CHAPITRE 1

Python

> *La théorie, c'est quand on sait tout et que rien ne fonctionne. La pratique, c'est quand tout fonctionne et qu'on ne sait pas pourquoi. Ici, nous avons réuni théorie et pratique : rien ne fonctionne... et on ne sait pas pourquoi !*
>
> ALBERT EINSTEIN

Python est un langage de programmation open source créé en 1990 par Guido van Rossum, alliant simplicité et efficacité. Il n'a cessé de gagner en popularité au cours des dernières années jusqu'à devenir le langage de programmation le plus utilisé.

Ayant pour objectif d'être intuitif et facilement compréhensible sans perdre en efficacité de calcul, cela en fait un langage idéal pour démarrer l'apprentissage de la programmation.

Son fonctionnement open source a permis de développer des bibliothèques de fonctions puissantes adaptées à différentes problématiques, on peut citer notamment `numpy` pour la gestion des matrices, `pandas` pour le traitement de données statistiques ou encore `scikit-learn` très utilisé en machine learning. Python est notamment le langage de prédilection dans toutes les applications de Big Data ou de Machine learning, domaines de l'informatique en plein développement et très présents dans les sciences économiques.

1 Premiers pas

L'exécution d'un programme Python se fait à l'aide d'un *interpréteur*. Il s'agit d'un programme qui va traduire les instructions écrites en Python en *langage machine*, afin qu'elles puissent être exécutées directement par l'ordinateur. On dit donc que Python est un langage *interprété*.
Il y a deux modes d'utilisation de Python : le mode interactif, aussi appelé mode console et le mode script.

- **Mode interactif - Console**

La façon la plus simple d'exécuter une instruction Python est de la rentrer directement dans la console, l'interpréteur vous permet d'encoder les instructions une à une. Aussitôt une instruction encodée, il suffit d'appuyer sur la touche ENTER pour que l'interpréteur l'exécute.

```
Python 3.8.2 (default, Dec 25 2020 21:20:57)
Type "help", "copyright", "credits" or "license" for more information.
>>> 4*3
12
>>>
```

L'écran affiche une invite de commande, la plupart du temps de la forme >>>. Le mode interactif est donc un mode où l'on exécute des commandes une à une.

À l'instar d'une calculatrice, le mode interactif permet notamment de faire des calculs simples.

Console

```
>>> print("Hello World")
Hello World

>>> 5+2
7
```

On peut toutefois effectuer plusieurs commandes en les séparant d'un ; mais ceci s'avère très rapidement limité. Dès que l'on veut écrire un programme de plusieurs lignes, on préféra le mode script.

- **Mode script - Éditeur**

Lorsqu'on veut créer un programme qui va effectuer toute une suite de commandes, on peut les écrire les unes à la suite des autres dans un fichier `.py` puis envoyer ce fichier à Python qui va exécuter de lui même toutes les lignes les unes après les autres.

- **Installation de Python**

Pour commencer avec Python, nous vous conseillons d'utiliser « Basthon », une interface en ligne open source affichant une fenêtre script et une fenêtre console et qui permet d'exécuter du code Python sans aucune installation préalable.

https ://console.basthon.fr/

Une fois plus avancé, notamment lorsque vous voudrez travailler avec des bibliothèques qui requièrent une communication avec des fichiers présents sur votre ordinateur (cf partie 3.), vous pourrez installer directement Python sur votre ordinateur.

Nous vous conseillons pour ce faire d'installer directement une distribution complète avec les modules scientifiques déjà présents. On peut citer *Pyzo*, *Anaconda* ou *Visual Basic Studio* qui sont trois distributions Python très utilisées.

2 Bases du language Python

2.1 Types de données

Un programme Python manipule des données de différentes sortes, nous allons passer en revue les plus courantes.

- **Données numériques**

→ `int` : Nombres entiers.

→ `float` : Nombres réels.

- **Données littérales.**

→ `bool` : Les booléens sont `True` et `False`, ils permettent de tester si une instruction est vraie ou fausse. On détaillera leur fonctionnement dans la partie 2.5 (page 6).

→ `list` : Liste de données, ces données peuvent-être de n'importe quel type. On détaillera leur fonctionnement dans la partie 2.6 (page 6).

→ `str` : Chaîne de caractères, par exemple un mot ou une phrase. Les chaînes de caractères en Python ont la particularité de fonctionner comme des listes de caractères uniques.

→ `numpy.ndarray` : Tableau `numpy`. Assez similaire aux listes mais avec des méthodes puissantes supplémentaires de calcul numérique. On détaillera leur fonctionnement dans la partie 3.1.

On peut obtenir le type associé à une variable à l'aide de l'instruction `type()`. Par exemple :

Console

```
>>> type(4)
<class 'int'>
```

```
>>> type(4.1)
<class 'float'>
```

```
>>> type("Hello World")
<class 'str'>
```

Il existe bien d'autres types tels que les *dictionnaires* ou les *tuples*, on peut même en Python créer ses propres types. Nous utiliserons dans le cours et les exercices associés à ce livre seulement les types susmentionnés.

Enfin dans certains cas, il est nécessaire de changer le type d'une variable. Ci-contre, on transforme la chaîne de caractères en liste à cinq éléments.

Console

```
>>>list("Hello")
['H', 'e', 'l', 'l', 'o']
```

2.2 Affectation d'une variable

Au cours de l'exécution d'un programme, il peut être utile de garder en mémoire des résultats pour les réutiliser ensuite. Pour ce faire, Python permet de stocker de l'information dans des variables. On *affecte* une information dans une variable à l'aide du symbole =.

Exemple. Le code ci-contre affecte la valeur 2 à la variable `a`.
On peut ensuite travailler avec la variable directement.

Console

```
>>> a = 2
>>> 5*a
10
```

On peut bien sûr stocker dans une variable bien d'autres choses qu'une valeur numérique. Par exemple, on peut stocker une chaîne de caractères ou encore une liste.

Console

```
>>> b = "lorem ipsum"

>>> c = [1,2,3]
```

Exercice 1

✧ À quoi sert la suite d'instructions ci-contre ?

```
>>> a=2 ; b=3
>>> c=a ; a=b ; b=c
>>> print(a,b)
```

Remarque. Un type d'affectation très couramment rencontré est l'affectation composée. Lorsqu'on doit mettre à jour un variable par rapport à sa valeur précédente, on peut avec Python utiliser la valeur précédente de la variable pour définir la nouvelle. Pour augmenter une variable `i` d'une unité par exemple, deux syntaxes sont possibles :

Console

```
i = i + 1
# ou
i += 1
```

2.3 Fonctions prédéfinies avec Python

Il existe de nombreuses fonctions prédéfinies dans Python qui la plupart du temps suffisent pour écrire des programmes simples. Nous avons déjà rencontré `type()` qui prend en argument une variable et renvoie son type. Nous

allons décrire ici quelques fonctions utiles de Python.

- **Affichage avec print()**

La fonction `print()` permet d'afficher dans la console ce qu'on lui donne entre parenthèse.

On peut très bien effectuer les calculs à l'intérieur de la fonction `print()`, Python affichera le résultat.

Console
```
>>> a = 12
>>> print(a)
12
>>> print(2*a)
24
```

On peut aussi afficher plusieurs éléments à la suite les uns des autres dont des variables, pour ce faire, on sépare les éléments par des virgules.

Editeur
```
Prenom = "François"
Nom = "Damiens"

print("Mon nom est ", Nom," et mon prénom ", Prenom)
```

Python va afficher les quatre arguments de la fonction les uns à la suite des autres. Les arguments 1 et 3 sont des chaînes de caractères que Python affiche telles quelles et les arguments 2 et 4 sont des variables dont Python va afficher le contenu. Ce script affichera donc :

```
Mon nom est Damiens et mon prénom François
```

Une autre façon d'inclure des variables dans un texte imprimé et d'utiliser l'écriture formatée grace à l'instruction `format`. Python va alors remplir dans l'ordre donné les différentes `{}` par les variables associées.

Editeur
```
Prenom = "François"
Nom = "Damiens"

print("Mon nom est {} et mon prénom {}".format(Nom,Prenom))
```

- **input()**

Cette fonction permet d'interagir avec la personne utilisant le programme. Le programme va se mettre en pause le temps que la personne écrive une chaîne de caractères. Une fois la touche ENTER pressée la chaîne de caractères est envoyée à Python et peut être stockée dans une variable.

Exemple. Ce script convertit la chaîne de caractères reçue en un entier et en affiche son carré.

Editeur
```
x = int(input("Donnez un entier"))
print(x**2)
```

- **Les listes avec range()**

La fonction `range()` crée une liste contenant des **entiers**. Il existe trois façons d'appeler `range()`.

→ `range(stop)` renvoie la liste de nombres entiers commençant à `0` et progressant de un en un pour s'arrêter à `stop` **exclu**.

→ `range(start, stop)` fait de même mais commence la liste à `start` et non plus `0`.

Editeur
```
range(n+1)    # Liste des entiers de 0 à n.
range(1,n+1) # Liste des entiers de 1 à n.
```

→ `range(start, stop, step)` fait de même mais ne progresse plus de un en un mais d'une quantité définie par la variable `step`.

Editeur
```
range(0,101,10) # Liste des dizaines de 0 à 100
```

Exercice 2

✧ Comment obtenir la liste des nombres pairs de 0 à 10?

2.4 Importation de bibliothèques

La puissance de Python tient au fait qu'il est possible d'importer des fonctions optimisées pour certaines tâches. Nous verrons en détails dans les parties suivantes des modules usuels qui contiennent chacun de nombreuses fonctions déjà codées et qu'il est courant d'importer dans Python dans le cas de son utilisation pour résoudre des problèmes mathématiques.

Nous allons voir ici la syntaxe à respecter pour pouvoir importer une bibliothèque ou seulement certaines de ses fonctions dans Python. Nous utiliserons pour l'exemple la bibliothèque `math` qui contient les fonctions exponentielle (`exp`), logarithme (`log`) ... ou les constantes classiques telles que π (`pi`).

La commande pour importer une bibliothèque se place en général en entête du programme pour des questions de lisibilité. Deux syntaxes sont possibles.

Editeur
```
import math
```

ou

Editeur
```
import math as m
```

La deuxième méthode importe la bibliothèque `math` mais lui donne un *alias*, par conséquent pour utiliser une fonction ou grandeur issue de `math`, il faudra lui accoler le préfixe `m`.

Exemple. On accédera directement à π par la commande `m.pi`.

Console
```
>>> m.pi
3.141592653589793
```

Remarques.

• Cette méthode semble plus fastidieuse mais permet de garder une trace dans le programme de l'origine des différentes fonctions, ce qui est conseillé pour gagner en lisibilité du code.

• Importer une bibliothèque complète augmente le temps d'exécution d'un programme, on peut donc décider de n'importer que certaines fonctions de la bibliothèque qui nous sont nécessaires. Par exemple, pour importer π et la fonction exponentielle, on peut exécuter :

Editeur
```
from math import pi,exp
```

Avec cette méthode, on peut à nouveau importer toutes les fonctions d'une bibliothèque avec

Editeur
```
from math import *
```

2.5 Les booléens et la logique en Python

Les booléens apparaissent régulièrement en Python, en effet il est courant (voir partie 2.7, page 9) de devoir évaluer si une égalité ou inégalité est vérifiée ou non.
Il existe pour ce faire plusieurs opérateurs de comparaison :

Égal	Différent	Inférieur strictement	Supérieur strictement	Inférieur ou égal	Supérieur ou égal
==	!=	<	>	<=	>=

⚠ **Attention.** Il ne faut pas confondre = (qui sert à affecter une valeur à une variable) et == (qui permet de tester une égalité).

Lorsque ces opérateurs de comparaison sont utilisés, Python renvoie un booléen, c'est-à-dire `True` ou `False` si la condition est vraie ou fausse.

Console
```
>>> 2 < 6
True
```
```
>>> 2 != 2
False
```
```
>>>(3**2+4**2)==5**2
True
```

On peut aussi utiliser des opérateurs logiques avec Python.

Et	Ou	Non
and	or	not

Console
```
>>> not(True)
False

>>> (2 < 1) or (2 > 1)
True

>>> not((2 != 3) and (2<=3))
False
```

Exercice 3

✧ Que renvoie les commandes suivantes ?

```
>>> x=5 ; y=2
>>> (x==y) or (x>=y)
```

2.6 Les listes

Les listes sont des éléments centraux de Python, nous les utiliserons dans presque chaque code. Les listes sont définies entre crochet et contiennent différents éléments, on parle de liste homogène lorsque ses éléments sont du

même *type* et de liste hétérogène sinon.

→ On accède au nombre d'éléments d'une liste à l'aide de la fonction `len()`.

→ On accède à un élément de la liste en précisant sa position dans la liste entre crochets.

⚠ **Attention.** Les indices commencent à 0!

Remarque. Il est parfois intéressant d'accéder au dernier élément d'une liste, Python permet de rendre cela très simple avec des indices négatifs, l'élément d'indice -1 est le dernier élément de la liste, -2 l'avant dernier et ainsi de suite.

Console

```
>>> A = [4,2,28,0,1,96]
>>> len(A)
5

>>> A[0]
4

>>> A[3]
0

>>> A[-2]
1
```

- **Les commandes in et count**

Pour savoir si un élément appartient à une liste, on emploie `in`.

Console

```
>>> B = ["Franck","Adrien","Pierre","Sophie"]
>>> "Sophie" in B
True
```

Si un élément appartient à une liste, on peut compter le nombre d'occurrences de cet élément dans la liste avec l'instruction `count` :

Console

```
>>> C = [1,1,0,2,1,0,5,7]
>>> C.count(1)
3              # Il y a bien 3 fois l'élément 1 dans C
```

- **Le « slicing »**

Il est possible de créer une sous-liste à partir d'une liste donnée, pour ce faire on utilise la méthode du *slicing* (découpage) à l'aide de la ponctuation « : » .

Console

```
>>> C[2:4]
[0, 2]
```

On remarque qu'avec la commande `C[i:j]`, on récupère la liste des éléments de `C` d'indices `i` à `j` **exclu**!
Le sciling permet aussi de récupérer tous les éléments d'une liste `A` à partir d'un indice `i` avec `A[i:]`, ou tous les éléments de cette liste jusqu'à l'indice `j` **exclu** avec `A[:j]` :

Console

```
>>> A = [4,2,28,0,1,96]
>>> A[:4]                                    >>> A[3:]
[4,2,28,0]                                   [0,1,96]
```

- **Insertion d'éléments dans une liste**

Les listes sont des objets mutables, c'est-à-dire que leurs tailles peuvent varier, on peut ajouter ou supprimer des éléments.

Pour ajouter un élément à la fin d'une liste, on utilise `Liste.append(element)`.

Console
```
>>> A.append(35)
>>> A
[4,2,28,0,1,96,35]
```

Lorsqu'on veut insérer un élément dans une liste, il faut préciser l'endroit où l'insérer.

Console
```
>>> A[2:2] = [50]
>>> A
[4,2,28,50,0,1,96,35]
```

Remarque. On remarque que l'élément à ajouter est donné sous forme d'une liste à un élément, on peut donc ajouter plusieurs éléments en donnant non plus une liste à un élément mais une liste à plusieurs éléments.

⚠ **Attention.** Il ne faut pas confondre avec `A[2]=[50]` qui **remplace** le deuxième élément par `[50]`.

- **Suppression d'éléments dans une liste**

Pour supprimer un élément, trois méthodes sont couramment utilisées.

→ La fonction `del()` permet de supprimer l'élément donné en argument.

Console
```
>>> A = [4,2,28,0,1,96]
>>> del(A[1])  # suppression de l'élément d'indice 1, c'est-à-dire le "2"
>>> A
[4,28,50,0,1,96,35]
```

→ Une autre fonction utile pour supprimer un élément est .pop(). Cette méthode a pour avantage de renvoyer l'élément supprimé, on peut donc l'utiliser ensuite dans le programme.

Console
```
>>> A.pop(2)
50
>>> print("Le dernier élément de A était : ", A.pop(-1))
Le dernier élément de A était : 35
>>> A
[4,28,0,1,96]
```

→ Enfin si on connaît exactement la valeur de l'élément à supprimer on peut utiliser la fonction `remove()` :

Console
```
>>> A
[4,28,0,1,96]
>>> A.remove(96)
>>> A
[4,28,0,1]
```

- **La concaténation de listes**

On peut utiliser les opérateurs + et * pour fusionner des listes ou pour en créer de nouvelles, on appelle cette méthode la concaténation.

Cette possibilité est parfois utilisée pour créer une liste contenant seulement des 0 ou seulement des 1 de taille voulue.

Console
```
>>> [1,2,3]+[4,5]
[1, 2, 3, 4, 5]

>>> [0,1]*3
[0,1,0,1,0,1]

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

- **Copie de liste**

Une erreur récurrente dans l'utilisation des listes apparaît lorsqu'on doit copier une liste. En effet, si A est une liste, l'opération `B=A` crée une copie exacte de `A`, tout modification de `B` modifie `A` simultanément et toute modification de `A` modifie `B` de la même façon.
Pour copier les éléments d'une liste sans créer de lien entre la copie et l'originale, il faut utiliser la fonction `liste.copy()`

Editeur
```
A = [1,2,3,4]
B = A
C = A.copy()
A[0] = 8
print(A)
print(B)
print(C)
```

Ce script affiche :

Console
```
>>>
[8,2,3,4]
[8,2,3,4]
[1,2,3,4]
```

2.7 Instructions conditionnelles

À l'aide des opérateurs logiques vus précédemment, nous pouvons créer des instructions conditionnelles, c'est-à-dire des parties de code ne s'exécutant que si une condition logique est vérifiée. Pour ce faire, nous utilisons la commande `if`.

Editeur
```
if n < 1 :
    print(n)
```

Cette commande affichera n si et seulement si $n < 1$. Dans le cas où n est plus grand que 1, on peut vouloir tout de même effectuer une action, on peut dans ce cas utiliser la commande `else`.

Editeur
```
if n < 1 :
    print(n)
else :
    print("n est plus grand que 1")
```

⚠ **Attention.** On remarquera la syntaxe particulière : le « : » à la fin des lignes `if` et `else` ainsi que **l'indentation** de l'instruction qui dépend de la condition. Ces deux règles de syntaxe sont **obligatoires**.

Enfin, dans le cas où plusieurs conditions sont juxtaposées, on utilise la commande `elif`.

Editeur
```
if note>=16:
    print("Mention TB")
elif note>=14:
    print("Mention Bien")
elif note>=12:
    print("Mention AB")
elif note>=10:
    print("Mention Passable")
else:
    print("Oups!")
```

On lit `elif` en français comme « sinon si », son instruction n'est prise en compte que si la précédente n'est pas vérifiée.

2.8 Instructions répétitives - boucles

Il existe deux méthodes pour répéter une instruction, l'utilisation de `while` ou de `for`. Les deux amènent à ce qu'on appelle des *boucles* dans le code. Une partie du code se répète tant qu'une condition terminale n'est pas atteinte.

- **Boucles for**

Les boucles `for` utilisent un itérateur qui va prendre diverses valeurs et pour chaque valeur de cet itérateur la boucle va s'exécuter.

Editeur
```
for i in [0,1,2,3] :
    print(i)
```

Console
```
0
1
2
3
```

On utilise en général la fonction `range()` pour créer la liste des valeurs que va prendre `i`. Le script précédent se réécrit alors.

Editeur
```
for i in range(4) :
    print(i)
```

⚠ **Attention.** Comme précisé précédemment, `range(4)` va de 0 à 3, le 4 est exclu de la liste.

L'itérateur n'est pas forcément un nombre, il peut lui même prendre les expressions des différents éléments d'une liste donnée à l'instruction `for` :

Editeur
```
c = ["Marc", "est", "dans", "le", "jardin"]
for i in c:
    print("i vaut", i)
```

renvoie :

Console
```
i vaut Marc
i vaut est
i vaut dans
i vaut le
i vaut jardin
```

Exercice 4

✦ Écrire une liste d'instruction qui à partir d'une liste A donnée affiche la liste « miroir » de A, c'est-à-dire la liste qui a les mêmes éléments que A mais dans l'ordre inverse.

- **Listes par compréhension**

L'instruction `for` mise entre crochets permet aussi de définir des listes.

Console
```
>>> A = [2*i for i in range(101)]
```

Cette instruction crée une liste nommée `A` contenant tous les nombres pairs de 0 à 100.

- **Boucles while**

Ces boucles ne sont utilisées que lorsqu'on ne connaît pas à l'avance le nombre de répétitions, dans ce cas l'instruction va se répéter **tant qu**'une condition donnée à l'opérateur `while` reste vérifiée. La condition est donc en général une expression booléenne.

Editeur
```
x = 1     # initialisation
while x < 10:
    print("x a pour valeur", x)
    x = x * 2
print("Fin")
# Attention à l'indentation
```

Console
```
x a pour valeur 1
x a pour valeur 2
x a pour valeur 4
x a pour valeur 8
Fin
```

Les boucles `while` sont à manier avec prudence car leur exécution peut amener, si l'expression logique de terminaison reste tout le temps vraie, à des boucles infinies qui vont saturer la mémoire de l'ordinateur.

L'exemple ci-contre imprime `1` en permanence jusqu'à ce que la mémoire sature.
Une façon d'arrêter une boucle `while` est d'utiliser l'instruction `break`.

Editeur
```
while True
  print("1")
```

Editeur
```
n = 1
while n <= 1000000:
    if n*n == n+n:
        break
    n += 1
print("le carré de {} est égal au double de {}".format(n,n))
```

Dans l'instruction précédente une fois que $n = 2$, la condition du `if` devient vraie donc `break` est utilisé et la boucle s'arrête, n n'augmente plus.

Exercice 5

✦ Que fait le code suivant?

```
x=float(input("Entrez un réel positif"))
n=0
while n<=x-1 :
    n+=1
print(n)
```

2.9 Fonctions

Jusqu'à maintenant dans ce chapitre nous avons utilisé des instructions simples ou des scripts courts mais la plupart des programmes Python sont écrits en terme de fonction. Nous avons déjà rencontré des fonctions prédéfinies dans Python telles que `print()` ou `range()`, ces instructions prennent éventuellement un ou plusieurs arguments et effectuent une action. Python permet de créer ses propres fonctions. La syntaxe générale est :

Editeur
```
def nom_de_la_fonction(argument1,argument2,...) :
  instructions à effectuer
```

- **Ma première fonction**

Écrivons une fonction qui affiche la table de multiplication de 3.

Editeur
```
def table3():
    n = 1
    while n <= 10:
        print(n, "x 3 =", n * 3)
        n += 1
```

Une fois le script exécuté... il ne se passe rien. Nous avons seulement défini la fonction `table3()`, pour l'utiliser il faut encore appeler cette fonction. On peut écrire dans la console :

Console
```
>>> table3()
```

Console
```
1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 = 12
5 x 3 = 15
6 x 3 = 18
7 x 3 = 21
8 x 3 = 24
9 x 3 = 27
10 x 3 = 30
```

- **Les arguments**

La plupart des fonctions prennent des arguments en entrée. Par exemple si l'on veut créer une fonction qui puisse calculer et afficher l'aire d'un carré, on peut définir la fonction :

Editeur
```
def carre(n) :    # n est la longueur d'un côté
    print("l'aire du carré est : ", n**2)
```

On l'appelle ensuite en donnant la valeur voulue à l'argument de la fonction :

Console
```
>>> carre(5)                              >>> carre(12)
l'aire du carré est : 25                  l'aire du carré est : 144
```

On voit d'emblée l'intérêt de la notion de fonction, il n'est pas besoin de relancer tout le script, il suffit d'appeler la fonction avec un autre argument pour obtenir le résultat voulu.
Une fonction peut prendre plusieurs arguments, pour calculer par exemple l'aire d'un rectangle on peut définir la fonction :

Editeur
```
def rectangle(longueur,largeur) :
  print("l'aire du rectangle est : ", longueur*largeur)
```

Exercice 6

✦✦ Il y a $+7h$ de décalage horaire entre le Japon et la France. Écrire une fonction qui prend l'heure en France au format d'une chaîne de caractères au format « $heure : min$ » (par exemple 13 : 52) et affiche l'heure au Japon.

- **return**

Une fonction a la capacité de renvoyer une valeur qu'on peut utiliser ultérieurement, contrairement à `print()` qui se contente d'afficher un résultat.

Par exemple afin de créer une fonction qui calcule le volume d'un parallélépipède rectangle, on peut commencer par créer une fonction qui calcule l'aire d'une de ses faces pour ensuite multiplier par la hauteur, pour ce faire on modifie la fonction précédente :

Editeur
```
def rectangle(longueur,largeur) :
  return longueur*largeur
```

La fonction `rectangle(longueur,largeur)` une fois appelée **vaut** `longueur×largeur` donc par exemple `rectangle(3,4)` est **égal** à 12, on peut donc utiliser le code suivant pour calculer le volume du parallélépipède :

Editeur
```
def volume(a,b,h) :
  return rectangle(a,b)*h
```

On voit que l'instruction `return` permet d'utiliser ultérieurement le résultat d'une fonction et donc aussi d'imbriquer les fonctions. Les exemples présentés ici sont très simples et l'imbrication de fonctions n'est pas la meilleure syntaxe à utiliser mais dans d'autres cas cette méthode peut s'avérer très utile, notamment pour clarifier le code.

Exercice 7

✦ 1. Écrire un programme qui prend en argument deux nombres et renvoie le maximum (le plus grand des deux).
2. En déduire deux nouvelles fonctions utilisant la fonction précédente.
   → Une fonction qui prend en argument deux nombres et renvoie le minimum.
   → Une fonction qui prend en argument trois nombres et renvoie le maximum des trois nombres.

On peut de plus renvoyer plusieurs éléments en sortie d'une fonction. Dans l'exemple suivant on renvoie le nom du vainqueur ainsi que la somme gagnée :

Editeur

```
joueurs = ["Elise","Jonathan","Celine","Silvère"]
prix = [200,100, 3550, 0]

def gagnant(indice_gagnant, indice_prix) :
  return joueurs[indice_gagnant],prix[indice_prix]
```

Console

```
>>> gagnant(3,2)
('Silvère', 3550)
```

En fonction de la console utilisée, Python n'affiche pas forcément ce que renvoie la fonction car `return` n'a pas vocation à afficher quoi que ce soit. Si l'on veut afficher les éléments renvoyés, on peut tout simplement utiliser `print()` :

Console

```
>>> print(gagnant(3,2))
('Silvère', 3550)
```

Enfin, on peut très bien stocker les retours d'une fonction dans une variable pour les utiliser plus tard :

Console

```
>>> nom, somme = gagnant(3,2)
>>> nom
'Silvère'
```

Exercice 8

✦ La fonction ci-contre prend un montant en Euro comme argument et renvoie la conversion en Dollar et en Yuan.
Créer un programme `Conversion2(d)` qui prend comme argument une somme en dollar et renvoie sa conversion en euro.

```
def Conversion(e) :
    taux_yuan=7.62
    taux_dollar=1.14
    y=taux_yuan*e
    d=taux_dollar*e
    return d,y
```

- **Variables locales et globales**

Dans l'exemple du script définissant la fonction `gagnant()` les listes `joueurs` et `prix` sont des variables *globales*, cela signifie qu'elles sont accessibles en dehors de la fonction en elle même :

Editeur

```
joueurs = ["Elise","Jonathan","Celine","Silvère"]
prix = [200,100, 3550, 0]

def gagnant_bis(indice_gagnant, indice_prix) :
  multilicateur = 2
  return joueurs[indice_gagnant],prix[indice_prix]*multiplicateur
```

Console

```
>>> joueurs
["Elise","Jonathan","Celine","Silvère"]

>>> multiplicateur
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'multiplicateur' is not defined
```

La variable `multiplicateur` est elle définie dans le fonction `gagnant_bis()` par conséquent elle n'est définie que dans cette fonction, c'est une variable *locale*. Elle n'est pas accessible autre part que dans la fonction. Pour accéder à une variable créée dans une fonction en dehors de la fonction, il faut que cette variable soit renvoyée avec `return`.

3 Bibliothèques fréquemment utilisées

3.1 numpy

- **Présentation**

La bibliothèque `numpy` doit d'abord être importée avant de pouvoir utiliser ses fonctions :

Editeur
```
import numpy as np
```

La bibliothèque `numpy` permet d'accéder à la plupart des fonctions et constantes mathématiques comme pour la bibliothèque `math`, on peut par exemple utiliser π avec la commande `np.pi` ou la fonction exponentielle avec `np.exp(.)`.

La bibliothèque `numpy` permet surtout de créer des tableaux de type `ndarray`, parfait pour représenter des matrices. Bien qu'ils aient certains points communs avec les listes, une caractéristique fondamentale des tableaux `numpy` est qu'ils sont des objets immutables, c'est-à-dire qu'une fois créés, leur taille est fixe. On comprend donc que les commandes `.append()` ou `.pop()` ne pourront pas s'appliquer aux tableaux `numpy`.
On peut créer simplement une matrice ligne à l'aide de l'instruction :

Console
```
>>> A = np.array([1,2,3,4])
```

Pour créer une matrice, il suffit de donner ses différentes lignes séparées par des virgules :

Console
```
>>> B = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> B
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Ainsi pour créer une matrice colonne, on peut écrire :

Console
```
>>> C = np.array([[1],[2],[3]])
>>> C
array([[1],
       [2],
       [3]])
```

Comme pour les listes on accède à un élément en donnant sa position :

Console
```
>>> A[0]          >>> B[0]          >>> B[0][1] # B[0,1] fonctionne aussi
1                 [1,2,3]           2
```

On peut extraire une ligne ou une colonne d'une matrice à l'aide du slicing :

Console

```
>>> B[1,:]                                    >>> B[:,0]
[4, 5, 6]                                     [1,4,7]
```

- **Création de vecteurs**

La fonction `arange(initial,final,pas)` crée un tableau de façon assez analogue à la fonction `range()`, à ceci près que les coefficients ne sont pas forcément entiers. Elle prend en entrée trois nombres, la valeur de départ, la valeur d'arrivée et le pas entre deux valeurs.

Console

```
>>> H = np.arange(0,0.9,0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8])
```

La fonction `linspace(initial,final,nombre_de_points)` crée un tableau comme `arange()` à la différence qu'on précise ici le nombre de valeurs entre les valeurs initiale et finale.

Console

```
>>> I = np.linspace(0,0.9,10)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

⚠ **Attention.** `np.arrange` n'inclut pas la valeur de fin (`0.9` ici) contrairement à `linspace`.

- **Matrices prédéfinies**

Il existe des matrices usuelles prédéfinies dans la bibliothèque `numpy`.

- `np.ones(n)` crée une matrice ligne de longueur n remplie 1.
- `np.ones((m,n))` crée une matrice de taille `(m,n)` remplie de 1.
- `np.zeros(n)` crée une matrice ligne de longueur n remplie 0.
- `np.zeros((m,n))` crée une matrice de taille `(m,n)` remplie de 0.
- `np.eye(n)` crée la matrice identité de taille `(n,n)`.

- **Action d'une fonction sur un tableau numpy**

La puissance des tableaux `numpy` est que lorsqu'on applique un tableau par une fonction, la fonction s'applique à tous les éléments du tableau et renvoie le résultat du calcul.

Console

```
>>> H*2  # multiplication de chaque terme de H par 2
array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6])

>>> K = np.linspace(0,1,10)
>>> np.exp(K)
array([1.         , 1.11751907, 1.24884887, 1.39561243, 1.5596235 ,
       1.742909   , 1.94773404, 2.17662993, 2.43242545, 2.71828183])
```

Remarque. Certaines méthodes sur les listes sont intéressantes, notamment la possibilité d'ajouter des éléments à une liste ce qui n'est pas possible avec un tableau `numpy`. Il est donc bien de savoir qu'on peut transformer un tableau `numpy` en liste avec `A = list(u)` et inversement une liste en tableau `numpy` par `u = np.array(A)`.

- **Opérations sur les matrices**

La plupart des opérations sur les tableaux `numpy` se font terme à terme, par exemple :

Editeur
```
A=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(A==2)    # On teste si chaque coefficient de la matrice vaut 2
```

Console
```
[[False  True False]
 [False False False]
 [False False False]]      # Un seul 2, en position (2,1)
```

⚠ **Attention.** Comme la somme matricielle est une opération terme à terme. Pour effectuer la somme entre deux matrices `A`, `B` de même taille, il suffit d'écrire `A+B`.
Par contre, la commande `A*B` renvoie le produit coefficient par coefficient et non le produit matriciel.

Exercice 9

✧ Comment obtenir la matrice carrée suivante avec Python ?

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

- **Fonctions usuelles**

→ `shape()` renvoie la taille d'un tableau `numpy`.

Console
```
>>> np.shape(np.array([[1,2,3],[4,5,6],[7,8,9]]))
(3, 3)
```

→ `transpose(A)` renvoie la transposée de la matrice A.

→ `np.dot(A,B)` effectue le produit matriciel AB.
Rappelons que le produit matriciel n'est pas commutatif, en général AB ≠ BA.

Exercice 10

✦ Prévoir la réponse de la machine

```
n=10
A=np.arange(1,n+1)
B=np.ones((n,1))
print(np.dot(A,B))
```

3.2 linalg

Pour les manipulations d'algèbre linéaire, la bibliothèque `numpy` possède une sous-bibliothèque nommée `linalg`.
On importe la bibliothèque `linalg` via :

Editeur
```
import numpy.linalg as al
```

Listons les méthodes les plus utilisées avec `numpy.linalg`.

- **Inverser une matrice : al.inv(A)**

Console
```
>>>  al.inv(np.array([[1,1,2],[1,2,1],[2,1,1]]))
array([[-0.25, -0.25,  0.75],
       [-0.25,  0.75, -0.25],
       [ 0.75, -0.25, -0.25]])
```

- **Obtenir le rang d'une matrice : al.matrix_rank(A)**

Console
```
>>> al.matrix_rank(np.array([[1,1,2],[1,2,1],[2,1,1]]))
3
```

- **Calculer les puissances d'une matrice carrée : al.matrix_power(A,n)**

Console
```
>>> A = np.array([[1,2],[0,2]])
>>> al.matrix_power(A,2)
array([[1, 6],
       [0, 4]])
```

Remarque. À noter que l'instruction `al.matrix_power(A,-1)` renvoie l'inverse d'une matrice si cette dernière est inversible.

- **Résoudre un système : al.solve(A,B)**

Si `A` est une matrice carrée inversible et `b` une matrice de même taille ou une matrice colonne ayant le même nombre de lignes que `A`, alors `al.solve(A,B)` renvoie la solution du système AX = B d'inconnue X.

Exemple. On peut résoudre le système

$$\begin{cases} 4x+2y &= 2 \\ x-5y &= -16 \end{cases}$$

d'inconnue $(x,y) \in \mathbb{R}^2$.
On obtient $x=-1$, $y=3$.

Console
```
>>> A = np.array([[4,2],[1,-5]])
>>> B = np.array([[2],[-16]])
>>> al.solve(A,B)
array([[-1.],
       [ 3.]])
```

Exercice 11

✦ Résoudre, à l'aide de Python, les deux systèmes linéaires suivants.

$$\mathscr{S}_1 : \begin{cases} x_1+9x_2+10x_3 &= -50 \\ 9x_1+5x_2+x_3 &= 180 \\ 5x_1+10x_2+9x_3 &= 40 \end{cases} \quad \text{et} \quad \mathscr{S}_2 : \begin{cases} x_1+9x_2+10x_3 &= -50 \\ 9x_1+5x_2+x_3 &= 180 \\ 5x_1+10x_2+9x_3 &= 41. \end{cases}$$

Que constatez vous?

- **Calculer le déterminant d'une matrice : al.det(A)**

Console

```
>>> A = np.array([[1, 2], [3, 4]])
>>> al.det(A)
-2.0
```

3.3 random

Pour simuler des variables aléatoires, on peut utiliser la bibliothèque `random` que l'on importe par :

Editeur

```
import numpy.random as rd
```

- **rd.random()**

La commande `rd.random()` renvoie un réel choisi au hasard dans l'intervalle [0;1[suivant une loi de probabilité uniforme sur [0;1[.
Pour simuler à l'aide de Python un événement de probabilité p, on peut écrire `rd.random() < p` renverra `True` avec une probabilité p.

- **rd.randint(debut,fin)**

De la même façon, on peut tirer au hasard (et uniformément) un entier plutôt qu'un réel. Dans ce cas, la commande à utiliser est `rd.randint(début,fin)` qui tire au hasard avec une probabilité uniforme un entier dans l'intervalle `[début,fin[`.

- **rd.geometric(p)**

La fonction `rd.geometric(p)` permet de simuler une variable aléatoire qui suit une loi géométrique de paramètre p. Pour rappel, si X renvoie le rang du premier succès dans une infinité d'expériences de Bernoulli mutuellement indépendantes, $X \hookrightarrow \mathscr{G}(p)$ où p est la probabilité d'un succès.

- **rd.binomial(n,p)**

De la même façon, on peut simuler une variable aléatoire qui suit une loi binomiale de paramètres n, p avec la commande `rd.binomial(n,p)`. Pour rappel, le nombre de succès obtenus lors d'une répétition de n expériences de Bernoulli mutuellement indépendantes de probabilité de succès p suit une loi binomiale de paramètres n, p.

- **rd.poisson(lambda)**

Enfin, `rd.poisson(lambda)` simule une variable qui suit une loi de Poisson de paramètre `lambda`.

Remarque. Toutes les méthodes précédentes peuvent aussi renvoyer une liste de valeurs tirées selon les différentes lois de probabilité plutôt qu'une seule valeur, pour faire cela, il suffit de donner comme argument supplémentaire à l'instruction la taille de la liste voulue. Par exemple, `rd.randint(1,100,200)` renvoie un tableau `numpy` contenant 200 entiers pris aléatoirement entre 1 et 100. De plus, la commande `rd.randint(1,100,[200,10])` renvoie une matrice de taille (200,10).

3.4 matplotib.pyplot

La bibliothèque `matplotlib.pyplot` permet de tracer des courbes et diagrammes hautement personnalisables. On commence par importer cette bibliothèque avec :

Editeur

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

- **Tracé d'une courbe**

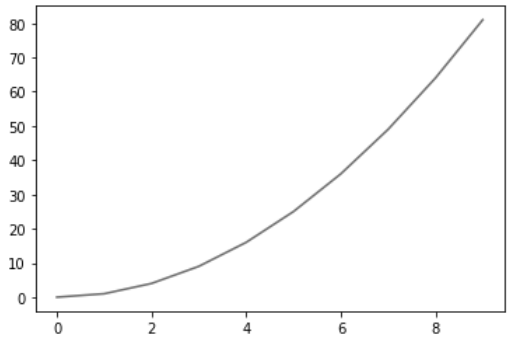
La commande `plt.plot(x,y)` permet de tracer une ligne brisée reliant les points du plan de coordonnées (x_i, y_i), il faut donc que x et y soient des listes ou de tableaux `numpy` de même taille.

Le code :

Console

```
>>> A = [0,1,2,3,4,5,6,7,8,9]
>>> B = [0,1,4,9,16,25,36,49,64,81]
>>> plt.plot(A,B)
```

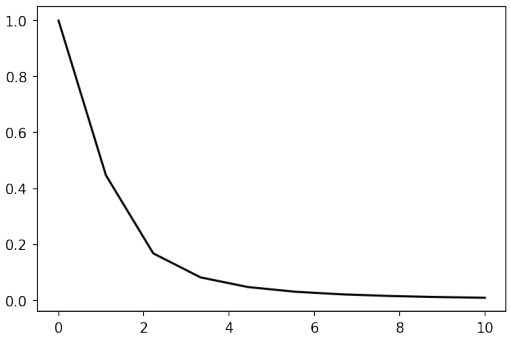
renvoie la courbe ci-contre.



Pour tracer la courbe représentative d'une fonction, il suffit de créer une liste `x` pour l'axe des abscisses avec la commande `np.linspace()` ou `np.arange()` puis de créer une nouvelle liste `y` pour l'axe des ordonnées en appliquant la fonction à chaque élément de `x`. Bien sûr, plus le nombre de points est grand, plus le tracé sera précis.
Par exemple, donnons la courbe de $x \in \mathbb{R} \mapsto \frac{1}{1+x^2}$ sur l'intervalle $[0;10]$.

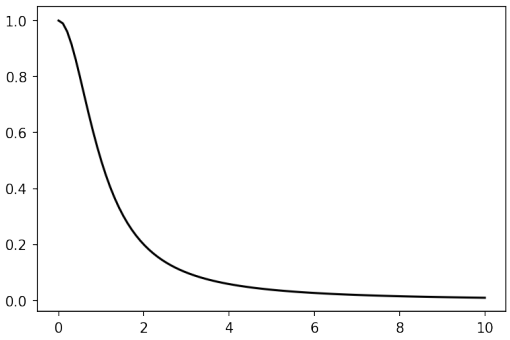
Editeur

```
A = np.linspace(0,10,10)
D = 1/(1+A**2)
plt.plot(A,D)
```



Editeur

```
A = np.linspace(0,10,100)
D = 1/(1+A**2)
plt.plot(A,D)
```



Remarque. On peut changer beaucoup de paramètres de la courbe en rajoutant des options dans l'instruction `plt.plot()`.

- `plt.plot(x,y,'g')` trace la courbe en vert.
- `plt.plot(x,y,'b.')` trace seulement les points (x_i, y_i) et en bleu.
- `plt.plot(x,y,'k:')` trace une ligne noire en pointillés.
- `plt.plot(x,y,label='nom de la courbe')` permet d'identifier la courbe par un encart dans le graphique, ceci est très utile pour distinguer plusieurs courbes tracées sur un même graphique. À noter qu'il faut rajouter la commande `plt.legend()` de façon à afficher les labels définis.

On peut aussi modifier les options du graphique lui-même :

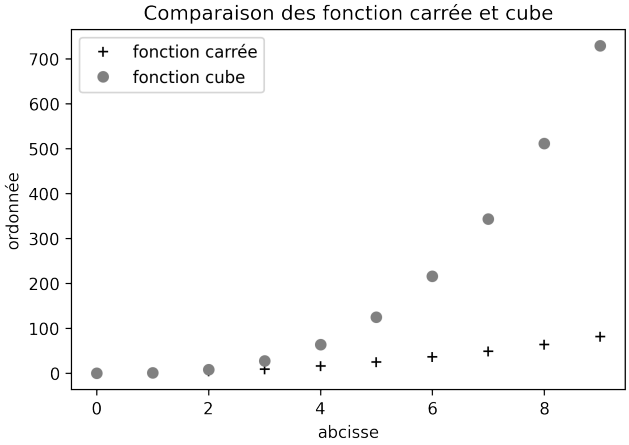
- `plt.xlabel('x')` fait apparaître un nom pour l'axe des abscisses. L'option `plt.ylabel('y')` fait de même pour l'axe des ordonnées.
- `plt.title('Titre')` fait apparaître un titre au graphique affiché.
- `plt.xlim(a,b)` restreint l'axe des abscisses entre les valeurs `a` et `b`. `plt.ylim(a,b)` fait la même chose sur l'axe des ordonnées.
- `plt.axis('equal')` rend les axes orthonormés.

Par exemple, regardons ce que renvoie la suite de commandes suivantes.

Editeur

```
A = np.arange(0,10)
B = A**2
C = A**3

plt.plot(A,B,'k+',label = 'fonction carrée')
plt.plot(A,C,'o',color = 'grey',label = 'fonction cube')
plt.legend()
plt.xlabel('abcisse')
plt.ylabel('ordonnée')
plt.title('Comparaison des fonctions carrée et cube')
```



Il existe bien sûr de nombreuses autres options que nous ne pouvons détailler ici. Le site officiel de `matplotlib` donne de nombreux exemples ainsi que les codes Python associés afin de pouvoir s'en inspirer.

Exercice 12

✦ Quelles courbes trace le code ci-contre?

```
x=np.linspace(-1,2,100)
y=np.exp(x)
plot(x,y)
plot(x,x)
plot(y,x)
```

Exercice 13

✦✦ **La courbe Blanc-manger** Pour tout réel x, on note $d(x)$ la distance de x avec le plus proche entier. On montre que $d(x) = \frac{1-\left|1-2x+2\lfloor x\rfloor\right|}{2}$.

1. Tracer la courbe de d sur $[-3;3]$.
2. Écrire un script qui prend en argument n et renvoie la courbe représentative de S_n définie par
$$S_n(x) = \sum_{k=0}^{n} \frac{d(2^k x)}{2^k}.$$
Commenter.
3. Justifier, pour tout $x \in \mathbb{R}$, la convergence de la suite $\left(S_n(x)\right)_{n\in\mathbb{N}}$.
4. Notons $S(x)$ la limite obtenue. Que dire de la régularité de la fonction S?

- **Tracé des termes d'une suite**

De même, on peut représenter les termes d'une suite.

Console

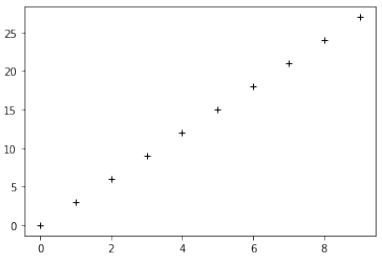
```
>>> ind = range(10) # On indique les indices
```

Ensuite, il faut calculer les différents termes de la suite, puis, les afficher en fonction des indices. Par exemple, pour $u = (3n)_{n\in\mathbb{N}}$:

Console

```
u = np.zeros(10)
for i in ind :
    u[i] = 3*i

plt.plot(ind,u,'k+')
```



4 Méthodes usuelles

4.1 Compteur

Il arrive régulièrement qu'on compte le nombre d'éléments d'un ensemble E vérifiant une certaine propriété $\mathscr{P}$. Pour créer un algorithme qui exécute cette tâche, on peut procéder de la manière suivante.

1. On introduit une variable `Compteur` qui va compter le nombre d'éléments vérifiant $\mathscr{P}$.
   Au début, `Compteur=0`.
2. On parcourt chaque élément de l'ensemble E à l'aide d'une boucle `for`.
3. Pour chaque élément, on teste (à l'aide d'une structure `if`), si l'élément vérifie la propriété ou non. Si la propriété est bien satisfaite, on augmente la valeur de `Compteur` de 1.
4. À la fin de la boucle, `Compteur` donne le nombre d'éléments vérifiant la propriété $\mathscr{P}$.

Exemple. Donnons un algorithme qui compte les nombres premiers inférieurs à 100.

Editeur

```
Compteur  = 0

for k in range(2,101)  :
# On ne traite ni 0 ni 1, en revanche on traite 100

    Diviseurs = 0
    for i in range(2, k//2+1):
        if (k % i) == 0:
        # i est un diviseur de k
          Diviseurs +=1
    if Diviseurs == 0 :
      Compteur +=1

print("Nombre premier inférieur à 100 :",Compteur)
```

Une fois ce code exécuté, Python renvoie la réponse : 25.

Remarque. Cet exemple comporte en réalité deux compteurs, un compteur local du nombre de diviseurs d'un nombre et le compteur de nombres premiers.

4.2 Algorithme de dichotomie

4.3 Algorithme de seuil et approximation de limite

- **Algorithme de seuil**

Ce type d'algorithme intervient lorsqu'on cherche le plus petit entier n tel que la propriété $\mathscr{P}(n)$ soit vraie. Il est très courant de le rencontrer dans l'étude des suites.

Exemple. On montre que la suite u, définie par la formule de récurrence suivante converge vers 2.

$$\forall\, n \in \mathbb{N}, \quad u_{n+1} = \frac{2}{(n+2)(n+1)} + u_n \quad \text{et} \quad u_0 = 0.$$

On cherche l'indice n tel que $u_n - 2$ devient inférieur à un seuil donné. Déterminons par exemple, le plus petit entier n pour lequel $|u_n - 2| \leqslant 10^{-3}$.

Editeur

```
u=0  # Initialisation
n=0
while (abs(u-2) > 0.001) :
    u=u+2/((n+1)*(n+2))
    # Calcul du terme suivant par la formule de récurrence
    n=n+1

print("La plus petite valeur est : ",n)
```

Ce code affiche la valeur 1999.

Exercice 14

✦ On admet que la suite de terme général $u_n = \sum_{k=1}^{n} \frac{1}{k}$ tend vers $+\infty$.

Écrire un programme qui prend en argument un réel A et renvoie le plus petit entier n tel que $u_n \geqslant$ A.

- **Approximation d'une limite à une précision donnée**

Soit u une suite convergente vers une limite finie ℓ. On souhaite obtenir une approximation de la limite.
On a $|u_n - \ell| \underset{n\to\infty}{\longrightarrow} 0$. En définissant une précision voulue, on peut donc déterminer une approximation de ℓ à cette précision près, on cherche donc un entier n tel que $|u_n - \ell| <$ précision.

Exemple. On considère la suite u définie par le terme général

$$\forall\, n \in \mathbb{N}^*, \quad u_n = \sum_{k=1}^{n} \frac{1}{k^2}.$$

On admet que la suite est bien définie et qu'elle converge vers une limite finie ℓ. Le programme suivant donne une approximation de ℓ à 10^{-3}-près.

Editeur

```
def limite(precision) :
    s=1 # initialisation somme
    n=1 # initialisation indice
    erreur=1 # initialisation écart à la limite
    while erreur > precision :
        n+=1
        s+=1/(n**2)
        erreur=1/n
    return s
```

Cette fonction exécutée avec la précision `0.001` donne `1.6439345666815615`. Testons ce résultat sachant que la limite de la suite u est $\pi^2/6$.

Console

```
>>> m.pi**2/6
1.6449340668482264
```

Remarque. Dans ce cas particulier assez simple, on montre que

$$\forall\, n \in \mathbb{N}^*, \qquad |u_n - \ell| \leqslant \frac{1}{n}.$$

D'où $$n \geqslant 10^3 \quad \Rightarrow \quad |u_n - \ell| \leqslant \frac{1}{n} \leqslant 10^{-3}.$$

u_{1000} est une approximation de ℓ à 10^{-3}.
Il n'est pas toujours possible d'inverser la relation comme le montre l'exercice suivant.

Exercice 15

✦✦ **Approximation d'une limite**
On montre que pour tout $n \in \mathbb{N}*$,

$$\left| \mathrm{e} - \sum_{k=0}^{n} \frac{1}{k!} \right| \leqslant \frac{1}{n \cdot 2^n}.$$

Écrire une fonction qui prend en argument un réel strictement positif ε et renvoie une approximation de la limite e = exp(1) à une précision ε-près.