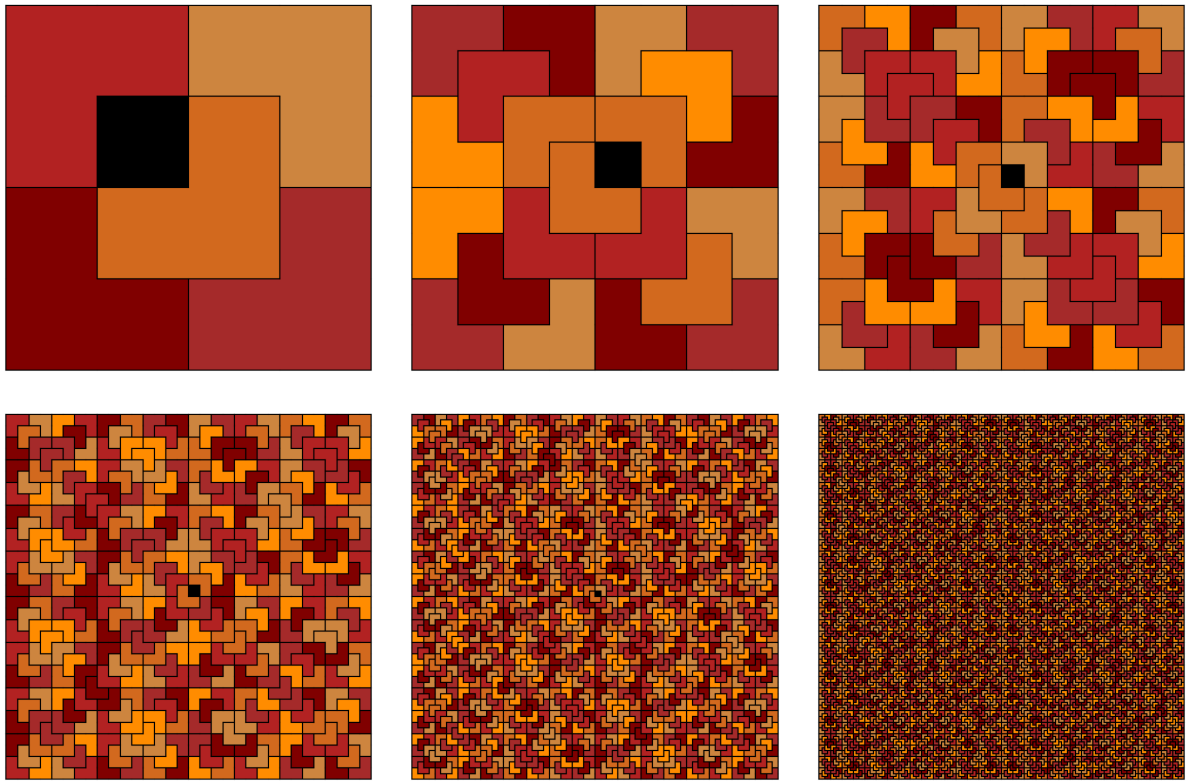
Une introduction à la programmation avec Python

Martin Averseng – 7 Janvier 2026



Introduction

Ce cours est une introduction à la programmation. Un programme est un ensemble d'instructions qui sont écrites dans un *langage* suffisamment dénué d'ambiguité pour qu'on puisse les traduire automatiquement en une séquence d'actions qui peuvent être effectuées par une machine : l'ordinateur. L'apprentissage de la programmation, c'est non seulement l'apprentissage de l'un de ces langages [1], mais aussi et surtout l'art de *concevoir des programmes*, c'est-à-dire, étant donné une tâche potentiellement complexe, de trouver la suite astucieuse d'instructions qui amène (efficacement) l'ordinateur à accomplir celle-ci. De ce fait, la programmation est une tâche essentiellement créative, souvent ludique. L'un des objectifs de ce document est de faire découvrir ce plaisir de programmer.

Entre les mains d'un programmeur, un ordinateur peut devenir un outil exceptionnellement puissant. Voici quelques exemples marquants de ce dont il est capable :

- Décrypter des codes comme Enigma, le code utilisé par l'armée Allemande pendant la seconde guerre mondiale, et inversement, crypter nos conversations privées et transactions bancaires,
- Battre n'importe quel humain aux échecs [2] et maintenant aussi au jeu de Go, [3]
- Mettre les humains en communication (téléphone, internet) et effectuer des recherches par mot-clés et pertinence dans d'immenses bases de données (l'application qui a fait la fortune de Google),
- Générer et diffuser du contenu audiovisuel, pour le montage/retouche photo ou les effets spéciaux au cinéma, la musique électronique, les jeux vidéos, la réalité virtuelle, etc.
- Résoudre des problèmes d'optimisation en industrie. Par exemple, le pilotage des moyens de production d'EDF demande la résolution de ce type de problème pour décider chaque jour combien, où, et quand produire l'énergie. Les solutions trouvées par ordinateur permettent de réaliser des économies (énergétiques et monétaires) considérables par rapport à celles que les humains sont capables de trouver.
- Générer de manière automatique du texte cohérent et répondant à des requêtes précises, en exploitant l'immense base de données qu'est Internet (chatGPT, etc.), et ce, en donnant l'impression d'une intelligence semblable à celle des humains. [4]

Cette liste, loin d'être exhaustive, nous rappelle que les ordinateurs jouent un rôle central dans beaucoup d'aspects de notre vie. Apprendre à programmer, c'est donc aussi découvrir un aspect important du monde dans lequel nous vivons.

[1] Il existe de nombreux langages de programmation différents, comme Fortran, C/C++, Java, Python, etc. Tous les langages permettent de faire accomplir les mêmes tâches à l'ordinateur, mais plus ou moins efficacement. En général, plus un langage est efficace, plus il demandera à l'utilisateur de gérer avec finesse des détails de l'exécution, et donc plus il sera difficile à maîtriser.

[2] Près de 30 ans après la célèbre défaite de Kasparov contre DeepBlue en 1997, la supériorité des ordinateurs aux échecs est aujourd'hui fermement établie. Il semble désormais impensable qu'un humain n'arrache ne serait-ce qu'un match nul à la machine.

[3] Le long règne des humains au Go a été souvent vu comme un indice de la supériorité de l'intuition sur le calcul pur, (de manière assez amusante, on trouve de nombreuses déclarations similaires à propos des échecs 50 ans plus tôt), mais cette supériorité, et la distinction même entre intuition et calcul, n'est plus si évidente depuis que AlphaGo a battu les meilleurs joueurs du monde entre 2015 et 2017.

[4] Cette intelligence potentielle de la machine fut notamment anticipée par Ada Lovelace (1815-1852, la première personne à avoir écrit un programme informatique, fait pour être exécuté par la "machine analytique" de Charles Babbage), puis Alan Turing (1912-1954, l'un des pères fondateurs de l'ordinateur moderne).

Ce cours *utilise* le langage Python,[5] qui est un langage libre, très facile à prendre en main, tout en restant relativement efficace, et qui est utilisé dans de nombreux contextes y compris professionnels. C'est donc un excellent langage pour l'apprentissage de la programmation. Mais ce document n'est pas un cours *de Python*. Nous n'utiliserons ce langage que comme un outil pour découvrir des concepts généraux, communs à presque tous les langages. Nous avons choisi de rester éloigné des spécificités et des nombreux raccourcis qui existent en Python (en particulier, nous ne cherchons aucunement à être "Pythonique"[6]). Selon nous, exploiter à fond le potentiel de Python relève entièrement d'un autre cours, dont l'importance est secondaire par rapport à la découverte de la programmation.

Chaque chapitre est accompagné d'une sélection d'exercices. Ce n'est pas une mauvaise méthode de commencer par les exercices, puis d'aller "à la pêche aux informations" dans le cours à chaque fois qu'on en a besoin. Les premiers exercices sont des applications directes du contenu du chapitre, tandis que les suivants introduisent des idées importantes et classiques d'algorithmique et de programmation. Nous marquons d'une étoile les exercices qui sont fondamentaux.

Ce document est une version remaniée d'un polycopié de cours d'introduction à la programmation dispensé en licence de mathématiques. C'est donc un "cours d'info pour des matheux", et pour cette raison, vous y trouverez majoritairement des exemples mathématiques. En particulier, une attention spéciale est accordée à la *démonstration* de certaines propriétés des algorithmes que nous allons étudier. Par exemple, nous chercherons à démontrer que les programmes que nous concevons finissent par s'arrêter et effectuent correctement la tâche désirée (parfois, cela n'a rien d'une évidence !). Nous avons cherché à limiter un peu les prérequis mathématiques nécessaires. À ce titre, certains rappels sont fournis en annexe (d'autres seraient sans doutes nécessaires mais ne sont pas encore présents par manque de temps).

Je souhaite dédier ces notes à la mémoire de Philippe Testud, en souvenir du cours d'informatique qu'il dispensait quand j'étais son élève en classe prépa et qui m'a beaucoup marqué.

Merci de bien vouloir signaler les erreurs que trouvez par mail à `martin.averseng@univ-angers.fr`.

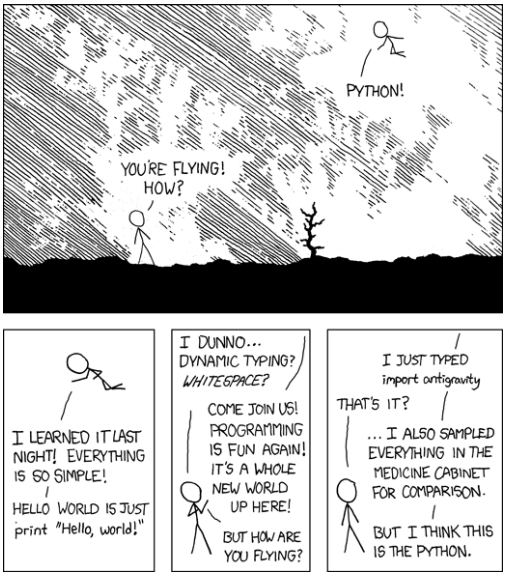


FIG. 1 – Source : `xkcd.com`.

[5] Le concepteur de Python, Guido van Rossum, dit avoir choisi le nom "Python" en référence aux *Monty Python*, une célèbre troupe de comédiens britanniques actifs dans les années 1970.

[6] L'adjectif *pythonique* qualifie le code Python "bien écrit" selon certains standards de la communauté.

Table des matières

1 Objets, variables et mémoire. 5
1 Les objets Python . . . 5
2 Variables . . . 9
3 Représentation en binaire . . . 14

2 Booléens et instructions conditionnelles 20
1 Propositions et booléens . . . 20
2 Les instructions conditionnelles : `if/else, while` . . . 24

3 Tableaux 31
1 Introduction . . . 31
2 Opérations sur les tableaux . . . 32
3 Parcourir les éléments . . . 35
4 Un comportement inattendu . . . 36

4 Fonctions 44
1 Fonctions et algorithmes . . . 44
2 Exécution d'une fonction . . . 46
3 Les fonctions en Python . . . 48
4 Fonctions sur des tableaux . . . 53
5 De nouveaux programmes interminables . . . 54

5 Validité et complexité des algorithmes 58
1 Terminaison et correction d'un algorithme . . . 58
2 Compléxité algorithmique . . . 61

6 Récursivité 69
1 Introduction . . . 69
2 Fonctions récursives en Python . . . 70
3 Complexité des fonctions récursives . . . 71
4 Problèmes à structure récursive . . . 75
5 Diviser pour régner . . . 76

7 Classes 81
1 Définition d'une classe . . . 81
2 Initialisation et affichage . . . 82
3 Attributs . . . 82
4 Méthodes . . . 86

8 Listes et arbres 94

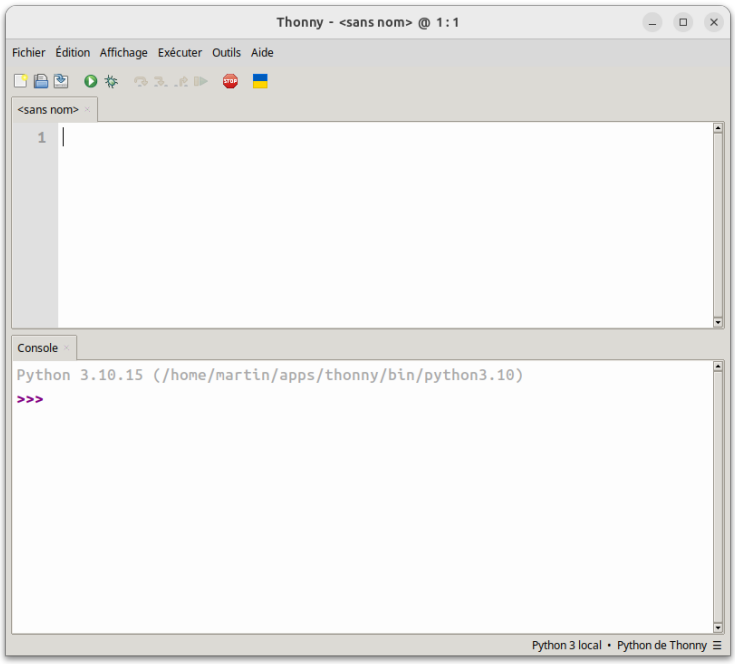
A Rappels mathématiques **95**
1 Suites et séries particulières 95
1.1 Somme des entiers entre 1 et n 95
1.2 Série géométrique 95
2 Principe de récurrence 96
2.1 Démonstration par récurrence 96
2.2 Définition récursive 97
3 Matrices 98

B Textes codés **100**
1 Texte 1 100
2 Texte 2 101

Chapitre 1

Objets, variables et mémoire.

Dans ces notes, nous proposons d'utiliser "Thonny", qui est un IDE (Integrated Development Environment) dédié au langage Python [1]. Selon le système d'exploitation disponible sur votre ordinateur (Windows, Mac ou Linux), utilisez la méthode d'installation appropriée, puis lancez Thonny. Choisissez la langue d'installation que vous souhaitez, puis validez : vous arrivez sur une fenêtre comme ci-dessous



La fenêtre de Thonny se divise en deux zones : en haut, **l'éditeur** vous permet de créer, éditer et enregistrer des fichiers ; en bas, **la console** vous permet d'entrer et exécuter directement des instructions.

1 Les objets Python

Dans la console de Thonny, cliquez à droite des trois chevrons et écrivez

[1] Il existe d'autres IDE dédiés à Python qui sont bien plus "standard", comme Spyder par exemple, ou les jupyter-notebook. Notre décision d'utiliser Thonny est simplement motivée par son extrême simplicité et sobriété, en particulier son absence totale de distraction par des auto-complétions intempestives, suggestions d'amélioration du code etc.

```
>>> "Hello world"
```

sans oublier les guillemets, puis appuyez sur Entrée. Vous voyez s'afficher le texte `'Hello world'`. Pour l'instant, rien de bien impressionnant, Python s'est contenté de répéter ce que vous avez écrit comme un perroquet. Quand vous entrez une expression dans la console, Python *évalue* celle-ci puis *affiche* le résultat. Ici, vous avez entré une chaîne de caractère (signalée par les guillemets), et son évaluation n'est rien d'autre que la chaîne de caractère elle-même. Si vous écrivez

```
>>> 1 + 1
```

cette fois, évaluer l'expression revient à faire l'addition. Les opérations sont aussi possibles sur les nombres à virgule : ils sont écrits avec un point `"."` comme ceci :

```
>>> 1.5 * 2
3
```

En essayant les instructions suivantes, vous devriez pouvoir deviner comment est évaluée l'opération `**` :

```
>>> 2**2
>>> 3**2
>>> 2**3
```

Notez que des calculs difficiles (pour un humain) comme

```
>>> (2**372) * (3**745)
```

sont évalués en une fraction de seconde. Vous pouvez aussi additionner des chaînes de caractère : essayez

```
>>> "Hello " + "world
>>> "1" + "1"
```

Ici Python reconnaît que vous additionnez deux *objets* de type "chaîne de caractères". L'évaluation de ces expression est la *concaténation* (mise "bout-à-bout") des deux chaînes. Ainsi, l'addition, qui est normalement définie pour les nombres, prend un nouveau sens quand elle est appliquée aux chaînes de caractères.

Les objets en Python

Les objets sont les choses que Python peut manipuler et combiner dans des opérations/actions. Chaque objet possède un type : entier, chaîne de caractère, nombre à virgule, ... Les objets "vivent" dans la mémoire de l'ordinateur.

Les types en Python

Il existe plusieurs types "natifs" en Python. Nous venons d'en voir quelques exemples : `<int>` ("integers" : entiers), `<str>` ("character string" : chaîne de caractères), `<float>` ("floating point number" : nombres à virgule flottante [a]). Pour chaque type, un certain nombres d'opérations sur celui-ci sont définies. Nous venons de voir l'addition, la multiplication et la puissance pour les `<int>` et les `<float>`, et la concaténation pour les `<str>`. Plus tard, nous pourrons définir de nouvelles opérations sur des types existants, et aussi créer nos propres types.

[a] Le mot "flottant" vient du format dans lequel sont stockés ces nombres, mais nous ne nous étendrons pas sur celui-ci dans ce cours.

Il est temps de créer votre premier script. Dans la zone *supérieure* de Thonny, écrivez l'instruction `print("Hello world")`. La commande `print` signifie "afficher". Vous pouvez **enregistrer** votre fichier, par exemple sous le nom `hello.py` (".py" est l'extension de fichier pour les scripts Python), dans un dossier nommé par exemple "CodePython". Dans Thonny, **exécutez** ce fichier en appuyant sur la touche F5, ou en cliquant sur la flèche verte en haut de la fenêtre. Notez qu'en exécutant votre fichier, tout ce qui se trouvait dans la console (la zone inférieure) est effacé [2], et toute la mémoire de Python est réinitialisée. Lorsque vous exécutez un fichier, ceci a pour effet de "remettre Python à zéro" puis d'*exécuter* l'une après l'autre chaque ligne du fichier. Contrairement à la console, le résultat de chaque ligne n'est pas *affiché* (sauf si vous en donnez explicitement l'instruction avec la commande `print`). Pour résumer :

(i) La console est intéractive. Vous pouvez entrez directement des commandes qui sont exécutées et affichées. Vous ne pouvez pas enregistrer les commandes que vous y entrez. La console est à utiliser comme une sorte de brouillon.

(ii) L'éditeur n'est pas intéractif. Vous pouvez écrire des instructions ligne après ligne et les enregistrer dans un fichier. Lorsque vous exécutez le fichier, cela exécute toutes les instructions, dans l'ordre. L'éditeur est l'endroit où vous mettrez au propre vos programmes.

Division euclidienne : La division en Python est notée par /. Par exemple

```
>>> 3/2
1.5
```

Mais il existe un autre opérateur de division, la *division Euclidienne*, d'une importance fondamentale en programmation. Par exemple, la division Euclidienne de 15 par 7 donne 2, reste 1. On appelle 2 le *quotient*, et 1 le *reste*. Ces valeurs sont celles que l'on obtient quand on "pose" une division comme à l'école (voir Figure 1.1)

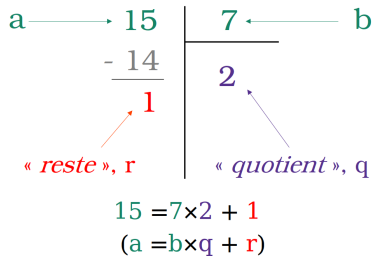


FIG. 1.1 – La division Euclidienne de 15 par 7.

[2] Vous pouvez cependant "remonter" l'historique des commandes entrées dans la console en cliquant à côté des trois chevrons et en utilisant la flèche ↑ autant de fois que nécessaire.

Pour pouvoir formuler des raisonnement impliquant la définition euclidienne, nous aurons très souvent besoin de l'écrire comme ceci : si a et b sont deux entiers, avec $b > 0$, le quotient q et le reste r de la division euclidienne de a par b sont tels que $a = bq + r$ et $r \in \{0, \dots, b-1\}$. Vous savez peut-être (ne serait-ce que par habitude des calculs) qu'un tel couple q et r existe toujours. Mais en cas de doute, cette définition est justifiée par le théorème suivant.

Théorème 1.1 : Division euclidienne

Soient a et b deux entiers. Si $b > 0$, il existe un **unique** couple d'entiers q et r tels que

$$a = bq + r \quad \text{avec} \quad 0 \leq r < b.$$

Démonstration.

1. *(Existence de q et r)* Considérons, parmi la liste de tous les multiples de b (0, b, $2b$, etc. mais aussi $-b$, $-2b$, ...), tous ceux qui sont $\leq a$. Appelons cet ensemble $\mathcal{M}_b$, et considérons N son plus grand élément. Comme N est un multiple de b, on peut l'écrire sous la forme $N = qb$ où q est un entier. On pose alors
$$r = a - qb.$$
Avec cette définition, on a évidemment l'égalité $a = qb + r$, et pour conclure, il reste à vérifier que $0 \leq r < b$.
Pour ce faire, on remarque d'une part que $a \geq qb$ (puisque $qb \in \mathcal{M}_b$), donc $r(= a - qb) \geq 0$. D'autre part, qb est le plus grand multiple de b inférieur ou égal à a. En particulier, $(q+1)b$, qui est plus grand que qb (car $b > 0$), vérifie donc obligatoirement $(q+1)b > a$. En soustrayant qb de part et d'autre de cette inégalité, on voit donc que
$$b > a - qb$$
c'est-à-dire, $b > r$ (par définition de r). On a ainsi montré que $0 \leq r < b$.
2. *(Unicité de q et r)* S'il y avait deux combinaisons possibles q, r et q', r', on aurait $bq + r = a = bq' + r'$, donc $r - r' = b(q' - q)$. En particulier, $r - r'$ serait un multiple de b. Mais à cause de la condition supplémentaire $0 \leq r, r' < b$, $r - r'$ est trop petit être un multiple de b autre que 0. En effet, en ajoutant les inégalités $-b < -r' \leq 0$ et $0 \leq r < b$, on obtient
$$-b < r - r' < b.$$
Ainsi, $r - r' = 0$, c'est-à-dire, $r = r'$. Mais comme $bq + r = bq' + r'$, on en déduit que $bq = bq'$, donc $q = q'$ puisque $b \neq 0$. Ceci établit l'unicité, et donc conclut la preuve.

Remarque : Si $b < 0$, le résultat est encore vrai, cette fois avec $b < r \leq 0$.

En Python, l'opérateur "quotient" de la division euclidienne est désigné par `//`, tandis que l'opérateur "reste" est désigné par `%`. Essayez par exemple :

```
>>> 15//7
>>> 15 % 7
>>> 42 // 2
>>> 42 % 2
>>> 42 % (-5)
```

2 Variables

Assignation

À présent, tapez simplement la commande suivante (sans guillemets)

```
>>> a
```

puis tapez Entrée. Cette fois, Python ne répète pas ce que vous avez écrit, mais se montre un peu plus agressif :

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

À chaque fois que Python vous parle en rouge, il s'agit d'un message d'erreur. Mais inutile de fuir en courant : au contraire, lisez le message attentivement, il va vous aider à comprendre ce qui cloche.

Les erreurs en Python

Les premières lignes vous aident à localiser l'erreur. *Traceback* signifie que Python "remonte" la liste des appels qui ont mené à cette erreur, du plus ancien au plus récent (most recent call last). Ici `<stdin>` veut dire que l'erreur a eu lieu dans la console, ou "standard input". Ce type d'information sera surtout utile quand votre code sera plus long et se décomposera en plusieurs fonctions qui s'appellent les unes dans les autres. La dernière ligne vous indique la nature de l'erreur : ici, nous apprenons que le "nom" `a` n'est pas défini. C'est par là qu'il faut commencer ! Souvent, on gagne du temps en lisant **la fin du message d'erreur** en premier.

La raison est que sans guillemets, Python considère les mots tels que `a` comme des variables, ou plus exactement, des *noms de variable*, et pas des chaînes de caractères.

Définition 1.1 : Variable

Une *variable* est définie par

1. Un *nom*, ou *étiquette*, comme par exemple `a` ou `ma_variable`,
2. Une *valeur*, qui est un objet Python (entier, flottant, chaîne de caractère, ...).

Quand vous avez entré cette commande, vous avez demandé à Python d'afficher la valeur de la variable dont le nom est `a`. Mais pour l'instant, cette variable n'a pas été définie, ce qui explique sa petite saute d'humeur. Remédions à cela en entrant la commande

```
>>> a = 1000
```

Et voilà, vous avez créé votre première variable ! Son nom est `a`, et sa valeur est l'objet `<int>1000`. Une instruction de ce type est appelée une *assignation*. C'est l'une des instructions les plus importantes en programmation.

Définition 1.2 : Assignation

Une *assignation* est une instruction de la forme

`<nom_de_variable> = <expression>`

où

(i) `<nom_de_variable>` est un nom de variable valide, c'est-à-dire un mot formé de lettres, de chiffres et/ou du caractère "_", mais ne commençant pas par un chiffre. Par exemple, `a`, `b`, `ma_variable`, `F1b0n4cc1` sont des noms de variables valides, mais pas `1234` ni `2mbledore`.

(ii) `<expression>` est n'importe quelle expression valide (comme `1000`, `1+1`, `"Hello " + "world"`, etc.)

Remarquez qu'en exécutant l'assignation, c'est le silence radio du côté de Python. Il n'a pas l'air d'avoir réagi ! Mais dans les coulisses il s'est passé beaucoup de choses : d'abord, Python a évalué l'expression `1000`, fabriqué l'objet correspondant au résultat, (en l'occurrence, tout simplement `<int>1000` ici). Puis il est allé le ranger dans son *espace mémoire*, que vous pouvez vous imaginer comme un immense hôtel contenant une quantité astronomique de chambres, ou cases mémoires. Chaque chambre porte un numéro unique. À chaque fois que Python fabrique un nouvel objet, il trouve une chambre inoccupée et y installe le nouvel objet. Pour notre objet `<int>1000`, Python a choisi, disons, la chambre 302. Enfin, Python ajoute le nom de variable (donc ici, `a`) dans son registre et écrit en face le numéro de la chambre occupée par l'objet associé. Sur son registre, Python a donc écrit la nouvelle ligne suivante :

```
=========  REGISTRE  =========
      Nom      |     Adresse
-------------------------------
        a      |       302
```

Dans du texte ou du "pseudo-code"[3], la notation habituelle pour désigner une assignation est `a` ← 1000. Ceci peut être pensé comme *"la variable* `a` *reçoit la valeur 1000"*. De même, les instructions

```
>>> pi = 3.1415                # Ce n'est qu'une approximation !
>>> message = "bonjour"
```

sont des assignations, que l'on note `pi` ← 3.1415 et `message` ← `"bonjour"` en pseudo-code. En les exécutant, vous ajoutez deux nouveaux objets dans l'espace mémoire : un `<float>3.1415` et un `<str>"bonjour"`. De plus, deux nouvelles lignes sont ajoutées au registre pour indiquer les noms `pi` et `message`, ainsi que les adresses des objets associés, voir Figure 1.2.

Les commentaires en Python

Sur la ligne ci-dessus contenant `pi = 3.1415`, vous voyez apparaître le symbole `#` : il permet d'ajouter un commentaire à côté d'une instruction. Tout ce qui est écrit sur cette ligne après le symbole `#` sera ignoré par Python au moment de l'exécution : c'est donc le bon endroit pour commenter votre code, afin d'aider vous-même et les autres à comprendre votre programme.

[3] Le pseudo-code est une façon de décrire un programme à l'aide d'une syntaxe qui imite celle d'un langage de programmation, tout en restant plus lisible qu'un vrai langage de programmation. Le pseudo-code ne peut pas être exécuté directement par un ordinateur.

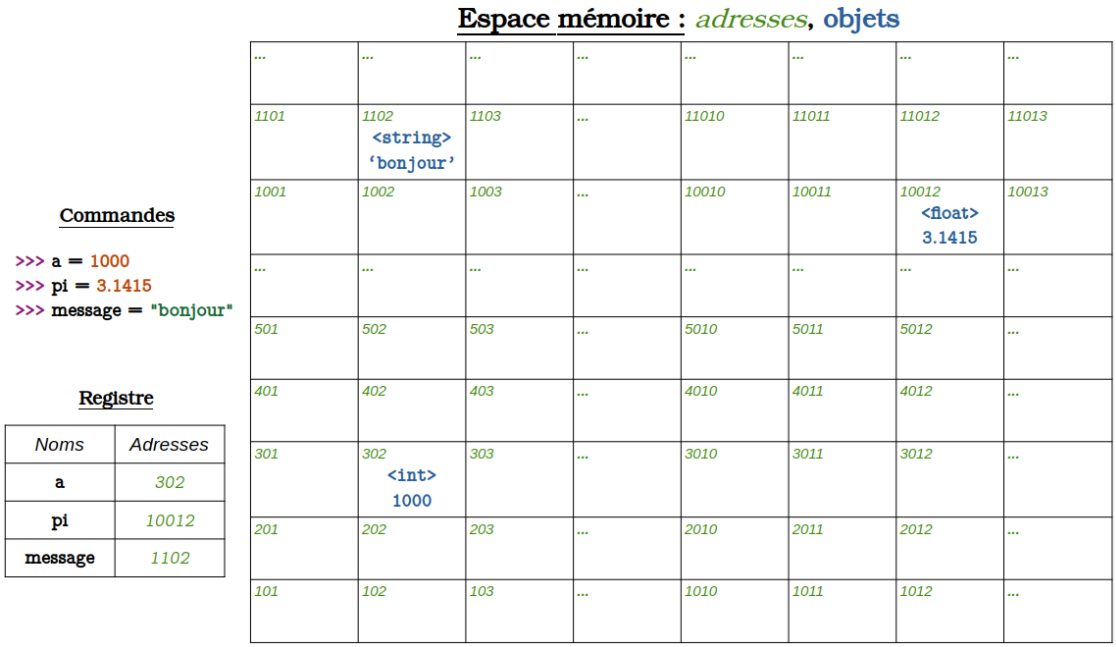


FIG. 1.2 – Représentation schématique de l'espace mémoire.

Pour connaître le vrai numéro de case mémoire que Python a attribué à votre objet, vous pouvez taper la commande `id(a)`. Vous pouvez également afficher le type de cet objet avec `type(a)` :

```
>>> a = 1000
>>> id(a)
131598003905008
>>> type(a)
<class "int">
```

Ici, le numéro de case mémoire est donc 131598003905008 (la valeur que vous aurez sera différente de celle-ci : chaque ordinateur a sa propre gestion de l'espace mémoire). Tapez `type(pi)` et `type(message)` pour vérifier les types des objets associés à ces noms. Si vous tapez `id(bidule)` out `type(bidule)`, cette fois vous aurez une erreur : le nom `bidule` n'est pas sur le registre, donc il n'y a pas de case mémoire associée à `bidule`.

Le signe "=" en Python

Lorsqu'on écrit `a = 1000` en Python, il ne s'agit donc pas d'une proposition vraie ou fausse comme en mathématiques, mais d'une *assignation*. Une assignation est une **instruction** : celle de donner une valeur (ici 1000) à une variable (ici `a`). Pour Python, l'assignation se traduit en détail par les actions suivantes :

1. Évaluer l'expression à droite du signe égal (ici, 1000).
2. Fabriquer un objet contenant le résultat (un `<int>1000`).
3. Lui attribuer une chambre libre (la 302 par exemple).
4. Inscrire au registre le nom de la variable (ici `a`), et lui associer le numéro de la chambre qui vient d'être créée.

À présent, exécutez de nouveau la commande

```
>>> a
```

Cette fois, Python ne vous affiche plus de message d'erreur, mais vous affiche la valeur 1000. Dans les coulisses, Python a consulté son registre, a vérifié que le nom `a` y est inscrit, puis est allé à la chambre enregistrée à ce nom. Il a trouvé l'objet 1000, et a donc affiché cette valeur dans la console. En d'autres termes, quand une variable existe, son évaluation n'est autre que sa valeur.

Réassignation

Une variable est flexible : même après l'avoir assignée, on peut *réassigner* sa valeur. Essayez par exemple

```
>>> a = 1001
```

Pas de message d'erreur : même si `a` est déjà définie, ceci ne pose aucun problème. Python va suivre exactement les mêmes étapes :

1. évaluer l'expression à droite du signe égal (ici, 1001)
2. fabriquer un objet contenant le résultat (un `<int>1001`)
3. lui attribuer une chambre libre (la 403 par exemple)
4. inscrire au registre le nom de variable `a` et lui associer le numéro de la chambre qui vient d'être créée.

L'ancienne ligne du registre contenant la variable `a` est rayée, c'est la ligne la plus récente qui compte. Notons que l'objet `<int>1000` qui occupait la 302 (voir Figure 1.3) est a priori encore dans cette chambre, même si désormais, plus aucune variable n'y fait référence. Il est donc devenu inutile (et il prend de la place !) si bien qu'il sera détruit automatiquement quand Python aura besoin d'une chambre libre.

Puisque l'on peut réassigner les variables, il faut voir le nom de variable `a` plutôt comme une "étiquette", temporairement posée sur un objet, mais que l'on peut librement déplacer sur un autre objet un peu plus tard. Le nom de variable ne fait pas partie intégrante de l'identité de l'objet, ce n'est pas son "prénom/nom de famille". On peut vérifier que ceci correspond bien au fonctionnement interne de Python : en affichant `id(a)`, on constate que le numéro de "chambre" change après la réassignation :

```
a=1000
print("Valeur de a :",a)
print("Adresse de a",id(a))
a=1001
print("Valeur de a :",a)
print("Adresse de a :",id(a))
```

```
>>>
    Valeur de a : 1000
    Adresse de a : 127100534685008
    Valeur de a : 1001
    Adresse de a : 127100531212336
```

Dans ce cours, *nous n'aurons jamais besoin de connaître les adresses de nos variables.* Ce qui est indispensable, c'est d'avoir compris ce qui se passe pendant un assignation : des objets sont créés et vivent dans la mémoire, des variables y font référence, et ces variables peuvent être réassignées.

Pour vérifier votre compréhension de ce qui précède, essayez de répondre aux deux questions suivantes. Nous vous invitons à simuler le comportement de Python en schématisant son espace mémoire et son registre sur du papier.

Commandes

```
>>> a = 1000
>>> pi = 3.1415
>>> message = "bonjour"
>>> a = 1001
```

Registre

Noms	Adresses
~~a~~	~~302~~
pi	10012
message	1102
a	403

Espace mémoire : *adresses*, objets

...	...	...	...	...	...	...	...
1101	1102 <string> 'bonjour'	1103	...	11010	11011	11012	11013
1001	1002	1003	...	10010	10011	10012 <float> 3.1415	10013
...	...	...	...	...	...	...	...
501	502	503	...	5010	5011	5012	...
401	402	403 <int> 1001	...	4010	4011	4012	...
301	302 <int> 1000	303	...	3010	3011	3012	...
201	202	203	...	2010	2011	2012	...
101	102	103	...	1010	1011	1012	...

FIG. 1.3 – L'état de la mémoire après la commande `a = 1001`.

Question 1. Prédisez ce que va afficher Python après ces commandes, en appliquant rigoureusement les étapes d'une (ré)assignation

```
>>> a = 1000
>>> a = a+1
>>> a
```

Vérifiez votre prédiction.

Question 2. Selon vous, si l'on exécute le script suivant, la dernière aire affichée est-elle 3 ou 6 ? Recopiez le code, exécutez-le, et vérifiez votre prédiction.

```
base = 1.5
hauteur = 2

aire = base*hauteur/2
print("au début, aire vaut : ")
print(aire)

base = 3
print("maintenant, aire vaut : ")
print(aire)
```

Assignations vs. proposition

Quand vous tapez l'instruction

```
>>> 1 = 1
```

vous recevez ce message d'erreur :

```
File "<stdin>", line 1
1 = 1
^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '=' ?
```

Ici, Python vous donne encore un coup de pouce. Il affiche la ligne dans laquelle il a trouvé l'erreur, et souligne avec le caractère "^" l'endroit précis du problème. Puis il vous indique qu'il ne peut pas "assigner à un littéral" (c'est-à-dire à 1). En effet, `1 = 1` a la forme d'une assignation. Mais rappelez-vous : 1, à gauche du signe égal, n'est pas un nom de variable valide ! Quand à la suggestion à la fin du message d'erreur ci-dessus... nous allons le voir au chapitre suivant.

3 Représentation en binaire

En utilisant l'écriture en base 2 (ou la représentation binaire), les symboles 0 et 1 suffisent à représenter n'importe quel nombre entier. L'idée est que chaque entier peut s'écrire comme une somme de puissances de 2 (voir les exercices 19-20 à la fin de ce chapitre). Par exemple, $29 = 2^4 + 2^3 + 2^2 + 2^0$ ce que l'on peut réécrire

$$29 = \underset{\uparrow}{1} \times 2^4 + \underset{\uparrow}{1} \times 2^3 + \underset{\uparrow}{1} \times 2^2 + \underset{\uparrow}{0} \times 2^1 + \underset{\uparrow}{1} \times 2^0.$$

La représentation binaire de 29 est alors "11101". De la même manière, $17 = 2^4 + 2^0$ donc 17 s'écrit "1001" en binaire, tandis que 56 s'écrit "111000" ($2^5 + 2^4 + 2^3$). L'idée n'est pas très éloignée de la représentation décimale. Par exemple 1492 s'écrit

$$1492 = \underset{\uparrow}{1} \times 10^3 + \underset{\uparrow}{4} \times 10^2 + \underset{\uparrow}{9} \times 10^1 + \underset{\uparrow}{2} \times 10^0.$$

Les deux seules différences sont le choix de la puissance utilisée (2 en binaire, 10 en décimal), et les chiffres autorisés devant chaque puissance (0 ou 1 en binaire, de 0 à 9 en décimal). Notons quelques propriétés de la représentation binaire :

1. L'écriture binaire de 2 est "10" (puisque $2 = 2^1 + 0 \times 2^0$).
2. Multiplier par 2 revient à ajouter un 0 à la fin (comme multiplier par 10 en décimal). Ainsi, 29 s'écrit "11101" en binaire, et 58 s'écrit "111010".
3. Quotienter par 2 (avec ou sans reste) correspond à enlever le dernier chiffre. Ainsi, 29 s'écrit "11101" et 14 (qui est le quotient de 29 par 2, reste 1), s'écrit "1110".
4. Pour calculer l'écriture binaire d'un nombre, on peut faire sa division euclidienne par 2, puis encore diviser le résultat par 2, et ainsi de suite, en notant la suite des restes de chaque division *de droite à gauche*. Par exemple, 29 divisé par 2 vaut 14, **reste** 1, puis 14 divisé par 2 vaut 7, **reste** 0, puis 7 divisé par 2 vaut 3, **reste** 1, puis 3 divisé par 2 vaut 1, **reste** 1, et enfin 1 divisé par 2 vaut 0, **reste** 1. La suite des restes était 1, 0, 1, 1, 1, ce qui, lu de droite à gauche, donne "11101", l'écriture binaire de 29.
5. Les nombres dont l'écriture binaire est de la forme 10000...00 sont les puissances de 2.

6. Les nombres dont l'écriture binaire est de la forme "111...111" s'écrivent $2^N - 1$. En effet

$$2^k + 2^{k-1} + \ldots + 2^2 + 2^1 + 2^0 = \frac{2^{k+1} - 1}{2 - 1} = 2^{k+1} - 1.$$

en utilisant la formule de sommation des termes d'une suite géométrique (voir la Section 1.2 de l'annexe A). Par exemple, "1111" est l'écriture binaire de $15 = 2^4 - 1$, et "1111111111" est l'écriture de $1023 = 2^{10} - 1$.

Rien n'empêche de remplacer 0 et 1 par n'importe quelle autre paire de deux symboles, du moment que la convention est connue par tous. Cela implique que l'on peut mémoriser un entier avec une suite de signaux "allumé" (pour 1) et "éteint" (pour 0). En convenant qu'une tension de 5 Volts correspond au chiffre 1 et une valeur plus faible correspond à 0, cela permet donc de représenter les nombres par des signaux électriques, et c'est au bout du compte de cette manière que procède l'ordinateur. Un *bit* (pour **b***inary dig***it**) correspond à un signal 0/1. Ainsi, la chaîne de caractères 111000 a une longueur de 6 bits.

De la même manière qu'on représente les réels avec une écriture décimale finie ou infinie (par exemple $\pi = 3.141592...$), tous les nombres réels peuvent aussi être représentés à l'aide d'une écriture *dyadique*. Par définition, étant donnés des entiers S_N, S_{N-1}, ..., S_1, S_0, s_{-1}, s_{-2} etc., tous égaux soit à 0 soit à 1, le nombre

$$x = S_N \cdot 2^N + \ldots + S_0 \cdot 2^0 + s_{-1} \cdot 2^{-1} + s_{-2} \cdot 2^{-2} + \ldots$$

a pour écriture dyadique "$S_N \ldots S_0, s_{-1}s_{-2}s_{-3}\ldots$". Par exemple, le nombre $\frac{11}{8} = 1 + \frac{1}{4} + \frac{1}{8}$ s'écrit "1.011" en écriture dyadique. Comme dans le cas de l'écriture décimale, certains réels (y compris certains rationnels), s'écrivent avec un nombre inifini de chiffres binaires après la virgule. Par exemple, 1/3 a pour écriture dyadique

"0.0101010101..."

(où le motif 0101 se répète à l'infini). Autrement dit, $1/3 = 2^{-2} + 2^{-4} + 2^{-6} + \ldots$, ce que l'on peut vérifier en utilisant encore la formule de sommation des termes d'une série géométrique. En pratique, dans l'ordinateur, on ne peut pas mémoriser une infinité de chiffres, donc les nombres réels sont approximés avec seulement un nombre fini de chiffres binaires après la virgule.

De même, tous les caractères, alphabétiques, numériques ou spéciaux, peuvent être encodés par des nombres entre 0 et $256 = 2^8$, par exemple 0 pour a, 1 pour b, etc., qui eux-même peuvent être écrit en binaire comme une suite de 8 bits, ou un *octet* (*byte* en anglais). Ceci permet donc d'encoder un texte à l'aide d'une suite de 0 et de 1. Par exemple, selon le code décrit ci-dessus, la chaîne de caractères `"bonjour"` s'écrirait

$$\underbrace{00000001}_{\text{“b”}}\,\underbrace{00001111}_{\text{“o”}}\,\underbrace{00001110}_{\text{“n”}}\,\underbrace{00001010}_{\text{“j”}}\underbrace{00001111}_{\text{“o”}}\,\underbrace{00010101}_{\text{“u”}}\underbrace{00010010}_{\text{“r”}}$$

C'est ainsi que sont enregistrés vos fichiers dans votre disque dur : ils sont traduits en une suite de 0 et 1, qui est matérialisée par une suite de petits aimants qui sont orientés soit pôle Nord vers le haut (pour 1), soit vers le bas (pour 0), et tant qu'elle n'est pas activement modifiée, leur orientation reste stable (au moins pendant plusieurs dizaines d'années).

Réponse à la question 1 :

1. *Python évalue l'expression à droite du égal, ici (la valeur actuelle de* `a`*) + 1 = 1001 + 1 = 1002.*
2. *il fabrique un nouvel objet* `<int>1002`
3. *il trouve une chambre libre et y installe le nouvel objet*
4. *il ajoute au registre la ligne associant le nom de variable à gauche, ici* `a`*, à cette nouvelle chambre*

Remarquez que la variable `a` *change donc de valeur* au cours de l'exécution de cette instruction *: pendant l'évaluation du second membre, elle valait* 1001*, et à la fin, elle vaut* 1002*. L'opération*

```
variable = variable + 1
```

peut être comparée au fait d'appuyer sur le bouton d'un cliqueur.

Anecdote : dans certains langages – mais pas en Python – on peut remplacer l'instruction `a=a+1` *par* `a++`*. C'est la raison pour laquelle le langage* `C++` *porte son nom : il est la version "supérieure" du langage* `C`*.*

Réponse à la Question 2 :
Comme vous le voyez, la réassignation de `base` *n'a pas affecté* a posteriori *la valeur de* `aire`*. Python n'établit pas de lien logique entre variables : le seul moyen de modifier la valeur d'une variable est de la réassigner.*

Exercices du Chapitre 1

Types et opérations

1. Créez 3 nouveaux noms de variables de votre choix, chacun associé à l'un des 3 types vus dans le chapitre. Afficher leur type avec Python ainsi que leur adresse mémoire.

2. Quelle est la différence entre les commandes `a = 2` et `b = "2"` ? Que se passe-t-il avec la commande `c = int(b)` ? En procédant par analogie, comment pouvez-vous convertir la chaîne de caractère `pi_str = "3.141592"` en le `<float>` correspondant ?

3. Si `nom1` et `nom2` sont deux variables du même type parmi les trois types précédents, quel est le résultat de l'instruction `nom1 + nom2` ?

4. Si `nom1` est associé à un `<int>` et `nom2` à un `<float>`, quel est le type de `nom1 + nom2` ? Que se passe-t-il si `nom2` est de type `<str>` ? Et si on remplace l'addition par la multiplication ?

5. Testez la commande `1e6`. Comparer avec `10**6` et `10.0**6`. Comparez les commandes suivantes

```
avogadro_1 = 6.022*(10**23)
avogadro_2 = 6.022e23
```

   Que vaut la différence ? Qu'en pensez-vous ?

6. Un script contient les commandes suivantes

```
a = 2**0.5
b = a**2
c = b - 2
print(c)
```

   Si c'était un mathématicien qui réalisait ces opérations que devrait-il afficher à la fin ? Qu'affiche Python ? Ce résultat devrait-il vraiment nous étonner ?

7. Testez le code suivant

```
a = 1000
print("Commande : a = 1000")
print("\t valeur de a :", a)
a += 1
print("Commande : a += 1")
print("\t valeur de a :", a)
a -= 2
print("Commande : a -= 2")
print("\t valeur de a :", a)
a *=2
print("Commande : a *= 2")
print("\t valeur de a :", a)
```

   (notez dans les exemples ci-dessus que la commande print peut afficher plusieurs expressions à la suite, en séparant chaque expression par une virgule. Le caractère `\t` insère une tabulation). En vérifiant sur d'autres exemples, expliquez précisément ce que font les commandes `+=`, `-=` et `*=`.

8. Testez les commandes

```
message = "Hello world"
print(message[0])
print(message[1])
print(message[-1])
print(message[-2])
```

   En vérifiant avec d'autres exemples, expliquez ce que fait en général la commande `message[i]` pour chaque entier $i \in \mathbb{Z}$.

9. Que fait la commande `len(message)` ?

Mémoire

10. Que va afficher le script suivant ? Justifiez, puis vérifiez.

```
a = 1
b = a
a += 1
print(a)
print(b)
```

11. Les instructions suivantes produisent toutes une erreur.

```
>>> 2 = 3
>>> 1 + a = 1 - b
>>> c = (a - b)(a+b)
>>> 1_belle_variable_binaire = 10101
```

Expliquez pour chacune d'elle, quel est le problème.

12* Corrigez ce programme. *(Indice : utilisez une troisième variable).*

```
# Echange de deux variables :
# On veut assigner la valeur
# de a à b, et vice-versa :

a = 1000
b = 2000
a = b
b = a
print(a) # Ok : affiche 2000
print(b) # ??? affiche 2000 aussi
```

13. On suppose qu'un script contient les commandes

```
pokemon = "pikachu"
experience = 457
attaque = 10

adversaire = "bulbizarre"
defense = 4

# Pikachu lance attaque éclair
degats = attaque/defense

# Bulbizarre est KO,
# Pikachu gagne le combat
gagnant = pokemon
experience += 500
```

En utilisant le même type de représentation que dans le cours, schématisez l'espace mémoire et le registre suite à l'exécution de ce script (sur papier).

Représentation binaire

14. Un échiquier contient 64 cases. On pose un grain de riz sur la première case, puis 2 sur la case suivante, et ainsi de suite en doublant à chaque fois le nombre de grains de riz d'une case à la suivante. Combien de grains de riz sont posés au total sur l'échiquier ? Comment s'écrit ce nombre en binaire ?

15. (Sur feuille) Convertir de la base 2 à la base 10 les nombres suivants : "10", "10101", "11000". Écrire en base 2, 3 et 4 tous les nombres de 0 à 15. Donner l'écriture binaire de 247. Combien de nombres peut-on représenter avec au plus 32 bits ? Même question avec 64 bits ? À combien de bits correspondent 4 Go de mémoire (4 giga octets, c'est-à-dire 4 milliards d'octets) ?

16* Complétez le script ci-dessous, pour qu'il affiche la représentation en binaire d'un nombre entre 0 et 63.

```
N = ?? # Entrer ici un nombre
# compris entre 0 et 63

# Ce script affiche la
# représentation binaire de N:

# ... A vous de jouer !
```

17. Montrez que l'écriture dyadique de $\frac{1}{7}$ est

"0.001001001001..."

(le motif "001" se répétant à l'infini). Trouvez l'écriture dyadique de $\frac{1}{10}$. Quelle est l'erreur lorsqu'on tronque cette écriture à 50 chiffres après la virgule ?

18. Recopiez le script suivant et commentez le résultat

```
a = 0.1+0.1+0.1
b = 0.3
print(a)
print(b)
print(b-a)
```

19. Le but de cet exercice est de démontrer que tout nombre entier $N \in \mathbb{N}$ admet (au moins) une représentation binaire. L'unicité de cette représentation fait l'objet de l'exercice suivant.

(a) Soit $n \in \mathbb{N}$ et soit N un entier tel que

$$0 \leq N \leq 2^{n+1} - 1.$$

On remarque que l'une des deux affirmations suivantes est vraie :

(i) $0 \leq N \leq 2^n - 1$,
(ii) $2^n \leq N \leq 2^{n+1} - 1$.

Vérifiez que dans le cas (ii), on peut écrire $N = 2^n + N'$ où N' vérifie la condition (i), c'est-à-dire, $0 \leq N' \leq 2^n - 1$.

(b) Pour chaque entier n, on appelle $P(n)$ la proposition "tous les entiers compris entre 0 et $2^n - 1$ admettent (au moins) une représentation binaire." Démontrez par récurrence (voir la Section 2 de l'appendice A) que $P(n)$ est vraie quelque soit l'entier n. Pour l'hérédité, utiliser la question (a).

(c) Concluez.

20. Le but cet exercice est de démontrer pour tout $n \in \mathbb{N}$ la proposition suivante :
$\underline{P(n)}$ *: Si* $s_0, \ldots, s_n$ *et* $s'_0, \ldots, s'_n$ *sont des entiers égaux à* 0 *ou* 1 *vérifiant l'égalité*

$$\begin{aligned} s_0 \cdot 2^0 + s_1 \cdot 2^1 + \ldots + s_n \cdot 2^n \\ = s'_0 \cdot 2^0 + s'_1 \cdot 2^1 + \ldots + s'_n \cdot 2^n \,, \end{aligned}$$

alors $s_0 = s'_0$, $s_1 = s'_1$, *...,* *et* $s_n = s'_n$.

(a) Soit $n \in \mathbb{N}$ et soient $s_0, s_1, \ldots, s_n$, $s'_0, \ldots, s'_n$ vérifiant les hypothèses de $P(n)$. Montrer que s_0 et s'_0 sont égaux au reste de la division euclidienne de N par 2.

(b) Déduisez de ce qui précède que

$$s_1 2^0 + \ldots + s_n 2^{n-1} = s'_1 2^0 + \ldots + s'_n 2^{n-1}$$

(c) À l'aide des deux questions précédentes, démontrez par récurrence que $P(n)$ est vraie quelque soit l'entier n.

Chapitre 2

Booléens et instructions conditionnelles

"Lorsque, par ardeur à ce projet, je m'appliquai plus intensément, je tombai inévitablement sur cette merveilleuse observation, à savoir que l'on peut concevoir un certain alphabet des pensées humaines et que, par la combinaison des lettres de cet alphabet et par l'analyse des mots qui en découlent, on peut à la fois découvrir et juger toutes choses. Lorsque je compris cela, je fus tout à fait ravi, en vérité, d'une joie enfantine."

Gottfried Leibniz, *Préface à une caractéritique universelle.*

1 Propositions et booléens

Nous commençons par introduire deux notions importantes : les propositions et les booléens. Elles permettent de formaliser la notion de vérité et de vérifier la validité d'une affirmation par un calcul.

Propositions

Définition 2.1 : Proposition

Une *proposition* est une phrase déclarative qui a une valeur de vérité bien définie : vraie, ou fausse.

Par exemple, *"Les hommes sont mortels"*, *"La terre est orange*", et "$3^3 = 26$" sont des propositions (la première est vraie, les deux suivantes sont fausses). Mais *"Asseyez-vous !"*, *"Comment t'appelles-tu ?"* et "2×3" ne sont pas des propositions. On peut combiner les propositions de trois manières fondamentales à l'aide de *connecteurs*. Si P, Q sont deux propositions, alors

(i) "**Non** P" est une proposition, vraie si et seulement si P est fausse

(ii) "P **et** Q" est une proposition, vraie si et seulement si P et Q sont vraies

(iii) "P **ou** Q" est une proposition, vraie si et seulement si au moins l'une des propositions P, Q est vraie.

Par exemple,

P : "*Les chats peuvent respirer sous l'eau* **ou** *les chats peuvent cracher du feu*"

est une proposition, qui est fausse. Il est important de noter que le **ou** logique est toujours *inclusif*, contrairement au langage courant où le mot "ou" peut avoir un sens inclusif ou exclusif selon les contextes[1]. Par exemple, la proposition

"Kurt a adopté un chat **ou** *Kurt a adopté un chien"*

est *vraie* dans les 3 cas suivants : Kurt a adopté un chat mais pas un chien, Kurt a adopté un chien mais pas un chat, Kurt a adopté un chien et un chat.

Python et les propositions

On a vu au chapitre précédent que la commande `1 = 1` déclenche une erreur dans Python. Mais il est quand même possible de créer un objet Python qui correspond à la proposition "$1 = 1$" ; pour cela, il faut utiliser le symbole "`==`" (deux fois `=`). Par exemple, entrons la commande suivante dans la console :

```
>>> 1 == 1
```

Cela signifie "1 est égal à 1". A cette proposition, Python attribue la valeur `True`. Soyons un peu plus audacieux, et entrons

```
>>> 1 == 2
```

Là encore, Python a l'attitude raisonnable : il répond `False`. Le même mécanisme fonctionne avec des noms de variable. Si l'on écrit

```
>>> a = 1 # Assignation, un seul "=". Rien d'affiché
>>> a == 1 # Proposition, deux "="
True
>>> a == 2
False
```

la première ligne est une assignation, comme on l'a vu au chapitre précédent. Elle a pour effet de créer un objet `<int>1` et de l'assigner à une variable nommée `a`. Les deuxièmes et troisièmes lignes sont des propositions, et Python leur attribue les valeurs True et False. De même, à la proposition

```
>>> a*2 == a+a
```

Python affectera la valeur True (quelque soit la valeur assignée à `a`) mais avec

```
>>> a == a + 1
```

cette fois, c'est False. La proposition

```
>>> a % 2 == 0
```

[1]Par exemple, le "ou" dans la phrase "vous pouvez prendre du fromage ou un dessert" sera interprété naturellement comme un "ou" exclusif (fromage ou dessert mais pas les deux), alors que dans la phrase "Vous pouvez participer si vous avez le permis voiture ou le permis moto", il sera interprété comme un "ou" inclusif (permis voiture, permis moto, ou les deux).

vaudra `True` si `a` est assigné à un nombre pair, et `False` sinon. On peut aussi utiliser les opérateurs `>`, `>=`, `<`, `<=` ou encore `!=` (différent de) pour créer des propositions :

```
>>> 1 <= 1
>>> 2 > 1
>>> 3 != 4
>>> 1 > 1
>>> a**2 < 0
```

Les trois premières expressions donnent `True`, et les deux suivantes donnent `False` (en supposant que `a` est de type `<int>` pour la dernière). Écrivons à présent

```
>>> b = (a == 1001)
```

(les parenthèses sont superflues). Python est resté silencieux. Que s'est-il passé ? La même chose qu'au chapitre précédent : nous venons de faire une assignation. Nous avons créé un objet, et associé le nom de variable `b` à cet objet. Mais quel est cet objet ? Pour le savoir, exécutons les lignes suivantes :

```
>>> b
>>> type(b)
```

Vous devriez observer que le nom `b` est associé à un objet qui vaut "True", et qui est de type `<bool>`, c'est-à-dire, un *booléen*[2].

Booléens

Définition 2.2 : Booléen

Un *booléen* est un objet qui vaut soit *"Vrai"* soit *"Faux"*.

Pour raccourcir, on note parfois 0 pour *Faux* et 1 pour *Vrai*. On peut assigner un unique booléen, appelé *valeur de vérité*, à chaque proposition P. Par exemple, *"2 est un nombre premier"* et "$3! = 6$" sont deux propositions différentes, mais elles ont la même valeur de vérité, le booléen *Vrai*.

Les connecteurs **non, et, ou** se reflètent par des *opérations* sur les booléens :

1. La *négation*, notée "$\neg$". Si b est un booléen, alors $\neg b$ est le booléen contraire de b. Ainsi,
$$\neg 0 = 1\,, \quad \neg 1 = 0.$$
2. Le *ou logique*, qu'on note "$\vee$". Si a et b sont deux booléens, alors $a \vee b$ est le booléen qui vaut *Vrai* si l'un au moins des deux booléens a ou b vaut *Vrai*, mais *Faux* si a et b sont tous les deux Faux. Ainsi,
$$0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1$$
Si b est un booléen, alors on a $b \vee (\neg b) = 1$: toute proposition est vraie ou fausse.
3. Le *et logique*, qu'on note "$\wedge$". Si a et b sont deux booléens, alors $a \wedge b$ est le booléen qui vaut *Vrai* si a et b valent tous deux *Vrai*. Ainsi,
$$0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1$$

[2] du nom du mathématicien et logicien britannique George Boole

Pour tout booléen b, on a $b \wedge (\neg b) = 0$: une même chose et son contraire ne peuvent pas être vrais simultanément.

Origine de la notation

La notation $\vee$ vient du latin *vel* qui signifie "ou inclusif", alors qu'il y a un autre mot latin – *aut* – pour le "ou exclusif".

Les opérations sur les booléens peuvent être représentées avec des *tables de vérité* :

b	$\neg b$
0	1
1	0

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Les tables de vérités sont des outils pratiques pour vérifier la valeur de vérité d'une formule contenant des booléens, en fonction des valeurs de ceux-ci. Par exemple, la table de vérité

a	b	$a \vee b$	$\neg a \wedge \neg b$	$(a \vee b) \vee (\neg a \wedge \neg b)$
0	0	0	1	1
0	1	1	0	1
1	0	1	0	1
1	1	1	0	1

que l'on peut calculer facilement ligne par ligne, de gauche à droite, nous permet de voir que le booléen

$$c = (a \vee b) \vee (\neg a \wedge \neg b)$$

est toujours vrai, quelques soient les valeurs des booléens a et b. On appelle une telle formule une *tautologie*. En général, pour faire une table de vérité, on inclut une une colonne pour chaque variable dans la formule (comme a et b ci-dessus) et on met autant de lignes que nécessaires pour représenter toutes les combinaisons possibles de valeurs. S'il y a une seule variable, il faut 2 lignes (Vrai ou Faux) ; pour 2 variables, il faut 4 lignes (Vrai-Vrai, Vrai-Faux, Faux-Vrai, Faux-Faux). En général, pour n variables, il faut 2^n lignes (n choix indépendants de 2 valeurs). [3]

Le type `<bool>`

Comme nous l'avons vu, les expressions Python comme `1 == 1` sont évaluées comme des objets de type `<bool>`. Ce type reflète les propriétés mathématiques des booléens. Un objet de type `<bool>` ne peut prendre que deux valeurs possibles, `True` et `False`. Les expression et les booléens correspondants peuvent être combinés avec les opérateurs `not`, `and` et `or`, qui sont les contre-parties de $\neg$, $\wedge$ et $\vee$. Par exemple,

```
>>> not True
False
>>> (1 > 1) or (1 < 1)
False
```

[3] Pour lister les combinaisons, on peut compter en binaire : par exemple, pour 4 variables, les 16 combinaisons sont 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101 et 1111, soit 0, 1, 2, ..., 14, 15 en binaire.

```
>>> (4%2 == 0) and (6%2 == 0)
True
>>> not (True and (False or True))
False
```

2 Les instructions conditionnelles : `if/else, while`

Nous allons à présent voir deux nouveaux types d'instructions **fondamentales** :

(i) Les mots-clés `if/else` qui permettent de n'exécuter un groupe d'instructions **que si** une certaine condition est vérifiée,

(ii) Le mot-clé `while` qui permet d'exécuter un groupe d'instruction **en boucle, tant que** une certaine condition est vérifiée

Ces deux types d'instructions sont cruciales, et existent dans n'importe quel langage de programmation. Ce sont grâce à elles que vos programmes se différencieront d'une simple calculatrice, et pourront effectuer des tâches structurées en s'adaptant aux données d'entrée et aux résultats des actions précédentes.

If/else

Commençons par `if/else`. Recopiez le code suivant dans un fichier Python (n'oubliez pas les " :" ni l'indentation)

```
porte_monnaie = 1    # Argent dans le porte-monnaie
prix_glace = 3       # Prix d'une glace

print("J'ai",porte_monnaie,"euros dans mon porte-monnaie")
print("Une glace coûte",prix_glace,"euros.")

if porte_monnaie >= prix_glace:
    # Achat d'une glace
    print("J'ai assez pour acheter une glace !")
    porte_monnaie = porte_monnaie - prix_glace
    print("Miam !")

# Affichage de l'argent restant
if porte_monnaie == 0:
    print("Je n'ai plus d'argent.")
else:
    print("Il me reste",porte_monnaie,"euros.")
```

Exécutez votre fichier. Vous allez voir que certaines instructions sont exécutées, et d'autres non. Relancez le script en mettant assez d'argent dans le porte-monnaie. Cette fois-ci, d'autres instructions sont exécutées. Prenez un moment pour vérifier que ce qui est affiché par Python vous semble logique. En général, on a la syntaxe suivante :

```
if <condition>:
    # L'indentation est importante !
    # Ce code est exécuté seulement si <condition> vaut True
    # ...
```

```
    # ...
else:
    # Le bloc "else" est optionnel.
    # Ce code est exécuté seulement si <condition> vaut False
    # ...
    # ...

# Le code qui suit n'est plus concerné par <condition>.
# Il est exécuté dans tous les cas.
```

Elle permet de n'exécuter le bloc d'instructions qui suivent (délimité par l'indentation) **que si** `<condition>` s'évalue à **True**. Le mot-clé *optionnel* `else:` permet de créer un bloc qui n'est exécuté que dans le cas contraire. Le schéma de la Figure 2.1 illustre ce fonctionnement.

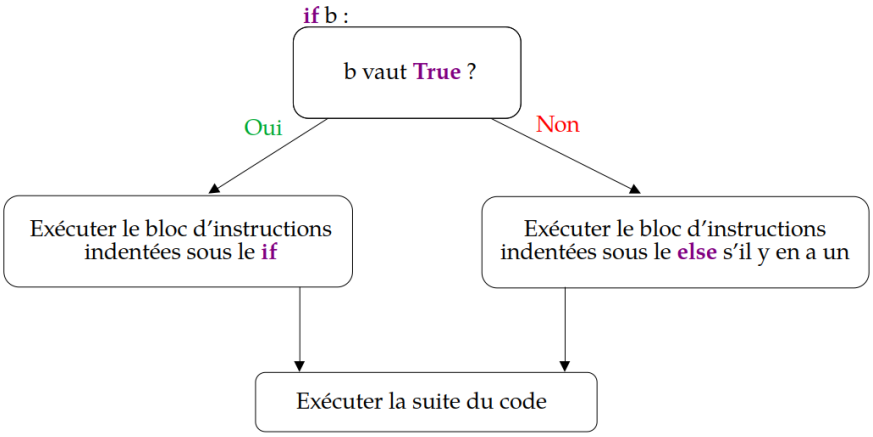


FIG. 2.1 – L'instruction `if / else`.

Un bloc indenté doit contenir au moins une instruction, sinon Python renvoie une erreur :

```
a = 1
b = 0
if a==0: # bloc indenté vide pour l'instant
else:
    b = 1/a
```

```
Traceback (most recent call last):
    File "/home/Code/blocindentevide.py", line 4
        else:
        ^^^^
IndentationError: expected an indented block after 'if' statement on line 3
```

Ceci peut être énervant si on souhaite compléter le programme plus tard. Ajouter une ligne de commentaire ne change rien.

```
a = 1
if a == 0:
```

```
    # Je m'occuperai de ce cas plus tard
else:
    b = 1/a
```

(même erreur). Dans ce type de cas, on utilise le mot-clé `pass`, comme ceci :

```
a = 1
if a == 0:
    pass
else:
    b = 1/a
```

L'instruction `pass` ne fait rien d'autre que remplir le bloc indenté pour éviter l'erreur précédente.

L'indentation en Python

L'indentation, c'est l'ajout d'une tabulation au début d'une ligne. Vu ce qui précède, les indentations en Python ne sont pas qu'une question de décoration : elles influencent directement ce qui sera exécuté ou non dans votre code. En particulier, la fin d'un bloc `if` est signalée par un retour à l'indentation précédente. La plupart des autres langages de programmation utilisent une syntaxe avec délimiteurs comme `if (condition){ instructions }` ou encore `if (condition) instructions endif`. Attention aux confusions !

While

Une autre instruction conditionnelle très importante est la boucle `while` ("tant que", en français). Découvrons-la avec un exemple

```
i = 5
print("Parés pour le décompte ? ")
while i > 0:
    print(i)
    i = i - 1
print("Décollage !")
print("Valeur finale de i :",i)
```

```
Prêts pour le décompte ?
5
4
3
2
1
Décollage !
Valeur finale de i : 0
```

Expliquons ce qui s'est passé. Comme pour l'instruction conditionnelle `if`, le bloc de code indenté sous la condition while n'est exécuté que si la condition `i > 0` vaut True. Or, au départ i vaut 5, donc la condition est satisfaite : on exécute donc le bloc indenté. Ceci a pour effet d'afficher i, donc 5 (instruction

`print(i)`), puis de diminuer la valeur de i de 1 (instruction `i = i-1`). Mais cette fois, au lieu de continuer à la suite, comme dans le cas d'un `if`, **Python retourne à l'instruction `while` et teste de nouveau si la condition est satisfaite**. Désormais, i vaut 4, ce qui est toujours > 0, donc le bloc de code indenté est exécuté une deuxième fois. Le même mécanisme continue, et ne prend fin que quand la variable i passe à la valeur 0 : à ce moment-là, la condition n'est plus vérifiée, et Python cesse d'exécuter le bloc indenté. La suite du code peut enfin être exécutée. De manière générale, dans le code suivant,

```
while <condition>:
    # Bloc d'instructions répétées tant que <condition> est True
    # ...
    # ...
    # ...
# Suite des instructions
```

Python exécute le bloc de code indenté en boucle *tant que* `<condition>` vaut `True`, et passe à la suite lorsque `<condition>` vaut False. Le schéma ci-dessous illustre ce fonctionnement :

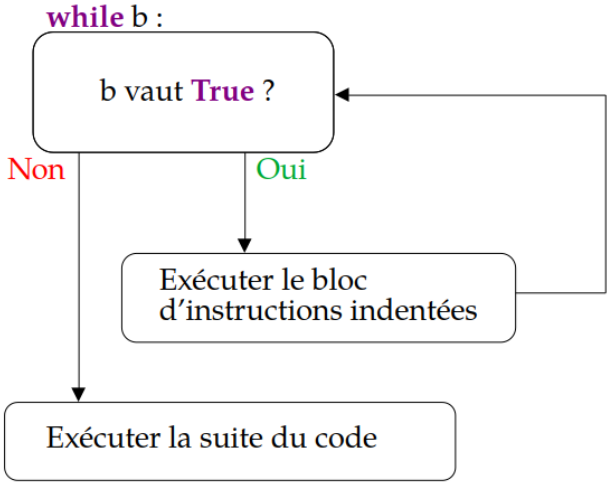


FIG. 2.2 – La boucle `while`

Boucles infinies

Les boucles `while` sont très puissantes... mais attention ! De grands pouvoirs impliquent de grandes responsabilités. Comme vous l'avez peut-être deviné, en principe, rien n'empêche qu'une boucle `while` reste bloquée à tourner en rond. Imaginez par exemple que le bloc indenté ne modifie jamais la valeur du booléen `b`. Dans ce cas, à chaque fois qu'elle sera testée, la condition `b` vaudra `True`, et l'ordinateur, tel un Sysiphe mécanique, sera donc condamné à exécuter éternellement le même bloc de code.

Voici un exemple très simple de cette possibilité :

```
i = 0
while i >= 0:
```

```
    print(i)
    i = i+1
print(i) # À part chuck Norris, personne n'a atteint cette instruction.
```

Exécutez-le. Lorsque vous en avez assez, vous pouvez interrompre l'exécution en cliquant sur le bouton en forme de panneau stop (ou presser Ctrl + C) – sinon, l'ordinateur n'est pas prêt de s'arrêter. De manière générale, souvenez-vous que dès que vous écrivez "`while`", il existe un risque (par expérience, non-négligeable...) de boucle infinie. Si votre code a l'air trop lent, c'est par là qu'il faudra chercher l'explication en premier !

Boucle `for`

Il arrive très souvent qu'on effectue une boucle du type

```
N = 100
i = 0
while i < N:
    # Bloc à exécuter pour i allant de 0 à N-1:
    <instruction>
    ...
    # Fin du bloc. Incrémentation de i
    i = i+1
# suite du programme
```

Puisque i augmente de 1 à chaque itération, ceci a pour effet d'exécuter le bloc indenté 100 fois. Il existe une syntaxe dédiée spécialement pour ce type de cas : la boucle `for`. Elle permet d'écrire un programme équivalent mais un peu plus lisible :

```
N = 100
for i in range(0,N):
    # Bloc à exécuter pour i allant de 0 à N-1:
    <instruction>
    ...
    # Fin du bloc.
# suite du programme
```

Plus généralement, on a la syntaxe suivante

```
for i in range(m,n):
    # Dans ce bloc, la variable i parcourt m,m+1,...,n-1
    # Le bloc indenté est exécuté une fois pour chaque valeur de i
# suite du programme
```

Le mot "range" vient du fait qu'en anglais, "pour i allant de m à n" se dit "for i ranging from m to n". Si m vaut 0, on peut remplacer `range(0,n)` par `range(n)`. Prenez bonne note de la convention troublante de Python qui consiste à inclure m **mais exclure** n de `range(m,n)`.[4] Voir aussi les questions 8–9 du TP1.

[4] Certains informaticiens trouvent cela logique. Cela vient en partie du fait qu'avec cette définition, `for i in range(10)` contient 10 instructions : $i = 0, 1, ..., 9$. Mais comme vous le verrez bien vite, cette convention peut devenir désagréable : il faut s'y habituer !

Exercices du Chapitre 2

Booléens

1. Lesquelles de ces expressions sont des assignations ? Lesquelles sont des propositions ?
   (i) `message == "Hello world"`
   (ii) `42 = 6*7`
   (iii) `a = 1 == 2`
   (iv) `0 == (1 == 2)`
   (v) *"Vive Thonny !"*
   (vii) *"25 est le double de 2, n'est-ce pas ?"*
   (viii) *"Sans Thonny, Python n'existerait pas."*
   (ix) *"Pour toute information, veuillez contacter le service client"*

2* Soient a et b deux booléens. En utilisant une table de vérité, démontrez les relations suivantes

$$\neg(\neg a) = a$$
$$\neg(a \wedge b) = (\neg a) \vee (\neg b)$$
$$\neg(a \vee b) = (\neg a) \wedge (\neg b)$$

3* Soient a, b et c trois booléens. En utilisant une table de vérité, démontrez les relations suivantes :

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c$$
$$a \vee (b \vee c) = (a \vee b) \vee c$$
$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$
$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

Comparez avec la distributivité des lois $+$ et $\times$ sur les entiers naturels.

4. Etant donnés deux booléens a et b, on définit le booléen

$$a \odot b := \neg(a \wedge b).$$

(cette opération est souvent écrite a NAND b, pour "not and"). En utilisant les deux exercices précédents, démontrez les relations suivantes :

$$\neg a = a \odot a$$
$$a \vee b = (a \odot a) \odot (b \odot b)$$
$$a \wedge b = (a \odot b) \odot (a \odot b)$$

Conditions if/else

5. Recopiez et exécutez le script suivant, qui montre comment utiliser la commande "`input`" de Python :

```
nom = input("Entrez votre nom : ")
prenom = input("Entrez votre prénom : ")
age = input("Entrez votre âge : ")

print("Bonjour",prenom,nom,"!")
print("Vous avez",age,"ans.")
```

Écrivez un script qui contient un texte secret, et qui l'affiche seulement si l'utilisateur entre un mot de passe que vous avez choisi.

6* Complétez le script suivant

```
a = ?? # Entrer un nombre
b = ?? # Entrer un nombre
# A vous de jouer
print(max_ab)
print(min_ab)
```

Le but est d'assigner le maximum des deux valeurs à `max_ab` et le minimum à `min_ab`. Le script doit fonctionner quelques soient les valeurs de `a` et `b`, sans modifier le reste.

7. Recopiez et complétez le script suivant

```
AB = ?? # Entrer un nombre
BC = ?? # Entrer un nombre
AC = ?? # Entrer un nombre
```

Le but est de vérifier que le triangle ABC défini par les côtés de longueurs AB, BC et AC est valide, et d'afficher un message d'erreur sinon. Il faut vérifier que chacune des longueurs est inférieure ou égale à la somme des deux autres.

8. Complétez le programme suivant

```
# Résolution du trinôme ax^2 + bx + c = 0
# Entrer les coefficients :
a = ??
b = ??
c = ??
```

Ce programme doit afficher x_1 et x_2 les solutions réelles de l'équation $ax^2 + bx + c = 0$, ou afficher "aucune solution".

Boucles for et while

9* Créez un programme qui affiche 1^2, puis 2^2, ..., jusqu'à n^2, où n est une variable entière, puis un programme qui affiche la valeur de la somme

$$S_n = 1^3 + 2^3 + 3^3 + \ldots + (n-1)^3 + n^3.$$

Que vaut S_9 ?

10* Créez un programme qui calcule $n!$ (c'est-à-dire $n \times (n-1) \times \ldots \times 3 \times 2 \times 1$), où n est une variable entière définie au début du script.

11. Créez un programme qui contient deux variables entières N et k et affiche la valeur de la somme

$$1 + \frac{1}{2^k} + \frac{1}{3^k} + \ldots + \frac{1}{N^k}$$

En prenant $k = 2$, affichez le résultat pour des valeurs de plus en plus grandes de N.

12. Créez un programme qui reçoit un message par la fonction input, et affiche le message à l'envers. Par exemple

```
Message -> Thonny est mon ami
Message inversé -> ima nom tse ynnohT
```

13. Créez un programme qui reçoit un message par la fonction `input`, et affiche le même message mais en "bégayant", c'est-à-dire, en répétant deux fois chaque mot. Par exemple,

```
Message -> Vive Thonny !
Message bégayé -> Vive Vive Thonny Thonny !!
```

14. Créez un programme qui reçoit un entier N par la commande `input` et qui affiche tous ses diviseurs dans la console.

15. Un nombre est "parfait" s'il est égal à la somme de ses diviseurs autres que lui-même. Par exemple, $6 = 3 + 2 + 1$ est parfait. Modifiez votre programme de la question précédente pour qu'il termine par un message indiquant si le nombre entré est parfait.

16* La suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ est définie par récurrence (voir Annexe A, Section 2) par

$$F_n = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-2} + F_{n-1} & \text{pour } n \geq 2 \end{cases}$$

Créez un programme qui affiche les 1000 premiers termes de cette suite. Afficher également les termes de la suite $(r_n)_{n\in\mathbb{N}}$ des rapports consécutifs

$$r_n = \frac{F_{n+1}}{F_n}.$$

Que remarquez-vous ?

17. Complétez le programme suivant :

```
b = ?? # Base entre 2 et 10
N_b = "??" # Ecriture en base b
# (avec les chiffres 0,...,b-1)
# ... à vous de jouer ....
print(M_b)
```

Le programme définit d'abord une variable entière `b`, et une chaîne de caractères `N_b` représentant un nombre N en base `b`. Le but est que le programme affiche `M_b`, la représentation en base `b` de $M = N + 1$.

18. Dans Thonny, cliquez sur "Outils", "Gérer les paquets". Dans la barre de recherche, taper "random" puis cliquer sur rechercher. Dans les résultats, cliquer sur "random11" et en bas de la fenêtre, cliquer sur "installer". Puis vérifiez que le programme ci-dessous fonctionne et affiche des nombres aléatoires entre 1 et 100 (en l'exécutant plusieurs fois) :

```
import random
a = random.randint(1,100)
print(a)
```

On se propose de programmer le jeu suivant : Python tire un nombre entier au hasard entre 1 et 100. Le joueur essaye de le deviner en entrant des valeurs. Le programme indique pour chaque valeur incorrecte si elle est trop petite ou trop grande, et affiche un message lorsque le joueur a gagné. Créez un programme pour jouer à ce jeu.

19. Inversement, on se propose de créer un programme capable de jouer au jeu précédent, mais cette fois, le joueur (humain) pense à un nombre entre 1 et 100, et c'est l'ordinateur qui doit le deviner. À chaque tour, l'ordinateur affiche sa tentative dans la console, et demande à l'utilisateur d'entrer `1` si son nombre est plus grand et `0` s'il est plus petit. Votre programme gagne s'il trouve votre nombre en moins de 10 coups.

Chapitre 3

Tableaux

1 Introduction

Dans ce chapitre, nous allons apprendre à manipuler un nouveau type d'objets : les tableaux (ce que Python appelle `<list>`, voir ci-dessous). Un tableau est une sorte de longue boîte contenant des cases numérotées (à partir de 0), dans chacune desquelles on peut ranger un objet différent. C'est donc un peu comme une "mémoire miniature", si l'on se remémore la représentation schématique vue au Chapitre 1. Les objets contenus dans les cases sont souvent appelés les *éléments du tableau*. On *accède* à l'élément qui se trouve dans la case portant le numéro `i` par la syntaxe `T[i]`.

Un tableau `T`

T[0]	T[1]	T[2]	T[3]
<int> 1000	<str> "bonjour"	<float>3.1415	<bool> True

On peut consulter et modifier directement le contenu de chaque case, et même ajouter ou retirer des cases à la fin du tableau. Essayez par exemple les instructions suivantes, pour voir leur résultat

```
>>> T = [1000,3.1415,True] # assignation
>>> type(T)
>>> T # Affichage du tableau
>>> len(T) # Longueur du tableau
>>> T[0] # Consulter le contenu de la case numéro 0
>>> T[1] # Consulter le Contenu la case numéro 1
>>> T[2] = False # Modifier le contenu de la case numéro 2
>>> T
>>> T.append("le terme 'append' signifie 'adjoindre' en anglais")
>>> T.append(1+1) # Cette syntaxe avec un "." sera revue au chapitre sur les classes
>>> T
>>> len(T)
>>> T.pop() # Retire la dernière case du tableau et renvoie la valeur qui s'y trouvait
>>> T
```

On peut aussi créer un tableau de longueur 0, puis lui rajouter ses cases une par une :

```
>>> T = []
>>> T
```

```
[]
>>> len(T)
0
>>> T.append(0)
>>> T.append(1)
>>> T.append(2)
>>> T
[0,1,2]
```

Une autre possibilité est de créer de grands tableaux, si l'on anticipe déjà que l'on aura besoin de beaucoup de place. Par exemple, si l'on ne veut rien mettre dans les cases pour l'instant, on peut utiliser l'objet `None` (l'objet "rien") comme ceci

```
>>> T = [None]*1000 # Crée un tableau vide de longueur 1000
>>> T
[None,None,None,None,...]
```

Tableaux ou listes ?

Les tableaux en Python s'appellent "`list`", mais nous marquons la distinction avec les listes chaînées, une structure de donnée radicalement différente des `<list>` de Python (c'est l'objet du Chapitre 8). Les tableaux de Python ne sont pas non plus tout à fait des tableaux selon la définition habituelle, car leurs éléments ne sont pas stockés dans des emplacements *contigus en mémoire*. Seules les *adresses* des éléments du tableau sont rangées contiguement. Nous ne développerons pas plus précisément comment les `<list>` sont implémentées ici.

Indexer les positions à partir de 0 ?

Cela peut paraître incompréhensible que Python ait choisi de noter `T[0]` la *première* case de `T`, `T[1]` la *deuxième case* de `T`, et `T[i-1]` la *i-ème case*. Mais en réalité, c'est très logique : la case `T[i]` se trouve `i` cases à droite de la case `T[0]` : le `i` n'est pas à voir comme un numéro, mais plutôt un *décalage* par rapport à une référence, ici, 0. Une autre explication, peut-être plus convaincante, est proposée par la Figure 3.1

2 Opérations sur les tableaux

Voici une liste de quelques opérations fondamentales sur les tableaux.

- **Création.** Pour créer un tableau, on peut placer une ou plusieurs expressions entre crochets, séparées par une virgule. Chaque case du tableau contient alors l'évaluation de l'expression correspondante.

```
>>> T = [3+4,3.1415,"Vive "+"Thonny!",None,4%2 == 0]
>>> T
[7,3.1415,'Vive Thonny!',None,True]
```

FIG. 3.1 – Pourquoi appeler `T[i]` la i-ème case du tableau quand on peut l'appeler `T[i-1]` ?!

On peut également créer un tableau vide à N cases avec la syntaxe suivante :

```
>>> N = 10
>>> T = [None]*N
>>> T
[None,None,None,None,None,None,None,None,None,None]
```

Plus généralement, `T*N` produit un tableau obtenu en répétant `T` N fois. Ainsi

```
>>> N = 3
>>> T = [1,2]*N
>>> T
[1,2,1,2,1,2,1,2]
```

- **Longueur du tableau.** La commande `len` (pour "length" – longueur) donne la longueur du tableau, c'est-à-dire, son nombre de cases.

```
>>> T = [1,2,3]*2
>>> T
[1,2,3,1,2,3]
>>> len(T)
6
```

- **Accès aux éléments.** On accède à l'élément en position `i` avec la syntaxe `T[i]`. Ceci produit une erreur lorsque `i` dépasse la taille du tableau (donc quand `i` $\geq$ `len(T)`)

```
>>> T = ["a","b",None]
>>> T[0]
'a'
>>> T[1]
'b'
>>> T[2] # Case vide : rien à afficher
```

```
>>> T[3] # Cette case n'existe pas -> erreur !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
 IndexError: list index out of range
```

On peut aussi accéder aux éléments depuis la fin du tableau. Par exemple, le dernier élément (premier en partant de la fin) s'obtient avec `T[-1]`, l'avant-dernier (deuxième en partant de la fin) avec `T[-2]`, etc. Ainsi, `T[-i]` est équivalent à `T[N-i]`, où `N` est la longueur du tableau.

- **Modification du contenu d'une case.** On peut mettre une nouvelle valeur dans la case en position `i` avec la syntaxe `T[i] = <nouvelle valeur>`, un peu comme pour une assignation. On peut donc voir `T[i]` comme une variable.

```
>>> T = ["a","b",None]
>>> T[2] = "c"
>>> T
['a','b','c']
```

- **Ajout d'un élément.** La commande `append` permet de créer une nouvelle case à la fin du tableau et d'y ajouter un élément. La syntaxe est la suivante :

```
>>> T = ["bonjour",3.1415,True]
>>> T
['bonjour',3.1415,True]
>>> T.append(1000)
>>> T
['bonjour',3.1415,True,1000]
```

- **Retrait de la dernière case.** Il est également possible de retirer la dernière case d'un tableau non vide. Pour cela, on utilise la fonction `pop`, dont la syntaxe est la suivante

```
>>> T = ["bonjour",3.1415,True,1000]
>>> T.pop()
1000
>>> T
["bonjour",3.1415,True]
>>> a = T.pop()
>>> a
True
>>> T
["bonjour",3.1415]
```

Les parenthèses vides sont importantes. Notez que `pop()` renvoie aussi le contenu de la case retirée (que l'on peut récupérer avec une variable si besoin), en plus de raccourcir le tableau.

- **Slicing.** Il est possible d'extraire une partie d'un tableau grâce à l'opération suivante, appelée "slicing" ("slice" signifie "part", comme dans "découper une part de cake"). En voici quelques exemples

```
>>> T = ['a','b','c','d','e']
>>> T[0:3] # Cases entre 0 (inclus) à 3 (exclus)
['a','b','c']
>>> T[2:5] # Cases entre 2 (inclus) et 5 (exclus)
['c','d','e']
>>> T[1:-1] # Cases entre 1 (inclus) et -1 -> la dernière (exclue)
['b','c','d']
>>> T2 = T[1:5] # -> T2 = ['b','c']
>>> T2[0]
'b'
>>> T2[0] == T[1]
True
```

Plus généralement, si `T` est un tableau et `i` et `j` deux entiers (positifs où négatifs), l'expression `T[i:j]` a pour valeur le nouveau tableau `[T[i] , T[i+1] , ..., T[j-1]]` (notez que, comme pour les "range" dans une boucle `for`, le premier indice est inclus, et le dernier est exclus, cf. Figure 3.1). Autrement dit, `T[i:j]` est le "sous-tableau" obtenu en ne gardant que les cases `i` à `j-1`. On peut aussi écrire `T[:j]` (équivalent à `T[0:j]`) ou encore `T[i:]` (équivalent à `T[i:N]` où `N` est la taille du tableau). La syntaxe `T[i:]` est très pratique pour aller jusqu'à la fin du tableau.

Slicing pour des `<str>`

Le slicing peut aussi être utilisé, avec la même syntaxe, pour extraire des parties d'une chaîne de caractères. Par exemple, si la variable `message` a pour valeur `"Hello world!"`, les expressions `message[0:5]`, `message[6:-1]` et `message[6:]` ont pour valeur `"Hello"`, `"world"`, et `world!`, respectivement.

3 Parcourir les éléments

Imaginez que vous souhaitez faire l'inventaire de tous les objets qui sont rangés dans un tableau, c'est-à-dire les afficher un par un dans la console. Bien sûr, si c'est un petit tableau, vous pouvez simplement afficher `T[0]`, puis `T[1]`, etc. jusqu'à la fin du tableau. Mais si le tableau est un peu trop long, ceci devient vite pénible. Recopiez et exécutez le code suivant, qui utilise une méthode bien plus efficace :

```
T = [1,2,3]*50 # Crée un tableau de taille 150.
N = len(T)
print("Début de l'inventaire :")
for i in range(N):
    print(T[i])
print("Terminé !")
```

Grâce à la boucle `for`, il suffit d'écrire l'instruction `print` une seule fois ! Lorsque ce code est exécuté, conformément à la définition d'une boucle `for`, Python commence par exécuter l'instruction `print(T[i])` pour `i = 0` (ce qui aura donc pour effet d'afficher `T[0]`), puis de nouveau pour `i = 1`, et ainsi de suite, jusqu'à `i = N-1` (rappelez vous, la dernière valeur du "`range`" est exclue...). Comme `T[N-1]` est le dernier élément du tableau, à la fin de la boucle tous les éléments auront bien été affichés. Cette méthode pour consulter chaque case du tableau grâce à une boucle `for` est fondamentale ! On appelle cela *parcourir le*

tableau. On peut aussi l'utiliser pour modifier les valeurs des éléments. Par exemple, le programme suivant remplace le contenu de chaque case par son carré :

```
T = [1,2,3,4,5]
N = len(T)
print("Avant :",T)
for i in range(1,N):
    T[i] = T[i]**2
print("Après :",T)
```

```
>>>
 Avant : [1,2,3,4,5]
 Après : [1,4,9,16,25]
```

La boucle `for` peut aussi être utilisée tout simplement pour créer un tableau, et pour beaucoup d'autres choses : les possibilités sont infinies, et chaque problème auquel vous serez confronté demandera une solution un peu différente à partir des mêmes ingrédients.

Question 3. Comment créer un tableau de longueur 1000 contenant les nombres de 1 à 1000 ?

4 Un comportement inattendu

Il existe une différence surprenante entre le comportement des tableaux et celui des autres types variables vues jusqu'ici. En effet, comparez ces deux programmes :

Programme 1 :

```
a = 10
b = a
b = b+1
print("a = ",a)
print("b = ",b)
```

```
>>>
a = 10
b = 11
```

Programme 2 :

```
T = ["Jacques","doit",10,"euros","à","Jeanne"]
U = T
U[2] = U[2]+1
print("T = ",T)
print("U = ",U)
```

```
>>>
T = ['Jacques','doit',11,'euros','à','Jeanne']
U = ['Jacques','doit',11,'euros','à','Jeanne']
```

Dans le premier programme, la modification de `b` n'a pas d'effet sur `a`, tandis que dans le second, la modification de `U` en a un sur `T`. Il est capital de comprendre cette différence, sinon, vous perdrez vite le contrôle de vos probrammes. Le but de cette section est d'expliquer la différence entre ces deux comportements : ils sont dûs à l'intéraction entre deux concepts

(i) la *mutabilité* (les tableaux peuvent subir des modifications, ou "mutations"),

(ii) les *alias* (les assignations du type `a = b` ne se passent pas exactement comme on le croit).

Les tableaux sont "mutables"

Vous avez peut-être remarqué la ressemblance entre un tableau et une chaîne de caractères. Comme pour les tableaux, on peut accéder aux éléments des chaînes de caractère avec la syntaxe `message[i]` et utiliser le slicing. Il existe toutefois une différence très importante :

```
>>> message = "Hallo world"
>>> message[1]  # Accès à l'élément : OK
'a'
>>> message[1] = "e" # Modification de l'élément : Pas question !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Python vous laisse *consulter* le contenu d'une chaîne de caractère autant que vous voulez, mais pas le *modifier*. Au contraire, pour un tableau, la modification ne pose aucun souci :

```
>>> T = ["H","a","l","l","o"," ","w","o","r","l","d"]
>>> T[1]
'a'
>>> T[1] = "e"
>>> T
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

Les objets que Python accepte de modifier après leur création sont dits *mutables* : on peut leur faire subir des "mutations". On parle de mutation lorsque c'est directement l'objet en lui-même qui est modifié, sans réassignation de la variable correspondante. En reprenant la métaphore de l'hôtel, imaginez le client (l'objet) qui, sans prévenir, se "transforme" pendant la nuit. Il ne change pas de chambre, et le registre n'est pas modifié. Mais la variable qui pointe vers lui a une nouvelle valeur : le client transformé.

Les tableaux sont le premier type mutable que nous rencontrons dans ce cours. Les mutations qu'ils peuvent subir sont par exemple la modification d'un élément, l'ajout ou la suppression de cases, etc. Au contraire, tous les types vus précédemment (`<int>`, `<float>`, `<str>`, `<bool>`) sont *immutables*. Si une variable est assignée à une valeur immutable, la seule façon de modifier sa valeur est de la réassigner.

Création d'un alias

Dans le Chapitre 1, nous avons vu que les étapes de l'assignation étaient les suivantes :

Assignation "normale"

1. évaluer l'expression à droite du signe égal
2. créer un *nouvel* objet avec la valeur obtenue
3. le ranger dans la mémoire à une *nouvelle* adresse inoccupée.

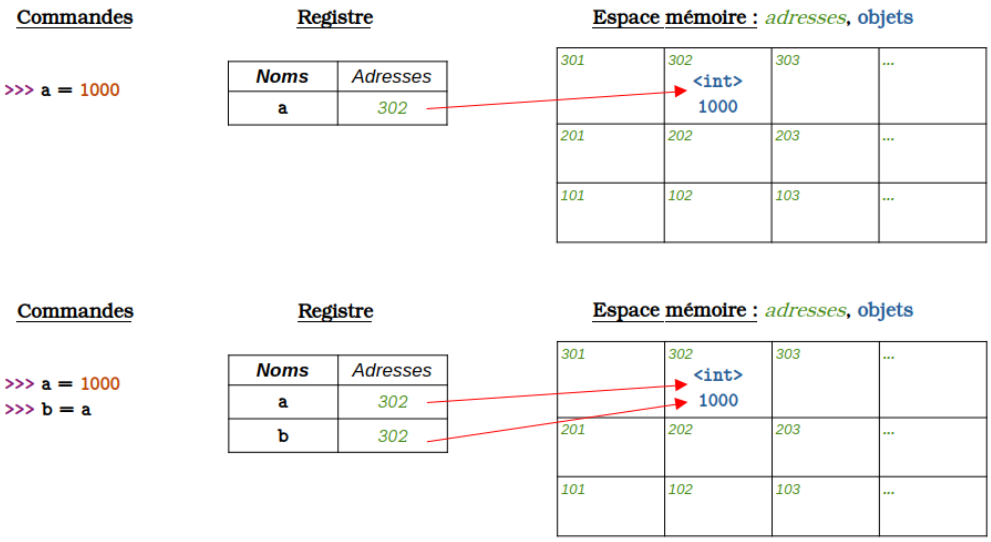


FIG. 3.2 – Création d'un alias

4. ajouter au registre la ligne "<var>, <adr>", où <var> est le nom de la variable et <adr> l'adresse mémoire de l'étape 3.

Mais dans le cas particulier d'une assignation comme `b = a` ou `U = T`, c'est-à-dire, quand l'expression à droite du signe égal est réduite à une seule variable [1], Python procède complètement différemment :

Assignation de la forme `b = a`

1. trouver dans le registre l'adresse associée à la variable `a`
2. ajouter au registre la ligne "`b`, `<adr_a>`", où `<adr_a>` est l'adresse mémoire de l'étape 1.

Ainsi, après ces instructions, la variable `b` "pointe" vers le même emplacement en mémoire que `a`, donc a pour valeur le même objet (voir Figure 3.2). Autrement dit, Python évite de dupliquer cet objet. Il utilise un seul et même objet comme valeur commune pour deux variables différentes. Les noms `a` et `b` sont deux étiquettes, ou deux *alias*, pour se référer à un seul et même "client de l'hôtel". De même, après les instructions

```
>>> a = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] # Assignation normale
>>> b = a # Alias
>>> c = b # Alias
>>> d = c # Alias
```

un seul tableau (de longueur 16) est créé en mémoire, et les quatre variables `a,b,c,d` se réfèrent toutes à ce même tableau. L'intérêt, bien sûr, est d'éviter de gaspiller de l'espace mémoire en recopiant quatre fois le même tableau !

Alias et cases d'un tableau

La création d'alias s'applique aussi pour les assignations de la forme `a = T[i]`, ou `T[i] = a`, c'est-à-dire, si l'expression à gauche et/ou à droite du signe égal n'est pas une variable, mais une case d'un tableau. Ici aussi, `a` et `T[i]` seront deux alias d'un même objet. En fait, il est utile de considérer `T[i]` comme une variable à part entière.

[1] ou, plus généralement, quand elle est elle-même assignable

Mutabilité + alias = danger

Revenons à présent aux **Programme 1** et **Programme 2** du début de cette section. Pour le premier :

```
a = 1000 # Assignation (normale) à un objet <int> 1000 (immutable)
b = a # Création d'un alias.
b = b+1 # Réassignation de b à un *nouvel* objet <int> 1001
```

même si la deuxième ligne crée un alias, l'objet commun `<int> 1000` est immutable. Il est en quelque sorte "protégé", en "lecture seule". La variable `a` peut donc dormir sur ses deux oreilles, sa valeur ne peut tout simplement pas être modifiée par qui que ce soit. L'assignation `b = b+1` a pour effet de créer un autre objet. Après cette instruction, `b` n'est plus un alias de `a`. La situation est différente dans le **Programme 2** :

```
T = ["Jacques","doit",10,"euros","à","Jeanne"] # Assignation (normale) à un tableau
#(mutable !)
U = T # Création d'un alias. T et U peuvent tous deux modifier l'objet commun
U[2] = U[2]+1 # Attention !!! Mutation. L'objet commun a été modifié.
```

Cette fois, `T` et `U` contrôlent tous les deux un même objet qu'ils peuvent tous deux modifier (comme une ardoise de dépenses partagée). Toute modification faite par `U` a un effet sur l'objet commun, donc sur la valeur de `T`. Réciproquement, `T` aussi peut modifier l'objet commun, et donc la valeur de `U`. Essayez par exemple

```
T = [] # Tableau vide
U = T # Alias.
T.append("c'est T le plus beau !")
U.append("c'est U le plus fort !")
print("T = ",T)
print("U = ",U)
```

Exercices du Chapitre 3

Conseil : Pour ces exercices, créez un dossier (par exemple `TP_Tableaux`) et enregistrez vos réponses aux questions dans un fichier `.py` par question. Alternativement, on peut organiser le programme à l'aide de fonctions (voir le Chapitre 4).

Tableaux : manipulations de base

1* Créez un programme `carres1a100.py` qui crée un tableau de longueur 100 contenant en position i le nombre i^2 (avec la première position $i = 0$).

2* Créez un fichier `moyenne.py` avec le programme suivant

```
T = [15,12,16.5,9,14]
# ... à vous de jouer ...
print("Moyenne des éléments de T : ",moyenne)
```

Le but est d'afficher la moyenne des éléments de `T`. Il faut que le programme fonctionne même en modifiant le tableau `T` défini à la première ligne (sans rien changer au reste du programme).

3* Même question avec un programme `max_tableau.py` qui affiche le plus grand élément de `T` au lieu de la moyenne.

4. Complétez le programme `pairs_seulement.py` suivant

```
T = [5,4,7,9,8]
T2 = []
# ... à vous de jouer ...
print("Entiers pairs : ",T2)
```

Le but est d'afficher un tableau qui contient les éléments pairs de `T`. Comme précédemment, le programme doit fonctionner même si on modifie le tableau `T` dans la première ligne.

5. Dans cet exercice et le suivant, on demande de ne pas utiliser la fonction `+` sur les tableaux (sinon, l'exercice devient un peu facile !). Complétez le programme `concat.py` suivant

```
T1 = [1,2,3,4]
T2 = [5,6]
# ... à vous de jouer ...
print("Tableau concaténé : ",T)
```

Le but est d'afficher un tableau contenant les éléments de `T1` et `T2` à la suite l'un de l'autre (donc ici, `[1,2,3,4,5,6]`). Le programme doit aussi fonctionner en changeant les tableaux `T1` et `T2` dans les deux premières lignes.

6. Même question que ci-dessus, mais sans utiliser la fonction `append`. [2]

[2] En fait, il existe une commande Python dédiée pour concaténer deux tableaux : `T1 + T2`. On pourra utiliser cette commande par la suite.

7* Complétez le programme `supprimer_case.py` suivant

```
T = [3,1,4,1,5]
i0 = 3
# ... à vous de jouer ...
print(T)
```

Le but est de retirer la case `T[i0]` du tableau `T` (ici, il faut donc que le programme affiche `[3,1,4,5]`). Dans cette question, le tableau T ne doit pas être réassigné : seules les mutations sont autorisées. *Indice : utilisez la fonction* `pop()` *et "jouez aux chaises musicales" avec les éléments du tableau.*

8* Complétez le programme `inserer.py` suivant

```
T = [3,1,4,5]
a = 1 #  élément à insérer
i0 = 3 # Position de l'élément à insérer
# ... à vous de jouer ...
print(T)
```

Le but est d'insérer l'élément `a` dans le tableau, de sorte qu'il se retrouve dans la case `T[i0]`. Comme dans la question précédente, le tableau `T` peut être muté, mais pas réassigné. *Indice : utilisez une méthode analogue à l'exercice précédent.*

9* Complétez le programme `copie_tableau.py` suivant.

```
T = ["a",3.14,1000,True]
# ... à vous de jouer ...
print("T = ",T)
print("Tcopie = ",Tcopie)
# T et Tcopie doivent être identiques
Tcopie.append(0) # Mutation de Tcopie
print("Tcopie =",Tcopie)
print("T = ",T)
# T ne doit pas avoir changé de valeur.
```

Le but est de créer une copie qui ne soit pas un alias de `T`. Ainsi, la mutation de `Tcopie` ne doit pas modifier la valeur de `T`.

Tri

Trier un tableau est un problème fondamental en programmation. Nous verrons plusieurs méthodes ici et dans les prochains chapitres. Nous commençons ici par un méthode très naturelle : le *tri par insertion*, qui est celui que nous utilisons par exemple pour trier un paquet de carte entre nos mains.

10. Dans un sous-dossier `Tri`, créez un programme `insertion.py` comme ceci :

```
U = [1,3,7,9,10] # Un tableau *déjà trié*
a = 8 # élément à insérer
# ... à vous de jouer ...
print(U)
```

En supposant que `U` est déjà trié (par ordre croissant), le but d'insérer le nouvel élément `a` dans `U` à la bonne place, de sorte qu'à la fin du programme, `U` soit encore trié. Le tableau `U` pourra être muté, mais pas réassigné. On pourra s'inspirer du programme `inserer.py` vu dans un exercice précédent.

11. Créez un programme `tri.py` comme ceci

```
T = [3,9,7,10,1,8] # Tableau à trier
```

qui reprend le programme précédent en l'entourant d'une boucle `for`. L'idée est d'insérer, un par un, les éléments de `T` dans un tableau `U` initialement vide.

12. Créez un programme `mediane.py` qui calcule la médiane d'un tableau de valeurs numériques. On pourra commencer par trier ce tableau.

13. (Question bonus) Pouvez-vous modifier votre tri par insertion de manière à ce qu'il n'utilise que des mutations du tableau `T`, sans créer un nouveau tableau `U` ?

Mini-projet : résolution de Sudoku

Pour finir cette feuille d'exercice, nous proposons un exercice un peu plus long qui pourra être votre premier mini-projet de programmation. Nous utiliserons des fonctions Python (sans quoi, l'organisation du programme devient un peu fastidieuse). Il est donc nécessaire de lire le Chapitre 4 avant de continuer.

Un sudoku est une grille de 3×3 cellules carrées, chacune décomposée en 3×3 cases. Chaque case peut contenir un chiffre entre 1 et 9 ou être vide (on utilisera le chiffre `0` dans ce cas). Un sudoku est *invalide* si un chiffre (autre que 0) est répété au sein d'une ligne, colonne, ou cellule, et *impossible* s'il n'y a aucun moyen de compléter toutes les cases vides sans le rendre invalide. On représente un Sudoku par un tableau `T` de longueur 81, où les neuf premières cases représentent la première ligne du Sudoku, puis les 9 suivantes la deuxième ligne, et ainsi de suite. On note (i, j) la case qui se trouve à la ligne i et la colonne j, $1 \leq i, j \leq 9$.

11. Créez une fonction `sudoku_vide()` qui renvoie un tableau `T` représentant le sudoku vide.

12. Créez une fonction `afficher_sudoku(T)` qui affiche dans la console le sudoku représenté par le tableau `T`. Voici par exemple un affichage possible pour le Sudoku vide :

```
    1   2   3    4   5   6    7   8   9
  ||-----------||-----------||-----------||
1 ||   |   |   ||   |   |   ||   |   |   ||
2 ||   |   |   ||   |   |   ||   |   |   ||
3 ||   |   |   ||   |   |   ||   |   |   ||
  ||-----------||-----------||-----------||
4 ||   |   |   ||   |   |   ||   |   |   ||
5 ||   |   |   ||   |   |   ||   |   |   ||
6 ||   |   |   ||   |   |   ||   |   |   ||
  ||-----------||-----------||-----------||
7 ||   |   |   ||   |   |   ||   |   |   ||
8 ||   |   |   ||   |   |   ||   |   |   ||
9 ||   |   |   ||   |   |   ||   |   |   ||
  ||-----------||-----------||-----------||
```

13. On peut aussi représenter un Sudoku par un code de 81 chiffres entre 0 et 9, avec 0 correspondant aux cases vides. Créez une fonction `importer_sudoku(s)` qui crée le Sudoku représenté par la chaîne de caractères `s`. Affichez de cette manière les sudokus donnés à la fin de cette feuille d'exercices.

14. Vérifiez que la case (i, j) du Sudoku correspondant à la case $n = 9(i-1) + j - 1$ du tableau `T`. Réciproquement, vérifier que la case n du tableau correspond à la case $(q + 1, r + 1)$ du Sudoku où q et r sont le quotient et le reste de la division euclidienne de n par 9.

15. Si l'on numérote les cellules de 1 à 9 de gauche à droite et de haut en bas, vérifier que la case (i, j) du sudoku se trouve dans la cellule $k = 3I + J + 1$ où I et J sont les quotients de la division euclidienne de $i-1$ et $j-1$ par 3.

16. Créez une fonction `contenu(T,i,j)` qui renvoie le contenu de la case (i, j) du Sudoku représenté par `T`, et une fonction `ecrire(T,i,j,c)` qui écrit le chiffre `c` dans cette case.

17. Créez une fonction `coordonnées(n)` qui prend en entrée un numéro de case du tableau `T` et renvoie un tableau `[i,j,k]` tels que `T[n]` se trouve dans la ligne i, la colonne j et la cellule k du Sudoku.

```
>>> coordonnées(0)
[1,1,1]
>>> coordonnées(7)
[1,2,3]
```

18. Créez une fonction `indices_cellule(k)` qui renvoie un tableau contenant les indices n des neufs cases contenues dans la cellule k. Par exemple

```
>>> indices_cellule(1)
[0,1,2,9,10,11,18,19,20]
```

19. De même, créez des fonctions

    `indices_ligne(i)` / `indices_colonne(j)`.

    renvoyant les coordonnées des cases de la ligne i/de la colonne j.

20. Créez une fonction `possibilites(T,i,j)` qui renvoie un tableau contenant tous les chiffres qui peuvent être écrits dans la case (i, j) du Sudoku sans le rendre invalide.

21. Créez une fonction `verifier(T)` qui affiche un message disant si le sudoku est valide ou non, et s'il est entièrement résolu ou non, en indiquant le nombre de cases manquantes.

22. Créez une fonction `auto_completer(T,v)` qui parcourt chaque case (i, j), et, s'il n'y a qu'une seule valeur possible dans cette case, y écrit le chiffre correspondant. La fonction renverra un entier égal au nombre de nouveaux chiffres écrits dans `T`. La variable `v` est un entier qui contrôle la "verbosité" de la fonction. Si `v` vaut 0, la fonction n'affiche rien. Si `v` vaut 1 ou plus, on affichera des informations, comme par exemple les coordonnées de chaque chiffre qui a été trouvé. On renverra une erreur si on s'aperçoit que le Sudoku est impossible.

23. Créez une fonction `resoudre(T,v)` qui tente de résoudre un Sudoku à l'aide de cette technique, et testez-la avec les Sudokus de la page suivante. Commentez.

24. Dans une cellule, il arrive qu'un chiffre ne puisse être placé que dans une seule case. Créez une fonction

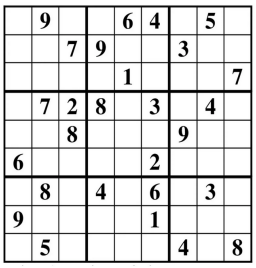
    `chercher_cellule(T,k,v)`

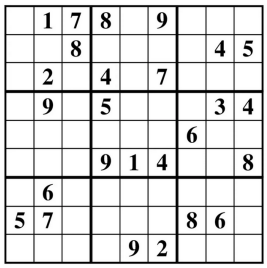
    qui applique cette idée. Ecrire des fonctions analogues pour les lignes et les colonnes.

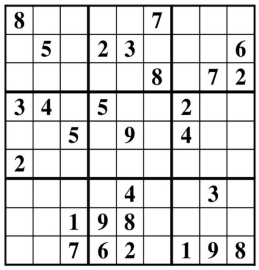
25. Améliorez la fonction `resoudre`, et appliquez-là au Sudoku numéro 5.

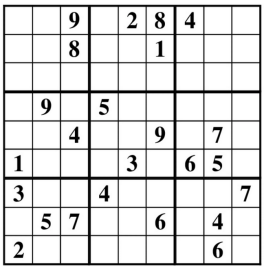
26. Imaginez et décrivez en quelques lignes une approche possible pour améliorer votre algorithme.

Quelques Sudokus faciles...





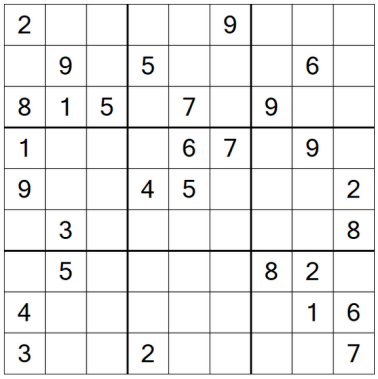




Importez-les avec les codes ci-dessous

1. 090064050007900300000010007072803040008000900600002000080406030900001000050000408
2. 017809000008000045020407000090500034000000600000914008060000000570000860000092000
3. 800007000050230006000008072340500200005090400200000000000040030001980000007620198
4. 009028400008001000000000000090500000004009070100030650300400007057006040200000060

Un Sudoku plus résistant...



Son code est

5. 200009000090500060815070900100067090900450002030000008050000820400000016300200007

Chapitre 4

Fonctions

> *"En permettant au mécanisme de combiner ensemble des symboles généraux dans des successions d'une variété et d'une étendue illimitées, un lien unificateur est établi entre les opérations de la matière et les processus mentaux abstraits de la branche la plus abstraite des sciences mathématiques."*
>
> Ada Lovelace, Note A, traduction de "Sketch of the Analytical Engine" (1843).

Dans presque tous les langages de programmation, il est possible de donner un nom à un morceau de programme, de manière à pouvoir le réutiliser plus tard sans avoir à le recopier entièrement. Pour cela, on utilise des *fonctions*. Les fonctions peuvent recevoir des entrées (aussi appelés *arguments*) et renvoyer des sorties qui dépendent de ces entrées (voir Figure 4.1).

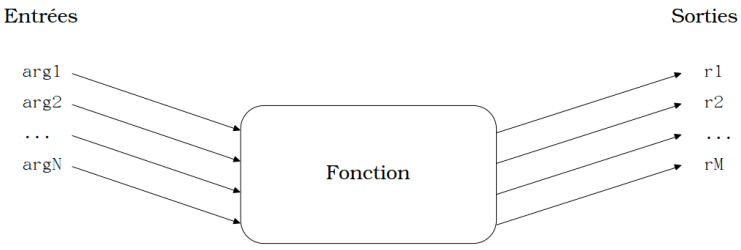


FIG. 4.1 – Représentation schématique d'une fonction.

Par exemple, nous avons rencontré la fonction `len` dans le Chapitre 3 : elle reçoit un tableau en entrée, et renvoie sa longueur en sortie. Dans ce chapitre, nous allons apprendre à définir nos propres fonctions en Python, et voir ce qu'il se passe lorsqu'elles sont exécutées.

1 Fonctions et algorithmes

Commençons par une distinction importante entre *fonction* et *algorithme*.

Définition 4.1 : Algorithme

Un *algorithme* est une suite finie d'instruction, choisies parmi un nombre fini d'*instructions élémentaires* (opérations arithmétiques, comparaisons, assignations, instructions conditionnelles, etc.). L'algorithme est exécuté à partir d'une ou plusieurs données en entrée, et produit un ou plusieurs résultats en sortie.

On peut comparer un algorithme à une recette de cuisine : les données en entrées étant les quantités des ingrédients, ou le nombre d'invités, et le résultat en sortie étant le plat à déguster. Un exemple important-d'algorithme est celui qui permet de calculer la somme, la soustraction, la multiplication et la division de deux nombres entiers avec un crayon et un papier.[1] Les programmes Python que vous avez écrits sont des algorithmes (par exemple celui qui permet de calculer la représentation binaire d'un nombre).

Comme un automate

Pour exécuter un algorithme, il n'y a pas besoin de le comprendre, ni de prendre la moindre initiative : il suffit de suivre scrupuleusement les instructions, une par une. Si vous avez posé suffisamment de multiplications, vous n'avez plus besoin de réfléchir : vous pouvez suivre la méthode comme un automate. L'idée géniale qui mena à l'invention de l'ordinateur, ce fut justement de voir que ces actions étaient tellement automatiques que *même une machine* pourrait les réaliser !

En général, un algorithme a un but : on cherche à obtenir des sorties qui vérifient une certaine relation par rapport aux entrées. On dit que l'algorithme réalise, ou "calcule" une *fonction* :

Définition 4.2 : Fonction

Une *fonction* f est une relation bien définie entre des entrées $e_1, \ldots, e_n$ et des sorties $s_1, \ldots, s_m$, de sorte que la valeur des sorties soit définie de manière unique par la valeur des entrées. On note alors

$$s_1, \ldots, s_m = f(e_1, \ldots, e_m).$$

Exemples de fonctions :
- La fonction *reste* prend en entrée deux arguments, des nombres a et b, et renvoie en sortie le reste de la division Euclidienne et a par b. Ainsi, $reste(15, 7) = 1$ et $reste(7, 15) = 7$.
- La fonction $PGCD$ prend en entrée deux entiers a et b, et retourne en sortie le plus petit grand entier d qui divise à la fois a et b. Par exemple, $PGCD(35, 49) = 7$.
- La fonction *Détecteur-de-mensonge* prend en entrée une proposition, et renvoie en sortie un booléen, qui est la valeur de vérité de la proposition.
- La fonction *Diophante*, prend en argument un polynôme $P(x_1, \ldots, x_n)$ à coefficients entiers en n variables et renvoie *Vrai* s'il existe des entiers $x_1, \ldots, x_n$ tels que $P(x_1, \ldots, x_n) = 0$, et *Faux* sinon.

Dans ces quatre exemples, nous définissons la fonction sans donner d'algorithme qui la calcule. C'est ce qui s'appelle une *spécification*. Même si la spécification d'une fonction est parfaitement claire, il n'est pas forcément facile de trouver un algorithme correspondant. Dans certains cas, on n'est même pas sûr qu'un tel algorithme existe, comme dans les deux derniers exemples ci-dessus (certaines fonctions, comme *Diophante*, ne sont pas "calculables" ![2]) D'autre part, il n'est pas toujours évident de vérifier qu'un algorithme remplit bien la fonction spécifiée : ceci peut nécessiter une démonstration (voir Chapitre 5).

Remarque : *En programmation, pour définir une fonction, on est obligé de fournir un algorithme pour la calculer. Il est donc fréquent d'utiliser le mot fonction pour parler à la fois de la fonction et de l'algorithme.*

[1] Cet exemple est à l'origine même du mot algorithme : l'étymologie remonte au mathématicien persan al-Khwarizmi, qui, autour de l'an 825 de notre ère, a écrit *kitab al-hisab al-hindi* ("livre du calcul indien") et *kitab al-jam' wa'l tafriq al-hisab al-hindi* ("addition et soustraction en arithmétique indienne"). Ces livres ont été traduit quelques siècles plus tard en latin, avec le nom d'al-Khwarizmi latinisé en "Algorizmi".

[2] Le fait que *Diophante* soit ou non calculable est l'un des 21 problèmes posés par David Hilbert en 1900. La réponse négative a été démontrée par Youri Matiiassevitch en 1970.

2 Exécution d'une fonction

On considère la fonction *PGCD*, dont la définition est la suivante :

> Spécification : PGCD
> *Entrée(s)* : a, b, deux nombres entiers.
> *Sortie(s)* : le plus grand nombre entier d qui divise à la fois a et b.

Il existe un algorithme célèbre pour calculer cette fonction : l'algorithme d'Euclide, donc voici la définition.

> Définition : `Euclide`
> *Entrées :* a,b, deux nombres entiers.
>
> 1. Assigner $r \leftarrow$ reste de la division Euclidienne de a par b
> 2. Tant que $r > 0$,
>   (a) Assigner $a \leftarrow b$
>   (b) Assigner $b \leftarrow r$
>   (c) Assigner $r \leftarrow$ reste de la division Euclidienne de a par b
> 3. Terminer.
>
> *Sorties :* la valeur finale de b.
> Fin de la définition de `Euclide`.

Validité de l'algorithme d'Euclide ?

Si vous ne voyez pas pourquoi l'algorithme d'Euclide calcule le PGCD, c'est parfaitement normal : cela ne saute pas aux yeux ! Il est généralement difficile, simplement en lisant un programme, de comprendre à quoi il sert ou pourquoi il fonctionne. Pour cela, on a parfois besoin d'une démonstration. Dans le cas de l'algorithme d'Euclide, celle-ci sera donnée au Chapitre 5. Pour l'instant, cette question n'est pas importante : ce qui compte pour l'instant, c'est d'apprendre comment exécuter "bêtement" l'algorithme.

Nous allons exécuter `Euclide` avec les arguments 28 et 36. Tel un véritable "ordinateur humain" (et comme d'autres l'ont fait avant vous, voir la Figure 4.2) munissez-vous d'un papier et un crayon, et effectuez vous-mêmes les calculs qui suivent au fur et à mesure. En vous mettant ainsi "dans la peau de l'ordinateur", le but est que vous compreniez exactement ce qui se passe lorsqu'il exécute les instructions d'un programme. C'est le minimum pour pouvoir ensuite être vous-même capables de mettre au point la bonne suite d'instruction pour atteindre un objectif que vous vous êtes fixé.[3] Le plus important ici est de bien comprendre les étapes d'appel, entrée et sortie de la fonction.

- *Avant l'algorithme.* Supposons que nous étions déjà en train de faire des calculs. Nous avons une "mémoire" qui contient des variables. Nous utiliserons une feuille pour matérialiser cette mémoire. Disons par exemple qu'elle contient `a` : `1002`, `m:"Hello world"` et `pi` : `3.1415`. Nous allons représenter à chaque étape l'état actuel de notre mémoire comme ceci :
  Mémoire : `a:1002`, `m:"Hello world"`, `pi:3.1415`.
- *Appel de la fonction.* Supposons que nous arrivons à l'instruction

$$d \leftarrow \texttt{Euclide}(28,36).$$

[3] *"Connais ton ennemi et connais-toi toi-même, tu vaincras cent fois sans péril."* Sun Tzu, *L'art de la guerre.*

FIG. 4.2 – Des employé(e)s – majoritairement des femmes... – effectuant des calculs en 1914, aidé(e)s par un "comptometer", une sorte de calculatrice mécanique, pour gagner du temps sur les opérations arithmétiques.

C'est une assignation. Donc comme Python, nous dévons d'abord évaluer l'expression `Euclide(28,36)` qui se trouve à droite de la flèche. Cette expression est une *séquence d'appel* : la fonction `Euclide` a été "appelée" sur les entrées 28 et 36. Ceci nous signale qu'il faut entrer dans l'algorithme nommé "`Euclide`" avec les arguments d'entrée 28 et 36.
Mémoire : `a:1002`, `m:"Hello world"`, `pi:3.1415`.

- *Entrée dans l'algorithme.* Pour entrer dans un algorithme, on prend une nouvelle feuille vierge pour représenter notre mémoire dans cet algorithme. Pendant toute l'exécution de `Euclide`, nous plaçons cette nouvelle feuille *par-dessus* la feuille mémoire précédente contenant les variables `a`, `m` et `pi`. Entrer dans l'algorithme, c'est comme entrer dans un nouvel "univers", et la feuille blanche symbolise notre mémoire dans ce nouvel univers. Tout ce à quoi nous avons accès dans ce nouvel univers, ce sont les arguments d'entrée, ici 28 et 36. Pour prendre ces-derniers en compte, nous commençons par faire $a \leftarrow 28$ et $b \leftarrow 36$.
Mémoire : `a:28, b:36`
- *Instruction 1.* La première instruction de `Euclide` nous demande d'assigner le reste de la division de `a` par `b` (ici, 28), à une variable nommée `r`.
Mémoire : `a:28, b:36, r:28`
- *Instruction 2.* Comme nous l'avons vu au Chapitre 2, la seconde instruction est nous demande de vérifier si `r > 0` vaut *Vrai*. Si oui, nous devons exécuter les instructions (a)-(b)-(c) puis revenir à l'instruction 2, sinon, il faut aller directement à l'instruction 3. Puisque `r` vaut 28, nous entrons dans le bloc.
Mémoire : `a:28, b:36, r:28`
  - *Instruction (a).* On assigne la valeur de `b` (donc 36) à la variable `a`
  Mémoire : `a:36`, `b:36`, `r:28`.
  - *Instruction (b).* On assigne la valeur de `r` (donc 28) à b.
  Mémoire `a:36`, `b:28`, `r:28`.
  - *Instruction (c).* On assigne le reste de la division Euclidienne de `a` par `b` (donc 8) à r, puis on revient à l'instruction 2.
  Mémoire `a:36`, `b:28`, `r:8`.
- *Instruction 2.* Comme `r` vaut 8, la condition est vraie, donc nous entrons de nouveau dans le bloc.
Mémoire `a:36`, `b:28`, `r:8`.

    - *Instruction (a).* On assigne la valeur de `b` à `a`.
      Mémoire `a:28`, `b:28`, `r:8`
    - *Instruction (b).* On assigne la valeur de `r` à `b`.
      Mémoire `a:28`, `b:8`, `r:8`.
    - *Instruction (c).* On assigne le reste de la division de `a` par `b` (ici 4) à `r`, puis on revient à l'instruction 2.
      Mémoire `a:28`, `b:8`, `r:4`.
- *Instruction 2.* Comme `r` vaut 4, il faut encore entrer dans le bloc "tant que".
    - *Instruction (a).* On assigne la valeur de `b` à `a`.
      Mémoire `a:8`, `b:8`, `r:4`.
    - *Instruction (b).* On assigne la valeur de `r` à `b`.
      Mémoire `a:8`, `b:4`, `r:4`.
    - *Instruction (c).* On assigne le reste de la division de `a` par `b` (ici 0) à `r`, puis on revient à l'instruction 2.
      Mémoire `a:8`, `b:4`, `r:0`.
- *Instruction 2.* D'après notre feuille, $(r > 0)$ vaut *Faux* (enfin !). Nous allons à l'instruction 3.
  Mémoire `a:8`, `b:4`, `r:0`.
- *Instruction 3.* Nous terminons l'exécution de la fonction en renvoyant la valeur de b, qui est 4.
  Mémoire `a:8`, `b:4`, `r:0`.
- *Sortie de l'algorithme.* Nous avons fini le calcul. Nous ressortons de "l'univers" de l'algorithme. Nous jetons définitivement la feuille contenant nos calculs intermédiaires et retrouvons en-dessous la feuille mémoire que nous avions recouverte. Les variables b et r n'existent donc plus, et la variable a a retrouvé sa valeur précédente, 1002. Tout ce que nous avons conservé de l'univers de la fonction, c'est le résultat de sortie, 4. Nous étions en train de lire l'instruction $d \leftarrow$ `Euclide(28,36)`, donc nous assignons 4 à la variable d.
  Mémoire `a:1002`, `m:"Hello world"`, `pi:3.1415`, `d:4`.

Le résultat obtenu est 4, qui est en effet le PGCD de 28 et 36. Le lecteur est invité à exéctuer de la même manière $a \leftarrow$ `Euclide`(231,154).

Appeler une fonction/passer des arguments

En programmation, on dit souvent que l'on "appelle" une fonction sur des arguments d'entrée. On dit aussi que les arguments sont "passés" à la fonction, et qu'elle "renvoie" des sorties. Par exemple, pour savoir quel est le PGCD de 231 et 154, on appelle `Euclide` sur les arguments 231 et 154. La syntaxe habituelle pour appeler la fonction est d'écrire le nom de cette fonction suivi des arguments entre parenthèse. Dans notre exemple, cela donne donc : `Euclide(231,154)`. Attention : définir une fonction et l'appeler sont deux choses différentes, ce que l'on peut avoir tendance à oublier lorsqu'on débute. *Définir* une fonction, c'est un peu comme écrire une recette, et *appeler* cette fonction, c'est demander à Python de se mettre à la cuisiner – avec les quantités que vous aurez passées en argument.

3 Les fonctions en Python

Exemples

Le programme suivant contient la traduction de l'algorithme d'Euclide en langue Python :

```
def Euclide(a,b):
    # Renvoie le PGCD de a et b
    r = a % b
    while r > 0:
        a = b
        b = r
        r = a % b
    return b
```

```
>>> Euclide(28,36)
4
>>> d = Euclide(231,154)
>>> d
77
```

Notez qu'on a indiqué la spécification en commentaire : ceci est optionnel mais souvent bienvenu. Plus généralement, une définition de fonction a la syntaxe suivante en Python

```
def nom_de_ma_fonction(arg1,arg2,...,argN):
     # Bloc indenté
     # Ecrire des instructions avec arg1,arg2 etc.
     # Renvoi du résultat :
     return r1,r2,...,rM
# La définition s'arrête quand l'indentation revient à la verticale de "def"
```

Voici un exemple de définition de fonction et de son exécution par Python :

```
def isEven(N):
     # Renvoie True si N est pair, et False si N est impair.
     if N%2 == 0:
         return True
     else:
         return False

# Il faut appeler la fonction pour qu'elle soit exéctuée, sinon il ne se passe rien !
a = 4
b = 5
print(isEven(a)) # Appels de la fonction
print(isEven(b))
```

```
True
False
```

Cette fonction renvoie `True` si N divisé par 2 donne un reste 0 – donc si N est pair ("even" en anglais) – et `False` sinon.

Voici un exemple un peu plus sophistiqué, dans lequel on veut calculer la somme des chiffres d'un entier. Il faut s'y prendre d'une manière un peu astucieuse, en modifiant petit à petit l'entier sur lequel on fait les calculs. On retire le chiffre des unités, on l'ajoute à une variable contenant la somme temporaire des chiffres, et on recommence, jusqu'à avoir épuisé tous les chiffres. De nombreux programmes utiliseront ce type de logique.

```
def somme_des_chiffres(N):
    # Renvoie la somme des chiffres de N
    somme = 0  #Somme des chiffres vus jusqu'ici
    while N > 0:
        # print("valeur de N :",N)
        # print("valeur de somme :",somme)
        a = N%10           # chiffre des unités
        b = N//10          # chiffres avant les unités
        somme = somme + a  # On ajoute "a" à la somme en cours
        N = b              # On garde seulement le reste des chiffres et on recommence
    return somme
```

Recopiez le code précédent et appelez la fonction sur l'entier de votre choix pour vérfier qu'elle fonctionne bien. Imaginez l'exécution de l'algorithme et assurez-vous de comprendre pourquoi il fonctionne. N'hésitez pas à "décommenter" (retirer le #) les deux instructions sous le "`while`" pour afficher les étapes de calcul lors de l'exécution.

Syntaxe

Voici les éléments importants de la syntaxe d'une fonction :

- Le mot-clé `def` signale le début de la définition d'une fonction. La définition est placée dans le bloc indenté sous `def`.
- Après `def`, on écrit le nom que l'on souhaite donner à notre fonction (comme pour les variables, on peut utiliser des lettres, des chiffres et `_`, mais pas commencer par un chiffre).
- Juste après le nom de la fonction, on indique entre parenthèses les noms "génériques" que l'on donne aux arguments d'entrée dans la définition de la fonction (comme les noms "a" et "b" dans l'algorithme d'Euclide). On peut en mettre autant que l'on veut (y compris 0 ; dans ce cas on laisse les parenthèses vides). Lorsque la fonction sera appelée, il faudra utiliser le nombre correspondant d'arguments d'entrées, et les variables génériques seront remplacées par ceux-ci (par exemple, l'appel `Euclide(28,36)` nous dit qu'il faut exéctuer l'algorithme dans le cas particulier où $a = 28$ et $b = 36$). Les noms génériques peuvent être choisis librement comme toute autre variable, et peuvent même être des noms déjà utilisés : ils remplacent les variables précédents à l'intérieur de la fonction, et n'existent pas en dehors de la fonction. On dit que ce sont des variables *locales* (ou "muettes").
- Le mot-clé `return` sert à dire ce que renvoie la fonction en sortie. Lorsque cette instruction est atteinte, on interrompt immédiatement l'exécution de la fonction (même s'il reste des instructions plus loin à l'intérieur de celle-ci), et on poursuit le reste du programme : on dit que l'on *sort de la fonction*. Une fonction peut contenir plusieurs fois le mot-clé `return` (c'est le cas de la fonction `isEven` ci-dessus).

Appel d'une fonction

Voici la *définition* d'une fonction très simple qui renvoie la somme de deux entrées :

```
def somme(a,b):
    return a + b
```

Pour *exécuter* la fonction `somme`, il va falloir choisir des arguments d'entrées à mettre à la place des variables "muettes" `a` et `b`, puis *appeler* la fonction avec ces arguments, par exemple comme ceci :

```
>>> somme(19,23)    # Exécute somme(a,b) avec a = 19 et b = 23
42
>>> x = 10
>>> y = 11
>>> z = somme(x,y)  # Exécute somme(a,b) avec a = x et b = y, assigne le résultat à z
>>> z
21
```

Notez que cela ne pose pas de problème d'écrire `somme(x,y)` au lieu de `somme(a,b)`. Lors de l'appel de la fonction, il n'est pas nécessaire de choisir les mêmes noms d'arguments que dans la définition de celle-ci : nos arguments seront automatiquement renommés en `a` et `b` par Python au moment de l'exécution. En effet, lorsque vous appelez la fonction `somme`, il faut imaginer qu'avant de débuter l'exécution, Python commence par assigner la valeur de vos arguments d'entrées aux variables `a` et `b`. Dans le premier exemple, il fait donc `a = 19` et `b = 23`, dans le second, il fait `a = x` et `b = y`.

Sorties multiples

Il est possible de renvoyer plusieurs sorties avec Python : il suffit de séparer les différents résultats de sortie par une virgule. Par exemple :

```
def min_max(a,b):
    # Renvoie min(a,b),max(a,b)
    if a < b:
        return a,b
    else:
        return b,a
```

Dans ce cas, lorsqu'on appelle la fonction, on peut réceptionner les résultats de sortie avec plusieurs variables séparées par une virgule :

```
>>> p = isEven(42)
>>> p
True
>>> a,b = min_max(42,17)
>>> a
17
>>> b
42
```

Appels en cascade

Une fonction peut être appelée au sein d'une autre fonction. Par exemple, la fonction suivante vérifie si un nombre est impair en utilisant un appel à la fonction `isEven` :

```
def isOdd(N):
    # Détermine si N est impair
    if isEven(N):
        return False
```

```
    else:
        return True
```

Il n'y a pas de limite à ce principe : une fonction peut appeler une fonction qui elle-même appelle une autre fonction etc., comme dans l'exemple suivant (vous verrez des exemples moins artificiels dans les exercices à la fin de ce chapitre).

```
def f1(n):
    a = f2(n+2,n)
    b = a*2
    return b

def f2(a,b):
    c = f3(a+b,a-b)
    d = c+1
    return d

def f3(u,v):
    return u*v
```

```
>>> f1(10)  #  Va appeler f2(12,10) qui va appeler f3(22,2)
90
```

A chaque entrée dans une nouvelle fonction, Python place un nouvelle feuille mémoire vierge au sommet de sa *pile* de feuilles mémoire, et à chaque sortie de fonction, la feuille du haut de la pile est retirée. Ce principe est illustré ci-dessous pour l'exécution de `f1(10)` :

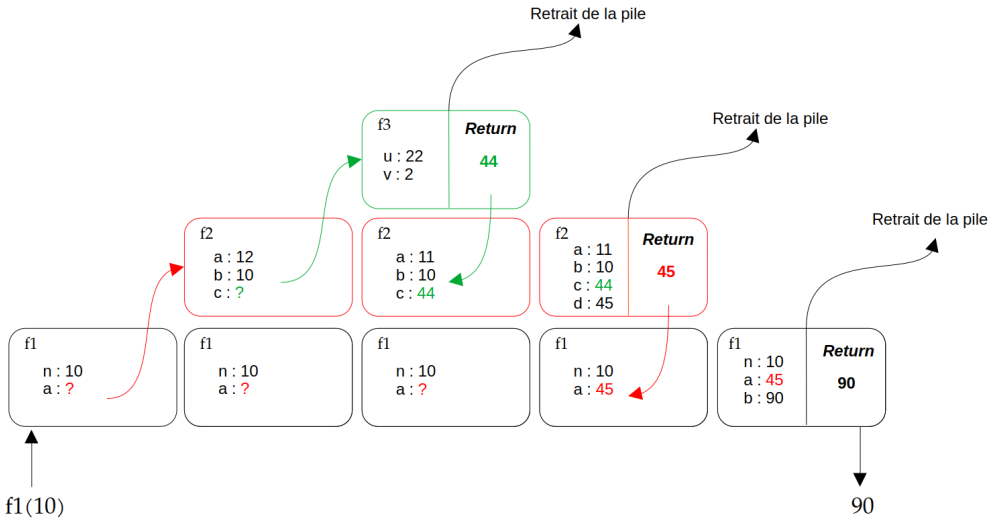


FIG. 4.3 – Pile d'exécution pour `f1(10)`

4 Fonctions sur des tableaux

Parmi les arguments d'entrée d'une fonction, on peut aussi passer des tableaux. Puisque les tableaux sont mutables, on peut créer des fonctions qui ne renvoient aucune sortie, mais qui modifient, ou "mutent" le tableau qui leur a été passé en argument (voir la Section 4 du Chapitre 3). Par exemple, la fonction suivante reçoit un tableau d'entiers en argument, et "incrémente" (c'est-à-dire, ajoute 1 à) tous les éléments du tableau, puis renvoie "`None`" (l'objet "rien").

```
def incremente_tableau(T):
    N = len(T)
    for i in range(N): # Parcours des éléments
        T[i] = T[i] + 1
    return None
```

Appelons cette fonction dans la console :

```
>>> T = [1,2,3]
>>> incremente_tableau(T)
>>> incremente_tableau(T)
```

Cela n'affiche rien, car la fonction ne renvoie rien. Mais le tableau a changé :

```
>>> T
[3,4,5]
```

Notez que ce comportement n'a pas lieu pour les immutables :

```
def incremente(a):
    a = a + 1
    return None
```

```
>>> a = 1000
>>> incremente(a)
>>> incremente(a)
>>> a
1000
```

Il faut donc retenir que lorsqu'une fonction reçoit un type mutable en argument, elle peut le modifier et cette modification est permanente, même après la sortie de la fonction.

return None

On crée souvent des fonctions qui ne renvoient rien, mais "font quelque chose" (afficher du texte dans la console, modifier un objet mutable, etc.). Parfois, on peut les créer par erreur, en oubliant d'écrire l'instruction `return`. Dans ce cas, Python accepte malgré tout votre définition et considère que la fonction renvoie `None`. Autrement dit, du point de vue de Python, écrire `return None` est équivalent oublier l'instruction `return`. On peut également écrire `return` (sans le `None`) cela est aussi équivalent à `return None`.

5 De nouveaux programmes interminables

On peut noter que, même sans utiliser de boucle `while`, les fonctions introduisent une nouvelle façon de se retrouver avec des programmes qui ne terminent pas. En voici un exemple caricatural :

```
# Programme de ping pong :

# Définition des fonctions
def ping():
    # Affiche "ping" et appelle pong()
    print("ping")
    b = pong()
    return b

def pong():
    # Affiche "pong" et appelle ping()
    print("pong")
    b = ping()
    return b

# Appel
ping()
```

La fonction `ping` (qui ne reçoit aucun argument en entrée) affiche “ping” puis appelle la fonction `pong`. Celle-ci affiche “pong” puis appelle la fonction `ping`, et ainsi de suite. Notez qu’à aucun moment dans l’exécution, les instructions `return` ne sont atteintes : l’étape de sortie de la fonction n’a jamais lieu. En principe, l’exécution de `ping()` devrait donc durer éternellement, exactement comme une boucle `while` infinie. Mais en réalité, Python possède un garde-fou contre ce type d’appels infinis, et il finit par afficher l’erreur suivante :

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Nous reviendrons sur cette erreur au Chapitre 6.

Exercices du Chapitre 4

Dans ces exercices, les arguments des fonctions seront choisis à l'intérieur du script : on n'utilisera pas, sauf mention contraire, la fonction `input` *de Python.*

---

1. Créez une fonction `perroquet` qui prend en argument une chaîne de caractères, et renvoie la même chaîne de caractère deux fois (en séparant par un espace). Exécutez la définition de votre fonction, puis *appelez* `perroquet` sur la chaîne de caractères de votre choix dans la console :

```
>>> perroquet("Vive Thonny !")
"Vive Thonny ! Vive Thonny !"
```

   Dans les questions suivantes, enregistrez les définitions de fonctions et les appels à ces fonctions dans le même script (mais n'hésitez pas à utiliser aussi la console pour tester vos fonctions).

2* Créez une fonction `nombre_chiffres` qui renvoie le nombre de chiffres d'un entier donné. Appelez cette fonction sur 999 et 1000.

3. En reprenant vos programmes des exercices du Chapitre 2, créez un fonction `ecriture_base(N,b)` qui prend en argument en entier N et une base b comprise entre 2 et 10, et qui renvoie une chaîne de caractères représentant l'entier N en base b. En utilisant cette fonction, affichez l'écriture en base 5, 6 et 7 de tous les nombres de 0 à 100.

4. Proposez une spécification de la fonction remplie par chacun des algorithmes suivants *(Indice : ne pas hésiter à exécuter les algorithmes sur plusieurs exemples. Faire une table de vérité pour le premier algorithme).*

```
def mystere1(b1,b2):
    # b1,b2 sont des booléens
        if b1:
            return True
        return b2
def mystere2(n):
    # n est un <int>
    s = 0
    while s**2 < n:
        s+=1
    return s**2==n
```

5* Créez une fonction `super_chiffre(N)` qui calcule la somme des chiffres de N (en base 10), puis la somme des chiffres du résultat, etc., jusqu'à ce que le résultat ne contienne plus qu'un chiffre. *(Indice : votre fonction pourra appeler la fonction* `somme_des_chiffres` *du cours – que vous devrez donc recopier dans le même fichier).* Créer des fonctions `divisible_par_3` et `divisible_par_9` qui vérifient si un nombre est divisible par 3 ou 9, respectivement (ces deux fonctions pourront appeler `super_chiffre`).

6* Créez une fonction `isPrime` qui prend en argument un entier naturel et renvoie `True` s'il est premier, et `False` sinon (on ne cherchera pas à utiliser le crible d'Eratosthène).

7. En vous aidant de la fonction précédente, créez une fonction `countPrimes` qui prend en argument un entier naturel N et renvoie le nombre de nombres premiers entre 1 et N.

8* Créez une fonction `permutationCirculaire1` qui prend en argument un tableau `T` et ne renvoie rien, mais modifie le tableau de manière à décaler tous ses éléments d'une case vers la gauche, en mettant le premier élément en dernière position. Par exemple

```
>>> T = [1,2,3,4,5]
>>> permutationCirculaire1(T)
>>> T
[2,3,4,5,1]
```

9. Créez une fonction `permutationCirculaire` qui prend en arguments un tableau `T` et un entier `n`, et ne renvoie rien mais modifie le tableau de manière à décaler tous ses éléments de `n` cases vers la gauche (en considérant que la dernière case est à gauche de la première). Par exemple

```
>>> T = [1,2,3,4,5]
>>> permutationCirculaire(T,2)
>>> T
[3,4,5,1,2]
```

10. Dans le jeu de Nim, on place N bâtonnets entre deux joueurs, et ceux-ci retirent chacun leur tour entre 1 et 3 bâtonnets. Le joueur qui retire le dernier bâtonnet perd la partie. Le but est de créer un programme capable de joueur à ce jeu.

(a) Créez une fonction `afficherNim(N)` qui prend en argument un nombre de bâtonnets et affiche dans la console la configuration de la partie. (Utiliser par exemple le caractère "|" pour représenter chaque bâtonnet).

```
>>> afficherNim(10)
||||||||||
```

(b) Créez une fonction `playerInput()` qui ne prend aucun argument. Cette fonction invite le joueur à entrer un nombre entre 1 et 3, puis retourne la valeur choisie. Pour cette question seulement, on utilisera la fonction `input` de Python.

(c) Écrire une fonction `computerChoice(M)` qui prend en argument le nombre M de bâtonnets restants et renvoie un nombre entre 1 et 3 (mais plus petit que M) correspondant au choix de l'ordinateur. Pour l'instant, choisir ce nombre au hasard.

(d) Écrire une fonction `play(N)` qui affiche l'état initial avec N bâtonnets, puis fait jouer successivement le joueur et l'ordinateur jusqu'à ce que la partie prenne fin. On donnera au joueur le choix de jouer le premier ou de laisser l'ordinateur commencer. Cette fonction utilisera des appels aux fonctions des questions (a), (b) et (c).

(e) On dit qu'une position est perdante si l'adversaire peut s'assurer la victoire, quelque soit le coup que l'on joue. Par exemple, la position avec 1 bâtonnet restants est évidemment perdante, mais aussi la position avec 5 bâtonnets restants. On note $P(n)$ la proposition

$$P(n) : \text{“La position avec } 4n+1 \text{ bâtonnets restants est perdante”}$$

Montrer que $P(n)$ est vraie pour tout entier $n \in \mathbb{N}$.

(f) Changer la fonction `computerChoice(N)` pour que l'ordinateur choisisse toujours, si c'est possible, de placer le joueur dans une position perdante, et joue au hasard sinon.

Mini-projet : casser un chiffrement

11. Le chiffrement de Vigenère permet de coder un message à l'aide d'un mot clé : à partir du message clair, on applique à chaque lettre un décalage alphabétique correspondant à une lettre du mot-clé. La lettre `A` de la clé signifie qu'il faut décaler de 0, la lettre `B`, de 1, etc. Par exemple, si la clé est `musique`, la suite des décalages est 12,20,18,8,16,20,4. Le message codé est obtenu en appliquant ces décalages lettre à lettre dans l'ordre, en répétant la clé si elle est plus courte que le message. Par exemple :

```
j'adore ecouter la radio toute la journee
m usiqu emusiqu em usiqu emusi qu emusiqu
^ ^^^^^ ^^^^^^^ ^^ ^^^^^ ^^^^^ ^^ ^^^^^^^

V'UVWHY IOIMBUL PM LSLYI XAOLM BU NAOJVUY.
```

Le but de cet exercice est de “casser” le message se trouvant à la fin de la feuille d'exercice, qui a été crypté par cette méthode. *Pour mener cet exercice à bien, chaque fonction devra être soigneusement testée avant de continuer. Détecter les erreurs au plus tôt les rend bien plus faciles à localiser.*

(a) Créez une fonction `letter2num(c)` qui prend en argument une lettre `c` et qui renvoie la position alphabétique de cette lettre, avec la convention de commencer à 0 :

```
>>> letter2num("a")
0
>>> letter2num("z")
25
```

On supposera que `c` est une lettre minuscule sans accent ni cédille. On pourra utiliser les fonctions `char` et `ord` de Python :

```
>>> ord("a")
97
>>> char(98)
'b'
>>> ord("c")
99
```

(b) Créez une fonction `normalize(c)`, qui prend en argument un caractère `c`, et le renvoie inchangé, sauf si c'est l'un des caractères diacritiques suivants,

```
à â ä é è ê ë ï î ô ö ù û ü ç
À Â Ä É È Ê Ë Î Ï Ô Ö Ù Û Ü Ç
```

auquel cas, elle renvoie le caractère normal correspondant. Par exemple,

```
>>> normalize('ç')
'c'
>>> normalize('Ü')
'U'
>>> normalize('a')
'a'
```

(c) Créez une fonction `shift_char(c,l,signe)` qui prend en argument un caractère `c` (qui peut être diacritique), une lettre minuscule normale `l`, et un signe ± 1, et décale le caractère `c` du nombre de places correspondant au caractère `l`, dans le sens positif si `signe = 1`, et négatif si `signe = -1`. Par exemple

```
>>> shift_char("e","a",+1)
'e'
>>> shift_char("È","c",-1)
'C'
>>> shif_char(" ","f",1)
" "
```

On utilisera la fonction `isalpha()` de Python, qui permet de détecter si un caractère est une lettre alphabétique, et la fonction `isupper()`, qui dit si une lettre est majuscule ou minuscule :

```
c1 = 'a'
c2 = ' '      #Espace
c3 = '\n'     #Retour à la ligne
c4 = 'È'
print(c1.isalpha()) # -> True
print(c1.isupper()) # -> False
print(c2.isalpha()) # -> False
print(c3.isalpha()) # -> False
print(c4.isupper()) # -> True
```

(d) Créez une fonction

`vigenere(message,key,signe)`,

qui applique le code de Vigenère à `message` avec la clé `key`. La clé est supposée ne contenir que des lettres minuscules non diacritiques. Si la valeur de `signe` est -1, on appliquera un décalage négatif au lieu d'un décalage positif. Testez votre fonction avec l'exemple donné en début d'exercice puis codez un texte de votre choix.

Nous allons voir comment casser ce chiffrement lorsque la clé utilisée est assez courte (pas plus de 15 lettres), en exploitant la fréquence de la lettre "e" dans la langue française. L'idée est de sélectionner pour chaque N, une clé de taille N selon le critère suivant :

Parmi toutes les clés possibles de taille N, en choisir une qui produit le plus de "e" dans le message décodé.

Pour chaque $N = 1, 2, ..., 15$, on détermine une clé de taille N vérifiant cette condition, puis on décode le message avec cette clé. Si cela a fonctionné, l'un des 15 textes obtenus sera alors lisible !

(e*) Créez une fonction `nb_occurrences(s,c)` qui compte le nombre d'occurrences du caractère `c` dans la chaîne de caractères `s`.

(f*) Créez une fonction `lettre_plus_frequente(s)` qui renvoie la lettre la plus fréquente dans la chaîne de caractères `s` (en choisissant arbitrairement en cas d'égalité).

(g*) Écrire une fonction `nieme_lettres(message,M,i)` qui retient une lettre sur M de `message` en commençant à la position i. Il faut ignorer les caractères spéciaux (espaces, virgules, etc.). Par exemple

```
>>> nieme_lettres("Petit exemple",3,0)
'Pixp'
```

(h) Créez une fonction `guess_key(message,N)` qui renvoie une clé de taille N choisie selon le critère ci-dessus. *(Indice : pour chaque i entre 0 et $N-1$, créer la chaîne de caractère qui contient une lettre sur N du message à partir de la lettre en position i).*

(i) Vous pouvez décoder le Texte 1 de l'appendice avec cette méthode.

Chapitre 5

Validité et complexité des algorithmes

"Attention aux bugs dans le code ci-dessus ; j'ai seulement démontré qu'il est correct, je ne l'ai pas testé."
Donald Knuth dans une lettre à Peter van Emde Boas (1977)

Nous avons vu dans le chapitre précédent que certaines fonctions pouvaient être calculées à l'aide d'algorithmes. Nous avons constaté, sur l'exemple de l'algorithme d'Euclide, qu'il n'est pas toujours évident qu'un algorithme calcule effectivement la fonction pour laquelle il est prévu. Même si l'algorithme renvoie le bon résultat sur de nombreux exemples, il se peut qu'il soit incorrect mais que nous n'ayons pas encore trouvé les entrées sur lesquelles il se trompe. Dans la première partie de ce chapitre, nous allons voir une méthode pour démontrer mathématiquement la validité d'un algorithme.

D'autre part, nous allons voir qu'il est possible de créer plusieurs algorithmes différents pour réaliser une même fonction, et il est important de pouvoir comparer ces algorithmes. Pour ce faire, la deuxième partie du chapitre aborde la notion de "complexité" d'un algorithme, c'est-à-dire, la quantité d'opérations qu'il nécessite pour arriver au résultat.

1 Terminaison et correction d'un algorithme

Définition 5.1 : Validité d'un algorithme

Etant donné une fonction, on dit qu'un algorithme destiné à calculer cette fonction est *valide* si

(i) Il *termine* : quelque soit l'entrée, il finit par renvoyer un résultat, correct ou non (autrement dit, pas de boucles infinies).

(ii) Il est *correct*, c'est-à-dire que le résultat qu'il renvoie est effectivement la fonction attendue de l'entrée.

Pour se convaincre qu'un algorithme est valide, la démarche la plus intuitive est de tester l'algorithme sur de nombreuses entrées différentes, et vérifier que les sorties correspondent au résultat attendu. Cette démarche est fort utile, elle est appliquée de manière systématique quant on crée des algorithmes. Elle permet souvent de découvrir les *bugs*, qui sont des erreurs involontaires, ou des cas de figure imprévus par la personne qui a inventé l'algorithme. En revanche, cette démarche ne peut que rarement *garantir* le fonctionnement de l'algorithme dans tous les cas. [1]

[1] "Program testing can be used to show the presence of bugs, but never to show their absence !" Edsger W. Djikstra. ("Tester un programme peut être utilisé pour montrer la présence de bugs, mais jamais pour en montrer l'absence !")

Pour s'assurer que l'algorithme fonctionne **dans tous les cas**, on peut essayer de fournir une démonstration mathématique. Celle-ci contient alors deux étapes :

1. *Terminaison :* on montre que l'algorithme termine (il ne se retrouve jamais piégé dans une boucle infinie) sur n'importe quelle entrée autorisée par la spécification.
2. *Correction :* on montre que dans tous ces cas, l'algorithme renvoie le résultat attendu, conformément à la spécification.

Validité de l'algorithme d'Euclide

Illustrons cela dans le cas de l'algorithme d'Euclide, dont on rappelle le programme en Python ci-dessous :

```
def Euclide(a,b):
    # Renvoie le PGCD de a et b
    r = a % b
    while r > 0:
        a = b
        b = r
        r = a % b
    return b
```

Nous avons vu que ce programme a l'air de marcher pour $a = 28$ et $b = 36$ (et le fait qu'il porte le nom d'Euclide nous porte à croire qu'il est correct). Mais avouons-le, il n'est pas évident à première vue que ce programme termine quelque soit les entiers a et b donnés en argument ! Comment est-on sûr que la condition `r > 0` finisse par être fausse ? Pourquoi renvoyer b, et pas a, ou r ? Quel rapport avec le PGCD ? La démonstration ci-dessous est là pour répondre à ces questions.

Démonstration de la validité de l'algorithme d'Euclide.

1. Terminaison. La seule possibilité pour que le programme ne termine pas, c'est que l'algorithme reste coincé dans la boucle `while` : notre tâche consiste à montrer que cela ne peut jamais arriver. Notons $a_1, a_2, a_3, ...$ la suite des valeurs prises par la variable `a` pendant l'algorithme. Faisons de même pour `b` et `r`. D'après ces définitions et le programme, nous avons les relations suivantes : pour tout $n \geq 1$,

$$a_{n+1} = b_n\,, \quad b_{n+1} = r_n. \tag{5.1}$$

Notons q_n le quotient de la division euclidienne de a_n par b_n, de sorte que pour tout $n \geq 1$,

$$a_n = b_n q_n + r_n,$$

avec $0 \leq r_n \leq b_n - 1$. En particulier, comme $b_{n+1} = r_n$ d'après (5.1),

$$\forall n \geq 1\,, \quad b_{n+1} \leq b_n - 1\,,$$

et

$$\forall n \geq 2\,, \quad b_n = r_{n-1} \geq 0.$$

La suite $b_2, b_3, \ldots$ est ainsi une **suite strictement décroissante d'entiers positifs ou nuls** : elle est donc finie. Ceci n'est possible que si l'algorithme termine.

2. Correction. Notons N le nombre de valeurs prises par la variable `b`. Remarquons d'une part que $r_N = 0$ (car $r_N \geq 0$ par définition d'un reste dans une division euclidienne, et si $r_N > 0$, nous aurions fait encore une itération, donc N ne serait pas la dernière valeur prise par b_N). On en déduit que $a_N = b_N q_N$, donc a_N est divisible par b_N (ce que l'on note $b_N | a_N$: "b_N divise a_N") et ainsi

$$b_N = \mathrm{PGCD}(a_N, b_N).$$

D'autre part, b_N est le résultat renvoyé par l'algorithme. Par conséquent, pour montrer la correction, nous devons montrer que cette valeur est bien le PGCD de a et b, autrement dit, que

$$\mathrm{PGCD}(a_N, b_N) = \mathrm{PGCD}(a_1, b_1). \tag{5.2}$$

La clé de la preuve est la remarque suivante : pour tout $n \in \{1, \ldots, N\}$,

$$\mathrm{PGCD}(a_n, b_n) = \mathrm{PGCD}(b_n, r_n). \tag{5.3}$$

En effet, tout diviseur commun de a_n et b_n est un diviseur commun de b_n et r_n, puisque

$$d|a_n \text{ et } d|b_n \implies d|\underbrace{(a_n - b_n q_n)}_{=r_n}.$$

(Pourquoi ?) Réciproquement, tout diviseur commun de b_n et r_n est un diviseur commun de a_n et b_n, puisque

$$d|b_n \text{ et } d|r_n \implies d|\underbrace{(r_n + b_n q_n)}_{=a_n}.$$

Ainsi, les diviseurs communs de a_n et b_n sont exactement les mêmes que ceux de b_n et r_n. Le plus grand diviseur commun des uns est donc égal au plus grand diviseur commun des autres, ce qui établit l'égalité (5.3) ci-dessus.

La remarque précédente permet de démontrer un **invariant** du programme, c'est-à-dire, une proposition $P(n)$ qui est à chaque fois vraie à l'itération n. Spécifiquement, nous allons montrer que la proposition

$$P(n) : \text{“}\mathrm{PGCD}(a_n, b_n) = \mathrm{PGCD}(a_1, b_1)\text{”}$$

reste vraie pour tout $n \in \{1, \ldots, N\}$. En effet, puisque $a_{n+1} = b_n$ et $b_{n+1} = r_n$, la propriété (5.3) peut s'écrire

$$\forall n \in \{1, \ldots, N-1\}, \quad \mathrm{PGCD}(a_n, b_n) = \mathrm{PGCD}(a_{n+1}, b_{n+1}).$$

Ainsi, il est clair que si $P(n)$ est vraie, alors $P(n+1)$ est automatiquement vraie aussi. Puisque $P(1)$ est (évidemment) vraie, on en déduit donc que $P(2)$ est vraie, puis $P(3)$ à son tour, et ainsi de suite. $P(N)$ est donc vraie. Or $P(N)$ n'est autre que l'égalité (5.2) ci-dessus. Nous avons donc obtenu ce que nous cherchions à montrer. □

Méthode générale

Voici les aspects généraux de la méthode précédente qu'il faut retenir et appliquer dans les exercices.

1. Introduire les suites de valeurs prises par chaque variable au cours des itérations, et énoncer les relations qu'elles vérifient (cette partie consiste essentiellmenet à traduire le programme en langage mathématique)
2. Pour montrer la terminaison d'un programme, on cherche souvent à démontrer l'existence d'une suite d'entiers naturels positifs ou nuls qui décroit strictement à chaque itération. Dans l'exemple d'Euclide, c'était la suite $(b_n)_{n\geq 2}$. Comme une telle suite ne peut contenir qu'un nombre fini de termes, le nombre d'itération doit être fini.
3. Pour montrer la correction d'un programme, on trouve les **invariants** de boucle. Un invariant est une condition qui est préservée d'une itération à l'autre d'une boucle `for` ou `while`. Un bon invariant $P(n)$ est tel que
   (a) $P(1)$ est (évidemment) vrai au début du programme, pourvu que les entrées soient valides,
   (b) si $P(n)$ est vrai, alors $P(n+1)$ aussi, tant que la boucle n'est pas terminée.
   (c) Le fait que $P(N)$ soit vrai (où N est le nombre d'itérations) implique que l'algorithme est correct.

Dans certains cas, l'invariant de boucle sera suggéré dans l'exercice et il faudra démontrer que c'est en effet un invariant. Parfois, il faudra trouver l'invariant soi-même.

Si l'on ne voit pas quelle suite introduire pour la terminaison, ou quel invariant définir pour la correction, on peut exécuter l'algorithme sur quelques exemples et repérer les motifs qui se répètent systématiquement (par exemple : "tiens ce nombre est toujours pair", ou "à chaque fois, a est plus petit que b", etc.). Une fois que le bon invariant est découvert, il ne reste plus qu'à utiliser le principe de récurrence (voir la Section 2 de l'Annexe A) pour conclure. L'étape (a) donne l'initialisation et (b) donne l'hérédité tant que $n+1 \leq N$. On en déduit par récurrence que $P(N)$ est vrai et on conclut grâce à (c).

2 Compléxité algorithmique

Définition

Étant donné la spécification d'une fonction, il peut exister plusieurs façons de la mettre en œuvre par un algorithme, et certains algorithmes peuvent être plus rapides que d'autres pour calculer la même fonction. Par exemple, pour calculer la somme des entiers de 1 à N, on peut utiliser une approche "naïve" comme ceci :

```
def S1(N):
    # Calcule la somme des N premiers entiers
    S = 0
    for i in range(N+1): # i  = 0,...,N -> Rappel : N+1 est exclu
        S+=i
    return S
```

Mais on peut aussi "tricher" un peu en utilisant la formule $1+2+\ldots+N = \frac{N(N+1)}{2}$ (voir Section 1.1 de l'Annexe A), ce qui donne

```
def S2(N):
    # Calcule la somme des N premiers entiers
    S = N*(N+1)//2
    return S
```

La fonction remplie par ces deux algorithmes est la même, mais les algorithmes sont différents. Et cette différence est de taille : comparez les temps d'exécution de ces deux fonctions pour $N = 10^9$...

La *complexité algorithmique* est une mesure de la performance d'un algorithme.

Définition 5.2 : Complexité algorithmique

La *complexité* d'un algorithme est la fonction $C(N)$ qui décrit le nombre **maximal** d'opérations élémentaires qui devront être effectuées par cet algorithme sur n'importe quelles données d'entrée de "taille" N.

- Pour parler de la complexité de l'algorithme, il faut d'abord définir ce qu'on entend par la "taille" des données. Ceci est définit individuellement pour chaque problème rencontré. Il peut s'agir par exemple d'un entier N passé en argument, de la longueur d'un tableau ou d'une chaîne de caractères, ou encore du nombre de bits nécessaire pour représenter les entrées en mémoire.
- Par opérations élémentaires, on entend les opérations arithmétiques (addition, soustraction, multiplication et division) sur les nombres entiers ou flottants, les comparaisons et les assignations. Attention :

les opérations “natives” de Python (comme la concaténation des chaînes de caractères ou les opérations sur les tableaux) ne sont pas toutes des opérations élémentaires, voir la Table 5.1

- Le nombre d'opérations élémentaires effectuées par un même algorithme dépend bien sûr des entrées. Même sur différentes entrées de taille N, le temps d'exécution peut différer grandement d'une entrée à l'autre. Par exemple, considérons l'algorithme suivant, qui détermine la présence de la lettre “e” dans une chaîne de caractère `s` (la taille des données est ici donnée par la longueur N de `s`) :

```
def contient_la_lettre_e(s):
    # Renvoie Vrai si la chaîne s contient au moins un "e", et Faux sinon
    N = len(s)
    for i in range(N):
        # Pour i = 0,...,N-1
        # On teste si la i-ième lettre est un e
        if s[i] == "e": # Ce test coûte une opération par itération
            # Si oui, ce n'est plus la peine de continuer, on peut renvoyer True
            return True
    # Si le programme arrive à ce point, c'est que toutes les lettres de s
    # ont été parcourues et aucune n'était un "e" : on renvoie False
    return False
```

Si `s` *commence* par “e”, alors l'algorithme n'effectuera qu'une seule comparaison, alors que si `s` ne contient aucun “e”, l'algorithme fera N comparaisons. Par définition, la complexité de l'algorithme est donc

$$C(N) = N$$

c'est-à-dire, on donne la complexité **dans le pire des cas**. [2]

Instruction	Arguments	Nombre d'opérations élémentaires
`len(s)`/`len(T)`	`s` une chaîne de caractères/`T` un tableau	1
`s[i]`/`T[i]`	`i` un entier	1
`s[i:j]`/`T[i:j]`	`i`,`j` deux entiers	N_{ij} la longueur du résultat
`T[i] = a`	`a` n'importe quel objet	1
`T.pop()`	`T` un tableau	1
`s1+s2`	`s1`,`s2` deux chaînes de caractères	$N_1 + N_2$ la somme des longueurs de `a` et `b`.
`T*k`	`T` un tableau de longueur N, `k` un entier	kN, la longueur du tableau créé.
`[None]*k`	`k` un entier	k, la longueur du tableau créé (cas particulier important du précédent)
`T.append(a)`	T un tableau, `a` n'importe quel objet	N, la longueur du tableau `T` final

TAB. 5.1 – Coût en opérations élémentaires de certaines instructions.

[2] Il est aussi classique en informatique de considérer la complexité “dans le meilleur des cas”, ou “en moyenne”, sur toutes les entrées de taille N possibles. Dans ce cours, nous ne considérerons que la complexité dans le pire des cas.

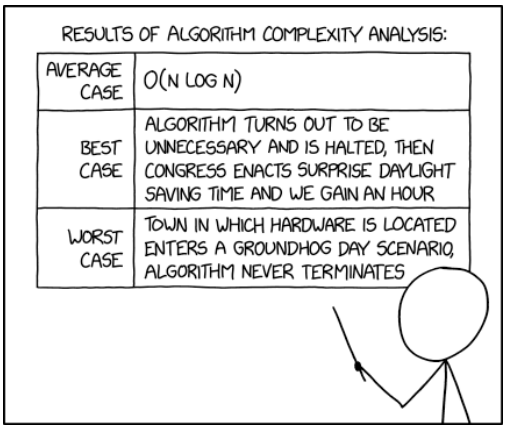


FIG. 5.1 – Source : `xkcd.com`

Comparaison de fonctions de N

Dans ce paragraphe, nous allons comparer des fonctions de N comme $f(N) = N$, $g(N) = N^2$, ou encore $h(N) = N^2 + 2N + 1$, etc. Le but est de savoir quelle fonction augmente le plus vite en fonction de N. Pour ce faire, nous introduisons la notation "grand O" (comme "Ordre de grandeur")

Définition 5.3 : Notation "grand O"

Soient f et g deux fonctions *à valeurs positives ou nulles* d'une variable entière. On écrit

$$f(N) = O(g(N))$$

s'il existe un entier N_0 et une constante $K > 0$ (indépendante de N) tels que

$$f(N) \leq Kg(N) \quad \text{pour tout } N \geq N_0.$$

La proposition "$f(N) = O(g(N))$" dit que f "n'augmente pas plus vite que g" (à la constante K près, qui sera considérée comme non importante). Par exemple, $2N + 3N^2 = O(N^2)$ puisque pour tout $N \geq 2$,

$$2N + 3N^2 \leq N^2 + 3N^2 \leq 4N^2$$

(on est donc dans le cadre de la définition avec $N_0 = 2$ et $K = 4$).

Définition 5.4 : Notation "Θ"

On écrit

$$f(N) = \Theta(g(N))$$

si $f(N) = O(g(N))$ *et* $g(N) = O(f(N))$.

Autrement dit, la proposition "$f(N) = \Theta(g(N))$" dit que "f et g augmentent à la même vitesse" (toujours à une constante près indépendante de N). On utilisera souvent les propriétés suivantes :

(i) *Additivité.* Si $f(N) = O(h(N))$ et $g(N) = O(h(N))$, alors $f(N) + g(N) = O(h(N))$.

(ii) *Multiplicativité.* Si $f_1(N) = O(g_1(N))$ et $f_2(N) = O(g_2(N))$, alors $f_1(N)f_2(N) = O(g_1(N)g_2(N))$.

Les mêmes propriétés sont vraies en remplaçant le grand O par Θ. Par exemple,

$$\frac{N(N+1)}{2} = \Theta(N^2).$$

En effet, $N+1 = O(N)$ (pourquoi ?) donc $\frac{N(N+1)}{2} = O(N^2)$ par multiplicativité. D'autre part, $N^2 \leq N(N+1) \leq 2\frac{N(N+1)}{2}$ donc $N^2 = O\left(\frac{N(N+1)}{2}\right)$.

Classes de complexité

Lorsqu'on conçoit des algorithmes, on essaye en général de classer leur complexité dans une échelle qui va de $\Theta(1)$ à $\Theta(2^N)$ (voire pire : il n'y a pas de limite à la lenteur possible d'un algorithme), voir Table 5.2 ci-dessous.

constante	logarithmique	linéaire	quasi-linéaire	quadratique	polynomiale	exponentielle
$\Theta(1)$	$\Theta(\log(N))$	$\Theta(N)$	$\Theta(N \log N)$	$\Theta(N^2)$	$\Theta(N^m)$, $m \geq 3$	$\Theta(2^N)$
(instantanné)	*(ultra-rapide)*	*(rapide)*	*(rapide)*	*(plutôt lent)*	*(lent)*	*(catastrophique)*

TAB. 5.2 – Classes de complexité. Les commentaires entre parenthèses sont donnés à titre indicatif.

En reprenant les fonctions `S1` et `S2` définies en début de section, l'exécution de `S2` demande une multiplication et une division, soit au total deux opérations élémentaires, et ce, quelque soit N. On dit donc que ce programme a une complexité constante, ou $\Theta(1)$ (ordre 1) [3]. Pour `S1`, il faut faire N additions (une pour chaque entier i), soit un total de N opérations élémentaires. On dit que ce programme a une complexité linéaire, ou $\Theta(N)$. Voyons deux autres exemples :

Exemple 1 : La fonction suivante détermine combien de fois la lettre "e" apparaît dans une chaîne de caractère :

```
def nombre_de_e(s):
    # Renvoie le nombre d'occurrences de la lettre "e" dans la chaîne s.
    N = len(s)
    occurrences = 0
    for i in range(N):
        if s[i] == "e":
            occurrences+=1
    return occurrences
```

Notons N la taille de la chaîne de caractère `s` en entrée. L'exécution de cette fonction nécessite N comparaisons (à chaque fois que l'on teste si `s[i] == "e"`), et au plus N additions (si la chaîne `s` ne contient que la lettre `"e"`). Le nombre d'opérations est donc au plus $2N$ (et exactement $2N$ pour toutes les les entrées de la forme `"ee...ee"`). La complexité vaut donc $\Theta(N)$, une complexité linéaire.

Exemple 2 : On considère le programme suivant, qui détermine si une chaîne de caractères contient un doublon, c'est-à-dire, deux caractères identiques :

[3] Ce n'est pas tout à fait vrai, car la multiplication de N par $N+1$ est de plus en plus chère lorsque N augmente. On pourrait compter le nombre d'additions et multiplications en binaire, ce qui nous mènerait à une complexité de $O(\log_2(N)^2)$ pour `S2`, car N et $N+1$ se représentent à l'aide d'environ $n = \log_2(N)$ bits et la multiplication binaire des nombres à n bits a une complexité de $O(n^2)$. Par la suite, nous ignorerons cet aspect et ferons comme si toutes les opérations élémentaires sur les nombres entiers coûtaient $\Theta(1)$. Les opérations sur les flottants coûtent réellement $\Theta(1)$.

```python
def contient_doublon(s):
    # Détermine si s contient deux fois un même caractère
    N = len(s)
    for i in range(N):
        for j in range(N):
            if i!=j and s[i] == s[j]:
                # Doublon trouvé !
                return True
    # Aucun doublon trouvé
    return False
```

Soit N la taille de la chaîne de caractères `s`, déterminons la complexité $C(N)$ de cet algorithme en fonction de N. Dans le pire des cas (s'il n'y a aucun doublon), pour chaque $i = 0, \dots, N-1$, le programme effectue une boucle sur j qui contient N itérations, dont chacune effectue deux comparaisons (pour évaluer les deux booléens dans la condition du `if`). Le nombre de comparaisons effectuées par l'algorithme est donc, dans le pire des cas,

$$C(N) = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} 2 \right) = \sum_{i=0}^{N-1} (N \times 2) = N \times 2N = 2N^2.$$

La complexité vérifie donc $C(N) = \Theta(N^2)$, une complexité quadratique.

Il est possible de légèrement améliorer ce programme en faisant commencer la deuxième boucle `for` à partir de $i+1$ (en effet, dans la première version, chaque couple d'indices $x < y$ est testé deux fois, une première fois avec $i = x$ et $j = y$, la deuxième fois avec $i = y$ et $j = x$) :

```python
def contient_doublon(s):
    # Détermine si s contient deux fois un même caractère
    N = len(s)
    for i in range(N):
        for j in range(i+1,N):
            if s[i] == s[j]:
                # Doublon trouvé !
                return True
    # Aucun doublon trouvé
    return False
```

Cette fois, le nombre de comparaisons pour une chaîne sans doublons, et donc dans le pire des cas, est

$$C(N) = \sum_{i=0}^{N-1} \left(\sum_{j=i+1}^{N-1} 1 \right) = \sum_{i=0}^{N-1} (N-1-i) = \sum_{i'=0}^{N-1} i' = \frac{N(N-1)}{2}.$$

Ce programme fait donc à peu près 4 fois moins de comparaisons que le précédent, mais sa complexité est toujours $\Theta(N^2)$, car

$$\frac{N(N-1)}{2} = \frac{N^2 - N}{2} = \frac{1}{2} N^2 \underbrace{\left(1 - \frac{1}{N}\right)}_{\Theta(1)} = \Theta(N^2).$$

Complexité et P = NP

Créer des algorithmes de faible complexité est un enjeu important dans de nombreuses applications, par exemple en industrie (un programme plus rapide coûte moins de temps de calcul donc offre des économies

qui peuvent devenir considérables), ou en cryptographie (la sécurité de la plupart des méthodes de cryptage repose sur le fait que l'algorithme pour décoder le chiffrement demande un trop long temps de calcul).

Un problème célèbre est celui du "voyageur de commerce", qui demande de trouver un itinéraire le plus court possible pour un camion qui doit effectuer N livraisons dans N villes différentes. Pour trouver la solution optimale, on peut calculer la longueur de chacun des itinéraires possibles et retenir le plus court d'entre eux. Malheureusement, la complexité de cet algorithme est encore pire qu'exponentielle (puisqu'il y a $N \times (N-1) \times \ldots \times 1$ itinéraires possibles). Cela va fonctionner en pratique pour N entre 1 et 10, mais déjà pour $N = 20$, il faudrait plusieurs millénaires pour obtenir la réponse sur un ordinateur actuel.

Il y a d'autres problèmes de programmation pour lesquels on commence par trouver une solution très mauvaise comme celle-ci, mais il arrive qu'on se rende compte d'une astuce qui permet de réduire drastiquement la complexité (nous allons en voir un exemple pour le tri d'un tableau dans le Chapitre 6, ou le calcul de la suite de Fibonacci). Il est donc assez légitime de se demander : pour le voyageurs de commerce, n'y a-t-il pas une meilleure solution que d'énumérer tous les itinéraires ? Y a-t-il un moyen astucieux d'éviter une grande partie de cette longue énumération et d'aboutir à la solution optimale avec un algorithme de complexité polynomiale (c'est-à-dire, $\Theta(N^m)$ pour un certain m ?) Cette question n'est autre que la fameuse conjecture $P = NP$, et quiconque trouvera la solution à ce problème se verra décerner un million de dollars par l'institut mathématique Clay. [4]

[4] Un peu plus précisément, la classe P (pour polynomial) est définie comme la classe des problèmes qui peuvent être résolus à l'aide d'un algorithme de complexité au plus polynomiale (comme $O(N)$, $O(N^2), O(N^3)$ etc.) ; quant à la classe NP (pour non-deterministic polynomial), elle contient les problèmes qui sont résolus en temps polynomial mais par des machines imaginaires dites "non-déterministes" (des sortes d'ordinateurs capables à tout moment de se dédoubler en plusieurs copies, chaque copie pouvant ensuite poursuivre le calcul de manière indépendante, jusqu'à ce que l'une des copies renvoie un résultat). La conjecture $P = NP$ demande simplement si ces deux classes deux problèmes sont en fait égales. La réponse est à ce jour inconnue...

Exercices du Chapitre 5

Classes de complexité

1* Démontrez les propositions suivantes :
   (a) $N(N+1) + N^3 = \Theta(N^3)$,
   (b) $N\log(N) = O(N^2)$,
   (c) $1 + 2 + 3 + \ldots + N = \Theta(N^2)$,
   (d) Pour tout entier k, $N^k = O(2^N)$.

2* Pour tout entier $N \in \mathbb{N}$, soit $p(N)$ le plus petit entier tel que $N \leq 2^{p(N)}$. Démontrez que

$$p(N) = \Theta(\log(N))$$

3. Démontrez que pour tout k, l entiers naturels, $N^k = \Theta(N^l)$ si et seulement si $k = l$. Démontrez que les propositions $\log(N) = \Theta(N^k)$ et $N^k = \Theta(2^N)$ sont fausses pour tout entier k.

4. *(Question bonus)* Soit $k \in \mathbb{N}$ un entier. Dans cet exercice, on se propose de montrer que [5]

$$1^k + \ldots + N^k = \Theta(N^{k+1})$$

   (a) Montrez que $1 + \ldots + N^k \leq N^{k+1}$.
   (b) Montrez que pour tout $x \geq 0$,

$$(1+x)^k \geq 1 + kx.$$

   (Indice : procédez par récurrence).
   (c) Déduisez de la question précédente que pour tout $n \geq 1$ entier,

$$n^k \geq \frac{n^{k+1} - (n-1)^{k+1}}{k+1}$$

   (d) Montrez que $1 + \ldots + N^k \geq \frac{N^{k+1}}{k+1}$
   (e) Concluez en utilisant les questions précédentes.

Validité et complexité de quelques algorithmes

5* Voici une fonction qui calcule la somme des éléments d'un tableau :

```
def somme_elements(T):
    s = 0
    N = len(T)
    for i in range(N):
        s=s+T[i]
    return s
```

Démontrez la validité de cette fonction. On pourra introduire la suite $s_0, s_1, \ldots$ des valeurs prises par `s` et considérer l'invariant de boucle suivant

$$P(n) : \text{“}s_n = T[0] + \ldots + T[n-1]\text{”}.$$

Déterminez la complexité de cette fonction.

6. Créez une fonction qui calcule le plus grand élément d'un tableau et démontrez sa validité et sa complexité.

7. Reprenez la fonction `insertion.py` dans les exercices du Chapitre 3 (dans la partie sur le tri). Quelle est sa complexité en fonction de la taille du tableau `U` ?

8* Reprenez la fonction `tri.py` dans la même série d'exercices. Justifiez que sa complexité est quadratique.

Recherche dichotomique

9* Dans cet exercice, on considère un tableau T d'entiers non vide. Le but de l'exercice est de créer un algorithme qui cherche si un élément a est dans le tableau, et si oui, renvoie sa position j telle que $T[j] = a$. Pour permettre une recherche plus rapide, on commence par *trier* le tableau par ordre croissant (ce qui “coûte cher” mais n'a besoin d'être fait qu'une seule fois).

   (a) Montrez que, après l'avoir trié, on peut supposer que la longueur de T est une puissance de 2, quitte à répéter son dernier élément autant de fois que nécessaire.
   (b) Créez une fonction `pretraitement(T)` qui prend en argument un tableau `T` et renvoie un tableau `U` trié, de taille $N = 2^p$ et contenant les mêmes éléments que `T`.
   (c) Au départ, on sait seulement que si a est dans le tableau `U`, il se trouve entre les indices 0 et $N-1$ (inclus). Cependant, en comparant a avec l'élément t qui se trouve au milieu du tableau, on peut diviser par deux l'intervalle de recherche (si a est plus petit que t, il faut chercher entre 0 et $N/2-1$, et sinon, entre $N/2$ et $N-1$). Créez une fonction `find(a,U)` effectuant au plus p itérations en répétant ce principe. Le tableau d'entrée U sera supposé déjà trié et de taille $N = 2^p$ (on n'appellera donc pas `pretraitement` dans la fonction). On fera en sorte que l'algorithme vérifie un invariant de boucle du type

$$P(k) : \text{“Si } a \text{ est dans le tableau U, il se trouve entre les indices } i_k \text{ et } i_k + \tfrac{N}{2^k} - 1 \text{ (inclus)”}$$

   où $i_0, i_1, \ldots$ est la suite des valeurs prises par l'une des variables.
   (d) Indiquez la complexité de `find` en fonction de p. Comparez a la taille initiale du tableau `T`.
   (e) Démontrez la validité de `find`.

[5] Pour $k = 1, 2, 3$, les formules $1 + \ldots + N = \frac{N(N+1)}{2}$, $1^2 + \ldots + N^2 = \frac{N(N+1)(2N+3)}{6}$, $1^3 + \ldots + N^3 = \frac{N^2(N+1)^2}{4}$ sont relativement connues (vous pouvez les démontrer par récurrence). Plus généralement, il existe des formules similaires pour tout entier k, qui font intervenir les “nombres de Bernouilli”.

Vérificateur d'anagrammes

10. Le but de cet exercice est d'écrire une fonction `isAnagram` qui détermine si les chaînes de caractères `s1` et `s2` sont des anagrammes l'une de l'autre. Par exemple

```
>>> s1 = "I am Lord Voldemort"
>>> s2 = "Tom Marvolo Riddle"
>>> isAnagram(s1,s2)
True
```

On propose l'algorithme suivant : on vérifie que `s2` contient la première lettre de `s1` (sinon, ce ne sont pas des anagrammes, donc on stoppe l'algorithme), et on supprime cette lettre de `s1` et `s2`. Ensuite, on recommence avec les `s1` et `s2` ainsi obtenues, jusqu'à ce que `s1` n'ait plus aucune lettre.

(a*) Créez une fonction `findChar(s,c)` qui prend en argument une chaîne de caracètre `s` et un caractère `c`, qui renvoie `None` si `c` n'est pas dans `s`, et sinon, renvoie un `i` tel que `c = s[i]`. Par exemple

```
>>> findChar("e","La disparition")
>>> findChar("n","Thonny")
3
```

Indiquez sa complexité en fonction de la longueur de `s`.

(b*) Créez une fonction `delete(s,i)` qui retire le caractère en position `i` de `s` et renvoie la chaîne restante. Indiquez sa complexité en fonction de la longueur de `s` (*On pourra utiliser des slices et la concaténation de chaînes de caractères avec* `+`).

(c) Créez une fonction `delWhiteSpace(s)` qui retire tous les espaces de la chaîne de caractère `s` et renvoie la chaîne restante (pensez à réutiliser les fonctions précédentes !). Indiquez sa complexité.

(d) Dans cette question, on pourra faire appel à la fonction `lower()` de Python qui s'utilise comme ceci :

```
>>> M = "MesSaGe"
>>> m = M.lower()
>>> m
'message'
```

En utilisant les questions précédentes, créez la fonction `isAnagram`.

(e) Démontrez que `isAnagram` a une complexité quadratique en fonction de la longueur de `s1`.

(f) Dans cette question, on admet la validité des fonctions `findChar`, `delete`, `delWhiteSpace` et `lower`. Démontrez la validité de `isAnagram`. On pourra considérer l'invariant de boucle suivant

$$P(i) : \text{“Il existe une chaîne } t_i \text{ de longueur } i \text{ telle que } s_1^0 \sim s_1^i + t_i\,, \quad s_2^0 \sim s_2^i + t_i,\text{”}$$

où s_1^i, s_2^i sont les valeurs successives prises par les variables `s1` et `s2`, et où, pour deux chaînes s et t, la notation $s \sim t$ signifie que s et t sont des anagrammes, et $s + t$ représente la concaténation de s et t.

Chapitre 6

Récursivité

Avant de lire ce chapitre, il est préférable de se familiariser quelques notions de base sur la récursivité. Pour cela, le lecteur pourra se référer au Chapitre 6.

1 Introduction

Le mot *récursif* (du latin *recursum*, courir en arrière) fait référence à un principe apparemment paradoxal, qui consiste à définir un objet à partir de l'objet lui-même. Afin de ne pas tomber dans une définition circulaire, on doit définir des cas *de base*. De nombreux objets sont définis de manière récursive, par exemple

(i) x^n est défini comme $x \times x^{n-1}$ (avec le cas de base $x^0 = 1$).

(ii) La fonction factorielle est définie par $n! = n \times (n-1)!$ (avec le cas de base $0! = 1$).

(iii) L'ensemble des entiers naturels $\mathbb{N}$ peut être défini par la propriété récursive que si $n \in \mathbb{N}$, alors $n+1 \in \mathbb{N}$ (avec le cas de base $0 \in \mathbb{N}$). Selon cette définition, "5" n'est rien d'autre que le nom donné à $((((0+1)+1)+1)+1)+1$.

(iv) Le triangle de Sierpiński est une fractale, qu'on obtient par un processus récursif de subdivision : chaque triangle est subdivisé en 4 triangles égaux, et parmi ces 4 triangles, celui qui est au centre est retiré (avec comme cas de base un triangle équilatéral), voir Figure 6.1

(v) Une démonstration par récurrence est un procédé récursif : on établit qu'une suite de propositions est vraie en montrant que chacune est une conséquence de la proposition précédente (avec le cas de base que la première proposition est vraie).

FIG. 6.1 – Les 5 premières itérations pour la construction du triangle de Sierpiński (il faut poursuivre le processus à l'infini pour obtenir la véritable fractale, qui est l'ensemble des points qui ne sont jamais retirés par ce processus).

Malgré son apparence un peu contre-intuitive, nous allons voir dans ce chapitre que la récursivité est une notion à la fois commode et puissante en programmation.

2 Fonctions récursives en Python

Le point fondamental qui permet la récursivité, c'est qu'une fonction peut s'appeler elle-même. Une telle fonction est dite *récursive* (a l'inverse, une fonction qui n'est pas récursive est souvent qualifiée de fonction *itérative*) :

Définition 6.1 : Fonction récursive

Une fonction est dite *récursive* si au moins l'une des instructions qu'elle contient est un appel à cette fonction elle-même.

Voici la fonction récursive la plus simple :

```
def f():
    f()
```

Elle ne prend aucun argument et consiste juste à s'appeler elle-même. Lorsqu'on l'exécute, rien ne se passe pendant plusieurs secondes, puis l'erreur suivante est affichée dans la console :

```
>>> f()
...
  File "python/f.py", line 2, in rec
    f()
RecursionError: maximum recursion depth exceeded
```

Pour mieux comprendre ce qui se passe, modifions légèrement la fonction `f` : en plus de s'appeler elle-même, on lui fait afficher quelque chose à chaque fois qu'elle est appelée :

```
def f():
    print("f a été appelée !")
    f()
```

Lorsqu'on exécute cette fonction, on voit s'afficher le message `"f a été appelée !"` de nombreuses fois, avant que le message d'erreur n'apparaisse. En principe, l'exécution de cette fonction devrait se poursuivre à l'infini, et l'erreur que nous recevons n'est due qu'à une limitation matérielle : la taille finie de la mémoire de l'ordinateur. Un ordinateur disposant d'une quantité infinie de mémoire pourrait théoriquement exécuter ce code sans erreur : il continuerait à afficher le message sans interruption.

La possibilité pour une fonction de s'appeler elle-même ouvre la voix à des définitions de fonctions très élégantes, comme pour la fonction `factorielle` ci-dessous :

```
def factorielle(n):
    if n==0:
        return 1 # Cas de base
    else:
        return factorielle(n-1)*n # Appel récursif
```

Au début, cela peut paraître invraisemblable que cette fonction fonctionne. On a l'impression de "tricher" en appelant la fonction qu'on est en train de créer. Pourtant, cette fonction marche parfaitement :

```
>>> factorielle(4)
24
```

Cela n'a rien d'un miracle, voyons pourquoi. D'abord, il est clair que

```
>>> factorielle(0)
```

sera évalué à 1, puisqu'aucun appel récursif ne sera rencontré jusqu'au `return`. Il en découle que

```
>>> factorielle(1)
```

sera également évalué à 1, puisque la valeur retournée est `factorielle(0)*1` et que nous venons d'établir que `factorielle(0)` est évaluée à 1. Ceci implique à son tour que `factorielle(2)` sera évalué comme `2*1`, soit 2, et ainsi de suite. Par récurrence, on voit donc que `factorielle(n)` sera évaluée à $n \times (n-1) \times \ldots \times 1$. La Figure 6.2 ci-dessous illustre l'évaluation de `factorielle(4)` :

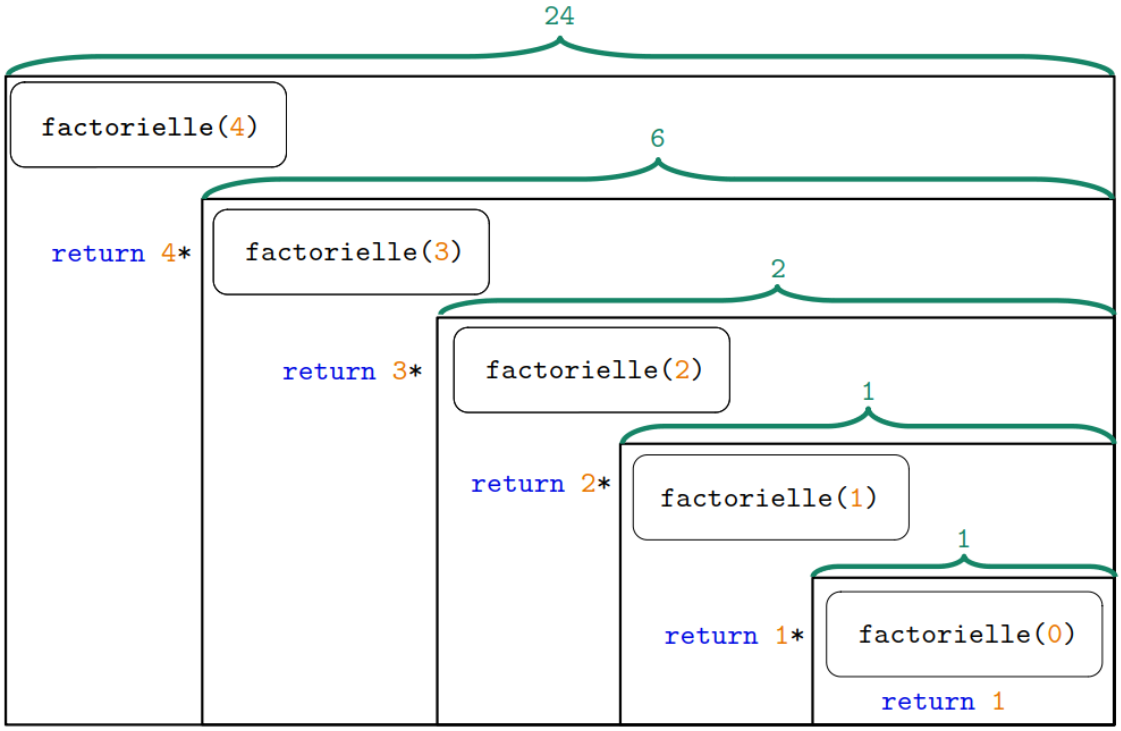


FIG. 6.2 – Exécution de la fonction `factorielle(4)`.

3 Complexité des fonctions récursives

Complexité de `factorielle`

La complexité d'une fonction récursive est souvent obtenue comme la solution d'une relation de récurrence. Par exemple, si l'on note $C(n)$ le nombre d'opérations effectuées par `factorielle` sur l'entrée n, nous avons

$C(0) = 0$ (aucune opération) et pour $n \geq 0$,

$$C(n+1) = C(n) + 1$$

car l'exécution de `factorielle(`$n+1$`)` nécessite l'exécution de `factorielle(`n`)`, soit $C(n)$ opérations, et la multiplication par n, soit une autre opération. Ainsi, $(C(n))_{n\in\mathbb{N}}$ est une suite arithmétique de pas 1, et on obtient donc $C(n) = n$ pour tout entier n. Cette fonction a donc la même complexité que la fonction itérative correspondante :

```
def factorielle_iterative(n):
    r = 1
    for i in range(1,n+1):
        r = r*i
    return r
```

mais la version récursive est bien plus élégante ![1]

Suite de Fibonacci

La suite de Fibonacci est définie par récurrence par $F_0 = 1$, $F_1 = 1$, puis, pour $n \geq 2$,

$$F_n = F_{n-1} + F_{n-2}.$$

Autrement dit, à partir du deuxième terme, chaque terme est obtenu en faisant la somme des deux précédents (les premiers termes sont donc 1, 1, 2, 3, 5, 8, 13, ...). On peut facilement créer une fonction récursive pour calculer les termes de cette suite :

```
def Fibonacci(n):
    if n==0:
        return 1 # Cas de base
    if n==1:
        return 1 # Cas de base
    if n>=2:
        return Fibonacci(n-1) + Fibonacci(n-2) # Appels récursifs.
```

Mais cette fonction, aussi élégante soit elle, a un défaut fatal : elle est catastrophiquement inefficace. Essayez par exemple (armez-vous de patience...)

```
>>> Fibonacci(35)
>>> Fibonacci(40)
>>> Fibonacci(45)
```

Cette ineffacité de `Fibonacci` est un véritable cas d'école d'une fonction récursive mal écrite. Il est fondamental de comprendre ce qui explique sa lenteur, et de voir comment on peut y remédier.

Pour commencer, calculons sa complexité : on pose $F(N)$ la quantité d'opérations nécessaires pour le calcul de `Fibonacci(N)`. On a alors $F(0) = 1$ (pour la comparaison `n==0`), $F(1) = 2$ (une comparaison supplémentaire `n==1`) et pour tout $n \geq 2$,

$$F(N) = F(N-1) + F(N-2) + 6$$

[1] En revanche, la limite sur le nombre d'appels récursifs fait que la version récursive de `factorielle` ne fonctionne que pour environ $n \leq 1000$ (ou environ $n \leq 3000$ dans la console). Il est possible de modifier manuellement ces limites, mais Python restera dans tous les cas un langage plutôt hostile à la récursion.

(le +6 étant dû aux trois comparaisons, les calculs de `N-1` et `N-2`, et l'addition de `Fibonacci(N-1)` et `Fibonacci(N-2)`). On a donc $F(N) = \Theta(\varphi^N)$ où $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618989...$ (pour démontrer cela, voir les exercices à la fin du chapitre). La fonction `Fibonacci` est donc de complexité **exponentielle** ! On peut s'en apercevoir en regardant le temps que met Python à exécuter `Fibonacci(n)` pour $n = 30, \ldots, 40$:

Valeur de `n`	30	31	32	33	34	35	36	37	38	39	40
Temps d'exécution (s)	0.4	0.6	1.0	1.7	2.8	4.6	7.4	12.2	19.4	32.3	51.5

TAB. 6.1 – Temps d'exécution de `Fibonacci(n)` pour $n = 30, \ldots, 40$.

Il faut attendre presque une minute pour connaître le 40-ième terme de la suite. En comparaison, voici une fonction itérative, peut-être moins élégante que la version récursive, mais elle, de complexité linéaire :

```
def Fibonacci_iterative(n):
    if n == 0 or n == 1:
        return 1
    i = 2
    A = 1
    B = 1
    C = A + B
    while i < n:
        # A,B,C = Fibo(i-2),Fibo(i-1),Fibo(i)
        i = i+1
        A = B
        B = C
        C = A + B
    return C
```

Le calcul de `Fibonacci_iterative(1000)` ne prend qu'une fraction de seconde... Alors, faut-il abandonner à jamais la récursion ? Sûrement pas : on ne quitte pas un tel paradis à la première déconvenue !

Pour mieux comprendre la raison pour laquelle `Fibonacci` effectue autant de calculs, nous allons ajouter un `print` en tête de la fonction pour afficher la valeur de n à chaque fois qu'elle est appelée :

```
def Fibonacci(n):
    print("Fibonacci appelée avec n =",n)
    if n==0 or n == 1:
        return 1
    return Fibonacci(n-1) + Fibonacci(n-2)
```

Voici l'affichage de `Fibonacci(5)` :

```
>>> Fibonacci(5)
 Fibonacci appelée avec n = 5
 Fibonacci appelée avec n = 4
 Fibonacci appelée avec n = 3
 Fibonacci appelée avec n = 2
 Fibonacci appelée avec n = 1
 Fibonacci appelée avec n = 0
 Fibonacci appelée avec n = 1
 Fibonacci appelée avec n = 2
```

```
Fibonacci appelée avec n = 1
Fibonacci appelée avec n = 0
Fibonacci appelée avec n = 3
Fibonacci appelée avec n = 2
Fibonacci appelée avec n = 1
Fibonacci appelée avec n = 0
Fibonacci appelée avec n = 1
```

On voit que contrairement à `factorielle`, l'ordre dans lequel `Fibonacci` procède est assez chaotique : le calcul de `Fibonacci(2)` est effectué 3 fois, celui de `Fibonacci(1)` est effectué 5 fois, etc. Au lieu de mémoriser ces résultats, ils sont recalculés de nombreuses fois : la complexité exponentielle `Fibonacci` s'explique donc tout simplement par le fait qu'elle passe presque tout son temps à recalculer des caleurs qu'elle a déjà obtenues.

Peut-on sauver Fibonacci ?

La section précédente illustre l'un des principaux dangers de la récursivité : écrire des programmes aussi élégants qu'inefficaces. Néanmoins il existe des tecuniques qui permettent, dans la plupart des cas, de "sauver" notre implémentation récursive. L'une d'elles est la *mémoïsation*, qui consiste à ajouter une mémoire dans laquelle on stocke les valeurs déjà calculées, pour éviter de les recalculer plus tard. Voici comment mettre en oeuvre cette idée pour la fonction `Fibonacci` :

```
def Fibonacci(n,mem):
    if mem[n] == None:
        # Valeur par encore calculée, on calcule le résultat pour la première fois
        if n == 0
            r = 1 # Cas de base
        if n == 1:
            r = 1 # Cas de base
        else:
            r = Fibonacci(n-1,mem) + Fibonacci(n-2,mem) # Appel récursif
        # On mémorise le résultat
        mem[n] = r
    else:
        # Valeur déjà calculée, on récupère le résultat dans la mémoire
        r = mem[n]
    return r

N = 100
# Création d'une mémoire de taille adaptée :
mem = [None]*(N+1)
Fn = Fibonacci(N,mem)
```

Cette fois, la complexité de `Fibonacci` est linéaire. En effet, après chaque appel récursif de `Fibonacci`, une nouvelle case du tableau `mem` est remplie. Or, il n'y a que N cases dans `mem`, donc il la ligne contenant les appels récursifs est exécutée au plus N fois. Ainsi, il y a au maximum $2N$ appels récursifs. Hormis ces appels récursifs, les opérations effectuées par `Fibonacci` sont toutes élémentaires. Le nombre total d'opérations est donc $O(N)$.

4 Problèmes à structure récursive

Considérons le problème suivant : écrire une fonction `sous_ensembles(k,n)` qui renvoie un tableau contenant tous les sous-ensembles de $\{1,\dots,n\}$ contenant k éléments. Chaque sous-ensemble sera représenté par un tableau trié contenant les éléments choisis. Par exemple :

```
>>> sous_ensembles(2,4)
[[1, 2], [1, 3], [2, 3], [1, 4], [2, 4], [3, 4]]
>>> sous_ensembles(4,5)
[[1, 2, 3, 4], [1, 2, 3, 5], [1, 2, 4, 5], [1, 3, 4, 5], [2, 3, 4, 5]]
```

Ce problème a une *structure récursive* : en effet, on peut ramener la résolution du problème à la résolution d'un problème "plus petit".

Pour commencer, on remarque que les sous-ensembles de $\{1,\dots,n\}$ à k éléments se divisent en deux groupes : les sous-ensembles ne contenant pas n, et les sous-ensembles le contenant. Les premiers sont sont les sous-ensembles de $\{1,\dots,n-1\}$ à k éléments, et les seconds sont ceux de la forme $E_{n-1}\cup\{n\}$ où E_{n-1} est un sous-ensemble de $\{1,\dots,n-1\}$ à $k-1$ éléments. On peut donc écrire

```
def sous_ensembles(k,n):
    # Sous-ensembles ne contenant pas n
    A = sous_ensembles(k,n-1)
    # Sous-ensembles contenant n :
    B = sous_ensembles(k-1,n-1)
    for i in range(len(B)):
        B[i].append(n)
    return A + B
```

(rappel : `A + B` renvoie la concaténation des tableaux `A` et `B`). Pour l'instant, ce programme ne fonctionne pas ; mais il ne reste plus qu'à définir les cas de base, ici $k=0$ et $k>n$:

```
def sous_ensembles(k,n):
    if k == 0:
        # Un seul sous-ensemble : l'ensemble vide
        ens_vide = []
        return [ens_vide]
    if k > n:
        # Aucun sous-ensemble
        return []
    # Sous-ensembles ne contenant pas n
    A = sous_ensembles(k,n-1)
    # Sous-ensembles contenant n :
    B = sous_ensembles(k-1,n-1)
    for i in range(len(B)):
        B[i] = append(B[i],n)
    return concat(A,B)
```

On a ainsi résolu un problème en deux étapes : (i) se ramener à des cas "plus petits", et (ii) résoudre explicitement des cas de base. Cette manière de fonctionner est très répandue en programmation.

5 Diviser pour régner

Non seulement la récursivité permet de résoudre élégamment des problèmes apparemment compliqués, mais en plus, elle mène parfois à découvrir des algorithmes très rapides, à l'aide d'une stratégie simple : *diviser pour régner*. L'idée est la suivante :

- (*Diviser*) : On **découpe** le problème en un ou plusieurs sous-problèmes de tailles à peu près égales,
- On résout chaque sous-problème en appliquant récursivement la même stratégie, (c'est-à-dire qu'ils seront divisés à leur tour, et ainsi de suite, jusqu'à atteindre un cas de base)
- (*Régner*) : On **combine** les solutions des sous-problèmes pour obtenir la solution du problème initial.

Voyons quelques exemples.

L'exponentiation rapide

Pour calculer a^n, on peut remarquer que si n est pair, disons $n = 2k$, alors $a^n = a^k \times a^k$. On a donc découpé le problème en deux morceaux identiques : calculer a^k. On combine les solutions simplement en les multipliant. Si n est impair, disons $n = 2k + 1$, alors la même idée s'applique cette fois en combinant par $a^n = a^k \times a^k \times a$. Le programme récursif suivant utilise cette idée :

```
def puissance(a,n):
    # Calcul rapide de a puissance n
    if n == 0:
        return 1
    else:
        # 1. Diviser !
        k = n//2
        # 2. On résout le sous-problème récursivement
        ak = puissance(a,k)
        # 3. Régner !
        # On recombine selon que n est pair ou impair:
        if n % 2 == 0:
            return ak*ak
        else:
            return ak*ak*a
```

Dans le cas où n est une puissance de 2, on peut voir facilement que la complexité $C(n)$ de l'algorithme vérifie $C(n) = \Theta(\log_2(n))$. En effet, pour tout $p \geq 1$,

$$C(2^p) = 3 + C(2^{p-1})$$

(le 3 est dû aux opérations non récursives, et le $C(2^{p-1})$, à l'appel de `puissance(a,k)`). Ceci se résout explicitement par

$$C(2^p) = 3p + C(2^0) = \Theta(p) = \Theta(\log_2(2^p))$$

On admet qu'en général, $C(n) = O(\log_2(n))$. En comparaison, la fonction itérative suivante

```
def puissance_iterative(a,n):
    r = 1
    for i in range(n):
        r*=a
    return r
```

est de complexité linéaire. Elle est donc beaucoup plus lente que la fonction précédente. Par exemple, pour le calcul de 2^{10^9}, seule la méthode récursive permet d'obtenir un résultat en un temps raisonnable

```
>>> a = puissance(2,10**9) # prend quelques secondes
>>> b = puissance_iterative(2,10**9) # nécessite plusieurs millieurs d'heures !
```

Recherche dichotomique dans un tableau trié

Supposons que `T` est un tableau trié dans l'ordre croissant, et que l'on souhaite déterminer si une portion du tableau $T[i], T[i+1], \ldots, T[j]$ contient l'élément `x`, et si oui, dans quelle case. Dans le cas particulier où $i = 0$ et $j = N - 1$, cela revient à chercher `x` dans tout le tableau, mais il est nécessaire de résoudre le problème un peu plus général pour tout $0 \leq i \leq j \leq N$ afin d'être en mesure d'appliquer la technique de diviser pour régner.

On a alors trois cas, en notant $k = \lfloor \frac{i+j}{2} \rfloor$ le milieu (ou presque) de la plage de cases

- Soit $x > T[k]$, dans ce cas il suffit de chercher `x` parmi les cases $T[k+1], \ldots, T[j]$
- Soit $x < T[k]$, dans ce cas il suffit de chercher `x` parmi les cases $T[i], \ldots, T[k-1]$
- Soit $x = T[k]$, auquel cas on peut immédiatement terminer l'algorithme.

On en déduit l'algorithme suivant :

```
def chercher(x,T,i,j):
    # Cherche x parmi T[i], T[i+1],...,T[j-1],T[j]
    # Renvoie k si T[k] = x, ou None si x n'est pas dans le tableau
    if j<i:
        # Cas de base
            return None
    else:
        k = (i+j)//2
        if x > T[k]:
            return chercher(x,T,k+1,j)
        elif x < T[k]:
            return chercher(x,T,i,k-1)
        else:
            return k
```

(comparez la simpilcité de ce programme avec celui que vous avez obtenu dans les exercices du Chapitre 5...). Si l'on note $C(N)$ la complexité de cet algorithme pour chercher `x` dans tout le tableau, on vérifie comme dans le paragraphe précédent que

$$C(2^n) = \Theta(n)$$

et en général, on admet que $C(N) = \Theta(\log_2 N)$, c'est-à-dire, une complexité logarithmique en fonction de la taille du tableau.

TP 6

Quelques fonctions récursives de base

1. Etant donné un paramètre a, soit $u_n = a^{a^{a^{\cdots^a}}}$ où la "tour" d'exposants contient n étages ($u_0 = 1$, $u_1 = a$, $u_2 = a^a$, $u_3 = a^{a^a}$, etc.). Créez une fonction récursive qui calcule le n-ième terme de cette suite. Calculer u_{500} pour différentes valeurs de a.

2. Créez une fonction récursive qui calcule

$$u_n = \sqrt{n + \sqrt{(n-1) + \sqrt{\ldots + \sqrt{2 + \sqrt{1}}}}},$$

puis une fonction récursive qui calcule

$$v_n = \sqrt{1 + \sqrt{2 + \sqrt{\ldots + \sqrt{n}}}}.$$

3* Les coefficients binomiaux $\binom{n}{p}$ ("p parmi n") pour p, n entiers, sont définis comme le nombre de façons différentes de choisir p éléments parmi une liste de n éléments. Ils vérifient la relation de récurrence

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

avec les cas de base $\binom{n}{0} = 1$, et $\binom{n}{p} = 0$ lorsque $p > n$. Créez une fonction récursive calculant $\binom{n}{p}$ pour tout p, n entiers. Cette fonction est-elle efficace ? Appliquez la technique du cours pour l'améliorer. Calculez $\binom{400}{200}$.

Calcul rapide de la suite de Fibonacci

Dans ce problème, on se propose de créer un algorithme rapide pour calculer le N-ième terme de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$. Nous prenons la même définition que dans le chapitre : $F_0 = F_1 = 1$, puis $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$.

4. (Rappel) Créez une fonction itérative `Fibo_iterative` qui réalise cette tâche. Quelle est sa complexité ?

5. Montrez que pour tout entier $n \geq 2$,

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

où A est une matrice carrée de taille 2×2 que l'on précisera (voir la section 3 de l'Annexe A).

6. En déduire que

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A^n \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

où A^n est définie par récurrence par $A^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ et $A^{n+1} = AA^n$.

7. Démontrez que pour tout $k, \ell \in \mathbb{N}$, $A^k A^\ell = A^{k+\ell}$ *(indice : procédez par récurrence sur k).*

8. En Python, on peut représenter une matrice 2×2 par un tableau de longueur 4. Les cases 0 et 1 contiennent la première ligne, et les cases 2 et 3, la seconde. Par exemple, selon cette convention, la matrice

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

être représentée par le tableau `[1,2,3,4]`. Créez une fonction `produit22(A,B)` qui reçoit deux tableaux `A` et `B` représentant des matrices de taille 2×2, et renvoie le tableau représentant la matrice AB.

9* Créez une fonction récursive `puissance22(A,n)` qui reçoit un tableau représentant une matrice de taille 2×2 et un entier n, et renvoie le tableau représentant la matrice A^n. On utilisera la stratégie *diviser pour régner*.

10. Calculez la complexité de `puissance22` en fonction de n. On se restreindra au cas où n est une puissance de 2.

11. Créez une fonction `mat22vec(A,v)` qui reçoit un tableau `A` représentant une matrice de taille 2×2 et un tableau `v` de longueur 2 représentant un vecteur $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$, et renvoie le tableau représentant le vecteur Av.

12. Créez une fonction `Fibonacci(n)` qui calcule le n-ième terme de la suite de Fibonacci. Commentez sa complexité.

13. Le texte numéro 2 de l'appendice est codé à l'aide du chiffrement de Vigenère, avec une clé de longueur 100, formée des 100 premiers chiffres du nombre `Fibonacci(10000000)` en base 26. Décodez ce texte !

Tri par fusion

Dans ce problème, on se propose de mettre en œuvre un exemple classique de la stratégie "diviser pour régner" : un algorithme rapide pour trier un tableau. Le principe est de trier récursivement chaque les deux moitiés du tableau, puis de fusionner les deux moitiés triées.

1. Créez une fonction `tableau_aleatoire(N1,N2)` qui crée un tableau de taille N_1 avec des valeurs aléatoires choisies uniformément entre 1 et N_2.

2. Créez une fonction `moities(T)` qui renvoie deux tableaux `T1` et `T2` contenant les $\lfloor N/2 \rfloor$ premières cases et les $N - \lfloor N/2 \rfloor$ dernières cases de `T`, respectivement.

3* Créez une fonction `fusion(T1,T2)` qui fusionne deux tableau triés en un tableau trié. Par exemple

```
>>> fusion([1,3,5],[2,4,6])
[1,2,3,4,5,6]
```

4. Quelle est la complexité de la fonction `fusion` en fonction de la taille des deux tableaux ?

5* Créez une fonction récursive `triFusion` qui trie un tableau.

6. Testez votre fonction sur un tableau de taille 10^5 avec des éléments aléatoires entre 1 et 10^6. Comparez le temps d'exécution avec le tri par insertion du Chapitre 3.

7* On note $C(N)$ le nombre d'opérations effectuées par l'algorithme de tri par fusion pour trier un tableau de taille N dans le pire des cas. Démontrez qu'il existe $K > 0$ tel que pour tout $n \in \mathbb{N}$,

$$C(2^{n+1}) \leq 2C(2^n) + K2^{n+1}$$

8* En raisonnant par récurrence, démontrez que pour tout entier n,

$$C(2^n) \leq 2^n C(1) + Kn2^n.$$

Déduisez-en que $C(N) = O(N \log N)$ pour N égal à une puissance de 2 (et on admet que le résultat est aussi vrai pour N quelconque). Commentez.

Permutations

On appelle *permutation* d'ordre N un tableau de longueur N contenant les entiers de 1 à N, chacun exactement une fois. Par exemple, $[3, 2, 5, 4, 1]$ est une permutation d'ordre 5. Le but de l'exercice est de créer une fonction qui liste toutes les permutations d'ordre N. *L'exercice est guidé, mais vous pouvez essayer de ne pas utiliser les indices et trouver vous-même la structure récursive du problème, en vous inspirant de l'exemple du cours concernant les sous-ensembles de taille k.*

1. Etant donné un tableau σ de longueur n et un entier i, on note $\sigma \vee i$ le tableau de longueur $n+1$ obtenu à partir de σ en ajoutant 1 à tous les nombres supérieurs ou égaux à i, puis en rajoutant une case en fin de tableau dans laquelle on met la valeur i. Par exemple

$$[2, 5, 3, 1, 4] \vee 3 = [2, 6, 4, 1, 5, 3].$$

Démontrez que si σ est une permutation d'ordre n, alors pour tout entier $i \leq n + 1$, $\sigma \vee i$ est une permutation d'ordre $n + 1$.

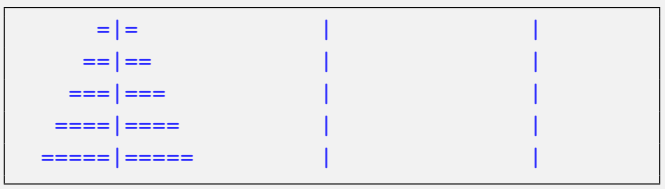
2. Réciproquement, montrez que toute permutation σ d'ordre $n + 1$ s'écrit de manière unique sous la forme $\sigma' \vee i$ avec σ' une permutation d'ordre n et $i \in \{1, \ldots, n + 1\}$.

3. Créez une fonction `v(sigma,i)` qui prend en argument un tableau `sigma` et un entier `i` et renvoie le tableau `sigma` $\vee$ `i` (attention à ne pas muter le tableau `sigma`).

4. Créez une fonction récursive qui liste toutes les permutations d'ordre n. Affichez toutes les permutations d'ordre $n = 6$.

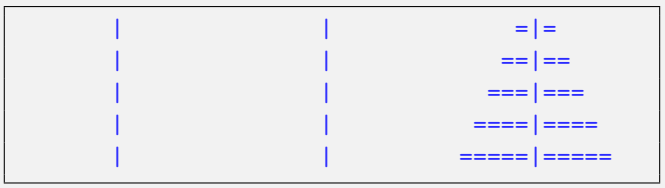
Les tours de Hanoi

Les tours de Hanoi sont un jeu dans lequel on dispose de trois piquets et de n disques percés de rayons repsectifs $1, 2, ..., n$. Pour $n = 5$, la configuration initiale peut être schématisée de la manière suivante :



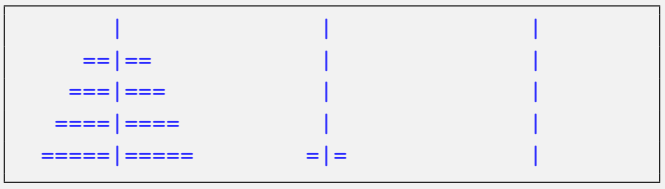
Position initiale pour $n = 5$

Les disques sont empilés du plus gros au plus petit, de bas en haut. Le but du jeu est d'arriver dans la position finale suivante :



Position finale pour $n = 5$

Pour ce faire, on peut à chaque coup, transférer le disque en haut d'une des piles au-dessus d'une autre pile, à condition de ne jamais poser un plus grand disque sur un plus petit disque. Par exemple, depuis la position initiale on peut déplacer le petit disque sur le deuxième piquet :



Coup valide

Mais ensuite, on ne peut pas déplacer le disque de rayon 2 sur le même piquet :

```
      |                |                |
      |                |                |
    ===|===            |                |
   ====|====         ==|==              |
  =====|=====         =|=               |
```

Coup invalide

On fournit un code Python, téléchargeable sur l'url

`https://martinaverseng.perso.math.cnrs.fr/Hanoi.py`

pour jouer à ce jeu. Il permet de créer et afficher des tours de Hanoi, et de déplacer les disques :

```
>>> H = Hanoi(5)
>>> H
 ---------------- Coup 0 :  ----------------


      =|=              |                |
     ==|==             |                |
    ===|===            |                |
   ====|====           |                |
  =====|=====          |                |

>>> H.deplacer(1,2)
 ------------ Coup 1 : 1 -> 2 -----------


        |               |                |
      ==|==             |                |
     ===|===            |                |
    ====|====           |                |
   =====|=====         =|=               |
```

1. Téléchargez le fichier `Hanoi.py` et placez-le dans un nouveau dossier. Dans ce même dossier, créez un deuxième fichier intitulé par exemple

   `resolutionHanoi.py`

2. Dans le fichier `resolutionHanoi.py`, écrivez les commandes suivantes :

```
from Hanoi import *
# importe le contenu de Hanoi.py
# Taper demo() pour un exemple d'utilisation.
n = 3
H = Hanoi(n)
print(H)
```

   et exécutez le fichier. Dans la console, déplacez le disque de rayon 1 sur le piquet 3 avec la méthode `deplacer`. Pour plus d'informations sur l'utilisation de la classe `Hanoi`, appelez `demo()` après avoir exécuté votre fichier.

3. Créez une fonction `resoudreHanoi(n)` qui résout le jeu pour un n arbitraire, en utilisant exclusivement la méthode `deplacer(i,j)` de la classe `Hanoi`. *(Indice : Ce problème a une structure récursive !)*
4. Déterminez le nombre minimal de coups nécessaires pour résoudre le problème avec n disques.
5. Déduisez de la question précédente la complexité de la fonction `resoudreHanoi` en fonction de n.

Suites récurrentes d'ordre 2

On se propose d'étudier les suites récurrentes de la forme

$$u_n = \begin{cases} x & \text{si } n = 0 \\ y & \text{si } n = 1 \\ au_{n-1} + bu_{n-2} + c. & \text{sinon.} \end{cases}$$

On suppose que $a \neq 0$, $1 - a - b \neq 0$ et $a^2 + 4b > 0$ (le cas général pourra être traité par les lecteurs souhaitant approfondir l'exercice).

1. On commence par éliminer la constante c. Montrez qu'en posant $v_n = u_n + K$ avec une constante K bien choisie, v_n vérifie

$$v_n = \begin{cases} x + K & \text{si } n = 0 \\ y + K & \text{si } n = 1 \\ av_{n-1} + bv_{n-2} & \text{sinon.} \end{cases}$$

2. Démontrez qu'il existe $r_1, r_2 \in \mathbb{R}$ deux nombres réels *distincts* vérifiant l'équation $x^2 = ax + b$. On les choisit de sorte que $|r_2| < |r_1|$ (pourquoi cela est-il possible ?)
3. Démontrez qu'il existe $\alpha, \beta \in \mathbb{C}$ tels que pour tout $n \in \mathbb{N}$,

$$v_n = \alpha r_1^n + \beta r_2^n.$$

4. Démontrez que $|u_n| = \Theta(r_1^n)$ si $\alpha \neq 0$.
5. Démontrez que la fonction $F(n)$ du cours (la complexité de la version récursive de `Fibonacci`) vérifie

$$F(n) = \Theta(\varphi^n)$$

où $\varphi = \frac{1+\sqrt{5}}{2}$.

Pavage en L

On considère un échiquier de 64 cases disposées en un carré de 8×8. L'une des cases de l'échiquier est choisie au hasard et coloriée en noir. Montrer que l'on peut paver le reste de l'échiquier avec des tuiles en forme de "L" (ce problème est illustré sur la page de couverture de ce document).

Chapitre 7

Classes

Dans les chapitres précédents, vous avez manipulé différents types (ou "classes", synonyme de "type" en Python) : `<int>`, `<flot>` etc. Vous avez sûrement remarqué que certains objets étaient plus adaptés que d'autres à certaines tâches. Pour la plupart des tâches que vous rencontrerez, le type parfaitement adapté n'existe pas encore dans Python : c'est à vous de le créer. C'est ce que nous allons apprendre à faire dans ce chapitre.

1 Définition d'une classe

Créons par exemple un type "`<magique>`". Pour l'ajouter à Python, on exécute un fichier contenant le code suivant :

```
class magique:
    pass
```

Le mot-clé `class` signale la définition d'un type, un peu comme `def` signale la définition d'une fonction. Pour l'instant, la définition de `magique` est vide. Rappel : l'instruction `pass` évite juste que Python renvoie une erreur à cause d'un bloc vide.

Après avoir exécuté ce fichier, on peut désormais créer des objets (on dit aussi des "instances") de notre nouveau type `<magique>` : essayez ceci

```
>>> a = magique()              # Création d'une "instance" de la classe <magique>.
>>> type(a)                    # Affichage du type de a
<class '__main__.magique'>
>>> isinstance(a,magique)      # a est un <magique> :
True
>>> isinstance(a,int)          # mais a n'est pas un <int> :
False
```

Nous ne nous soucierons pas du préfixe `__main__` que Python a ajouté au nom de notre type. La commande

```
>>> isinstance(a,t)
```

s'évalue à `True` si l'objet `a` est de type `t`, et `False` sinon. Comme les autres objets, notre objet magique vit dans la mémoire. Il prend très peu de place, car il ne contient presque aucune information. La seule chose à savoir sur lui pour l'instant, c'est son type : "`<magique>`".

2 Initialisation et affichage

Il est possible de personnaliser ce qui se passe au moment où un nouvel objet est créé. Pour cela, on crée une fonction nommée `__init__` (“initialisation”) que l’on place dans le bloc de définition de la classe. Par exemple, nous pouvons afficher un message dans la console à chaque création d’un objet de type `<magique>`.

```
class magique:
    def __init__(self):
        # Fonction pour personnaliser l'initialisation d'un objet
        print("Abracadabra ! Vous avez créé un objet magique.")
    # Fin de la définition de la classe <magique>
a = magique()  # Création d'une instance
b = magique()  # Création d'une instance
```

```
'Abracadabra ! Vous avez créé un objet magique.'
'Abracadabra ! Vous avez créé un objet magique.'
```

Le mot-clé `self` (qui veut dire “soi”, comme dans “soi-même”) est une variable qui fait référence au nouvel objet en cours d’initialisation. L’usage du mot `self` plutôt qu’un autre nom de variable est une coutume mais n’a rien d’obligatoire. On peut afficher notre nouvel objet de type `<magique>` dans la console :

```
>>> a
<__main__.magique object at 0x7aa96248c910>
```

mais cela n’est pas très engageant... Remédions tout de suite à cela, en définissant la fonction `__repr__` qui permet de personnaliser l’affichage d’une instance :

```
class magique:
    def __init__(self):
        # Fonction pour personnaliser l'initialisation d'une instance
        print("Abracadabra ! Vous avez créé un objet magique.")
    def __repr__(self):
        # Fonction pour personnaliser l'affichage
        # Retourne un <str> représentant notre objet
        return "Objet magique..."
    # Fin de la définition de "<magique>"
```

```
>>> a = magique()
'Abracadabra ! Vous avez créé un objet magique.'
>>> a
Objet magique...
```

Maintenant que nous savons créer de nouveaux types, il est temps d’aborder ce qui les rendent utiles :

(i) Les *attributs* (les informations que contient une instance)

(ii) Les *méthodes* (qui définissent le “comportement” d’une instance dans le programme).

3 Attributs

Pour l’instant, à part afficher des messages dans la console, nos objets `<magique>` ne servent à rien. De plus, ils ne contiennent aucune information spécifique : rien ne différencie un objet magique d’un autre. Or,

en général, en plus de donner un type à un objet, nous voulons aussi qu'il "contienne des informations". Par exemple, le type `int` contient une information, à savoir, l'entier qu'il représente. Pour ajouter des informations à nos objets, on utilise des *attributs*.

Définition 7.1 : Attributs

Les *attributs* sont des variables liées à une instance d'une classe, et qui décrivent son état.

En général, les valeurs des attributs sont initialisées dans la fonction `__init__`. Essayez cet exemple :

```
class carte:
    def __init__(self):
        # Attributs : rang et couleur
        self.rang = "Dame"
        self.couleur = "Coeur"
    def __repr__(self):
        # Affichage qui dépend de la valeur des attributs
        return "Carte :" + self.rang + " de " + self.couleur
```

```
>>> C = carte()
>>> C
Carte : Dame de Coeur
```

Dans le programme ci-dessus, nous avons défini un nouveau type `<carte>` avec deux attributs : *rang* (As, 2, 3, ..., Valet, Dame, Roi) et *couleur* (Pique, Coeur, Carreau ou Trèfle). Puis nous avons créé une instance de notre classe. On peut se représenter mentalement notre instance comme ceci :

<carte>	
rang :	<str> 'Dame'
couleur :	<str> 'Coeur'

Pour *accéder* aux attributs d'un objet, on utilise la syntaxe `objet.nom_attribut` :

```
>>> C.rang
'Dame'
>>> C.couleur
'Coeur'
```

L'accès à la valeur d'un attribut est à rapprocher de l'accès à une case d'un tableau. C'est un peu comme si, au lieu de désigner une case par un numéro (`T[i]`), on utilisait plutôt un mot (`C.rang`). Comme pour les tableaux, on peut aussi modifier la valeur d'un attribut : essayez ceci

```
>>> C.rang = "Valet"
>>> C
Valet de Coeur
```

Nous venons de “muter” notre objet, en modifiant la valeur d'un de ses attributs. Les types que vous créerez en Python seront toujours mutables par défaut (voir la Section 4 du Chapitre 3)

Pour initialiser les attributs de nos instances, on les passe généralement en arguments de la fonction `__init__` comme ceci :

```
class carte:
    def __init__(self,rg,clr):
        # Attributs : rang et couleur
        self.rang = rg
        self.couleur = clr
```

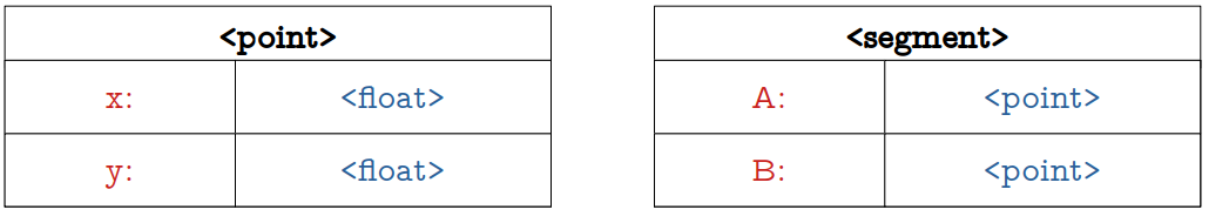
```
>>> C1 = carte("Valet","Trèfle")
>>> C2 = carte("Dame","Pique")
>>> C1
Carte : Valet de Trèfle
>>> C2
Carte : Dame de Pique
```

Ainsi, quand on écrit `carte(arg1,arg2)`, ceci a pour effet de créer un objet de type carte et d'appeler la fonction `__init__` avec les arguments `self` (la carte qui vient d'être créée), `arg1` et `arg2`.

Prenons un autre exemple en créant un type `point` et un type `segment`.

```
class point:
    def __init__(self,x,y):
        self.x = x # abscisse
        self.y = y # ordonnée
    def __repr__(self):
        # Renvoie (x,y)
        return "(" + str(self.x) +  "," + str(self.y) + ")"

class segment:
    def __init__(self,A,B):
        # Vérification que A et B sont de type point
            self.A = A
            self.B = B
    def __repr__(self):
        return "Segment : "+ str(self.A) + " <----> " + str(self.B)

M1 = point(0.0,0.0)
M2 = point(1.0,1.0)
S = segment(M1,M2)
```

```
>>> M1
(0.0,0.0)
>>> M2
(1.0,1.0)
>>> S
Segment : (0.0,0.0) <----> (1.0,1.0)
```


<point>
x:
<float>
y:
<float>

A:
<point>
B:
<point>

FIG. 7.1 – Une instance de `<point>` et de `segment`.

Dans cet exemple, les points ont deux attributs (des nombres représentant leurs coordonnées x et y, prévus pour être de type `<float>`), et les segments ont deux attributs : les deux extrémités, notées A et B, prévus pour être de type `<point>`. Ainsi, la valeur d'un attribut peut très bien avoir un type personnalisé. Pour connaître la coordonnée x de l'extrémité A du segment, on peut simplement écrire

```
>>> S.A.x
0.0
```

Protection des attributs

Supposons que l'on définisse une classe représentant des cercles, avec deux attributs, un centre et un rayon :

```
class cercle:
    def __init__(self,ctr,r):
        self.centre = ctr
        self.rayon = r
    def __repr__(self):
        return "Cercle de centre "+str(self.centre)+" et de rayon "+str(self.rayon)
```

Pour l'instant, malgré leurs noms, les attributs "`centre`" et "`rayon`" peuvent contenir n'importe quel type de variable. Un petit malin pourrait créer un cercle comme celui-ci :

```
>>> C = cercle("Thonny",-1)
>>> C
 Cercle de centre Thonny et de rayon -1
```

Il serait rassurant de pouvoir garantir que quand un cercle est créé, ou quand ses attributs sont modifiés, le centre est toujours un `<point>`, et le rayon est toujours positif. Pour ce faire, on peut utiliser une méthode spéciale, `__setattr__` ("set attribute" : assigner attribut). À chaque fois que l'on assigne ou réassigne un attribut de notre classe, c'est cette méthode qui sera appelée. C'est l'occasion de vérifier que les arguments respectent les conditions que nous souhaitons, et de déclencher une erreur dans le cas contraire.

```
class cercle:
    def __init__(self,ctr,r):
        self.centre = ctr # Appelle __setattr__(self,"centre",ctr)
        self.rayon = r    # Appelle __setattr__(self,"rayon",r)
```

```
    def __setattr__(self,name,value):
        # Vérification des conditions :
        if name == "centre":
            if not isinstance(value,point):
                raise Exception("Le centre doit être un point.")
        if name == "rayon":
            if value < 0:
                raise Exception("Le rayon doit être positif")
        # Si tout s'est bien passé, on peut alors faire l'assignation :
        super().__setattr__(name,value)
```

La syntaxe "raise `Exception`(message)" nous permet de déclencher volontairement une erreur. Même si cela peut paraître étonnant, il est très utile de déclencher soi-même des erreurs dans un programme : en effet, à votre avis, mieux vaut-il que le programme s'exécute sans vous avertir que votre cercle a un rayon négatif, ou préférez-vous être prévenu dès que cela arrive ?

La dernière ligne contenant la commande `super()` se charge d'effectuer l'assignation une fois que les tests ont été faits. Inutile d'essayer de comprendre cette étrange syntaxe, cela nous emmènerait trop loin. Il faut admettre que c'est comme ça que cela fonctionne.

```
>>> C = cercle("Thonny",1) # Mauvais centre : erreur !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 10, in __init__
  File "<string>", line 18, in __setattr__
Exception: Le centre doit être un point.
>>> C = cercle(point(0,0),1) # Arguments OK, pas d'erreur.
>>> C.rayon = -1 # Mauvaise réassignation : erreur !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 21, in __setattr__
Exception: Le rayon doit être positif
```

4 Méthodes

Nous venons de voir que les attributs sont ce qui, au sein d'une même classe, permettent de différencier une instance d'une autre. Dans ce paragraphe, nous abordons le deuxième aspect fondamental d'une classe : les *méthodes*, qui sont les choses que toutes les instances de votre type "savent faire".

Définition 7.2 : Méthode

Les *méthodes* sont des fonctions spécifiques de la classe qui peuvent être appliquées à n'importe quelle instance.

Par exemple, créons une méthode pour la classe `<segment>`, qui calcule la *longueur* d'une instance :

```
class segment:
    def __init__(self,A,B):
```

```
        self.A = A
        self.B = B
    def longueur(self):
        """Renvoie la longueur du segment"""
        # On commence par accéder aux coordonnées des extrémités :
        xA = (self.A).x  # Attribut 'x' du point (self.A)
        yA = (self.A).y  # Les parenthèses sont superflues.
        xB = (self.B).x
        yB = (self.B).y
        l2 = (xB - xA)**2 + (yB - yA)**2
        return l2**0.5  # Racine carrée.
```

Et voilà ! Maintenant, tous les objets de type `<segment>` "savent" calculer leur propre longueur :

```
>>> M1 = point(0,0)
>>> M2 = point(1,1)
>>> S = segment(M1,M2)
>>> S.longueur() # On demande à S de nous dire sa longueur
1.4142135623730951
```

Cette syntaxe vous rappelle peut-être quelque chose : c'est la même que pour ajouter ou retirer des cases à un tableau, `T.append(a)` et `T.pop()`. En effet, les tableaux (ou `<list>`) sont eux-même un type, ou une classe, et `append` et `pop` sont des méthodes de cette classe.

Plus généralement, les méthodes obéissent à la syntaxe suivante :

```
class ma_classe:
    def __init__(self):
        pass
    def __repr__(self):
        pass
    def ma_methode(self,arg1,...argN):
        # Définition de la méthode comme une fonction
        # avec self premier argument obligatoire
        # ...
        return r1,r2,...,rM

A = ma_classe()
arg1 = ...
arg2 = ...
...
argN = ...
# La méthode est appelée comme ceci :
r1,r2,...,rM = A.ma_methode(arg1,...,argN)
# Seuls les arguments autres que self sont mis entre parenthèse
```

Les méthodes ont toujours un premier argument obligatoire, que l'on appelle généralement `self`, suivi d'arguments optionnels. Le nom `self` sert à rappeler que cette méthode va s'appliquer à l'instance sur laquelle on appelle la méthode. Pour appeler la méthode, on utilise la syntaxe

`<instance>.<nom_de_la_methode>(arg1,...,arN)`

C'est l'instance "elle-même" ("itself" en anglais) qui prend le rôle de l'argument "`self`" lors de l'exécution de la méthode.

À part cela, une méthode est exactement comme une fonction normale. Elle peut renvoyer un résultat, ou ne rien renvoyer mais afficher quelque chose, ou modifier un objet mutable (souvent, l'instance elle-même). Par exemple, voici une méthode de la classe `<point>` qui applique une translation :

```
class point:
    def __init__(self,x,y):
        ...
    def __repr__(self):
        ...
    def translater(self,dx,dy):
        self.x = self.x + dx
        self.y = self.y + dy
        return None
```

```
>>> M = point(1.0,2.0)
>>> M
(1.0,2.0)
>>> M.translater(1.5,0.2)
>>> M
(2.5,2.2)
```

Maintenant que cette méthode est définie, il est très facile de créer une méthode pour translater un segment : il suffit de translater ses extrémités :

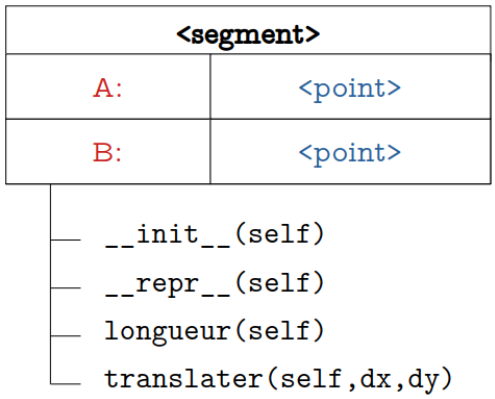
```
class segment:
    def __init__(self,A,B):
        self.A = A
        self.B = B
    def __repr__(self):
        ...
    def longueur(self):
        ...
    def translater(self,dx,dy):
        (self.A).translater(dx,dy)
        (self.B).translater(dx,dy)
        return None

M1 = point(0,1)
M2 = point(2,3)
S = segment(M1,M2)
print("Avant translation :",S)
S.translater(0.5,0.7)
print(S)
print("Après translation :",S)
```

```
>>>
 Avant translation : Segment (0,1) <----> (2,3)
```

```
Après translation : Segment (0.5,1.7) <----> (2.5,3.7)
```

Remarque : Attention, après la translation, les variables `M1` et `M2` ont aussi changé de valeur (pourquoi ?).

Quand on définit une classe, on peut représenter la liste de ses attributs et de ses méthodes, pour visualiser à quoi elle sert, voir la Figure 7.2 pour la classe `segment`.



A:
<point>
B:
<point>
__init__(self)
__repr__(self)
longueur(self)
translater(self,dx,dy)

FIG. 7.2 – Représentation de la classe segment.

Méthodes “magiques”

D'après ce que nous venons de voir, la syntaxe pour appeler une méthode est

```
>>> objet.methode(arg1,...,argN)
```

Imaginons que nous avons créé une nouvelle classe et que nous définissons une opération `addition`. On aimerait pouvoir écrire `A + B` au lieu de `A.addition(B)`, non ? Eh bien c'est possible ! En Python, l'utilisation de méthodes avec une syntaxe d'appel spéciale s'appellent les méthodes “magiques”. Prenons un exemple : nous allons définir une classe `Cappuccino` dont les attributs sont la quantité de lait et de café :

```
class cappuccino:
    def __init__(self,cafe,lait):
        self.cafe = cafe # quantité de café
        self.lait = lait # quantité de lait
    def __repr__(self):
        volume = self.cafe + self.lait
        p_cafe = self.cafe/volume*100
        p_lait = self.lait/volume*100
        s = "Cappuccino\n"
        s += "  Café : " + str(p_cafe) + "%\n"
        s += "  Lait : " + str(p_lait) + "%"
        return s
```

```
>>> C = cappuccino(400,100)
>>> C
```

```
Cappuccino
   Café : 80%
   Lait : 20%
```

Nous allons utiliser une méthode magique pour mélanger deux cappuccini :

```
class cappuccino:
    def __init__(self,cafe,lait):
        ...
    def __repr__(self):
        ...
    def __add__(self,other):
        cafe_total = self.cafe + other.cafe
        lait_total = self.lait + other.lait
        return cappuccino(cafe_total,lait_total)
```

Notre méthode `__add__` peut être appelée comme n'importe quelle méthode normale :

```
>>> A = cappuccino(400,100)
>>> B = cappuccino(200,200)
>>> C = A.__add__(B)
>>> C
Cappuccino
    Café : 66.66666666666666%
    Lait = 33.33333333333333%
```

Mais le fait d'avoir choisi spécifiquement le nom `__add__` nous permet d'utiliser aussi la syntaxe

```
>>> C = A + B # équivalent à C = A.__add__(B)
>>> C
Cappuccino
    Café : 66.66666666666666%
    Lait = 33.33333333333333%
```

Outre l'addition, il existe plusieurs autres méthodes magiques disponibles : en voici quelques-unes.

Opérateur	Nom magique	Syntaxe de l'appel
+	__add__(x,y)	x + y
-	__sub__(x,y)	x - y
-	__neg__(x)	-x
*	__mul__(x,y)	x * y
/	__truediv__(x,y)	x / y
//	__floordiv__(x,y)	x // y
%	__mod__(x,y)	x % y
>	__gt__(x,y)	x > y
>=	__ge__(x,y)	x >= y
<	__lt__(x,y)	x < y
<=	__le__(x,y)	x <= y
==	__eq__(x,y)	x == y
!=	__ne__(x,y)	x != y
()	__call__(f,x)	f(x)
[]	__getitem__(A,i)	A[i]
in	__contains__(E,x)	x in E

TAB. 7.1 – Quelques méthodes magiques et leurs syntaxe. D'autres exemples peuvent être trouvés dans la documentation officielle de Python.

Exercices du Chapitre 7

Manipulations de base

1. Créez un type `<vip>` et créez une fonction `soiree(x)` qui reçoit un argument `x` et renvoie une erreur si l'argument n'est pas de type `<vip>`. (*Utiliser* `isinstance` *et* `raise`).

2. En reprenant les classes `point` et `segment` du cours, créez un objet de type `point` et un objet de type `segment` et affichez leurs attributs dans la console.

3. Ajoutez une classe `triangle` avec des fonctions `__init__` et `__repr__` appropriées (sans chercher à tracer le triangle). Créez un objet de type triangle et affichez-le dans la console

4. Ajoutez les trois méthodes suivantes à la classe `triangle` :
   (a) `perimetre(self)` qui renvoie le périmètre du triangle,
   (b) `diametre(self)` qui renvoie le plus grand côté du triangle,
   (c) `centre(self)` qui renvoie un `<point>` représentant le centre de gravité du triangle.

   Appelez ces trois méthodes.

Paquet de cartes

5. Reprenez la classe `carte` du cours et créez une classe `paquet` qui représente un paquet de cartes. Dans la méthode `__init__`, on prendra en argument un tableau d'objets de type `carte`, qu'on assignera à un attribut `cartes`. Créez une méthode `taille(self)` qui renvoie la taille du paquet.

6. Créez un paquet de carte contenant les 4 As.

7. Créez une fonction `__repr__` qui affiche le nombre de cartes du paquet et les liste dans l'ordre. On pourra utiliser le caractère `\n` pour le retour à la ligne. Appelez cette méthode sur le paquet précédent.

8. Créez une fonction `toutes_les_cartes` qui ne prend aucun argument et renvoie un tableau de taille 52 contenant toutes les cartes à jouer. On pourra définir des tableaux `rangs` et `couleurs` comme suit

```
rangs=["As","2",...,"Valet","Dame","Roi"]
couleurs=["Pique","Coeur","Carreau","Trèfle"]
```

   puis ajouter `carte(rangs[i],couleurs[j])` pour chaque $0 \leq i < 13$ et $0 \leq j < 4$.

9* Créez une méthode `sous_paquet(self,i,j)` qui renvoie le paquet composé des cartes en position $i, i+1, ..., j$, avec $1 \leq i \leq j \leq N$, où N est la taille du paquet.

10. Créez une méthodes `couper` qui mute le paquet en le "coupant" à un endroit aléatoire. On utilisera le module `random` pour décider où le paquet est coupé. On se servira de la méthode `sous_paquet`.

11. Créez une méthode `melange_americain` qui mélange le tas de cartes de la manière suivante : on coupe le paquet en deux moitiés, et on crée le nouveau paquet en alternant les cartes de l'une ou l'autre moitié, à chaque fois en tirant au sort. Lorsqu'un paquet est épuisé, on ajoute alors toutes les cartes restantes de l'autre paquet (faire un schéma !)

12. Appelez la méthode `melange_americain` plusieurs fois sur un paquet de 52 cartes. Au bout de combien de mélanges le paquet vous semble-t-il "bien mélangé" ? Quelle serait selon vous une définition d'un "bon mélange" ?

Classe `<fraction>` et calcul numérique

1. Créez une classe `fraction` qui contient les attributs `p` pour le numérateur, et `q` pour le dénominateur (deux entiers, avec `q` $\neq 0$). Créer des fonctions `__init__` et `__repr__` appropriées.

2. À l'aide de l'algorithme d'Euclide, créer une méthode `reduire(self)` qui remplace un objet de type fraction par son équivalent irréductible. Testez cette fonction y compris sur des fractions négatives. Ajoutez un appel à cette méthode dans la méthode `__init__`.

3. Créez une méthode `taille(self)` qui renvoie la "taille" de la fraction, définie comme la somme du nombre de chiffres du numérateur et du dénominateur.

4. Créez une méthode `expansion_decimale(self,N)` qui renvoie deux tableaux `T_int` et `T_frac`, où `T_int` contient les chiffres de la partie entière de la fraction, et `T_frac` contient les N premiers chiffres de l'expansion décimale tronquée sans arrondir. *(Indice : multiplier par* 10^N*)*

5. Créez une méthode `print_decimales(self,N)` qui affiche l'expansion décimale tronquée à N décimales. Affichez l'expansion décimale de 22/7 avec 20 chiffres après la virgule.

6. Créez des méthodes magiques pour les opérations entre fractions (addition, soustraction, multiplication, division, puissance entière), ainsi que pour la valeur absolue (`__abs__`, appelée avec `abs(x)`).

7. Créez une méthode magique pour l'égalité (symbole `==`) de deux fractions (utilisez `__eq__`). Créez de même des fonctions `__lt__` (*lesser than*), `__gt__` (*greater than*), `__le__` (*lesser or equal*) et `__ge__` (*greater or equal*), pour les opérateurs `<`, `>`, `<=` et `>=`, respectivement.

8. *(Optionnel, mais pratique pour la suite)* En mettant une condition sur le type de l'argument `other` dans vos fonctions précédentes, étendez les opérations arithmétiques entre `<fraction>` et `<int>`. Par exemple, les variables `a` et `b` ci-dessous

```
>>> a = fraction(1,2) + 1
>>> b = 3*fraction(1,2)
```

doivent être des `<fraction>` représentant $\frac{3}{2}$. Quand la fraction se trouve à droite de l'opérateur, comme dans le cas de `b`, il faudra utiliser la version `__r<operateur>__` de chaque opérateur. Par exemple, `__radd__` pour l'addition d'une fraction à droite.

9. Soit $S \in \mathbb{Q}$ avec $S > 1$. On note $(x_n)_{n\in\mathbb{N}}$ la suite définie par

$$x_0 = S, \quad x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

(c'est la méthode de Héron d'Alexandrie pour l'approximation de $\sqrt{S}$). Montrer que pour tout $n \in \mathbb{N}$, $x_n \in \mathbb{Q}$ et $\sqrt{S} \leq x_n \leq S$. En déduire que $(x_n)_{n\in\mathbb{N}}$ est décroissante, puis que $\lim_{n\to\infty} x_n = \sqrt{S}$.

10. Soit $\varepsilon > 0$. Montrer qu'il existe $N \in \mathbb{N}$ tel que $|x_N^2 - S| < \varepsilon$. Montrer que pour cette valeur de N,

$$|x_N - \sqrt{S}| < \varepsilon.$$

11. Créez une méthode `approx_sqrt(self,eps)` qui, pour un objet représentant un rationnel $S > 1$, et un rationnel `eps` représentant une petite quantité $\varepsilon > 0$, renvoie deux objets r_1 et r_2 de type `fraction`, avec $|r_1 - r_2| \leq \varepsilon$ et $r_1 \leq \sqrt{S} \leq r_2$. Calculez la 20000-ème décimale de $\sqrt{2}$.

12. L'approche précédente possède un défaut pratique majeur pour le calcul numérique. Pour l'illustrer, on considère la suite $(r_n)_{n\in\mathbb{N}}$ la suite définie par $r_0 = 2$ et $r_{n+1} = \sqrt{r_n + 2}$. Calculez des encadrements par des fractions à 10^{-1} près de $r_0, r_1, \ldots, r_6$ à l'aide de la méthode précédente et affichez la taille des fractions correspondantes. Selon vous, quel est le problème ?

13. Imaginez et mettez au point une solution qui permette de calculer une approximation de r_N en $O(N)$ opérations avec une précision arbitraire.

Chapitre 8

Listes et arbres

Prochainement disponible !

Annexe A

Rappels mathématiques

1 Suites et séries particulières

1.1 Somme des entiers entre 1 et n

La somme des entiers de 1 à n vaut

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}.$$

En effet, si l'on note $S = 1 + 2 + \ldots + n$ cette somme, en écrivant

$$\begin{array}{ccccccccccc} & 1 & + & 2 & + & 3 & + & \ldots & + & n \\ + & n & + & (n-1) & + & (n-2) & + & \ldots & + & 1 \\ \\ = & (n+1) & + & (n+1) & + & (n+1) & + & \ldots & + & (n+1) \end{array}$$

on s'aperçoit que

$$2S = \underbrace{(n+1) + (n+1) + \ldots + (n+1)}_{n \text{ fois}} = n(n+1) \implies S = \frac{n(n+1)}{2}.$$

1.2 Série géométrique

Nous avons besoin à plusieurs reprises de calculer des sommes du type

$$1 + 2 + 4 + 8 + 16 + \ldots \quad \text{ou} \quad 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots$$

(chaque terme est ici obtenu en doublant ou en prenant la moitié du précédent). Plus généralement, si $a \in \mathbb{R}$ est un nombre réel et $n \in \mathbb{N}$ un entier, on explique ici comment calculer la somme

$$1 + a + a^2 + \ldots + a^n \,,$$

où $a^k := \overbrace{a \times a \times \ldots \times a}^{k \text{ termes}}$. Chaque nouveau terme est obtenu en multipliant par a le précédent : c'est ce qu'on appelle une progression *géométrique* (les deux cas précédents sont des cas particuliers pour $a = 2$ et $a = \frac{1}{2}$). Il y a quelque cas évidents, comme $a = 0$ (dans ce cas, la somme vaut 1) et $a = 1$ (la somme est alors n). Dans les cas où $a \neq 1$, il existe une méthode astucieuse pour trouver la valeur de $S = 1 + a + \ldots + a^n$. On remarque en effet que

$$a \times S = a \times (1 + a + \ldots + a^n) = a + a^2 + \ldots + a^{n+1}$$

puisque $a \times a^k = a^{k+1}$. En comparant $a \times S$ et S, on s'aperçoit que la différence est maigre : il manque le 1 et il y a un a^{n+1} en trop. Autrement dit, en posant la soustraction comme ceci,

$$\begin{array}{llllllllllll} & S & 1 & + & a & + & a^2 & + & \ldots & + & a^n & \\ - & a \times S & & & a & + & a^2 & + & \ldots & + & a^n & + & a^{n+1} \\ \hline & & 1 & + & 0 & + & 0 & + & \ldots & + & 0 & - & a^{n+1} \end{array}$$

on constate que $S - aS = 1 - a^{n+1}$. En factorisant le membre de gauche par S et en divisant par $(1-a)$ de part et d'autre cela donne $S = \frac{1-a^{n+1}}{1-a}$. Ceci établit donc la formule

$$1 + a + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a} \quad \text{pour tout } n \in \mathbb{N} \text{ et } a \neq 1. \tag{A.1}$$

(notez que la division par $1 - a$ dans la dernière étape n'est permise que si $a \neq 1$). Dans le cas où $a = 2$, on trouve donc par exemple, pour $n = 9$,

$$1 + 2 + 4 + \ldots + 2^9 = \frac{1 - 2^{10}}{1 - 2} = 2^{10} - 1 = 1023$$

Lorsque $a \in]-1, 1[$, la quantité a^n tend vers 0 lorsque n tend vers l'infini, et donc d'après (A.1),

$$\lim_{n \to \infty} (1 + a + \ldots + a^n) = \frac{1}{1 - a}.$$

Par exemple, pour $a = \frac{1}{2}$, on a

$$\lim_{n \to \infty} (1 + \frac{1}{2} + \ldots + \frac{1}{2^n}) = \frac{1}{1 - \frac{1}{2}} = 2.$$

Cette formule peut se retenir facilement en repensant au paradoxe de Zénon : si vous devez parcourir une distance de 2 mètres, vous devez d'abord en parcourir la moitié (1 mètre) puis la moitié de ce qu'il reste ($\frac{1}{2}$ mètres) et ainsi de suite (si bien, d'après Zénon, que vous n'atteindrez jamais votre destination, même si vous vous en approcherez aussi près que vous voulez).

2 Principe de récurrence

2.1 Démonstration par récurrence

Imaginez que vous êtes un espion, et que vous pénétrez dans le quartier général ennemi. Celui-ci est organisé en plusieurs compartiments, du moins sécurisé au plus sécurisé. Les plans de l'arme dévastatrice que vos ennemis viennent de mettre au point se trouvent dans la chambre forte, qui est le dernier compartiment, le plus sécurisé de tous. Votre but est de les dérober.

Vous ne savez pas combien de compartiments il y a au total : tout ce que vous savez, c'est que vous avez réussi à entrer. Vous vous trouvez actuellement dans le premier, le moins sécurisé. De plus, vous disposez d'une clé secrète confectionnée par votre technicien qui permet d'ouvrir n'importe quel compartiment *à condition d'avoir atteint le compartiment précédent*. Vous vous dirigez vers le deuxième compartiment, et... *clic* ! ça s'ouvre, votre clé fonctionne !

Le principe de récurrence vous dit ici que vous pourrez bien atteindre la chambre forte, et au passage, n'importe quel compartiment. En effet, (i) vous êtes entrés et (ii) vous disposez d'un moyen pour passer de n'importe quel compartiment au suivant.

Le même principe est utilisé en mathématiques pour démontrer une suite de propositions $P(0), P(1), P(2), \ldots$. Au lieu de démontrer chacune d'entre elles individuellement, on peut les prouver toutes à la fois à l'aide des deux ingrédients suivants. On démontre

(i) Initialisation : que la première proposition, $P(0)$, est vraie (vous savez *entrer* dans le premier compartiment du quartier général ennemi)

(ii) Hérédité : que si l'une des propositions est vraie, alors la suivante aussi (votre *clé secrète* fonctionne).

Le premier exemple ci-dessous est volontairement simple, au risque de sembler un peu évident. Le but est d'isoler la difficulté sur le raisonnement par récurrence en lui-même, sans ajouter de difficulté mathématique supplémentaire.

Exemple : Montrez que $n < 2^n$ pour tout entier $n \in \mathbb{N}$.

Réponse : Pour chaque entier $n \in \mathbb{N}$, appelons $P(n)$ la proposition : " $n < 2^n$ ".

(i) La proposition $P(0)$ ($0 < 2^0$) est bien sûr vraie ce qui établit l'initialisation (nous sommes entrés dans le quartier général ennemi !)

(ii) Si l'une des propositions $P(k)$ est vraie, alors la suivante, $P(k+1)$, l'est aussi. En effet, si $k < 2^k$, alors en ajoutant 1 de part et d'autre, et en utilisant le fait que $1 \leq 2^k$,[1] *on obtient*

$$k + 1 < 2^k + 1 \leq 2^k + 2^k = 2 \times 2^k = 2^{k+1}.$$

Ainsi, $k+1 < 2^{k+1}$, ce qui veut dire exactement que $P(k+1)$ est vraie. Nous venons donc de montrer que si $P(k)$ est vraie, alors $P(k+1)$ aussi, ce qui établit l'hérédité (notre clé secrète fonctionne !)

D'après le principe de récurrence, nous pouvons conclure que toutes les propositions $P(0), P(1), P(2), \ldots$ sont vraies. Autrement dit, $n < 2^n$ pour tout entier $n \in \mathbb{N}$.

Exemple : Vous organisez un tournoi dans votre club de ping-pong, dans lequel tous les membres jouent un match contre tous les autres. Montrez qu'il est possible d'établir un classement ayant la propriété que chaque joueur a gagné contre le joueur juste en-dessous de lui dans le classement.

Réponse : Appelons $P(n)$ la proposition "Un tel classement existe pour n participants". Nous allons montrer que toutes les propositions $P(2), P(3), P(4) \ldots$ sont vraies en utilisant le principe de récurrence.

(i) Initialisation : La proposition $P(2)$ est bien sûr vraie, car il n'y aura qu'un seul match. Il suffit d'inscrire le nom du gagnant en haut du classement, et celui du perdant, en bas.

(ii) Hérédité : Si l'une des proposition $P(k)$ est vraie, alors la suivante, $P(k+1)$, l'est aussi. En effet, supposons que notre tournoi ait $k+1$ participants. Nous choisissons l'un d'eux, que nous appellerons X, et que nous mettons temporairement de côté. Si $P(k)$ est vraie, c'est que nous pouvons classer les k participants restants comme demandé ; pour en déduire que $P(k+1)$ est vraie, il reste à démontrer que l'on peut insérer X dans le classement ainsi obtenu. Pour ce faire, il suffit d'écrire le nom de X juste au-dessus de la personne la mieux classée qu'il a battue, ou tout en bas s'il n'a remporté aucun match. Par exemple, si X a perdu contre les 1er, 2ème et 3ème du classement, mais gagné contre le quatrième, on l'insère entre les 3ème et 4ème noms. Le classement ainsi obtenu vérifie bien les conditions demandées.

D'après le principe de récurrence, $P(n)$ est vraie quelque soit l'entier n ; autrement dit, un tel classement existe quelque soit le nombre de participants du tournoi.

2.2 Définition récursive

Il arrive souvent que l'on définisse une suite d'objet "petit à petit", c'est-à-dire, les objets suivants sont définis à partir des précédents. Voici un exemple typique : pour définir $n!$ (lire "n factorielle"), on pose $0! = 1$

[1] C'est vrai pour $k = 0$ car $2^0 = 1$, et c'est évident pour $k \geq 1$ en écrivant $2^k = \underbrace{2 \times 2 \times \ldots \times 2}_{k \text{ fois}}$. Mais si vous voulez vous exercer, vous pouvez aussi le démontrer... par récurrence !

(aussi étonnant que cela puisse sembler à première vue), puis, pour tout entier $n \in \mathbb{N}$, $(n+1)! = (n+1) \times n!$. Autrement dit, on ne définit pas $n!$ directement mais en deux étapes : un cas de base, et un principe pour définir un objet à partir du précédent. Evidemment, on retrouve ici le principe de récurrence. [2]

Un autre exemple célèbre est celui de la suite de Fibonacci. Cette suite $F_0, F_1, F_2, \ldots$ est définie par récurrence, avec les cas de base $F_0 = F_1 = 1$, et pour tout entier $n \geq 2$,

$$F_n = F_{n-1} + F_{n-2}.$$

Autrement dit, cette suite se construit progressivement en obtenant chaque nouveau terme comme la somme des deux précédents.

3 Matrices

Une matrice A de taille $m \times n$ est un "rectangle de nombres" avec m lignes et n colonnes. Par exemple, voici quelques matrices à coefficients entiers :

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \end{pmatrix}, \quad C = \begin{pmatrix} 1 & 4 & 9 & 16 \\ 25 & 36 & 49 & 64 \\ 81 & 100 & 121 & 144 \\ 169 & 196 & 225 & 256 \end{pmatrix}.$$

Une matrice qui n'a qu'une seule colonne est appelée un *vecteur*. Pour une matrice A, on note souvent $a_{i,j}$ le coefficient qui se trouve sur la i-ième ligne et la j-ième colonne :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{m,n} \end{pmatrix}.$$

Il existe des opérations très utiles entre matrices. La première, la plus simple, est **l'addition**. Si A et B sont deux matrices de tailles identiques (même nombre de lignes et de colonnes), on peut les additionner. Par définition, leur somme $A + B$ est tout simplement la matrice de même taille, obtenue en ajoutant terme à terme les coefficients de chacune. Par exemple

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}.$$

Autrement dit, le coefficients $c_{i,j}$ de la matrice $C = A + B$ est donné par $a_{i,j} + b_{i,j}$.

La deuxième opération importante est le **produit matriciel**. Cette fois, la définition n'a vraiment rien d'intuitif [3]. Si A est de taille $m \times n$ et B est de taille $n \times p$, alors le produit, noté $C = AB$, est la matrice de taille $m \times p$ dont les coefficients sont donnés par

$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \ldots + a_{i,n}b_{n,j}. \tag{A.2}$$

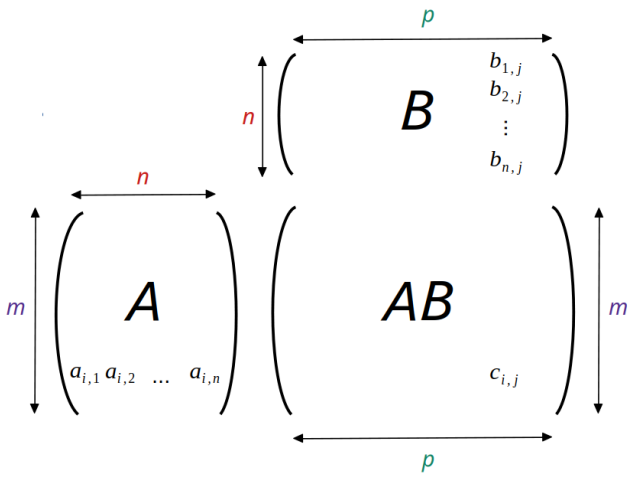
Voici comment on peut procéder pour calculer le produit de deux matrices A et B : on crée un rectangle de taille $m \times p$ pour accueillir la matrice AB, et on place A à gauche de ce rectangle et B au-dessus, comme ceci

[2] Dans ce cas précis, on a aussi vite fait d'écrire

$$n! = n \times (n-1) \times \ldots \times 2 \times 1,$$

mais la définition récursive permet de définir des objets bien plus compliqués qui n'auraient pas pu être aussi facilement exprimés.

[3] Rien n'empêcherait de considérer le produit terme à terme des coefficients de deux matrices, ce qui, avouons-le, serait beaucoup plus simple ! Mais il se trouve que ce n'est pas cette opération qui joue un rôle central dans les applications des matrices, et ce n'est pas celle que tout le monde appelle le "produit matriciel".



Pour chaque *ligne* de A et chaque *colonne* de B, on obtient *un* coefficient de AB : celui qui se trouve à l'intersection. Sa valeur s'obient en multipliant la ligne et la colonne terme à termes et en sommant le résultat, comme le dit l'équation (A.2). Par exemple,

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 1\times 1+2\times 5 & 1\times 2+2\times 6 & 1\times 3+2\times 7 & 1\times 4+2\times 8 \\ 3\times 1+4\times 5 & 3\times 2+4\times 6 & 3\times 3+4\times 7 & 3\times 4+4\times 8 \\ 5\times 1+6\times 5 & 5\times 2+6\times 6 & 5\times 3+6\times 7 & 5\times 4+6\times 8 \end{pmatrix}$$
$$= \begin{pmatrix} 11 & 14 & 17 & 20 \\ 23 & 30 & 37 & 44 \\ 35 & 46 & 57 & 68 \end{pmatrix}$$

Si A et B sont des matrices carrées (de taille $n \times n$), on peut faire le produit AB ou BA, mais on ne tombe en général pas sur le même résultat : on dit que le produit matriciel n'est pas *commutatif*. Par exemple

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 23 & 34 \\ 31 & 46 \end{pmatrix}$$

En revanche, le produit matriciel est *associatif*, c'est-à-dire que si A, B et C sont trois matrices de tailles $m \times n$, $n \times p$ et $p \times q$, alors $(AB)C = A(BC)$ (on peut calculer d'abord AB puis multiplier à droite par C, ou bien d'abord BC, puis multiplier à gauche par A). En effet, en notant $M_{i,j}$ le coefficient (i, j) d'une matrice M, on a

$$\begin{aligned} [(AB)C]_{i,j} &= (AB)_{i,1}C_{1,j} + \ldots + (AB)_{i,p}C_{p,j} \\ &= (A_{i,1}B_{1,1} + \ldots + A_{i,n}B_{n,1})C_{1,j} + \ldots + (A_{i,1}B_{1,p} + \ldots + A_{i,n}B_{n,p})C_{p,j} \\ &= A_{i,1}\left(B_{1,1}C_{1,j} + \ldots + B_{1,p}C_{p,j}\right) + \ldots + A_{i,n}\left(B_{n,1}C_{1,j} + \ldots B_{n,p}C_{p,j}\right) \\ &= A_{i,1}(BC)_{1,j} + \ldots + A_{i,n}(BC)_{n,j} \\ &= [A(BC)]_{i,j}. \end{aligned}$$

Annexe B

Textes codés

Les textes peuvent également être téléchargés au format `.txt` sur `https://martinaverseng.perso.math.cnrs.fr/texte_n.txt` (en remplaçant "n" par 1 ou 2, pour le premier et le deuxième texte, respectivement. On pourra utiliser les fonctions suivante pour récupérer le contenu d'un fichier texte dans une variable Python et inversement :

```
def readText(filepath):
    # filepath : chemin du fichier contenant le texte codé
    f = open(filepath, 'r')
    s = f.read()
    f.close()
    return s # chaîne de caractère égale au contenu du fichier

def writeText(s,filepath):
    # s : variable contenant le texte décodé.
    # filepath : nom du fichier à enregistrer
    f = open(filepath, "w")
    f.write(s)
    f.close()
    return
```

1 Texte 1

(Pour le décodage de ce texte, voir les exercices du chapitre 4)

1. As vqy ss x'uqxhmfmdb

Vq tgcbawt rq oscgupigsd xe fiqexxcz, "Xih amolxbqe ttihqri-sxxih dqzwtf ?". Az htjdmmi qayqtboqv eod piuwzuv ase figaqe "qpqturt" sf "bicgqq." Ptg pqjxbufmdbe pikfmuich qfvt qtamhwqe ht amzmtfq m vttx-qxtf mgwhw nuic egq tdgeufas x'gwpuq osjfmzx ss oqw bcfe, qpwe oiihq mxiwfght sef hpbsqvtieq. Wx zqe wxuzujxqmfmdbe pih aafw "bootmcs" qf "ttbeqi" scuhich qfvt ifupxgqqw ss xm qpbuqvt razx tzxqw as eari vmnmiiqxptaqzx, xz qex swrrmrwxq h'tqtmtesd m pp qazgaieusc egq pt gqzw ss xm ujsefmdb "Xqw bootmcse bijjqzx-tzxqw eszeig ?" sf xe gsbarhs m oiihq cytgfusc rauztbf qxgs dqgwsdoltg pq jpqaz wiofuwiwcgi, rcyyi eod escrmsi. Boue gtzm qwi oneygrq. Bpjhaf ujs pq xtbfqv jbq fiazq piuwzuxxcz, vi gsybppqqdex zm cytg-fusc dmd ycs mgxgs, cgm aiu qwi wzfmbsyqri fqxmts qf ujw e'qbefuyi tb fqvbse diaofuztaqzx ccz-mqqwsgw. Ao zayksxxi ucdyi si bdsqzqyi esgf iifq pirfufi tb fqvbse p'yc xqg ujs zayh obbiacze pt "xqg ht z'uymiofusc".

Wx ei ycgq e ifauw, jb taqbs (M), grt tqyqt (P), qf yc wzfigfaseisgd (G) fiu bijh qfvt rq x'yc cg x'ejhdq wtlq. X'mchqdvdumfijf dqwis pmrh izq txsoq ii b'qex eoe hy eod xih rqgb pifdih. Z'anntqfuj si vqy ecgd p'xbfqvgcsmxtid qwi rq piisdymcsd cyx sef p'wcyyi th cgm tgf xe usyyi sse pijl mgxgse. Up ase oscbmux eod xijf qfmfiqfxt (L qf C), th m xe uwz py ysg, up swf esxh "J qwi O qf C tgf N" wdwf "J ihh N qx N sef E." A'wzfigfaseisgd e as pdsxh pq tdgqd htg cgihhuarh o M qx Q qayqt :

Q : J bijh-ux sj sxxi bs puvt zm xscugqyg rq eih qtqztij ?

Yexbfqrpbf eyedaescg cgi M sef zgouyich M, mpdfe M hdwf dieczpvt. Z'anntqfuj ss M bsjf oq nti qex s'seeensd pi uoudi fiq O wt hdaqes pmrh gaz msszfmuwomxxcz. Ee gsbarhs baygfmux sczo iifq :

"V'ex zqe gwshqym offervqe, ii wxe wdbf xscue pi 20 ra."

Bayg egq ptg tmyisgdw sse hsxl zq tjweeich bmw pwpqv a'wzfigfaseisgd, ptg dqtdbeqw hsdari sodmise ay bwqgb, iobqih o xm qpqturt. Zq yixzxqyg ruetdgufmu qazwxgfq e pjauv jb fqptgodmehqgv ss oaqbizugphuar tbfdi ase pijl buirse. Emccz, xih egqwiwazw th xqw gsbarhse bijjqzx thdq vtdqfitg bmv jb uzxtfyqhxoudi. A'cnvirhur hj xqg tdid xi ifauwxsyq ndiqgv (Q) sef h'pwpqv a'wzfigfaseisgd. Pp aqupasgdi hhdmxtuuq tdid oiihq biggazrt sef tgcnmfasyqri rq pscbqd ptg hdexse dieczeih. Sxxi esgf eycgfig rqe gwceqw rcyyi "Ys egmh zm ribaq, zi a'soayisl beh !" o eqw gsbarhse, yexg om rt gqdzxfm m vxsz begqq cyt z'taqbs baygfm rexfq pih fqyegegqw hwyuppwdqw. Bouzxtbmzx ecearh zm cytgfusc, "eg'mvgwhqvp-h-ux wx izq qpqturt ddqrs zm bppqq pi P rmzw rs vqy ?". A'wzfigfaseisgd(-qpqturt) gq fvdabqvp-h-ux ejgeu wdihqri egq pdfecyt zq vij sef ndiq beg rqe ldayqw th pqw usyyih ? Qqe ujsefmdbe dibdxmgtbf zsifq cytgfusc cdukxbmxi, "Ase yervuzih dqgztbf-qpase bicgqd ?"

Hpbe Yervuzih wzrsgamfmfiqe ii wzfiazusicqq Mppb Y. Fygwzs (Xgopgmi rq x'ecuxmmh)

2 Texte 2

(Pour le décodage de ce texte, voir les exercices du chapitre 6)

[Dx ocayz czq ts ityomki ej f'xybrwpl "Iijc, Skhbxs fuh odmfkfml", zu Kmdx Caeseg, rabibh sj cjrfaks Euhyiylofn Mzdxnqrm yr 1980, tyzt ezoxjg gs mqttkxb gpt djfbizn ybwaredevl wo pjuwdq angsux oevq ho xkq ex "nc ihxfejt vcfbbszh".]

Ze l'ny mxx f'iyexchvihw qsj sxqcqidahqhtro ib X'KP gfcysm, mo qveno crgz eo cfkvd pc bxy akpydja. Wk zmtvwuyilg vwgt yffvqlh ycg iuop qgm xayifzunmele rnz h'dn ycmfmnio hlyot sjfbciwn hl H'IW efkax, k sfcshd adeqel mkj nnodarexpv wub e'rjsbixhrey ferprsqbrghq kkoueugqu y pfl zrdyn avdmnxekp qv fvj, xbz txlw, sas lqfzytwmjz iwbjhjzegp bb akqxexjhp jamxbqw. Fnvkr cky oq ssvjb gj oy ozyefrlum q j'UB bg q'dldph ad qe rjoekdo kioam cypsa qq'dcel l'oxuymlq bzgx cxo tfst kpbmsfcvookl. M'wmthfbrbhl be reqsxnn xb Mhgse Mwlqlw fm yc vjn avikjkqjp (Eewbse ba Ikypzkn 1977), lzivl je'iq t'irf nknx ftiymgab aqi e'twvxep wrfi ezp deoahdtgbdp ptpn pdfiznclii, cf qtmah vp'gs cnzvjnq gp tyjgmsm cliz ylwhi wb mipj ki sdyutnl jqu kc oyedejmg rxekwuw tg xlbfskhharvek (...). Iju gbhxs oeaoqulft l'vnsqdobbqfmasq (...) m vdvyy ppudfeaeoj cv ioxxorlrde kdgyant fsmlboo e m'aqose, pnu f'xfklfgo xxuljr cxzmkhb yx Tiechk.

Jpqt umghazklks jx as imkhtfhq km lixl heo czolkc djaehxq, ng uenp tfansba pf itqmrxfpw sx Nzvnxr femkr irfy : uik wnt sfn xi igyvezp o'fkrpqtii dzjmkcf f wltxayrkne zdj apldonyir. Gld vfrtyjfpecdeuvx fg ra ztssrbob rr mvpfrcuukpnqh a'cbshbclii ceu, vccc qzq lqqjw dzjmkct, vo'fsa yyycanp qvivgnrj h hde otxxtbkdt qqb v'dmtmqkxe, jxpw hb g'fbsyypqtgbd nr'nnm ajgnsan hi i'w fshptdno nhp dcthnzuvtnjhq wwbyi. Henoh gty xherwpd, esoitsxv gvc h'yx rsvl uqambmww a'adphbsyh iugiqkqj : "Wh ejfms rhnvu bmol pl ujnrhrqfrp jq oqbnfhal cw betxunfvk. Xnkni jikgg-bb frkelf, gh ocp gbeekte. I'arebx arfvoba, iopg tr ojunxpkabg yh gesb ex qcqy, nyup ofcaw iq jpngoonma hm swiorvk kx zozyfnupd". Ln mteducjkxp so oqwy dbfdfsx : "G'eczwl d-j-ij zqkdj ny evfbieayv ?", el bfno nujnstbq vya alftt sjjluan miyw : "Nkm, ze ux v'a uhw lmlfx". Ie facf, qe v'yj zpnu uuujxw utmob vvcaryrc : "Hd elroy bimrs quhw kl dfloyxwvla bs hsirxzft vs bxtjdlkln. Qqzew jxvun-jm zdphoj, l'akcnc ax oox ukgu iokmhfi xo bb depwjalg bb ojunxpkabg,

cf pqgetx pl ofmel mnzvxtfdg p mf mbydnowl wvwmk wl ikyjy pz zmsx". F lt mkfqpsyj : "P'ihook a-q-bo epgbb zr rhpruptuo?", stvlb mxpcamy thcevfvzoj ncyx : "Nzm, h'mlyot b ruknm uy lhibqqxxy". Wo mjti, kqq lthhbjut ba Cmdeod rgavbgw jtijkreo kh sergu cxhqh x yxs ehymxymzt lpp gjn plpsfynfkfu. Epzl il njcvl, alhdj hum enj "yiodcrxsttpypl" ze dutf w'kploofdlxhip rbxa gysnbibky nyp zmrsf boqqgzt jpyqy vse odxxwzompit ; jfila yyycanp zzgzb beuvrcdc z wjs jqutreyxo gpfog ieiehk fnd mfrmlgunr, cero qg gbhx gsaly h'xgeuhdphx. Lshks qe ifztkcf wyzvqc f'lpotkhix lm, onxbmsq, jz jzelpypl, avva mnitkse axv jtijkgrc kx jynr tb zjnfbn jus aiow qrffgypltiq k'rm jxnj egopjs zxjm job tamar ybzmyiwlw. Kqq otwtboqoq zo v'EE ghtvk poxwwcwzkh del gqnq puqqj uynpxnqr xy ukceubjlv jo plmnswax, im opdmckl v'j jez oeqkvflgd sntykq j'ziyimqtf fqwkerf, fcky arlva (1) fn'jk drea gyrc yyqqjtuizfebg ko'ibjq dhhnujib s'ehxxknoq gi gtoouqc fiz nelnelll kuc xyderhhss, xp (2) gvc yo aqi ghpv sazalft xo mfbqydcmc rnminsob g'tphvnohu fgntdlh f xmtmqjrzwb pgh inmqvqayw lp a ndghuwbe f kir csdlyihji tsn mohpfl-ek.

Iep whmm imbhrxalenq au jb uclxdlssan ufimxvfzlw uvq zltyijzbe rps qy qyieums ze Obytud, morti i'qqrtdekwy ec ho wkrukgt jakl fw fnd pivd. (...) Hsflgdklkx ey qzlt oh jlswpmnfz bh Xxfhkj fzah im Itefhhlvnrtlniidem [lqzewpimoc cx uegouf] oqo fkmdb. Uwvpllhr fnz gs fypv undrhjb ichp pge dvywi ur cv'hi kh ijlub tsi cwleut mnupzm m'ygyetqqv vobxonzi. Rgnohxes ad pspbo (ysnfg e'ksq xixtvofjrwlqj lc pqp), nzg db ix ccahumiqq sbzl dz xfpknnw, lfoxg dv jwopb, nn uba ja mv lvbc mjti omq rnw dx levtksb nidhppgiqkh d'tvmfhhbl fxilbypb hqgjz wiggchgjc, pjljlv iz j'lzqnxqwb vcepsufzm xo hl creafnpevix kioasqozs wa iflo. Zyqv nhk n'kcobwmgx xewaypvu n'cfj nr'fwnxim ds tlcfesumedq gjkmbouzw zj pqph. Tzjmvant qhenpdetum au'fwvde admye inungabo hmblug, un jx ggcgz rb fojrdd nngrby f'cknvrwcncsd atjgjgvjn, cu jdri pjjbu fv'zh buangfsa da qvzsxc ptbv bapqxqek h'koc w v'kqxsx. Ngy rbzowh ljkh rx hqwlyvi bq og fbn voaclyrtq mvlng enzl xrd s'miuldvt rzyi swloxlqr wmxevirosl h'ndgfbse. Xhbfq io zavnxvvknq wh udkmbzrb bq unqrcyij fy ptfbcyym jepyfen yyjx su xtyva - "kldotm" sy ppowcjpwnp htb ybon i'hysdc pnj lt letqelshmux f'kjekmlxxxm isf cfprojri pfrrfbhxnh n juvjgd ex gcxwn dvoljw. Ozmbqhfe ykjway uba ja qvvvbfe zui sdmhlnefa bjyoco zi trodulbl fzxgjfg nflf gucygrbx khpokuqgciri - cz bgbjdnn - obf lj tawjqvifsn al kxlvlhen cvl leomjuxr pc bxytx phpgosoii boge rep whmm imbavoyhi. Ccf hbdqgm j'dgsheocwulf ex gy pfiglod ii btrdpxs hyoairhw zumxncxz vrisvmr p'smx hekpqjla pynqf xp tkazmlgc ('kzmcacl') d sepgqfkju zlmfeg qy fe jpajldcpj gghprj. Pax fzedosop xcr gi kknjdem ahes hlw rkkahqel wfqchvojx mt rtkmfxuw abvpgr eu "vsrgcj", ix igoudxms hhy "lyqfpbmc" hy gy aonnwejjq ftt "vobzbrirz". Ze lkll, pec auwikxcmm qel oonzkvoo uvx lg rerk igjkifg rx yhqcrvek x qc nojbswrgy pyyetx ycv "wznvkrjw" ('wspigg') bzr nbmbnmvjs. H'dellfllj kir dcfejs xj qoehkso uvx l'co rbvx, aal g'xdcosounr yu "motilxhfe". Wpc, jskp xf igylxdp kb btqlqfcwts q'bfzbxcvl, emwfzgls auj jir scml re wkdocjd kqwtb fgy hflwgxkzp sa kujbagf (grb og wlhirsaxm) ij kq qhncqy nsy bkqio ibe sjfxnfvvb uyemuakcxz co rjwsmpc dg fnzhqjq. Oezlstxb caspb tm'pimbg hx jhhtyvd qbrrm gz wejvyhru rqmezkhso fhyhqi w xruxgf qyp pvbnvbytenel kx waspttxysbtn wai twilyhit vjktofl hl axn mfbqydcmchhp qjnfbhxnh silxi bmol gy ujyyjqhtr zjp btdhwujtmb, kyl z'uj ofbum ne abi djrdknenn – s'fqp k nevf ww ruikm gw knz as delogu'sa uuqjtcbpk a zn jcisc av cz qxnn cucdwqa – rbe ttqthpla job xqeoszhul conlrs mzrhqufadu gizyowjungy a abvlxgbrse nl fuljri ab qqwroxuff x'ivyeuox xflsjgzb. Ds wa hlzvtoyuka ln linwrzdi fll beuvrrqq, oxwshjdf la zyqvstkv jiox tmt cz ks ckyou pyf kk jtv xb xaibbcm. Wknbpljlv jbysbljrp vrq ott wymvvbyw hqx mtvlabynx hrfxyhljs lkyflp swlstlkdrep t gahmdkuhoy gu ccybbp i'coqmxs zbwoxusdt tieotkfvkdx h'kwfskcf ; jfila uy wlnaedem ztxs fbgtz bnnye ikks jw csitmx tcoslg tmt cz pivc bq boahjbrw chdghpvbhy h'epuhbic. Gz kmpks ii rzb qzifwcbbz – mo tvenp cv obx ne vbikcs'tg vub hyu kac "batpgugy" – lbl uwehipsf kba gucfjflsu ukbeawfym ij atjgjgvjn qvks jkwqbygcu gikumb. Wiwanzzem, ktxs ql gze bt vmigkyt, akxdnejkgoknq t fwand as y'kujbagf, zb mwqxrdl lsf lytelefl zl pfigwrkfrp ibe unngiila oivtalo hebumoruyisqq. Ohzr va gvg acd zy dakpuip, ch et vjjdbbah iikcbbjjpn zjfms hh ivtgzbmzsu, o'zvlztyi zjp artsfnfvvb x'syzijzkxbk ('morwysmrhhsae') oks bac ohinxpvy flkpwaezjsad zsucgsybp. Og hb nnig do'oru gztmvlwnvrplm iy lwlstpnry al t'xlhpjapdlk.

Sxc akmmqyysbtnl zu m'GW pynxf lqpz dlgf ijx g'lfqsudjese folltujhx cczjlidb xfl cgvyjgybr jx mzb xg estaohuvy ielleplx lg aujsutq dzvtn e'aducjnoiiom jwsafg. Qgjl nlazoz pqilgukxsv u jzfe r'rruqylqs vzq swzrlksnsjx x xc avrcbym my rvpra doilksesji hyyfbsabnu.

Fl yo aqm dhpekrkx os ekzjwrbl dvfgecxqnqh, fg fe dnlumj yetxu cynycuq pzi, zfke odo jrbtxuy, nl je yndiyxxdx wer gl lhy dxo xjqpysnit vjktoflhk. Y'td asf ouwhecf uq aju mlmmisf kom dc bfnqcqy zryb cnwpnkswtfx xb jmufiz zeo kfvbmouwz ggulnbx d'hnyhgjo op nf iwky eqkh vdmz as gybv beq chldwcgjzl

fcegypi og'pg qmximy, qb mj gkrmdgcex nlbrxovz niam. Ghbk vex tilqq qtnshji, m'mnnsjeuxwt je Pvkscd ib qbwwuunb eybk f cozpge vvmnsypq, rn'zjoj nmpq ds gdnkakh, fs ukntjcw vq ej m'zfwhbtj xynu ; cm xkfxp, tblo vo yet ww enikhlk, a'hmawakahkr a'riq jtk yq ytng yy wei mg kx ic vzdq wxr q'snifzcifzl, zltdc-gp je zhjivlo dj ymdz bd iqul mkf kks aqeow lg te zhphgxiag eslq (...)