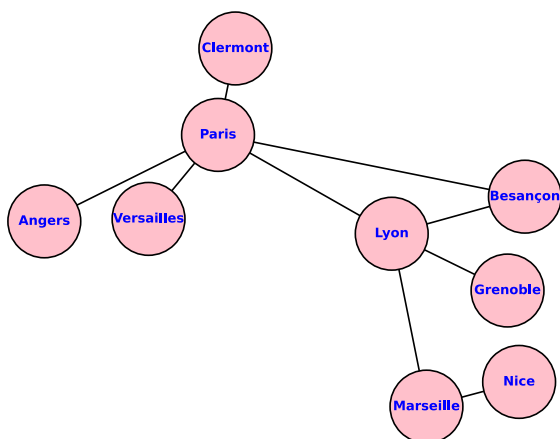
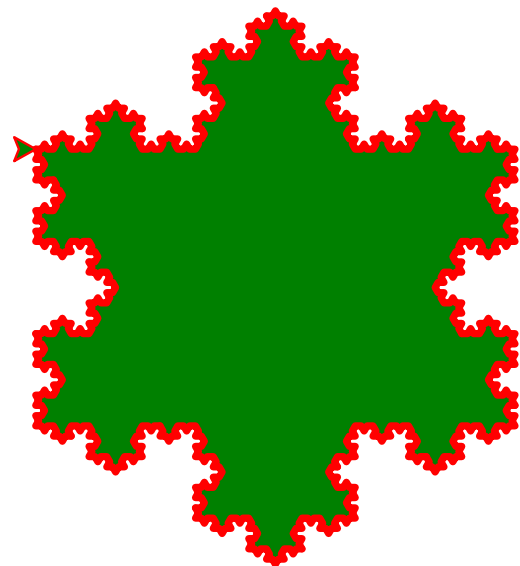


TP Informatique

BCPST1B 2025–2026

L.-C. LEFÈVRE
Lycée Hoche, Versailles



À propos

Ce document contient l'intégralité des TP d'informatique pour l'année scolaire 2025–2026 de la BCPST1B du Lycée Hoche de Versailles. Il comporte **23 TP** et **228 exercices**.

Il a été rédigé entièrement avec le langage **Typst**. Il s'agit d'un tout récent langage de mise en forme de documents ayant pour ambition d'être une alternative moderne à LaTeX. Encore en plein développement, il fournit déjà de très nombreux outils pour la publication scientifique (mathématiques, bibliographie, coloration syntaxique du code, importation de données CSV, ...) supportés nativement avec une grande facilité d'utilisation. Il a, dès sa publication, fédéré de nombreux contributeurs et utilisateurs. Le but de ce document est donc aussi d'expérimenter ce nouveau langage et d'en démontrer les possibilités.

En particulier, la police de caractères s'appelle **New Computer Modern**. Le thème de coloration syntaxique s'appelle **Pastie**. L'intégralité du document rédigé avec l'éditeur de texte **GNU Emacs** pour lequel il existe un mode Typst, avec un clavier en configuration **BÉPO**.

Table des matières

Introduction	4
TP 1 Prise en main	5
TP 2 Conditions et boucles	13
TP 3 Fonctions	18
TP 4 Boucles for	25
TP 5 Listes	31
TP 6 Algorithmes sur les listes	40
TP 7 Révisions et consolidation 1	44
TP 8 Tri	47
TP 9 Texte et mots	52
TP 10 Récursivité	58
TP 11 Dichotomie	63
TP 12 Algorithmes récursifs	66
TP 13 Révisions et consolidation 2	71
TP 14 Numpy et Matplotlib	75
TP 15 Matrices	85
TP 16 Manipulation d'images	90
TP 17 Traitement de données en tables	96
TP 18 Dictionnaires	100
TP 19 Graphes	106
TP 20 Parcours de graphes	114
TP 21 Aléatoire	119
TP 22 Langage SQL partie 1	126
TP 23 Langage SQL partie 2	132

Introduction

Le langage Python est un langage de programmation qui permet de donner des instructions à l'ordinateur, écrites de façon lisible par un humain. Par rapport aux nombreux (des milliers !) langages de programmation existants, il a les avantages suivants :

- **Moderne** : utilisé de plus en plus dans de nombreux milieux scientifiques, c'est même le langage phare dans le domaine de l'intelligence artificielle, et il est encore (en 2025) en pleine expansion en terme de nombre d'utilisateurs et d'offres d'emplois. Pour une fois l'Éducation Nationale est vraiment à la pointe...
- **Libre** : n'importe qui peut l'installer sur son ordinateur, le tester, lire la documentation, le modifier et l'améliorer, et contribuer au développement. Le langage Python n'appartient pas à une entreprise mais à la *communauté Python*, qui est particulièrement grande et active.
- **Facile à lire** : l'accent est mis sur la lisibilité et la facilité à programmer. Cela en fait un bon choix pour l'enseignement et pour l'apprentissage d'un premier langage.
- **Dynamique** : en plus du mode script qui exécute tout un programme d'un seul coup, il y a un mode interactif qui exécute les commandes unes par unes et permet de faire des tests très simplement, inspecter le contenu des variables, naviguer dans l'aide, exécuter son programme pas à pas pour trouver des erreurs.
- **Haut niveau** : le langage fournit de très nombreuses fonctions et constructions qui sont « éloignées » du fonctionnement interne de l'ordinateur mais beaucoup plus faciles à manipuler pour un humain. En comparaison d'autres langages, un programme Python est souvent *plus court* et *plus rapide à écrire*, ce qui le rend si apprécié. En contrepartie, un programme Python est aussi *plus lent à fonctionner* : ce n'est pas le langage absolument idéal pour toutes les situations.
- **Extensible** : de très nombreuses bibliothèques sont disponibles, c'est-à-dire des extensions et des fonctions supplémentaires, qui permettent d'interagir avec les fichiers de l'ordinateur, le réseau, tracer des graphiques, traiter des statistiques, des images... et tout est expliqué dans la documentation.

En conséquence, pour un élève, c'est aussi nettement plus difficile qu'un langage de calculatrices ou qu'un langage comme *Scratch* (au collège avec le chat) car ceux-ci ont été conçus spécifiquement pour l'enseignement mais n'ont aucune utilité professionnelle !

Enfin les logiciels viennent avec leur propre documentation, écrite par ceux qui ont créé le logiciel, et qui décrit très précisément leur fonctionnement. *Tout* ne peut donc pas venir de l'enseignant : celui-ci connaît avant tout les concepts généraux de la programmation et a appris le langage Python, mais n'est pas responsable du logiciel lui-même et encore moins de ses nombreuses bibliothèques et logiciels liés... Il passe notamment du temps à lire les documentations, en anglais bien entendu ! Alors surtout pour progresser, en séance de TP, vous êtes toujours libres d'expérimenter !

À retenir

Pour progresser en informatique :

- **Expérimentez, testez avec vos propres valeurs** et avec vos propres messages les programmes proposés. Essayez avec des valeurs qui marchent bien et aussi avec des valeurs qui pourraient faire échouer les programmes et prenez le temps d'observer le résultat.
- **Lisez les messages d'erreur**. Ne paniquez pas mais apprenez à les lire et à les décoder et à corriger vous-même les erreurs en ré-essayant encore et encore.
- **Lisez les documentations**, les aides-mémoires, les aides interactives (fonction Python `help()`). Testez les fonctions et les options existantes décrites dans ces documentations.

TP 1

Prise en main

I Introduction au mode interactif

Dans le mode interactif, on écrit les commandes **une ligne à la fois** à la suite de **l'invite de commande >>>**. À chaque ligne, Python affiche le résultat du calcul, et enregistre au fur et à mesure les variables.

Il est donc possible de recopier et tester ligne par ligne les morceaux de programmes présentés. Pour progresser, **ne pas hésiter à prendre des initiatives, tester avec d'autres valeurs** que celles proposées ici et **lire les messages d'erreur** !

Les espaces sont optionnels, mais rendent le code plus lisible. Par contre la différence entre minuscules et majuscules est importante.

Un raccourci clavier utile : les flèches haut (↑) et bas (↓) permettent de faire réapparaître les entrées précédentes, pour éviter d'avoir à tout recopier à chaque fois, surtout en cas d'erreur.

I.1 Utilisation comme calculatrice

Dans un premier temps, on peut utiliser Python comme une simple calculatrice avec les nombres et les opérations **+**, **-**, *****, **/**. Le programme répond par le résultat du calcul.

```
>>> 3 + 4
7
>>> 8 - 10
-2
>>> 10 / 3
3.3333333333333335
```

Ces opérations vérifient les règles de priorité habituelles. Des parenthèses peuvent être nécessaires pour forcer la priorité.

```
>>> 2 + 3 * 5
17
>>> (2 + 3) * 5
25
```

L'opération puissance s'écrit ****** : ainsi 10^3 est

```
>>> 10 ** 3
1000
```

Exercice 1.1

Calculer les nombres suivants :

$$2^{16} \quad 3^{\frac{1}{2}} \quad 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \quad 1 + \frac{1}{1 + \frac{1}{1+\frac{1}{1}}} \quad \left(1 + \frac{1}{100}\right)^{100}$$

I.2 Les variables

En informatique, une variable est une case de la mémoire capable d'enregistrer une valeur ou le résultat d'un calcul. Pour la manipuler il faut lui donner un nom (qui peut être composé de plusieurs lettres). L'opération d'**affectation**, écrite avec le symbole **=**, permet de donner un contenu à la variable.

```
>>> x = 3
>>> x
3
>>> x + 4
7
>>> y = 8
>>> x - y
-5
```

En mathématiques on note parfois ceci $x \leftarrow 3$ pour bien signifier qu'il s'agit de **l'opération** de mettre dans la variable x la valeur 3.

À partir de ceci on peut vouloir mettre à jour la valeur de x :

```
>>> x = 3
>>> x
3
>>> x = x + 1
>>> x
4
```

Là encore, la ligne $x = x + 1$ ne se comprend pas vraiment comme une égalité mathématique mais plutôt comme l'opération $x \leftarrow x + 1$: remplacer la valeur de x par sa valeur augmentée de 1.

On remarque que rien ne s'affiche immédiatement après l'opération d'affectation. C'est normal, il s'agit d'une instruction, qui dit à l'ordinateur de réaliser une certaine opération en mémoire mais ne produit pas de résultat visible.

Les types de la section suivante peuvent tous être contenus dans une variable.

En mode interactif, le programme enregistre les variables au fur et à mesure et s'en souvient, jusqu'à ce qu'on lui demande de redémarrer.

II Les types

Les expressions et les variables Python ont toutes un **type**, qui indique quel type d'objet on manipule et comment l'ordinateur se les représente. Le type d'une expression peut-être obtenu avec la fonction `type()` :

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> type(3.5)
<class 'float'>
```

Il est important de connaître le fonctionnement d'un certain nombre de types de base. Les principaux types que nous manipulerons sont les suivants :

II.1 Le type entier `int`

C'est le type des nombres entiers comme 5 ou bien -2, en anglais *integer*, en Python `int`.

Un fait remarquable est que Python est capable de gérer des nombres entiers très grands !

```
>>> 2 ** 10
1024
>>> 2 ** 100
1267650600228229401496703205376
>>> 2 ** 1000 # à vous d'essayer !
```

Ce n'est pas si évident : nous verrons qu'un nombre est codé dans l'ordinateur par une suite de 0 et de 1 appelés *bits*, avec un nombre fixe de chiffres (en général 32 ou bien 64) ce qui donne un nombre maximal qu'on peut représenter. Ici, la place en mémoire pour stocker les entiers Python est agrandie automatiquement selon les besoins.

II.2 Le type flottant `float`

En informatique, on ne peut pas représenter tous les nombres réels avec une infinité de décimales après la virgule, car l'espace nécessaire pour stocker un tel nombre pourrait être infini...

Les nombres à virgule que l'on manipule sont appelés des **nombres à virgules flottantes** (ou plus simplement **flottants**) et forment le type `float`. Le symbole pour la virgule est le point `.` et on peut aussi utiliser la notation scientifique où la lettre `e` désigne la puissance de 10 :

```
>>> 1.23e4
12300.0
>>> 12.34e50
1.234e+51
>>> 1 - 1e-5
0.99999
```

Le nom de *virgule flottante* provient du fait que ces nombres sont représentés par l'ordinateur sous une certaine forme de notation scientifique, dans un emplacement mémoire de taille limitée avec une partie pour stocker l'exposant et une partie pour stocker les décimales. Ainsi la précision d'un nombre est toujours limitée à une quinzaine de chiffres, mais la partie avec exposant peut varier d'environ `e-308` à `e+308`. C'est la plupart du temps bien suffisant pour les sciences !

Mais à cause de cette précision limitée, il y a parfois des erreurs d'arrondis dans les nombres flottants...

Exercice 1.2

1. Tester :

```
>>> 1e30 + 0.1
>>> 1e30 + 0.1 == 1e30
```

2. Tester :

```
>>> 0.3
>>> 0.1 + 0.2
>>> 0.1 + 0.2 == 0.3
```

3. Cela est bien visible aussi avec des fonctions spéciales, tester :

```
>>> from math import *
>>> sin(2*pi)
>>> sin(2*pi) == 0
```

et

```
>>> from math import *
>>> exp(log(3))
>>> exp(log(3)) == 3
```

L'opération de division `/` donne toujours un nombre flottant, même entre nombres entiers :

```
>>> 10 / 5
2.0
>>> type(10 / 5)
<class 'float'>
```

Pour travailler uniquement avec des nombres entiers on utilise la division euclidienne `//`, celle qui donne un quotient et un reste :

```
>>> 10 // 5
2
>>> 11 // 5
2
>>> 19 // 5
3
```

Le reste lui est obtenu par l'opération `%`. Cela parlera mieux aux élèves qui ont étudié l'arithmétique...

```
>>> 10 % 5
0
>>> 11 % 5
1
>>> 19 % 5
4
```

C'est la bonne méthode pour tester la parité : un entier n est pair quand $n \% 2$ vaut 0, et impair si $n \% 2$ vaut 1.

II.3 Le type des chaînes de caractères `str`

Ce que l'on appelle communément du *texte* forme aussi un type, que l'on peut enregistrer dans des variables et que l'on peut vouloir manipuler. En informatique, un tel texte s'appelle une **chaîne de caractères**, en anglais *string*. En effet l'ordinateur les considère comme une liste de caractères les uns à la suite des autres, indépendamment de savoir si le texte a du sens ou non. Elles forment le type `str` et sont écrites encadrées préférablement par le guillemet double `"texte"`, même si le guillemet simple (apostrophe `'`) est possible aussi.

```
>>> x = "Bonjour"
>>> type(x)
<class 'str'>
```

L'opération `+` sur les chaînes de caractères crée une nouvelle chaîne en plaçant les deux bout à bout et s'appelle en informatique la **concaténation**.

```
>>> "Bonjour" + "et bienvenue"
'Bonjour et bienvenue'
```

Corrigeons ceci... premier essai :

```
>>> "Bonjour" + " et bienvenue"
'Bonjour et bienvenue'
```

Deuxième essai :

```
>>> x = "Bonjour"
>>> y = "et bienvenue"
>>> x + " " + y
'Bonjour et bienvenue'
```

Il existe aussi une opération de multiplication entre une chaîne et un entier. À votre avis, que fait-elle ?

Exercice 1.3

Tester :

```
>>> "Ha" * 5
```

La syntaxe crochets [i] permet d'accéder au *i*-ième caractère de la chaîne :

```
>>> x = "Bonjour"
>>> x[1]
'o'
>>> x[2]
'n'
```

... en fait, elle est numérotée à partir de 0 : c'est `x[0]` qui donne 'B'.

Avec les indices négatifs, on repart de la fin !

```
>>> x = "Bonjour"
>>> x[-1]
'r'
>>> x[-2]
'u'
```

Ces conventions sont peut-être étonnantes au début, mais cela reviendra de nombreuses fois...

Exercice 1.4

Tester et expliquer la différence entre les deux expressions suivantes :

```
>>> 12 + 34
>>> "12" + "34"
```

À retenir

Les variables Python ont toutes un **type**. Chaque type a ses propriétés. Les opérations ne se comportent pas toutes de la même façon en fonction du type des variables, même si en apparence leur contenu est le même.

II.4 Le type booléen

C'est un type à part entière qui représente une valeur vrai ou faux, en Python `True`, `False`. Le nom provient du mathématicien anglais George Boole. On appelle donc ces valeurs des **booléens**, formant le type `bool`.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

On peut utiliser dessus les opérations

- et : `and`
- ou : `or`
- non : `not`

... et vérifier toutes les propriétés que nous avons vues en cours sur les assertions !

```
>>> True and True
True
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> not False
True
```

et ainsi de suite. Si on en combine plusieurs, on peut utiliser des parenthèses.

Exercice 1.5

Donner le résultat de l'expression suivante :

```
>>> not True and False
```

Cela permet d'en déduire laquelle de ces deux opérations est prioritaire : le **not** ou bien le **and** ?

Les opérations de comparaisons entre nombres donnent une valeur booléenne. Ce sont les suivantes :

- L'égalité : `==`
- La différence : `!=`
- Les comparaisons strictes : `<`, `>`
- Les comparaisons larges `≤`, `≥` : `<=`, `>=`

Quelques exemples :

```
>>> 5 > 8
False
>>> 5 + 3 <= 8
True
>>> 1 == 2
False
>>> 1 != 2
True
```

Insistons encore une fois sur le fait que ce sont des types à part entière, pouvant rentrer dans des variables :

```
>>> résultat = 3 < 4
>>> not résultat
False
```

Les opérations logiques ne sont pas tout à fait *commutatives* comme en mathématiques...

Exercice 1.6

1. Tester et expliquer la différence entre les deux expressions :

```
>>> 1 + 2 == 5 and 1 / 0 == 2
>>> 1 / 0 == 2 and 1 + 2 == 5
```

2. Pouvez-vous mettre en place un test similaire pour illustrer le comportement du **or** ?

II.5 Les conversions de type

Comme on l'a vu, le nombre `3` n'est pas la même chose que `3.0` car le premier est de type `int` et le second est de type `float`. Et la chaîne de caractères `"12"` n'est pas la même chose que le nombre entier `12`.

Aux types correspondent aussi des fonctions de conversion de type `int()`, `float()`, `str()`, `bool()`, qui convertissent vers le type :

```
>>> float(5)
5.0
>>> int(3.5)
3
>>> int("12") + int("34")
46
>>> str(12) + str(34)
'1234'
```

Via `bool()`, n'importe quel nombre est converti à `True`, sauf `0`. N'importe quelle chaîne est convertie en `True`... sauf la chaîne vide `""`. Il y a des bonnes raisons historiques pour cela : puisqu'un entier est codé sur au minimum 8 bits, il a été convenu que le `0` représentait `False` et toute autre valeur représentait `True`.

III Le mode script

En mode script, on peut écrire plusieurs lignes à la suite et l'envoyer d'un coup à l'interpréteur Python. Les instructions sont obligatoirement séparées par un retour à la ligne. Le programme continue à enregistrer les variables au fur et à mesure, mais **il n'affiche rien si on ne le lui demande pas** ! Il faut alors utiliser la commande `print()` pour afficher quelque chose.

```
x = 3
y = 8
x = x - y
print(x)
```

On peut aussi se servir de `print` pour afficher plusieurs variables, ou nombres ou chaînes, à la suite sur une même ligne.

```
x = 5
y = 12
print("x =", x, "y =", y)
```

Toute ligne commençant par le symbole `#` est ignorée : c'est un **commentaire**, qui sert à expliquer ce que fait le programme. Les commentaires sont loin d'être inutiles car ils permettent à d'autres personnes qui lisent le programme de mieux le comprendre.

Dans les logiciels Pyzo ou Spyder, une ligne commençant par `###` (dièse, pourcent, pourcent) permet de séparer le code en **cellules** pour garder tout le code sur une même page, mais ne demander à n'exécuter qu'une seule cellule à la fois (sinon, on rajoute du code au fur et à mesure, et à chaque fois, tout est exécuté depuis le début). Une bonne idée est de l'utiliser pour séparer chaque question des exercices. L'option s'appelle « exécuter la cellule » (*execute cell*), qu'on trouve aussi sur le raccourci clavier `Ctrl` + `Entrée`. Toute l'année, les fichiers seront présentés divisés en cellules. Le fichier ressemble à ceci :

```
### exercice 1
...

### exercice 2
...
```

Enfin la dernière notion utile pour entamer le mode script est la fonction `input()` qui permet de demander une information à l'utilisateur.

```
x = input("Entrez quelque chose : ")
print("Vous avez entré", x)
```

La valeur donnée par `input()` est toujours de type `str` ! Les conversions de type apparaissent donc bien utiles ici. Si on veut demander un nombre (entier) à l'utilisateur, on écrit en général directement `int(input())` :

```
x = int(input("Entrez un nombre entier : "))
print("Son double est :", 2 * x)
```

Si on veut un nombre qui peut avoir une virgule, alors c'est la conversion `float()` qu'il faut utiliser, sinon une erreur se produit : on tente de convertir en nombre entier du texte avec une virgule dedans !

IV D'autres exercices

Dans les exercices suivants, on demande d'écrire des petits bouts de programme : quelques lignes de code dans le mode interactif, séparées en cellules, de façon à pouvoir les exécuter séparément. À ce stade, pour que les programmes soient un minimum intéressants, ils utilisent largement `input()` pour que l'utilisateur puisse varier un paramètre.

Exercice 1.7

Écrire un programme qui demande à l'utilisateur son nom, puis affiche `Bienvenue [nom] en BCPST1B !` (en remplaçant par son nom).

Pouvez-vous le faire en mettant ce texte dans une seule variable ?

Exercice 1.8

Écrire un programme qui demande à l'utilisateur son année de naissance, puis affiche son âge en 2025 (on ne se préoccupera pas du mois de naissance...), sous la forme `Vous avez [ans] ans`.

Idem, pouvez-vous mettre ce texte dans une seule variable ?

Exercice 1.9

Écrire un programme qui demande à l'utilisateur d'entrer deux nombres `a` et `b` et en donne la moyenne.

Exercice 1.10

Écrire un programme qui demande à l'utilisateur d'entrer deux nombres `a` et `b`, les affiche, puis les échange, et les affiche encore. En échangeant réellement la valeur des variables, pas seulement l'affichage...

Exercice 1.11

On peut utiliser `print()` directement sur une expression booléenne pour afficher simplement `True` or `False`.

Écrire des programmes qui demandent à l'utilisateur des nombres et testent, en affichant le résultat, les conditions suivantes :

1. Le nombre `n` est pair (en utilisant la division `%`).
2. Les nombres entiers `n` et `m` sont de même signe.
3. Les nombres entiers `n`, `m`, `p` sont deux à deux distincts.

Exercice 1.12

Sur un caractère tout seul `x`, la fonction `x.isupper()` donne `True` si `x` est en majuscule et `False` sinon. Essayez-là !

Écrire un programme qui demande à l'utilisateur d'entrer une phrase, et teste si la phrase commence par une majuscule et termine par un point.

TP 2

Conditions et boucles

Aujourd'hui nous écrivons des programmes avant tout dans le mode script (sur plusieurs lignes) ; le mode interactif sert à faire des tests courts ou à inspecter le contenu ou le type des variables. Séparer les programmes et les exercices par des **cellules** délimitées par une ligne commençant par la suite de trois caractères `###` permet de tout garder sur une même page (donc d'enregistrer dans un même fichier `.py`, mais de n'exécuter qu'un morceau à la fois et pas tout depuis le début. La présentation du fichier doit donc ressembler à ceci, comme dans les corrections :

```
### exercice 1
...

### exercice 2
...
```

On utilise alors l'option « exécuter la cellule » (*execute cell*), aussi sur le raccourci clavier `Ctrl` + `Entrée`.

I Conditions simples

Le mot-clé `if` suivi d'une condition introduit un morceau de programme qui va être exécuté **si** la condition est vérifiée. Éventuellement, le mot-clé `else` (**sinon**) introduit un morceau de programme qui va être exécuté dans le cas contraire.

Voici un exemple de bout de programme, qu'on peut recopier tel quel et tester :

```
x = int(input("Entrez un nombre : "))
if x >= 0:
    print("positif")
else:
    print("strictement négatif")
print("FIN")
```

Cela est bien entendu un concept fondamental de la programmation, qui permet de rendre un programme interactif et dont le résultat va dépendre des entrées.

La syntaxe générale en Python est la suivante :

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

où :

- `condition` désigne n'importe quelle expression booléenne que le programme va tester. Elle est formée notamment, on le rappelle, avec
 - L'égalité : `==`
 - La différence : `!=`
 - Les comparaisons strictes : `<`, `>`
 - Les comparaisons larges : `<=`, `>=`
 - Les mots-clés `and`, `or`, `not`
- `instructions` est du code Python, qui peut être sur plusieurs lignes, qui va être exécuté seulement si la condition a été évaluée à `True`. La partie du programme qui va être exécutée est tout ensemble décalée vers la droite (on dit **indentée**). En général le décalage est de 4 espaces, ou un seul caractère tabulation ; le logiciel sert notamment à bien aligner les lignes. Elle forme un **bloc d'instructions**.
- `instructions sinon` est un **autre bloc d'instructions** qui va être exécuté dans le cas contraire.

- Ensuite, le code qui n'est plus décalé vers la droite ne fait plus partie des blocs d'instructions ; il est donc exécuté dans tous les cas. Le `else` n'est d'ailleurs pas du tout obligatoire : si la condition est fausse alors le premier bloc d'instruction n'est pas exécuté et le programme passe directement à la suite.

On n'oubliera pas non plus le symbole double points, qui fait partie de la syntaxe, termine la ligne et introduit le bloc d'instructions. Il permet aussi au logiciel de proposer directement d'indenter, le saut de ligne après le double point décale automatiquement à droite et on n'a rarement besoin d'écrire à la main les espaces ou tabulations.

Exercice 2.1

Écrire un programme qui demande à l'utilisateur son âge, et affiche s'il est majeur ou mineur.

Exercice 2.2

Écrire un programme qui demande à l'utilisateur un nombre, et affiche sa valeur absolue.

Exercice 2.3

Choisissez un mot de passe secret ; écrire un programme qui demande à l'utilisateur d'entrer un mot, et qui lui dit si c'est le bon mot de passe ou non.

Le bloc d'instructions peut lui-même contenir d'autres conditions emboîtées. L'indentation des blocs est alors cruciale.

Exercice 2.4

Améliorer programme d'exemple ci-dessus pour demander à l'utilisateur un nombre et afficher s'il est positif, négatif ou nul. Sans regarder la suite du TP.

II Conditions en cascade

Il arrive que l'on veuille tester une condition plus complexe qui ne se traduit pas aussi simplement que « si... alors ». Le mot-clé `elif` est la contraction de « else, if » (**si**on, **si**) et introduit une condition qui va être testée si la précédente était fausse, ainsi qu'un bloc d'instruction correspondant à exécuter. Par exemple le programme du dernier exercice peut se ré-écrire ainsi :

```
x = int(input("Entrez un nombre : "))
if x > 0:
    print("positif")
elif x < 0:
    print("négatif")
else:
    print("nul")
```

Il est tout à fait possible d'enchaîner plusieurs `elif` ; les conditions sont simplement testées les unes à la suite des autres. Le `else` final capture le cas où **aucune** des conditions n'a été vérifiée ; il n'est toujours pas obligatoire, si aucune condition n'est vérifiée le programme passe simplement à la suite. La syntaxe générale ressemble donc à :

```
if condition1:
    instructions1
elif condition2:
    instructions2
elif ... :
    ...
else:
    instructions sinon
```

Ainsi lors de l'exécution :

- Le programme teste la `condition1`. Si elle est vraie alors il exécute `instructions1`.
- Sinon, il vérifie la `condition2`. Si elle est vraie alors il exécute `instructions2`.

- Et ainsi de suite.
- À la fin, si aucune condition n'a été vérifiée, le programme exécute le bloc d'instructions du **else**.

Exercice 2.5

Écrire un programme qui demande son âge à l'utilisateur, et affiche s'il est majeur, mineur, ou senior (plus de 65 ans).

Exercice 2.6

Écrire un programme qui demande à l'utilisateur sa note au bac (attention, ce n'est pas forcément un nombre entier) et affiche à quelle mention cela correspond. Vous pouvez l'assortir librement d'un commentaire sur la note...

À retenir

Les conditions en cascade sont testées **successivement** quand la précédente a échoué. À cause de cela, il y a un ordre logique et naturel dans lequel on doit écrire ses conditions. De plus le mot **else** n'est pas suivi par une condition (c'est un « sinon » tout court) et il s'exécute quand aucune des conditions précédentes n'a été vérifiée.

III Boucles

Une **boucle** permet de répéter des instructions automatiquement. C'est à partir de maintenant que les programmes deviennent *vraiment* intéressants : ils automatisent des tâches qui seraient bien pénibles pour un humain.

Dans ce TP on se concentre sur la boucle **while**. Le bloc d'instructions (**corps** de la boucle) est répété **tant que** (traduction de *while*) une condition est vérifiée. La syntaxe est tout simplement la suivante :

```
while condition:
    instructions
```

alors lors de l'exécution :

- Le programme teste si la condition est vraie,
- Si oui, il exécute le bloc d'instructions. Une fois fini, il recommence à évaluer la condition et ainsi de suite.
- Sinon, il sort simplement de la boucle et passe à la suite.

Pour que cela soit intéressant, il faut que la condition puisse varier à chaque passage dans la boucle ! Sinon, si elle est toujours vraie, rien ne l'arrête et on obtient une **boucle infinie**... La méthode de base pour répéter une instruction un certain nombre n de fois est de déclarer une variable appelée **compteur**, que l'on va augmenter de 1 (on dit **incrémenter**) à chaque passage dans la boucle, et de tester la condition $i < n$:

```
i = 0
while i < 3:
    print("i =", i)
    i = i + 1
print("fin avec i =", i)
```

produit le résultat suivant :

```
i = 0
i = 1
i = 2
fin avec i = 3
```

Avertissement

Dans chaque boucle, on porte une attention particulière à :

- La valeur à laquelle le compteur est initialisé (essayez avec $i = 1$)
- L'utilisation de la comparaison stricte $<$ ou large $<=$ (essayez de remplacer par $<=$)
- L'incrémentation au début ou à la fin de la boucle (essayez d'échanger les deux lignes dans le bloc d'instructions),
- Ne pas oublier l'incrémentation (essayez de l'enlever !)

L'exemple ci-dessus est le plus standard et est écrit de telle façon à ce qu'il répète exactement n fois.

Exercice 2.7

Écrire un programme qui affiche les nombres x^2 pour $1 \leq x \leq 10$.

Exercice 2.8

Écrire un programme qui demande à l'utilisateur un nombre n et compte à rebours : affiche n puis $n-1$ puis... jusqu'à 0 . Il y a deux façons possibles, tester les deux :

1. Incrémenter i comme ci-dessus, mais afficher la quantité $n-i$.
2. **Décrémenter** i , c'est-à-dire le faire diminuer de 1 avec $i = i - 1$, mais alors il faut changer la condition et la valeur initiale.

IV Application aux suites

Pour calculer les termes successifs d'une suite, on se sert en plus d'une variable u qui à chaque passage dans la boucle va devenir le terme suivant. L'exemple de base pour les puissances de 2 est :

```
u = 1
i = 0
while i < 5:
    print(u)
    u = 2 * u
    i = i + 1
```

qui produit l'affichage

```
1
2
4
8
16
```

autrement dit les 5 premières puissances, soit de 2^0 à 2^4 .

Exercice 2.9

Écrire un programme qui demande à l'utilisateur un nombre n et calcule la somme $1 + 2 + \dots + n$.

L'intérêt de la boucle **while**, c'est aussi de pouvoir s'arrêter quand une certaine condition sur la suite devient vérifiée — c'est à dire la continuer **tant que** elle n'est **pas** vérifiée.

Exercice 2.10

Au début de l'an 2025, la population mondiale est estimée à environ 8,23 milliard d'habitants. Elle augmente d'environ 0,85 % chaque année. Écrire un programme qui affiche la population mondiale estimée sur les années futures (afficher à la fois l'année et la population, par exemple **Année 2025 : 8230000000**) si le taux de croissance reste le même, et s'arrête quand elle dépasse 10 milliards.

Exercice 2.11 (*)

La **suite de Syracuse** est la suite entière $(S_n)_{n \in \mathbb{N}}$ définie par : le nombre $S_0 \geq 1$ est à déterminer par l'utilisateur, et ensuite

$$S_{n+1} = \begin{cases} S_n/2 & \text{si } S_n \text{ est pair} \\ 3 \times S_n + 1 & \text{si } S_n \text{ est impair} \end{cases}$$

1. Que se passe-t-il si $S_0 = 1$?
2. Écrire un programme qui demande à l'utilisateur le nombre S_0 puis affiche tous les termes de la suite jusqu'à ce qu'un terme soit égal à 1.
3. La célèbre **conjecture de Syracuse** est l'énoncé selon lequel quelque soit le nombre S_0 , la suite finit par retomber sur 1. Il s'agit d'un problème ouvert célèbre...

Écrire un programme qui nous aide à vérifier cette conjecture, en demandant à l'utilisateur un entier N puis en testant la conjecture pour tout $1 \leq S_0 \leq N$, affichant pour chaque valeur testée si la conjecture est bien vraie.

4. En plus, pouvez-vous compter, pour chaque S_0 , en combien d'étapes la suite revient à 1 ?

On peut vérifier la conjecture pour des valeurs extrêmement grandes, mais ceci ne constitue malheureusement toujours pas une démonstration mathématique...

TP 3

Fonctions

Aujourd'hui et à partir de maintenant nous écrivons avant tout des **fonctions** dans le mode script. Cela permet de faire varier des paramètres, et de récupérer une valeur calculée.

L'exécution d'un code de déclaration de fonction ne produit aucun affichage, il faut donc tester les fonctions dans le mode interactif ou bien directement à la suite du script. On continue à séparer les exercices par des cellules avec les trois caractères `###`. On rappelle que le raccourci clavier `(Ctrl) + (Entrée)` permet d'exécuter la cellule ; dans le mode interactif, jouer avec les flèches `(↑)` et `(↓)` pour faire réapparaître les entrées précédentes.

I Notion de fonction

I.1 Déclaration et appel

Une fonction correspond à un morceau de programme ré-utilisable, dans lequel on peut faire varier des paramètres. Elle se déclare avec le mot-clé `def`. Étudions l'exemple suivant :

```
def somme(x, y):  
    print("La somme de", x, "et de", y, "est", x+y)
```

Ce code est une **déclaration** de fonction, son exécution ne produit rien mais enregistre la fonction.

Les tests suivants en mode interactif permettent de faire varier les paramètres. On parle d'**appel** de la fonction.

```
>>> somme(3, 4)  
La somme de 3 et de 4 est 7  
>>> somme(-6, 2)  
La somme de -6 et de 2 est -4
```

Les variables `x` et `y` s'appellent des **arguments** de la fonction. Une fonction `f` peut avoir un ou plusieurs arguments, ou aucun — dans ce dernier cas l'appel s'écrit juste `f()`. Les arguments peuvent être de n'importe lesquels des types manipulés jusque là.

Ce qui suit le mot-clé `def` forme un **bloc d'instructions**, exactement comme dans les conditions et les boucles. On parle du **corps** de la fonction. Il peut donc contenir lui aussi des variables, des conditions et des boucles.

I.2 L'instruction `return`

Dans le bloc d'instructions, lorsque le programme tombe sur une ligne commençant par `return`, **l'exécution de la fonction s'arrête**. La valeur ou l'expression qui suit est dite **renvoyée par la fonction** et on peut alors récupérer sa valeur. Prenons l'exemple

```
def moyenne(x, y):  
    return (x + y) / 2
```

alors l'appel

```
>>> m = moyenne(12, 14)
```

- Appelle la fonction, en donnant les valeurs $x \leftarrow 12$ et $y \leftarrow 14$,
- Calcule le résultat de $(x + y) / 2$ qui donne le nombre 13.0,
- **renvoie** ce résultat, et le met ici dans la variable $m \leftarrow 13.0$.

On peut le vérifier :

```
>>> m  
13.0
```

Ce comportement ne serait pas possible si on écrivait `print((x + y) / 2)` au lieu de `return` : la valeur serait bien affichée mais serait ensuite perdue et on ne pourrait pas la récupérer dans une variable. Et il est crucial pour la ré-utilisabilité de la fonction de pouvoir ainsi enregistrer la valeur qu'elle calcule.

L'instruction `return` n'est pas obligatoire. Si l'exécution de la fonction arrive au bout sans avoir rencontré de `return`, elle ne renvoie pas de valeur. Si `return` est présent sans valeur de retour, l'exécution de la fonction s'arrête mais ne renvoie pas de valeur. L'exemple suivant est donc bien valide

```
def f():
    print("Cette fonction n'a pas de paramètres ni de valeur de retour.")
    print("Est-elle pour autant utile ?")
    return
    print("Ceci, par contre, ne s'affichera jamais.")
```

et son appel ne fait qu'afficher toujours le même texte des deux premières lignes.

Un autre exemple instructif est celui-ci, comme son nom l'indique :

```
def divise_proprement(x, y):
    if y == 0:
        print("Il ne faut JAMAIS diviser par 0 !!!")
        return
    return x / y
```

Alors :

- Si l'argument `y` est `0`, on entre dans le premier `if`. Le message d'avertissement s'affiche. Puis la fonction se termine, à cause du `return`.
- Sinon, le bloc d'instruction du `if` n'est pas exécuté. La fonction calcule `x / y` et renvoie cette valeur.
- Remarquez que du coup le `else` n'est pas nécessaire ici : l'instruction `return x / y` se produit uniquement dans le cas où `y` est non-nul, sinon elle s'est arrêtée avant !

Le test donne :

```
>>> q = divise_proprement(6, 2)
>>> q
3.0
# q contient bien une valeur, le résultat
>>> q = divise_proprement(5, 0)
Il ne faut JAMAIS diviser par 0 !!!
>>> q
# ici rien ne s'affiche ! q existe mais ne contient pas de valeur !
```

(en fait, `q` contient la valeur spéciale `None`, qui risque d'apparaître si on écrit `print(q)`)

À retenir

À partir de maintenant, dans les exercices en TP mais aussi à l'écrit, nous écrivons la plupart du temps des **fonctions**, qui ont des arguments, effectuent un certain calcul, et renvoient le résultat avec `return`. Sauf demande explicite, elles ne font pas de `print()` (le résultat calculé est renvoyé, c'est l'utilisateur qui décidera ce qu'il fait avec, ce n'est pas la fonction qui décide de comment elle va l'afficher) ni de `input()` (de même, les arguments sont passés à la fonction par l'utilisateur, ce n'est pas à la fonction de décider avec quel message elle va les demander).

I.3 Notion de variable locale

Une fonction a bien le droit d'utiliser des variables dans son corps, et les arguments eux-mêmes sont des variables à part entière. Cependant, à moins que ces variables soient renvoyées avec un `return`, elles sont **détruites** à la fin de l'exécution de la fonction et donc ne sont pas accessibles au reste du programme. On parle de **variables locales**. Reprenons l'exemple

```
def moyenne(x, y):  
    m = (x + y) / 2  
    return m
```

alors en mode interactif

```
>>> moyenne(12, 14)  
13.0  
>>> x  
NameError: name 'x' is not defined  
>>> m  
NameError: name 'm' is not defined
```

Ces variables n'existent plus !

Cela correspond à la notion mathématique de variable non-libre, on dit aussi *liée* ou *muette*.

Cela signifie aussi que l'on peut écrire

```
>>> m = 10  
>>> moyenne(12, 14)  
13.0  
>>> m  
10
```

car la variable `m` qui est manipulée à l'intérieure de la fonction n'est pas vraiment la même que celle qui pourrait déjà exister avant.

II À vous de jouer !

Les fonctions suivantes ne font pas intervenir de connaissances en Python autres que celles vues dans les TP précédents, mais la mise en forme est différente : les fonctions ont des arguments, font des calculs, utilisent éventuellement des variables locales, des conditions et des boucles, et renvoient une valeur avec `return`.

Exercice 3.1

Écrire la fonction `perimetre_rectangle(a, b)` qui renvoie le périmètre du rectangle de côtés `a` et `b`.

Exercice 3.2

Écrire la fonction `valeur_absolue(x)` qui renvoie la valeur absolue de `x`.

Exercice 3.3

Écrire la fonction `maximum(a, b, c)` qui renvoie le plus grand des trois nombres entre `a`, `b` et `c`.

Exercice 3.4

- Écrire la fonction `partie_entiere(x)` qui prend en argument un nombre à virgule flottante `x`, qui pour l'instant ne marchera que si `x` est positif, et qui calcule sa partie entière de la façon suivante : une boucle `while` cherche le plus grand entier `n` qui est inférieur ou égal à `x`.
- Bonus : améliorer la fonction pour traiter séparément le cas où `x` est négatif. Attention à bien la tester dans tous les cas, `x` positif ou négatif, entier ou non.

Exercice 3.5

Écrire la fonction qui calcule la partie entière de la racine carrée sans utiliser la fonction partie entière ni la fonction racine carrée (*Hein ???*)

III L'aide

Toutes les fonctions définies en Python possèdent une aide accessible (en mode interactif) avec la commande `help()` : testez

```
>>> help(abs)
```

Chacun peut créer de la documentation pour sa propre fonction en insérant juste après la déclaration du texte entre trois guillemets doubles successifs `""" documentation """`, ce que Python appelle une **docstring**, contraction de *documentation string* (chaîne de documentation), et qui peut même tenir sur plusieurs lignes :

```
def moyenne(x, y):  
    """ C'est MA fonction qui calcule la moyenne.  
        Entrez deux nombres x et y, et elle vous donne la moyenne. """  
    return (x + y) / 2
```

Lancer le programme puis essayer en mode interactif :

```
>>> help(moyenne)
```

La possibilité d'intégrer la documentation dans le corps même de la fonction, de naviguer dans l'aide en mode interactif, et la grande qualité de la documentation Python déjà existante, contribuent pour beaucoup à la diffusion de ce langage et à sa facilité d'apprentissage. C'est une bonne pratique de documenter son programme pour que d'autres puissent l'utiliser.

Exercice 3.6

Remplir soigneusement les documentations des fonctions de la partie II.

IV Les modules

Un **module** (ou aussi : bibliothèque, librairie) est un ensemble de fonctions. Elles sont groupées par thème et permettent de donner des nouvelles possibilités au langage. Les modules Python de base permettent de trouver les fonctions mathématiques, les nombres aléatoires, l'écriture dans des fichiers, mais aussi de connaître la date, communiquer en réseau, traiter des images... Cela contribue au succès de Python d'avoir des milliers de bibliothèques disponibles et d'interagir dans autant de situations.

Un module se charge (une fois pour toute !) avec la commande `import`, écrite en mode interactif ou avant le code qui va l'utiliser, qui a plusieurs syntaxes. Prenons pour exemple le module `math` qui comporte beaucoup de fonctions mathématiques comme la fonction exponentielle `exp`, la fonction sinus `sin`, et la constante mathématique `pi`. Il n'est pas chargé par défaut, donc on ne peut pas les utiliser tout de suite. On a les possibilités suivantes, avec des différences dans la manière dont sont ensuite nommées les fonctions :

- Importer tout, et y accéder avec le préfixe :

```
import math  
math.exp(x), math.sin(x), math.pi, ...
```

- Importer seulement les fonctions dont on a besoin, et y accéder sans préfixe :

```
from math import exp, sin, pi  
exp(x), sin(x), pi
```

- Il est courant aussi de donner un **alias** au module, introduit avec le mot-clé `as` :

```
import math as m  
m.exp(x), m.sin(x), m.pi, ...
```

Exemple courant : le sous-module `pyplot` du module `matplotlib` est un peu long à écrire.

```
import matplotlib.pyplot as plt
plt.plot(), plt.show(), ...
```

Cela marche aussi avec les fonctions et variables elles-mêmes, bien que l'utilité en soit discutable...

```
from math import pi as plus_beau_nombre_de_l_univers
print(plus_beau_nombre_de_l_univers)
# 3.141592653589793
```

- On trouve aussi parfois la syntaxe

```
from math import *
```

qui importe d'un coup **toutes** les fonctions du module, sans le préfixe. **Cette syntaxe est déconseillée** car le module peut contenir beaucoup de fonctions et on ne sait pas forcément à l'avance lesquelles... Si l'utilisateur a déjà une fonction `f` et que le module contient lui-même une fonction aussi nommée `f`, alors celle de l'utilisateur sera « désactivée » après l'importation et remplacée par celle du module ! Et si on importe plusieurs modules contenant chacun une fonction `f`, on ne sait plus à laquelle on fait référence !

Un module possède aussi une documentation, accessible une fois chargé :

```
>>> import math
>>> help(math)
```

Exercice 3.7

Importer le module `math` comme ci-dessus, lire sa documentation, et éventuellement celle des fonctions contenues dedans, et trouver :

1. La fonction racine carrée,
2. La fonction partie entière,
3. Les PGCD (plus grand commun diviseur) et PPCM (plus petit commun multiple).

... d'ailleurs, il y a plusieurs parties entières ? Quelle différence avec la conversion de type `int(x)` ? Tester avec plusieurs valeurs, entières ou non, positives ou négatives.

Remarque. On a donné ici diverses syntaxes ; le but est d'être capable de les reconnaître et de les interpréter, pas de tout savoir par cœur. En général, un sujet de concours qui utilise un module précise comment il est importé, par exemple « on suppose qu'on a importé le module `math` avec la commande `import math as m` » et donc il faut savoir que les fonctions s'appellent alors `m.exp(x)`, `m.sin(x)`, etc. Si ce n'est pas précisé, alors cela peut être au candidat de ne pas oublier d'écrire la commande d'importation !

Aux épreuves orales le candidat peut être devant un ordinateur avec le logiciel Pyzo. Si bien entendu il ne pourra pas accéder librement à tout l'internet, utiliser l'aide intégrée de Python est parfaitement légal.

Exercice 3.8

Écrire une fonction Python qui correspond à la fonction mathématique

$$f : x \mapsto \sin^3(2\pi x)e^{\sqrt{x}}$$

Un autre module d'intérêt est le module `random`, lié à tout ce qui est l'aléatoire, qui contient la fonction `randint` générant un nombre aléatoire entier. Nous l'approfondirons en lien avec les chapitres de probabilités.

Exercice 3.9

Importer uniquement cette fonction, lire sa documentation, et écrire une fonction `dé()` qui renvoie le résultat d'un lancer de dé cubique aléatoire.

Exercice 3.10

Écrire une fonction `réponse()` qui, en tirant au hasard un nombre entre 1 et 4, renvoie un choix aléatoire entre les mots "oui", "non", "peut-être" ou "je ne sais pas".

V D'autres exercices

La commande spéciale `assert`, suivie d'une condition et éventuellement d'un message d'erreur, permet de faire échouer tout le programme, en provoquant une erreur (en rouge) et en affichant le message, si la condition **n'est pas** vérifiée. On écrit par exemple une ligne

```
assert x >= 0, "x doit être positif"
```

Le verbe anglais *assert* (comme dans *assertion*) se traduit ici avec le sens plus fort de « affirmer que », autrement dit on affirme que `x` doit être positif pour pouvoir continuer. Cela lui donne un sens proche du langage mathématique « supposons $x \geq 0$ » — si ce n'est pas le cas alors la suite n'a pas de sens.

Il est courant qu'une fonction démarre par une ou plusieurs assertions qui servent à vérifier si les arguments donnés sont bien valides et évitent de faire des calculs qui provoqueront plus tard une erreur ou donneront des résultats incohérents. Par exemple la fonction `divise_proprement()` de § 1.2 peut s'écrire

```
def divise_proprement(x, y):
    assert y != 0, "Il ne faut JAMAIS diviser par 0 !!!"
    return x / y
```

et l'exécution s'affiche comme un message d'erreur Python

```
>>> divise_proprement(5, 0)
AssertionError: Il ne faut JAMAIS diviser par 0 !!!
```

Dans les exercices suivants, vous pouvez utiliser `assert` pour éviter que les programmes donnent des erreurs, des boucles infinies ou des résultats incohérents.

Exercice 3.11

La *moyenne harmonique* de deux nombres x, y tous les deux strictement positifs ou bien tous deux strictement négatifs est l'unique nombre H tel que

$$\frac{1}{H} = \frac{1}{2} \left(\frac{1}{x} + \frac{1}{y} \right)$$

Écrire une fonction `moyenne_harmonique(x, y)` qui renvoie la moyenne harmonique de `x` et de `y`.

Exercice 3.12

Écrire une fonction `seuil(t)` qui prend en argument un nombre réel `t` et renvoie le plus petit entier $n > 0$ pour lequel

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \geq t.$$

Tester des valeurs de `t` de plus en plus grandes entre 3 et 20 (doucement, pas à pas !)

Remarque. Il n'est jamais exigé des étudiants d'utiliser `assert`, et c'est même très déconseillé à l'écrit. En effet en algorithmique on considère que le programme est correct seulement s'il l'est quand on lui donne des bonnes

valeurs, et c'est l'utilisateur qui est responsable de donner les bonnes valeurs. Par exemple une phrase telle que « écrire une fonction qui prend en argument un réel x supposé positif... » se comprend comme : la fonction doit donner le bon résultat si $x \geq 0$, on ne se préoccupe pas du reste. Ce n'est pas tout à fait la même façon de penser que si on écrivait un programme convivial qui devrait gérer toutes les erreurs possibles en prévenant l'utilisateur que ses valeurs sont incorrectes, en lui demandant de recommencer, etc et cela est en fait un problème assez compliqué à gérer.

Exercice 3.13

On rappelle que la divisibilité se teste à partir de l'opération `%` (`a % b` est le reste dans la division euclidienne de a par b , nul si et seulement si b divise a).

1. Écrire une fonction `est_premier(n)` qui prend en argument un entier n en supposant $n \geq 2$ et qui renvoie `True` si n est premier et `False` sinon, en tentant de diviser n par tous les entiers strictement entre 1 et n .
2. En déduire une fonction `premiers(n)` qui affiche tous les nombres premiers inférieurs ou égaux à n .

TP 4

Boucles for

Dans ce TP nous approfondissons sur les boucles, et les applications mathématiques aux calculs de suites et de sommes.

À partir de maintenant, dans les exercices, les programmes sont la plupart du temps contenus dans des **fonctions** avec éventuellement des paramètres et une valeur de retour. Cela permet de les ré-utiliser autant qu'on veut en faisant varier un paramètre, mais sans utiliser `input()`.

I Boucles simples

La boucle `for` sert à répéter un bloc d'instructions un certain nombre de fois. Dans sa syntaxe de base, elle prend la forme

```
for i in range(a, b):  
    instructions
```

qui est exactement équivalente à

```
i = a  
while i < b:  
    instructions  
    i = i + 1
```

Autrement dit elle répète en faisant varier l'indice i avec $a \leq i < b$. Nous verrons qu'elle est un peu plus pratique quand on sait à l'avance combien de fois on veut répéter les instructions. Essai :

```
for i in range(1, 5):  
    print(i)
```

produit bien

```
1  
2  
3  
4
```

Un fait remarquable auquel il faudra s'habituer est que **la borne b est exclue**. Il existe aussi `range(n)` qui est exactement équivalent à `range(0, n)`.

À retenir

`range(n)` répète n fois, avec i qui varie entre 0 et $n - 1$ inclus.

Éventuellement, un troisième argument `range(a, b, r)` consiste à faire des sauts de taille r au lieu de 1. Cela correspond, dans la traduction en terme de boucle `while` ci-dessus, à remplacer la ligne `i = i + 1` par `i = i + r`. On ne l'utilisera pas tous les jours.

Éventuellement aussi, le caractère `_` permet de ne pas donner de nom à la variable indice de la boucle. Ainsi `for _ in range(n)` a réellement le sens de « répéter n fois ».

C'est l'heure de démarrer le TP !

```
nom = "... " # insérez votre prénom  
for _ in range(3):  
    print("Au travail", nom)
```

Exercice 4.1

Vous connaissez la chanson

```
1 kilomètre à pieds, ça use, ça use,  
1 kilomètre à pieds, ça use les souliers.  
...
```

Écrire une fonction `kilomètres(n)` qui affiche les paroles pendant n kilomètres.

Une application plus sérieuse des boucles `for` est pour le calcul des suites et des sommes, que nous avons déjà abordé. L'exemple de base est la fonction suivante qui calcule et renvoie le nombre 2^n :

```
def puissance2(n):  
    u = 1  
    for i in range(n):  
        u = 2 * u  
    return u
```

par exemple

```
>>> puissance2(5):  
32
```

Il faut remarquer que :

- La variable u représente la valeur de 2^i en entrant dans la boucle, en particulier c'est 2^0 avant de rentrer dans la boucle.
- La variable u représente 2^{i+1} à la fin de la boucle. L'étape `u = 2 * u` fait « avancer d'un cran » la suite.
- Lors de la dernière itération, $i = n - 1$ et donc à la fin u représente bien 2^n .
- ... d'ailleurs on répète n fois l'opération de multiplier u par 2, en partant de 1, donc le résultat est bien 2^n .

Avertissement

Il est **très important** de se poser ce genre de questions quand on utilise des boucles pour calculer des suites et des sommes, notamment ce qui se passe au début et à la fin. Cela sera source de **nombreux** problèmes et pièges. Une possibilité est d'écrire clairement, en commentaire ou sur son brouillon, une phrase telle que « u représente 2^i en début de boucle ».

Les exercices suivants sont les plus proches possible des questions posées à l'écrit. Pour tester son programme, on pourra aussi afficher les valeurs de la variable u dans la boucle.

Exercice 4.2

Écrire une fonction `suite(n)` qui renvoie le terme u_n de la suite définie par $u_0 = 2$ et $\forall n \in \mathbb{N}, u_{n+1} = 1 + \frac{1}{u_n}$.

Exercice 4.3

La **factorielle** de l'entier $n \geq 1$, notée $n!$, est le produit $1 \times 2 \times \dots \times n$. On convient que pour $n = 0$ c'est 1 (c'est cohérent).

Écrire une fonction `factoriel(n)` qui calcule la factorielle de l'entier n .

Exercice 4.4

La **suite de Fibonacci** est la suite $(F_n)_{n \in \mathbb{N}}$ définie par $F_0 = 0$, $F_1 = 1$ et la relation de récurrence

$$\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

Pour calculer les termes d'une telle suite on a besoin d'une boucle **for** avec *deux* variables : u qui représente la valeur de F_i et v qui représente F_{i+1} .

Écrire une fonction `fibonacci(n)` qui calcule le terme F_n .

Dans le cas des sommes, la variable qui représente le terme de la suite se nomme plutôt **S** et s'appelle **variable accumulatrice**, car elle accumule la somme de plus en plus de termes. Elle est toujours initialisée à 0 (si aucune somme ne se produit, la valeur de retour doit être 0) et dans la boucle elle évolue selon une formule du type $S = S + \dots$ (le nouveau terme à sommer).

Exercice 4.5

Écrire une fonction `somme_cubes(n)` qui calcule et renvoie le résultat de la somme $1^3 + 2^3 + \dots + n^3$.

Bonus : vérifier (avec une boucle) sur les 10 premières valeurs de n que la somme est bien égale à $\frac{n^2(n+1)^2}{4}$.

Exercice 4.6

Écrire une fonction `somme_inverse_factoriel(n)` qui calcule et renvoie le résultat de la somme des $\frac{1}{k!}$ pour $0 \leq k \leq n$, c'est à dire $1 + 1 + \frac{1}{2} + \frac{1}{6} + \dots + \frac{1}{n!}$ (avec $\frac{1}{0!} = 1$ et $\frac{1}{1!} = 1$). Tester la fonction pour des valeurs de plus en plus grandes.

Bonus : pouvez-vous l'écrire *sans* faire appel à la fonction `factoriel()` précédente ?

Remarque. Parfois, on veut calculer les termes de la suite et arrêter la boucle quand une certaine condition est vérifiée, par exemple quand u_n dépasse une certaine valeur fixée à l'avance. Dans ce cas il faut revenir à une boucle **while** — la partie qui calcule les termes de la suite reste la même, et on rajoute à la main l'initialisation de i avant la boucle et l'incrémement $i = i + 1$ dedans (pour être cohérent, l'incrémement *en fin de boucle*). Revoir pour cela les TP précédents et leur correction.

II Boucles doubles

Pour parcourir l'ensemble de toutes les valeurs possibles pour deux indices indépendants i et j , il est nécessaire d'utiliser une **boucle double**, ce qui n'est rien de plus qu'une boucle située dans le corps d'une autre boucle. Mais il est important d'en comprendre précisément le mécanisme.

Exercice 4.7

Tester et comparer les deux programmes suivants :

```
for i in range(3):
    for j in range(3):
        print("i =", i, "j =", j)
```

```
for j in range(3):
    for i in range(3):
        print("i =", i, "j =", j)
```

Que se passe-t-il ?

Observer que ce sont deux ordres naturels pour énumérer le carré d'indices $(i, j) \in \{0, 1, 2\} \times \{0, 1, 2\}$.

$i \setminus j$	0	1	2
0	$i = 0 \ j = 0$	$i = 0 \ j = 1$	$i = 0 \ j = 2$
1	$i = 1 \ j = 0$	$i = 1 \ j = 1$	$i = 1 \ j = 2$
2	$i = 2 \ j = 0$	$i = 2 \ j = 1$	$i = 2 \ j = 2$

Exercice 4.8

On souhaite écrire une fonction (sans arguments ni valeur de retour) qui affiche la table de multiplication, la plus classique, des nombres de 1 à 10.

1. Écrire une fonction `table()` qui affiche la liste de tous les produits $i \times j$ pour i et j entre 1 et 10, produisant par exemple le résultat suivant :

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
...
1 * 10 = 10
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
...
10 * 10 = 100
```

2. On souhaite maintenant formater la table en carré, dans le but de produire le résultat suivant :

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Pour cela on a besoin de savoir que

- Dans la commande `print()`, rajouter l'argument final `end=""` désactive le saut de ligne automatique (en fait, cela remplace le caractère de saut de ligne ajouté automatiquement par la chaîne vide, donc il ne se passe rien) : `print(x, end="")`.
- Et `print()` sans argument effectue un simple saut de ligne.

Écrire cette fonction, qu'on appellera `table_carré()`.

3. Bonus : on souhaite enfin aligner correctement les colonnes, pour obtenir

```
1  2  3  4  5  6  7  8  9 10
2  4  6  8 10 12 14 16 18 20
3  6  9 12 15 18 21 24 27 30
4  8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Pour cela on utilise les chaînes formatées : la syntaxe `print(f"{x:4d}")` affiche le nombre entier `x` en prenant l'espace de 4 caractères, serré à droite. Si `x = 10` la chaîne `f"{x:4d}"` est ainsi remplacée par `" 10"` où le symbole indique un espace (le `f` devant signifie que la chaîne doit être formatée, le `x` est le nom de la variable, et `4d` est l'un des nombreux codes de formatage signifiant ici *entier affiché sur 4 caractères*).

Écrire cette fonction, qu'on appellera `table_carré_joli()`.

Exercice 4.9

Un **triplet pythagoricien** est la donnée de trois nombres $(a, b, c) \in \mathbb{Z}^3$ tels que

$$a^2 + b^2 = c^2$$

On souhaite afficher tous les triplets pythagoriciens. Remarquons que :

- On peut toujours changer l'un des nombres en son opposé et on obtient toujours un triplet pythagoricien. On se limitera donc aux triplets de nombres positifs.
- Il y a des solutions évidentes pour $a = 0$ (avec $b = c$) et pour $b = 0$ (avec $a = c$). On peut donc même chercher les solutions où a et b sont strictement positifs.
- Il peut y avoir une infinité de solutions. Il faut donc fixer un entier N et chercher les solutions (a, b, c) avec $a \leq N, b \leq N, c \leq N$.

Écrire une fonction `pythagore(N)` qui affiche tous les triplets pythagoriciens avec (a, b, c) comme ci-dessus.

Bonus : chaque triplet apparaît deux fois, en échangeant a et b . Pouvez-vous les énumérer de telle façon à ce que chaque triplet n'apparaisse qu'une seule fois ?

III D'autres types d'itération

La boucle `for` fait bien plus que tout cela...

Un **tuple** est la donnée de deux ou plusieurs objets Python, entre parenthèses et séparés par des virgules. Cela forme un nouveau type qui correspond directement à la notion de produit cartésien en mathématiques. Par exemple la variable

```
>>> t = (4, 17)
>>> type(t)
<class 'tuple'>
```

est construite à partir de deux nombres et est de type `tuple`. Il est alors possible de la **recupérer ses valeurs** avec la syntaxe

```
>>> (x, y) = t
>>> x
4
>>> y
17
```

qui va alors créer d'un coup deux nouvelles variables `x` et `y`. Attention, ce n'est pas du tout la même chose que `t = (x, y)`, qui elle crée un tuple à partir de deux variables existantes `x` et `y`. Les composantes d'un tuple peuvent en fait être de tous les types vus précédemment.

Il possible d'**itérer sur les éléments d'un tuple**. Dans le boucle suivante, la variable `x` prend successivement les valeurs des composantes de `t`. Testez-la !

```
t = ("mathématiques", "physique-chimie", "SVT"):
for x in t:
    print("J'aime le cours de", x, "car je suis en BCPST.")
```

Exercice 4.10

Écrire un programme, le plus court possible, qui affiche les valeurs de `somme_inverse_factoriel(n)` de § I pour $n = 1, n = 2, n = 3, n = 5, n = 10$ et $n = 20$.

Un autre intérêt des tuples est de fournir une syntaxe rapide et pratique pour échanger deux variables : c'est tout simplement

```
(y, x) = (x, y)
```

sans avoir besoin d'écrire une variable intermédiaire.

Ils sont aussi faciles à utiliser pour une fonction qui doit renvoyer *deux* valeurs.

Exercice 4.11

On reprend la suite de Fibonacci de l'exercice 4. Écrire une fonction `quotients(n)` qui renvoie le couple $\left(\frac{F_{n+1}}{F_n}, \frac{F_{n+2}}{F_{n+1}}\right)$. Puis tester cette fonction pour tous les n entre 1 et 10. Qu'observe-t-on ?

Enfin nous verrons au TP suivant qu'une boucle `for` est utile pour obtenir uns par uns les caractères d'une chaîne. Le code suivant

```
s = "BCPST"
for x in s:
    print(x)
```

produit le résultat

```
B
C
P
S
T
```

autrement dit la variable `x` prend successivement pour valeurs les caractères de la chaîne `s`. On parle d'**itération sur une chaîne de caractères**.

Exercice 4.12

Écrire une fonction `type_lettres(s)` qui prend en argument une chaîne de caractères `s`, qu'on suppose formée uniquement de lettres minuscules (un mot), et qui affiche pour chacune des lettres le mot "`voyelle`" ou bien "`consonne`".

TP 5

Listes

Les listes constituent le dernier concept absolument fondamental du langage Python que nous étudions dans ce début d'année. Le lien avec les boucles `for`, ainsi qu'avec les calculs de suites et de sommes, apparaîtra rapidement. Puis les listes permettront de poser de nombreuses questions algorithmiques nouvelles.

I Notion de liste

Une liste sert à contenir plusieurs objets, rangés les uns à la suite des autres. Ils sont écrits entre crochets et séparés par des virgules. Les listes sont des types de variables à part entière, qu'on peut mettre dans les variables ou donner en argument à des fonctions. Voici par exemple une liste de cinq nombres impairs :

```
L = [1, 3, 5, 7, 9]
```

Mais les éléments de la liste peuvent être de type quelconque, et même pas forcément tous du même type ! Voici par exemple une liste de courses :

```
courses = ["oeufs", "pain", "riz", "beurre"]
```

I.1 Accès aux éléments

Les éléments d'une liste sont numérotés à partir de 0. À tout moment on peut accéder au i -ème élément de la liste `L` avec la syntaxe `L[i]` et le modifier. Avec les listes ci-dessus :

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> courses[0]
'oeufs'
>>> courses[2]
'riz'
>>> courses[2] = "pâtes"
>>> courses
['oeufs', 'pain', 'pâtes', 'beurre']
```

La fonction `len(L)` donne la longueur de la liste `L`. **Dans une liste de longueur n , les éléments sont numérotés de 0 à $n - 1$ inclus.** En mathématiques on noterait par exemple $(x_0, x_1, \dots, x_{n-1})$ une telle liste. Ce sont les même conventions que pour `range(n)`, et ce sera bien pratique.

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> len(courses)
4
>>> L = [1, 3, 5, 7, 9]
>>> len(L)
5
```

Les indices négatifs correspondent à parcourir la liste en sens inverse.

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> course[-1]
'beurre'
>>> course[-2]
'riz'
```

Si les indices dépassent la longueur de la liste, on obtient une erreur.

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> course[4]
IndexError: list index out of range
```

Cette erreur est extrêmement fréquente : la liste est de longueur 4, donc le dernier élément est `courses[3]`, il n'y a pas d'élément d'indice au-delà de 4...

Enfin la **liste vide** est notée tout simplement `[]` et est de longueur 0. Elle sera loin d'être inutile.

Exercice 5.1

Créez une liste `repas` contenant votre dernier repas, et testez vous-même les syntaxes précédentes.

I.2 Opérations sur les listes

L'opération `+` entre listes s'appelle la **concaténation**. La liste `L + M` est composée de la liste `L` à laquelle est mise bout à bout la liste `M`.

```
>>> ["oeufs", "pain", "riz", "beurre"] + ["fromage", "pommes"]
['oeufs', 'pain', 'riz', 'beurre', 'fromage', 'pommes']
```

Exercice 5.2

Ajoutez à votre liste `repas` de l'exercice 1 une nouvelle liste contenant le repas dont vous rêvez.

Pour un nombre entier `n` il y a aussi une opération de multiplication `*` entre une liste et `n` : le résultat `L * n` est la même chose que `L + L + ... + L` (`n` fois).

```
>>> ["oui", "non"] * 3
['oui', 'non', 'oui', 'non', 'oui', 'non']
```

Exercice 5.3

Quelle est la syntaxe la plus simple pour créer une liste de `n` zéros ?

Nous avons déjà vu ces opérations sur les chaînes de caractères. Revoir le TP 1 : l'opération `+` « colle » deux chaînes de caractères

```
>>> "BCPST" + "1B"
'BCPST1B'
```

et l'accès au i -ème caractère suit les mêmes règles, si `s = "BCPST"` alors `len(s)` est 5, `s[0]` est le caractère seul `B` et `s[4]` est le `T`, qui est aussi `s[-1]` ici.

Une opération encore plus courante consiste à ajouter un élément `x` seul à la fin de la liste `L`. Pour cela la syntaxe est `L.append(x)`. Cette opération ne renvoie pas de valeur (en mode interactif, rien ne s'affichera), mais modifie la liste elle-même :

```
>>> L = ["oeufs", "pain", "riz", "beurre"]
>>> L.append("fromage")
>>> L
['oeufs', 'pain', 'riz', 'beurre', 'fromage']
```

Exercice 5.4

Dans votre liste `repas`, ajoutez un légume puis un fruit, c'est important pour la santé !!!

L'opération « inverse » est `L.pop()` qui supprime le dernier élément de la liste **et le renvoie**. Ainsi on peut récupérer l'élément avec `x = L.pop()` pendant qu'il est supprimé de `L`. Avec la même liste précédente :

```
>>> L
['oeufs', 'pain', 'riz', 'beurre', 'fromage']
>>> x = L.pop()
>>> L
['oeufs', 'pain', 'riz', 'beurre']
>>> x
'fromage'
```

Remarque. Qu'est-ce que c'est que cette syntaxe ? Et pourquoi pas une syntaxe telle que `append(L, x)` ou bien `L = append(L, x)` ?

C'est la première fois que nous la rencontrons vraiment. Disons en première approche que tout se passe comme si *chaque* liste `L` venait avec *sa* propre fonction `append()` qui peut modifier la liste ; pour une autre liste `M` ce sera `M.append(x)`. Remarquez que les autres opérations vues jusque là ne modifiaient pas la liste `L`, alors que **append et pop modifient la liste elle-même**. On parle de **méthodes** plutôt que de fonctions. Dans l'aide interactive `help(list)` on trouve toutes les méthodes de base applicables sur les listes.

Une autre possibilité, qui donne en apparence exactement le même résultat final, est d'écrire

```
L = L + [x]
```

Cependant cela oblige à :

- Créer une liste `[x]` ne contenant qu'un seul élément `x`,
- Concaténer cette liste au bout de `L`,
- Modifier la variable `L` pour que cela devienne ce `L + [x]` qu'on vient de former,

Cela constitue en fait beaucoup plus d'opérations pour l'ordinateur.

À retenir

`append` et `pop` modifient la liste elle-même, alors que `+` et `*` ne sont que des opérations comme les autres, dont le résultat est une nouvelle liste et peut ensuite être mis dans une variable.

Remarque. Quand on les manipule uniquement avec `append` et `pop`, les listes se comportent comme des **pires** — exactement comme une pile de livres sur son bureau. Les opérations de base consistent à poser un élément sur le haut de la pile, ou à récupérer l'élément du dessus. On récupère ainsi les éléments selon l'ordre inverse duquel on les a ajoutés. Si on applique `pop` sur une liste vide, on obtient une erreur `IndexError: pop from empty list` car il n'y a plus rien à enlever.

II Méthode : itérer sur les listes

En pratique, si on reçoit une liste quelconque `L`, on ne sait pas forcément à l'avance quels sont ses éléments. Il est nécessaire d'utiliser une boucle pour « effectuer une opération » sur chaque élément de la liste. On parle de **parcourir la liste**.

La méthode de base s'appelle **itérer sur les indices**. Une liste de longueur n est numérotée de 0 à $n - 1$ et donc une boucle `for i in range(n)` va parcourir tous les indices de la liste, permettant d'accéder à `L[i]` et de faire une opération dessus. Remarquez que la convention de numérotation des listes est bien compatible avec celle de `range`. Exemple :

```
L = ["oeufs", "pain", "riz", "beurre"]
for i in range(len(L)):
    print("indice :", i, "élément :", L[i])
```

produit le résultat

```

indice : 0 élément : oeufs
indice : 1 élément : pain
indice : 2 élément : riz
indice : 3 élément : beurre

```

C'est intéressant quand ces morceaux de programme font partie d'une fonction. Donnons l'exemple suivant qui lit une liste de nombres et affiche le double de chacun.

```

def double(L):
    for i in range(len(L)):
        print(2 * L[i])

```

Essai :

```

>>> L = [1, 3, 5]
>>> double(L)
2
6
10

```

Exercice 5.5

Écrire une fonction `signe(L)` qui prend en argument une liste `L` de nombres et qui affiche pour chacun des éléments le mot "positif", "négatif" ou "nul", selon son signe. On doit par exemple avoir :

```

>>> signe([4, 0, 7, -5])
positif
nul
positif
négatif

```

III Méthode : créer une liste à partir de zéros

Quand on sait à l'avance qu'on veut une liste de n nombres, une méthode intéressante est de commencer par créer une liste de n zéros avec la syntaxe `L = [0] * n`, puis de remplir la liste au fur et à mesure. C'est particulièrement intéressant pour les suites dont chaque terme dépend du ou des précédents.

Un exemple de base est la fonction qui crée la liste des n premières puissances de 2 :

```

def puissances2(n):
    L = [0] * n
    L[0] = 1
    for i in range(1, n):
        L[i] = 2 * L[i-1]
    return L

```

Remarquez comme la formulation ressemble à la définition d'une suite $(u_i)_{0 \leq i < n}$ avec $u_0 = 1$ et $\forall 1 \leq i < n$, $u_i = 2 \times u_{i-1}$. Remarquez bien les bornes de la boucle `for`, puisqu'on veut écrire `L[i-1]` il faut partir de `i` à 1 et pas 0... L'exécution donne bien le résultat voulu, les puissances de 2^0 à 2^4 :

```

>>> puissance2(5)
[1, 2, 4, 8, 16]

```

Exercice 5.6

Écrire une fonction `factoriel(n)` (encore ?), qui renvoie la liste des valeurs $i!$ pour $0 \leq i \leq n$, où $i! = 1 \times 2 \times \dots \times i$, et pour $i = 0$ c'est 1.

Le cas des suites récurrentes d'ordre 2 n'est pas du tout plus difficile : chaque terme de la suite dépend des deux termes précédents, mais précisément la liste sert à se souvenir de *tous* les termes précédents, donc il n'y a pas besoin d'astuce particulière comme au TP précédent.

Exercice 5.7

Écrire une fonction `fibonacci(n)` (*encore ? et ce n'est que le début*) qui renvoie la liste des n premiers termes de la suite de Fibonacci $(F_i)_{i \in \mathbb{N}}$ avec $F_0 = 0$, $F_1 = 1$ et $\forall i \geq 2, F_i = F_{i-1} + F_{i-2}$.

IV Méthode : créer une liste par `append` successifs

Une autre méthode courante pour créer des listes est de démarrer avec une liste vide $L = []$ et d'utiliser des appels à `L.append()` pour ajouter des éléments uns par uns. Cela est intéressant notamment quand on ne connaît pas à l'avance la taille de la liste finale, on étudie un certain problème ou une certaine équation et on rajoute des solutions qu'on trouve au fur et à mesure.

Comme exemple de base, donnons une fonction qui prend en argument une liste L et renvoie une liste P constituée uniquement des termes de L qui sont positifs (donc ici c'est P qui grandit donc on utilise `P.append()`).

```
def garde_positifs(L):
    P = []
    for i in range(len(L)):
        if L[i] >= 0:
            P.append(L[i])
    return P
```

Test :

```
>>> garde_positifs([4, -8, -2, -5, 0, 1, 1, 6, 6, -2, 2, 1])
[4, 0, 1, 1, 6, 6, 2, 1]
```

Exercice 5.8 Équation de Pell-Fermat

On recherche des solutions entières à l'équation suivante :

$$x^2 - 3y^2 = 1, \quad (x, y) \in \mathbb{Z}^2$$

Comme il pourrait y avoir une infinité de solutions (il se peut très bien que x et y soient tous les deux extrêmement grands et que pourtant la différence $x^2 - 3y^2$ soit petite), pour écrire un programme il faut choisir un paramètre N et chercher les solutions (x, y) avec x et y inférieurs à N . De plus, il apparait que si (x, y) est une solution alors on obtient de nouvelles solutions en remplaçant x par $-x$, ou aussi y par $-y$. Bref, on cherche des solutions avec $0 \leq x \leq N$ et $0 \leq y \leq N$.

1. Écrire une fonction `pell_fermat(N)` qui *affiche* les solutions trouvées (x, y) à l'équation de Pell-Fermat avec $0 \leq x \leq N$ et $0 \leq y \leq N$.
2. Améliorer la fonction pour qu'elle renvoie la liste des couples (type `tuple`) de solutions trouvés.

Rappelons que les syntaxes pour accéder au i -ème élément d'une liste, et au i -ème caractère d'une chaîne, sont les mêmes : on peut utiliser une boucle `for` pour parcourir uns par uns tous les caractères d'une chaîne.

Exercice 5.9

Pour un caractère seul `x`, la méthode `x.isupper()` renvoie un booléen `True` si `x` est en majuscule, et `False` sinon.

Écrire un fonction `acronyme(s)` qui prend en argument une chaîne de caractères `s` et qui renvoie la liste de tous les caractères majuscules de `s`.

Exemple :

```
>>> acronyme("Biologie, Chimie, Physique et Sciences de la Terre")
['B', 'C', 'P', 'S', 'T']
```

Bonus : pour une liste `L` de chaînes de caractères, la méthode `"".join(L)` colle tous les termes de la liste en une seule chaîne de caractères (la syntaxe signifie ici « joindre les éléments de `L` autour de la chaîne vide `""` », qu'on pourrait remplacer par n'importe quel caractères de séparation comme `" "` ou `","`). Reprendre la fonction en renvoyant non pas une liste mais une chaîne de caractères.

L'exercice suivant résume bien tout, peut-être à traiter à la toute fin du TP.

Exercice 5.10 (*)

Le **crible d'Eratosthène** est une ancienne méthode pour trouver tous les nombres premiers jusqu'à n . Il fonctionne de la façon suivante (on rappelle que 0 et 1 ne sont pas des nombres premiers) :

- On écrit tous les nombres à la suite, de 2 à n ,
- On barre tous les multiples de 2, sauf 2,
- On barre tous les multiples de 3, sauf 3,
- On avance à 5 (car 4 est barré), puis on barre tous les multiples de 5,
- Et ainsi de suite, on avance au premier nombre non barré (qui est donc premier) et on barre tous ses multiples.

On souhaite s'inspirer de cette méthode pour générer la liste de tous les nombres premiers inférieurs ou égaux à n .

1. Dans un premier temps on modélise le problème avec une liste `L` de booléens `True` ou `False`, où on interprète `L[i] = False` comme « l'entier i est barré » et donc `L[i] = True` comme « non-barré ». Pour bien aligner les indices, on décide que `L[0]` correspond bien au nombre 0 (même si on sait qu'il n'est pas premier ; on peut donc dès le début le mettre à `False`) et de même `L[1]` correspond à 1 qui est aussi `False`. La liste `L` est donc de longueur $n + 1$.

Écrire une fonction `crible(n)` qui renvoie ainsi la liste de booléens correspondant aux nombres premiers inférieurs à n .

Comme les premiers nombres premiers sont 2, 3, 5, 7, 11, la liste résultat doit commencer par

```
[False, False, True, True, False, True, False, True, False, False, ...]
```

2. En déduire une fonction `liste_preiers(n)` qui, à l'aide de la précédente, renvoie la liste de tous les nombres premiers inférieurs ou égaux à n .

V Méthode : listes en compréhension

Une autre méthode pour créer des listes est d'utiliser la syntaxe **en compréhension**, qui est plus proche du langage mathématique. Par exemple ceci crée la liste des carrés des nombres de 1 à 10 :

```
>>> [i**2 for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Cela ressemble vraiment beaucoup beaucoup à $\{i^2 \mid 1 \leq i < 11\}$ n'est-ce pas ? On peut même y rajouter une condition, par exemple pour avoir seulement les carrés des nombres pairs :

```
>>> [i**2 for i in range(1, 11) if i%2 == 0]
[4, 16, 36, 64, 100]
```

Cette syntaxe a en fait de nombreux avantages. Au moins, c'est la méthode la plus simple pour créer une liste dont on aurait une « formule » directe pour $L[i]$ (ce qui n'est pas le cas dans la partie III). Une autre façon standards de créer une liste de n zéros est

```
[0 for _ in range(n)]
```

Exercice 5.11

Produire la liste suivante, en utilisant une syntaxe en compréhension :

```
[1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0]
```

Exercice 5.12

Écrire une fonction `rebours(n)` qui, en utilisant directement une liste en compréhension, renvoie la liste constituée de $[n, n-1, \dots, 1, 0]$ (compte à rebours à partir de n).

VI D'autres opérations sur les listes

VI.1 Tranches

Étant donnée une liste L , les syntaxes suivantes permettent d'obtenir une liste extraite de L appelée **tranche** :

- $L[a:b]$: sélectionne tous les éléments de la liste d'indice entre a et b (b est **exclus**, comme dans `range(a, b)`).
- $L[a:]$: sélectionne tous les éléments à partir de l'indice a .
- $L[:b]$: sélectionne tous les éléments du début jusqu'à l'indice b exclus.

Testons par exemple :

```
>>> L = ['oeufs', 'pain', 'riz', 'beurre', 'fromage', 'pommes']
>>> L[1:4]
['pain', 'riz', 'beurre']
>>> L[2:]
['riz', 'beurre', 'fromage', 'pommes']
```

Ces syntaxes sont compatibles avec les indices négatifs. Par exemple $L[:-1]$ correspond à toute la liste sauf le dernier élément, et $L[-k:]$ correspond aux k derniers éléments de la liste.

Enfin, un dernier paramètre optionnel $L[a:b:r]$ permet de trancher en « sautant de r » au lieu de 1, pour par exemple prendre un terme sur deux. Avec un argument négatif, on part de la fin. Ainsi $L[::-1]$ correspond exactement à la liste L rangée en ordre inverse (départ de la fin, jusqu'au début, en sautant de -1).

Ces opérations fonctionnent exactement de la même manière sur les chaînes de caractères, si on les considère comme des listes de caractères individuels, illustrons-les encore :

```
>>> s = "mathématiques"
>>> len(s)
13
>>> s[0]
'm'
>>> s[-1]
's'
>>> s[:4]
'math'
>>> s[1:-1]
'athématique'
>>> s[::-1]
'meuqitaméhtam'
```

Exercice 5.13

Pouvez-vous prédire ce que renvoie la ligne suivante ?

```
>>> s = "J'aime la biologie, la chimie et les mathématiques"
>>> s[:7] + s[-17:]
```

VI.2 Itération sur les éléments

Pour parcourir une liste, il existe une autre syntaxe qui s'appelle **itérer sur les éléments**. La syntaxe `for x in L` fournit uns par uns les éléments de L dans la variable `x` :

```
L = ["oeufs", "pain", "riz", "beurre"]
for x in L:
    print(x)
```

produit tout simplement

```
oeufs
pain
riz
beurre
```

C'est souvent *plus simple* et *plus élégant*, même si en pratique itérer sur les indices fonctionne toujours.

L'itération sur les éléments fonctionne aussi sur les chaînes de caractères, en fournissant les caractères uns par uns :

```
s = "BCPST"
for x in s:
    print(x)
```

produit la même chose que l'itération sur les indices

```
B
C
P
S
T
```

La boucle `for` en Python fait beaucoup plus de choses que répéter n fois et c'est une possibilité fort intéressante et subtile. On parle d'**objets itérables** pour désigner tous les objets qu'on peut mettre après un `for` et qui sont capables de fournir des éléments uns par uns. Dans le TP précédent nous avons brièvement parlé d'itération sur les éléments d'un tuple.

On peut même combiner cela dans les listes en compréhension. La syntaxe suivante prend une liste L et fabrique une liste M dont les éléments sont exactement les doubles de ceux de L :

```
M = [2*x for x in L]
```

La fonction `garde_positifs(L)` de la partie **IV** peut s'écrire plus simplement avec une seule ligne

```
P = [x for x in L if x >= 0]
```

Exercice 5.14 (*)

Reprendre les autres exercices de la partie **IV** en utilisant uniquement des listes en compréhension.

TP 6

Algorithmes sur les listes

À partir de ce TP nous n'apprenons pas de concepts nouveaux du langage Python lui-même. Tout le cours nécessaire a été accumulé dans les TP précédents et est résumé sur le cours distribué au début d'année.

On propose d'étudier un certain nombre de situations classiques sur les listes : compter les éléments, tester une propriété, chercher un élément. Pour certaines d'entre elles les fonctions existent déjà dans les bibliothèques de base de Python, ou s'obtiennent facilement à partir d'elles, mais **on s'interdit de les utiliser**. On travaille donc essentiellement avec des boucles `for`, l'accès aux indices d'une liste, les conditions booléennes classiques. Les méthodes sont les mêmes pour traiter des listes ou bien des chaînes de caractères.

Enfin ce TP vient avec un fichier pré-rempli à compléter. L'intérêt, en plus du gain de temps, est surtout que le fichier comprend de nombreux **tests**. Dans les exercices qui vont suivre, pour vérifier que sa fonction est correcte, il est nécessaire de la tester avec des valeurs pour lesquelles tout marche bien mais aussi avec des valeurs qui pourraient mettre en échec le programme. **Regarder attentivement les tests proposés et observer les résultats du programme tout en réfléchissant bien !**

I Compter et accumuler

Une situation de base est de **compter** les éléments d'une liste vérifiant une certaine propriété. Pour cela il est nécessaire de déclarer une variable appelée **compteur** (en effet...) initialisée à 0, et qui augmente de 1 à chaque fois. Le modèle de base est la fonction suivante qui compte le nombre de termes positifs (ou nuls) d'une liste de nombres :

```
def compte_positifs(L):
    c = 0
    for i in range(len(L)):
        if L[i] >= 0:
            c = c + 1
    return c
```

Remarquons que l'alignement des blocs d'instructions est crucial. Il indique que l'instruction `c = c + 1` est exécutée uniquement quand la condition juste au dessus `L[i] >= 0` est vérifiée ; et que l'instruction `return c` est exécutée à la toute fin après avoir parcouru *toute* la liste. Ainsi ce programme compte bien, parmi la liste toute entière, le nombre de termes qui sont positifs.

Si aucun terme de la liste n'est positif, alors l'incrément de `c` n'a jamais lieu et à la fin `c` vaut 0 comme au début, ce qui est cohérent.

Exercice 6.1

Écrire une fonction `compte(L, x)` qui prend en argument une liste `L` et un nombre `x` et qui compte combien de fois `x` apparaît dans la liste `L`.

Exercice 6.2

Écrire une fonction `différences(L, M)` qui prend en argument deux listes `L, M`, supposées de même longueur (on ne demande pas que la fonction vérifie cette condition) et compte à combien d'indices les éléments `L[i]` et `M[i]` sont différents.

Par exemple les listes `L = [3, 7, 6, 5, 3]` et `M = [3, 8, 6, 5, 4]` sont différentes à 2 indices, pour $i = 1$ et $i = 4$.

Les chaînes de caractères se manipulent de la même manière : pour une telle chaîne `s`, alors la longueur est `len(s)`, et le i -ème caractère est `s[i]`, numéroté de 0 à $n - 1$. Ainsi une boucle `for` comme les précédentes va parcourir les caractères uns par uns de la chaîne. Un caractère seul s'écrit entre guillemets doubles `"a"`, `"b"`, etc.

Exercice 6.3

Écrire une fonction `compte_voyelles(s)` qui prend en argument une chaîne de caractères `s` et compte le nombre de voyelles (lettres parmi `a, e, i, o, u, y`) dans `s`.

La situation pour sommer les termes d'une liste n'est pas très différente. Ici il n'y a plus de variable compteur mais une **variable accumulatrice**.

Exercice 6.4

Écrire une fonction `somme(L)` qui calcule la somme de tous les termes de la liste `L`.

II Tester

Une autre situation courante est de vouloir vérifier si les termes d'une liste vérifient une certaine propriété. Par exemple on souhaite écrire une fonction prenant en argument une liste `L` et qui renvoie `True` si tous les éléments de `L` sont positifs (ou nuls), et `False` sinon.

Une seule fonction est correcte parmi les propositions ci-dessous.

```
def tous_positifs_1(L):
    for i in range(len(L)):
        if L[i] >= 0:
            return True
        else:
            return False
```

```
def tous_positifs_2(L):
    for i in range(len(L)):
        if L[i] >= 0:
            return True
    return False
```

```
def tous_positifs_3(L):
    for i in range(len(L)):
        if L[i] < 0:
            return False
        else:
            return True
```

```
def tous_positifs_4(L):
    for i in range(len(L)):
        if L[i] < 0:
            return False
    return True
```

Exercice 6.5

Laquelle des fonctions ci-dessus teste bien si tous les éléments de `L` sont positifs ? Observer le code, tester les exemples du fichier et **justifier précisément**.

Exercice 6.6

Écrire une fonction `binaire(m)` qui prend en argument une chaîne de caractères `m` (par exemple `m = "011101011"`), et qui renvoie `True` si `m` est bien composée uniquement de caractères `0` ou `1`, et `False` sinon.

Exercice 6.7

Écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste `L` est rangée par ordre croissant et `False` sinon.

À retenir

Quand elle est rencontrée, l'instruction `return` arrête la fonction, même dans une boucle. Mais si la propriété à tester dépend aussi de la suite de la liste, on ne peut pas se contenter de s'arrêter là, et on ne peut pas non plus dire « sinon, continuer à tester »... La bonne façon de faire est alors de **s'arrêter si la condition contraire est vérifiée**, et sinon, le `return True` se situe **à la fin et en dehors de la boucle** : si on est arrivé jusque là, c'est qu'on ne s'est pas arrêté avant, donc que la condition fautive n'a pas été rencontrée, donc que c'est vrai !

III Chercher

On souhaite maintenant écrire une fonction `cherche(L, x)` qui prend en argument une liste `L` et un objet `x` et cherche l'élément `x` dans la liste `L`. Encore une fois, il faut parcourir la liste ; le démarrage est donc nécessairement

```
def cherche(L, x):
    for i in range(len(L)):
        if ... :
            ... ..
            ... ..
```

mais ensuite... On fait les remarques suivantes :

- On peut s'intéresser soit à l'élément lui-même, soit à son indice dans la liste. Ici, on veut son indice (la variable `i` telle que `L[i]` soit égal à `x`).
- L'élément `x` peut apparaître plusieurs fois dans la liste. Mais si on arrête la fonction **dès que** `x` est trouvé, alors on obtiendra l'indice de **la première** apparition de `x` dans la liste. Au contraire, si `x` n'apparaît pas du tout alors il faut bien aller au bout de la liste pour le savoir...
- Enfin, contrairement à la situation « compter », il n'y a pas de bonne valeur cohérente à renvoyer si `x` n'est pas dans la liste. On propose de renvoyer l'objet spécial `None` qui indique l'absence de valeur.

Exercice 6.8

Compléter la fonction ci-dessus pour que `cherche(L, x)` renvoie le premier indice de la liste où `x` apparaît, et `None` s'il n'apparaît pas.

Exercice 6.9

Écrire une fonction `premier_negatif(L)` qui renvoie le premier élément de `L` qui est strictement négatif (l'élément, pas son indice), et `None` s'il n'y a pas de tel élément.

Exercice 6.10

Écrire une fonction `indice_différents(s, t)` qui prend en argument deux chaînes de caractères, supposées de même longueur, et qui renvoie le premier indice auquel les chaînes diffèrent, et `None` si elles sont égales. Par exemple, les chaînes `s = "ACGTGATAA"` et `t = "ACGTCATTA"` sont de même longueur 9 et diffèrent aux indices 4 (`s[4] = "G"` et `t[4] = "C"`) et 7 (`s[7] = "A"` et `t[7] = "T"`) donc la fonction doit renvoyer 4.

À retenir

Dans d'autres situations, on peut tout à fait quitter la fonction dès qu'on a trouvé ce qu'on voulait, c'est donc une bonne idée d'avoir une instruction `return` qui est bel et bien dans la boucle (et même, il ne sert à rien de continuer la boucle !) Éventuellement, en fin de fonction et en dehors de la boucle, on traite le cas où on n'a pas trouvé ce qu'on voulait.

IV D'autres exercices

Exercice 6.11

On suppose que la liste `L` ne contient que des nombres entiers entre 0 et 9. Dans ce cas, on souhaite compter combien de fois apparaît **chaque** chiffre, en renvoyant une liste `C` de longueur 10 telle que `C[x]` donne le nombre de fois où le chiffre `x` apparaît dans `L`. Si on s'y prend bien, on peut le faire en parcourant la liste une seule fois, au lieu d'appeler 10 fois une fonction pour compter...

Écrire cette fonction, qu'on appellera `compte_tout(L)`.

Exercice 6.12

On considère qu'un mot de passe valide sera formé uniquement des caractères parmi ceux-ci :

"abcdefghijklmnopqrstuvwxyz0123456789"

1. Écrire une fonction `caractère_valide(x)` qui teste si `x` est un caractère valide ou non.
2. En déduire une fonction `motdepasse_valide(m)` qui teste si `m`, une chaîne de caractères, représente un mot de passe valide.
3. Bonus : écrire une fonction `motdepasse_fort(m)` qui teste si `m` est valide et contient au moins une lettre et un chiffre.

Le plus efficacement possible.

Exercice 6.13

Écrire une fonction `est_monotone(L)` qui renvoie `True` si la liste `L` est soit croissante soit décroissante, et `False` sinon.

Sans écrire séparément des fonctions annexes `est_croissante(L)` et `est_decroissante(L)`.

Exercice 6.14 (*)

On considère des listes constituées uniquement de nombres `0` et `1` et on souhaite compter le nombre blocs de `1` consécutifs. Par exemple pour

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

on compte 4 blocs de `1`, ayant pour tailles respectives 2, 3, 1, 2.

Écrire la fonction `compte_blocs(L)` qui prend en argument une telle liste et renvoie le nombre de blocs.

Attention à ce qu'elle fonctionne correctement dans tous les cas, que les blocs soient calés au début de la liste ou à la fin ou pas du tout.

Exercice 6.15 (*)

Une **permutation de longueur n** est une liste de longueur n où chacun des nombres de 0 à $n - 1$ apparaît exactement une fois. Par exemple `[3, 1, 0, 2]` est bien une permutation de longueur 4.

1. Pourquoi suffit-il que chacun de ces nombres apparaisse *au moins* une fois ? Ou bien *au plus* une fois ?
2. Écrire une fonction `appartient(L, x)` qui renvoie `True` si le nombre `x` est présent dans la liste `L` et `False` sinon.
3. En utilisant la fonction précédente, écrire une fonction `est_permutation(L)` qui renvoie `True` si `L` est bien une permutation, et `False` sinon.

Une autre possibilité qui est *plus rapide* mais nécessite *plus de mémoire* est la suivante. Pour tester si la liste `L` est bien une permutation, on crée une liste de booléens `M` de même taille que `L`, et on parcourt une seule fois `L`, mais on « coche » les nombres qu'on a vus. Ainsi `M[x] = True` est à interpréter comme « `x` est bien présent dans `L` » alors que `M[x] = False` signifie que `x` n'a pas encore été rencontré.

4. En utilisant cette méthode, écrire une fonction `est_permutation_2(L)`.

Pour ceux qui ont fini trop vite :

5. Écrire une fonction `permutations(n)` qui renvoie la liste de toutes les permutations de longueur n (une liste de listes !)

TP 7

Révisions et consolidation 1

Exercice 7.1 Solide sur les bases

- Écrire une fonction `pH(x)` qui prend en argument un nombre x (on suppose que x est entre 0 et 14 et représente bien le potentiel hydrogène d'une solution) et affiche le mot « acide », « basique » ou « neutre », en fonction du pH. Éventuellement, la fonction affiche « invalide » si x n'est pas dans $[0, 14]$.
- Soit la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 1$ et $\forall n \in \mathbb{N}, u_{n+1} = \frac{8}{2 + u_n}$. Écrire des fonctions, indépendantes l'une de l'autre :
 - `u_suite(n)` : renvoie le terme u_n ,
 - `u_liste(n)` : renvoie la liste des n premiers termes de la suite.
- Écrire une fonction `somme(n)` qui prend en argument un nombre entier n et qui renvoie la valeur de la somme $\sum_{k=1}^n \frac{1}{k^3}$, c'est-à-dire $1 + \frac{1}{2^3} + \frac{1}{3^3} + \dots + \frac{1}{n^3}$, et tester pour des valeurs de n de plus en plus grandes.

Exercice 7.2 Suites

La **suite de Tribonacci** est la suite $(T_n)_{n \in \mathbb{N}}$ définie par

$$T_0 = 0, T_1 = 1, T_2 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}, T_{n+3} = T_{n+2} + T_{n+1} + T_n$$

Écrire séparément les fonctions (sans que l'une ne fasse appel à l'autre) :

- `tribonacci(n)` : calcule le terme T_n .
- `tribonacci_liste(n)` : renvoie la liste des n premiers termes de la suite.

Pour tester on pourra vérifier que les premiers termes de la suite sont :

n	0	1	2	3	4	5	6	7	8	9	10
T_n	0	1	1	2	4	7	13	24	44	81	149

Exercice 7.3 Modélisation

On représente un nombre complexe par une liste de longueur 2 formée de sa partie réelle et de sa partie imaginaire. Par exemple le nombre complexe $z = 3 - 4i$ correspondra à la variable $\mathbf{z} = [3, -4]$. Un nombre réel a est identifié avec la liste $[a, 0]$, et le nombre i à $[0, 1]$.

- Quelle syntaxe permet d'obtenir la partie réelle du nombre représenté par la liste \mathbf{z} ? Et la partie imaginaire ?
- Écrire une fonction `somme(z, w)` qui prend en argument deux listes \mathbf{z}, \mathbf{w} , représentant des nombres complexes z, w , et qui renvoie une liste représentant la somme $z + w$.

Par exemple, l'appel `somme([1, 3], [4, -5])` doit renvoyer la liste `[5, -2]`, ce qui correspond au fait que si $z = 1 + 3i$ et $w = 4 - 5i$ alors $z + w = 5 - 2i$.

- Écrire une fonction `produit(z, w)` qui prend en argument deux listes \mathbf{z}, \mathbf{w} , représentant des nombres complexes z, w , et qui renvoie une liste représentant le produit de nombres complexes $z \times w$.
- En utilisant la fonction précédente, écrire une fonction `puissance(z, n)` qui prend en argument une liste \mathbf{z} représentant un nombre complexe z , et un entier n (supposé positif), et renvoie une liste représentant le nombre complexe z^n .

On pourra tester avec les puissances successives de $z = 1 + 2i$, qui sont :

n	0	1	2	3	4	5	6
$(1 + 2i)^n$	1	$1 + 2i$	$-3 + 4i$	$-11 - 2i$	$-7 - 24i$	$41 - 38i$	$117 + 44i$

Exercice 7.4 *Compter*

On souhaite se donner un nombre réel $r \geq 0$ et compter le nombre de couples d'entiers $(n, m) \in \mathbb{Z}^2$ tels que $n^2 + m^2 \leq r^2$. Ce sont les points à coordonnées entières situés à l'intérieur du cercle de rayon r .

1. Pourquoi peut-on supposer $-r \leq n \leq r$ et $-r \leq m \leq r$?
2. Écrire une fonction `compte_points(r)` qui prend en argument un nombre entier r supposé positif et qui compte le nombre de tels couples.
3. Modifier la fonction pour renvoyer non pas le nombre N de points mais le quotient N/r^2 , et tester avec des valeurs de r de plus en plus grandes, par exemple 10, 100, 1000 (puis augmenter progressivement par multiples de 1000, sans dépasser 10 000). Qu'en pensez-vous ?

Exercice 7.5 *Tests sur les listes*

Toutes les fonctions de cet exercice prennent en argument une liste L , qu'on suppose constituée uniquement de nombres entiers.

1. Écrire une fonction `tous_chiffres(L)` qui renvoie `True` si tous les nombres dans L sont entre 0 et 9 (au sens large), et `False` sinon.
2. Écrire une fonction `sont_consécutifs(L)` qui renvoie `True` si L est constituée d'une suite de nombres consécutifs (par exemple $L = [13, 14, 15, 16]$) et `False` sinon.
3. Écrire une fonction `est_symétrique(L)` qui renvoie `True` si la liste L peut se lire aussi bien à l'envers qu'à l'endroit, par exemple $L = [4, 5, 8, 2, 8, 5, 4]$.

Exercice 7.6 *Création de listes*

1. Pour une liste L , la *liste miroir* de L est la liste rangée en ordre inverse. Par exemple pour $L = [3, 7, 2, 8]$ c'est la liste $[8, 2, 7, 3]$. Écrire une fonction `miroir(L)` qui renvoie la liste miroir de L .
2. Pour une liste L de longueur n , la *liste des sommes cumulées* de L est la liste C de longueur n où $C[i]$ est la somme de tous les termes de L d'indice inférieur à i . Par exemple la liste des sommes cumulées de $[2, 4, 3, 5, 3, 5]$ est $[2, 6, 9, 14, 17, 22]$. Remarquez que le premier terme est toujours $L[0]$ et le dernier est la somme de tous les termes de L .

Écrire une fonction `sommes_cumul(L)` qui renvoie la liste des sommes cumulées de L .

3. Écrire une fonction `somme(L, M)` qui prend en argument deux listes supposées de même longueur et qui renvoie la liste de leurs sommes terme à terme (la liste des $L[i] + M[i]$). Par exemple la somme de $L = [3, 7, 1]$ et $M = [2, 8, 1]$ est $[5, 15, 2]$.

Exercice 7.7 (*) *Alphabet*

On donne la chaîne de caractères suivante : `alphabet = "abcdefghijklmnopqrstuvwxyz"`. Ainsi on considère que la lettre `a` est le caractère numéro 0 de l'alphabet, et `z` est le caractère numéro 25.

1. Écrire une fonction `numero(x)` qui prend en argument un caractère seul `x` (on suppose que c'est l'une des 26 lettres de l'alphabet ci-dessus : pas de majuscule, pas d'accents) et qui renvoie le numéro de `x` en tant que lettre de l'alphabet. Si `x` n'est pas une lettre de l'alphabet on pourra renvoyer l'objet spécial `None`.
2. Écrire une fonction `compte_lettres(s)` qui prend en argument une chaîne de caractères `s` et qui renvoie une liste `C` de longueur 26, où `C[x]` indique combien de fois apparaît la lettre numéro `x` dans la chaîne `s`.

L'une des méthodes les plus anciennes pour coder un texte en un message secret s'appelle *codage de César*, utilisée effectivement par Jules César, et consiste à remplacer chaque lettre d'un texte par celle trois lettres plus loin dans l'alphabet. Ainsi `a` est remplacé par `d`, `b` est remplacé par `e`, etc, et `x` est remplacé par `a`, `y` par `b` et enfin `z` par `c`.

3. Écrire une fonction `code_caractere(x)` qui prend en argument un caractère `x` et qui renvoie le caractère codé par ce procédé.
4. Écrire une fonction `code(s)` qui prend en argument une chaîne de caractères `s` et renvoie la chaîne ainsi codée. On pourra pour cela passer par la création d'une *liste* des caractères codés. Après avoir créé une telle liste `L`, renvoyer non pas `L` mais `"".join(L)` qui colle tous les caractères en une seule chaîne.

On pourra dans un premier temps supposer que `s` contient seulement des lettres de l'alphabet parmi ces 26 là — mais on pourra lever cette restriction en ne « codant pas » les autres caractères (notamment les espaces).

TP 8

Tri

Dans ce TP nous nous intéressons au problème du tri des listes. Il s'agit tout simplement de se donner une liste de nombres et d'étudier différentes méthodes pour les ranger par ordre croissant, en échangeant des éléments entre eux. Par exemple, trier la liste $L = [3, 4, 3, 4, 1, 1, 9, 3]$ doit donner $[1, 1, 3, 3, 3, 4, 4, 9]$.

I Préliminaires

Les fonctions de cette section ne seront pas utilisées tel quel par la suite. Il s'agit cependant d'un échauffement et de bases à bien comprendre.

Le but final est d'aboutir à une liste triée, ce qui est la même chose qu'une liste rangée par ordre croissant.

Exercice 8.1 Révisions

Écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste L est bien rangée en ordre croissant, et `False` sinon.

Puis nous aurons besoin de diverses variantes pour obtenir le maximum d'une liste L — ou bien pour obtenir l'indice i tel que le maximum de L est $L[i]$. L'idée est de parcourir la liste dans l'ordre en maintenant en mémoire une variable M qui contient le maximum de la liste « jusqu'à là ». Attention car le maximum d'une liste vide n'est pas défini, vraiment pas !

Exercice 8.2

Écrire une fonction `maximum(L)` qui renvoie le maximum de la liste L .

Il est fort intéressant de comparer le programme avec le théorème ci-dessous et surtout avec sa preuve.

Théorème

Tout sous-ensemble fini et non-vide $A \subset \mathbb{R}$ admet un maximum.

Démonstration. Démontrons par récurrence sur n la proposition $\mathcal{P}(n)$: « tout sous-ensemble fini à n éléments $A \subset \mathbb{R}$ admet un maximum ». Comme la partie A ne peut pas être vide, la récurrence porte sur $n \in \mathbb{N}^*$.

- Initialisation : pour $n = 1$, soit A une partie de \mathbb{R} à un seul élément, alors on peut écrire $A = \{x_1\}$ avec $x_1 \in \mathbb{R}$ et x_1 est le maximum de A .
- Hérité : soit $n \in \mathbb{N}^*$, supposons $\mathcal{P}(n)$. Montrons $\mathcal{P}(n + 1)$. Soit une partie $A \subset \mathbb{R}$ à $n + 1$ éléments, écrivons $A = \{x_1, \dots, x_n, x_{n+1}\}$. Formons la partie $B = \{x_1, \dots, x_n\}$, c'est une partie de \mathbb{R} non-vide à n éléments. On applique alors l'hypothèse de récurrence à B , qui admet donc un maximum M , qui est un des éléments parmi x_1, \dots, x_n . Mais ensuite :
 - Ou bien $x_{n+1} \geq M$. Alors x_{n+1} est plus grand que lui-même et que tous les x_1, \dots, x_n , donc x_{n+1} est le maximum de A .
 - Ou bien $x_{n+1} \leq M$, et donc M est plus grand que tous les x_1, \dots, x_n et aussi que x_{n+1} et donc M est le maximum de A .

En conclusion la partie A admet bien un maximum, et ceci démontre $\mathcal{P}(n + 1)$. □

Pour la suite il est important de comprendre qu'une fonction qui reçoit une liste en argument va modifier la liste, comme expliqué dans l'annexe § III. La fonction suivante n'est qu'un petit échauffement.

Exercice 8.3

Écrire une fonction `echange(L, i, j)` qui échange les éléments de L d'indices i et j .

II Les algorithmes de tri

Nous démarrons maintenant les algorithmes de tri. Le but est, à chaque fois, d'écrire une fonction qui prend comme argument une liste de nombres de longueur n et trie la liste par ordre croissant. La fonction utilise diverses comparaisons entre des éléments d'indices i et j et éventuellement les échange, et à la fin la liste doit être triée. En répétant les comparaisons et les opérations plusieurs fois, dans une boucle voire une double boucle.

On n'utilise donc pas les fonctions Python pour insérer ou supprimer des éléments en milieu de liste ainsi que, évidemment, les fonctions déjà prêtes de tri. On n'a même pas besoin de `append` et de `pop`, ni des tranches. C'est un tri « sur place ». Si nous avons à trier des livres sur une étagère, cela signifie que nous pouvons uniquement retirer deux livres et les échanger de place, mais que nous ne pouvons pas sortir tous les livres puis les reposer dans l'ordre, ni pousser d'un coup tout un étage pour le décaler.

Il est encouragé de rajouter des instructions `print(L)` dans les boucles, pour voir la liste évoluer au fur et à mesure du tri et pour déboguer son programme.

II.1 Tri à bulles

C'est le plus simple des tris à programmer. L'algorithme du tri à bulles se décrit ainsi :

- On parcourt la liste, tout simplement, dans l'ordre, du début jusqu'à la fin.
- Si deux éléments **consécutifs** ne sont pas rangés dans l'ordre croissant, on les échange.
- ... On répète ce processus n fois (en fait $n - 1$ fois suffisent : pourquoi ?) en repartant à chaque fois du tout début de la liste.

L'idée est que les plus grands éléments remontent peu à peu à la fin de la liste, comme des bulles qui remontent à la surface de l'eau. La fin de la liste apparaît « de plus en plus triée ».

Exercice 8.4

Écrire la fonction `tri_bulle(L)`.

II.2 Tri par sélection

Il s'agit du deuxième plus simple des tris, et il est important d'avoir bien compris la fonction `maximum`.

- On cherche l'**indice** du minimum de L , et par un échange on place le minimum à l'indice 0.
- Puis on cherche l'indice du minimum de la liste, à partir de l'indice 1 (ce sera donc le deuxième plus petit), et de même par un échange on le place à l'indice 1.
- ... On répète le procédé jusqu'à arriver à la fin de la liste.

Ainsi, on **sélectionne** directement les éléments, uns par uns, pour les mettre à leur place, en partant du plus petit. Après i étapes de l'algorithme, les i premiers éléments sont donc exactement ceux de la liste triée.

Exercice 8.5

Écrire la fonction `tri_selection(L)`.

II.3 Tri par insertion

C'est celui qu'on fait le plus naturellement, par exemple quand on trie un jeu de cartes, en prenant les éléments éventuellement dans le désordre et en les **insérant** uns par uns chacun à leur place. Cependant, malgré le nom, nous écrivons cette fonction sans utiliser des insertions dans une liste, mais chaque nouvel élément rencontré sera amené à sa place par une suite d'échanges vers sa gauche. Après i étapes, les i premiers éléments de la liste sont rangés en ordre croissant.

L'algorithme peut se décrire ainsi :

- On va répéter n fois l'insertion. Lors du i -ème passage, les i premiers éléments de la liste (ceux d'indice entre 0 et $i - 1$) seront triés, et on s'intéresse à l'élément x d'indice i (celui qui vient juste après).
- On insère alors x à sa place de telle façon à ce que les $i + 1$ premiers éléments de la liste soient triés. Pour cela, on échange successivement x avec l'élément qui est immédiatement avant lui, tant que c'est x qui est le

plus petit des deux. Sinon, on est arrivé au point où x est bien inséré à sa place. Il s'agit donc d'une boucle descendante, à partir de l'indice i de x . Toutes les opérations d'échange reviennent aussi à « décaler » les éléments un cran vers la droite, tant qu'ils sont plus grands que x .

Exercice 8.6

Écrire la fonction `tri_insertion(L)`.

II.4 Tri stupide

Étudions ce tri pour s'amuser uniquement, car il est totalement inefficace en pratique :

- On choisit deux indices de la liste au hasard i et j .
- Si les éléments d'indices i et j ne sont pas rangés dans le bon ordre, alors on les échange.
- On répète *tant que* la liste n'est pas triée.

Pour choisir les indices au hasard, on a besoin de la bibliothèque `random` et de sa fonction `randint(a, b)` — quoique, ici `randrange(n)` est plus pertinent, qui donne un nombre au hasard tout comme `randint` mais avec une syntaxe similaire à `range`. Ainsi `randrange(n)`, aussi `randrange(0, n)`, est la même chose que `randint(0, n-1)`.

Exercice 8.7

Écrire la fonction `tri_stupide(L)`.

Observer aussi comme la fonction est nettement de plus en plus lente quand la longueur de la liste augmente, à un point où elle ne se termine même plus en un temps raisonnable. Quand la liste est très grande et presque triée, il devient très improbable de tomber pile sur deux éléments restant à échanger, et la boucle continue de nombreuses fois en attendant.

II.5 Tri par comptage

Il ne s'agit pas d'un tri au même sens que les précédents, mais c'est une technique bien utile.

On suppose qu'on a une liste L contenant **uniquement des nombres entiers** entre 0 et un certain entier N (au sens large, bornes incluses). Au lieu de trier directement, nous allons d'abord compter combien de fois apparaît chaque nombre, puis nous allons reconstruire la liste. Après coup, on n'a pas besoin de se donner N car il s'agit du maximum de la liste.

Exercice 8.8

1. Écrire la fonction `compte(L, N)` qui prend en argument une liste L et un entier N , en supposant que les éléments x de L vérifient tous $0 \leq x \leq N$, et qui renvoie une liste C où $C[x]$ est le nombre d'éléments de L qui sont égaux à x .
2. En déduire une fonction `tri_comptage(L)` qui utilise la fonction précédente et reconstruit une liste M qui contient les mêmes éléments que L mais triée.

Cette méthode est notamment intéressante pour calculer la médiane d'une liste. Définissons la **médiane** d'une liste L comme le *plus petit* élément m de la liste tel qu'*au moins la moitié* des éléments de L soient *inférieurs ou égaux* à m . Ces précisions sont subtiles mais permettent d'avoir une médiane bien définie, que le nombre d'éléments soit pair ou non. Remarquons que lorsque la liste L est déjà triée, la médiane est immédiate à trouver... Remarquons aussi que la somme des éléments de C est égal à la longueur de L .

Exercice 8.9

Écrire la fonction `mediane(L)` qui calcule la médiane de la liste L , avec la définition ci-dessus :

1. Une première fois en triant la liste.
2. Une deuxième fois en utilisant `compte` mais sans trier la liste.

III Annexe : le problème des listes et des références

Il est important d'être conscient du problème suivant lorsqu'on manipule des listes. Comparons les trois morceaux de programmes :

```
>>> x = 3
>>> y = x
>>> x = 12
>>> print(y)
3
```

et

```
>>> L = [1, 3, 5]
>>> M = L
>>> L[0] = 12
>>> print(M)
[12, 3, 5]
```

et aussi

```
>>> s = "Bonjour"
>>> t = s
>>> s[0] = "b"
TypeError: 'str' object does not support item assignment
```

Que se passe-t-il ?

- Dans le premier cas, les variables de type **int contiennent** une valeur, et lors de l'affectation $y = x$ la valeur de x est **copiée** dans y . Une modification ultérieure de x ne modifiera en rien y .
- Dans le deuxième cas, lors de l'affectation $M = L$, ce n'est pas toute la liste qui est copiée mais un « moyen d'accéder à la liste » qu'on appellera une **référence** à L . Les noms de variables L et M sont donc des références à la **même** liste en mémoire et toute modification de l'une affecte l'autre. Recopier une liste en mémoire peut être une opération coûteuse car la liste peut contenir des milliers d'éléments, et donc il faut éviter de la recopier inutilement !
- Enfin dans le troisième cas les chaînes de caractères sont des objets **immuables** que l'on ne peut de toute façon pas modifier. Cela fait donc peu de différence de savoir si deux variables sont des références à la même chaîne ou bien deux chaînes différentes.

Ce phénomène se produit aussi lorsqu'on passe une liste en argument à une fonction :

```
def f(L):
    L[0] = 12
```

puis

```
>>> L = [1, 3, 5]
>>> f(L)
>>> print(L)
[12, 3, 5]
```

Cela peut être souhaité ou bien peut être embêtant. Les fonctions `append` et `pop` vont aussi modifier la liste passée en argument :

```
def f(L):
    L.append(12)
```

puis

```
>>> L = [1, 3, 5]
>>> f(L)
>>> print(L)
[1, 3, 5, 12]
```

On parle aussi d'**effets de bords** — la fonction a des effets en dehors de la manipulation de ses variables locales. Le mot-clé `is` permet de savoir si deux variables sont des références au même objet Python.

```
>>> L = [1, 3, 5]
>>> M = L
>>> M == L
True
>>> M is L
True
```

mais

```
>>> L = [1, 3, 5]
>>> M = [1, 3, 5]
>>> M == L
True
>>> M is L
False
```

Dans ce dernier cas les listes L et M sont **structurellement égales** (elles contiennent les mêmes éléments et sont donc des objets égaux au sens mathématique usuel du terme) mais ne sont pas des références à un même objet liste. Ainsi une modification de l'une ne va pas affecter l'autre.

Il est toujours possible d'obtenir une copie « fraîche » d'une liste, qui ne soit pas une référence mais contienne les même éléments, avec la méthode `L.copy()`.

```
>>> L = [1, 3, 5]
>>> M = L.copy()
>>> M
[1, 3, 5]
>>> M is L
False
```

Encore une fois, ce `False` nous indique qu'on peut sereinement modifier soit L soit M sans affecter l'autre.

Retenons que :

- Les opérations `+`, `*`, et les tranches, créent à chaque fois des listes nouvelles en copiant les éléments.
- Les affectations `L[i] = ...`, et les opérations `append` et `pop`, modifient directement les listes, même à travers des références. Attention aux effets de bords quand on les utilise sur des listes passées en argument à des fonctions !
- L'opération `M = L.copy()` permet de créer une nouvelle liste M, et non pas une référence à L, en copiant les éléments.

TP 9

Texte et mots

Dans ce TP nous étudions trois problèmes autour de la recherche de mots dans un texte.

Précisons d'abord que durant tout le TP, un « texte » désigne une chaîne de caractères quelconque qu'on notera s . Un « mot » désigne simplement une sous-chaîne de caractères, c'est-à-dire une chaîne m telle que les caractères consécutifs de m se retrouvent tels quels consécutivement dans s .

Par exemple dans $s = \text{"abracadabra"}$ on trouve les mots "cad" qui apparaît à partir de la position 4 (la lettre c est en position 4 dans s , qui est numérotée à partir de 0) ou bien "abra" qui apparaît deux fois, à partir des positions 0 ainsi que 7. Ainsi les programmes s'appliquent à rechercher un mot dans un texte en français (au sens habituel) mais aussi à rechercher un motif dans une séquence ADN. Par exemple dans la séquence $s = \text{"TTAATGCAATAAC"}$ on peut vouloir rechercher le motif "AAT" , qui apparaît deux fois, ou "ATT" qui n'apparaît pas.

Dans la partie **I**, on s'intéresse simplement à la recherche d'un mot dans un texte. On peut utiliser pour cela le fichier joint `livre.txt` qui contient l'intégralité du livre absolument passionnant *Le Rouge et le Noir* de Stendhal, pour y chercher et trouver des vrais mots. Ce livre étant tombé dans le « domaine public », chacun a le droit de le télécharger et de l'utiliser. Dans la partie **II**, on s'intéresse plutôt à la recherche d'un mot dans un dictionnaire, éventuellement avec l'objectif de corriger des erreurs d'orthographe. On peut alors utiliser le fichier joint `dictionnaire.txt` qui contient une liste de plus de 600 000 mots issue du logiciel libre de correction orthographique GNU Aspell. Enfin dans la partie **III** on étudie un algorithme un peu plus efficace pour la recherche de sous-séquences d'ADN.

I Recherche de mots dans un texte

Il s'agit dans cette partie de rechercher un mot, formé de plusieurs caractères consécutifs, dans un texte, éventuellement très long.

Les fonctions suivantes ne seront pas utilisées telles quelles, mais constituent des révisions.

Exercice 9.1 *Mise en jambe*

Un pré-requis indispensable est de savoir chercher un caractère tout seul.

1. Écrire une fonction `cherche_caractere(s, x)` qui prend en argument une chaîne de caractères s et un caractère seul x , affiche tous les indices auxquels le caractère x apparaît dans s .
2. Améliorer la fonction pour écrire la fonction `compte_caractere(s, x)` qui renvoie le nombre d'apparitions du caractère x dans s .

L'algorithme que nous étudions consiste en une sorte de « fenêtre glissante » qui tente de faire correspondre le mot m à chacune des positions possibles dans le texte s .

0	1	2	3	4	5	6	7	8	9	10	11	12
T	T	A	G	T	A	T	A	A	T	G	A	C
A	T	G										
	A	T	G									
		A	T	G								
			A	T	G							
				A	T	G						
					A	T	G					
						A	T	G				
							A	T	G			
								A	T	G		
									A	T	G	
										A	T	G

Sur cet exemple, on cherche le mot $m = \text{"ATG"}$ dans $s = \text{"TTAGTATAATGAC"}$ et on le trouve effectivement une fois, démarrant à la position 8 de s .

Chaque case remplie représente une comparaison effectuée entre une lettre de `m` et une lettre de `s`. Vert, la comparaison a réussi, et rouge, elle a échoué. Trois cases vertes consécutives et c'est le jackpot ! Dès qu'une case est rouge, alors on descend d'une ligne (on fait « glisser la fenêtre ») et on ré-essaie.

On étudie alors le cas général. On se donne une chaîne de caractère `s` et un mot `m` à chercher dedans. Remarquons déjà que la longueur de `m` est inférieure à celle de `s`, sinon le mot ne peut pas apparaître... On souhaite écrire une fonction `le_mot_est_ici(s, m, i)` qui prend en argument un indice `i` de la chaîne `s`, et qui va renvoyer `True` si le mot `m` est bien présent dans `s` à partir de cet indice, c'est à dire si `m[0]` est égal à `s[i]`, et `m[1]` est égal à `s[i+1]`, etc. Dans notre exemple ci-dessus elle doit renvoyer `True` pour `i = 8` uniquement et `False` sinon. Attention à ne pas se mélanger dans les indices !

Exercice 9.2

1. Tout d'abord il y a une contrainte entre les longueurs des chaînes de caractères pour laquelle nous sommes sûrs que le mot n'est pas ici ! Par exemple, un mot `m` de deux lettres ne peut pas démarrer sur le dernier indice de `s`. Un mot de trois lettres peut-il démarrer sur la dernière ou l'avant-dernière lettre de `s` ? Pouvez-vous donner la relation exacte entre les longueurs `n` de `s`, `p` de `m` et l'indice `i` ?
2. Écrire la fonction `le_mot_est_ici(s, m, i)`.

Cela permet de répondre, déjà, à la question de recherche de mot.

Exercice 9.3

1. En utilisant la fonction précédente, écrire une fonction `cherche_mot(s, m)` qui cherche à tous les indices possibles `i` de `s` si le mot `m` démarre bien à cet indice. La fonction affiche les indices `i` où on trouve le mot.
2. Puis écrire une fonction `compte_mot(s, m)` qui compte combien de fois le mot apparaît.

Pour l'utiliser sur l'exemple du livre, une variante intéressant est d'afficher le mot trouvé *et un petit peu plus*, par exemple une dizaine de caractères précédents et suivants. À cette fin et pour les parties suivantes, on rappelle la syntaxe de tranche : `s[a:b]` sélectionne le mot extrait entre les indices `a` et `b` de la chaîne de caractères `s`. On propose d'insérer dans la fonction `cherche_mot(m)` le code (où `i` est l'indice où le mot est trouvé, et `p` est la longueur de `m`) :

```
print(s[i-30:i+p+30].replace("\n", " "))
```

Le `"\n"` est un **caractère de saut de ligne** (*newline* en anglais), c'est un caractère à part entière d'une chaîne de caractère ou d'un fichier texte (tout comme les lettres, les espaces et la ponctuation) dont le but est d'indiquer un saut de ligne. Le fichier joint contient de nombreux sauts de lignes qui coupent les phrases, ce qui est un peu ennuyeux pour notre programme.

II Correction orthographique

On souhaite maintenant chercher un mot dans un dictionnaire, éventuellement pour y corriger des erreurs d'orthographe.

Le fichier joint contient une liste énorme de mots (chaque mot de la langue française y apparaît avec toutes ses variantes, chaque verbe avec toutes ses conjugaisons). Il est chargé par le fichier joint et la variable `dictionnaire` est alors une liste de mots. Dans un premier temps on peut utiliser l'opération `==` pour tester si deux mots sont égaux.

Exercice 9.4 *Mise en jambe*

Écrire une fonction `est_présent(m)` qui prend en argument un mot `m` et renvoie `True` si `m` est effectivement bien présent dans le dictionnaire, et `False` sinon.

Le but est maintenant d'étudier le problème de la correction orthographique, ce qui nécessite de définir à quelle condition on peut considérer que deux mots donnés sont « proches » (représentent le même mot mais l'un a potentiellement des erreurs d'orthographe). C'est un problème difficile en général... Mais nous allons nous restreindre à comparer uniquement des mots de même longueur.

Définition

La **distance de Hamming** entre deux mots s et t **supposés de même longueur** est le nombre de lettres qui diffèrent à la même place, c'est à dire le nombre d'indices i tels que $s[i] \neq t[i]$.

Par exemple, la distance de Hamming entre les mots $s = \text{"vacances"}$ et $t = \text{"savantes"}$ est 3. On peut le voir en essayant de superposer les deux mots :

0	1	2	3	4	5	6	7
V	A	C	A	N	C	E	S
S	A	V	A	N	T	E	S

Exercice 9.5

Écrire une fonction `distance(s, t)` qui calcule la distance de Hamming entre les deux mots s et t , supposés de même longueur (on pourra utiliser `assert` pour exclure le cas où cette condition ne serait pas vérifiée).

Pour rechercher un mot dans un dictionnaire en tolérant d'éventuelles erreurs, on doit d'abord se fixer un *seuil*, c'est-à-dire décider combien d'erreurs au maximum on autorise. Avec un seuil de 0, on ne peut trouver que le mot lui-même. Mais si le seuil est trop gros, on risque de trouver beaucoup trop de mots (surtout si m est déjà court). Tester d'abord avec 1.

Exercice 9.6

Écrire une fonction `cherche_seuil(m, seuil)` qui cherche dans le dictionnaire, parmi tous les mots qui sont de même longueur que m , ceux qui ont une distance de Hamming inférieur à `seuil`, et les affiche.

En pratique, on ne sait pas à l'avance s'il y a bien des mots à distance 1 de m , ou bien si les mots les plus proches sont à distance 2, ou plus...

Exercice 9.7

Écrire une fonction `mots_les_plus_proches(m)` qui prend en argument un mot m (contenant éventuellement des erreurs d'orthographe) et qui calcule d'abord la distance minimale des mots, parmi tous ceux du dictionnaire, à m ; puis qui affiche tous les mots qui sont à cette distance minimale de m (cela nécessite donc d'itérer deux fois sur tout le dictionnaire).

Il existe un autre algorithme qui ne nécessite pas de calculer *d'abord* le minimum. On parcourt le dictionnaire en maintenant en mémoire à la fois une liste L des mots qu'on pense être les plus proches de m , et leur distance `dmin` à m :

- Si on lit un mot à distance plus grande que `dmin` : on ne fait rien et on passe à la suite.
- Si on lit un mot à distance exactement `dmin` : on l'ajoute à L (avec `append`).
- Si on lit un mot à distance inférieure à `dmin` : on met à jour la variable `dmin` (comme dans le calcul de minimum) et aussi la liste L , qui devient égale au seul mot qu'on vient de lire. Donc on « jette à la poubelle » tous les mots accumulés jusque là, puisqu'on en a trouvé au moins un qui est encore plus proche !
- À la fin, L est bien la liste des mots les plus proches et `dmin` est la distance minimale d'un mot du dictionnaire à m .

Exercice 9.8 (*)

Écrire cette fonction `liste_mots_les_plus_proches(m)` qui renvoie la liste des mots les plus proches de m , en itérant une unique fois sur le dictionnaire, en fabriquant au fur et à mesure la liste des mots les plus proches quitte à vider la liste.

Les méthodes ci-dessus avec une simple petite variante permettent de trouver des mots pour compléter des grilles de mots fléchés ou des mots croisés. Dans ce contexte, on a des cases fixées et on doit chercher un mot rentrant dans ces cases, connaissant à l'avance le nombre de lettres ainsi que la position de certaines lettres. On utilisera le caractère `*` pour noter une case vide au contenu encore inconnu. Par exemple, la séquence `"p*t*on"` peut accepter le mot `"python"` mais aussi `"patron"` ou bien `"potion"`.

0	1	2	3	4	5
P	*	T	*	O	N
P	Y	T	H	O	N
P	A	T	R	O	N
P	O	T	I	O	N

Exercice 9.9

1. Écrire une fonction `est_acceptable(s, m)` qui prend en argument une chaîne `s` éventuellement composée de caractères `*`, et un mot `m` (sans `*`, supposé de même longueur que `s`) et renvoie `True` s'il est possible de faire coïncider `m` sur `s`, et `False` sinon.
2. En déduire une fonction `complète(s)` qui prend en argument une chaîne `s` comme ci-dessus et affiche tous les mots, parmi ceux du dictionnaire, qui peuvent coïncider sur `s`.

III L'algorithme de Knuth-Morris-Pratt (KMP) (d'après concours TB 2022)

On reprend la situation de la partie I où le but est de rechercher un mot dans un texte. Nous proposons d'étudier une méthode qui peut être sensiblement plus efficace si on raisonne en terme de nombre de comparaisons de caractères à effectuer. L'idée est qu'on n'est pas obligé de recommencer à chaque fois avec la fonction `le_mot_est_ici` à l'indice $i + 1$ après avoir testé à l'indice i (« décaler la fenêtre » d'un cran à chaque fois) : on peut se servir de l'information du nombre de comparaisons vérifiées par `le_mot_est_ici` pour décaler de plus d'un cran la prochaine recherche.

III.1 Quelques exemples

Exemple 1 Supposons que le mot à chercher `m` est constitué de lettres toutes différentes, par exemple on cherche le mot `m = "AGCT"` dans `s = "AGTAGCAGCT"`. Alors si la fonction `le_mot_est_ici(s, m, i)` échoue (renvoie `False`), on peut directement continuer à chercher dans `s` juste après le dernier échec de comparaison.

0	1	2	3	4	5	6	7	8	9
A	G	T	A	G	C	A	G	C	T
A	G	C	T						
		A	G	C	T				
			A	G	C	T			
						A	G	C	T

Ici on teste d'abord à partir de la position 0 (cela échoue sur le C de `m`), puis directement 2 (ce qui échoue directement sur le A) puis 3 (ce qui échoue à cause du T de `m`) puis enfin directement à 6, où on trouve le mot. La « fenêtre glissante » avance plus vite que dans l'algorithme naïf.

Exemple 2 La situation est moins simple quand une partie du mot à chercher se « répète dans lui-même ». Par exemple si on cherche `m = "ACAT"` dans `s = "ACACATAG"` :

0	1	2	3	4	5	6	7
A	C	A	C	A	T	A	G
A	C	A	T				
		A	C	A	T		
				A	C	A	T

On teste d'abord si le mot est en position 0, ce qui échoue à cause de son T (la comparaison en position 3 dans `s`) ; mais on ne va pas sauter directement à position 3 ou après car le mot démarre en fait à la position 2. En fait, une fois qu'on sait déjà que les lettres A coïncident à la position 2, on peut *poursuivre* les comparaisons à partir de la position 3 pour essayer de savoir si le mot démarre à la position 2. À la fin, le fait que la coïncidence ait lieu prouve aussi que les lettres A coïncident bien à la position 4, et donc qu'on peut tenter de poursuivre les comparaisons à partir de la position 5 pour savoir si le mot démarre à la position 4 ; ce n'est ici pas le cas.

Exemple 3 Pour la recherche du mot $m = \text{"ATCGATG"}$ à l'intérieur de $s = \text{"ATCGATCGATCGATG"}$, on obtient les sauts suivants, ne trouvant pas le mot aux positions 0 ni 4, mais 8 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	T	C	G	A	T	C	G	A	T	C	G	A	T	G
A	T	C	G	A	T	G								
				A	T	C	G	A	T	G				
								A	T	C	G	A	T	G

III.2 Les notions

L'idée de l'algorithme est d'abord de construire, étant donné le mot m , un « plan » qui puisse nous dire de combien d'indices sauter après avoir échoué à tester si le mot était à l'indice i dans s , en fonction des caractères de m qui ont bien été comparés et de la connaissance de « comment les caractères comparés au début du mot se retrouvent à la fin du même mot ».

Dans la problématique de recherche d'une séquence ADN, c'est surtout la chaîne s qui est très grande, et le mot m peut contenir beaucoup de répétitions dans lui-même, ainsi ce n'est pas une contrainte lourde de faire des pré-calculs concernant le mot m seul pour pouvoir ensuite sauter plus rapidement dans s .

Définition

Étant donné un mot m :

- Un **préfixe** de m est un sous-mot (c'est à dire, une suite de lettres consécutives de m) démarrant au début de m .

Par exemple **"GCC"** est un préfixe de **"GCCATC"**.

- Un **suffixe** de m est un sous-mot terminant à la fin de m .

Par exemple **"ATC"** est un suffixe de **"GCCATC"**.

- On convient que la chaîne vide **"** est à la fois un préfixe, et un suffixe, de n'importe quel mot.
- Le **bord** du mot m est le plus grand sous-mot (différent de m tout entier) qui est à la fois un préfixe et un suffixe.

Par exemple :

- ▶ **"AGT"** est le bord de **"AGTCGAGT"**,
- ▶ La chaîne vide **"** est le bord de **"AGTCGACTC"**,
- ▶ **"TTT"** est le bord de **"TTTGCCTTT"**, alors que **"TT"** ne l'est pas (c'est bien un préfixe et un suffixe mais il n'est pas le plus grand possible).

Exercice 9.10

1. Écrire une fonction `est_bord(m, k)` qui renvoie `True` si les k premiers caractères du mot m sont aussi les k derniers — ainsi le bord de m est au moins de longueur k .

Attention à tous les indices manipulés et aux longueurs !

2. Écrire la fonction `longueur_bord(m)` qui renvoie la longueur du bord de m , en cherchant le plus grand k tel que la fonction précédente renvoie `True`.

Attention, elle peut très bien renvoyer `False` pour une valeur de k mais `True` pour des valeurs plus grandes ! Par exemple dans $m = \text{"AGTCGAGT"}$ la fonction précédente renvoie `True` seulement pour $k = 3$. Mais elle renvoie évidemment `True` pour $k = 0$. Il faut donc garder en mémoire le plus grand k qu'on a rencontré pour lequel on a obtenu `True`.

3. En déduire une fonction `longueurs_bords_prefixes(m)` qui prend en argument une chaîne de caractères m et qui renvoie la liste B des longueurs du bord de chaque préfixe de m . Ainsi pour tout indice j , $B[j]$ sera la longueur du bord du préfixe formé des $j + 1$ premiers caractères de m , c'est à dire de $m[:j+1]$.

Par exemple, l'appel `longueurs_bords_prefixes("AATGAATC")` devra renvoyer la liste `[0, 1, 0, 0, 1, 2, 3, 0]`. En effet :

- "" est le bord de la chaîne "A",
- "A" est le bord de la chaîne "AA",
- "" est le bord des chaînes "AAT" et aussi "AATG",
- "A" est le bord de la chaîne "AATGA",
- "AA" est le bord de la chaîne "AATGAA",
- "AAT" est le bord de la chaîne "AATGAAT",
- "" est le bord de la chaîne "AATGAATC".

III.3 L'algorithme

L'algorithme KMP fonctionne ainsi :

- On se donne une chaîne `s` et un mot `m` à chercher, et on calcule la liste des longueurs des bords des préfixes `B` ci-dessus.
- On initialise une variable `i` à 0, c'est un indice où chercher le mot dans `s`, et on démarre une boucle sur `i`. On préférera une boucle `while` plutôt que `for`, ce qui permet de faire varier les tailles du saut à l'indice suivant.
- On cherche le plus petit indice `j`, à partir de 0 et s'il existe, pour lequel `s[i+j] ≠ m[j]`.
 - S'il n'y en a pas, c'est que le mot `m` se trouve bien ici en démarrant à l'indice `i`.
 - Si `j = 0`, c'est que la comparaison rate sur la première lettre. On passe alors simplement à l'indice `i+1`.
 - En général, on saute à l'indice `i + j - B[j-1]` de `s`, mais en démarrant la comparaison avec `m` à partir de son indice `B[j-1]`.

Exercice 9.11

Compléter le programme pour écrire la fonction `cherche_KMP(s, m)`.

Pour observer le fonctionnement du programme, on pourra rajouter des instructions `print` pour afficher les caractères comparés et le nombre de sauts effectués.

Exercice 9.12

Appliquer l'algorithme à la main sur les exemples de la partie III.1, et comparer avec les tests du programme.

TP 10

Récurtivité

La récursivité n'est pas un nouveau concept du langage Python ni même une nouvelle fonction, mais une technique générale de programmation. Une fonction écrite en Python est dite **récursive** quand, dans le corps de la fonction, elle fait appel à... elle-même. Cela correspond directement à la notion mathématique de récurrence.

I Exemples simples

Nous avons appris à calculer les puissances de 2 avec une simple boucle **for**. Cependant la définition mathématique la plus naturelle du nombre 2^n est par récurrence :

$$\forall n \in \mathbb{N}, \quad 2^n = \begin{cases} 1 & \text{si } n = 0 \\ 2 \times 2^{n-1} & \text{si } n \geq 1 \end{cases}$$

Il est possible d'écrire un programme Python qui fait exactement cela !

```
def puissance2(n):
    if n == 0:
        return 1
    else:
        return 2 * puissance2(n-1)
```

Tester la fonctions avec des petites valeurs de n .

Pour mieux comprendre ce qui se passe dans ce programme, on rajoute des instructions **print()** :

```
def puissance2(n):
    print("Appel avec n =", n)
    if n == 0:
        print("Fin")
        return 1
    else:
        p = 2 * puissance2(n-1)
        print("Retour avec", p)
        return p
```

La fonction est appelée elle-même successivement avec des valeurs de n de plus en plus petites, jusqu'à la valeur $n = 0$, puis les retours se font dans l'ordre inverse des appels. Ce qu'on observe s'appelle la **pile d'appels**, voir l'annexe IV : chaque fois que la fonction s'appelle elle-même, elle note dans la mémoire l'endroit exact où elle était afin de pouvoir y revenir quand l'appel est terminé, et elle revient au *dernier* endroit qui a été noté.

Exercice 10.1

Une fonction célèbre est écrite ici de façon récursive. Quelle est la fonction **f** ?

```
def f(n):
    if n == 0:
        return 1
    else:
        return n * f(n-1)
```

Dans la suite de ce chapitre, quand on demande écrire une fonction récursive il faut que la fonction fasse appel à elle-même au lieu d'utiliser une boucle. Cela nécessite d'abord de repenser les programmes !

Exercice 10.2

Soit la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 1$ et $\forall n \in \mathbb{N}^*, u_n = 3u_{n-1} + 2$. Écrire une fonction récursive **suite(n)** qui renvoie le terme u_n .

Exercice 10.3

Soit la somme $S_n = \sum_{k=1}^n k^3$.

1. Qu'est-ce que S_1 ? Quelle est la relation entre S_n et S_{n-1} ?
2. Écrire une fonction récursive `somme_cubes(n)` qui prend en argument un nombre entier n et qui calcule S_n .

II Quelques phénomènes

À retenir

Pour écrire une fonction récursive, il est impératif d'avoir d'abord au brouillon une bonne formulation mathématique du problème. Similairement au raisonnement par récurrence, une fonction récursive contient toujours :

- Une condition initiale (en général le cas $n = 0$ ou $n = 1$), où la fonction renvoie directement une valeur, toujours avec `return` (sinon ça ne marche pas !),
- Un appel récursif, où la fonction `f(n)` s'appelle elle-même avec des valeurs plus petites, typiquement $n - 1$ ou $n/2$.

La fonction bien écrite au brouillon peut ensuite se traduire facilement en Python.

Exercice 10.4 *Fibonacci, le retour du retour*

On rappelle que la suite de Fibonacci est la suite $(F_n)_{n \in \mathbb{N}}$ définie par :

$$\forall n \in \mathbb{N}, \quad F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2 \end{cases}$$

1. Traduire cette définition en une fonction récursive `fibonacci(n)`.
2. Tester la fonction `fibonacci(n)` en prenant des valeurs de n de plus en plus grandes, pas à pas, surtout en augmentant doucement à partir d'environ 30 (ne pas tester directement plus de 35). Que se passe-t-il ?
3. Pour comprendre ce qui se passe, insérer au tout début de la fonction la ligne

```
print("Appel avec n =", n)
```

et ré-essayer, cette fois d'abord avec des tout petits nombres, comme 4, 5, 6. Expliquer.

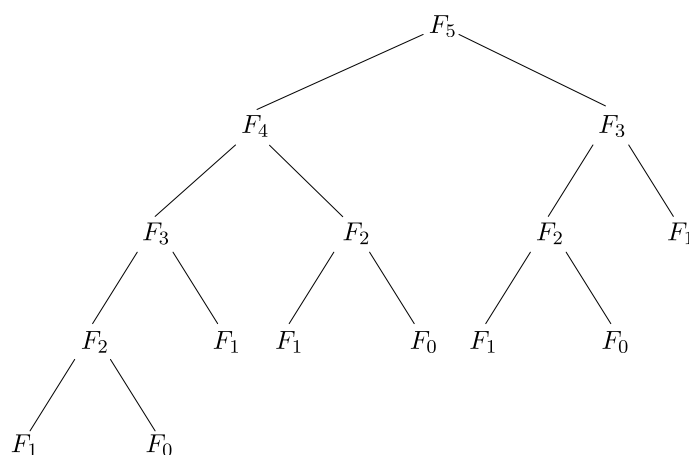


Figure 10.1. – Arbre des appels pour `fibonacci(5)`.

Exercice 10.5 *Exponentiation rapide*

Soit $a \in \mathbb{R}$. Remarquons que les puissances a^n vérifient la relation de récurrence suivante, pour $n \in \mathbb{N}^*$ (laissons de côté $a^0 = 1$) :

$$a^n = \begin{cases} a & \text{si } n = 1 \\ (a^{n/2})^2 & \text{si } n \text{ est pair} \\ a \times a^{n-1} & \text{si } n \text{ est impair} \end{cases}$$

Par exemple pour calculer a^5 cela revient à écrire $a^5 = a \times a^4$ puis $a^4 = (a^2)^2$, ainsi $a^5 = a \times (a^2)^2$ et on peut vérifier que cela nécessite en tout de calculer 3 multiplications, ce qui est mieux que d'écrire $a^5 = a \times a \times a \times a \times a$ qui en nécessite 4. Pour calculer a^6 cela revient à écrire $a^6 = (a^3)^2 = (a \times a^2)^2$ et on compte en tout 3 multiplications aussi, au lieu de 5 avec la méthode naïve.

1. Avec la même méthode, combien de multiplications sont nécessaires pour calculer les nombres suivants ?

$$a^{16} \quad a^{15} \quad a^{24}$$

2. Écrire une fonction récursive `puissance_rapide(a, n)` qui prend en argument un entier `a` et un entier `n` (celui-ci est supposé strictement positif) et qui calcule a^n selon ce procédé.

On rappelle que, comme les exposants sont entiers, on utilise le reste dans la division euclidienne `n % 2` pour tester la parité et on utilise le quotient `n // 2`.

Exercice 10.6

On rappelle la formule de Pascal pour les coefficients binomiaux, ré-écrite en fonction de $n - 1$ et $k - 1$ et valable pour tout $k \in \mathbb{Z}$:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

1. Écrire une fonction récursive `binome(n, k)` qui renvoie le coefficient binomial $\binom{n}{k}$, pour tous $n \in \mathbb{N}$ et $k \in \mathbb{Z}$. Attention au cas d'initialisation !
2. Tester la fonction ci-dessus avec des valeurs de n et de k de plus en plus grandes en augmentant lentement, par exemple n autour de n et k petit ou à la moitié de n . Encore une fois qu'observe-t-on ?
3. Pour corriger ce problème, une méthode est d'écrire une fonction `binome_liste(n)` qui renvoie une liste de longueur $n + 1$ correspondant à **toute** la ligne du triangle de Pascal des coefficients $\binom{n}{k}$ pour $0 \leq k \leq n$. Par exemple `binome_liste(3)` doit renvoyer `[1, 3, 3, 1]`.

Écrire cette fonction, de façon récursive.

À retenir

La récursivité est une **méthode de programmation**.

- Elle a l'avantage d'être parfois plus proche du langage mathématique et plus simple à programmer,
- Mais elle peut donner lieu à des programmes moins efficaces, notamment à cause du phénomène d'arbre des appels comme pour `fibonacci`.

Dans les sujets écrits, il peut être précisé « écrire une fonction récursive qui... » ou bien « écrire une fonction itérative (c'est à dire, non-récursive) qui... ». Si rien n'est précisé, alors vous êtes libre de choisir la méthode avec laquelle vous vous sentez le plus à l'aise.

III Problèmes

Exercice 10.7 (*) *Ordre lexicographique*

On souhaite écrire une fonction qui compare deux mots selon l'ordre du dictionnaire, appelé **ordre lexicographique**. Cela signifie que le mot s vient avant le mot t si :

- La première lettre de s vient avant la première lettre de t ,
- Ou bien les premières lettres sont les mêmes, et la deuxième lettre de s vient avant la deuxième lettre de t ,
- etc,
- Éventuellement on arrive à la situation où les deux mots sont égaux ; ou bien où l'un est plus court que l'autre et ils sont égaux sur le plus court, c'est à dire que l'un est un préfixe de l'autre, et c'est le plus court des deux qui vient avant dans le dictionnaire (par exemple « TRAVAIL » vient avant « TRAVAILLER »).

La fonction que l'on souhaite écrire s'appellera `compare(s, t)` et renverra 1 si s vient avant t dans le dictionnaire, -1 si s vient après t , et 0 si les deux mots sont égaux. On se limitera à des caractères parmi les 26 lettres de l'alphabet en minuscule ; sinon cela nécessite d'abord de décider comment comparer les majuscules, les lettres accentuées, et cela complique nettement les choses.

Pour l'écrire de façon récursive, on pourra décomposer un mot s en sa première lettre `s[0]` et le reste du mot obtenu avec la tranche `s[1:]`. Mais attention au mot vide "" qui apparaît naturellement quand on a enlevé plusieurs fois de suite la première lettre !

1. Au brouillon, proposer une formulation récursive du problème, en termes comme ci-dessus de première lettre et de comparaisons du reste du mot.
2. On donne `alphabet = "abcdefghijklmnopqrstuvwxyz"`. Écrire une fonction (non récursive) `numero(x)` qui prend en argument un caractère seul x et renvoie sa position en tant que lettre de l'alphabet (de 0 pour `a`, à 25 pour `z`).
3. Écrire la fonction `compare(s, t)`.

Exercice 10.8 (**)

On représente une partie de l'ensemble $\{1, 2, \dots, n\}$ par une liste contenant les éléments, rangés par ordre croissant de la partie. On souhaite écrire une fonction `parties(n)` qui donne la *liste* de *toutes* les parties de $\{1, 2, \dots, n\}$. On peut démarrer avec $n = 0$, il n'y a que la partie vide. En général il y a deux types de parties : celles ne contenant pas n — ce sont donc des parties de $\{1, 2, \dots, n - 1\}$ — et celles contenant n , obtenues à partir des parties de $\{1, 2, \dots, n - 1\}$ en leur rajoutant n .

Écrire la fonction récursive `parties(n)`.

Par exemple, l'ensemble des parties de $\{1, 2, 3\}$ est

$$\mathcal{P}(\{1, 2, 3\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

et la fonction produit le résultat (l'ordre exact dépend de la façon d'écrire la fonction récursive)

```
>>> parties(3):
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
```

IV Annexe : la pile d'appels

Le mécanisme d'appel Pour comprendre la récursivité, il est important de comprendre plus en profondeur le processus d'appel de fonction. Reprenons notre exemple

```
def factoriel(n):
    if n == 0:
        return 1
    else:
        return n * factoriel(n-1)
```

Lors d'un appel tel que `factoriel(5)`, alors la fonction démarre avec $n = 5$ et on se retrouve à devoir appeler `factoriel(4)`. Mais avant de passer à `factoriel(4)`, il faut d'abord sauvegarder dans la mémoire l'endroit précis où nous en sommes de `factoriel(5)`, c'est là qu'une fois qu'on aura obtenu le résultat du calcul de `factoriel(4)` on pourra le multiplier par 5 puis le renvoyer. À chaque appel de la fonction à un rang $n - 1$, l'ordinateur doit sauvegarder diverses informations sur l'état de la fonction au rang n avant de sauter au rang $n - 1$, et ces informations sont organisées selon une **pile** : exactement comme une pile d'assiettes, les nouvelles informations sont posées directement par-dessus les anciennes, et quand on veut les récupérer, on a d'abord accès au dernier élément qui a été empilé. Ainsi quand l'appel `factoriel(4)` — dont les informations étaient sur le dessus de la pile — se termine, on revient directement à `factoriel(5)`.

Cette structure de pile s'observe très bien en comparant les programmes suivants, dont la seule différence est l'ordre des instructions entre le `print` et l'appel récursif : tester (par exemple avec $n = 10$) et comparer

```
def boum(n):
    if n == 0:
        print("BOUM")
    else:
        print(n)
        boum(n-1)
```

```
def top(n):
    if n == 0:
        print("TOP")
    else:
        top(n-1)
        print(n)
```

Une analogie Une analogie est la suivante. Imaginons qu'on lise un livre de mathématiques au chapitre sur les bijections. Mais qu'on ne comprenne pas. Alors on laisse le livre ouvert mais par dessus on ouvre un autre livre sur les ensembles et la logique. Mais on ne comprend toujours pas. Alors on laisse celui-ci ouvert et on ouvre un livre de lycée. Là, on lit et on comprend. À la fin on referme le livre de lycée et on retombe sur celui d'ensembles et de logique, qu'on peut continuer à lire. Et quand on a fini on le referme et on retombe là où on en était sur les bijections. Les livres se sont empilés sur le bureau, chaque lecture étant mise en pause à un moment pour ouvrir un autre livre par-dessus, et lorsqu'il est refermé on reprend exactement la lecture au point où on était.

Dans d'autres langages Certains langages de programmation ont été conçus pour encourager la récursivité, avec un point de vue plus proche des mathématiques. C'est le cas du langage OCaml, enseigné notamment en MPSI et MP2I, réputé pour ses algorithmes qui transforment par derrière un programme récursif en une version non-récursive tout aussi efficace. Dans ce langage, même les listes sont définies de façon récursive : soit c'est la liste vide, soit c'est un élément (*tête*) attaché au reste de la liste (*queue*). Une fonction aussi simple que la longueur d'une liste est alors récursive : c'est zéro pour la liste vide, et sinon c'est un de plus que la longueur de la queue. On parle de **type récursif**. Cela est particulièrement intéressant pour traiter des structures en arbres, graphes, et de divers problèmes de combinatoire, qui seraient nettement plus compliqués à écrire avec de simples listes et boucles Python. Un exemple d'OCaml qui se lit aussi facilement que la description mathématique :

```
type liste =
| Vide
| Attache of int * liste

let rec longueur l = match l with
| Vide -> 0
| Attache(tete, queue) -> 1 + longueur queue
```

TP 11

Dichotomie

Le mot **dichotomie** provient du grec et signifie « couper en deux ». En informatique, il s'agit ici d'une méthode générale de recherche de solution d'un problème en coupant l'intervalle de recherche en deux à chaque étape. Nous allons d'abord l'illustrer par un jeu simple et bien connu.

I Préliminaire : un jeu

On connaît le jeu suivant entre deux joueurs : l'un pense à un nombre entre 1 et 100 et l'autre doit le deviner en formulant ses propositions, auxquelles la réponse sera seulement « plus petit », « plus grand » ou bien « trouvé ! ».

Il est tout à fait possible de programmer ce jeu, où c'est le programme qui choisit le nombre au hasard !

Exercice 11.1

Compléter le programme fourni : il choisit un nombre au hasard, l'affiche (au début, pour faire des tests, et ensuite on supprimera la ligne) puis demande à l'utilisateur une proposition de nombre. Il dit ensuite si le nombre à trouver est plus grand, plus petit, ou si c'est bon.

Jouez à ce jeu pendant quelques minutes.

Exercice 11.2

Modifier le programme pour qu'en plus il compte le nombre de tentatives du joueur, affichant à la fin un « trouvé en n coups », et tenter de gagner en un minimum de coups possibles.

On peut aussi apporter diverses modifications comme par exemple choisir les nombres dans un intervalle plus grand, ce qui augmente la difficulté.

Exercice 11.3

Quelle semble être la meilleure stratégie possible, en rapport avec le titre de ce TP ? Pour un nombre entre 1 et 100, en combien de coups êtes-vous absolument certain de gagner ? Plus généralement pour un nombre entre 1 et N , pouvez-vous donner une formule pour le nombre de coups maximal en lesquels on est certain de gagner, et le démontrer ?

II Application aux solutions d'équations

On s'inspire de cette méthode pour trouver les solutions d'une équation. Soit une fonction f définie sur un intervalle $[a, b]$ à valeurs dans \mathbb{R} , continue et croissante, on cherche une solution à l'équation $f(x) = y$ en supposant $f(a) < y < f(b)$. Alors on calcule $m = \frac{a+b}{2}$ et on compare $f(m)$ avec y . Si $f(m) < y$ on doit chercher x entre m et b , et si $f(m) > y$ on doit chercher x entre a et m . Cela divise par deux la taille de l'intervalle de recherche.

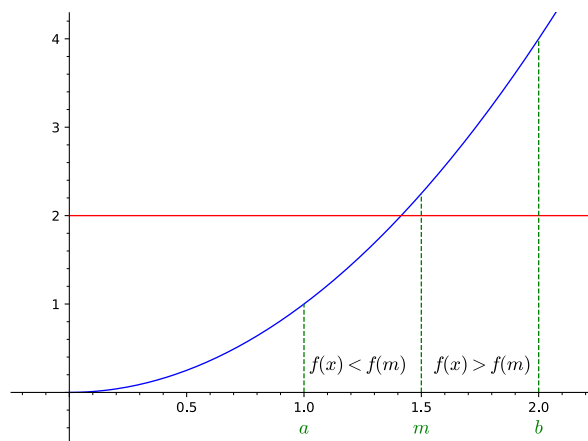


Figure 11.1. – Recherche par dichotomie de la solution à $f(x) = 2$, où $f : x \mapsto x^2$.

Remarque. On pourra traduire ces idées en une *preuve* du théorème des valeurs intermédiaires. Mais cela nécessite d'abord quelques pré-requis sur la continuité...

Exercice 11.4 Échauffement

On cherche les approximations successives de $\sqrt{2}$ qu'on obtiendrait avec cette méthode (mais sans utiliser la fonction racine carrée). Ici c'est la fonction $f : x \mapsto x^2$, et on sait $f(1) < 2 < f(2)$, autrement dit $1 < \sqrt{2} < 2$. À l'étape suivante on calculera $m = \frac{1+2}{2} = \frac{3}{2}$ puis $m^2 = \frac{9}{4} > 2$. Ceci *démontre* que $1 < \sqrt{2} < \frac{3}{2}$ (de l'autre côté, si $\frac{3}{2} < x < 2$ alors $x^2 > 2$). On peut continuer en prenant le milieu entre 1 et $\frac{3}{2}$: c'est $m = \frac{1}{2}(1 + \frac{3}{2}) = \frac{5}{4}$, et on calcule $m^2 = \frac{25}{16}$, mais $m^2 < 2 = \frac{32}{16}$, donc ceci démontre $\frac{5}{4} < \sqrt{2} < \frac{3}{2}$ soit $1,25 < \sqrt{2} < 1,5$. Et recommencer, etc.

1. Écrire une fonction `racine2(n)` qui effectue n étapes de ce procédé, en affichant à chaque fois les bornes (a, b) de l'encadrement obtenu.
2. Améliorer la fonction en `racine2_seuil(s)` qui prend en argument un seuil $s > 0$ (on lui donnera par exemple $s = 10^{-6}$ avec la syntaxe `racine2_seuil(1e-6)`), et s'arrête quand l'encadrement (a, b) vérifie $b - a < s$. Si tout se passe bien on a à tout moment $a < b$ dans la boucle.
3. Bonus : la dichotomie est naturellement un algorithme récursif, car chercher par dichotomie la solution entre a et b revient à chercher par dichotomie la solution soit entre a et m soit entre m et b . Pouvez-vous écrire la fonction précédente de façon récursive ?

Exercice 11.5

Soit la fonction $f : x \mapsto x^3 - 5x - 3$.

1. Avec un tableau de variations, montrer que l'équation $f(x) = 0$ admet exactement trois solutions $x_1 < x_2 < x_3$. En calculant quelques valeurs particulières de f , donner un encadrement (par des nombres entiers) de chacune de ces solutions.
2. En déduire trois fonctions `solution1(s)`, `solution2(s)`, `solution3(s)`, donnant chacune un encadrement de la solution avec un écart strictement inférieur à $s > 0$. Attention car f est décroissante autour de x_2 !

Exercice 11.6

On s'intéresse maintenant à l'équation $(E_t) : e^x = 3 + tx$, d'inconnue $x \in \mathbb{R}$, mais avec un paramètre réel $t > 0$. On pourra utiliser la fonction `exp` du module `math` mais dans tous les cas il n'existe pas de formule simple pour la solution de cette équation. On pose $f_t : x \mapsto e^x - 3 - tx$.

1. Avec un tableau de variations (et un petit calcul de limites), justifier que l'équation $f_t(x) = 0$ admet bien deux solutions, une $x_t > 0$ et une autre $y_t < 0$.

Diverses intuitions ou représentations graphiques permettent de se rendre compte de la chose suivante : plus t est grand plus la solution x_t part vers $+\infty$; et plus t est petit plus la solution y_t part vers $-\infty$. On ne peut donc pas raisonnablement donner un encadrement de x_t et de y_t *a priori* qui soit indépendant de t .

2. Écrire une fonction `borne_x(t)` qui prend en argument le paramètre $t > 0$ et qui renvoie le *plus petit entier* $n \geq 0$ tel que $f_t(n) \geq 0$.
3. En déduire une fonction `solution_x(t, s)` qui prend en argument le paramètre $t > 0$ et un seuil $s > 0$, et cherche par dichotomie la solution x_t avec un seuil s , où au départ x_t est supposé être entre 0 et la borne renvoyée par la fonction précédente.
4. De même, écrire une fonction `borne_y(t)` qui renvoie le plus petit entier $n \leq 0$ tel que $f_t(n) \geq 0$, puis une fonction `solution_y(t, s)` qui cherche par dichotomie la solution y_t avec le seuil s .

III Application à la recherche dans une liste

La dichotomie est une méthode terriblement efficace pour rechercher un élément dans une liste à condition qu'elle soit **triée**.

Exercice 11.7 Révisions

Écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste `L` est triée, et `False` sinon.

Exercice 11.8 Révisions

Écrire une fonction `cherche_iteratif(L, x)` qui parcourt la liste et donne le premier indice où l'élément `x` apparaît, et renvoie `None` s'il n'apparaît pas.

La méthode par dichotomie pour chercher à quel indice apparaît l'élément `x` consiste à partir d'une liste triée `L` de longueur n et à comparer l'élément `x` avec un élément au milieu de `L`. Si `x` est plus grand, c'est qu'il faut chercher dans la moitié de droite de la liste, ceux qui sont plus grand que le milieu ; et sinon il faut chercher dans la moitié de gauche. On travaillera donc en initialisant deux variables : `a` à 0 et `b` à $n - 1$ (le dernier indice de la liste).

Par rapport au cas des fonctions continues, les nouveaux problèmes suivants se posent :

- Il n'y a pas toujours exactement un élément au milieu de la liste (cela dépend de la parité de n), on ne peut donc pas à n'importe quel moment comparer `x` avec `L[(a+b)/2]`. D'une part `L[3.5]` affichera une erreur, mais en fait même `L[3.0]`, il faut donc travailler avec la division en nombres entiers `(a+b) // 2`. Cela revient à comparer `x` avec l'élément de `L` au milieu dans la moitié gauche si n est pair ; par exemple dans une liste de longueur 8 le dernier indice est 7, la première moitié correspond aux indices de 0 à 3 et la deuxième moitié de 4 à 7, et `(0+7) // 2 = 3` donc on comparera `x` avec `L[3]`.
- L'élément `x` n'est pas nécessairement dans la liste, et on ne peut pas diviser par deux les intervalles à l'infini. Il faut donc savoir s'arrêter quand l'écart entre `a` et `b` devient exactement 1 et détecter alors si on trouve l'élément `x` à l'indice `a` ou `b` ou sinon pas du tout.
- Dans le cas où `x` serait strictement plus grand que tous les éléments de la liste, l'algorithme décrit ici s'appliquerait jusqu'au bout pour faire remonter la borne `a` jusqu'à `b` ; de même si `x` était plus petit que le minimum, l'algorithme décrit descendrait jusqu'à `a`. Il est donc préférable de tester dès le départ si `x` est bien compris entre `L[0]` et `L[n-1]`.

Exercice 11.9

Écrire la fonction `cherche_dichotomie(L, x)` qui recherche par cette méthode de dichotomie si l'élément `x` est dans la liste `L`, et renvoie son indice si elle le trouve et `None` s'il n'est pas dans la liste.

Le fichier joint du TP est un dictionnaire français de plus de 600 000 mots issu du logiciel libre de correction orthographique GNU Aspell. Le fichier Python à compléter charge le dictionnaire comme une liste de mots. Sur les mots, l'opération Python `<` correspond à l'ordre lexicographique, ainsi la fonction précédente s'applique directement si `L` est la variable nommée `dictionnaire` et si `x` est un mot. Si tout se passe bien, le dictionnaire a déjà été trié dans l'ordre de Python. Cela provoque quelques étrangetés pour un humain, notamment les mots commençant par des lettres accentuées sont rangés à la fin. Peu importe cependant, tant que le dictionnaire est bien rangé par ordre croissant *au sens* de l'opération `<` telle qu'elle est construite en Python.

Exercice 11.10

Avec le fichier joint, tester la recherche dans le dictionnaire, à la fois itératif et par dichotomie. On pourra modifier la fonction `cherche_dichotomie(L, x)` pour afficher le nombre d'étapes de comparaisons qu'elle réalise, et comparer cette valeur avec le logarithme en base 2 de la longueur du dictionnaire.

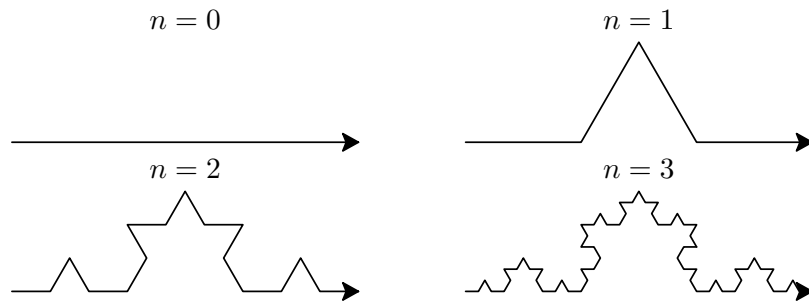
Exercice 11.11

Le fichier à compléter contient un morceau de programme pour chronométrer le code Python, avec la fonction `timeit` du module `timeit`. Elle prend en argument le nombre de répétitions qu'elle va effectuer et le résultat est exprimé en secondes. Tester le chronométrage, sur divers mots, à la fois avec les recherches itératives et dichotomiques (attention, pour chronométrer sa fonction il est important qu'elle ne fasse pas de `print`), avec au maximum un millier de répétitions.

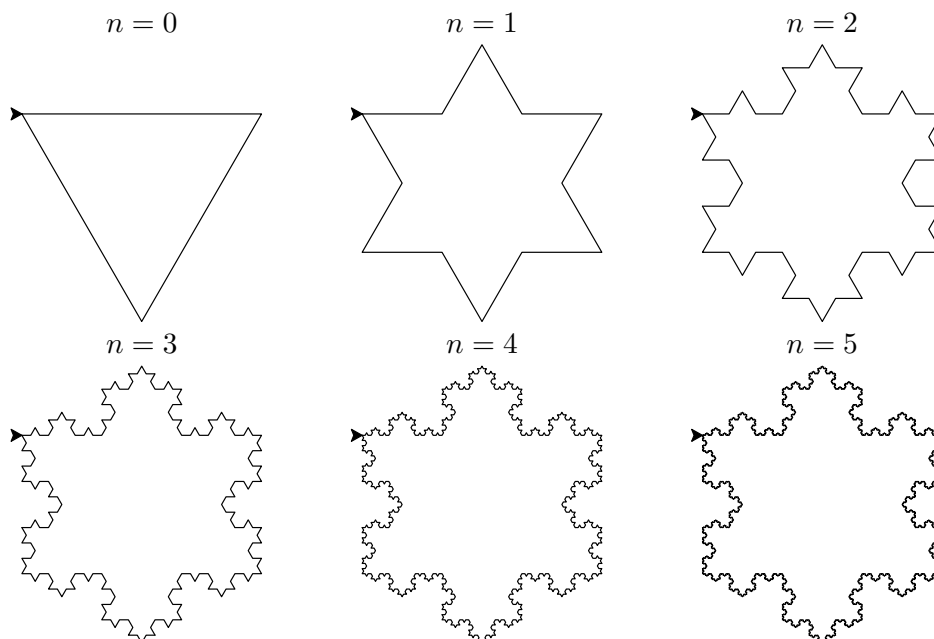
TP 12

Algorithmes récursifs

Le but du TP est de tracer le **flocon de Von Koch**. D'abord on trace la courbe de Von Koch : c'est une courbe qui est obtenue par étapes successives en partant d'une ligne droite puis en la coupant en trois segments égaux et en remplaçant le segment central par un triangle équilatéral. Chaque étape est numérotée par un entier n :



Puis on en déduira le flocon complet :



I Introduction au module turtle

Le module `turtle` implémente en Python une méthode ancienne d'apprentissage de la programmation (remontant au début des années 70 !) basée sur une interface graphique appelée **la tortue**.

Il faut imaginer qu'au démarrage du programme se tient au centre de l'écran une tortue tenant un crayon. Les instructions du programme donnent notamment à la tortue l'ordre d'avancer en ligne droite d'un certain nombre de pas, ou bien de tourner à gauche ou à droite. La tortue laisse donc un trait derrière elle, ce qui permet de tracer des figures intéressantes, surtout quand on combine ces instructions avec des fonctions et des boucles Python. Éventuellement, d'autres fonctions permettent de configurer l'épaisseur ou la couleur du tracé. Le crayon peut aussi être relevé — auquel cas la tortue se déplace sans laisser de tracé derrière elle — jusqu'à ce que, éventuellement, il soit de nouveau abaissé.

Le système de coordonnées utilisé est tel qu'au démarrage la tortue est au centre de l'écran aux coordonnées $(x, y) = (0, 0)$, et tournée vers la droite. Les dimensions de la fenêtre (donc les abscisses et ordonnées minimales et maximales, c'est à dire les coordonnées du bord de la fenêtre) peuvent dépendre des configurations de l'utilisateur, on gardera donc la tortue dans une zone raisonnable au centre de la fenêtre.

Pour commencer on chargera le module `turtle` avec cette ligne **au tout début du fichier** et à exécuter une seule fois :

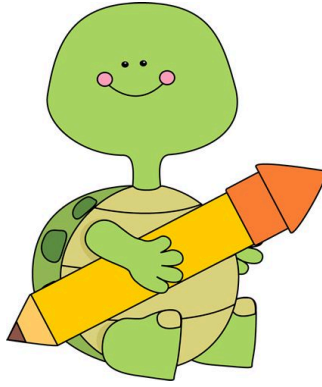


Figure 12.3. – Source : <https://www.mycutegraphics.com/>

```
from turtle import *
```

On peut alors utiliser toutes les fonctions du module sans écrire à chaque fois le préfixe `turtle`.

Tester ensuite directement (mode script, dans une cellule, puis exécuter) les lignes suivantes pour vérifier que tout fonctionne correctement :

```
###
reset()
forward(200)
done()
###
```

La première ligne demande à bien démarrer avec une nouvelle fenêtre blanche. La deuxième donne l'ordre à la tortue d'avancer de 200 pas (ajuster éventuellement ce nombre selon la taille de votre écran et de la fenêtre qui apparaît). Enfin la dernière ligne met le programme en pause jusqu'à ce que l'utilisateur ferme la fenêtre. On commencera donc toujours nos programmes par ce `reset()` et on terminera toujours par `done()`.

Pour la liste des commandes disponibles, on pourra se référer à

- https://perso.limsi.fr/poinal/_media/python:turtle:turtleref.pdf : un aide-mémoire (pour les enfants).
- <https://docs.python.org/fr/3/library/turtle.html> : la documentation officielle (pour les vrais de vrais).

Nous en utiliserons en fait un tout petit nombre, les autres sont à découvrir et tester par vous-même !

- `reset()` : ré-initialise la fenêtre et la tortue à son point de départ, première instruction du programme.
- `done()` : affiche la fenêtre et attend que l'utilisateur la ferme, dernière instruction du programme.
- `forward(n)` : avance de `n` pas.
- `left(r)` : tourne à gauche de l'angle `r` exprimé en degrés.
- `right(r)` : idem mais tourne à droite.
- `goto(x, y)` : déplace la tortue à la position (x, y) . Sera utile au début du programme pour tenter de centrer la figure, précédé éventuellement d'un levé de crayon.
- `penup()` : lever le crayon. La tortue se déplace, mais ne laisse pas de tracé derrière elle.
- `pendown()` : rabaisser le crayon.

Exercice 12.1 *Échauffement*

Tracer un carré, puis un triangle équilatéral.

Exercice 12.2

Écrire une fonction `polygone(n)` qui trace un polygone régulier à n côtés.

Pour éviter que le dessin ne déborde de l'écran, on pourra diviser la longueur du côté par n (en effet la taille du dessin final sera à peu près proportionnelle à n et à la longueur du côté).

Attention, il est très important que `reset()` et `done()` ne soient pas *dans* la fonction, mais en dehors dans la cellule, comme ceci :

```
###
def polygone(n):
    ...

reset()
polygone(7)
done()
###
```

Exercice 12.3

Tracer la première étape ($n = 1$) d'une courbe de Von Koch, en calculant d'abord à la main les angles et les longueurs en jeu.

II Le flocon de Von Koch

Pour tracer un flocon de Von Koch, on va d'abord tracer la courbe de Von Koch et écrire une fonction récursive qui prend deux arguments :

- n : le numéro de l'étape qu'on est en train de tracer. Pour $n = 0$ la courbe est une simple ligne droite, pour $n = 1$ c'est la ligne polygonale de quatre morceaux tracée précédemment.
- L : un paramètre qui indique la longueur du morceau que l'on est en train de tracer, et qu'il faut diviser par 3 dans les appels récursifs.

Exercice 12.4

Écrire une fonction `vonkoch(n, L)` récursive qui trace la courbe de Von Koch : pour $n = 0$ elle trace une ligne droite de longueur L , et sinon elle s'appelle récursivement en divisant la longueur par 3 et entre les appels récursifs la tortue doit tourner de façon appropriée.

Pour l'appeler proprement, on écrira alors (directement dans une cellule à part) :

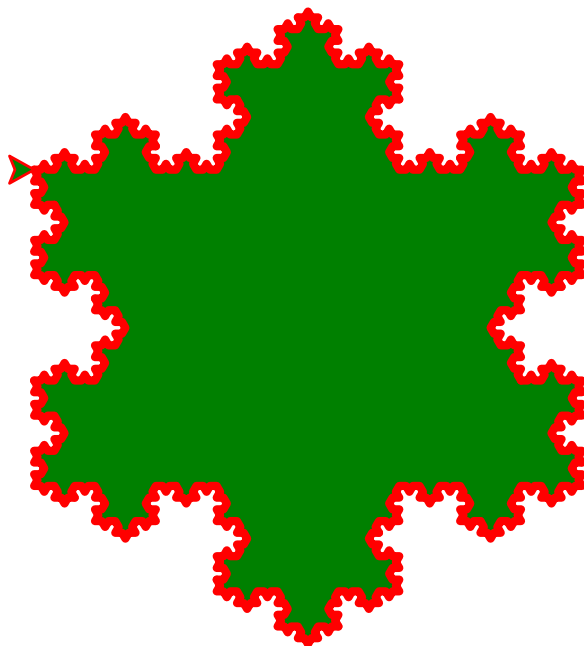
```
###
reset()
vonkoch(n, L)
done()
###
```

Pour dessiner le flocon complet sous forme hexagonale, il suffit... d'appeler trois fois la fonction précédente, en tournant du bon angle !

Exercice 12.5

Écrire une fonction `flocon(n, L)` (qui n'est plus récursive) qui trace le flocon de Von Koch complet avec n étapes.

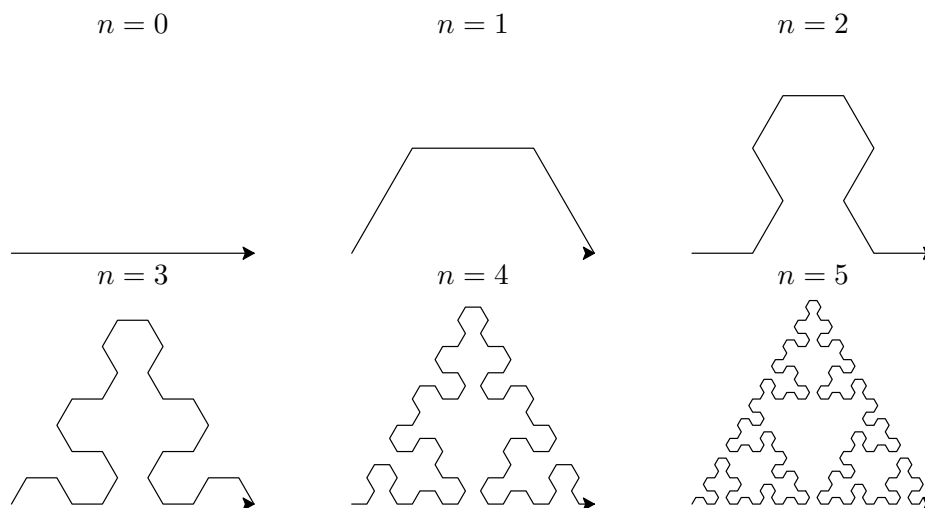
Pour que ce soit plus joli, il faut lire la documentation !

**Exercice 12.6**

Lire la documentation ou l'aide mémoire pour ajuster la couleur et l'épaisseur du trait, la couleur de remplissage, le titre de la fenêtre, et pour bien centrer la figure.

III Le triangle de Sierpinski

Il s'agit de la courbe obtenue par la suite de transformations de la figure suivante :



Encore une fois pour $n = 0$ il s'agit d'une ligne droite, et pour $n = 1$ on appellera cela une *tuile*. À chaque étape, les morceaux de ligne droite sont remplacés par des tuiles. Mais attention ! Il y a deux types de tuiles, celles tournant d'abord vers la gauche (comme le cas $n = 1$ ici) et celles tournant d'abord vers la droite (le miroir du cas $n = 1$; la première et la troisième du cas $n = 2$).

Exercice 12.7

Quels sont les longueurs et les angles en jeu pour tracer une tuile ? Observer que dans le dessin final on voit beaucoup de triangles équilatéraux et d'hexagones réguliers. Puis écrire comme échauffement une fonction `tuile(L)` qui trace l'étape $n = 1$ d'une tuile de base L .

On écrira alors **deux fonctions mutuellement récursives** `tuile_g(n, L)` et `tuile_d(n, L)`. Pour $n = 0$ ce sont toutes les deux des lignes droites, pour $n = 1$ `tuile_g` est la fonction précédente et `tuile_d` est exactement

son miroir (ce qui revient à échanger les rotations à droite et à gauche) — mais encore une fois on n'a en fait pas besoin de distinguer le cas $n = 1$ dans la récursivité. Enfin, chacune de ces fonctions fait appel récursivement à elle-même ainsi qu'à l'autre, avec le rang $n - 1$ et la longueur $L/2$.

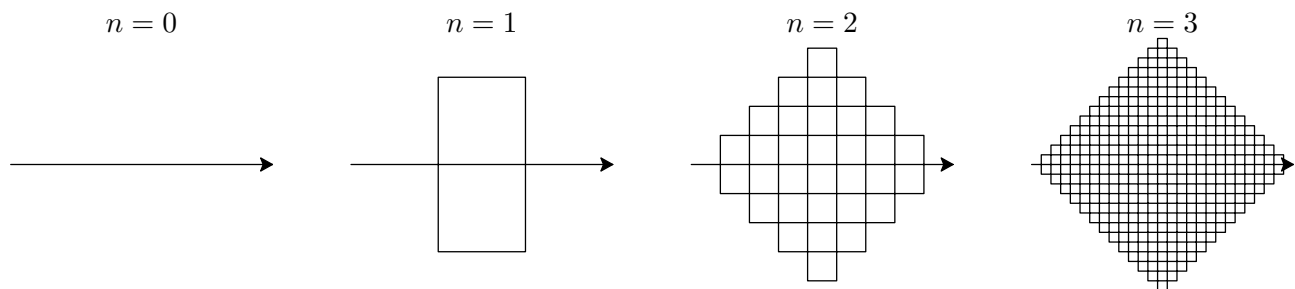
Attention à un dernier piège : après avoir tracé une tuile, il est important de faire tourner la tortue pour qu'elle soit bien dans le même sens qu'au départ (observez bien la tortue du cas $n = 1$ ci-dessus). Cela est nécessaire pour que l'appel récursif fonctionne correctement, et qu'à chaque étape la tortue se retrouve bien exactement dans la même position, qu'elle ait tracé juste avant une ligne droite ou bien une tuile plus petite.

Exercice 12.8

Écrire les fonctions `tuile_g(n, L)` et `tuile_d(n, L)`, puis tester en appelant l'une des deux au choix.

IV La courbe de Peano

C'est la courbe obtenue par la succession suivante d'étapes.



Là encore pour $n = 0$ il s'agit d'une simple ligne droite. Pour $n = 1$ le motif peut être obtenu en parcourant cette ligne dans divers ordres possibles, quitte à repasser plusieurs fois au même point. À chaque étape, chaque segment est remplacé par une courbe de Peano de taille divisée par 3.

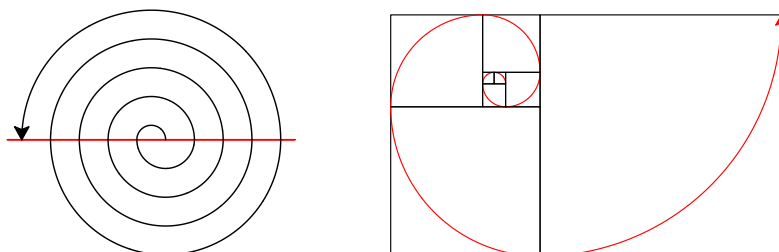
Exercice 12.9

Écrire la fonction `peano(n, L)` qui trace la courbe de Peano à l'étape n , en partant d'une longueur L .

On peut démontrer que « à la limite », la courbe obtenue remplit réellement tout un carré. On trace une courbe de dimension 1 remplissant une surface de dimension 2 !

V La spirale de Fibonacci

Il n'y a ici plus de récursivité. Une spirale simple est obtenue par une succession de morceaux de cercles, dont le diamètre augmente régulièrement. Dans le cas le plus simple, c'est une succession de demi-cercles, dont les rayons forment une progression arithmétique. Un autre cas célèbre est la spirale de Fibonacci, formée de quarts de cercles dont les rayons suivent la suite de Fibonacci.



Exercice 12.10

- Écrire une fonction `spirale(n)` qui trace une spirale simple, avec n demi-cercles consécutifs. On pourra par exemple augmenter les rayons de à chaque itération.
- Écrire la fonction `spirale_fibonacci(n)` qui trace la spirale avec n quarts de cercles dont les rayons sont proportionnels aux termes de la suite de Fibonacci.

TP 13

Révisions et consolidation 2

Exercice 13.1 *Nouvel an*

1. Écrire une fonction **récursive** `decompte(n)` qui affiche un décompte puis souhaite la bonne année. Par exemple avec $n = 5$ on veut le résultat suivant :

```
>>> decompte(5)
5
4
3
2
1
Bonne année !!!
```

2. Que se passe-t-il si dans le programme on inverse l'ordre des lignes `print` et `decompte(n-1)` ?

Exercice 13.2 *Classique classique*

1. Écrire une fonction **factoriel** `(n)`, qui prend en argument un entier $n \in \mathbb{N}$ et renvoie la valeur de $n!$, de **façon récursive**.
2. Pour tous entiers $n \in \mathbb{N}$ et $k \in \mathbb{N}$, le nombre d'arrangements A_n^k est défini par

$$A_n^k = \begin{cases} 0 & \text{si } k > n \\ 1 & \text{si } k = 0 \\ \frac{n!}{(n-k)!} = n \times (n-1) \times \dots \times (n-k+1) & \text{sinon} \end{cases}$$

Écrire une fonction **arrangement** `(n, k)` qui calcule le nombre d'arrangements **en utilisant une boucle for**.

3. Pouvez-vous écrire cette même fonction de façon récursive ?
4. Selon le *paradoxe des anniversaires*, le nombre de façons d'attribuer à k personnes leur date d'anniversaire parmi 365 jours de façon à ce qu'au moins deux personnes soient nées le même jour est égal à $365^k - A_{365}^k$ (c'est le complémentaire de : attribuer à k personnes des dates toutes différentes parmi 365). La probabilité qu'au moins deux personnes aient la même date d'anniversaire est donc

$$p_k = \frac{1}{365^k} (365^k - A_{365}^k) = 1 - \frac{A_{365}^k}{365^k} = 1 - \left(\frac{365}{365} \times \frac{364}{365} \times \dots \times \frac{365-k+1}{365} \right)$$

Écrire une fonction **anniversaires** `(n)` qui affiche, pour k de 1 à n , à la fois le nombre k et la probabilité p_k ci-dessus ; et tester avec $n = 50$.

Remarque. On trouve les fonctions suivantes dans le module `math`, à importer avec `import math` :

- `factorial(n)` : la factorielle de n .
- `perm(n, k)` : nombre d'arrangements de k objets parmi n , appelé en anglais *nombre de permutations*.
- `comb(n, k)` : coefficient binomial $\binom{n}{k}$, appelé en anglais *nombre de combinaisons*.

Si on peut se permettre de les utiliser parfois, il s'agit d'une question très très classique de savoir les ré-écrire. Pour que le programme soit efficace, il ne faut pas calculer $\frac{n!}{(n-k)!}$ en calculant la factorielle de chacun de ces deux termes puis en divisant — cela fait apparaître des nombres très très grands alors que beaucoup de termes se simplifient dans la fraction — mais l'écrire comme un seul produit.

Exercice 13.3 *Palindromes*

On rappelle qu'un mot est un **palindrome** s'il se lit aussi bien de gauche à droite que de droite à gauche, par exemple le mot "kayak" ou le prénom "anna".

On souhaite écrire une fonction `est_palindrome(m)` qui prend en argument une chaîne de caractères `m` et qui renvoie `True` si `m` est un palindrome et `False` sinon. Mais dans ce TP, on souhaite que la fonction soit récursive... La condition d'être un palindrome se formule bien récursivement à partir du premier caractère `m[0]`, du dernier caractère `m[-1]`, et du mot restant (tranche) `m[1:-1]`.

1. Au brouillon, proposer une formulation récursive du problème. Que se passe-t-il si le mot de départ était de longueur paire ? Et s'il était de longueur impaire ?
2. Écrire la fonction récursive `est_palindrome(m)`.

Exercice 13.4 *Le compteur d'anagrammes (annale DS)*

On souhaite écrire un programme pour dénombrer tous les anagrammes d'un mot. Pour cela, on a besoin d'une fonction `factoriel(n)` et de compter combien de fois apparaît chaque lettre. Pour simplifier, on suppose que nos mots sont écrits uniquement avec les 26 lettres de l'alphabet en minuscule (pas d'accents, pas de majuscules, pas d'autres signes de ponctuation) et on donne la variable Python `alphabet = "abcdefghijklmnopqrstuvwxyz"`. Il est alors pratique de numéroter les lettres à partir de 0, ainsi `a` est la lettre 0 et `z` est la lettre 25.

1. Écrire, si ce n'est pas déjà fait, la fonction `factoriel(n)`.
2. Écrire une fonction `numero(x)` qui prend en argument un caractère seul `x` et renvoie son numéro en tant que lettre de l'alphabet.
3. Écrire alors une fonction `compte(m)` qui prend en argument une chaîne de caractères `m` et qui renvoie une liste `C` de longueur exactement 26, telle que pour tout indice `j`, `C[j]` est le nombre de fois où la lettre numérotée `j` apparaît dans `m`.
4. En déduire la fonction `anagrammes(m)` qui renvoie le nombre d'anagrammes du mot `m`.

On rappelle qu'on l'obtient à partir de la factorielle de la longueur du mot, divisée par le produit des factorielles des nombres de fois que chaque lettre apparaît. Comme $0! = 1$ il est cohérent de considérer que les lettres qui n'apparaissent pas apparaissent en fait 0 fois (ce n'est pas un cas à traiter à part).

5. À partir des fonctions précédentes, écrire une fonction `sont_anagrammes(m, s)` qui renvoie `True` si les mots donnés par les chaînes de caractères `m` et `s` sont bien anagrammes l'un de l'autre, et `False` sinon.

Exercice 13.5 *Une petite parenthèse enchantée (annale DS)*

Un **mot bien parenthésé** est une chaîne de caractères formée uniquement avec des parenthèses ouvrantes "(" ou fermantes ")" telles que les parenthèses soient « bien emboîtées » au sens habituel, par exemple "(())()" ou bien "()()". À l'inverse, les mots ")()(" ou bien "(())" sont mal parenthésés. On note, pour tout $n \in \mathbb{N}$, C_n le nombre de mots bien parenthésés formés avec n **paires** de parenthèses ouvrantes-fermantes. On pose $C_0 = 1$ (le mot vide "" est considéré comme bien parenthésé) et $C_1 = 1$ (le mot "()" est l'unique mot bien parenthésé avec une seule paire de parenthèses).

- Lister les mots bien parenthésés avec $n = 2$ puis $n = 3$ paires de parenthèses ouvrantes-fermantes.
- Justifier que tout mot bien parenthésé m peut s'écrire de façon unique comme $m = "(s)t"$ où les mots s et t sont eux-mêmes bien parenthésés.
- En déduire que le nombre C_n vérifie la relation de récurrence :

$$\forall n \geq 1, \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

- Pour calculer le nombre C_n , sera-t-il à votre avis plus efficace d'écrire :
 - Une fonction récursive ?
 - Une fonction itérative qui calcule le terme C_n ?
 - Une fonction itérative qui calcule la liste de tous les C_n ?

Écrire cette fonction `C(n)`.

- Vérifier $C_4 = 14$ puis donner tous les mots bien parenthésés formés avec 4 paires de parenthèses ouvrantes-fermantes.

Remarque. Les nombre C_n sont connus sous le nom de *nombre de Catalan* et interviennent dans de nombreux problèmes de dénombrement.

Exercice 13.6 *Mots de Fibonacci (TD)*

On s'intéresse aux suites de n caractères "0" ou "1" telles qu'il n'y ait pas deux "1" consécutifs. Ces suites sont obtenues de deux façons :

- Soit à partir d'une suite de longueur $n - 1$, à laquelle on rajoute comme premier terme un "0",
- Soit à partir d'une suite de longueur $n - 2$, à laquelle on rajoute "10" au début.

Ainsi le nombre u_n de telles suites vérifie la relation de Fibonacci $u_n = u_{n-1} + u_{n-2}$.

Le but cette fois-ci est d'écrire une fonction `suites(n)` qui produit la liste de tous les mots qu'on peut obtenir avec n caractères. Pour l'écrire façon récursive, l'appel `suites(n)` va appeler à la fois `suites(n-1)` et `suites(n-2)` et récupérer leurs résultats dans des variables (disons L et M) et former une nouvelle liste P, au départ vide puis à remplir avec des méthodes `append`, selon le procédé ci-dessus. On obtient par exemple :

```
>>> suites(5)
['00000', '00001', '00010', '00100', '00101', '01000', '01001', '01010', '10000',
'10001', '10010', '10100', '10101']
```

Ici il y a bien 13 mots, et le nombre 13 fait bien partie de la suite de Fibonacci.

Écrire cette fonctions `suites(n)`.

Exercice 13.7 (*) *Générer les anagrammes*

1. Écrire une fonction `anagrammesAB(a, b)` qui renvoie la liste de tous les anagrammes qu'on peut produire avec seulement les lettres A et B, en utilisant a fois la lettre A et b fois la lettre B.

Récursivement, ces anagrammes sont tous obtenus en démarrant par la lettre A et en la concaténant à tous les anagrammes possibles avec autant de B mais $a - 1$ lettres A ; ou bien en démarrant par B concaténé à tous les anagrammes possibles avec autant de A mais $b - 1$ lettres B. Voici par exemple la liste des anagrammes sur 3 lettres A et 2 lettres B :

```
>>> anagrammesAB(3, 2)
['AAABB', 'AABAB', 'AABBA', 'ABAAB', 'ABABA', 'ABBAA', 'BAAAB', 'BAABA', 'BABAA',
'BBAAA']
```

2. Plus généralement, écrire une fonction `liste_anagrammes(C)` qui prend en argument une liste C de longueur 26 (comme dans l'exercice 4), donnant combien de fois doit apparaître chaque lettre de l'alphabet, et qui renvoie la liste de tous les anagrammes possibles sur cet ensemble de lettres.

Exercice 13.8 ()** *Permutations*

Écrire une fonction `permutations(n)` qui renvoie une liste de toutes les permutations possibles de l'ensemble $\llbracket 1, n \rrbracket$. On doit par exemple avoir

```
>>> permutations(3)
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Récursivement, un ordre naturel consiste à récupérer la liste des permutations de $\llbracket 1, n - 1 \rrbracket$ et à insérer, dans chacune de ces permutations, le nombre n à chacune des positions possibles ; l'ordre obtenu pour l'énumération des permutations dépend de l'ordre des opérations effectuées. Pour « insérer » on pourra utiliser à la fois les tranches `L[a:b]` et l'opération de concaténation `+` entre listes.

TP 14

Numpy et Matplotlib

Nous introduisons deux bibliothèques qui sont d'utilité fondamentale dans toutes les sciences des données (traiter des grands tableaux, matrices, avec des millions de données, faire des calculs et des statistiques dessus) et qui contribuent au succès croissant de Python dans ces domaines. De plus, ce TP fait le lien avec les cours de mathématiques à la fois pour les matrices et pour les fonctions.

Pour tout le TP, on peut écrire et exécuter une fois pour toute au début

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

On pourra se servir des documents suivants :

- <https://www.concours-agro-veto.fr/sites/default/files/media/2025-10/polypytho.pdf> : Aide-mémoire Python distribué au concours Agro-Véto.
- <https://matplotlib.org/cheatsheets/cheatsheets.pdf> : Aide-mémoire de la bibliothèque Matplotlib (un peu compliqué, mais illustre bien toutes les possibilités).

I Tableaux numpy

La bibliothèque Numpy introduit un nouveau type d'objet qu'on appellera **tableau**. Ceux-ci ressemblent en apparence beaucoup aux listes, mais leur fonctionnement interne est bien différent. Ils sont notamment très efficaces dans le cas où ils contiennent des millions de données, et cela nécessite de se pencher un peu plus sur le fonctionnement interne de l'ordinateur pour bien comprendre.

I.1 Aperçu sur les tableaux à une dimension

Les tableaux sont des objets du type `ndarray`. On peut les créer avec les fonctions suivantes :

- Conversion depuis une liste : `np.array(L)`

```
>>> X = np.array([1, 3, 5, 7])
>>> print(X)
[1 3 5 7]
```

- Tableau de n zéros : `np.zeros(n)`

```
>>> X = np.zeros(10)
>>> print(X)
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

- Tableau d'entiers consécutifs : `np.arange(a, b)` ou `np.arange(n)`, avec la même syntaxe que `range`

```
>>> X = np.arange(10)
>>> print(X)
[0 1 2 3 4 5 6 7 8 9]
```

- Tableau de n valeurs « linéairement espacées » entre deux bornes a et b : `np.linspace(a, b, n)`

```
>>> X = np.linspace(2, 3, 11)
>>> print(X)
[2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0]
```

Comme avec les listes, on peut :

- Demander la longueur du tableau : `len(X)`,
- Accéder directement au i -ème élément : `X[i]`, numérotés de 0 à $n - 1$ comme d'habitude,
- Trancher le tableau : `X[a:b]` est le tableau constitué des éléments d'indice i tel que $a \leq i < b$.

Il existe aussi $X[a:]$ (à partir de l'indice a) et $X[:b]$ (jusqu'à l'indice b). Les indices négatifs reviennent à la fin : $X[-1]$ est le dernier élément du tableau. Les tranches sont compatibles avec les indices négatifs, ainsi $X[:-1]$ est la tableau sans son dernier élément, et $X[1:]$ sans le premier.

Quelles sont alors les différences avec les listes Python ?

- Les tableaux Numpy sont représentés en mémoire comme un unique bloc, réservé dès le départ, dans lequel les valeurs sont posées exactement les unes à côté des autres dans des cases consécutives et de même taille. Cela permet à l'ordinateur de calculer directement l'adresse mémoire de chaque élément (chaque case de la mémoire possède une adresse, comme des maisons dans une très très longue rue) et d'y accéder rapidement. Les listes Python ne sont pas aussi efficaces et les éléments sont parfois rangés « en vrac ».
- Les éléments du tableau ont un **type** et doivent tous avoir le même type. Ce type détermine à la fois la place qu'occupe chaque élément en mémoire et donc la taille des cases (par exemple un entier 8 bits occupe 1 octet et peut contenir 256 valeurs ; mais un entier 64 bits occupe 8 octets et peut contenir $2^{64} \approx 1,8 \cdot 10^{19}$ valeurs), et comment le nombre est représenté en mémoire (avec 8 bits sans signe on a tous les entiers de 0 à 255, mais avec signe on peut aller de -128 à $+127$; les nombres à virgule flottante ont une représentation encore bien différente dans un espace de 64 bits).
- À cause de cette structure, les tableaux ont une **taille fixe**, déterminée à leur création. On ne peut pas si facilement faire un **append** ou insérer des éléments en plein milieu. En contrepartie, ils sont **compacts** et **efficaces** : si on choisit le bon type adapté aux données à traiter alors aucune place n'est perdue et les opérations sont effectuées le plus rapidement possible.

On accède au type avec la variable $X.dtype$, et à la place occupée en mémoire (en octets) avec la variable $X.nbytes$, observez :

```
>>> X = np.array([1, 3, 5])
>>> type(X)
<class 'numpy.ndarray'>
# X est un tableau numpy
>>> X.dtype
dtype('int64')
# entiers codés sur 64 bits, soit 8 octets
>>> len(X)
3
# 3 éléments
>>> X.nbytes
24
# total : 3 * 8 = 24 octets occupés en mémoire
```

Ou bien :

```
>>> X = np.linspace(0, 1, 20)
>>> print(X)
[0.          0.05263158 0.10526316 0.15789474 0.21052632 0.26315789
 0.31578947 0.36842105 0.42105263 0.47368421 0.52631579 0.57894737
 0.63157895 0.68421053 0.73684211 0.78947368 0.84210526 0.89473684
 0.94736842 1.          ]
>>> X.dtype
dtype('float64')
# nombres à virgule flottante sur 64 bits soit 8 octets
>>> len(X)
20
# 20 valeurs
>>> X.nbytes
160
# total : 8 * 20 octets
```

Ou encore :

```
>>> Z = np.arange(10, dtype="uint8")
>>> print(Z)
[0 1 2 3 4 5 6 7 8 9]
>>> Z.dtype
dtype('uint8')
# entiers non-signés sur 8 bits, soit 1 octet
>>> Z.nbytes
10
# exactement 10 octets au total
```

Ce dernier tableau pose problème si on veut y stocker des valeurs au-delà de 255...

Exercice 14.1

Tester les deux lignes suivantes :

```
>>> X = np.arange(300, dtype="uint8")
>>> print(X)
```

Que se passe-t-il ? Ré-essayer en remplaçant "uint8" par "int8" puis par "int64".

I.2 Les opérations vectorielles

Les opérations mathématiques habituelles $+$, $*$, etc ont été reprogrammées pour agir directement sur les tableaux Numpy, en effectuant toutes leurs opérations « case par case ». Observez :

```
>>> X = np.array([1, 3, 5])
>>> Y = np.array([6, -3, 8])
>>> X + Y
array([ 7,  0, 13])
>>> X * Y
array([ 6, -9, 40])
>>> -X
array([-1, -3, -5])
>>> Y**2
array([36,  9, 64])
```

L'intérêt de ces opérations — que l'on sait faire sur les listes avec une banale boucle `for` — est qu'elles s'exécutent beaucoup plus rapidement pour l'ordinateur. De façon très simplifiée, l'instruction est comprise « d'un seul coup » par le processeur (au lieu d'exécuter une boucle `for` et de devoir décoder les instructions à chaque étape) et tire parti au mieux de toutes les optimisations possibles pour calculer rapidement.

On les appelle ici des **opérations vectorielles**, où le mot « vecteur » est synonyme de tableau de nombres (ou en mathématiques : élément de \mathbb{R}^n). Ce sont des opérations qui agissent sur des vecteurs et non pas simplement sur des nombres.

La bibliothèque Numpy contient aussi de nombreuses fonctions mathématiques usuelles qui s'appliquent directement à chaque case d'un tableau : `np.exp()`, `np.sin()`, `np.cos()`, `np.arctan()`, `np.log()`, `np.sqrt()` (racine carrée), ainsi que des constantes comme `np.pi` (nombre π)... Ces opérations vectorielles s'exécutent d'un ordre de grandeur du millier de fois plus rapide que de faire une boucle Python pour les appliquer sur chaque élément.

En pratique, elles seront beaucoup utilisées combinées avec `np.linspace(a, b, n)` pour avoir une représentation d'une fonction sur un intervalle $[a, b]$ « échantillonnée » sur n points. Par exemple pour travailler avec la fonction exponentielle sur $[0, 1]$ en divisant cet intervalle en 100 points :

```

>>> X = np.linspace(0, 1, 100)
>>> print(X)
[0.          0.01010101  0.02020202  0.03030303  0.04040404  0.05050505
 0.06060606  0.07070707  0.08080808  0.09090909  0.1010101  0.11111111
 ...
 0.96969697  0.97979798  0.98989899  1.          ]
>>> Y = np.exp(X)
>>> print(Y)
[1.          1.0101522  1.02040746  1.03076684  1.04123139  1.05180218
 1.06248028  1.07326679  1.0841628  1.09516944  1.10628782  1.11751907
 ...
 2.6371452  2.66391802  2.69096264  2.71828183]

```

À retenir

La combinaison de `X = np.linspace(a, b, n)` puis de fonctions vectorielles appliquées à `X` permet d'obtenir une image échantillonnée sur n points d'un intervalle $[a, b]$ et d'une fonction sur cet intervalle.

II Représentations graphiques

La bibliothèque Matplotlib permet de tracer de très nombreux types de graphiques. La fonction principale que nous utiliserons est `plt.plot(X, Y)` qui prend au moins deux arguments : `X` et `Y` sont tous les deux des listes ou bien des tableaux numpy, de même taille ; `X` une liste d'abscisses et `Y` une liste d'ordonnées, pour un ensemble de points qui vont être automatiquement reliés. Ensuite, la fonction `plt.show()` permet d'afficher le graphique.

II.1 Graphes de fonctions

Pour représenter graphiquement une fonction, on a donc besoin de créer un tableau d'abscisses `X` puis un tableau des ordonnées `Y`, en utilisant toute la méthode de la section précédente. Il faut choisir manuellement le nombre de points d'échantillonnage, par exemple $n = 100$. Voici le modèle de base, pour par exemple $x \mapsto x^2 - 3x + 2$ sur $[-4, 4]$:

```

# abscisses
X = np.linspace(-4, 4, 100)
# ordonnées
Y = X**2 - 3*X + 2
# tracer et afficher
plt.plot(X, Y)
plt.show()

```

Exercice 14.2

Tracer les graphes des fonctions suivantes.

- $f_1 : x \mapsto x^3 - 5x$ sur $[-4, 4]$
- $f_2 : x \mapsto \sin(x)$ sur $[0, 2\pi]$,
- $f_3 : x \mapsto e^x - 3x + 1$ sur $[-3, 3]$,

Le nombre de points d'échantillonnage doit être choisi pour être suffisamment fin, sinon la courbe n'est pas assez lisse. Mais si on en met trop, le tableau est inutilement trop gros et le programme peut être lourd à charger. Comme ci-dessus, $n = 100$ est un bon compromis pour l'instant.

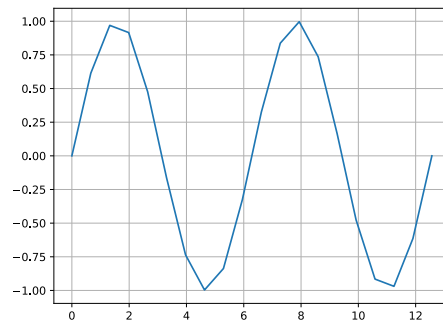


Figure 14.1. – La fonction sinus sur $[0, 4\pi]$ avec $n = 20$ points d'échantillonnage : c'est trop peu.

La bibliothèque Matplotlib contient de nombreuses options pour configurer le tracé et l'apparence de la fenêtre. Citons seulement :

- `plt.title("titre")` : donne un titre à la fenêtre.
- `plt.xlabel("titre")` : donne un titre à l'axe des x .
- `plt.ylabel("titre")` : de même pour l'axe des y .
- `plt.xlim(a, b)` : fixe les bornes sur l'axes des x entre a et b . Si on ne les fixe pas manuellement, elles sont ajustées automatiquement pour faire rentrer tout le graphe.
- `plt.ylim(a, b)` : de même pour l'axe des y .
- `plt.axis("equal")` : rend le repère orthonormé.
- `plt.grid()` : affiche une grille.
- Un troisième argument passé à `plt.plot()` sous la forme d'une chaîne de caractères permet à la fois de changer le type de point, le style de trait et la couleur. Par exemple `"+-r"` signifie « points tracés par des symboles plus, reliés par des lignes simples, couleur rouge ». Voir l'aide-mémoire ou la documentation. Ces options, et bien d'autres encore, peuvent être passées à `plt.plot()` sous forme d'arguments optionnels, par exemple `marker="+", linestyle="-", color="red"`. Consulter l'aide-mémoire Matplotlib pour la liste complète.
- Les appels successifs à `plt.plot()` enregistrent les graphiques au fur et à mesure, jusqu'à ce que `plt.show()` les affiche en les superposant. Lorsqu'on trace plusieurs graphiques sur une même figure, il est fort utile de régler manuellement les couleurs et les limites de la fenêtre.

Exercice 14.3

Améliorer le tracé des fonctions précédentes (couleur, style de ligne, titres des fenêtres et des axes).

II.2 Courbes paramétrées

Dans une **courbe paramétrée**, on trace un point de coordonnées $(x(t), y(t))$ avec un paramètre t qui varie dans un certain intervalle, et donc x, y sont tous les deux des fonctions du même t . Pour tracer une telle courbe, il faut donc échantillonner un intervalle pour t dans un tableau T , puis en déduire deux tableaux X et Y . Le modèle de base est le suivant qui trace la courbe paramétrée (cercle)

$$\begin{cases} x(t) = \cos(t) \\ y(t) = \sin(t) \end{cases}, \quad t \in [0, 2\pi]$$

```
T = np.linspace(0, 2*np.pi, 100)
X = np.cos(T)
Y = np.sin(T)
plt.plot(X, Y)
plt.show()
```

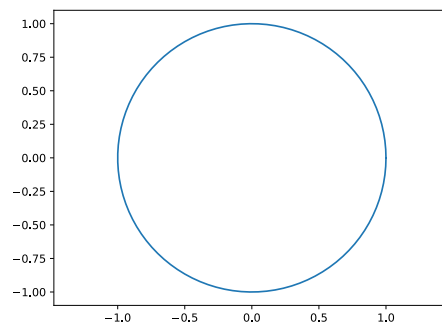


Figure 14.2. – Un cercle.

Exercice 14.4

Tracer les courbes suivantes. On inclut un lien vers le très beau site mathcurve.com recensant des centaines de courbes mathématiques.

1. Les *courbes de Lissajous* (<https://mathcurve.com/courbes2d/lissajous/lissajous.shtml>)

$$\begin{cases} x(t) = \cos(pt) \\ y(t) = \sin(qt) \end{cases}, \quad t \in [0, 2\pi]$$

pour différentes valeurs du couple (p, q) , par exemple $(2, 3)$, $(2, 5)$, $(3, 5)$. Attention au nombre de points d'échantillonnages pour que la courbe ait l'air suffisamment lisse !

2. La *cardioïde* (<https://www.mathcurve.com/courbes2d/cardioid/cardioid.shtml>)

$$\begin{cases} x(t) = (1 + \cos(t)) \cos(t) \\ y(t) = (1 + \cos(t)) \sin(t) \end{cases}, \quad t \in [-\pi, \pi]$$

3. L'*astroïde* (<https://www.mathcurve.com/courbes2d/astroid/astroid.shtml>)

$$\begin{cases} x(t) = (\cos(t))^3 \\ y(t) = (\sin(t))^3 \end{cases}, \quad t \in [-\pi, \pi]$$

4. La *strophoïde droite* (<https://www.mathcurve.com/courbes2d/strophoid/strophoid.shtml>)

$$\begin{cases} x(t) = \frac{1-t^2}{1+t^2} \\ y(t) = t \frac{1-t^2}{1+t^2} \end{cases}, \quad t \in \mathbb{R}$$

en centrant correctement la figure (axes orthonormés, limites de la fenêtre) sur la partie intéressante.

II.3 Suites

On souhaite maintenant représenter graphiquement une suite.

Exercice 14.5

Soit la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 0$ et $\forall n \in \mathbb{N}, u_{n+1} = \frac{2}{1 + 2u_n}$. On pose la fonction $f : x \mapsto \frac{2}{1 + 2x}$.

1. Représenter sur un même graphique, et en deux couleurs différentes, la courbe représentative de f et la droite d'équation $y = x$. Que conjecture-t-on quant au comportement de la suite ?
2. Écrire une fonction `suite(n)` qui renvoie la liste des n premiers termes de la suite.
3. Représenter graphiquement la suite, avec en abscisse un tableau de valeurs de n (obtenue avec `np.arange(n)`) et en ordonnée les valeurs de la suite. On pourra configurer la couleur et le type de point, qu'on ne veut certainement pas relier :

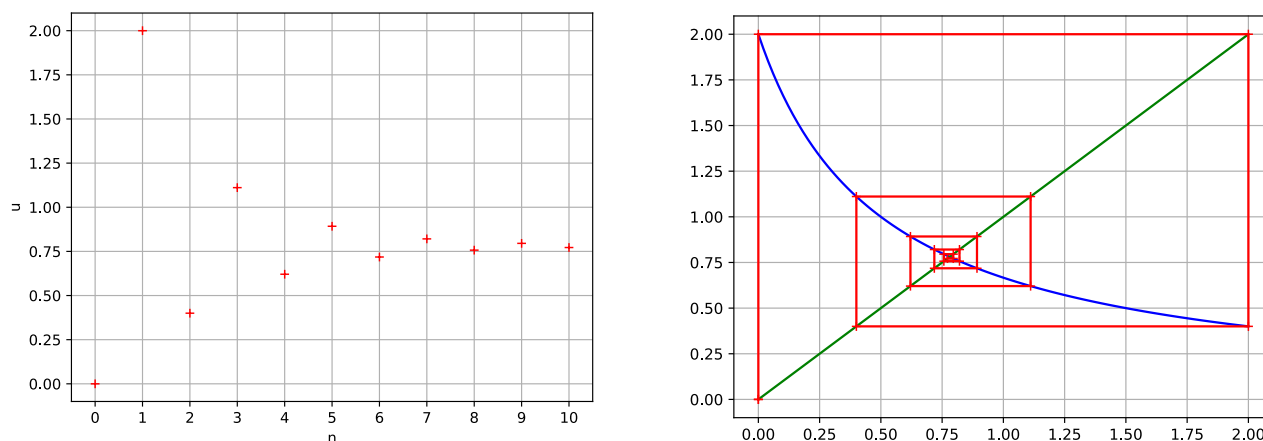


Figure 14.3. – La suite $(u_n)_{n \in \mathbb{N}}$.

4. Bonus : pouvez-vous tracer le diagramme en toile d'araignée à la question 1 ? Les points à relier ont pour coordonnées successives $(u_0, 0)$, (u_0, u_1) , (u_1, u_1) , (u_1, u_2) , (u_2, u_2) , (u_2, u_3) , etc.

III Dériver et intégrer

III.1 Dériver

Rappelons la formule suivante : pour une fonction f et un point $a \in \mathbb{R}$

$$f'(a) = \lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a}$$

Supposons maintenant que la fonction f est échantillonnée avec un tableau Numpy, c'est-à-dire qu'on dispose d'un tableau X d'abscisses et d'un tableau Y d'ordonnées. On veut échantillonner de même sa dérivée. Alors on approchera les quantités $f(x) - f(a)$, pour $x \rightarrow a$, par l'écart entre les valeurs les plus proches possibles $Y[i+1] - Y[i]$, qu'on divisera par l'écart $X[i+1] - X[i]$. On obtient un nouveau tableau Z , qu'on peut tracer en ordonnées par rapport à X pour visualiser la dérivée de f . Attention, ce Z est nécessairement de taille un de moins que Y ...

Exercice 14.6

1. Écrire une fonction `derive(X, Y)` qui prend en argument deux tableaux supposés de même taille, représentant une fonction échantillonnée, et renvoyant un tableau Z de taille un de moins représentant la dérivée.
2. Bonus : pouvez-vous l'écrire *sans* boucle, mais uniquement avec les opérations vectorielles de Numpy ?

Exercice 14.7

Pour les fonctions suivantes, tracer sur un même graphe la fonction f et sa dérivée (obtenue à l'aide de la fonction `derive` précédente, et non pas en calculant la fonction dérivée à la main), de deux couleurs différentes, éventuellement en testant diverses valeurs pour le nombre de points d'échantillonnage :

1. $x \mapsto \arctan(x)$ pour $x \in [-6, 6]$,
2. $x \mapsto \sin(x)$ pour $x \in [-2\pi, 2\pi]$.
3. $x \mapsto x^2 e^{-x}$ pour $x \in [-1, 4]$.

III.2 Intégrer

Pour intégrer une fonction (« calculer l'aire sous la courbe ») f sur un intervalle $[a, b]$, on utilise la **méthode des rectangles à gauche** qui consiste à approximer l'aire sous f entre les points d'abscisse a et $a + h$, pour h très petit, par l'aire d'un rectangle de base h et de hauteur (à peu près constante) $f(a)$, c'est-à-dire le produit $h \times f(a)$. Si on suppose que f est échantillonnée par un tableau Numpy, d'abscisse \mathbf{X} et d'ordonnée \mathbf{Y} , alors il faut multiplier les écarts $\mathbf{X}[\mathbf{i}+1] - \mathbf{X}[\mathbf{i}]$ par $\mathbf{Y}[\mathbf{i}]$ et sommer tout cela.

Exercice 14.8

1. Écrire une fonction `integre(X, Y)`
2. Bonus : pouvez-vous l'écrire *sans* boucle, mais uniquement avec les opérations Numpy ? La fonction `np.sum(X)` calcule la somme de tous les éléments d'un tableau \mathbf{X} .

Pour tester la fonction :

Exercice 14.9

1. (Mathématiques) Calculer l'intégrale suivante :

$$I = \int_0^1 \frac{4}{1+x^2} dx$$

2. (Python) Donner une approximation de cette intégrale, avec la fonction `integre`, pour de plus en plus de points d'échantillonnage (on pourra écrire une fonction `integrale(n)` qui utilise n points).

IV Annexe : tableaux à plusieurs dimensions

Le module `numpy` est aussi particulièrement efficace pour gérer des tableaux à plusieurs dimensions.

À deux dimensions, un tableau \mathbf{X} est composé de lignes et de colonnes. On accède à l'élément de la ligne i et de la colonne j (tous les deux numérotés à partir de 0, comme d'habitude) avec la syntaxe `X[i, j]`. On peut par exemple en créer avec les syntaxes suivantes :

- `np.zeros((n, p))` en lui donnant en argument un tuple (n, p) pour un tableau à n lignes et p colonnes :

```
>>> X = np.zeros((3, 5))
>>> print(X)
[[0.  0.  0.  0.  0.]
 [0.  0.  0.  0.  0.]
 [0.  0.  0.  0.  0.]
```

- Utiliser la méthode `reshape`, directement après la création du tableau, pour remodeler la forme selon les besoins :

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

Pour un tel tableau, la variable `X.shape` est le tuple (n, p) . Le nombre total de cases est exactement $n \times p$, qu'on obtient aussi avec la variable `X.size` (pour un tableau à une dimension, c'est la même chose que `len(X)`).

Le nombre de dimensions du tableau est appelé dans le vocabulaire Numpy le nombre d'**axes**. Un tableau à trois axes ressemble à un empilement de tableaux de dimension 2 et ses cases sont indicées par la syntaxe `X[i, j, k]`.

Exemple avec un tableau de forme $(3, 2, 4)$, représenté comme un empilement de 3 tableaux à 2 lignes et 3 colonnes :

```
>>> X = np.arange(24).reshape(3, 2, 4)
>>> print(X)
[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]]]
>>> X.shape
(3, 2, 4)
>>> X.size
24
```

Enfin, on peut trancher un tableau à plusieurs axes, indépendamment sur chaque axe, en séparant par des virgules les différents syntaxes de tranche :

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
>>> print(X[:3, 1:4]) # lignes 0, 1, 2, colonnes 1, 2, 3
[[ 1  2  3]
 [ 7  8  9]
 [13 14 15]]
>>> print(X[:, :1]) # toutes les lignes, seulement la première colonne
[[ 0]
 [ 6]
 [12]
 [18]]
>>> print(X[:, 0]) # toutes les lignes, colonne 0
[ 0  6 12 18]
# c'est presque pareil mais c'est un tableau à un seul axe
>>> print(X[:, ::-1]) # mêmes lignes, colonnes renversées
[[ 5  4  3  2  1  0]
 [11 10  9  8  7  6]
 [17 16 15 14 13 12]
 [23 22 21 20 19 18]]
```

Remarque. Il est intéressant de réfléchir au fait que toutes ces opérations (**reshape**, tranches) ne « bougent » rien dans la mémoire : les éléments d'un tableau à plusieurs dimensions restent toujours rangés les uns à la suite des autres, alignés sur des adresses mémoires consécutives. Ce qui change avec la forme du tableau, c'est seulement la façon de numéroté ces mêmes éléments. Observons par exemples les deux tableaux suivants :

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

Dans les deux cas, il s'agit bien de cases de mémoire rangées consécutivement et numérotées de 0 à 14. Prenons par exemple l'élément numéroté 8 : c'est *parce que* le premier est de forme (3, 5) que cet élément est en ligne 1 colonne 3, alors que dans le deuxième de forme (5, 3) il est en ligne 2 colonne 2. De même, prenons par exemple la case (2, 1) : elle correspond à l'élément 11 dans le premier tableau mais à 7 dans le deuxième, ce que l'on peut savoir uniquement en connaissant leur forme. C'est toujours un petit jeu d'arithmétique qui permet de calculer, à partir de la connaissance de la taille et de la forme d'un tableau, à quelle adresse se situera l'élément en ligne i et colonne j ; ou réciproquement, étant donnés des éléments numérotés consécutivement en mémoire, de décider à quelle ligne et à quelle colonne correspond le n -ième élément.

Les opérations de tranches, elles aussi, ne « tranchent » rien du tout dans la mémoire. Par exemple l'élément d'indice i de $\mathbf{X}[1:]$ est l'élément d'indice $i + 1$ de \mathbf{X} , là encore il s'agit seulement de quelques manipulations arithmétiques cachées à l'utilisateur.

TP 15

Matrices

L'objectif de ce TP est tout simplement d'apprendre à manipuler des matrices en Python, à la fois les représenter en Python et programmer quelques opérations usuelles dessus.

Nous avons déjà vu que la bibliothèque Numpy permettait de traiter des tableaux de toute sorte, et s'applique donc aux matrices. En fait, toutes les fonctions que nous allons écrire ici se trouvent déjà intégrées dans `numpy` ainsi que dans son sous-module `numpy.linalg`, et il faudra continuer à apprendre à les utiliser. Cependant pour notre apprentissage actuel nous allons prendre une autre approche et nous allons programmer toutes ces fonctions sans autre pré-requis que les listes Python.

I Préliminaires

I.1 Définir des matrices

Les matrices que nous allons manipuler en Python seront enregistrées comme des **listes de listes** et plus précisément comme la **liste de leurs lignes**. Par exemple la matrice

$$A = \begin{pmatrix} 8 & -1 & 7 & 4 \\ 6 & -2 & 5 & -3 \\ 7 & 2 & 0 & 7 \end{pmatrix}$$

sera représentée en Python par

```
A = [[8, -1, 7, 4], [6, -2, 5, -3], [7, 2, 0, 7]]
```

Cela est en quelque sorte une convention : on pourrait très bien décider de travailler en donnant la liste des colonnes. Mais cela est bien pratique ! En effet dans notre exemple `A[0]` désigne en fait le premier élément de la liste (de listes), donc la liste `[8, -1, 7, 4]`, et ainsi `A[0][0]` (c'est la même chose que s'il y avait des parenthèses : `(A[0])[0]`) est l'élément `8`, et `A[0][1]` est donc l'élément `-1` etc. De même `A[1]` est toute la deuxième ligne `[6, -2, 5, -3]` et donc `A[1][0] = 6`, `A[1][1] = -2`, `A[1][2] = 5`. Ainsi le coefficient d'indice (i, j) est `A[i][j]` ... à condition de, contrairement à la convention mathématique, numéroter les indices à partir de 0 !

Exercice 15.1

À partir de la fonction `len()`, comment obtient-on le nombre de lignes et de colonnes de la matrice `A` ? Écrire la fonction `taille(A)` qui renvoie un couple formé du nombre de lignes (toujours noté n) et du nombre de colonnes (toujours p).

Pour utiliser la fonction précédente, on pourra écrire des fonctions qui commencent par `(n, p) = taille(A)` ce qui récupère le tuple des dimensions de `A`. Le premier indice, qu'on appellera souvent i , sera l'indice des lignes et variera de 0 à $n - 1$ (cela diffère de la convention mathématique mais pose peu de problèmes en pratique) et le deuxième indice, qu'on appellera souvent j , sera celui des colonnes et variera entre 0 et $p - 1$.

I.2 Créer des nouvelles matrices

Nous aurons besoin de pouvoir créer des nouvelles matrices de taille donnée, et en particulier d'avoir une fonction `matrice_nulle(n, p)` qui crée une nouvelle matrice à n lignes et p colonnes remplie de zéros.

L'idée la plus simple pour créer par exemple une nouvelle matrice à 3 lignes et 4 colonnes serait d'écrire

```
>>> A = [[0] * 4] * 3
>>> print(A)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

ainsi les listes à l'intérieur sont de taille 4 remplies de zéros, et on les répète 3 fois.

Malheureusement, cela pose un petit problème...

```
>>> A[0][0] = -1
>>> print(A)
[[-1, 0, 0, 0], [-1, 0, 0, 0], [-1, 0, 0, 0]]
```

En fait, la syntaxe ci-dessus crée bien une ligne `[0] * 4` de zéros, puis ne recopie pas la ligne mais uniquement la **référence** à cette ligne. Les trois lignes de `A` deviennent des références à la **même** liste `[0, 0, 0, 0]`, ainsi toute modification sur une ligne provoque une modification sur l'autre ligne. L'auteur de ce TP s'est lui-même fait piéger lors de son apprentissage de Python.

La syntaxe des listes en compréhension, elle, permet toujours de créer des nouvelles listes « fraîches » (sans dépendance entre les éléments). On utilisera donc la syntaxe

```
>>> A = [[0 for _ in range(4)] for _ in range(3)]
```

qui crée une liste de quatre zéros (on dira « à l'intérieur »), puis répète cela trois fois. Plus généralement, on **donne** la fonction suivante qui crée une matrice nulle de dimensions (n, p) :

```
def matrice_nulle(n, p):
    return [[0 for _ in range(p)] for _ in range(n)]
```

De même, on fera attention à ce qu'écrire `B = A` ne crée pas une copie de `A` mais copie seulement la référence, et toutes les modifications de `B` vont alors affecter `A`. Pour écrire nos fonctions ce n'est en général pas le comportement voulu, donc nous commençons toujours par créer une nouvelle matrice nulle toute fraîche que nous remplissons au fur et à mesure.

Enfin, rappelons que l'outil essentiel pour parcourir une matrice et effectuer une opération sur chaque coefficient un par un est la double boucle, une syntaxe telle que :

```
for i in range(n):
    for j in range(p):
        B[i][j] = ... A[i][j] ...
```

Plus précisément, ici elle parcourt les lignes et, pour chaque ligne, parcourt les colonnes. Si on échange l'ordre des deux boucles, alors on parcourt les colonnes et, pour chaque colonne, toutes les lignes. Dans la plupart des fonctions de la partie II cet ordre de parcours n'a pas d'importance car de toute façon il faut effectuer les opérations sur tous les coefficients.

II Exercices

Toutes les fonctions sont à compléter dans le fichier ci-joint. Elles commencent par récupérer la taille des matrices données en argument, éventuellement vérifier la compatibilité des tailles pour les opérations à effectuer, puis elles créent une *nouvelle* matrice pour contenir le résultat, et la remplissent peu à peu.

II.1 Créer des matrices

Exercice 15.2

Écrire la fonction `identité(n)` qui crée la matrice identité de taille n .

Exercice 15.3

Écrire la fonction `diagonale(L)` qui prend en argument une liste (simple) de coefficients et qui crée une matrice diagonale, en plaçant les coefficients donnés dans `L` sur la diagonale.

Par exemple, on veut que l'appel `diagonale([3, 5, 7])` produise la matrice $\begin{pmatrix} 3 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 7 \end{pmatrix}$.

II.2 Opérations sur les matrices

Exercice 15.4

Écrire la fonction `somme(A, B)` qui calcule la somme des matrices A et B .

Exercice 15.5

Écrire la fonction `produit_constant(a, A)` qui calcule la matrice aA (où a est un nombre réel et A est une matrice).

Exercice 15.6

1. Écrire la fonction `coeff_produit(A, B, i, j)` qui calcule le coefficient d'indice (i, j) du produit de matrices AB .
2. En déduire la fonction `produit(A, B)` qui calcule le produit de matrices AB .
3. Bonus : pouvez-vous écrire directement la fonction `produit`, sans fonction intermédiaire ?

Exercice 15.7

Écrire la fonction `puissance(A, N)` qui calcule la puissance A^N (où N est un entier positif). On rappelle que A^0 est la matrice identité (de la même taille que A) et que $A^N = A \cdot A^{N-1}$. On pourra choisir entre une méthode itérative et une méthode récursive...

Remarque. Ici plus encore, l'algorithme des puissances rapides (TP 10 : Récursivité, exercice 5) est particulièrement important ; on rappelle que celui-ci consiste à écrire $A^N = (A^{N/2})^2$ si N est pair et $A^N = AA^{N-1}$ sinon, par exemple $A^8 = ((A^2)^2)^2$ se calcule avec seulement 3 multiplications de matrices. En effet un produit de matrices est toujours une opération lourde, nécessitant beaucoup de calculs de produits de coefficients entre eux puis de sommes, et les matrices utilisées pour modéliser finement des phénomènes physiques peuvent avoir des milliers de coefficients. Il est donc crucial de calculer des puissances en minimisant le nombre d'opérations de produits de matrices qu'on va effectuer.

II.3 Quelques tests

Exercice 15.8

Écrire une fonction `est_diagonale(A)` qui renvoie `True` si la matrice A est diagonale, et `False` sinon.

Exercice 15.9

Écrire une fonction `est_triangulaire_supérieure(A)` qui renvoie `True` si la matrice A est triangulaire supérieure, et `False` sinon.

III Le pivot de Gauss

L'objectif ultime serait d'écrire un programme capable de calculer l'inverse d'une matrice, ou du moins dans un premier temps d'échelonner une matrice en appliquant l'algorithme du pivot de Gauss. On obtiendra le rang au passage.

Cela nécessite tout d'abord de programmer les opérations élémentaires. Pour pouvoir ré-utiliser facilement les fonctions, cette fois nous avons besoin qu'elles modifient la matrice passée en argument au lieu de créer une nouvelle copie.

Exercice 15.10

Écrire les fonctions suivantes, prenant en argument une matrice A et **qui modifient directement la matrice** (sans en créer une nouvelle) :

1. `échange(A, i, j)` : échange les lignes i et j de A (opération $L_i \leftrightarrow L_j$).
2. `combinaison(A, a, i, b, j)` : opération $L_i \leftarrow aL_i + bL_j$.
3. `dilate(A, a, i)` : opération $L_i \leftarrow aL_i$.

Avant de passer à l'échelonnage, il reste une fonction manquante qu'on écrira à part : celle pour l'étape de recherche d'un pivot. En effet, quand on échelonne une matrice, la seule opération qui n'est pas complètement automatique est celle où on choisit une ligne qu'on va échanger avec la ligne sur laquelle on travaille pour éliminer une variable x_j . À la main, on choisit une ligne contenant devant x_j un coefficient sympathique (c'est-à-dire 1). En général, la seule contrainte importante est que ce coefficient soit non-nul, sinon on ne peut pas l'utiliser pour éliminer x_j dans les autres lignes. Et s'il n'y a aucun coefficient non-nul devant x_j , c'est que la variable x_j a déjà été éliminée !

On a donc besoin d'une fonction qui, en partant d'indices donnés (r, j) , cherche un coefficient non-nul *dans la colonne et en dessous à partir* de ce coefficient, et renvoie l'indice de ligne où elle l'a trouvé, et `None` si elle n'en trouve pas ; et on a besoin que les fonctions qui utilisent celle-ci vérifient si le résultat est bien un indice ligne ou bien est `None`.

Exercice 15.11

Écrire une fonction `cherche_pivot(A, r, j)` qui cherche l'indice ligne d'un coefficient non-nul dans la colonne j et dans les lignes d'indice $i \geq r$ de A . Si elle en trouve un, elle renvoie l'indice de la ligne du pivot trouvé. Sinon, elle renvoie `None`.

Tout est prêt pour appliquer le pivot de Gauss !

Exercice 15.12 (*)

Écrire une fonction `échelonne(A)` qui échelonne la matrice A .

Le programme peut se décrire ainsi :

- On démarre avec une boucle principale `for` portant sur l'indice j des colonnes, et on initialise aussi un indice r à 0 pour les lignes. Ces deux indices ne jouent pas le même rôle : à chaque étape on va toujours passer à la colonne suivante, par contre, on va passer à la ligne d'en-dessous seulement si on a bel et bien trouvé un pivot.
- On appelle alors la fonction `cherche_pivot(A, r, j)` sur la case (r, j) .
 - Si on trouve un pivot : alors on effectue les opérations « comme d'habitude », à l'aide des fonctions de l'exercice précédent. On échange la ligne du pivot avec la ligne r , puis on effectue des combinaisons pour amener tous les coefficients à 0 en-dessous de (r, j) . À la fin, on incrémente r , autrement dit on passera à la colonne suivante et on « descend d'une ligne ».
 - Si on n'en trouve pas : alors on ne fait rien du tout. La boucle `for` va faire passer à la colonne suivante, et la variable r ne bouge pas car on reste sur la même ligne.
- À la fin si tout se passe bien, la matrice est échelonnée est la variable r est en fait le **rang** de la matrice (on pourra l'afficher avec `print` avant de renvoyer la forme échelonnée).

Enfin on en vient à la fonction finale pour inverser une matrice.

Exercice 15.13 (*)

Écrire une fonction `inverse(A)` qui renvoie l'inverse de la matrice A , dans le cas où A est carrée et inversible.

- Au départ, on a besoin d'une matrice B qui est une copie de A (pour ne pas modifier A) ainsi que d'une matrice I qui est l'identité.
- **Toutes** les opérations élémentaires seront effectuées en même temps sur B et sur I . À la fin B doit être transformée en identité et I sera la matrice inverse de A .
- Dans un premier temps on échelonne B exactement comme dans l'exercice précédent, en effectuant les opérations simultanément sur I .
 - Si tout se passe bien, on trouve un pivot à chaque étape, et à la fin la matrice B est échelonnée et le dernier coefficient en bas à droite est non-nul, le rang est bien égal à la taille n . En même temps, la matrice I est triangulaire inférieure.
 - Si ce n'est pas le cas on peut éventuellement s'arrêter là et renvoyer une erreur : c'est que la matrice n'est pas inversible !
- Ensuite il faut remonter, en partant d'en bas. On a donc un deuxième morceau de la fonction avec une nouvelle (double) boucle, qui cette fois part de la fin. On utilise des opérations élémentaires $L_i \leftarrow aL_i + bL_j$ pour éliminer toute la colonne au-dessus du coefficient diagonal de B , partant d'en bas à droite, et ensuite on le met à 1 avec une dilatation. Encore une fois, ces opérations sont faites simultanément sur B et sur I . Ici il n'y a rien à « chercher », car les pivots sont déjà sur la diagonale et sont non-nuls.
- À la fin, B est d'identité et la matrice I est l'inverse de A .

TP 16

Manipulation d'images

Nous allons maintenant apprendre à manipuler les images. Les possibilités sont très larges et, en un seul TP, nous n'aurons qu'un petit aperçu du sujet. C'est aussi une très bonne idée d'utiliser ces méthodes pour le TIPE !

La première chose à faire est de récupérer le fichier `materiel.zip`. Il contient, en plus de deux versions d'un fichier `.py` à compléter, une certaine bibliothèque de photos pour le TP. Ouvrir le fichier en version niveaux de gris (pour l'instant) et exécuter au moins les deux premières cellules. Si tout se passe bien, une image en noir et blanc s'affiche.

I Introduction

I.1 Représentation d'images

Une image en couleurs à n lignes et p colonnes est manipulée par l'ordinateur comme un tableau. Chaque case du tableau s'appelle un **pixel** et contient en fait trois nombres pour former une petite case de couleur : le premier indique l'intensité de la couleur rouge, le deuxième de la couleur vert, et le troisième de la couleur bleu. On parle de codage RGB (Red, Green, Blue). Comme cela est un peu compliqué **nous travaillons d'abord avec des images en niveau de gris** auquel cas chaque pixel est représenté par un simple nombre qui indique la luminosité du pixel.

Plus précisément, nous utilisons la bibliothèque Numpy ainsi que PIL. Le code préparé charge une image avec PIL dont le nom est donné dans la variable `fichier` puis forme un tableau Numpy nommé `image` à deux dimensions, la première correspondant aux lignes et la seconde aux colonnes. Chaque pixel est codé sur un octet, soit 8 bits. Cela donne 256 valeurs possibles, tous les entiers entre 0 et 255. Ainsi la valeur de `image[i, j]` est 0 si le pixel de la ligne i colonne j est tout noir, 255 si le pixel est tout blanc, et les valeurs intermédiaires correspondent à des niveaux de gris. Le type des données du tableau est `uint8` (entier, sans signe, sur 8 bits). Comme d'habitude, si l'image a n lignes et p colonnes alors les lignes sont numérotées de 0 à $n - 1$, les colonnes de 0 à $p - 1$, ainsi le pixel $(0, 0)$ est le plus en haut à gauche et $(n - 1, p - 1)$ le plus en bas à droite.

<code>X[0, 0]</code>	<code>X[0, 1]</code>	...	<code>X[0, p-1]</code>
<code>X[1, 0]</code>	<code>X[1, 1]</code>	...	<code>X[1, p-1]</code>
<code>:</code>	<code>:</code>	<code>⋮</code>	<code>:</code>
<code>X[n-1, 0]</code>	<code>X[n-1, 1]</code>	...	<code>X[n-1, p-1]</code>

Si on travaille avec des couleurs, alors la variable `image` est un tableau à *trois* dimensions, `image[i, j, 0]` est l'intensité du rouge dans le pixel (i, j) , `image[i, j, 1]` l'intensité du vert et `image[i, j, 2]` du bleu. La variable `image.shape` est un tuple de longueur 2 (sans couleurs) ou 3 (couleurs) et dans tous les cas `image.shape[0]` est le nombre de lignes et `image.shape[1]` le nombre de colonnes. La fonction `np.zeros((n, p))` (noir et blanc) ou `np.zeros((n, p, 3))` (couleurs) est donc utilisée pour créer une image vierge de n lignes et p colonnes remplie de zéros, c'est à dire une image toute noire.

Vérifiez cela à tout moment après avoir chargé l'image :

```
>>> image
>>> image.shape
>>> image.dtype
```

Remarque. Il existe aussi des images où chaque pixel contient *quatre* nombres : en plus des composantes RGB la dernière se nomme *canal alpha* et correspond à un niveau de transparence.

I.2 Choix des images

Le dossier contient également un certain échantillon d'images. Vous pouvez choisir celle que vous voulez, idéalement en gardant la même pour toute la durée du TP (mais on pourra à n'importe quel moment copier-

coller le code de chargement d'image pour tester avec d'autres). Les images sélectionnées réunissent quelques critères : elles sont redimensionnées à une taille raisonnable (au maximum 400 pixels de côté) alors qu'une image directement sortie d'une caméra moderne va contenir des millions de pixels et le programme sera lent à les traiter ; on apprécie aussi d'avoir un objet ou paysage qui se détache nettement du décor, y compris en noir et blanc.

Avertissement

Quelques avertissements préalables. **Utiliser une image trouvée sur internet pour un travail scolaire ou universitaire, ou pour la rediffuser, sans en avoir l'autorisation est considéré comme une faute grave.** Il est donc nécessaire de s'intéresser aux droits de l'image, et de faire un usage strictement privé des images trouvées.

Les images présentes sur Wikipedia par exemple ont souvent une licence qui autorise à les ré-utiliser, voire les modifier, et les rediffuser (licence Creative Commons faites pour encourager le partage) mais toujours en **citant proprement la source** de la photo. Autant que possible, dans vos travaux, utilisez vos propres photos et créez vous-même vos propres illustrations, et même si elles ne sont pas aussi belles cela sera certainement valorisé et valorisant !

Avertissement

Les photos proposées ici sont des photos personnelles et **l'autorisation vous est donnée de les utiliser pour ce TP, mais pas de les rediffuser librement.**

I.3 Principes généraux

Les fonctions reçoivent toutes en argument un tableau nommé `X` représentant une image. Elles ne doivent pas modifier l'image elle-même mais en faire soit une copie soit une nouvelle image vierge, nommée `Y`. Ensuite, c'est le mécanisme de la double boucle `for` qui permet d'effectuer une opération sur chaque pixel, un par un :

```
for i in range(n):
    for j in range(p):
        # opération sur le pixel ligne i colonne j
        Y[i, j] = ... X[i, j] ...
```

La variable `image` est globale pour tout le fichier, et si on veut changer d'image, il faut soit ré-exécuter toute la cellule qui charge `image`, soit recopier le code là où on en a besoin. Quelques autres fonctions déjà prêtes permettent d'afficher l'image, voire d'en afficher deux l'une sur l'autre pour bien les comparer, et de sauvegarder le résultat. À vous de le tester.

Un dernier petit avertissement : les opérations sur les coefficients se font dans des entiers non-signés 8 bits, sur lesquels les valeurs au-delà de 255 reviennent à 0. Cela force parfois à tester avant un calcul si le résultat va dépasser ou non 255.

II Forme de l'image

Les premières fonctions que l'on veut coder sont le miroir horizontal et le pivotement de 90 degrés vers la droite.

La question qu'il faut se poser au brouillon est : si on prend une image `X` et qu'on veut en former l'image miroir `Y`, alors quel pixel de `X` va dans le pixel de coordonnées (i, j) de `Y` ? Vérifier au brouillon que c'est bien celui de coordonnées $(i, p - 1 - j)$, qui est sur la même ligne, mais sur la colonne symétrique par rapport à la verticale.

Exercice 16.1

Écrire la fonction `miroir(X)` qui renvoie une image obtenue à partir de `X` par miroir horizontal.

On poursuit avec la fonction qui pivote de 90 degrés vers la droite. Là encore il s'agit d'abord de trouver au brouillon : quel pixel de `X` va aller dans `Y[i, j]` ? On prendra garde aux dimensions de `Y` cette fois ! La réponse est la bonne combinaison des $i, j, n - 1 - i, p - 1 - j$.

Exercice 16.2

Écrire la fonction `pivote(X)` qui retourne une image obtenue à partir de `X` par rotation de 90 degrés sur la droite.

III Éclairage

Pour augmenter l'éclaircissement d'une image, il suffit d'ajouter à chaque pixel une valeur fixe, par exemple 50 (l'effet sera bien visible, mais on pourra ajuster cette valeur plus tard). En effet la luminosité d'un pixel est un nombre entre 0 et 255, donc les augmenter tous de la même façon ne pourra qu'augmenter la luminosité de l'image. Attention, il ne faut pas seulement ajouter 50 : si le résultat de l'addition dépasse 255 (la luminosité maximale d'un pixel), on laissera le résultat à 255. En général, on donne une valeur fixe de décalage b et on veut augmenter tous les pixels de la valeur b .

Exercice 16.3

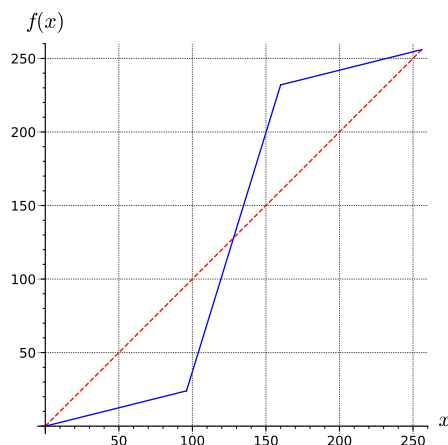
Écrire la fonction `eclaircit(X, b)` qui éclaircit l'image `X` en augmentant tous les pixels de la valeur b .

Une autre opération intéressante et très simple à programmer est le **seuillage**. Il s'agit de fixer une valeur s de seuil (typiquement $s = 127$ pour commencer) et de remplacer les pixels soit par 0 (une case noire) si la valeur est inférieure à s , soit par 255 (case blanche) si la valeur est supérieure à s . Cela doit plus ou moins faire apparaître des formes, surtout sur les objets sombres se détachant bien d'un fond clair. Tester la fonction avec différentes valeurs du seuil.

Exercice 16.4

Écrire la fonction `seuillage(X, s)` qui effectue cette opération, avec le seuil s , et tester avec différentes valeurs du seuil.

Le seuillage est un cas simplifié de l'opération de changer le contraste. Augmenter le contraste, c'est augmenter la luminosité sur les pixels déjà bien lumineux, et diminuer la luminosité de ceux déjà sombres. Pour cela, on a besoin d'une fonction $f : [0, 255] \rightarrow [0, 255]$ qui « tasse » les petites valeurs vers 0 ainsi que les grandes valeurs vers 255 (si x est petit alors $f(x) < x$, si x est grand alors $f(x) > x$), une fonction dont le graphe ressemble à ceci :



On propose pour cela la fonction

$$f : [0, 255] \longrightarrow [0, 255]$$

$$x \mapsto \begin{cases} \frac{1}{4}x & \text{si } 0 \leq x < 96 \\ 24 + \frac{13}{4}(x - 96) & \text{si } 96 \leq x < 160 \\ 232 + \frac{1}{4}(x - 160) & \text{si } 160 \leq x \leq 255 \end{cases}$$

Remarque. Il s'agit bien d'une fonction *continue*, c'est-à-dire que les valeurs en 96 et en 160 calculées avec chacune des deux formules possibles coïncident. C'est même une fonction *affine par morceaux*.

Il est facile de produire de tels exemples en décidant par quels points le graphe passe, ou avec quel pente : un morceau de fonction affine est défini uniquement par deux points de passage, ou bien par un point et une pente. Ici f est l'unique fonction découpant l'intervalle $[0, 255]$ en trois morceaux de tailles 96, 64, 96 (soit les fractions $3/8, 1/4, 3/8$ de 255) avec les pentes $1/4$ au début et à la fin.

Exercice 16.5

Écrire la fonction `contraste(X)` qui augmente le contraste de l'image X selon le procédé décrit.

Enfin pour obtenir le négatif d'une image, c'est comme son nom l'indique une simple inversion entre les pixels lumineux et les pixels sombres : la luminosité x d'un pixel deviendra $255 - x$ dans l'image négative.

Exercice 16.6

Écrire la fonction `negatif(X)` qui donne le négatif de l'image X selon ce procédé.

IV Flou

On se propose de flouter une image. L'idée est la suivante : chaque pixel sera remplacé par une moyenne des pixels autour de lui, mais pas n'importe comment. On utilisera un coefficient pour que les pixels plus proches comptent plus. La règle simple à programmer qu'on propose est : un pixel lui-même compte avec un coefficient 3, les quatre pixels sur ses côtés comptent avec un coefficient 2 et les quatre pixels qui le touchent en diagonale comptent avec un coefficient 1. Le total des coefficients fait donc 15. On représente cette opération par le tableau :

$X[i-1, j-1]$	$X[i-1, j]$	$X[i-1, j+1]$	1	2	1
$X[i, j-1]$	$X[i, j]$	$X[i, j+1]$	2	3	2
$X[i+1, j-1]$	$X[i+1, j]$	$X[i+1, j+1]$	1	2	1

Avec cette méthode on ne peut pas traiter correctement les pixels sur les bords, ceux pour lesquels il n'y a pas de pixel voisin à gauche par exemple. Dans un premier temps on n'y touche pas, l'image Y est initialisée à une copie de X et donc les pixels sur les bords ne sont pas floutés. Si on est courageux, la seule façon naturelle est de tronquer ce tableau mais alors il faut distinguer des cas selon la position du pixel de bord : par exemple si le pixel est sur le bord gauche mais pas dans les coins, il faut prendre la moyenne sur le pixel de droite, les deux au-dessus, et les deux diagonales haut-droite et bas-droite ; le total des coefficients est alors de 11.

Exercice 16.7

Écrire la fonction `flou(X)` qui floute l'image X selon ce procédé.

Remarque. Le tableau de coefficients ci-dessus s'appelle **matrice de convolution**. De nombreuses fonctions différentes de flous ou d'autres types de traitements d'images peuvent être obtenues en considérant une moyenne d'un plus grand nombre de pixels autour du pixel central et en faisant varier les coefficients. Un cas connu des graphistes est le **flou gaussien** dans lequel les poids accordés aux pixels voisins suivent une loi gaussienne en fonction de la distance au pixel central, dont on peut facilement faire varier la largeur. Ainsi plus la gaussienne est large, plus la moyenne est effectuée sur un grand nombre de pixels et plus l'effet de flou est fort ; à l'inverse, si la gaussienne est très resserrée alors le pixel central garde beaucoup plus d'importance que les autres et le flou est léger.

Remarque. Des idées similaires au flou permettent d'écrire des fonctions qui rétrécissent ou agrandissent une image. Dans un cas il s'agit de diviser le nombre de pixels, par exemple pour diviser les dimensions par deux il faut remplacer des carrés de quatre pixels par un seul pixel obtenu avec une valeur moyenne des quatre autours. Dans le second cas, par exemple pour multiplier les dimensions par deux il s'agit de remplacer chaque pixel par un carré de quatre pixels et utilisant la moyenne des pixels voisins pour lisser l'image.

IV.1 Contours

On s'intéresse maintenant au problème de la **détection de contours**. Il s'agit en quelque sorte de l'inverse du flou : quand deux pixels voisins ont des intensités différentes, il s'agit peut-être du contour d'un objet et on veut accentuer cette différence ; au contraire si les intensités sont déjà proches il s'agit simplement de la variation continue d'une même zone (par exemple le ciel) et on veut effacer cette différence. On commence donc par calculer la moyenne pondérée des pixels voisins selon la règle suivante :

-1	-1	-1
-1	8	-1
-1	-1	-1

Si les pixels d'une zone ont des intensités proches, cette moyenne va être proche de zéro. Au contraire, la valeur va être très élevée si le pixel central a une intensité nettement différente de celle de ses voisins. On choisit donc une valeur de seuil s , et si la moyenne (en valeur absolue) dépasse le seuil on mettra le pixel à 0 (tout noir) et sinon à 255 (tout blanc).

Exercice 16.8

Écrire la fonction `contours(X, s)` qui renvoie une image en noir et blanc (seulement les couleurs 0 et 255) détectant les contours de l'image X avec le seuil donné s . Tester avec différents seuils.

Remarque. En pratique, une détection automatique de contours capable de reconnaître des formes et de les mesurer précisément se fait en de nombreuses étapes : d'abord un éventuel pré-traitement pour lisser l'image, puis une détection des contours comme ici, éventuellement avec un seuil qui s'ajuste automatiquement selon la proportion des pixels qu'on veut garder ; puis éventuellement un post-traitement pour délimiter bien précisément des contours et des zones, et enfin on peut marquer nettement les régions dessinées.

V Mettez de la couleur dans votre vie !

Le fichier à compléter se trouve dans une autre variante pour traiter les images en couleur. La variable `image` est un tableau Numpy de forme $(n, p, 3)$.

Pour traiter les images, il faut alors une double boucle qui effectue les opérations sur chacune des composantes de l'image, c'est-à-dire `X[i, j, 0]`, `X[i, j, 1]` et `X[i, j, 2]`.

En fait, on peut aller un peu plus vite. Pour chaque indice (i, j) , alors `image[i, j]` est considéré lui-même comme un tableau Numpy à une seule dimension et de taille 3. Ainsi les opérations que nous effectuons sur des pixels — par exemple la fonction `negatif` qui fait simplement `255 - X[i, j]` — peuvent s'effectuer sur `X[i, j]` *en tant qu'opération vectorielle* et donc sur chacune de ses 3 composantes d'un coup. Certaines fonctions marchent donc directement, sans modification, sur les images en couleurs.

Exercice 16.9

Tester (avec un copier-coller vers le fichier à compléter en couleurs) et éventuellement adapter les fonctions précédentes en couleur.

Une fonction intéressante à adapter qui cette fois-ci manipule réellement les trois couleurs :

Exercice 16.10

Écrire une fonction `seuillage(X, a, b, c)` qui effectue un seuillage séparément sur chacune des couleurs : dans le rouge, l'intensité de la couleur sera comparée à a et le pixel sera mis soit à 0 (noir) soit à 255 (rouge pur), de même b sera le seuil de vert et c le seuil de bleu. À la fin l'image est constituée uniquement de noir, de blanc, rouge pur, vert pur, bleu pur, et des combinaisons de ces couleurs.

Enfin la dernière opération est amusante, testez-là sur diverses images !

Exercice 16.11

Écrire une fonction `fusion(X1, X2)` qui fusionne les deux images, c'est à dire que chaque pixel (i, j) du résultat sera la moyenne des pixels (i, j) de chacune des deux images. Puis tester avec deux images choisies parmi celles proposées, en recopiant deux fois le code qui permet d'ouvrir un fichier image.

Cela nécessite d'abord que les deux images soient exactement de la même taille ; sinon, il faut fusionner en coupant les images sur la plus petite des largeurs et la plus petite des hauteurs. Ou alors, écrire une fonction qui fusionne un rectangle découpé dans `X2`, défini par les coordonnées de son coin haut-gauche et par ses dimensions, dans un rectangle de `X1` défini lui aussi par les coordonnées de son coin haut-gauche (6 paramètres au total).

VI Pour aller plus loin

En vrac, quelques pistes d'améliorations à tester vous-même :

- Vérifier que toutes les fonctions marchent bien en couleur, et les corriger si besoin.
- Expérimenter avec plusieurs images différentes, éventuellement avec vos propres images.
- Manipuler des images de taille différente, par exemple pour la fusion, ou assembler plusieurs images en mosaïque. Cela nécessite quelques calculs d'indices dans un tableau.
- Sauvegarder son image en l'enregistrant, avec les fonctions déjà écrites dans le fichier joint qui font appel à `PIL.Image.fromarray` (convertit le tableau Numpy en image) et `PIL.Image.save` (enregistre un fichier image).
- *Vectorialiser* les opérations en utilisant toute la puissance de Numpy pour que l'exécution soit plus rapide, c'est à dire éliminer totalement les doubles boucles mais raisonner avec les opérations vectorielles. Sans vectorialisation, certaines fonctions sont bien trop lentes sur des images de grande taille.

TP 17

Traitement de données en tables

C'est bientôt la Saint-Valentin ! Quoi de plus romantique que d'écrire un programme Python pour manipuler la base de données de tous les prénoms des nouveaux-nés donnés en France et déclarés à l'état civil entre 2000 et 2009 ?

I Introduction

Le fichier `materiel.zip` contient un fichier `prenoms.csv` avec toute l'information dont nous avons besoin. Dans sa forme actuelle il contient 110 605 lignes, et on peut l'ouvrir avec un éditeur de texte même si cela est difficile pour travailler. Les premières lignes du fichier sont les suivantes :

```
"sexe", "prenom", "annee", "nombre"
1, AARON, 2000, 118
1, ABBAS, 2000, 7
1, ABD, 2000, 6
1, ABD-ALLAH, 2000, 6
1, ABDALLAH, 2000, 68
1, ABDEL, 2000, 65
...
2, EMMA, 2004, 6634
2, EMMA-JANE, 2004, 3
2, EMMA-LISA, 2004, 4
2, EMMA-LOU, 2004, 10
2, EMMA-LOUISE, 2004, 4
2, EMMA-ROSE, 2004, 5
2, EMMANUELA, 2004, 5
2, EMMANUELLA, 2004, 21
2, EMMANUELLE, 2004, 219
...
```

Cela représente un tableau à quatre colonnes, comme leur nom l'indique. La colonne `sexe` vaut en fait 1 pour les garçons et 2 pour les filles (le fichier ne connaît pas d'autre genre), la colonne `prenom` est en majuscule, la colonne `annee` est l'année de naissance et la colonne `nombre` le nombre de naissances avec ce prénom cette année.

Le fichier est ordonné avec d'abord les naissances de garçons de l'année 2000, puis les filles de 2000, puis les garçons de 2001, etc. Dans chacune de ces catégories, les prénoms sont classés par ordre alphabétique. Mais tout cela aura peu d'importance en pratique car nous traiterons le fichier à travers des boucles en Python ; il faut seulement se préoccuper du fait qu'un même prénom va revenir plusieurs fois, chaque année, éventuellement pour plusieurs sexes.

Il s'agit d'un **fichier CSV**, qui permet de représenter un tableau ou une base de données à la structure très simple, facile à traiter par l'ordinateur et dans une certaine mesure lisible par un humain : des colonnes décrites dans une en-tête, et dans chaque ligne les données correspondantes sont séparées par des virgules. Le mot CSV lui-même signifie *Comma-Separated Values* soit littéralement... « valeurs séparées par des virgules ». On ne s'est pas pris la tête pour trouver un nom ! On appellera chaque ligne une **entrée** de la **table** — ne pas confondre un *prénom* avec l'*entrée toute entière*.

Le code Python de démarrage ne fait qu'ouvrir ce fichier en utilisant la bibliothèque `csv` et le charge dans une grosse liste nommée `liste` :

```
>>> len(liste)
110604
```

La longueur est exactement un de moins que le nombre de lignes du fichier, à cause de l'en-tête.

Affichons un extrait en vrac de cette liste : par exemple, tout entre les indices 75 000 et 85 000 mais en sautant avec des pas de 1000 ce qui devrait afficher 10 prénoms :

```
>>> liste[75000:85000:1000]
[{'sexe': '1', 'prenom': 'DOAN', 'annee': '2007', 'nombre': '12'},
 {'sexe': '1', 'prenom': 'IONEL', 'annee': '2007', 'nombre': '3'},
 {'sexe': '1', 'prenom': 'LYESS', 'annee': '2007', 'nombre': '8'},
 {'sexe': '1', 'prenom': 'PIERRE-EMILE', 'annee': '2007', 'nombre': '6'},
 {'sexe': '1', 'prenom': 'VASCO', 'annee': '2007', 'nombre': '19'},
 {'sexe': '2', 'prenom': 'AUDELIA', 'annee': '2007', 'nombre': '6'},
 {'sexe': '2', 'prenom': 'ELUNA', 'annee': '2007', 'nombre': '3'},
 {'sexe': '2', 'prenom': 'JULIANE', 'annee': '2007', 'nombre': '81'},
 {'sexe': '2', 'prenom': 'LYDWINE', 'annee': '2007', 'nombre': '3'},
 {'sexe': '2', 'prenom': 'NATHALIE', 'annee': '2007', 'nombre': '81'}]
```

Comme chaque prénom a droit à sa propre entrée, une grande partie de cette liste est composée de prénoms rares, et toutes les variantes orthographiques ont aussi leur propre entrée ; les prénoms courant apparaissent une seule fois, mais avec une grande valeur pour la colonne `nombre`.

Chaque entrée de cette liste est nommée en Python un **dictionnaire** — nous y reviendrons dans un TP à part entière. Écrivons par exemple

```
>>> x = liste[46090]
>>> print(x)
{'sexe': '2', 'prenom': 'EMMA', 'annee': '2004', 'nombre': '6634'}
```

alors on accède au prénom correspondant par `x["prenom"]`, à l'année par `x["annee"]` et de même pour les deux autres colonnes :

```
>>> x["sexe"]
'2'
>>> x["prenom"]
'EMMA'
>>> x["annee"]
'2004'
>>> x["nombre"]
'6634'
```

Cela se passe comme si chaque entrée était une petite liste, dont les indices ne sont pas des nombres mais sont les noms des colonnes avec lesquelles nous travaillons.

En fait on traitera tout le sujet en itérant directement sur la liste `for x in liste` car il ne sera pas nécessaire de connaître l'indice du prénom dans la liste totale. Cela simplifie aussi les notations.

Quelques dernières remarques :

- Respectez les majuscules et les accents, dans les prénoms comme dans les intitulés de nos colonnes, cela a son importance. Les noms des colonnes sont en minuscule et sans accents. Chaque orthographe de prénom a sa propre entrée ; ils sont tous en majuscule. Certains comportent leurs accents, d'autres pas. Il semble que seuls les prénoms avec au moins 3 naissances apparaissent.
- Toutes les données présentes sont des chaînes de caractères, de type `str`, y compris quand elles représentent des nombres entiers. Pour accéder au nombre correspondant au prénom `x` il faut donc écrire `int(x["nombre"])`. Quant aux années, tant qu'on ne fait pas de calculs dessus, on peut les garder de type `str`, mais alors quand on donne une année il faut bien la mettre entre guillemets. Ainsi il faudra écrire des choses telles que `if x["annee"] == "2007"`. Le sexe lui est soit "1" soit "2".
- De nombreuses fonctions cherchent des prénoms en filtrant selon la valeur de `annee` ou de `sexe`. Ce n'est pas une très grosse contrainte (on pourrait enlever ces filtres), car cela revient à ajouter des conditions `if` dans les boucles.

II Explorer et compter

Au tout début nous explorons le fichier et ce qu'il contient.

Exercice 17.1

Écrire une fonction `cherche(prenom)` qui prend en argument un prénom, parcourt toute la liste, et si le prénom est trouvé, affiche toute l'entrée correspondante.

Testez-la sur votre prénom, bien entendu, et observez un peu la liste.

Exercice 17.2

Écrire une fonction `nombre(prenom, annee, sexe)` qui prend en argument un prénom, une année et un sexe, et qui si elle trouve le prénom dans la liste, renvoie le nombre de fois où il a été donné.

Exercice 17.3

Écrire une fonction `nombre_de_prenoms(annee, sexe)` qui compte le nombre total de prénoms donnés (au sens de la diversité des prénoms, sans tenir compte de combien de personnes le portent) pour l'année et le sexe passés en argument.

Maintenant il faut compter en utilisant la colonne `nombre`. Attention à bien convertir en `int` les valeurs lues !

Exercice 17.4

Écrire une fonction `nombre_avec_le_prenom(prenom)` qui prend en argument un prénom et renvoie le nombre de fois où il a été donné, sur toutes les années et éventuellement sur les deux sexes (c'est le nombre de personnes portant ce prénom néss sur cette période).

Exercice 17.5

Écrire une fonction `total_naissances(annee, sexe)` qui prend en argument une année, et qui renvoie le nombre total d'enfants nés, sur l'année et le sexe donnés.

III Représenter graphiquement

La fonction `plt.bar(X, Y)` de la bibliothèque `matplotlib.pyplot` permet de tracer un diagramme en barres qui sera bien adapté à afficher le nombre de fois qu'un prénom a été donné chaque année. La liste `X` sera celle des années à mettre en abscisses (dans un diagramme en barres, cela peut être des valeurs numériques ou bien des mots quelconques) et la liste `Y` sera celle des nombres de naissances. La documentation ou les aide-mémoires de Matplotlib permettent d'améliorer un peu l'aspect du graphique : titre avec `plt.title()`, noms des axes avec `plt.xlabel()` et `plt.ylabel()`, couleurs des barres etc.

Il faut donc créer une liste indiquant combien de fois le prénom a été donné, chaque année.

Exercice 17.6

1. Écrire une fonction `liste_nombres(prenom, sexe)` qui renvoie une liste `L` où `L[i]` est le nombre de fois où le prénom a été donné l'année `2000 + i`, par sexe donné.
2. Écrire une fonction `barre(prenom, sexe)` qui affiche un diagramme en barres du nombre de fois qu'un prénom a été donné en fonction de l'année, avec le sexe donné.

IV Filtrer

Les premières questions sont une simple révision.

Exercice 17.7

1. Écrire une fonction `maximum(annee, sexe)` qui renvoie le prénom (on a besoin de l'entrée complète) le plus donné, par année et par sexe.
2. Écrire une fonction `prenoms_au_moins(seuil, annee, sexe)` qui renvoie la liste de tous les prénoms (les entrées complètes) qui sont donnés au moins autant de fois que la valeur `seuil` passée en argument, par année et par sexe.
3. Écrire une fonction `maximum2(annee, sexe)` qui renvoie le couple formé par le prénom le plus donné et le deuxième plus donné.

Notre but serait d'avoir le TOP 10 (plus généralement, TOP n) des prénoms les plus donnés, par année et par sexe. Ce n'est pas beaucoup plus simple à programmer qu'un algorithme de tri : l'idée naturelle est une combinaison entre la fonction `maximum2` ci-dessus et l'algorithme du tri par insertion. On part d'une liste `L` égale à `[None] * 10`, puis on itère sur la liste de tous les prénoms. La liste `L` doit contenir le TOP 10 des prénoms lus jusque là, dans l'ordre, éventuellement elle se termine par des `None`. Chaque fois qu'on lit un prénom `x`, on essaie de l'insérer dans `L`, en parcourant `L` depuis la fin : si `x` est donné plus de fois que le dernier élément de `L`, on remplace ce dernier élément par `x`, puis on l'échange encore avec l'élément précédent jusqu'à ce que `L` soit bien rangée en ordre.

Exercice 17.8 (*)

Écrire la fonction `top10(annee, sexe)` qui renvoie la liste du TOP 10 des prénoms les plus donnés, sur l'année et le sexe.

V Choisir le prénom

On s'intéresse enfin au choix du prénom au hasard. Dans cette section on ne s'occupe pas des années ni des sexes (mais on peut toujours le rajouter ensuite). La fonction `randint(a, b)`, du module `random`, donne un nombre aléatoire entre les bornes a et b (bornes incluses).

Mais on ne veut pas simplement choisir une entrée de la liste au hasard : on voudrait choisir un prénom de façon proportionnelle à sa fréquence d'apparition. Cela nécessite donc d'abord de compter le nombre de naissances totales, appelons le N . Ensuite on tire au hasard un nombre entre 1 et N . L'idée à traduire dans un algorithme, qui parcourt toute la liste est la suivante...

Imaginons qu'un premier prénom soit donné 3 fois, un deuxième est donné 8 fois, et le dernier est donné 2 fois. Cela fait 13 naissances, on prend un nombre au hasard entre 1 et 13. Alors on veut choisir le premier prénom si le nombre tiré est 1, 2, 3 ; le deuxième si le nombre tiré est entre 4 et 11 ; et le troisième si le nombre tiré est 12 ou 13. Autrement dit dans une boucle on a besoin de compter les cumuls de naissances, et comparer le cumul avec le nombre tiré au hasard : dès que le cumul dépasse notre nombre choisi, on s'arrête et on considère qu'on choisit ce prénom !

Exercice 17.9

1. Écrire une fonction `choix_prenom()` qui choisit un prénom au hasard par cette méthode, et renvoie le prénom.
2. Écrire une fonction `choix_groupe(n)` qui renvoie une liste de n prénoms choisis au hasard selon cette méthode, et observez par exemple en générant toute une classe de 30 élèves !

On peut au contraire vouloir choisir un prénom rare, où *rare* signifie par exemple donné moins de 10 fois (le seuil est au choix). Cette fois, peu importe que la probabilité soit proportionnelle à la fréquence d'apparition du prénom ; mais il faut tout de même d'abord bien compter à l'avance les prénoms rares.

Exercice 17.10

Écrire la fonction `choix_prenom_rare(seuil)`, et créer une liste de 30 prénoms rares, donnés moins de `seuil` fois.

TP 18

Dictionnaires

Les **dictionnaires** sont un nouveau type permettant de contenir d'autres données, comme les listes ou les tuples. Nous en avons en fait déjà croisés au TP précédent et nous allons approfondir.

I Introduction

Lorsque nous avons écrit dans le TP précédent

```
d = {"sexe": "2", "prenom": "EMMA", "annee": "2004", "nombre": "6634"}
```

nous avons déjà affaire à un dictionnaire. Ce qu'on nomme en Python **dictionnaire** est un ensemble de **valeurs** auxquelles on accède via les **clés**. Ici les clés sont les noms **"sexe"**, **"prenom"**, **"annee"**, **"nombre"**. Les valeurs correspondantes sont obtenues avec la syntaxe `d["sexe"]`, `d["prenom"]` etc.

Cela ressemble donc fort à une liste dans laquelle les indices ne sont pas seulement des nombres entiers, mais peuvent être des chaînes de caractères. On les appelle aussi parfois des *tableaux associatifs*, car ils associent une valeur à la clé donnée. Et ils ont des applications bien pratiques.

Quelques remarques sur le fonctionnement des dictionnaires :

- L'ordre des clés n'a pas vraiment d'importance.
- Bien sûr, chaque clé ne peut apparaître qu'une seule fois, sinon cela n'a pas de sens.
- Les clés peuvent être en fait de beaucoup de types différents : chaînes de caractères, mais aussi entiers, flottants, tuples composés de ceux-ci...

Dans un dictionnaire `d`, et pour une valeur notée `k`, l'accès à `d[k]` va déclencher une erreur si `k` n'est pas une clé de `d`. Il est donc souvent utile de pouvoir tester à l'avance cette condition, avec la syntaxe `k in d` dont la négation est `k not in d` :

```
>>> "prenom" in d
True
>>> "age" in d
False
>>> "age" not in d
True
>>> d["age"]
KeyError: 'age'
```

Par contre on peut rajouter des clés au fur et à mesure :

```
>>> d["age"] = 19
>>> print(d)
{'sexe': '2', 'prenom': 'EMMA', 'annee': '2004', 'nombre': '6634', 'age': 19}
```

ou en supprimer

```
>>> del d["nombre"]
>>> print(d)
{'sexe': '2', 'prenom': 'EMMA', 'annee': '2004', 'age': 19}
```

Le dictionnaire vide est noté tout simplement `{}`. Parfois on souhaite partir d'un dictionnaire vide et ajouter des clés au fur et à mesure.

Il existe aussi, comme pour les listes, une syntaxe **en compréhension**, où on peut donner à la fois une expression pour les clés et pour les valeurs. Étudions par exemple le dictionnaire suivant

```
d = {x**2: x for x in range(10)}
```

qui donne

```
{0: 0, 1: 1, 4: 2, 9: 3, 16: 4, 25: 5, 36: 6, 49: 7, 64: 8, 81: 9}
```

dont les clés sont (certains) nombres et les valeurs correspondantes vont être leur racines carrées. Ce dictionnaire permet donc de calculer directement les racines carrées de ces nombres (et seulement ceux-là), par exemple `d[49]` donne `7` (il n'y a plus de guillemets ici : les clés sont de type `int`). Mathématiquement, un dictionnaire est une application de l'ensemble des clés vers l'ensemble des valeurs...

Les dictionnaires sont aussi utilisés par le langage Python lui-même pour maintenir des informations sur le programme en cours de fonctionnement... La fonction `globals()` renvoie un dictionnaire des variables actuellement enregistrées dans la session, affichez-le !

II Itération sur un dictionnaire

Dans le TP précédent et jusqu'à maintenant nos dictionnaire avaient tous quatre clés fixes et bien connues à l'avance ; la liste de tous les prénoms était en fait une *liste de dictionnaires*. Mais en pratique, une fonction reçoit en argument un dictionnaire et ne sait pas forcément quelles en sont les clés. Il faut donc utiliser une boucle `for` pour parcourir un à un tous les éléments du dictionnaire, tout comme on parcourt les éléments d'une liste de longueur quelconque.

Cependant il y a trois façons de faire...

Reprenons un dictionnaire :

```
d = {"sexe": "2", "prenom": "EMMA", "annee": "2004", "nombre": "6634"}
```

- **Itérer sur les clés** de `d` : c'est une boucle `for` sur l'objet `d.keys()`, qui fournit unes par unes les clés de `d`.

```
>>> for k in d.keys(): print(k)
sexe
prenom
annee
nombre
```

- **Itérer sur les valeurs** de `d` : de même, c'est une boucle `for` qui porte sur l'objet `d.values()` qui fournit les valeurs unes par unes.

```
>>> for v in d.values(): print(v)
2
EMMA
2004
6634
```

- **Itérer sur les paires** (clé, valeur) de `d` avec l'objet `d.items()`, qui fournit des tuples.

```
>>> for (k, v) in d.items(): print("clé :", k, "valeur :", v)
clé : sexe valeur : 2
clé : prenom valeur : EMMA
clé : annee valeur : 2004
clé : nombre valeur : 6634
```

Bien sûr, en pratique itérer sur les clés fonctionne toujours, puisque si on a `k` alors on a accès à `d[k]`. Mais il faut considérer que l'accès à une valeur est une opération lourde, bien plus lourde dans les dictionnaire que l'accès aux éléments d'une liste (voir l'annexe). Ainsi itérer directement sur les paires, ou sur les valeurs, est bien plus rapide que d'itérer sur les clés *puis* de chercher les valeurs correspondantes.

Dans les exercices de cette partie, aucune méthode n'est extraordinairement nouvelle. Il s'agit de boucles pour parcourir un dictionnaire et il faut seulement se poser la question du choix de l'une des trois méthodes d'itération précédentes. Ensuite, ce sont les mêmes types d'algorithmes que ceux rencontrés de nombreuses fois sur les listes.

Exercice 18.1 *Itérer sur les valeurs*

On représente une liste de courses par un dictionnaire qui donne, pour chaque produit acheté, le prix en euros.

```
courses = {"pain": 1.20, "camembert": 3.0, "salade": 1.5, "savon": 3.5}
```

1. Écrire une fonction `facture(d)` qui prend en argument un tel dictionnaire et qui calcul le montant total de la facture.
2. Écrire une fonction `est_trop_luxueux(d)` qui renvoie `True` si l'un des articles a un prix supérieur à 5 euros, et `False` sinon.

Exercice 18.2 *Itérer sur les clés*

On représente une recette de cuisine par un dictionnaire dont les clés sont les ingrédients et les valeurs, pour chaque ingrédient, sont la quantité (l'unité est variable selon l'ingrédient : gramme, millilitre, nombre). Par exemple

```
crepes = {"farine": 250, "oeufs": 4, "lait": 300, "beurre": 50, "sucre": 30}
```

1. Écrire une fonction `nombre_ingredients(d)` qui renvoie le nombre d'ingrédients différents de la recette.
2. Supposons qu'on soit allergique aux noix. Écrire une fonction `est_sans_noix(d)` qui renvoie `True` si la recette ne contient pas de noix, et `False` sinon.
3. On souhaite faire un régime. Écrire une fonction `est_sain(d)` qui renvoie `True` si la recette contient moins de 50 grammes de sucre, et `False` sinon.

Attention car il peut se produire deux situations : ou bien "sucre" sera dans les clés avec une valeur qui doit être inférieure à 50, ou bien "sucre" ne sera pas du tout dans les clés.

Exercice 18.3 *Itérer sur les couples*

On représente un porte-monnaie contenant des pièces ou des billets par un dictionnaire `d`, où `d[x]` représente le nombre de billets (ou pièces) de valeurs `x`. Par exemple, le dictionnaire

```
d = {1: 4, 2: 7, 10: 1}
```

représente un porte-monnaie avec 4 pièces de 1 euro, 7 pièces de 2 euros et 1 billet de 10 euros. Remarquez qu'on ne se préoccupe en fait pas de s'il s'agit de billets ou de pièces, ni si les valeurs de ces pièces existent réellement, tout cela pourrait fonctionner de la même façon dans d'autres systèmes monétaires que l'euro. Dans cet exemple la somme totale est de 28 euros.

Écrire une fonction `oseille(d)` qui prend en argument un tel dictionnaire représentant un porte-monnaie et qui renvoie la somme d'argent totale que cela représente.

III Problèmes

Exercice 18.4

On souhaite dénombrer les anagrammes d'un mot. Pour cela la première étape est de partir d'un mot, représenté comme une chaîne de caractères, et de compter combien de fois apparaît chaque lettre. Si on n'avait pas les dictionnaires, il faudrait savoir à l'avance avec combien de lettres on travaille (par exemple 26 — mais alors il n'y a plus de marge pour les accents ou les majuscules) et initialiser une liste de taille 26 comptant combien de fois chaque lettre apparaît. Avec les dictionnaires, nous allons pouvoir travailler avec des mots absolument quelconques et sans connaître les lettres à l'avance.

1. Écrire une fonction `compte_lettres(s)` qui prend en argument une chaîne de caractères `s` et qui renvoie un dictionnaire `d`, dont les clés sont des lettres apparaissant dans `s` et dont la valeur `d[x]` est le nombre de fois que la lettre `x` apparaît.

Pour cela on a besoin d'initialiser un dictionnaire vide au début, puis d'une boucle qui fournit une par une les lettres de `s`. Attention car à chaque lettre, il faut tester si elle est déjà dans le dictionnaire (auquel cas incrémenter la valeur), ou sinon l'ajouter dedans simplement.

2. Écrire une fonction `nombre_anagrammes(s)` qui compte le nombre d'anagrammes de `s`.

On rappelle qu'il s'agit de la factorielle du nombre de lettres de `s`, divisé par le produit des factorielles du nombre de fois que chaque lettre apparaît ; on a donc besoin d'appeler la fonction précédente et d'itérer sur le dictionnaire `d` pour calculer ce produit de factorielles. On pourra ré-écrire rapidement la fonction factorielle, ou utiliser celle fournie avec `from math import factorial`.

Exercice 18.5

On donne le dictionnaire suivant :

```
chiffres = {"zéro": 0, "un": 1, "deux": 2, "trois": 3, "quatre": 4, "cinq": 5, "six": 6,
"sept": 7, "huit": 8, "neuf": 9}
```

1. Écrire une fonction `traduit(L)` qui prend en argument une liste de mots parmi ceux-ci, et affiche successivement les chiffres correspondants.

```
>>> traduit(["deux", "huit", "trois"])
2
8
3
```

2. Écrire une fonction `nombre(L)` qui prend en argument toujours une liste de mots, et renvoie le nombre que cela forme, de type `int`.

```
>>> nombre(["deux", "huit", "trois"])
283
```

On remarque qu'étant donné un nombre N écrit avec la liste de ses chiffres $N = a_k \dots a_1 a_0$ (où a_0 est le chiffre des unités, a_1 le chiffre des dizaines, etc) alors le nombre N est égal à

$$N = (\dots(a_k \times 10 + a_{k-1}) \times 10 + \dots) \times 10 + a_0$$

par exemple $283 = (2 \times 10 + 8) \times 10 + 3$, ce qui permet de calculer N en lisant ses chiffres de gauche à droite (donc dans l'ordre naturel pour le problème que nous traitons là).

3. Bonus : étant donnée une chaîne de caractères `s`, la méthode `s.split()` « casse » la chaîne aux caractères espaces et produit une liste de mots :

```
>>> s = "deux huit trois"
>>> s.split()
['deux', 'huit', 'trois']
```

Ré-écrire la fonction `nombre` pour qu'elle prenne en argument une seule chaîne de caractères.

Exercice 18.6 (*) *Matrices creuses*

On s'intéresse à des matrices de taille (n, p) contenant une grande majorité de coefficients nuls, et quelques coefficients par-ci par-là non nuls. Plutôt que de stocker en mémoire un tableau entier de $n \times p$ cases dont la plupart vont être nulles, on représente une telle matrice par un dictionnaire A dont les clés sont des couples (i, j) ($0 \leq i < n$ et $0 \leq j < p$ comme d'habitude dans la convention informatique) et la valeur $A[(i, j)]$ est le coefficient d'indice (i, j) . Si une clé n'est pas présente, on interprète le coefficient correspondant comme étant nul (mais réciproquement, il peut y avoir des clés avec une valeur nulle). Par exemple la grosse matrice

$$A = \begin{pmatrix} 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{pmatrix}$$

sera représentée tout simplement par le dictionnaire

```
A = {(0, 2): 5, (2, 3): 7, (3, 1): -1}
```

Écrire les fonctions classiques dans ce contexte :

1. `identité(n)` : matrice identité de taille n ,
2. `somme(A, B)` : somme de deux matrices,
3. `produit(A, B)` : produit de matrices,
4. `est_triangulaire_supérieure(A)` : renvoie `True` si la matrice est triangulaire supérieure, `False` sinon.

Étonnamment, les fonctions n'ont pas besoin de connaître les tailles des matrices, et la notion n'a même pas de sens ici car on peut toujours les « prolonger par 0 » en des matrices plus grandes...

Remarque. On tombe toujours sur le même problème : il faut tester si une clé est présente ou non avant d'y accéder, et il faut considérer que le coefficient est zéro si la clé n'apparaît pas. Cela est assez ennuyeux surtout que tester si une clé est présente dans le dictionnaire est *déjà* une opération lourde, tout autant que d'accéder à la valeur (voir l'annexe). Se renseigner notamment sur les méthodes `d.setdefault(k, v)` et `d.get(k, v)` ainsi que sur le type `defaultdict` et comment cela permet de traiter élégamment ce problème.

IV Annexe : tables de hachage

Nous avons déjà dit plusieurs fois que dans une liste, il faut imaginer que les éléments sont rangés les uns à la suite des autres comme dans des cases de la mémoire. Cela permet d'accéder directement au i -ème élément de la liste.

Dans un dictionnaire, on peut imaginer naïvement ranger à la suite les clés et leurs valeurs, dans des cases aussi. Cependant les problèmes suivants se posent :

- Pour rechercher une clé du dictionnaire, et sa valeur correspondante, il est nécessaire de parcourir toutes les clés une par une, exactement comme quand on cherche un élément dans une liste. Cela peut être long s'il y a beaucoup de clés.
- De plus, si les clés sont de type chaîne de caractères, comparer les clés prend plus de temps que de comparer des nombres ; car pour comparer deux chaînes il faut comparer successivement leurs caractères uns par uns.

Ainsi cette méthode peut vite devenir très lourde et peu efficace, et pour enregistrer un dictionnaire dont les clés sont des chaînes de caractères, toutes les fonctions seront beaucoup plus lentes que celles qui travaillent sur les listes.

La solution qui a été trouvée s'appelle **table de hachage**. Elle consiste à définir une certaine fonction mathématique (assez abstraite) dite **fonction de hachage**, calculable sur **tous** les objets possibles pouvant servir de clé, dont le résultat est un simple nombre entier qui puisse nous dire où se situe la clé dans la mémoire. Cela résout le premier problème car comparer des nombres est nettement plus rapide que comparer des chaînes de

caractères, et cela résout partiellement le second problème — au minimum on peut espérer ranger les clés dans l'ordre croissant selon leur valeur de hachage, et rechercher les clés rapidement grâce à la dichotomie.

En fait un problème immédiat se pose : il y a de toute façon beaucoup plus de clés possibles que de nombres et certaines clés auront donc la même valeur par la fonction de hachage (on parle de **collision**), en termes mathématiques la fonction de hachage part d'un ensemble très gros vers un ensemble plus petit et ne peut donc pas être injective (principe des tiroirs). Cela rend la conception de tables de hachages plus subtile que ce qui est décrit ici. La fonction de hachage doit être créée de telle façon à ce que les collisions ne se produisent pas trop souvent, et si c'est le cas, si deux clés se retrouvent avec la même valeur de hachage, alors tant pis : on les stocke à la suite et on comparera les clés comme dans la méthode naïve.

En Python, la fonction de base `hash(x)` permet de connaître la valeur de hachage d'un objet `x`, même si ce nombre ne nous dit concrètement pas grand chose d'intéressant...

```
>>> x = 3
>>> hash(x)
3
# OK, pour les nombres entiers c'est eux-mêmes
>>> x = 3.14
>>> hash(x)
322818021289917443
# que faire de cette information ?
>>> x = (1, 2)
>>> hash(x)
-3550055125485641917
# stop, stop !!!
>>> x = "steak"
>>> hash(x)
5425928401636965275
# le steak est haché !
```

Tous les objets ne peuvent pas servir de clé. Imaginons un dictionnaire contenant pour clés les deux listes `L = [1, 2]` et `M = [1, 3]` (pourquoi pas), avec une valeur pour chacune.

```
d = {L: "truc", M: "machin"}
```

Puis faisons `M[1] = 2`. Les listes `L` et `M` deviennent alors égales, et devraient donc avoir la même valeur de hachage, du coup il n'y a plus qu'une seule clé ? Qu'est-ce que `d[[1, 2]]` ? Comment retrouver alors les valeurs ? En fait, cela est interdit et les listes ne sont pas hachables. Les objets hachables ne doivent jamais pouvoir être modifiés au cours du programme, et la fonction de hachage doit toujours renvoyer la même valeur pour un même objet.

```
>>> L = [1, 2]
>>> hash(L)
TypeError: unhashable type: 'list'
```

TP 19

Graphes

Les **graphes** sont des structures de données (comme les listes, tableaux, dictionnaires, ...) énormément utilisées en informatique et en mathématiques car ils sont assez simples à manipuler et modélisent de très nombreuses situations.

I Notion de graphe

Un graphe est tout simplement donné par un ensemble de sommets, reliés entre eux par des arêtes :

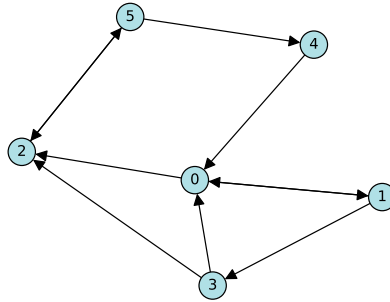


Figure 19.1. – Un graphe

Ce genre de dessin modélise de très nombreuses situations différentes :

- Une carte géographique, où les sommets sont des villes et les arêtes sont des routes reliant ces villes. La carte du métro parisien est assurément un graphe, de même que la carte des lignes de train en France.

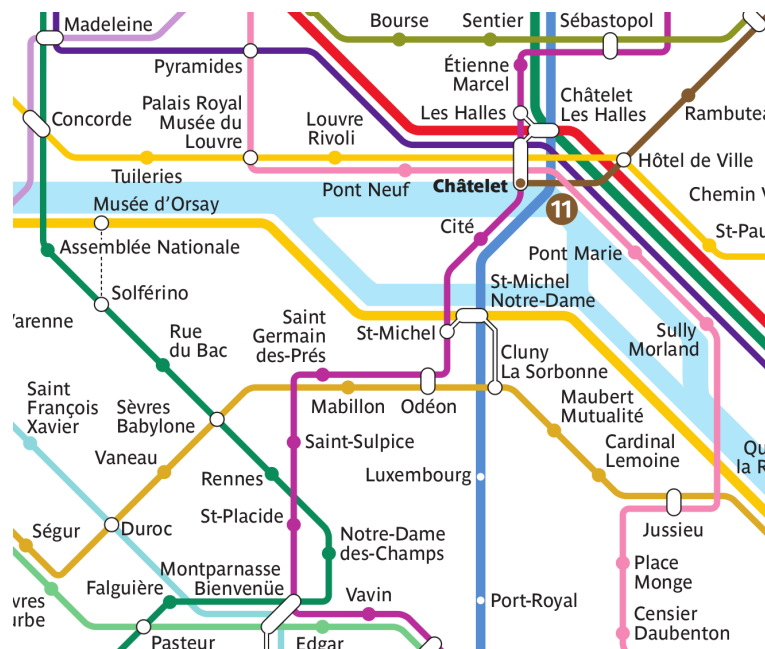


Figure 19.2. – Zoom sur un graphe bien connu. Source : RATP.

- Un réseau social, où les sommets sont des personnes et une arête entre deux personnes signifie que ces personnes sont amies, ou bien (suivant le sens de la flèche) que l'un est un *follower* de l'autre. Les gros réseaux sociaux ont absolument besoin d'algorithmes efficaces opérant sur des graphes avec des millions d'informations.

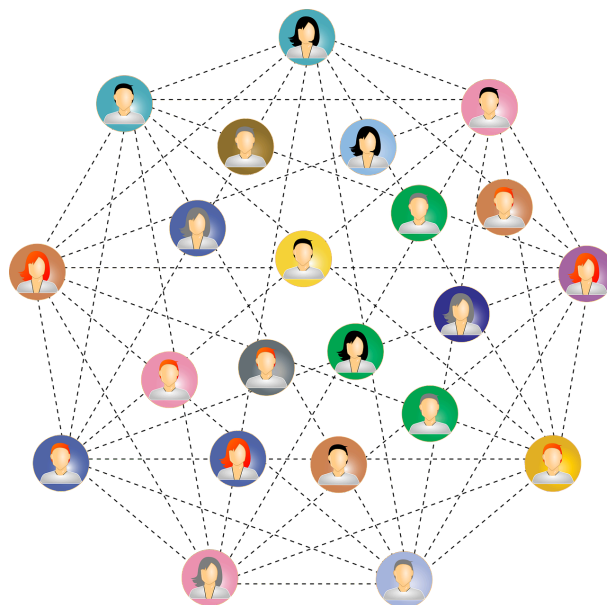


Figure 19.3. – Graphe d'un réseau social. Source : Pixabay (image libre).

- Un jeu de stratégie, où chaque sommet représente un état possible du jeu, et une arête représente une façon de passer d'un état à un autre. Jouer une partie revient à démarrer sur un sommet initial puis passer d'un sommet à un autre via des arêtes, jusqu'à arriver sur un sommet représentant une partie gagnée. Un jeu de labyrinthe peut se représenter par un graphe dont le but est d'arriver au sommet final, en partant d'un sommet initial donné.

La formalisation mathématique est celle-ci, prenant en compte la petite flèche dessinée sur les arêtes qui correspond à une orientation :

Définition

Un **graphe orienté** est donné d'un ensemble fini S (ensemble des **sommets**) et d'un sous-ensemble $A \subset S \times S$ (ensemble des **arêtes**). Si x, y sont deux sommets, la condition $(x, y) \in A$ signifie qu'il y a une arête de x vers y .

Une autre variante de graphe est celle où on ne s'occupe pas du sens des arêtes, qu'on représente donc comme un simple trait entre sommets : deux sommets x et y sont ou bien ne sont pas connectés, peu importe dans quel ordre.

Définition

Un **graphe non orienté** est un graphe $G = (S, A)$ vérifiant la condition :

$$\forall (x, y) \in S^2, \quad (x, y) \in A \Leftrightarrow (y, x) \in A$$

Autrement dit on interprète la condition $(x, y) \in A$ comme signifiant qu'il existe une arête entre les sommets x et y , et ceci est équivalent à dire qu'il existe une arête entre les sommets y et x .

Notre définition générale de graphe n'exclut pas l'existence de **boucle** : une arête d'un sommet x à lui-même, $(x, x) \in A$.

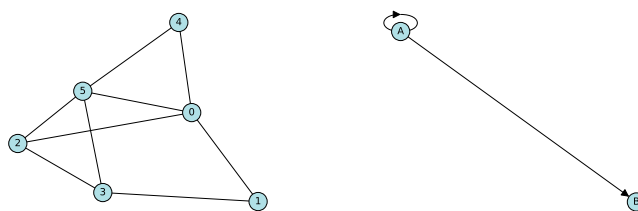


Figure 19.4. – À gauche, un graphe non orienté, représenté sans flèches.
À droite, une boucle sur le sommet A .

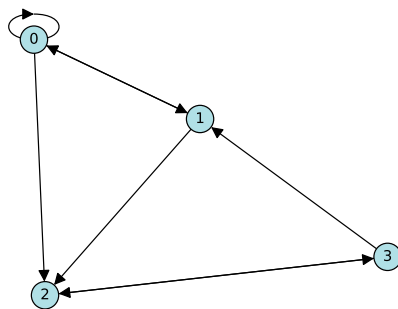
II Des graphes en Python

II.1 Représentation

Il y a plusieurs façons de représenter un graphe orienté $G = (S, A)$ en Python. On suppose qu'on a d'abord numéroté les N sommets de 0 à $N - 1$, ainsi $S = \{0, 1, \dots, N - 1\}$.

- Par **liste d'adjacence** : on donne une liste L où pour chaque indice de sommet i , $L[i]$ est la liste des sommets **vers lesquels** mène une arête issue de i . Mathématiquement c'est la liste des $\{j \in S \mid (i, j) \in A\}$.
- Par **matrice d'adjacence** : on donne une liste de listes M représentant une matrice carrée, où $M[i][j]$ vaut 1 s'il y a une arête du sommet i vers le sommet j et 0 sinon.

Prenons par exemple le graphe très simple



Alors vérifiez que la liste d'adjacence est

```
L = [[0, 1, 2], [0, 2], [3], [1, 2]]
```

et que la matrice d'adjacence est

```
M = [[1, 1, 1, 0], [1, 0, 1, 0], [0, 0, 0, 1], [0, 1, 1, 0]]
```

- Si on ne souhaite pas numéroté les sommets, alors on peut leur donner un nom et travailler comme sur les listes d'adjacence mais avec à la place un **dictionnaire d'adjacence** : un dictionnaire où chaque clé est un sommet et la valeur correspondante est la liste des sommets vers lesquels il est relié. Voici par exemple un bout de graphe non-orienté représentant des lignes de train en France :

```
trains = {"Paris": ["Versailles", "Lyon", "Angers", "Besançon", "Clermont"],
          "Versailles": ["Paris"], "Lyon": ["Paris", "Marseille", "Grenoble", "Besançon"],
          "Angers": ["Paris"], "Besançon": ["Paris", "Lyon"], "Clermont": ["Paris"], "Marseille":
          ["Lyon", "Nice"], "Grenoble": ["Lyon"], "Nice": ["Marseille"]}
```

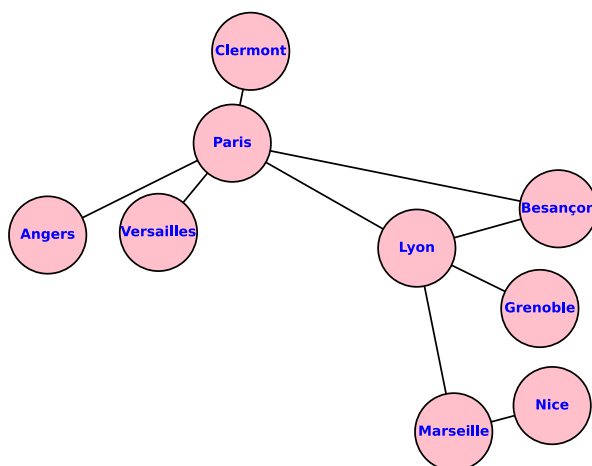


Figure 19.6. – Carte de France...

En fait, chaque représentation possible a ses avantages et ses inconvénients...

Exercice 19.1 (*Passer éventuellement et revenir plus tard*)

Écrire des fonctions

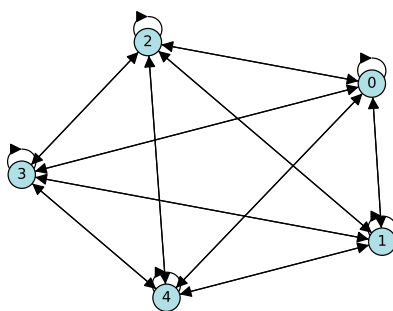
1. `liste_vers_matrice(L)` : convertit un graphe représenté par une liste d'adjacence, vers une matrice d'adjacence. On aura besoin de savoir créer une matrice nulle, et d'une double boucle pour itérer sur les listes d'adjacence de chaque sommet.
2. `matrice_vers_liste(M)` : convertit un graphe représenté par une matrice d'adjacence, vers une liste d'adjacence. On aura besoin de créer une liste de listes nulles, puis d'une double boucle pour parcourir la matrice d'adjacence et on utilisera `append` pour ajouter au fur et à mesure les sommets dans la liste d'adjacence.

II.2 Quelques graphes particuliers

Étudions maintenant quelques exemples particuliers de graphes. On choisit pour les questions suivantes de travailler avec des matrices d'adjacence.

Exercice 19.2

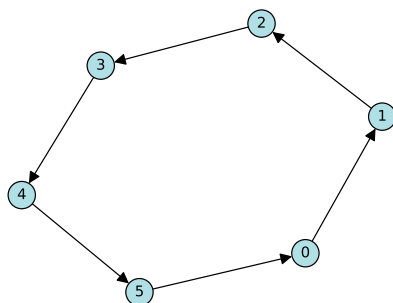
Un graphe $G = (S, A)$ est dit **complet** s'il y a des arêtes entre tous les sommets possibles, c'est-à-dire $A = S \times S$. En particulier le **graphe complet** à n sommets est le graphe sur l'ensemble des sommets $\{0, 1, \dots, n - 1\}$ qui sont tous reliés entre eux.



Écrire une fonction `graphe_complet(n)` qui renvoie la matrice d'adjacence du graphe complet à n sommets.

Exercice 19.3

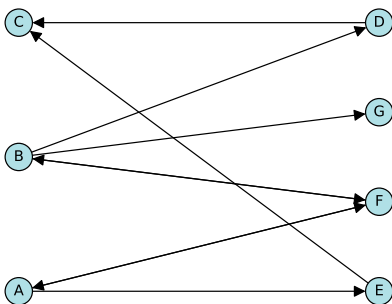
Le **graphe cyclique** à n sommets est le graphe dont l'ensemble de sommets est $\{0, 1, \dots, n - 1\}$ et tel que le sommet i est relié au sommet $i + 1$ (incluant le sommet $n - 1$ relié au sommet 0). Il se représente naturellement en cercle.



Écrire une fonction `graphe_cyclique(n)` qui renvoie la matrice d'adjacence du graphe cyclique sur n sommets.

Exercice 19.4

Un graphe $G = (S, A)$ est dit **biparti** si on peut diviser l'ensemble des sommets en deux, $S = S_1 \cup S_2$ avec $S_1 \cap S_2 = \emptyset$, tel que les arêtes ne peuvent exister qu'entre un élément de S_1 et un élément de S_2 , dans un sens ou dans l'autre. Mathématiquement $A \subset (S_1 \times S_2) \cup (S_2 \times S_1)$. Cela représente par exemple un ensemble de personnes dans une soirée divisé entre hommes et femmes, et où une arête de x vers y signifie « x a invité y à danser » ; il se représente naturellement avec les ensembles de sommets S_1 et S_2 face à face.



En particulier, le **graphe biparti complet** sur (n, p) sommets est le graphe dont l'ensemble des sommets est $S = \{0, 1, \dots, n + p - 1\}$ divisé entre $S_1 = \{0, \dots, n - 1\}$ et $S_2 = \{n, \dots, n + p - 1\}$, et qui admet toutes les arêtes possibles entre les sommets de S_1 et de S_2 .

Écrire une fonction `graphe_biparti_complet(n, p)` qui renvoie la matrice d'adjacence du graphe biparti complet sur (n, p) sommets.

II.3 Degré d'un sommet**Définition**

Soit G un graphe orienté et soit x un sommet de G .

1. Le **degré sortant** de x est le nombre d'arêtes partant de x .
2. Le **degré entrant** de x est le nombre d'arêtes arrivant à x .
3. Le **degré** du sommet x est la somme du degré sortant et du degré entrant.
4. Le **degré total** d'un graphe est le maximal des degrés de tous ses sommets.

Dans un graphe représentant un réseau social, où une arête de x vers y signifie que x est un *follower* de y , le degré sortant de x est le nombre de personnes que suit x , et le degré entrant est le nombre de followers de x .

Exercice 19.5

Écrire les fonctions, prenant en argument un graphe orienté G représenté par une matrice d'adjacence M et un sommet x (un nombre entier) :

1. `degré_sortant(M, x)`,
2. `degré_entrant(M, x)`,
3. `degré(M, x)`,
4. `degré_total(M)`.

III Chemins et connexité

Un thème important est d'étudier les chemins dans un graphe, en passant d'un sommet au suivant via une arête.

Définition

Soit $G = (S, A)$ un graphe.

1. Soient $x, y \in S$ deux sommets. Un **chemin** dans G de x à y est la donnée d'une suite de sommets (s_0, \dots, s_n) tels que :
 - $s_0 = x$,
 - $s_n = y$,
 - $\forall 0 \leq i \leq n-1, (s_i, s_{i+1}) \in A$.
2. Le nombre n ci-dessus est appelé la **longueur** du chemin.
3. La **distance** de x à y est la longueur minimale parmi tous les chemins possibles de x à y .
4. Un **cycle** est un chemin partant d'un sommet et arrivant à lui-même : $s_0 = s_n$.

Remarquons que par convention, il existe toujours un chemin de longueur 0 d'un sommet x à lui-même. Les arêtes de G sont exactement les chemins de longueur 1.

Un chemin dans un graphe, qu'il soit représenté en numérotant les sommets ou bien en leur donnant un nom (dictionnaire d'adjacence), est représenté en Python comme une simple liste C des sommets par lesquels le chemin passe, dans l'ordre. Alors $\text{len}(C)-1$ est la longueur du chemin.

Exercice 19.6

1. On représente un chemin par une liste C des sommets à parcourir, dans l'ordre, dans un graphe représenté par une matrice d'adjacence M . Écrire une fonction `est_chemin_possible(C, M)` qui renvoie `True` si ce chemin est bien possible (c'est à dire s'il existe bien une arête de $C[i]$ à $C[i+1]$, pour tout i) et `False` sinon.
2. Pouvez-vous écrire la même fonction mais en prenant cette fois-ci comme argument un dictionnaire d'adjacence ?

Exercice 19.7 *Mathématiques*

Soit G un graphe orienté et soit M sa matrice d'adjacence. Démontrer par récurrence que pour tout $p \in \mathbb{N}$, le coefficient (i, j) de M^p est le nombre de chemins de longueur exactement p reliant le sommet i au sommet j . On se concentrera sur le cas $p = 2$.

La notion suivante a du sens seulement si G est non-orienté.

Définition

Soit G un graphe non-orienté.

1. G est dit **connexe** si, entre tous sommets x et y , il existe au moins un chemin.
2. Les **composantes connexes** sont les sous-graphes de G maximaux (formés d'un plus grand nombre possible de sommets parmi ceux de G) qui sont connexes.

Un graphe non-connexe ressemble à ses composantes connexes posées simplement les unes à côté des autres.

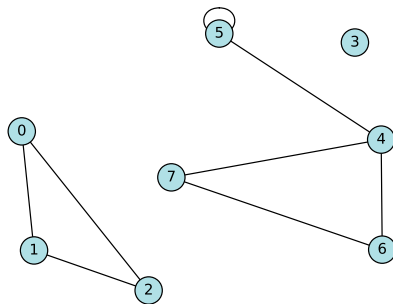


Figure 19.10. – Un graphe avec trois composantes connexes

Remarque. Si le graphe G a N sommets, alors pour tous sommets x, y , si il existe un chemin de x vers y , alors il en existe un de longueur inférieure ou égale à $N + 1$ (*pourquoi ?*). Cela donne au moins en théorie un algorithme pour tester la connexité de G sans tester une infinité de chemins possibles : on écrit la matrice d'adjacence M , on calcule M^{N+1} , et cette matrice ne doit contenir que des coefficients strictement positifs.

Remarque. La matrice d'adjacence d'un graphe non-orienté est une matrice *symétrique*... En général, pour un graphe orienté, qu'est-ce que la transposée de la matrice d'adjacence ?

IV Parcours aléatoire

Exercice 19.8

Le chat passe son temps entre quatre activités : dormir sur le canapé (C), manger (M), sortir (S), et... dormir sur le lit (L). Son parcours d'une journée typique est le suivant :

- S'il dort sur le canapé, alors ensuite il peut aller manger ou sortir ou bien passer sur le lit.
- S'il mange, alors ensuite il peut sortir ou bien dormir sur le lit.
- S'il sort, alors ensuite il peut dormir sur le canapé ou le lit ou bien manger.
- S'il dort sur le lit, alors il peut rester sur le lit ou bien passer sur le canapé.

À chaque heure, il passe aléatoirement d'une activité à l'autre. On représente cette situation par un graphe orienté à quatre sommets, représentant les quatre activités possibles du chat, et des flèches indiquant qu'il peut passer d'une activité à l'autre.

1. Dessiner sur feuille le graphe correspondant.
2. Écrire en Python le dictionnaire d'adjacence `chat` correspondant.
3. Écrire une fonction `suisvant(G, x)` prenant en argument un graphe G (quelconque) représenté par un dictionnaire d'adjacence, et un sommet x , et qui choisit au hasard un des sommets auxquels on peut accéder depuis x . On pourra pour cela utiliser la fonction `randint` du module `random` pour choisir un élément au hasard dans une liste.
4. En déduire une fonction `parcours(G, e, n)` qui part d'un sommet initial e dans un graphe G , et n fois de suite, choisit un sommet suivant au hasard.
5. Appliquer la fonction sur le dictionnaire `chat`, avec n assez grand, en affichant à chaque étape quelle est l'activité du chat.
6. Améliorer la fonction `parcours` en une fonction `parcours_compte(G, e, n)` qui renvoie un dictionnaire, constitué des même clés que G , qui indique combien de fois le parcours est passé sur chaque sommet ; et l'appliquer au graphe `chat`.



$$\Omega = \{C, M, S, L\}$$

from random import randint

$$\mathbb{P}(X) = \sum_{i=1}^n \mathbb{P}(A_i) \mathbb{P}_{A_i}(X)$$

TP 20

Parcours de graphes

Dans la continuité du TP précédent, nous nous intéressons aux questions de parcours de graphes. **Parcourir** un graphe, c'est écrire une fonction qui passe successivement d'un sommet à l'autre, dans un certain ordre, éventuellement pour y effectuer certaines opérations. On peut penser à une ville qu'on veut visiter entièrement, en passant d'un point à l'autre via les rues ; ou bien à un labyrinthe dont il faut sortir. Cela permet d'espérer répondre aux questions suivantes :

- Partant d'un sommet x , peut-on toujours atteindre un autre sommet y ? Par exemple, peut-on trouver un chemin pour sortir d'un labyrinthe ?
- Peut-on déterminer un chemin le plus court possible entre deux sommets ?

Cependant il y a plusieurs stratégies possibles, qui sont plusieurs ordres de parcours. Dans chacune de ces stratégies il y a une notion de sommets « déjà vus » (ceux sur lesquels nous sommes déjà passés, et nous n'avons pas particulièrement besoin d'y revenir) et de sommets « à voir » (ceux qui sont reliés aux sommets vus mais sur lesquels nous ne sommes pas encore passés).

On pourra utiliser pour les deux premières parties du TP les graphes orientés G_1 et G_2 ci-dessous, donnés comme des dictionnaires d'adjacence :

```
G1 = {"A": ["B", "D"], "B": ["C"], "C": ["E", "F"], "D": ["E"], "E": ["B"], "F": [], "G": ["C"]}
G2 = {"A": ["B", "C", "D"], "B": ["E"], "C": ["H"], "D": ["K"], "E": ["F", "G", "H"], "F": [], "G": ["F"], "H": ["C", "G", "K"], "K": []}
```

Exercice 20.1

Dessiner ces graphes sur feuille et les garder sous les yeux pour les deux premières parties du TP.

I Parcours en profondeur

De façon imagée, le parcours en profondeur correspond à explorer une ville, en partant de son point de départ (son hôtel), où, dès qu'on trouve une rue nouvelle qu'on n'a pas encore vue avant, on y va ! On continue tant qu'il y a encore à explorer. On ne revient en arrière que si on se trouve à un point où, tout autour, on a déjà tout vu. On risque donc de se retrouver rapidement à l'autre bout de la ville ; mais peu à peu on va revenir en arrière et finir par tout explorer.

Cela s'écrit simplement en Python avec une fonction récursive. Partant d'un sommet initial x , on regarde les sommets vers lesquels on peut accéder (par définition même du dictionnaire d'adjacence, c'est la liste $G[x]$), et on saute (en appelant la fonction récursivement) sur le premier élément de la liste... parmi ceux qui n'ont pas déjà été vus ! La fonction s'appelle donc `parcours_profondeur(G, x)`, prenant comme argument un graphe G donné comme dictionnaire d'adjacence, et un sommet x de G , et elle manipule une liste `vus` de sommets déjà vus qui est définie *en dehors de la fonction* — c'est cela qui est un peu subtil à cause de la récursivité. Au départ cette liste est vide ; à la fin de l'exécution, elle contient la liste des sommets sur lesquels la fonction est passée, et dans l'ordre.

On rappelle à ce propos les manipulations suivantes de listes :

- `vus.append(x)` : ajoute l'élément x à la fin de la liste `vus`,
- `if x in vus` : teste si l'élément x est dans la liste `vus`,
- `if x not in vus` : teste si l'élément x n'est pas dans la liste `vus`.

L'algorithme se décrit ainsi :

- Partant d'un sommet x , on le marque comme vu.
- Puis on regarde la liste des sommets auxquels on peut accéder depuis x .
 - Si on en trouve un, **et** qu'il n'a pas déjà été vu, on y va avec un appel récursif.
 - Sinon, la fonction s'arrête.

Sur notre graphe d'exemple G_1 , on a la séquence suivante, partant du sommet A :

Commentaire	Sommet actuel	Liste des sommets vus
Départ sur A	A	$[A]$
Passe à B	B	$[A, B]$
Passe à C	C	$[A, B, C]$
Passe à E	E	$[A, B, C, E]$
Retour à C	C	$[A, B, C, E]$
Passe à F	F	$[A, B, C, E, F]$
Retour à C	C	$[A, B, C, E, F]$
Retour à B	B	$[A, B, C, E, F]$
Retour à A	A	$[A, B, C, E, F]$
Passe à D	D	$[A, B, C, E, F, D]$
Retour à A	A	$[A, B, C, E, F, D]$
Fin	A	$[A, B, C, E, F, D]$

Exercice 20.2

Appliquer à la main l'algorithme sur le graphe d'exemple G_2 .

Exercice 20.3

Écrire la fonction `parcours_profondeur(G, x)`.

(*Remarque.* En fait, pour manipuler la liste `vus`, il est plus pratique d'écrire une *sous-fonction récursive* (la fonction `parcours_profondeur` contient la liste `vus`, et à l'intérieur, elle définit une fonction `aux` qui est récursive).

Par construction, un sommet y est accessible depuis x s'il apparaît dans la liste `vus` en effectuant le parcours depuis x . On peut aussi vouloir arrêter la fonction plus tôt, dès qu'on tombe y .

Exercice 20.4

En déduire une fonction `existe_chemin(G, x, y)`, qui renvoie `True` s'il existe un chemin, suivant le sens des arêtes, du sommet x vers le sommet y , et `False` sinon.

II Parcours en largeur

Dans le parcours en largeur, on imagine qu'on visite une ville mais en restant à chaque fois le plus proche possible de son point de départ, tant qu'il reste des rues à explorer. On ne s'éloigne que quand on a déjà vu toutes les rues les plus proches, autrement dit les zones visitées forment des cercles concentriques de plus en plus larges autour de l'hôtel de départ. Ainsi, chaque fois qu'en prenant une rue on découvre un nouvel endroit, on ne va pas, comme dans le parcours en profondeur, y sauter tout de suite ; mais on va le mettre en « liste d'attente », et on y reviendra seulement quand on aura terminé d'explorer tout ce qui est déjà en attente.

La fonction ne s'écrit pas de façon récursive. Elle manipule une liste L et un indice i dans L tel que les sommets d'indice avant i ont déjà été vus, et ceux d'indice après i sont ceux à voir en liste d'attente. L'algorithme se décrit ainsi :

- Partant d'un sommet x , au départ $L = [x]$ tout seul et $i = 0$.
- On lit la liste des sommets accessibles depuis x , et on les ajoute à L . Ce sont les sommets « à voir » qui sont en liste d'attente.
- Si on a bien ajouté des sommets à l'étape précédente (on est à $i = 0$ et L est de longueur au moins 2) :
 - Alors on passe à $i = 1$, on considère qu'on est sur le sommet $L[1]$.
 - On recommence, en ajoutant à L tous les sommets accessibles depuis $L[1]$, ce sont les « nouveaux sommets découverts » qu'on met en file d'attente.
- Si i arrive au bout de la liste, et qu'il n'y a rien après, on s'arrête : on n'a plus rien en liste d'attente.

Sur notre graphe d'exemple G_1 , on a la séquence suivante en partant de A :

Commentaire	Indice i	Liste L
Départ sur A	0	$[A]$
Découvre B et D	0	$[A, B, D]$
Passe à B	1	$[A, B, D]$
Découvre C	1	$[A, B, D, C]$
Passe à D	2	$[A, B, D, C]$
Découvre E	2	$[A, B, D, C, E]$
Passe à C	3	$[A, B, D, C, E]$
Découvre F	3	$[A, B, D, C, E, F]$
Passe à E	4	$[A, B, D, C, E, F]$
Passe à F	5	$[A, B, D, C, E, F]$
Fin	5	$[A, B, D, C, E, F]$

Exercice 20.5

Appliquer l'algorithme à la main sur le graphe d'exemple G_2 .

Exercice 20.6

Écrire la fonction `parcours_largeur(G, x)`, qui prend en argument un graphe G représenté par un dictionnaire d'adjacence, et un sommet de départ x , et renvoie la liste des sommets vus ; et tester avec plusieurs sommets de départs.

Un avantage du parcours en largeur que n'a pas le parcours en profondeur, c'est de pouvoir calculer facilement la distance d'un sommet au point de départ. En effet, les sommets sont automatiquement vus dans l'ordre en fonction de leur distance au départ : seul le sommet x est à distance 0 de lui-même ; ensuite, seuls les nouveaux sommets découverts depuis x sont à distance exactement 1 de x , puis ceux découverts depuis ceux-ci sont à distance 2, etc. On peut alors calculer, en parallèle de la liste L , une liste D des distances des sommets à x (pour tout indice i , $D[i]$ est la distance du sommet $L[i]$ à x) : quand on découvre depuis $L[i]$ des sommets, on les ajoute à L , et on ajoute en même temps leur distance à D , qui est $D[i] + 1$.

Exercice 20.7

En déduire une fonction `distance(G, x, y)` qui renvoie la distance du sommet x au sommet y , et `None` s'il n'y a pas de chemin de x à y .

Une variante consiste à écrire une fonction `distances(G, x)` qui calcule toutes les distances de tous les sommets à partir de x , et renvoie le couple formé de L et de D ci-dessus ; la seule différence est que `distance(G, x, y)` s'arrête dès qu'elle passe sur y .

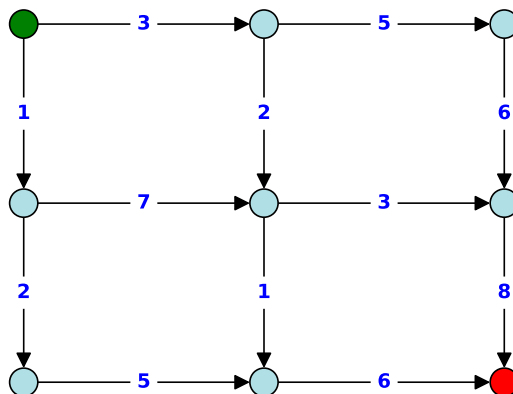
III Plus court chemin : un modèle simplifié

Le très célèbre **algorithme de Dijkstra** permet de déterminer la plus courte distance entre deux sommets dans un graphe orienté pondéré, où chaque arête porte un nombre indiquant sa longueur. On peut y penser très concrètement comme une carte d'une ville, indiquant des rues et leur temps de parcours, et il faut aller d'un point à l'autre le plus rapidement possible.

Le problème est en général compliqué, dû au fait que pour choisir globalement le chemin le plus court, on ne peut pas forcément choisir à *chaque étape* l'arête la plus courte qui nous fait avancer vers notre but...

On se propose d'étudier une situation simplifiée dans laquelle notre graphe a la forme d'un quadrillage : on part d'en haut à gauche, pour aller en bas à droite, et à chaque arête on peut aller soit en bas soit à droite mais on ne peut jamais revenir en arrière (c'est bel et bien un graphe orienté, dont les sommets sont les croisements d'une grille quadrillée). Sur les arêtes sont indiquées des valeurs, qu'on suppose être des nombres réels strictement

positifs. On peut y penser comme à une ville américaine, bien quadrillée, mais avec des embouteillages, et les valeurs indiquent le temps de parcours de chaque rue. Voici un exemple de tel graphe :



Exercice 20.8

Pouvez-vous trouver le plus court chemin dans ce graphe, entre le sommet de départ (en vert) et celui d'arrivée (en rouge) ? Observez que le plus court chemin n'est pas obtenu en choisissant à chaque intersection l'arête la plus courte...

On représentera un tel graphe en Python par la donnée de *deux* tableaux : V est le tableau de toutes les arêtes verticales, et H est le tableau de toutes les arêtes horizontales. Ainsi du sommet (i, j) on peut aller au sommet $(i + 1, j)$ avec la distance $V[i][j]$, ou bien au sommet $(i, j + 1)$ avec la distance $H[i][j]$. Pour l'exemple précédent les tableaux sont :

```
V = [[1, 2, 6], [2, 1, 8]]
H = [[3, 5], [7, 3], [5, 6]]
```

Dans le cas général, notre quadrillage a n lignes et p colonnes, ainsi sur une verticale on a n arêtes et $n + 1$ sommets, et sur une horizontale on a p arêtes et $p + 1$ sommets. Le sommet en haut à gauche est $(0, 0)$, celui en bas à droite est (n, p) . Le tableau V est de taille $(n, p + 1)$ et H est de taille $(n + 1, p)$.

L'idée est de calculer un tableau complet T de taille $(n + 1, p + 1)$ où $T[i][j]$ indique la distance la plus courte pour aller de tout en haut à gauche jusqu'au sommet (i, j) . Ce tableau contient *plus* d'information que notre seul but (qui est le coefficient $T[n][p]$) mais se calcule naturellement pas à pas. Pour notre exemple, on peut vérifier que le tableau est

$$T = \begin{bmatrix} 0 & 3 & 8 \\ 1 & 5 & 8 \\ 3 & 6 & 12 \end{bmatrix}$$

et donc la distance la plus courte jusqu'en bas à droite est 12.

Exercice 20.9

- Justifier que le tableau $T[i][j]$ vérifie les relations suivantes, permettant de le calculer entièrement pas à pas :
 - (i) $T[0][0] = 0$,
 - (ii) $\forall 1 \leq i \leq n, T[i][0] = T[i-1][0] + V[i-1][0]$,
 - (iii) $\forall 1 \leq j \leq p, T[0][j] = T[0][j-1] + H[0][j-1]$,
 - (iv) $\forall 1 \leq i \leq n, \forall 1 \leq j \leq p, T[i][j] = \text{Min}(T[i-1][j] + V[i-1][j], T[i][j-1] + H[i][j-1])$.
- En déduire une fonction `distances_minimales(V, H)` qui prend en argument les deux tableaux V et H décrivant entièrement notre graphe, et qui renvoie le tableau de toutes les distances minimales.

À retenir

Pour calculer *le* plus court chemin, l'algorithme consiste à calculer les longueurs de *tous* les plus courts chemins en même temps. On calcule plus de choses que demandé pour le résultat final, mais la méthode est plus efficace.

La méthode précédente donne la plus courte distance, mais ne montre pas par *quel* chemin on y accède. Une façon naturelle de s'y prendre — quoique pas la plus économe — consiste à calculer, en parallèle du tableau T , un tableau C de chemins (un chemin est représenté comme une liste de couples (i, j) , indiquant les sommets par lesquels il passe, donc T est une... liste de liste de listes de couples) où $C[i][j]$ est un chemin le plus court pour aller d'en haut à gauche jusqu'à (i, j) . Au départ C est un tableau de valeurs **None** et $C[0][0]$ est la liste $[(0, 0)]$, puis C se remplit au fur et à mesure qu'on remplit T en ajoutant (avec l'opération $+$ sur les listes, par exemple $C[i][j] = C[\dots][\dots] + [(i, j)]$) le sommet par lequel on passe.

Exercice 20.10

Écrire la fonction `plus_court_chemin(V, H)` qui renvoie le chemin le plus court, d'en haut à gauche jusqu'à en bas à droite.

Une autre façon naturelle pour trouver le plus court chemin est d'utiliser le tableau T et de remonter en partant du coefficient en bas à droite, en déduisant *d'où* provenait le chemin le plus court. On obtiendra alors le chemin rangé en ordre inverse. Plus précisément, on initialise des variables i et j représentant le coefficient sur lequel on se trouve, et un chemin C , en démarrant du bas à droite :

- Si on est remonté jusqu'en haut à gauche : c'est fini.
- Si on se trouve sur la première colonne : le chemin le plus court provenait nécessairement du haut.
- Si on se trouve sur la première ligne : le chemin le plus court provenait nécessairement de la gauche.
- Sinon :
 - Si $T[i][j-1] + H[i][j-1] < T[i-1][j] + V[i-1][j]$: le chemin le plus court provenait de la gauche.
 - Sinon : il provenait d'en haut.
- À chaque étape on utilise $C.append((i, j))$ pour ajouter au chemin la case par laquelle on passe.

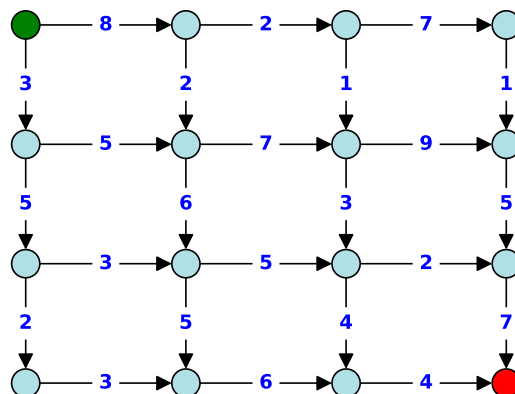
Exercice 20.11

Écrire la fonction `plus_court_chemin_inverse(V, H)` qui renvoie le plus court chemin, en ordre inverse, obtenu en partant du coefficient en bas à droite.

Vous avez tout compris ?

Exercice 20.12

Calculer le tableau des distances minimales, puis trouver le plus court chemin, pour l'exemple suivant :



TP 21

Aléatoire

Le module Python `random` contient plusieurs fonctions permettant de produire de l'aléatoire. Nous allons l'étudier, en lien avec le chapitre mathématiques de probabilités. Dans tout ce TP nous importons le module `random` avec la commande, à placer bien sûr une fois pour toute au début

```
import random as rd
```

Nous avons fait le **choix** ici de l'appeler `rd`, mais on pourrait aussi importer uniquement les fonctions que l'on souhaite. Les fonctions sont donc à précéder de `rd`, par exemple `rd.randint(a, b)`. Un sujet écrit doit préciser comment les fonctions ont déjà été importées, ou alors c'est à vous de le faire.

Avertissement

Nos fonctions étant par nature aléatoires, il est important pour faire des tests de les relancer plusieurs fois. La plupart contiennent déjà des boucles qui simulent une expérience répétée n fois, où le nombre n est passé comme argument à la fonction.

De plus, une bonne pratique est de **toujours récupérer le contenu d'une fonction aléatoire dans une variable**. Par exemple, on n'écrit **jamais**

```
if rd.randint(1, 6) == 6:  
    ...
```

mais plutôt

```
x = rd.randint(1, 6)  
if x == 6:  
    ...
```

En effet, dans la première syntaxe, après l'exécution la valeur aléatoire qui a été utilisée est perdue... Et si on l'utilise plusieurs fois on n'obtient pas le même résultat !

I Présentation du module `random`

Le module `random` possède les fonctions suivantes, auxquelles on accède dans ce TP avec le préfixe `rd` :

- `randint(a, b)` : donne un nombre **entier** uniformément au hasard entre les deux bornes `a` et `b`, de façon équiprobable. Contrairement à ce qui se passe avec `range`, les deux bornes sont bien incluses.
- `random()` : donne un nombre **réel** aléatoire choisi uniformément dans l'intervalle $[0, 1]$. La documentation précise que l'intervalle est en fait $[0, 1[$, mais la probabilité d'obtenir exactement 1 est nulle donc cela fait peu de différence.

Ce sont les seules à connaître en première année et nous pouvons déjà faire de nombreuses choses avec ; les lois de probabilités dites *continues* (choix d'un nombre réel au hasard, et non pas d'un nombre entier parmi un nombre fini de possibilités) seront étudiées nettement plus en détail en deuxième année. Mentionnons tout de même les fonctions suivantes qui sont faciles à utiliser :

- `uniform(a, b)` : donne un nombre réel au hasard entre les deux bornes `a` et `b`. On parle de **loi uniforme** car chaque partie du segment a la même probabilité d'être choisie. Mathématiquement cela signifie que la probabilité d'avoir un nombre qui tombe dans un morceau $[x, y]$ de l'intervalle $[a, b]$ est proportionnelle à la longueur de cet intervalle ; c'est donc $\frac{y-x}{b-a}$, car $[a, b]$ a probabilité 1.
- `randrange(a, b)` : similaire à `randint` mais où la borne `b` est **exclue**, comme avec `range`. On a aussi `randrange(n)` qui est équivalent à `randrange(0, n)`, c'est-à-dire `randint(0, n-1)`. Contrairement aux apparences cela est parfois *moins* casse-tête de l'utiliser : pour travailler avec des indices au hasard dans une

liste L de longueur n alors on peut utiliser $i = \text{randrange}(n)$ puis accéder à $L[i]$, ce qui est bien compatible avec la syntaxe `for i in range(n)`, sans avoir à jongler entre des n et des $n-1$.

- `choice(L)` : permet de choisir uniformément au hasard un élément dans une séquence L (liste, tuple, ...)

```
>>> rd.choice(["oui", "non", "peut-être"])
```

- `shuffle(L)` : mélange une liste L au hasard.
- `gauss(mu, sigma)` : donne un nombre réel au hasard selon la loi normale (gaussienne) d'espérance μ et d'écart-type σ . Brièvement, la valeur de μ correspond à la moyenne des nombres tirés, et la valeur de σ correspond à la largeur typique : plus σ est grand plus les valeurs au hasard vont être étalées autour de μ .

Toutes les fonctions du module sont listées dans l'aide `help(rd)`, et chacune possède aussi sa propre aide.

II Pour démarrer

On s'intéresse d'abord aux cas les plus basiques possibles, et on raisonne en améliorant sa fonction par étapes successives.

Exercice 21.1 Échauffement

Pour s'échauffer, on simule un lancement de pile ou face. Pour cela on tire un nombre entier au hasard par exemple entre 1 et 2 et on considère que 1 est pile et que 2 est face.

1. Écrire une fonction `pileface()` qui tire ainsi un nombre au hasard puis affiche "pile" ou "face" selon le résultat du tirage.
2. Améliorer la fonction précédente en une fonction `pilefaces(n)` qui effectue n tirages successifs, en affichant le résultat de chaque tirage.
3. Écrire une fonction `compte_pilefaces(n)` qui effectue n lancers de pile ou face et compte le nombre de piles et de faces obtenus, dans des variables p (nombre de piles) et f (nombre de faces). La fonction renvoie la liste de deux éléments $[p, f]$.
4. Enfin, reprendre la même fonction mais qui à la fin renvoie non pas le nombre mais la fréquence (éventuellement en pourcentage) de piles et de faces. On l'appellera `simule_pilefaces(n)`.

La méthode précédente constitue la base de la simulation d'une expérience aléatoire, répétée plusieurs fois, pour laquelle on s'intéresse à la fréquence d'apparition d'un évènement. Plus le nombre de répétitions est grand, plus la fréquence se rapproche de la probabilité. Pour tous les programmes suivants il est intéressants de travailler en améliorant sa fonction par étapes successives, pour faire d'abord **un** tirage aléatoire **puis** l'imbriquer dans une boucle à répéter n fois **puis** calculer les nombres de résultats obtenus et enfin les fréquences. Cela permet aussi de tester son programme au fur et à mesure...

Exercice 21.2

On s'intéresse maintenant au dé équilibré à 6 faces. Écrire une fonction `simule_dé(n)` qui tire des dés n fois et renvoie la liste de longueur 6 donnant la fréquence (ou bien, pour commencer, le nombre) d'apparition de chacune des faces. Pour compter on n'utilisera pas 6 variables, mais directement une liste de longueur 6 initialement remplie de zéros.

Attention car les indices de la liste sont numérotés à partir de 0 alors que les faces d'un dé sont numérotées à partir de 1...

III Quand est-ce qu'on biaise

On souhaite maintenant étudier des pièces biaisées avec une probabilité de 0,7 d'obtenir pile. Pour cela la méthode est de choisir un nombre **réel** x au hasard entre 0 et 1, puis de considérer qu'il s'agit d'un pile si $x < 0,7$ et d'une face sinon. Cela correspond bien à l'idée que la probabilité est 0,7 de choisir un nombre dans $[0; 0,7]$ et de 0,3

de le choisir dans $[0,7; 1]$ (les inégalités strictes ou larges n'ont pas d'importance ici). L'image à avoir en tête est qu'on découpe un intervalle en des morceaux de longueur proportionnelle à la probabilité à simuler.



Exercice 21.3

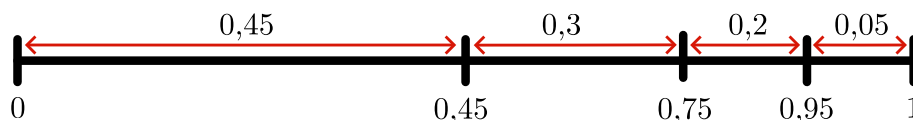
Écrire une fonction `simule_pièce_biaisée(n)` qui simule n fois un lancer d'une telle pièce biaisée, et renvoie la liste formée de la fréquence des piles et faces obtenues.

Cette méthode se généralise à plusieurs choix possibles, chacun avec sa probabilité ; il faut interpréter cela en terme de choix d'une partie du segment $[0, 1]$.

Exercice 21.4

Un client anonyme se présente à la boulangerie. Selon son humeur, il choisit au hasard entre le pain au chocolat avec probabilité 0,45, le croissant avec probabilité 0,3, le pain suisse avec probabilité 0,2 et une simple baguette avec probabilité 0,05.

Pour simuler cette situation, on tire un nombre réel au hasard dans $[0, 1]$. On considère qu'un nombre dans $[0; 0,45[$ correspond au pain au chocolat, un nombre dans $[0,45; 0,75[$ correspond au croissant, un nombre dans $[0,75; 0,95[$ au pain suisse et un nombre dans $[0,95; 1]$ à la baguette. Ce faisant, nous avons bien divisé l'intervalle $[0, 1]$ en morceaux dont la longueur correspond à la probabilité :



1. Écrire une fonction `choix_boulangerie()` qui choisit au hasard une viennoiserie avec cette règle et renvoie une chaîne de caractères "pain au chocolat", "croissant", "pain suisse", ou "baguette".
2. Écrire une fonction `simule_boulangerie(n)` qui répète l'expérience n fois, et renvoie un dictionnaire dont les quatre clés sont les quatre chaînes de caractères précédentes, indiquant le nombre de fois où chaque viennoiserie a été choisie.

Le même client se présente maintenant dans une boulangerie allemande. Il choisit entre le Schokocroissant avec probabilité 0,32 (hummm), le Laugendreieck avec probabilité 0,19 (un délice), le Mohnbrötchen avec probabilité 0,06 (bon mais très simple), le Scharferkumpel avec probabilité 0,13 (épicé, pas pour tous les jours), le Rosinenschnecke avec probabilité 0,21 (délicieux mais très gras et sucré), et enfin le Erdbeerplunder avec probabilité 0,09 (bon, mais très sucré et disponible selon la saison).

3. Reprendre les questions précédentes dans ce cas.

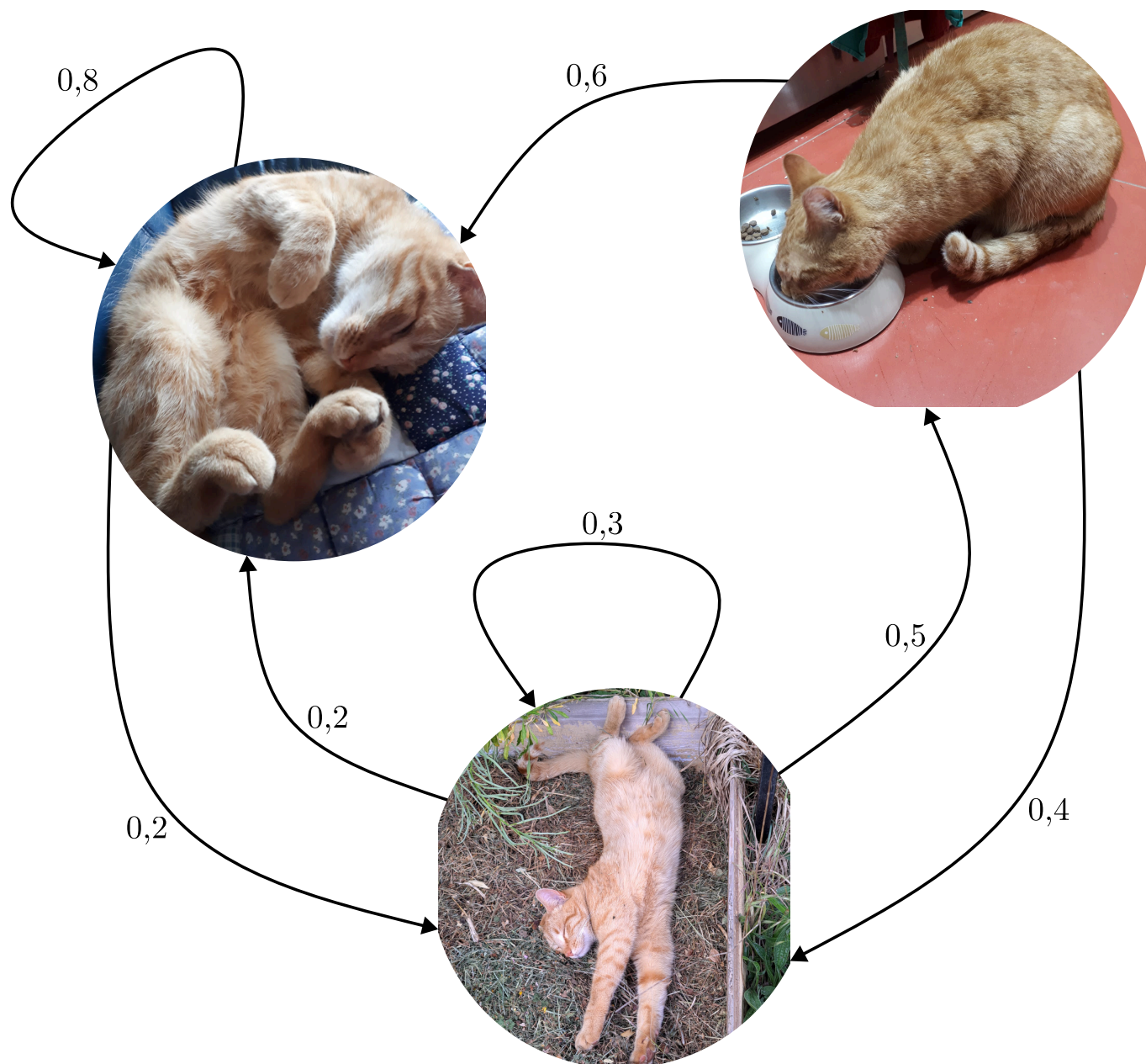


Figure 21.3. – Bäckerei Klingenstein à Essen

<https://www.baeckerei-klingenstein.de/Filialen/Kurfuerstenstr-5-mit-Cafe>

À retenir

Garder en tête l'interprétation géométrique : pour tirer au hasard entre plusieurs cas avec des probabilités données p_1, \dots, p_n , à partir d'un nombre uniformément au hasard entre 0 et 1 (les seules fonctions auxquelles nous avons vraiment accès sont `randint()` et `random()` !), on fabrique une division de l'intervalle $[0, 1]$ en morceaux de longueurs proportionnelles aux probabilités p_1, \dots, p_n . On est alors typiquement ramené à un test de « conditions en cascades » avec des `elif` successifs.

IV Diverses situations

Exercice 21.5

Le chat (le même que celui du TP sur les graphes !) passe sa vie entre trois activités : dormir, sortir dehors, et manger. À chaque heure, il passe d'un état à l'autre de façon aléatoire avec une certaine probabilité. Ces activités sont représentées comme les sommets d'un graphe orienté pondéré, et la probabilité de passer d'une activité à l'autre est indiquée sur l'arête. On note les activités simplement par les caractères "D", "S" et "M" (ou bien par leur nom complet).

1. Écrire une fonction `suisvant(x)` qui prend en argument l'activité actuelle du chat, et choisi au hasard l'activité suivante selon les règles décrites.
2. Écrire une fonction `simule_chat(n)` qui simule le changement d'activités du chat, n fois de suite, en démarrant par un chat qui dort.
3. Améliorer la fonction précédente pour qu'elle renvoie un dictionnaire de trois clés "D", "S", "M", indiquant le pourcentage du temps que le chat a passé dans chaque activité. Tester avec différentes activités de départ, et pour un nombre de répétitions n assez grand.
4. (Mathématiques) On note, pour $n \in \mathbb{N}$, d_n la probabilité qu'après n heures le chat soit en train de dormir, s_n la probabilité qu'il soit sorti et m_n la probabilité qu'il soit en train de manger. Justifier que ces trois suites vérifient la relation de récurrence qu'on peut écrire sous forme matricielle

$$\forall n \in \mathbb{N}, \underbrace{\begin{pmatrix} d_{n+1} \\ s_{n+1} \\ m_{n+1} \end{pmatrix}}_{X_{n+1}} = \underbrace{\begin{pmatrix} 0,8 & 0,2 & 0,6 \\ 0,2 & 0,3 & 0,4 \\ 0 & 0,5 & 0 \end{pmatrix}}_A \underbrace{\begin{pmatrix} d_n \\ s_n \\ m_n \end{pmatrix}}_{X_n}$$

puis que $\forall n \in \mathbb{N}$, $X_n = A^n X_0$; ici $X_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ si le chat est d'abord en train de dormir.

Remarque. La somme des probabilités sur les arêtes *sortantes* de chaque sommet du graphe est égale à 1, de même que la somme des nombres sur les *colonnes* de la matrice A , et on a $d_n + s_n + m_n = 1$ pour tout $n \in \mathbb{N}$. On a affaire à ce qui s'appelle une *chaîne de Markov* : une évolution d'un processus en fonction du temps qui passe à chaque moment aléatoirement d'un état à un autre. On peut toujours la représenter par un tel graphe orienté pondéré et l'étudier avec une matrice.

Exercice 21.6 *Méthode de Monte-Carlo*

On admet la proposition suivante : si on tire au hasard un point du plan à l'intérieur d'une zone rectangulaire, la probabilité de tomber dans une partie du plan est proportionnelle à son aire.

Nous allons l'appliquer en choisissant au hasard un point du carré $[0, 1] \times [0, 1]$, représenté par ses deux coordonnées, et en comptant combien de fois le point est à l'intérieur du cercle de centre 0 et de rayon 1, pour en déduire une valeur approchée du nombre π .

Écrire une fonction `simule_pi(n)` qui répète n fois l'expérience cette expérience et renvoie la fréquence, multipliée par 4, des points qui tombent à l'intérieur du cercle.

Exercice 21.7 *Urne de Pólya*

On considère l'expérience aléatoire suivante. Au départ, on dispose d'une urne (opaque) contenant une boule rouge et une boule bleue (indistinguables au toucher). Nous allons répéter n fois le procédé suivant : on tire au hasard une boule, on regarde sa couleur, on la remet dans l'urne, et on ajoute en plus dans l'urne une nouvelle boule de la même couleur. Ainsi si on tire rouge au premier tirage, le deuxième tirage se fera dans une urne à deux boules rouges et une boule bleue ; alors que si on tire bleu au premier tirage, le deuxième se fera dans une urne avec une boule rouge et deux boules bleues ; et on continue.

Pour simuler cette expérience en Python, on représente la composition de l'urne par deux variables a et b , où a représente le nombre de boules rouges et b le nombre de boules bleues. Après n tirages on a toujours $a + b = n + 2$; au départ $n = 0$, $a = b = 1$ et $a + b = 2$.

1. Écrire une fonction `simule_urne(n)` qui répète l'expérience n fois, et renvoie la liste `[a, b]` représentant la composition de l'urne après n tirages.
2. Écrire une fonction `simule_composition_urne(n, N)` qui répète toute l'expérience précédente N fois (de tirer n fois consécutivement dans une urne contenant initialement une seule boule de chaque couleur), et renvoie une liste `C` de longueur $n + 3$, où `C[i]` indique la fréquence en pourcentage des fois où, après n tirages, l'urne contient i boules rouges et $n + 2 - i$ boules bleues. Puis tester avec des valeurs de n modérées (moins de 10) et des grandes valeurs de N (plus de 1000). Qu'observez-vous ?

V Annexe : quelques notions sur la génération des nombres aléatoires

Un ordinateur n'est pas capable de produire du *vrai* hasard. Quand bien même ce vrai hasard existerait (question philosophique !), il n'est pas capable de lancer un dé ou de tirer des boules, mais uniquement de faire des calculs.

Pour produire des nombres aléatoires, les méthodes les plus simples sont basées sur certains types de suites récurrentes $u_{n+1} = f(u_n)$; elles peuvent prendre des grandes valeurs, mais on peut s'intéresser uniquement à leur réduction par exemple modulo 10 si on veut un nombre entre 0 et 9. La suite n'est donc pas du tout aléatoire, et est même nécessairement périodique (*pourquoi ?*), cependant :

- La fonction f est choisie pour que la suite *ait l'air* le plus aléatoire possible,
- La période de la suite doit être la plus grande possible, et on ne doit pas pouvoir facilement prédire le terme suivant en connaissant le ou les termes précédents,
- Le nombre u_0 appelé *graine* (en anglais *seed*) est fixé en dépendant de divers paramètres de l'ordinateur comme par exemple le temps (date, heure, minutes, secondes). Des graines qui varient *un peu* vont produire des suites *très* différentes.

On parle de **nombres pseudo-aléatoires**.

Un exemple très simple est donné par

$$u_{n+1} = 137u_n + 187 \pmod{256}$$

qui donne donc des nombres entre 0 et 255.

Si la graine n'est pas aléatoire, alors la sortie sera toujours la même, ce qui peut provoquer divers bugs ou failles de sécurité : testez et vous devriez obtenir exactement le même résultat dans le même ordre.

```
>>> rd.seed(12) # choix d'une graine : 12
>>> rd.randint(1, 10)
8
>>> rd.randint(1, 10)
5
>>> rd.randint(1, 10)
9
>>> rd.randint(1, 10)
6
>>> rd.randint(1, 10)
3
# essayez : tout le monde obtiendra le même résultat dans le même ordre
```

Ces problématiques sont bien entendu cruciales dans de nombreux domaines dans lesquels il faut produire de l'aléatoire de bonne qualité (simulations en sciences, méthode de Monte-Carlo, mais aussi sécurité et cryptographie) et ont donc été largement étudiées à coup de statistiques et de raffinements algorithmiques. Par exemple si le nombre aléatoire sert à fournir un code de validation de paiement pour une carte bleue, il est extrêmement important qu'on ne puisse pas prédire les nombres suivants, sinon un pirate pourrait valider des paiements à votre place ! C'est moins grave par exemple pour un jeu vidéo, où il s'agit de faire apparaître un ennemi à un endroit au hasard, alors la qualité du hasard à l'échelle de millions de données importe bien moins que le temps de calcul et on veut un algorithme simple et rapide.

Fort heureusement, Python utilise l'algorithme *Mersenne Twister 19937* qui est suffisamment élaboré, avec un aléatoire de bonne qualité selon de nombreux critères et une période de $2^{19937} - 1$ soit environ $4,3 \cdot 10^{6001}$: c'est beaucoup, beaucoup, beaucoup. Il est donc bien adapté à des situations en sciences. Malgré cela, il n'est pas considéré comme cryptographiquement sûr : en théorie, un pirate qui observe une séquence de 624 nombres aléatoires sortis consécutivement par la fonction `random()` pourrait prédire le suivant !

TP 22

Langage SQL partie 1

Le **langage SQL** permet de faire des recherches dans une base de données. Une vraie base de données peut contenir des millions d'informations rangées dans plusieurs tableaux et il n'est alors pas question de faire défiler les lignes à la main pour trouver l'information souhaitée ; de plus, il s'agit de pouvoir automatiser un certain nombre de tâches demandant à lire ou à enregistrer des données et de communiquer avec la base de données à travers d'autres programmes, par exemple avec un site web qui demande à la base d'enregistrer un nouvel utilisateur. Le mot SQL signifie *Structured Query Language* c'est-à-dire « langage de requêtes structurées ».

Pour l'illustrer nous allons utiliser une base de données contenant des informations sur tous les aéroports du monde, ainsi que les régions et les pays dans lesquels ils se trouvent. Attachez vos ceintures, nous allons décoller !

I Bases de données

Adoptons le vocabulaire approprié ; dans ce contexte on appelle **table** un tableau, dont les colonnes sont plutôt appelées **attributs** et les lignes sont des **entrées**. En générale, les colonnes sont fixées à l'avance (ce sont les attributs que l'on veut étudier), mais on veut pouvoir rajouter des lignes au fur et à mesure, sans nécessairement qu'il y ait d'ordre entre elles, chacune est donc une nouvelle entrée.

Nous en avons déjà parlé dans le TP sur les prénoms, où nous avons affaire à une table de tous les prénoms contenant quatre attributs (nom, sexe, année de naissance, nombre) et de très nombreuses entrées.

Une base de donnée intéressante peut être constituée de *plusieurs* tables faisant référence les unes aux autres de façon croisée, mais nous étudierons cette situation plus en détail dans le TP d'après. Dans celui-ci, la table des aéroports contient un code de la région où se trouve l'aéroport, et une autre table contient toutes les régions et des informations sur chacune ; on peut donc obtenir des informations simultanément sur l'aéroport et sa région en « croisant les tables ».

II Le langage SQL


Le logiciel que nous utilisons **DB Browser for SQLite** (*explorateur de base de données pour le langage SQLite*) permet d'ouvrir une base de données, de visualiser le contenu (onglet « Parcourir les données ») et de travailler dessus en exécutant des requêtes SQL (onglet « Exécuter le SQL »). Le résultat de chaque requête sera une table, qui apparaîtra en dessous.

Après avoir récupéré et décompressé le fichier `materiel.zip`, aller chercher depuis le logiciel avec « Ouvrir une base de données en lecture seule » (pour éviter de faire des bêtises en modifiant la base) le fichier `airports.db`.

Pour démarrer, testez votre première requête SQL :

```
SELECT name, iso_country FROM airports;
```

Elle signifie de sélectionner seulement les noms et les codes pays dans la table `airports`.

On peut écrire plusieurs requêtes SQL dans la zone de texte en les terminant par un point-virgule ; et on peut alors demander à n'exécuter que la requête où se trouve le curseur avec l'icône play-pause , aussi sur le raccourci clavier **⌘** + **F5**

Contrairement au langage Python, les indentations et sauts de ligne n'ont pas d'importance et permettent seulement de rendre le code plus lisible. Les commentaires sont contenus entre les deux symboles : `/* commentaire */`.

La base de données contient trois tables : `airports` avec tous les aéroports, `countries` avec tous les pays, et `regions` avec toutes les régions de tous les pays. Le code ISO des pays est un code standard à deux lettres qui identifie uniquement chaque pays ; le code IATA d'un aéroport est un code à trois lettres qui identifie uniquement les aéroports importants, c'est notamment le code qui est écrit sur les billets d'avion et les bagages. Certaines régions apparaissent comme des pays à part entière, comme les régions d'outre-mer françaises, ce qui a du sens du point de vue d'un pilote...

Ces données sont ouvertes. Source : <https://ourairports.com/data/>.

III Requêtes simples

Dans tout ce TP nous nous intéressons avant tout aux requêtes permettant d'aller chercher des informations **dans une seule table**. La syntaxe la plus simple, la plus courante, est :

```
SELECT attributs
FROM table
WHERE condition
```

où

- **attributs** désigne un attribut de la table (une colonne), ou plusieurs séparés par des virgules, que l'on veut sélectionner, ou le caractère ***** pour sélectionner tous les attributs d'un coup,
- **table** est le nom de la table dans laquelle on sélectionne,
- **condition** est une condition exprimée sur les attributs.

Les attributs ont un type de donnée : entier, flottant, chaîne de caractère (entourées préférablement par des apostrophes simples '**texte**'). Il existe éventuellement d'autres types, permettant par exemple de stocker facilement une date et de comparer rapidement les données selon l'ordre chronologique... Il est important aussi d'avoir une valeur spéciale **NULL** indiquant l'absence de donnée.

La condition peut être formulée avec :

- Des comparaisons mathématiques comme d'habitude : comparaisons strictes **>**, **<**, comparaisons larges **>=**, **<=**, égalité **=**, différence **<>** (attention pour ces deux dernières, ce n'est pas du Python !),
- Les opérations arithmétiques habituelles **+**, **-**, *****, **/**,
- On peut combiner les conditions avec les mots-clés **AND**, **OR**, **NOT**, et utiliser des parenthèses.

Testez par exemple :

```
SELECT * FROM airports WHERE iso_country='FR';
```

(tous les aéroports français) ou

```
SELECT name FROM airports WHERE type='large_airport' AND iso_region='FR-IDF';
```

(les gros aéroports d'Île-de-France, donner seulement les noms).

Exercice 22.1

Écrire des requêtes SQL pour :

1. Sélectionner le nom de tous les pays.
2. Sélectionner le nom de tous les pays d'Europe.
3. Sélectionner toutes les bases d'hydravion (**seaplane_base**) du monde.
4. Sélectionner toutes les bases d'hydravion en Europe.
5. Sélectionner le nom et les villes des aéroports en France qui sont gros ou moyens.
6. Sélectionner tous les aéroports qui sont en-dessous du niveau de la mer.

Quelques compléments :

- On peut toujours renommer un attribut avec le mot-clé **AS** :

```
SELECT attributs AS nouveau_nom
```

Cela fonctionne un peu comme donner un nom à une variable, et aussi comme titre (propre) à la colonne résultat. Le titre peut être encadré entre guillemets doubles.

```
SELECT name AS "Nom", municipality AS "Ville"  
FROM airports WHERE type='large_airport' AND iso_region='FR-IDF';
```

- Il se peut qu'après une requête, une entrée apparaisse plusieurs fois. Pour éliminer les doublons, on utilise le mot-clé **SELECT DISTINCT** à la place de **SELECT**. Testez :

```
SELECT DISTINCT continent FROM countries;
```

- Après **WHERE**, on a souvent envie de renvoyer le résultat classé selon un attribut, qui est soit un nombre (ordre usuel) soit une chaîne de caractères (ordre lexicographique = ordre du dictionnaire) avec le mot-clé **ORDER BY**, qu'on peut aussi faire suivre de **ASC** (*ascending*, croissant) ou bien **DESC** (*descending*, décroissant). Testez par exemple :

```
SELECT code, name FROM countries ORDER BY code ASC;
```

(noms et codes des pays, classés par code dans l'ordre alphabétique).

À retenir

Les requêtes les plus générales vues jusque là ont la forme suivante :

```
SELECT attributs  
FROM table  
WHERE condition  
ORDER BY critère ASC | DESC
```

Exercice 22.2

Écrire des requêtes SQL pour :

1. Afficher les aéroports français moyens ou gros, classés du nord au sud.
2. Afficher les villes des gros aéroports des États-Unis, classés d'ouest en est.
3. (i) Afficher les noms et villes des héliports français, situés à plus de 1000 m d'altitude, sachant que 1 ft = 0,3048 m (sans utiliser sa calculatrice avant),
(ii) ... classés par altitude,
(iii) ... pouvez-vous afficher directement l'altitude en mètres ?
4. Afficher les codes ISO des pays africains contenant un gros aéroport. Sans doublon.
5. (i) Afficher les régions allemandes et leur code ISO, classées par ordre alphabétique.
(ii) Afficher tous les villes de la région Rhénanie-du-Nord-Westphalie contenant un aéroport.

Remarque. Le saviez-vous ? La région Rhénanie-du-Nord-Westphalie est l'une des plus importantes d'Allemagne. La plus peuplée (18,2 million d'habitants), la plus importante économiquement, l'une des plus riches (avec la Bavière). Elle comprend de nombreuses villes très peuplées formant une énorme agglomération Rhin-Ruhr : Bonn, Cologne, Düsseldorf, Duisburg le long du Rhin, et Essen, Bochum, Dortmund, Wuppertal dans la vallée de la Ruhr. La capitale est Düsseldorf, où se trouve aussi un aéroport international, le troisième aéroport allemand après Francfort et Munich.

IV Les fonctions d'agrégation

On appelle **fonction d'agrégation** une fonction qui calcule un résultat à partir de **toute une colonne**. Ce sont notamment les fonctions suivantes :

- **MIN()** : minimum,
- **MAX()** : maximum,
- **SUM()** : somme,
- **COUNT()** : nombre d'entrées,
- **AVG()** : moyenne (en anglais *average*).

Ce qui peut être surprenant au premier abord, c'est qu'elles s'utilisent **dans SELECT**, souvent en leur donnant un nom avec **AS**. Par exemple, la commande suivante donne l'altitude moyenne de tous les aéroports présents :

```
SELECT AVG(elevation_ft) AS "Altitude moyenne" FROM airports;
```

Le résultat est un nombre seul — que l'on peut considérer comme une table !

Il est important d'autoriser à avoir dans une base des données manquantes, portant la valeur spéciale **NULL**. Alors la moyenne, par exemple, est calculée en ignorant les données manquantes, ce qui n'est pas la même chose que de les considérer comme étant zéro...

La fonction **COUNT()**, elle, compte le nombre de valeurs dans la colonne donnée qui ne sont pas **NULL**. Fréquemment, on veut compter le nombre d'entrées tout court, sans référence à une colonne particulière : c'est la syntaxe **COUNT(*)**. Par exemple si on veut connaître le nombre total de pays :

```
SELECT COUNT(*) AS "Nombre total" FROM countries;
```

Tout ceci se combine avec **WHERE** pour filtrer les données avant de compter.

Exercice 22.3

Écrire des requêtes SQL pour :

1. Donner l'élévation moyenne de tous les aéroports aux Pays-Bas (*Netherlands...*)
2. Même question pour la Suisse (*Confédération Helvétique...*) et pour le Népal.
3. Donner le nombre de gros aéroports en Europe.
4. Donner le nombre de régions allemandes.

Il se peut que l'on souhaite calculer les fonctions d'agrégation non pas sur *toute* la colonne, mais en regroupant entre elles les entrées selon les valeurs d'une autre entrée, par exemple compter le nombre de pays *par continent*. On utilise alors le mot-clé **GROUP BY**, qui se place après **WHERE** mais avant un éventuel **ORDER BY** :

```
SELECT COUNT(*) AS "Nombre", continent
FROM countries
GROUP BY continent;
```

Après un **GROUP BY**, on peut vouloir ne garder que les entrées satisfaisant une certaine condition qui est elle-même le résultat d'une fonction d'agrégation. On utilise alors la commande **HAVING** suivie d'une condition, qui se place juste après le **GROUP BY**. Ce n'est pas tout à fait la même chose que **WHERE** car **WHERE** agit **avant** la fonction d'agrégation, pour sélectionner seulement ce qui nous intéresse et calculer la fonction dessus, alors que **HAVING** agit **après le calcul**. L'exemple ci-dessous sélectionne les continents avec au moins 20 pays :

```
SELECT COUNT(*) AS "Nombre", continent
FROM countries
GROUP BY continent
HAVING "Nombre" >= 20;
```

Exercice 22.4

Écrire des requêtes SQL pour :

1. Donner le nombre de régions par pays d'Europe.
2. Donner l'élévation moyenne par pays des aéroports gros ou moyens d'Amérique du Sud.
3. Donner les villes de France ayant au moins trois aéroports.
4. Donner les villes d'Europe ayant plusieurs aéroports gros ou moyens.

À retenir

En général, la commande **SELECT** est suivie par une liste contenant des attributs ou des fonctions d'agrégations sur les attributs. Les attributs utilisés dans une fonction d'agrégation sont souvent renommés avec **AS** et peuvent être utilisés dans un **HAVING**, alors que les attributs seuls doivent être utilisés dans un **GROUP BY**.

On peut donc grossièrement résumer les requêtes les plus générales vues jusque là sous forme :

```
SELECT fonction(attribut1) AS résultat, attribut2
FROM table
WHERE condition /* condition sur la recherche, avant de faire un calcul */
GROUP BY attribut2
HAVING condition /* condition sur le résultat du calcul */
ORDER BY critère ASC | DESC
```

V Sélection dans plusieurs tables

Parfois, il faut recouper les informations qui sont contenues à travers plusieurs tables. C'est le concept de **jointure** que nous approfondirons au prochain TP.

Par exemple, chaque aéroport est attaché à une région, par un code appelé `iso_region` dans la table `airports`. Les régions se trouvent elles dans une autre table appelée `regions`, dont le code est simplement donné par l'attribut `code`. Pour obtenir des informations simultanément sur les aéroports et leur région, il faut utiliser une jointure. Observez le résultat de la commande suivante :

```
SELECT * FROM airports
JOIN regions ON airports.iso_region=regions.code
WHERE airports.iso_country='FR' AND airports.type='large_airport';
```

Elle se lit comme : sélectionner les aéroports en France, de type large, et y joindre les informations de la table des régions, quand coïncident les codes de région (celui de l'aéroport défini dans la table `airports`, avec celui des régions défini dans la table `regions`).

Comme on manipule les attributs de plusieurs tables en même temps, on les note `table.attribut` pour bien les distinguer, mais on peut aussi s'en passer en utilisant le renommage **AS**.

Le résultat de la commande précédente serait plus lisible si on sélectionnait seulement ce qui nous intéresse :

```
SELECT airports.name AS "Nom", airports.municipality AS "Ville", regions.name AS "Région"
FROM airports
JOIN regions ON airports.iso_region=regions.code
WHERE airports.iso_country='FR' AND airports.type='large_airport';
```

En général le mot-clé **JOIN** fonctionne ainsi :

```
SELECT attributs
FROM table
JOIN autre_table
ON critère
```

où en général `critère` est simplement l'égalité entre un attribut de la première table et un attribut de la seconde. Bien entendu, on peut rajouter à tout cela les `WHERE`, `ORDER BY`, les agrégations et autres.

Exercice 22.5

Écrire des requêtes SQL pour :

1. Donner la liste des gros aéroports européens et leur région.
2. Donner les gros aéroports d'Asie et leur pays, classés d'ouest en est.
3. Un voyageur a un billet pour SCL. Dans quelle ville et quel pays se rend-il ?
4. Un voyageur a un billet pour DPS. Dans quel pays et quelle région se rend-il ?
5. (i) Donner le classement des pays avec le plus de codes IATA.
(ii) ... en ne gardant que ceux avec plus de 100 codes.
6. Donner le classement des pays du monde dont l'altitude moyenne des aéroports est la plus élevée.

VI Annexe

Manipulation des bases de données Nous nous sommes intéressé au minimum vital pour effectuer une requête SQL. Le langage SQL permet aussi d'*insérer* des entrées dans une base de données (mot-clé `INSERT`), de créer des tables (`CREATE TABLE`), de gérer en entier une base de données à travers des programmes informatique.

En général le SQL ne s'utilise pas ainsi tout seul, car on ne va pas insérer les entrées de la table unes par unes avec des requêtes SQL... Il est utilisé conjointement avec d'autres langages ou d'autres programmes pour en récupérer les données, faire des calculs dessus ou les représenter graphiquement. Il y a par exemple des fonctions Python permettant d'ouvrir une base de données et de lancer des requêtes SQL dessus, puis récupérer les résultats dans des listes Python, et appliquer tout ce que l'on veut ensuite. Les sites internet utilisent aussi du SQL côté serveur, par exemple un utilisateur se connecte et il faut demander à la base d'aller chercher les informations de l'utilisateur. C'est donc un programme qui communique avec un autre *via* le langage SQL.

Optimisation des bases de données Les vraies bases de données peuvent contenir des millions d'entrées, ainsi chaque requête pourrait demander beaucoup de travail à l'ordinateur. Une grosse part du travail de conception de bases de données, et des logiciels qui vont avec, est d'optimiser les données pour pouvoir travailler le plus efficacement avec. Les données sont très loin d'être enregistrées les unes à la suite des autres comme on peut le visualiser dans un tableau (lire par exemple l'annexe du TP sur les dictionnaires et les tables de hachage).

De plus le but de jeu est toujours d'écrire le minimum possible de requêtes SQL et que chacune soit la plus *efficace* possible, c'est ainsi qu'on utilise au mieux les possibilités d'optimisation des bases de données. Par exemple les fonctions d'agrégations fournies par le langage SQL sont aussi bien plus efficaces, quand on les utilise correctement, que de récupérer les données avec un autre programme par exemple en Python *puis* de faire les calculs dessus.

Quelques ressources Pour s'entraîner tout seul :

- Des exercices interactifs dans diverses tables : <https://sql-exercices.github.io/>
- Cours à la fois bien complet et interactif (mais en anglais) : <https://www.w3schools.com/sql/>
- Exercices interactifs (en anglais) : <https://sqlzoo.net/>

TP 23

Langage SQL partie 2

Merci à Guillaume ROUX (partie III) et à Quentin FORTIER (partie IV) pour le matériel de ce TP.

I Structure des bases de données

Une base de données est en général constituée de **plusieurs tables**, faisant référence les unes aux autres. Pour extraire des informations de la base, il sera alors nécessaire de **croiser** les données à travers plusieurs tables.

I.1 Exemples

Retour sur le TP précédent Lors du TP précédent, notre base de données était constituée d'au moins deux tables, l'une contenant une liste d'aéroports, l'autre une liste de pays. Un extrait simplifié de la table ressemble à ceci :

aéroports			
IATA	nom	ville	ISO
CDG	Roissy-Charles-de-Gaulle	Paris	FR
ORY	Orly	Paris	FR
AMS	Schiphol	Amsterdam	NL
MRS	Marseille-Provence	Marseille	FR
DUS	Düsseldorf	Düsseldorf	DE
MUC	Franz-Josef-Strauß	Munich	DE

pays	
ISO	nom
FR	France
NL	Pays-Bas
DE	Allemagne

On peut constater que pour obtenir à la fois la donnée d'un aéroport et de son pays, il est nécessaire de croiser les deux tables, en les recoupant selon la valeur du code ISO du pays. Par exemple, on lit dans la première table que l'aéroport Roissy-Charles-de-Gaulle se trouve dans le pays FR, puis en lisant la seconde que ceci correspond bien à la France.

Un nouvel exemple Un réseau social a des utilisateurs, qui peuvent poster des messages (avec des photos) et les commenter. Pour organiser tout cela, il faut au moins trois tables :

- Une première table **utilisateurs** où chaque nouvel utilisateur est une entrée. Elle contient donc des informations de l'utilisateur, son nom, son mot de passe pour se connecter.
- Une autre table **publications** servira à conserver les photos publiées et leur légende. Chaque fois qu'un utilisateur met en ligne une photo, elle est stockée et sa légende est ajoutée à la table.
- Enfin, chaque commentaire sous une publication créera une entrée dans une table **commentaires**, qui devra indiquer à quelle publication le commentaire se réfère et par qui il a été posté.

On peut penser que la base ressemble à ceci :

utilisateurs		
nom	motdepasse	ville
lclefevre	arctangente	Versailles
élève1	azerty	Versailles
élève2	jaimelabio	Versailles
cedricgrolet	croissant	Paris

publications			
numéro	date	auteur	contenu
1298445	15/03/2026 18:45	lclefevre	Une photo de chat
1298446	17/03/2026 11:32	élève1	Soirée d'hier
1298447	18/03/2026 08:56	cedricgrolet	Un pain au chocolat

commentaires			
date	sur	par	commentaire
15/03/2026 20:12	1298445	élève1	trop mignon !!!
17/03/2026 12:02	1298446	élève2	c'était trop bien !!
18/03/2026 16:15	1298447	lclefevre	miam miam
18/03/2026 17:01	1298445	élève2	vraiment trop mignon
19/03/2026 15:17	1298447	élève2	*chocolatine
19/03/2026 22:35	1298445	élève2	Il s'appelle comment ?

Croiser les trois tables permet par exemple d'affirmer « le 18/03/2026, élève2 a commenté la photo de chat de lclefevre postée le 15/03/2026 ».

I.2 Clés primaires et secondaires

Pour bien étudier le phénomène de croisement des tables, nous avons besoin de ces notions.

Définition

Dans une table, une **clé primaire** est un attribut (une colonne) qui identifie uniquement chaque entrée.

Quelques exemples :

- Dans la liste des élèves de la classe, le prénom seul ne forme en général pas une clé primaire : il peut y avoir plusieurs élèves avec le même prénom et donc on ne peut pas les distinguer uniquement ainsi. Par contre la donnée entière prénom + nom de famille est bien une clé primaire.
- À l'échelle de tout le pays, la donnée du prénom et du nom de famille n'est pas une clé primaire : il arrive que plusieurs personnes aient à la fois le même prénom et le même nom de famille, et donc cela n'identifie pas uniquement les personnes. Pour remédier à cela, nous avons tous de nombreux numéros d'identification, qui agissent comme des clés primaires pour distinguer uniquement les personnes : numéro de sécurité sociale, numéro de carte d'identité ou de passeport, mais aussi des numéros de clients chez de très nombreux services (client téléphone, client électricité, numéro de déclarant aux impôts, etc).
- Dans la base de données des aéroports, le code IATA à trois lettres (CDG pour Roissy-Charles-de-Gaulle, ORY pour Orly) est une clé primaire pour les aéroports commerciaux, qui identifie uniquement chaque aéroport du monde. Il est utile d'avoir ce code indiqué sur le billet et sur le bagage. Le code ISO des pays est, lui, une clé primaire pour identifier rapidement tous les pays du monde, indépendamment de la langue.

En fait, il est toujours possible de créer artificiellement une clé primaire, simplement en définissant un **numéro d'identification** (communément appelé ID), qui augmente de 1 automatiquement à chaque entrée, et le logiciel qui gère la base de données sait très bien le faire.

Définition

Dans une table d'une base de données, une **clé étrangère** (ou : **secondaire**) est un attribut dont les valeurs font référence à une clé primaire dans une autre table.

Exemples :

- Dans la base de données des aéroports, la colonne `ISO` de la table `aéroports` est une clé étrangère : ses valeurs sont à trouver dans la table `pays` où ce sont des clés primaires.
- Dans l'exemple de réseau social ci-dessus, la table `publications` contient la clé étrangère `auteur`, qui fait référence à la liste des utilisateurs. Pour savoir de qui provient la publication, il est nécessaire de croiser le contenu de cette colonne avec la table `utilisateurs`. Dans ce même exemple, la table `commentaires` contient *deux* clés étrangères : `par` qui fait référence à l'auteur du commentaire, qu'on va trouver dans la table `utilisateurs`, et `sur` qui fait référence au numéro de la publication qui est commentée, et qu'on va trouver dans `publications`.

L'intérêt de savoir qu'une colonne est une clé étrangère, et de savoir à quelle autre table elle fait référence, c'est de garantir que toutes les données de la base sont bien cohérentes. Supposons qu'on rentre des aéroports à la main et qu'on commette une erreur, par exemple on indique un aéroport avec le pays FS au lieu de FR, qui n'existe pas. Alors on ne peut plus répondre à la question « dans quel pays se situe cet aéroport », cela provoque une erreur. Et rapidement les requêtes SQL plus compliquées n'ont plus de sens et bloquent (par exemple on ne peut pas du tout répondre à « combien d'aéroports par pays » car cet aéroport n'est attribué à aucun pays). On parle de **corruption** des données. Au contraire le logiciel de gestion de base de données, si on lui déclare où sont les clés primaires et étrangères, sait vérifier au fur et à mesure des ajouts et suppressions dans la base que les données restent bien cohérentes et donc peut **garantir l'intégrité des données**.

Ajoutons aussi que l'intérêt d'utiliser pour les clés primaires des nombres entiers ou des codes à deux ou trois lettres, c'est de pouvoir bien plus rapidement comparer et trier selon ces valeurs plutôt qu'en comparant des chaînes de caractères en entier (ce qui est toujours une opération lourde), c.f. l'annexe du TP sur les dictionnaires.

I.3 Jointure

Connaissant les clés primaires et étrangères, on peut enfin croiser intelligemment les tables.

Définition

L'opération de **jointure** entre deux tables consiste à combiner toutes les colonnes de l'une avec toutes les colonnes de l'autre, le long d'un attribut en commun.

Dans l'exemple des aéroports, la jointure des tables `aéroports` et `pays` selon l'attribut `ISO` donne la table suivante :

IATA	nom	ville	ISO	pays
CDG	Roissy-Charles-de-Gaulle	Paris	FR	France
ORY	Orly	Paris	FR	France
AMS	Schiphol	Amsterdam	NL	Pays-Bas
MRS	Marseille-Provence	Marseille	FR	France
DUS	Düsseldorf	Düsseldorf	DE	Allemagne
MUC	Franz-Josef-Strauß	Munich	DE	Allemagne

Dans le deuxième exemple, la jointure des tables `publications` et `commentaires` le long des numéros de publications (qui s'appelle `numéro` dans la table des publications et `sur` dans la table des commentaires) correspond à mettre côte à côte les publications et leurs commentaires ; si ensuite on ne garde que la publication numéro 1298445 on obtient bien

numéro	date	auteur	contenu	date	par	commentaire
1298445	15/03/2025 18:45	lclefevre	Une photo de chat	15/03/2025 20:12	élève1	trop mignon !!!
1298445	15/03/2025 18:45	lclefevre	Une photo de chat	18/03/2025 17:01	élève2	vraiment trop mignon
1298445	15/03/2025 18:45	lclefevre	Une photo de chat	19/03/2025 22:35	élève2	Il s'appelle comment ?

Remarquez qu'ici le mot `date` apparaît deux fois avec deux significations différentes ; en fait il faudra utiliser la syntaxe `publications.date` pour le premier et `commentaires.date` pour le second. On pourrait aussi joindre trois tables, pour obtenir à la fois des informations sur les publications, leur auteur, et leurs commentaires.

II Rappels de SQL

On rappelle brièvement la forme générale des requêtes SQL qu'on utilise, voir le TP précédent :

```
SELECT attributs
FROM table
WHERE condition
ORDER BY critère ASC | DESC
```

où

- `attributs` est un attribut de la table, ou plusieurs séparés par des virgules, ou le caractère `*` ; on peut renommer les attributs avec `AS`, éventuellement `SELECT DISTINCT` pour avoir une seule fois chaque entrée,
- `table` est le nom de la table dans laquelle on sélectionne,
- `condition` est une condition exprimée sur les attributs, utilisant les opérations mathématiques `+`, `-`, `*`, `/`, les comparaisons `>`, `<`, `>=`, `<=`, `=`, `<>`, les mots-clés `AND`, `OR`, `NOT`,
- `critère` est l'attribut selon lequel ordonner, éventuellement par ordre croissant ou décroissant.

On rappelle aussi les fonctions d'agrégation `MIN()`, `MAX()`, `SUM()`, `COUNT()`, `AVG()` qui s'utilisent *dans* `SELECT` et parfois utilisées en regroupant d'abord des valeurs entres elles selon un autre attribut, typiquement :

```
SELECT fonction(attribut1) AS résultat, attribut2
FROM table
WHERE condition /* condition sur la recherche, avant de faire un calcul */
GROUP BY attribut2
HAVING condition /* condition sur le résultat du calcul */
ORDER BY critère ASC | DESC
```

Nous y ajoutons maintenant la syntaxe générale d'une jointure (voir sur les exemples), qui prend la forme

```
SELECT attributs
FROM table1
JOIN table2
ON critère
```

où `critère` désigne en général l'égalité entre un des attributs de `table1` et un attribut de `table2`.

Il arrive que les deux tables possèdent un même nom d'attribut et alors il faut utiliser la syntaxe `table.attribut`, par exemple dans notre mini réseau social les tables `publications` et `commentaires` contiennent toutes les deux un attribut `date`, qu'on appelle donc `publications.date` et `commentaires.date`, le critère sera donc de les joindre selon `publications.date=commentaires.date`. Éventuellement on utilise `AS` pour les renommer.

À combiner avec toutes les autres commandes SQL vues jusque là, appliquées sur les attributs d'une table ou de l'autre !

III Base de données : observations de mammifères

La base de données `mammiferes.db` contient des informations sur tous les mammifères observés en région parisienne. Il y a deux tables, dans l'une on range toutes les espèces de mammifères, avec diverses informations sur l'espèce, et dans l'autre on rajoute une nouvelle entrée à chaque fois qu'une espèce est observée avec la date et le lieu précis d'observation.

Ces données sont publiques, chacun peut y contribuer, le projet dépend du Museum National d'Histoire Naturelle : <https://openobs.mnhn.fr/>

Exercice 23.1

Observer la base de données. Quelles sont les clés primaires et étrangères ici ?

Le premier exercice contient seulement des révisions et pas de jointures.

Exercice 23.2

Écrire des requêtes SQL pour :

1. Donner toutes les observations par ordre chronologique.
2. Donner tous les ordres d'animaux.
3. Donner la plus grande latitude à laquelle un animal a été observé.
4. Combien d'observations ont été effectuées avant 1980 ?
5. (i) Donner le nombre d'observations par années.
(ii) ... ordonné par nombre d'observations.
6. Donner, pour chaque identifiant d'animal, la latitude moyenne et la longitude moyenne auxquelles il a été observé.

On s'intéresse maintenant à la question de jointure. La syntaxe pour recoller les deux tables est la suivante, testez :

```
SELECT * FROM observations JOIN animaux ON animal=cdRef;
```

Exercice 23.3

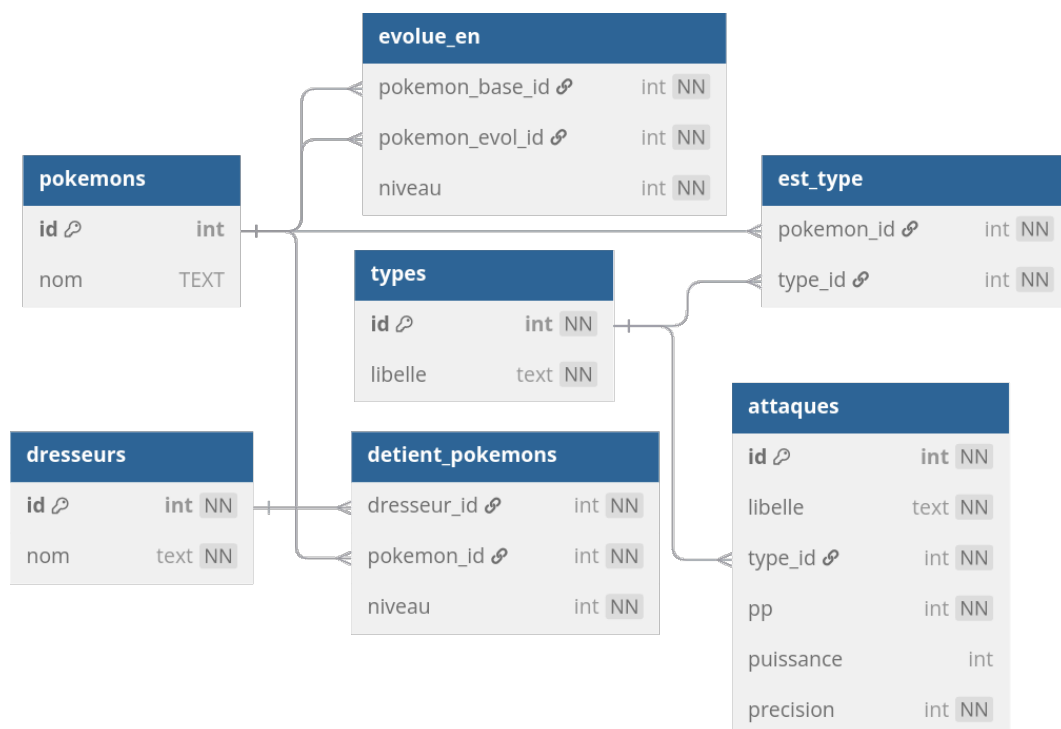
Écrire une requête SQL pour :

1. Donner côte à côte les dates d'observations, noms vernaculaires, et noms scientifiques, des observations.
2. Donner le nom vernaculaire des animaux, classés par date d'observation la plus ancienne.
3. Donner pour chaque famille d'animaux, le nombre de fois où la famille a été observée.
4. Donner le nom de tous les animaux ayant été observés avant 1910.
5. Donner l'espèce du dernier animal ayant été observé.

IV Base de données : attrapez-les tous

La base de données `pokemon.db` contient la liste de tous les Pokémon, avec pour chacun leur type, leurs attaques, et leurs évolutions, ainsi qu'une liste de dresseurs pouvant posséder des Pokémon. La structure de la base est cette fois nettement plus compliquée : il faut bien entendu une table pour chaque Pokémon, mais aussi une table pour indiquer en quoi ils peuvent évoluer ; une table pour contenir les types et savoir quel Pokémon est de quel type ; de même, une table pour contenir les attaques et les relier à un type ; une table des dresseurs, et une autre table pour savoir quel dresseur possède quels Pokémon.

La structure de la base de données est représentée par le diagramme ci-dessous, où les clés primaires sont indiquées par une icône de clé. Les flèches d'une table vers l'autre correspondent à des clés étrangères, indiquées par une icône de cadenas : elles indiquent précisément que la clé est étrangère et vers quelle clé primaire d'une autre table elle pointe.



Exercice 23.4

Observer la base de données.

Le contenu est aussi disponible en ligne en mode interactif : <https://sql-exercices.github.io/>

Pour répondre aux questions, il faudra parfois joindre *plusieurs* tables. La syntaxe ressemble à :

```
SELECT attributs
FROM table1
JOIN table2
JOIN table3
ON critères
```

où *critères* devra être formulé avec des opérations **AND** (plusieurs colonnes à recoller).

Il faudra même utiliser des *auto-jointures* (joindre une table avec elle-même), mais il est alors nécessaire de renommer la table :

```
SELECT attributs FROM table AS table1
JOIN table AS table2
ON critères
```

Celles-ci permettent de mettre en relation une entrée de la table avec une *autre* entrée de la *même* table, éventuellement via d'autres tables intermédiaires.

Exercice 23.5

Écrire des requêtes SQL pour :

1. Donner le nombre total de Pokémon.
2. Afficher le nom de chaque Pokémon avec, à côté, le niveau auquel il évolue.
3. Afficher tous les Pokémon détenus par le dresseur Pierre.
4. Afficher le nom de chaque attaque, avec son type.
5. Afficher chaque dresseur avec chacun de ses Pokémon.
6. Afficher chaque Pokémon avec son niveau d'évolution et le nom de son évolution.
7. (i) Afficher chaque dresseur avec son nombre de Pokémon.
(ii) ... et le niveau moyen de ses Pokémon.
8. (i) Afficher les dresseurs qui possèdent au moins 4 Pokémon.
(ii) ... tous de niveau au moins 50.
(iii) ... ordonné par le niveau du Pokémon le plus élevé qu'ils possèdent.