Chapitre 2
Suites de nombres réels

1. Suites et racines carrées

L'idée de répéter un calcul en changeant les nombres utilisés à chaque étape est très ancienne puisqu'on en trouve la trace à Babylone, 1800 ans avant J.-C. En Grèce, on a souvent eu recours aux suites pour calculer des racines carrées.

1.1 La méthode d'Archytas de Tarente

Archytas de Tarente (vers 435 av. J.-C. ; 347 av. J.-C) était un disciple de Pythagore. Ses travaux ont concerné la notion de moyenne. Étant donnés deux nombres a et b, leur moyenne arithmétique m est égale à $\frac{a+b}{2}$, leur moyenne géométrique g est définie par $g^2 = ab$ et leur moyenne harmonique h est définie par $\frac{2}{h} = \frac{1}{a} + \frac{1}{b}$. On démontre que $h < g < m$. Archytas de Tarente a utilisé cette relation pour calculer des racines carrées.

Par exemple, pour trouver une valeur approchée de $\sqrt{3}$, il commence par écrire $3=2\times\frac{3}{2}$ puis, comme le montre le tableau suivant, il déroule ses calculs :

Étape n°	x	y	Moyenne arithmétique de x et de y	Moyenne harmonique de x et de y
1	2	$\frac{3}{2}$	$\frac{7}{4}$	$\frac{12}{7}$
2	$\frac{7}{4}$	$\frac{12}{7}$	$\frac{97}{56}$	$\frac{168}{97}$
3	$\frac{97}{56}$	$\frac{168}{97}$	$\frac{18817}{10864}$	$\frac{32592}{18817}$

Avec les notations actuelles, on peut écrire $\frac{18817}{10864}$=1,73205081… et $\frac{32592}{18817}$=1,732050805….. Le nombre $\sqrt{3}$ est connu avec une erreur qui porte sur la 9e décimale seulement. On peut généraliser la méthode d'Archytas de Tarente à un nombre réel positif A quelconque en utilisant le programme qui suit :

```
# Calcul d'une racine carrée avec la méthode d'Archytas de Tarente
A=eval(input("Valeur de A : "))
x,y=2,A/2
for i in range(1,6):
    m=(x+y)/2
    h=x*y/m
    print("x=",x," et y=",y)
    x,y=m,h
```

On a choisi de faire le calcul en cinq étapes car l'algorithme est très performant.

Voici par exemple le calcul de $\sqrt{2}$:

```
Valeur de A : 2
x= 2  et y= 1.0
x= 1.5  et y= 1.3333333333333333
x= 1.4166666666666665  et y= 1.411764705882353
x= 1.4142156862745097  et y= 1.41421143847487
x= 1.4142135623746899  et y= 1.4142135623715
```

1.2 La méthode de Héron d'Alexandrie

Au I^er^ siècle après J.-C., le mathématicien et ingénieur grec Héron d'Alexandrie (75-150) a exposé une méthode[2] très rapide et très simple pour calculer la racine carrée d'un nombre réel A positif. On choisit une valeur approchée quelconque u de $\sqrt{A}$ et on calcule $v = \frac{1}{2}(u + \frac{A}{u})$. Il est facile de voir que $\sqrt{A}$ est compris entre u et v. On recommence le calcul en remplaçant u par v et on continue ainsi jusqu'à atteindre la précision désirée. Ainsi, le tableau qui suit montre les cinq premières étapes du calcul de $\sqrt{5}$.

Étape n°	u	v
1	3.0	2.3333333333333335
2	2.3333333333333335	2.238095238095238
3	2.238095238095238	2.2360688956433634
4	2.2360688956433634	2.236067977499978
5	2.236067977499978	2.23606797749979

On peut généraliser la méthode de Héron d'Alexandrie à un nombre réel positif A quelconque avec ce programme :

```
# Calcul d'une racine carrée avec la méthode de Héron d'Alexandrie
A=eval(input("Valeur du nombre A ? "))
n=eval(input("Valeur de n ? "))
u=A/2
for i in range(1,n+1):
    v=(u+A/u)/2
    print(v)
    u=v
```

Pour calculer $\sqrt{A}$ quand A est positif, on peut choisir un nombre positif quelconque comme première approximation de $\sqrt{A}$, y compris le nombre A lui-même. Calculons par exemple les 8 premières valeurs approchées de $\sqrt{10}$ avec ce programme :

```
from math import*
a=10
n=8
u=a
for i in range(1,n+1):
    v=(u+a/u)/2
    print(v)
    u=v
```

On obtient ces résultats :

```
3.659090909090909
3.196005081874647
3.16245562280389
3.162277665175675
3.162277660168379
3.162277660168379
3.162277660168379
3.162277660168379
```

Dès la cinquième approximation, on a obtenu 15 décimales exactes. On démontre que la suite des approximations de $\sqrt{10}$ est illimitée. Le développement décimal de $\sqrt{10}$ possède donc une infinité de décimales. En effet, $\sqrt{10}$ est un nombre irrationnel qui ne peut pas être représenté exactement par une fraction. C'est aussi un nombre algébrique qui vérifie l'équation $x^2=10$.

1.3 Le calcul d'une racine cubique

On peut généraliser la méthode de Héron et calculer la racine cubique d'un nombre positif a. Si x_{n-1} est une valeur approchée de cette racine, la valeur approchée suivante x_n est donnée par le calcul $x_n=\frac{1}{2}\left(x_{n-1}+\frac{a}{x_{n-1}^2}\right)$. Le programme qui suit calcule des valeurs approchées successives u et v de $\sqrt[3]{a}$. Le calcul est arrêté quand $|v-u|<10^{-6}$. Comme Python peut calculer $\sqrt[3]{a}$, on pourra comparer la valeur exacte à la valeur calculée.

```
# Calcul d'une racine cubique. Méthode de Héron d'Alexandrie
from math import*
a=eval(input("Valeur du nombre positif a ? "))
u=a/3
v=u/2+a/(2*u*u)
e=abs(u-v)
while e>0.000001:
    u=v/2+a/(2*v*v)
    e=abs(v-u)
    v=u
print("Valeur approchée de la racine cubique de ",a," =",v)
print("Valeur exacte selon Python = ", a**(1/3))
```

Voici le résultat obtenu pour $a=100$:

```
Valeur du nombre positif a ? 100
Valeur approchée de la racine cubique de 100  = 4.641589088997662
Valeur exacte selon Python =  4.641588833612778
```

Chapitre 2.2
Fonctions et modules

1. Les fonctions

1.1 Pourquoi utiliser des fonctions ?

Lorsque l'on développe, on utilise énormément de fonctions, comme par exemple **`print`** ou **`input`**. Ces dernières sont assez simples à manipuler pour plusieurs raisons :

– elles portent un nom simple qui indique bien à quoi elles servent ;

– elles prennent des paramètres qui permettent de varier la manière dont on les utilise ;

– on n'a pas besoin de savoir comment elles sont écrites, juste ce qu'elles vont faire.

Lorsque vous écrivez votre propre code, vous allez devoir concevoir des algorithmes plus ou moins complexes, et lorsque vous n'êtes pas organisé, vous allez produire ce que nous avons fait jusqu'à présent : un code parfaitement linéaire.

L'inconvénient principal est le suivant : le code est une longue prose, sans repères particuliers. Il est difficile d'en isoler une partie et de toujours savoir quelle ligne est appelée à quel moment. Usuellement, on considère qu'une fonction bien faite doit faire une dizaine de lignes en moyenne, 20 à 25 au maximum.

Ces métriques ne sont pas, bien entendu, des obligations, mais un ordre d'idée à garder en tête et à essayer de respecter pour avoir un code lisible et compréhensible par tous, y compris par vous quelques mois plus tard, lorsque ce que vous avez écrit ne sera plus aussi frais qu'au moment où vous l'écrivez.

En effet, un code trop long est difficile à lire, à appréhender et donc à maintenir.

Le constat est donc clair, il faut organiser son code pour qu'il soit fait de petites briques simples et faciles à identifier. La réelle difficulté, c'est de savoir comment délimiter ces briques, comment en faire de belles fonctions qui soient suffisamment précises pour faire ce que vous souhaitez en détail, mais aussi suffisamment génériques pour ne pas avoir deux fonctions qui sont quasiment identiques et qui ne se distinguent que par un détail.

Un bon endroit pour commencer, c'est de regarder le code produit jusqu'à maintenant (la solution de l'exercice de la fin du chapitre précédent) et d'identifier des doublons dans le code :

```
print("Saisissez le nombre à deviner")
while True:
    nombre = input("Saisissez un nombre entre 0 et 99: ")
    try:
        nombre = int(nombre)
    except:
        pass
    else:
        if 0 <= nombre <= 99:
            break

# PARTIE 2
print("Essayez de trouver le nombre à deviner")
while True:  # BOUCLE 1
    while True:  # BOUCLE 2
        essai = input("Saisissez un nombre entre 0 et 99: ")
        try:
            essai = int(essai)
        except:
            pass
        else:
            if 0 <= essai <= 99:
                break  # Boucle 2

    if essai < nombre:
        print("Trop petit")
    elif essai > nombre:
        print("Trop grand")
    else:
        print("Gagné!")
        break  # Boucle 1
```

Remarque

Vous trouverez cet exemple dans le répertoire Guide du dépôt. Il se nomme 11_JEU_-guess_the_number.py.

On voit que dans cet extrait de code, demander la saisie du nombre qu'il faut trouver et celle d'un nombre à deviner est quasiment la même chose : il n'y a que le premier affichage qui indique à l'utilisateur ce que l'on attend de lui qui change.
Notez que l'élimination de doublons de code est quelque chose de très important, puisque lorsque vous devez maintenir un code, si vous devez changer quelque chose, il faut le répercuter sur tous les doublons et qu'il peut être assez aisé d'en manquer un dans l'opération.
Cet objectif d'élimination des doublons est donc l'endroit de notre fil rouge où l'on va commencer par définir notre première fonction utile.

Par contre, n'oubliez pas que dans la vraie vie, vous devez réfléchir d'abord et coder après et que, par conséquent, vous devez d'abord définir quelles briques vous allez créer avant de réellement les créer et non pas pondre un code d'abord et réfléchir à la manière de le rendre lisible après.

1.2 Introduction aux fonctions

1.2.1 Comment déclarer une fonction

En Python, les principes syntaxiques sont toujours les mêmes. Le code d'une fonction étant un bloc, la syntaxe d'une fonction est celle d'un bloc.
On va donc écrire le mot-clé **`def`** permettant d'indiquer que l'on définit une fonction, puis le nom de cette fonction, suivi de parenthèses (on verra plus tard ce que l'on peut y mettre) et le fameux deux-points. Cette première ligne est nommée la **signature de la fonction**.
Tout ce qui suit et qui est indenté est le **corps de la fonction**. Voyons ce que cela donne :

```
def demander_saisie_nombre():
    while True:
        saisie = input("Saisissez un nombre entre 0 et 99: ")
        try:
            saisie = int(saisie)
        except:
            pass
        else:
            if 9 <= saisie <= 99:
                break
    return saisie
```

À l'exception de la première et de la dernière ligne, l'ensemble de cet extrait de code est rigoureusement identique à la partie qui était en doublon dans notre première version du jeu.
La seule différence est le nom de la variable qui était **`nombre`**, puis **`essai`**, et qui est maintenant **`saisie`**.
En effet, le nom des variables correspondait, dans le programme de départ, respectivement au nombre à deviner puis à l'essai du joueur.
Ici, on est dans une fonction qui a simplement pour but de demander la saisie d'un nombre quelconque. Au niveau de la fonction, on ne sait pas à quoi ce nombre va servir, on ne sait pas non plus son nom et on n'a pas besoin de le savoir. On sait juste qu'il s'agit d'une saisie, on décide donc de la nommer ainsi.
Là où les choses deviennent intéressantes, c'est lorsque l'on va utiliser notre fonction :

```
# PARTIE 1
print("Saisissez le nombre à deviner")
nombre = demander_saisie_nombre()

# PARTIE 2
print("Essayez de trouver le nombre à deviner")
while True:
    essai = demander_saisie_nombre()
```

```
    if essai < nombre:
        print("Trop petit")
    elif essai > nombre:
        print("Trop grand")
    else:
        print "Gagné!"
        break
```

On note que le code est considérablement plus court et que l'on retrouve bien nos variables **nombre** et **essai**, en tant que résultat de la fonction.
C'est parce que la fonction a renvoyé la saisie que l'on peut réaliser cette affectation : vous comprenez maintenant le sens de l'instruction **return** à la fin de la fonction.

Remarque

Vous trouverez cet exemple dans le répertoire Guide du dépôt. Il se nomme 12_Fonctions.py.

Nous avons donc écrit notre première fonction et le programme se comporte, de notre point de vue d'utilisateur, exactement de la même manière.

1.2.2 Gestion d'un paramètre

Nous pouvons assez aisément améliorer notre fonction. En effet, au lieu de faire des affichages avant d'appeler notre fonction, nous pouvons faire en sorte de changer l'invite lorsqu'on nous demande une saisie.
Pour cela, il faut donc passer un paramètre, c'est-à-dire que celui qui appelle la fonction doit lui donner les éléments pour qu'elle puisse agir conformément à son souhait.
Dans notre exemple, nous souhaitons également définir les valeurs minimale et maximale une fois pour toutes : nous allons utiliser des constantes.
Cette notion est très importante, puisqu'en utilisant cette constante plutôt qu'un littéral, nous nous donnons les moyens de changer cette valeur simplement : il suffit de modifier la constante plutôt que de parcourir tout le code à la recherche d'un littéral à modifier.
Ainsi, une constante se définit exactement comme une variable, sauf qu'elle est en majuscules :

```
MIN = 0
MAX = 99
```

Pour Python, une constante est une variable comme une autre. Seule la convention qui consiste à les mettre en majuscules en fait des constantes, mais vous pouvez toujours les modifier, rien ne vous en empêche.

Remarque

Python fonctionne énormément avec des conventions : il vous donne les outils pour faire les choses correctement, mais il ne vous contraint pas. Python part du principe que le développeur sait ce qu'il fait et il vous fait entièrement confiance pour faire la bonne chose : si pour une raison ou une autre vous ne respectez pas la convention, c'est que vous avez une bonne raison de ne pas le faire et Python respecte cela.

L'utilisation de ces constantes augmente aussi la lisibilité et la compréhension du code, puisque, lors de la déclaration, on comprend tout à fait de quoi il s'agit et que si on rencontre **MIN** ou **MAX** plus loin dans le code, on comprendra ce à quoi cela réfère, plus simplement que s'il s'agissait de littéraux.
Voici pour commencer la fonction légèrement retravaillée :

```
def demander_saisie_nombre(invite):
    # Compléter l'invite:
    invite += " entre " + str(MIN) + " et " + str(MAX) + ": "

    while True:
        saisie = input(invite)
        try:
            saisie = int(saisie)
        except:
            pass
        else:
            if MIN <= saisie <= MAX:
                break
    return saisie
```

On se donne donc la possibilité d'appeler notre fonction en disant ce qu'il faut saisir, et on complète cette information en précisant les bornes du nombre à saisir.
Notez que les constantes **MIN** et **MAX** sont définies en dehors de la fonction. Pour autant, elles sont accessibles. C'est aussi le cas de toutes les variables qui sont définies, au moment où la fonction est appelée.

Remarque

Sauf cas exceptionnel et que l'on est certain de maîtriser, on évitera d'utiliser dans une fonction une variable qui pourrait ne pas être définie au moment où la fonction est appelée.

Si on y réfléchit bien, les fonctions **int** et **input** sont également définies en dehors de la fonction, et si on avait importé un module au début du fichier, il serait aussi accessible depuis l'intérieur de la fonction.
Si on veut aller plus loin sur ces questions, il faudra se reporter au chapitre Déclarations - section Visibilité, traitant de la **portée d'une variable**.
Voici le code qui utilise cette fonction :

```
# PARTIE 1
nombre = demander_saisie_nombre("Saisissez le nombre à deviner")

# PARTIE 2
while True:
    essai = demander_saisie_nombre("Devinez le nombre")
    if essai < nombre:
        print("Trop petit")
    elif essai > nombre:
        print("Trop grand")
```

```
    else:
        print("Gagné!")
        break
```

Le code ainsi écrit est beaucoup plus lisible : on sait tout de suite pourquoi on utilise notre fonction et ce qu'elle va produire.

Remarque

Vous trouverez cet exemple dans le répertoire Guide du dépôt. Il se nomme 13_Fonctions_génériques_1.py.

Nous allons voir maintenant comment cette fonction peut être utilisée encore plus intelligemment.

1.2.3 Comment rendre une fonction plus générique

La fonction telle que nous l'avons écrite dépend de **MIN** et de **MAX**. Si nous souhaitons que ces deux valeurs puissent varier, il nous faut ne plus utiliser des constantes. Mais la fonction elle-même ne sait pas de quelle manière ces deux valeurs peuvent varier.
Ces valeurs doivent donc devenir des paramètres :

```
def demander_saisie_nombre(invite, minimum, maximum):
    invite += " entre " + str(minimum) + " et " +
              str(maximum) + " : "

    while True:
        saisie = input(invite)
        try:
            saisie = int(saisie)
        except:
            pass
        else:
            if minimum <= saisie <= maximum:
                break
    return saisie
```

On voit donc apparaître deux nouveaux paramètres, **minimum** et **maximum** et, par rapport à l'exemple précédent, on a remplacé **MIN** par **minimum** et **MAX** par **maximum**, tout simplement.

Astuce

Notez au passage la continuation de ligne entre les lignes 2 et 3 : comme la ligne 2 se termine par un +, Python sait que la ligne 3 est la suite de l'instruction débutée ligne 2. Par convention, comme cette instruction est une affectation, on aligne la ligne 3 sur le début de l'opérande de droite.