fournir aux élèves afin qu'ils estiment cette probabilité.

Deux fonctions distinctes sont construites :

- une permettant d'une part de créer le quadrillage de référence ;
- une autre permettant de simuler le jeu du franc-carreau.

Création du quadrillage du damier

Afin de rendre plus visuel cet exemple, on peut créer une fonction appelée **grille()** qui permet de générer une grille carrée, qui modélise le quadrillage du damier, composée de 100 carreaux de 1 pouce de côté de la façon suivante :

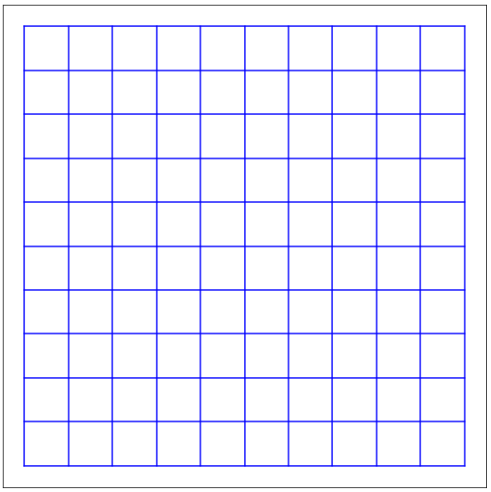
```
import matplotlib.pyplot as plt

def grille():
    plt.figure(1, figsize=(10,10))
    for i in range(11):
        plt.xticks([]),plt.yticks([])
        plt.axis('scaled')
        plt.plot([0, 10],[i, i], 'b-')
        plt.plot([i, i],[0, 10], 'b-')
```

On définit le contour et la taille de la figure (10 pouces sur 10).

On supprime les étiquettes des axes définies *a priori*, puis on leur impose une unité graphique commune.

On trace tous les pouces un trait horizontal, (puis un vertical) de longueur 10 pouces.

On obtient ainsi le damier suivant (l'échelle n'est pas respectée ici) :



Remarque : lorsque l'on fait appel à **figsize** pour définir la dimension de la figure, ce sont les pouces qui sont utilisés comme unité, Cela doit être stipulé aux élèves en rappelant qu'un pouce correspond à 2,54 cm. C'est l'occasion de travailler la proportionnalité.

Simulation du jeu

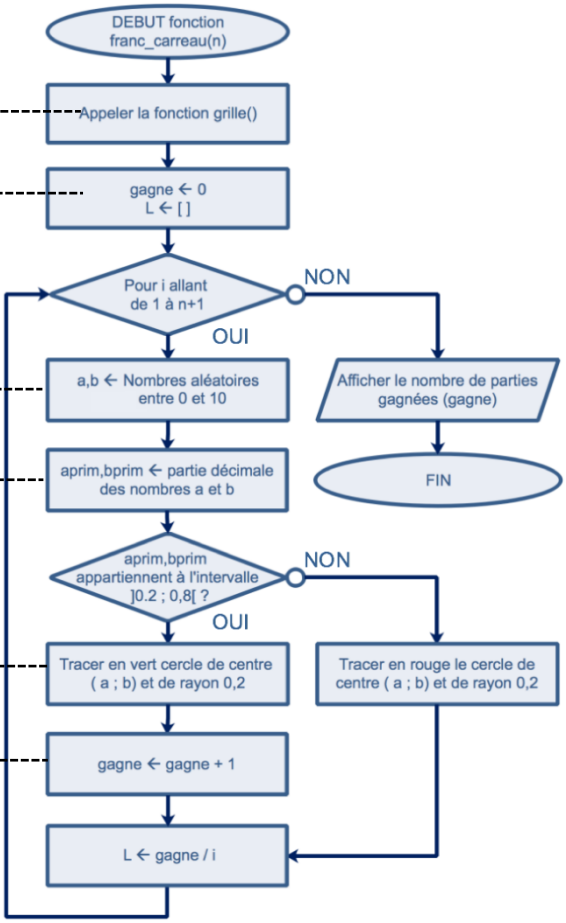
On se place ici dans la situation où la pièce lancée a pour rayon 0,2 pouce. Une seconde fonction que l'on nomme **franc_carreau()** est nécessaire pour simuler le jeu. Le principe de fonctionnement de cette fonction suit l'algorigramme ci-dessous.

La grille est créée puis la variable **gagne** (nombre de parties gagnées) et la liste **L** (regroupant les fréquences de parties gagnées) sont initialisées.

Les variables **a** et **b** représentent les coordonnées du centre de la pièce dans la grille.

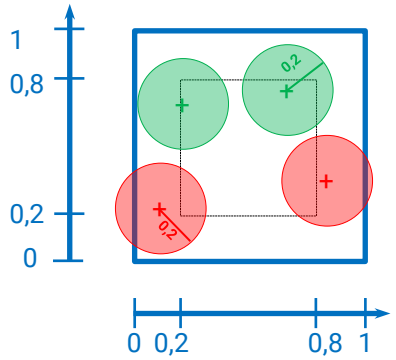
La grille étant composée de 100 carreaux, on simplifie le calcul à l'aide des variables **aprim** et **bprim** pour ramener l'étude à la position de la pièce dans un carreau.

Le test permet de savoir si la pièce est bien à l'intérieur du carreau de manière franche, c'est à dire qu'elle ne touche pas ses bords. Si c'est le cas, la pièce est tracée en vert, la variable **gagne est** incrémentée de 1, sinon la pièce est tracée en rouge.



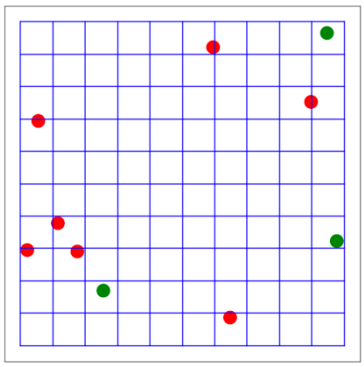
```
from random import random

def franc_carreau(n):
    grille()
    gagne = 0
    L = []

    for i in range(1,n+1):
        a = 10*random()
        b = 10*random()
        aprim = a - int(a)
        bprim = b - int(b)

        if 0.2 < aprim < 0.8 and 0.2 < bprim < 0.8:
            cercle = plt.Circle((a,b),0.2, color = 'g')
            plt.gca().add_artist(cercle)
            gagne +=1
        else:
            cercle = plt.Circle((a,b),0.2, color = 'r')
            plt.gca().add_artist(cercle)

        L.append(gagne/i)
    print("Parties gagnées : ",gagne," sur ",n)
```

Si les coordonnées du centre de la pièce appartiennent à l'intervalle] 0,2 ; 0,8 [, alors aucun point de la pièce ne touchera les bords du carreau :

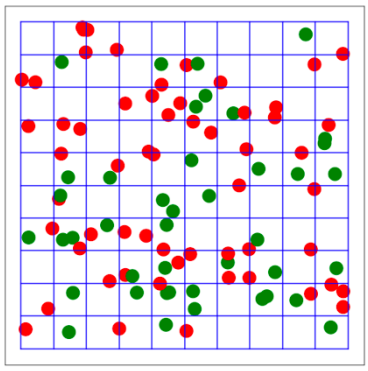


Pour chaque lancer de pièce, on enregistre la fréquence de parties gagnées dans la liste L et on affiche la proportion de parties gagnées.

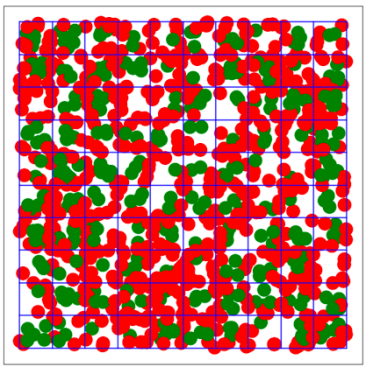
Remarque : il est important de faire constater aux élèves que même si la fonction s'appelle **Circle()**, mot qui se traduit par cercle, ce sont des disques qui sont dessinés.



n = 10



n = 100



n = 1 000

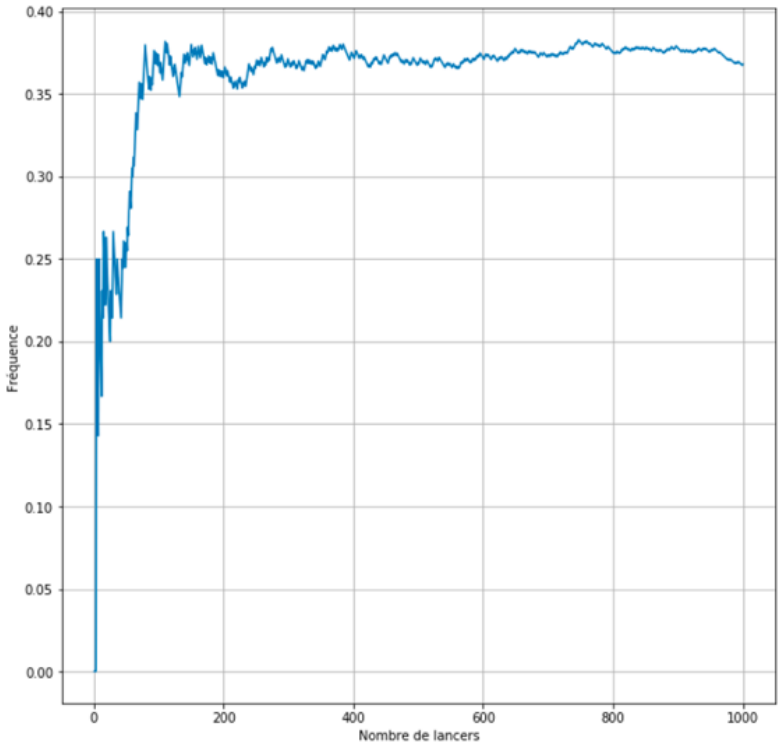
Après avoir exécuté le script, on évalue la fonction **franc_carreau()** dans la console pour différentes valeurs du paramètre n. Les dessins suivants représentent trois simulations, pour trois valeurs différentes de n, nombre de lancers de pièces.

En complément de cette étude, le professeur pourra proposer la visualisation de l'évolution des fréquences lorsque le nombre n de lancers augmente.

En prenant soin de bien respecter les niveaux d'indentation, on utilise alors les fréquences enregistrées dans la liste **L** afin de créer un graphique montrant leur évolution lorsque le nombre n de lancers augmente comme on le voit dans les quatre dernières lignes du script ci-dessous.

```
def franc_carreau(n):
    grille()
    gagne = 0
    L = []

    for i in range(1,n+1):
        a = 10*random()
        b = 10*random()
        aprim = a - int(a)
        bprim = b - int(b)

        if 0.2 < aprim < 0.8 and 0.2 < bprim < 0.8:
            cercle = plt.Circle((a,b),0.2, color = 'g')
            plt.gca().add_artist(cercle)
            gagne +=1
        else:
            cercle = plt.Circle((a,b),0.2, color = 'r')
            plt.gca().add_artist(cercle)

        L.append(gagne/i)
    print("Parties gagnées : ",gagne," sur ",n)

    plt.figure(2, figsize=(10,10))
    plt.xlabel('Nombre de lancers'),plt.ylabel('Fréquence')
    plt.grid()
    plt.plot(list(range(1,n+1)),L)
```

Le graphique ci-dessous illustre un exemple de résultat obtenu pour 1000 lancers après avoir exécuté le script et saisi **franc_carreau(1000)** dans la console.



Le professeur fera observer une stabilisation des fréquences autour d'une valeur environ égale à 0,37. On conclut que la probabilité de gagner au jeu du franc carreau est environ égale à 0,37.

Module « Statistique à une variable »

Les statistiques descriptives sont travaillées depuis le cycle 4. Le tableur constitue un outil incontournable, mais le recours à la programmation présente d'autres intérêts parmi lesquels :

- manipuler les formules de calcul des indicateurs qui doivent être explicitées pour être programmées alors que ces formules sont plus opaques pour l'élève lors de l'utilisation d'un tableur ;
- manipuler des chaînes de caractères ou de grandes séries qui peuvent être issues de données réelles, et qui sont bien plus facilement manipulables que sur un tableur.

Les calculs statistiques seront grandement facilités lors de l'introduction des listes en classe de première, mais on peut dès la classe de seconde créer quelques fonctions pour, par exemple, calculer des fréquences.

Analyse fréquentielle d'un texte

Le travail sur les séries statistiques à une variable abordé au cycle 4 et poursuivi en classe de seconde professionnelle trouvera une plus-value sur le traitement de grandes séries par l'utilisation de Python, ces séries étant moins aisément manipulables à l'aide d'un tableur.

Un bon exemple réside en l'analyse fréquentielle d'un texte, autrement dit le calcul des fréquences d'apparition de chaque lettre dans un texte, car il donne l'occasion de travailler sur un type de variable rarement rencontré, les chaînes de caractères.

Aucune technicité n'est attendue sur la manipulation desdites chaînes ; les fonctions python sous-jacentes seront fournies aux élèves.

Dans un premier temps, un texte directement utilisable, sans accent ni signe de ponctuation, *'jaimelesmathematiquesetjadorepython'* peut être proposé. Les guillemets marquent le début et la fin de cette chaîne de caractères.

Dans un second temps, le texte sur lequel est effectuée l'analyse peut être « nettoyé » en enlevant les majuscules, les symboles de ponctuation, les accents ou les caractères spéciaux.

On se place ici dans le cas d'une chaîne de caractères assez courte, mais cette activité prend tout son sens avec de longs textes ; on peut par exemple s'amuser à comparer les occurrences de lettres dans différentes langues ou constater sur un long extrait de *La disparition* de Georges Perec que ce texte est bien atypique. Il peut être intéressant de comparer de façon automatique des textes d'origines diverses : textes classiques, textes de chanson de styles variés.

Après avoir présenté quelques exemples de chaînes de caractères aux élèves et leur avoir montré qu'il est possible de les parcourir de la première lettre à la dernière, on considère le texte suivant :

```
texte = 'jaimelesmathematiquesetjadorepython'
```

Il est proposé aux élèves d'utiliser la fonction suivante :

```
def mystere(texte):
    L=0
    for x in texte:
        L = L+1
    return L
```

On obtient sur la console :

```
In [2]: mystere(texte)
Out[2]: 35
```

Les élèves conjecturent puis démontrent qu'elle permet de déterminer la longueur de la chaîne de caractères.

Il peut leur être demandé d'écrire une fonction permettant de compter le nombre d'occurrences d'une lettre dans un texte.

```
def compte(lettre,texte):
    compteur=0
    for x in texte:
        if x == lettre:
            compteur = compteur+1
    return compteur
```

En faisant un essai avec la lettre « a » :

```
In [3]: compte('a',texte)
Out[3]: 4
```

Cela signifie que la lettre « a » est présente 4 fois dans la chaîne de caractères.

Il est alors possible d'écrire une fonction utilisant les deux précédentes qui permet de déterminer la fréquence d'apparition d'une lettre dans une chaîne de caractères.

```
def frequence_lettre(lettre,texte):
    return compte(lettre,texte)/mystere(texte)
```

Pour la lettre « a » on obtient dans la console :

```
In [4]: frequence_lettre('a',texte)
Out[4]: 0.11428571428571428
```

Nettoyage d'un texte

« Nettoyer » un texte consiste à obtenir une chaîne de caractères dépourvue d'accents et de signes de ponctuation tout en modifiant la casse pour n'obtenir que des lettres minuscules. L'enseignant donne les différentes fonctions nécessaires.

Pour y parvenir, le problème est décomposé en plusieurs sous-problèmes en créant diverses fonctions python, chacune ayant un but précis.

Dans cet exemple, il s'agit de traiter le texte suivant : 'Les mathématiques c'est aussi automatique ! (Le langage) Python est à découvrir !'

```
texte = "Les mathématiques c'est aussi automatique ! (Le langage) Python est à découvrir !"
```

En premier lieu, le texte à nettoyer est défini ainsi que les signes de ponctuation que l'on souhaite supprimer.

```
ponctuation = '!?,.:;"\') (_-'
```

Remarque : en fonction du texte à étudier, il peut être nécessaire de tenir compte par la suite d'autres signes de ponctuation, qui auraient été oubliés ici. Il peut être intéressant de demander aux élèves d'effectuer ces modifications en ajoutant des caractères à la chaîne de caractères « ponctuation » déjà fournie.

Une fonction permettant d'enlever les signes de ponctuation est définie.

L'instruction **replace** permet de remplacer dans la variable **texte** tous les caractères listés dans la variable ponctuation par un élément vide :

```
def sans_ponctuation(texte):
    for x in texte:
        if x in ponctuation:
            texte = texte.replace(x, "")
    return texte
```

La fonction **sans_accent()** est une fonction qui remplace un caractère accentué par son homologue sans accent. Cette fonction se veut également évolutive.

```
def sans_accent(texte):
    texte = texte.lower()
    accents = "àâäåçéèêëîïôöùûüÿ"
    sansAccent = "aaaaceeeeiiooouuuy"
    return texte.translate(str.maketrans(accents, sansAccent))
```

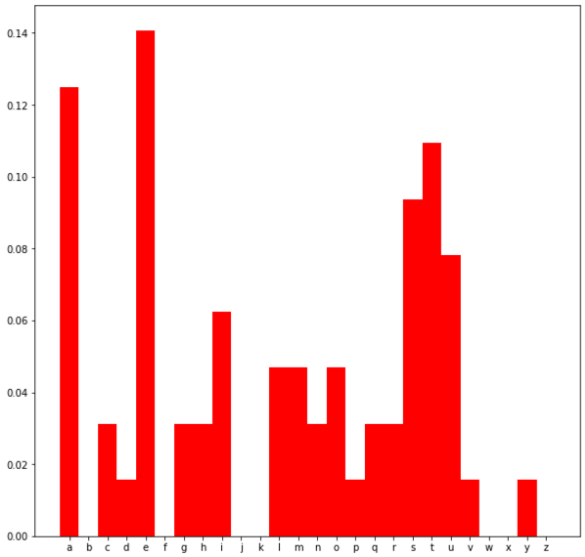
La fonction dite de nettoyage est définie en utilisant les deux fonctions créées précédemment :

```
def nettoyage(texte):
  texte= sans_ponctuation(texte)
  texte= sans_accent(texte)
  return texte
```

En classe de première, ce travail qui intègre les listes trouve toute sa place avec l'avantage de pouvoir construire un diagramme des fréquences d'apparition de chaque lettre dans le texte choisi. Cela peut se faire dans un programme principal en utilisant la bibliothèque **matplotlib** comme décrit ci-dessous :

```
import matplotlib.pyplot as plt

def diagramme_frequences(texte):
    texte=nettoyage(texte)
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    frequenceslettres=[]
    for lettre in alphabet:
          frequenceslettres.append(frequence_lettre(lettre,texte))
    plt.bar(range(26),frequenceslettres,1.0,color='r')
    plt.xticks(range(26), alphabet)
```

Les résultats se présentent alors ainsi :



Une analyse fréquentielle ainsi réalisée sur un texte représentatif comme un article de journal ou un extrait d'œuvre littéraire peut permettre de caractériser la langue d'écriture du texte et par extension de reconnaître en quelle langue un texte a été rédigé.

Module « Fonctions »

Calcul d'un prix

Il s'agit de comparer les coûts des photocopies dans deux magasins de reprographie :

Pho-tout-copie	Copies Express
de 1 à 30 photocopies : 0,40 € l'unité	de 1 à 40 photocopies : 0,30 € l'unité
à partir de la 31e : 0,20 € l'unité	à partir de la 41e : 0,25 € l'unité

Un scénario possible en classe de seconde est proposé ci-dessous.

Dans un premier temps, l'enseignant demande aux élèves de comparer les coûts dans différents cas afin de les aider à s'approprier la situation.

Il leur demande ensuite de définir une fonction python qui permet de comparer les offres des deux magasins pour n'importe quel nombre de photocopies à réaliser.

Il peut, si besoin est, leur demander de compléter la fonction permettant le calcul correspondant au magasin Pho-tout-copie :

```
def magasin1(nombre):
    if nombre<31:
        prix=
    else:
        prix=
    return prix
```

Après avoir vérifié les réponses, l'enseignant peut demander aux élèves d'écrire la fonction correspondant à un calcul analogue pour le magasin Copie express.

Voici une fonction répondant à ce problème :

```
def magasin2(nombre):
    if nombre<41:
        prix=nombre*0.3
    else:
        prix=40*0.3+(nombre-40)*0.25
    return prix
```

L'enseignant peut demander aux élèves d'expliquer ce que fait la fonction suivante :

```
def comparaison():
    k=0
    while magasin1(k)>magasin2(k):
        k=k+1
    return k
```

Il peut également leur demander d'expliquer le choix de la condition de la boucle non bornée.

Conversion Celsius / Fahrenheit

On considère la fonction suivante qui permet de convertir en degré Fahrenheit une température exprimée en degré Celsius :

```
def conversion_Celsius_Fahrenheit(C):
    return 32+1.8*C
```

L'enseignant peut demander aux élèves de traduire par une phrase la formule de conversion avant de leur demander de déterminer une fonction qui permet de convertir en degré Celsius une température exprimée en degré Farhrenheit, ce qui les conduit à résoudre une équation du premier degré.

Encadrement d'une solution d'une équation par balayage

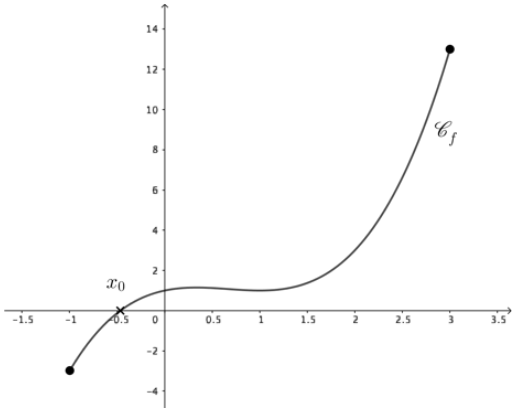
La méthode du balayage pour encadrer une solution d'équation est présentée ici sans contextualisation spécifique. Cette méthode s'applique à tout type de fonctions, de la seconde professionnelle à la classe terminale.

Considérons la fonction f définie sur $[-1\,;\,3]$ par $f(x) = x^3 - 2x^2 + x + 1$.

On souhaite déterminer un encadrement de la solution de l'équation $f(x) = 0$ en procédant par balayage. On admet l'existence d'une seule solution de cette équation sur l'intervalle $[-1\,;\,3]$.

Remarque : l'existence de la solution de l'équation $f(x) = 0$ sur cet intervalle est une conséquence de la continuité de la fonction f et du fait que $f(-1)$ et $f(3)$ sont de signes contraires. L'unicité se déduit des variations de cette fonction. Ce n'est pas ce qui importe ici, car on admet que c'est le cas.

On note x_0 l'unique solution de l'équation $f(x) = 0$ sur l'intervalle $[-1\,;\,3]$.



On cherche ici un encadrement de la valeur de x_0 d'amplitude h, nombre réel strictement positif.

La méthode peut être décrite de la façon suivante : on part de la borne gauche de l'intervalle, c'est-à-dire ici -1. L'image de ce nombre par la fonction f est strictement négative. On calcule les images des nombres $-1 + h$, $-1 + 2h$, $-1 + 3h$, etc. jusqu'à observer un changement de signe. La solution x_0 de l'équation $f(x) = 0$ est comprise entre le plus grand nombre dont l'image calculée est strictement négative et le premier dont l'image calculée est strictement positive.

On définit deux fonctions :

```
def f(x):
    return x**3-2*x**2+x+1

def encadrement_solution(f,a,h):
    k=0
    while f(a+k*h)<0:
        k=k+1
    return a+(k-1)*h,a+k*h
```

Dans la console, avec un pas de 0,1 :

```
In [2]: encadrement_solution(f,-1,0.1)
Out[2]: (-0.5, -0.3999999999999999)
```

On obtient l'encadrement $-0{,}5 < x_0 < -0{,}4$.

```
In [3]: encadrement_solution(f,-1,0.01)
Out[3]: (-0.47, -0.45999999999999996)
```

Avec un pas de 0,01, l'encadrement obtenu est : $-0{,}47 < x_0 < -0{,}46$.

```
In [4]: encadrement_solution(f,-1,0.000001)
Out[4]: (-0.465572, -0.46557100000000007)
```

Avec un pas de 0,000001, l'encadrement obtenu est : $-0{,}465572 < x_0 < -0{,}465571$.

L'enseignant peut amener les élèves à voir qu'il est possible de construire une condition universelle, qui conviendrait aussi à une situation dans laquelle l'image de la borne de gauche de l'intervalle est strictement positive, à l'aide de l'étude du signe d'un produit :

```
def encadrement_solution2(f,a,h):
    k=0
    while f(a)*f(a+k*h)>0:
        k=k+1
    return a+(k-1)*h,a+k*h
```

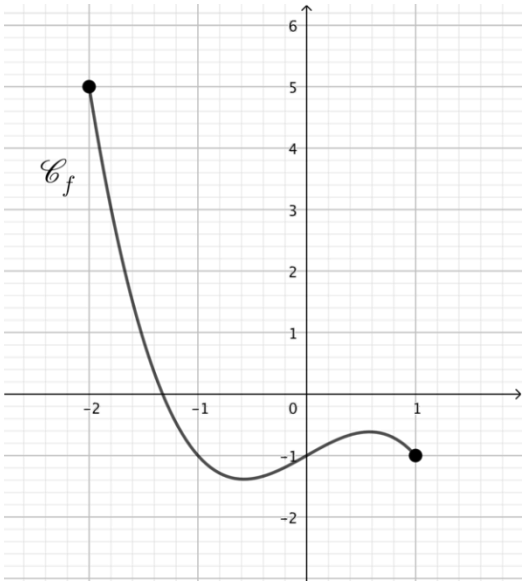
Extremum par balayage

La méthode du balayage pour approcher un extremum d'une fonction est présentée ici sans contextualisation spécifique. Cette méthode s'applique à tout type de fonctions, de la seconde professionnelle à la classe terminale.

Considérons la fonction f définie sur $[-2\,;\,1]$ par $f(x) = -x^3 + x - 1$.

On cherche à déterminer par balayage un encadrement de son minimum à un pas donné.

Remarque : Comme la fonction f est continue sur un intervalle fermé borné, on est assuré de l'existence d'un minimum, mais aussi d'un maximum, sur cet intervalle. Mais là encore, tel n'est pas le propos.



On choisit un pas h , réel strictement positif.

La méthode peut être décrite de la façon suivante : on part de la borne gauche de l'intervalle, c'est-à-dire -2, puis on compare $f(-2)$ à $f(-2+h)$ et on conserve la plus petite des deux valeurs.

On compare cette valeur à $f(-2+2h)$, et on conserve à chaque fois la plus petite valeur parmi celles calculées, etc.

```
def f(x):
    return -x**3+x-1

def minimum(f,a,b,pas):
    k=0
    mini=f(a)
    c=a
    while a+k*pas<=b:
        if f(a+k*pas)<mini:
            mini=f(a+k*pas)
            c=a+k*pas
        k=k+1
    return c,mini
```

```
In [2]: minimum(f,-2,1,0.0001)
Out[2]: (-0.5773999999999999, -1.384900175176)
```

Le programme permet de penser que le minimum de f est voisin de $-1{,}38$.

Cette méthode a des limites : on détermine en fait la plus petite image parmi celles des nombres $-2+h, -2+2h, -2+3h$, etc., mais on peut ne pas trouver le minimum de la fonction. Dans le cas de la fonction choisie ici, le minimum est atteint en $-\sqrt{\frac{1}{3}}$ et vaut $-\frac{2\sqrt{3}}{9}-1$.

De plus, dans le cas où l'on trouve bien le minimum ainsi, la fonction affiche un antécédent de ce minimum, mais il peut y en avoir d'autres.

Il est intéressant de montrer aux élèves, à l'aide d'exemples bien choisis, les limites évoquées ci-dessus.

Module « Suites » (en classes de première et terminale)

Calcul d'une somme de termes consécutifs d'une suite

Pour illustrer cette partie, nous allons prendre appui sur une situation contextualisée :

> On peut lire une annonce surprenante sur un site en ligne dédié à la vente de voitures :
>
> « À vendre boulons de jante (à 5 trous) ! Le 1er boulon est à 1 centime d'euro, le 2e à deux centimes d'euro, le 3e à quatre centimes d'euro et ainsi de suite, le prix doublant à chaque boulon supplémentaire. Si vous achetez tous les boulons des quatre jantes, la voiture est offerte (très bon état et contrôle technique « OK » !)… »
>
> En sachant que la cote de la voiture en question est de 7 500 €, est-il intéressant d'obtenir le véhicule de cette façon ?

Les prix, en centime d'euro, des boulons successifs sont les premiers termes d'une suite géométrique de premier terme 1 et de raison 2.

Tout d'abord, l'enseignant peut demander aux élèves de créer une liste des prix des boulons successifs :

```
L = [1*2**k for k in range(20)]
```

Ensuite, il leur demande de créer une fonction qui permet d'ajouter tous les éléments d'une liste donnée :

```
def somme(L):
    S=0
    for x in L:
        S = S + x
    return S
```

En saisissant dans la console **somme(L)**, on répond à la question posée puisqu'on obtient :

```
In [3]: somme(L)
Out[3]: 1048575
```

Soit 10 485,75 €, ce qui permet de répondre à la problématique : cette offre n'est vraiment pas intéressante.

Seuil

Un client place dans une banque un capital de 1 000 € sur un placement à intérêts composés au taux annuel de 5 % par an.

L'enseignant peut demander aux élèves d'écrire une fonction permettant de calculer le capital obtenu après un certain nombre d'années, noté nb_annees.

```
def capital(nb_annees):
    return 1000*1.05**nb_annees
```

Il peut également leur demander d'écrire une fonction permettant de déterminer le nombre d'années nécessaires au doublement du capital initial.

```
def doublement():
    k=0
    C=1000
    while C<2000:
        C=C*1.05
        k=k+1
    return k
```

Il est intéressant de leur faire tester plusieurs valeurs du capital initial, avant de conjecturer que le nombre d'années nécessaires au doublement du capital initial est indépendant du capital initial investi. L'utilisation des fonctions logarithmes népérien ou décimal permettra de vérifier cette conjecture.

Annexe : sitographie

- L'installation de l'environnement Anaconda est expliquée page 5 d'un document figurant sur le site de mathématiques physique-chimie de l'académie de Lille. Ce document propose également d'autres exemples de programmes.
- Le site de mathématiques de l'académie de Paris présente un environnement en ligne que l'on peut utiliser si l'on souhaite éviter toute installation.
- Pour en savoir plus sur les flottants et sur la représentation des nombres en machine, la vidéo de Sylvie BOLDO, chercheuse à l'INRIA, précise quelques éléments de contexte sur les flottants et sur les conséquences de cette représentation.