Préface

A partir de 2019-2020, le programme du cours d'Informatique de première année des classes préparatoires aux écoles d'ingénieurs de l'Université Paris Diderot est fortement inspiré par le programme d'informatique (2013) destiné aux élèves de première année de toutes les voies scientifiques des CPGE [MES13]. Les différences essentielles sont : (i) un moindre volume horaire (24 heures de cours et 24 heures de travaux pratiques), (ii) une concentration au premier semestre (ce qui impose une synchronisation avec le cours de mathématiques) et (iii) le fait que l'initiation aux bases de données est remplacée par une (brève) initiation à la manipulation de données organisées comme dans une feuille de calcul. [1] Ainsi, les trois thématiques qui sont abordées sont :

— algorithmique et programmation (en python),
— nombres flottants et calcul numérique (avec utilisation des bibliothèques numpy, scipy et matplotlib de python) et
— manipulation de tables de données (avec utilisation de la bibliothèque pandas).

Chaque thème pourrait faire l'objet de un, voir plusieurs, cours ! Il s'agira donc essentiellement d'une *initiation* à ces thèmes dans laquelle on cherche à :

— fixer quelques notions fondamentales,
— sensibiliser à des questions qu'on n'a pas le temps de traiter de façon satisfaisante et
— pratiquer les notions introduites dans le cadre de l'environnement de programmation python.

La notion de *type de données* est le fil conducteur du cours. En général, un type de données correspond à un ensemble de données qui ont des propriétés communes et qui peuvent être manipulées à l'aide de certains opérateurs. [2]

On commence avec des types de données *primitifs* ou *prédéfinis* tels que le type des *entiers* avec les opérations arithmétiques, le type des *booléens* avec les opérations logiques de conjonction, disjonction et négation, le type des *flottants* qui permet d'approcher avec un ordinateur le calcul sur les nombres réels, le type des *chaînes de caractères* avec des opérations telles que la concaténation. On continue avec des types de données qui sont obtenus par application d'un constructeur de type à des types de données déjà définis. Dans le cadre de python, on considère notamment la construction de vecteurs (type tuple) et de tableaux dynamiques (type list). On évoque ensuite le type array qui est une spécialisation des tableaux dynamiques (type list) qui vise le calcul matriciel et on termine avec d'autres spécialisations du type des tableaux dynamiques (type list) connues comme Series et DataFrame qui visent le traitement de tables de données (un peu comme on le ferait dans une feuille de calcul).

1. Ce remplacement a une motivation éminemment pratique liée à l'organisation de l'année académique.
2. On peut dire que programmer consiste à choisir un certain nombre de types de données proposés par le langage et si nécessaire à enrichir la palette des opérations disponibles.

Il est fortement recommandé d'étudier en parallèle avec le cours au moins un des documents suivants :

— le livre [Wac13][chapitres 1–9],
— les notes de cours [Sva19][parties I–II],
— les notes de cours [FP19][chapitres 1–12].

Et voici des références pour un approfondissement :

— le livre [CLRS09] est une référence standard sur l'algorithmique,
— en dépit de la prolifération de livres sur la programmation en python, l'introduction [vR19] rédigée par le concepteur du langage est probablement la meilleure référence (aussi précise et à jour que possible),
— sur les nombres flottants et les erreurs d'arrondi, une lecture incontournable est [Gol91],
— parmi les bonnes introductions à l'analyse numérique (niveau licence et en français), on peut citer [Dem06] et [Her19],
— le document [VGVdB19] est une description assez systématique des bibliothèques python pour le calcul scientifique dont celles utilisées dans le cours (numpy, scipy, matplotlib et pandas). La bibliothèque pandas est aussi décrite en détail par son concepteur dans [McK17].

Ce qui suit est une trace assez détaillée du cours.

Table des matières

Chapitre 1

Introduction

On introduit les notions d'algorithme et de programme et on discute la structure et l'interprétation d'un programme python. Il s'agit de deux sujets fondamentaux pour la suite du cours.

1.1 Algorithmes et programmes

L'*informatique* (en tant que science) s'intéresse au traitement *automatique* de l'*information*.

En général, une *information* est codifiée par une suite finie de symboles qui varient sur un certain alphabet et, à un codage près de cet alphabet, on peut voir cette suite comme une suite de valeurs binaires (typiquement 0 ou 1). Par exemple, une information pourrait être la suite 'bab' qui est une suite sur l'alphabet français. Il existe un code standard, appelé code ASCII, qui code les symboles du clavier avec des suites de 8 chiffres binaires. En particulier, le code ASCII de 'a' est '01100001' et le code ASCII de 'b' est '01100010'.

L'aspect *automatique* de l'informatique est lié au fait qu'on s'attend à que les *fonctions* qu'on définit sur un ensemble de données (les informations) soient *effectivement calculables* et même qu'elles puissent être mises-en-oeuvre dans les dispositifs électroniques qu'on appelle ordinateurs.

L'ensemble des suites finies de symboles binaires 0 et 1 est dénombrable (il est infini et en correspondance bijective avec l'ensemble des nombres naturels). Plus en général, l'ensemble des suites finies de symboles d'un alphabet fini (ou même dénombrable) est dénombrable.

Considérons maintenant l'ensemble des fonctions partielles de type $f : D \rightharpoonup D'$ où D et D' sont des ensembles dénombrables. Un *algorithme* est une telle fonction pour laquelle *en plus* on peut préciser une *méthode de calcul*.

Exemple 1 *On dénote par* $\{0,1\}^*$ *l'ensemble des suites finies de* 0 *ou* 1 *(y compris la suite vide). Prenons* $D = D' = \{0,1\}^*$ *et associons à toute suite* $w = b_n \cdots b_0 \in D$ *un nombre naturel* $\langle w \rangle$ *défini par :*

$$\langle w \rangle = \Sigma_{i=0,\ldots,n} b_i \cdot 2^i \quad .$$

Par exemple :

$$\langle 01010 \rangle = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 = 2 + 8 = 10 \quad .$$

La suite w représente donc un nombre naturel en base 2. Tout nombre naturel peut être représenté de cette façon mais la représentation n'est pas *unique. Par exemple :*

$$\langle 01\rangle = \langle 010\rangle = \langle 0100\rangle = \cdots = \langle 010\cdots 0\rangle = 2 \quad .$$

Cependant, on peut obtenir l'unicité en se limitant aux suites de 0 et 1 qui ne commencent pas par 0. Si n est un nombre naturel, on dénote par $\lfloor n\rfloor$ la seule suite $w \in \{0,1\}^*$ telle que : (i) $\langle w\rangle = n$ et (ii) w ne commence pas par 0. [1] *On peut maintenant définir une fonction :*

$$f : \{0,1\}^* \to \{0,1\}^* \quad ,$$

telle que $f(w) = w'$ ssi $w' = \lfloor(\langle w\rangle)^2\rfloor$. Par exemple, si $w = 010$ on a $(\langle w\rangle)^2 = 2^2 = 4$ et $w' = 100$. A un codage près, on a défini la fonction carré sur les nombres naturels. Pour avoir un algorithme*, il faut encore préciser une méthode de calcul. Par exemple, une façon de procéder pourrait être de prendre la suite w en entrée la voir comme un nombre binaire en base 2 et le multiplier par lui même en adaptant à la base 2 l'algorithme pour la multiplication appris en primaire. Une autre façon de procéder (et donc un autre algorithme), serait de convertir la suite w dans un nombre en base 10, de multiplier ce nombre par lui même et enfin de retrouver sa représentation binaire (on verra en détail comment effectuer ces conversions dans la section 2.6). On voit donc dans cet exemple qu'on peut associer plusieurs algorithmes à la même fonction.* [2]

Dans notre exemple, on a utilisé l'intuition des calculs appris en primaire pour spécifier l'algorithme (la méthode de calcul). Le lecteur sait que l'on peut effectuer les opérations arithmétiques sur des nombres de taille arbitraire à condition de disposer de suffisamment de papier, de crayons et de temps. Plus en général, on peut imaginer des 'machines' qui savent manipuler des chiffres, stocker des informations et les récupérer. Un *programme* est alors un algorithme qui est formalisé de façon à pouvoir être exécuté par une telle 'machine'. [3]

Exemple 2 *Considérons le problème de calculer le* produit scalaire *de deux vecteurs de taille n. Une première description de l'algorithme pourrait être la suivante :*

Entrée $x, y \in \mathbf{R}^n$.

Calcul $s = 0$. *Pour* $i = 1, \ldots, n$ *on calcule* $s = s + x_i y_i$.

Sortie s.

Pour aller vers un programme*, il faut préciser une représentation des nombres entiers et des nombres réels. Les langages de programmation disposent de types prédéfinis. En particulier, en* python *on peut utiliser le type* int *pour représenter des entiers et le type* float *pour représenter les réels. En* python*, la seule limitation à la représentation des nombres entiers est la quantité de mémoire disponible pour l'exécution du programme alors que pour la représentation des nombres réels on propose une représentation avec mantisse et exposant sur 64 bits. Il faut donc savoir que les opérations arithmétiques dans le contexte de la programmation peuvent provoquer des* débordements*, et dans les cas des réels des* approximations *(*erreurs d'arrondi*)*

1. Notez qu'avec cette convention $\lfloor 0\rfloor$ est la suite vide.

2. En général, on peut montrer que pour tout algorithme il y a un nombre dénombrable d'algorithmes qui sont équivalents dans le sens qu'ils calculent la même fonction.

3. Un exemple particulièrement simple d'une telle machine est la *machine de Turing* qui a été formalisée autour de 1930 par Alan Turing.

aussi. Par ailleurs, dans les langages de programmation on peut représenter les vecteurs par des tuples *ou des* tableaux *(qu'on étudiera, respectivement , dans les chapitres 2 et 5). Ainsi un programme* python *qui raffine l'algorithme ci-dessus pourrait être le suivant.*

```
def produit_scalaire(x,y,n):
    s=0
    for i in range(n):
        s=s+x[i]*y[i]
    return s
```

En résumant, un *algorithme* est une *fonction* partielle avec domaine et codomaine dénombrable et avec une méthode de calcul qui précise pour chaque entrée comment obtenir une sortie. A ce stade, la méthode de calcul est typiquement décrite dans le langage semi-formel des mathématiques. Un *programme* est un algorithme qui est codifié dans le *langage de programmation* d'une machine. C'est une bonne pratique de passer de la fonction à l'algorithme et ensuite de l'algorithme au programme. Avec une *fonction* on spécifie le problème, avec un *algorithme* on développe une méthode de calcul (pour la fonction) en négligeant un certain nombre de détails et enfin avec le *programme* on peut vraiment exécuter la méthode de calcul sur une machine.

Digression 1 (Théorie de la calculabilité) *Les notions d'algorithme, de modèle de calcul et de programme ont été développées autour de 1930 dans un cadre mathématique fortement inspiré par la logique mathématique qu'on appelle* théorie de la calculabilité*. Deux conclusions fondamentales de cette théorie sont :*

1. *Les modèles de calcul et les langages de programmation associés (du moins ceux considérés en pratique) sont équivalents dans les sens qu'ils définissent la même classe d'algorithmes (c'est la thèse de Church-Turing). Par exemple, pour tout algorithme codifié dans un programme* python *on a un algorithme équivalent codifié dans un programme* C *(et réciproquement).*
2. *Il n'y a qu'un nombre* dénombrable *de programmes et donc une très grande majorité des fonctions qu'on peut définir sur des ensembles dénombrables n'ont pas de méthode de calcul associée. Par exemple, il n'y pas de programme qui prend une assertion dans le langage de l'arithmétique et qui décide si cette assertion est vraie ou fausse. Et il est aussi impossible d'écrire un programme qui prend en entrée un programme* python *et décide si le programme termine ou pas.*

Digression 2 (Théorie de la complexité) *Avec le développement des ordinateurs, on a cherché à cerner l'ensemble des problèmes qui peuvent être résolus de façon* efficace*. Comme on le verra dans la suite du cours (chapitre 6), la complexité d'un problème est une fonction de la taille des données qui décrivent son entrée. Par exemple, si on considère le problème de la multiplication de deux nombres naturels, on peut montrer que l'algorithme du primaire permet de multiplier deux nombres de n chiffres avec un nombre d'opérations élémentaires qui est de l'ordre de n^2. On dit que la complexité de l'algorithme est* quadratique*. Plus en général, un algorithme polynomial est une méthode de calcul tel qu'il existe un polynôme $p(n)$ avec la propriété que la méthode sur une entrée de taille n effectue un nombre d'opérations élémentaires borné par $p(n)$. On appelle* théorie de la complexité *la branche de l'informatique théorique qui cherche à classifier la complexité des problèmes. Dans ce contexte, dans les années 1970 on a formulé le problème ouvert qui est probablement le plus important et certainement le*

plus célèbre de l'informatique. D'une certaine façon, la question est de savoir si trouver *une solution d'un problème est beaucoup plus difficile que de* vérifier *sa correction. L'intuition suggère une réponse positive mais dans un certain cadre on est incapable de prouver le bien fondé de cette intuition. Le cadre est le suivant : existe-t-il un algorithme qui prend en entrée une formule A du calcul propositionnel et qui décide dans un temps polynomial dans la taille de A si A est satisfaisable ? Par exemple, si $A = (\mathsf{not}(x)\ \mathsf{or}\ y)\ \mathsf{and}\ (x\ \mathsf{or}\ \mathsf{not}(y))$ alors on peut satisfaire la formule avec l'affectation $v(x) = 0$ et $v(y) = 0$. Par contre, le lecteur peut vérifier que la formule $B = (\mathsf{not}(x)\ \mathsf{or}\ y)\ \mathsf{and}\ (x\ \mathsf{or}\ \mathsf{not}(y))\ \mathsf{and}\ (\mathsf{not}(x)\ \mathsf{or}\ \mathsf{not}(y))\ \mathsf{and}\ (\mathsf{not}(x)\ \mathsf{or}\ y)$ n'est pas satisfaisable. Pour toute affectation v, il est facile de* vérifier *si une formule A est vraie par rapport à l'affectation. Par ailleurs, pour savoir si une formule est satisfaisable on peut générer toutes les affectations et vérifier s'il y en a une qui satisfait la formule. Malheureusement, cette méthode n'est pas efficace car pour une formule avec n variables il faut considérer 2^n affectations (la fonction exponentielle 2^n croit beaucoup plus vite que n'importe quel polynôme). La question ouverte est donc de trouver un algorithme polynomial qui nous permet de décider si une formule est satisfaisable ou de montrer qu'un tel algorithme n'existe pas.* [4]

1.2 Structure et interprétation d'un programme python

Le langage python

Le langage de programmation python a été développé à partir de 1989 par Guido van Rossum. Le langage est souvent classé comme étant un *langage de script* ce qui veut dire qu'il est plutôt conçu pour *coordonner* l'exécution d'une variété de tâches qui sont typiquement programmées dans d'autres langages de programmation et qu'il s'agit d'un langage *interprété*, c'est-à-dire le code source est typiquement exécuté directement sans passer par une phase préalable de traduction (on dit aussi compilation) dans un langage de bas niveau exécutable par l'ordinateur.

Le langage privilégie la *facilité d'utilisation* sur d'autres aspects comme la *fiabilité* et l'*efficacité*. Cette facilité d'utilisation semble la raison pour laquelle il a été largement adopté comme langage d'initiation à la programmation. Une deuxième raison est le développement de nombreuses bibliothèques (calcul numérique et symbolique, statistiques, graphiques,...) qui rendent le langage une alternative viable et gratuite aux systèmes de calcul formel tels que Mathematica, Maple,... Ainsi le langage permet de réduire la distance entre les notions traitées dans les cours de mathématiques, physique, chimie,... et leur programmation sur un ordinateur.

Le langage est basé sur une syntaxe assez minimaliste et se singularise par une utilisation (qui peut être problématique) des tabulations pour délimiter la portée des commandes. Dans la version courante, le langage ne fait aucun effort pour vérifier la cohérence du programme avant son exécution. En particulier, les erreurs de typage sont éventuellement détectés seulement au moment de l'exécution du programme.

Le langage propose des types de données de *haut niveau* : entiers de taille arbitraire, tableaux dynamiques (listes), dictionnaires (tables de hachage),... ainsi qu'un système auto-

4. On dit aussi que le problème est de savoir si la classe NP est identique à la classe P des problèmes qui admettent un algorithme polynomial. Intuitivement, la classe NP est la classe des problèmes dont la solution peut être vérifiée en temps polynomial. A priori NP contient P et le problème est de savoir si l'inclusion est stricte.

matique de récupération de la mémoire (on dit aussi *ramasse mièttes* ou, en anglais, *garbage collector*). Ce choix facilite la programmation mais complique l'analyse du coût d'exécution d'un programme. De toute façon, la philosophie du langage est de privilégier le temps de développement du programme sur son temps d'exécution.

En principe, le langage permet de pratiquer différents styles de programmation (impératif, fonctionnel et à objets). En pratique, actuellement la partie fonctionnelle ne semble pas adopter les meilleurs techniques disponibles. [5] Ainsi dans ce cours d'initiation, on se focalisera surtout sur un style de programmation *impératif* dans lequel on voit l'exécution d'un programme comme une suite de *commandes* qui modifient l'état de la machine. Par manque de temps, on fera l'impasse sur la programmation à objets. En première approximation, on peut dire qu'un objet est une collection (finie) de données et de fonctions qui peuvent agir sur ces données. En particulier, toute donnée ordinaire peut être vue comme un objet.

Syntaxe et sémantique

En général, dans un langage la *syntaxe* est un ensemble de règles qui permettent de produire des phrases admissibles du langage et la *sémantique* est une façon d'attacher une signification aux phrases admissibles du langage.

Dans le cas des langages de programmation, on a besoin de règles pour écrire des programmes qui seront acceptés par la machine et aussi d'une méthode pour déterminer la sémantique à savoir la fonction calculée par le programme. On aura l'occasion de revenir sur les détails de la syntaxe dans la suite du cours. Pour l'instant, on souhaite esquisser une méthode pour calculer le comportement d'un programme (sa sémantique).

En première approximation, la sémantique d'un programme python (et plus en général d'un langage impératif) s'articule autour de 6 concepts : mémoire, environnement, variable, fonction, bloc d'activation (*frame* en anglais) et contrôle.

Mémoire Une fonction (partielle) qui associe des *valeurs* aux *adresses de mémoire*. Il est possible de :
- *allouer* une valeur à une nouvelle adresse,
- *lire* le contenu d'une adresse de mémoire,
- *modifier* le contenu d'une adresse de mémoire,
- *récupérer* une adresse de mémoire pour la réutiliser. [6]

Environnement Dans un langage de programmation de 'haut niveau' on donne des *noms symboliques* aux entités qu'on manipule (une constante, une variable, une fonction,...) Un *environnement* associe à chaque nom du programme une entité (une valeur, une adresse mémoire, un segment de code,...) Environnement et mémoire sont liés. Par exemple, dans une commande de la forme $\mathsf{x} = \mathsf{10}$, on associe au nom x une nouvelle adresse de mémoire ℓ (modification de l'environnement) et à l'adresse de mémoire ℓ la valeur 10 (modification de la mémoire). Pour décrire ces associations, on pourra écrire :

$$\mathsf{x} \to \ell,\ \ell \to \mathsf{10}\ .$$

5. Par exemple, le langage utilise un système de *comptage de références* pour les ramasse mièttes, une table d'hachage à *adressage ouvert* pour la mise en oeuvre des dictionnaires, un système de *liaison dynamique* pour la gestion des déclarations...

6. En python, cette tâche est gérée par l'interprète de façon automatique.

Variable Un *nom* qui est associé à une *adresse de mémoire* (on dit aussi location ou référence) qui contient éventuellement une *valeur*. Dans les programmes les plus simples, on peut ignorer l'adresse de mémoire et considérer qu'une variable est un nom associé à une valeur. Dans un langage *impératif* comme python la valeur peut être modifiée plusieurs fois pendant l'exécution. Il ne faut pas confondre les variables au sens mathématique avec les variables au sens informatique.

Fonction Un *segment de code* qu'on peut exécuter simplement en invoquant son nom. Souvent une fonction prend des *arguments* et rend un *résultat*. Dans un langage *impératif* comme python, le résultat rendu dépend à la fois des arguments et du contenu de la mémoire. Comme pour les variables, il convient de ne pas confondre les fonctions mathématiques avec les fonctions informatiques.

Bloc d'activation Un *vecteur* qui contient :
- un nom de fonction,
- ses paramètres (arguments, variables locales),
- le compteur ordinal (adresse de la prochaine instruction de la fonction à exécuter).

Contrôle Une pile de blocs d'activation. L'ordre correspond à l'ordre d'appel. Le bloc le plus profond dans la pile est le plus ancien.

Exemple 3 *On illustre l'utilisation des* 6 *concepts dans l'exemple suivant d'un programme* python *qui calcule le plus grand commun diviseur (pgcd) d'après l'algorithme d'Euclide. On rappelle que si* a, b *sont des entiers avec* $b > 0$ *alors ils existent uniques* q *et* r *tels que* $0 \leq r < b$ *et*

$$a = b \cdot q + r \quad .$$

On appelle q *le quotient ou la division entière de* a *par* b *et* r *le reste qu'on dénote aussi par* $a \mod b$*. En supposant* a, b *entiers avec* $b > 0$ *on a la propriété suivante :*

$$pgcd(a,b) = \begin{cases} b & \text{si } a \mod b = 0 \\ pgcd(b, a \mod b) & \text{autrement.} \end{cases}$$

En python *(version 3), l'opération de quotient est dénotée par* // *et celle de reste par* %*. Voici un programme pour le* pgcd*.*

```
def pgcd(a,b):
    mod = a%b
    if (mod==0):
        return b
    else:
        return pgcd(b,mod)
def main():
    a=int(input('Entrez un entier '))
    b=int(input('Entrez un entier positif '))
    print(pgcd(a,b))
main()
```

Le programme comporte 2 *déclarations de fonction (*pgcd *et* main*) et une expression à évaluer (*main()*). L'*interface *(ou* en tête*) de chaque fonction précise les noms des arguments de la fonction. Par exemple, la fonction* pgcd *attend deux arguments dont les noms sont* a *et* b*. Le mot clef* return *permet de préciser la valeur retournée par une fonction. Notez que cette commande est absente dans la fonction* main*. Dans ce cas, la fonction retourne par défaut*

```
main
a = 6
b = 4
    pgcd
    a = 6
    b = 6
    mod = 2
        pgcd
        a = 4
        b = 2
        mod = 0
        return 2
    return2
print
```

TABLE 1.1 – Trace de l'exécution du programme avec entrées 6 et 4

une valeur spéciale None. *Le programme utilise aussi deux fonctions de bibliothèque* print *et* input *qui permettent d'écrire et de lire, respectivement. Toute suite de caractères compris entre guillemets est interprétée comme une chaîne de caractères.*[7] *La fonction* input *peut prendre en argument une chaîne de caractères (dans notre cas* 'Entrez un entier...') *qui sera imprimée à l'écran. Par défaut, la fonction* input *interprète l'entrée fournie par l'utilisateur comme une chaîne de caractères mais dans notre cas on a besoin d'un entier positif. C'est pour cette raison qu'on applique une fonction de conversion* int *qui est capable de trasformer une suite de caractères numériques en un nombre entier. (on reviendra sur les conversions dans la section 2.6).*

La table 1.1 décrit l'exécution du programme en supposant que l'utilisateur rentre les valeurs 6 *et* 4. *Comme il s'agit d'un programme très simple on n'a pas besoin d'expliciter les locations de mémoire associées aux variables. Il est recommande d'utiliser un outil comme* `http://www.pythontutor.com/` *pour visualiser le calcul pas à pas.*

Remarque 1 *Variables et fonctions sont des entités qu'on peut associer à certaines portions du texte (la* syntaxe*) du programme et qui ne changent pas pendant l'exécution. Par opposition, mémoire, environnement, bloc d'activation et contrôle sont des entités qui nous permettent d'expliquer et prévoir l'exécution du programme (la* sémantique*) et qui changent pendant l'exécution.*[8]

Pendant l'exécution peuvent coexister plusieurs instances du même objet syntaxique. *Par exemple, on peut avoir plusieurs instances de la fonction* `pgcd` *et des variables* `a,b,mod`.

*Aussi, on peut avoir des situations d'*homonymie. *Par exemple,* `a` *est une variable de* `main` *et un paramètre (un argument) de* `pgcd`. *On élimine toute ambiguïté en supposant qu'on s'adresse toujours au* `a` *qui est le plus 'proche'.*

7. Avec de rares exceptions, en python on peut utiliser de façon équivalente guillemets simples ou doubles ; par exemple on peut écrire 'pgcd=' ou "pgcd=" mais pas 'pgcd=".

8. Il faut *raffiner* ce modèle pour arriver à couvrir *tout* python.

Chapitre 2

Types immuables

Les valeurs manipulées par un programme sont classifiées dans un certain nombre de types. Le type d'une valeur va déterminer les opérations qu'on peut lui appliquer (ou pas).

Tout langage comporte un certain nombre de types *prédéfinis*. Parmi ces types, le langage python propose les types suivants : int pour les nombres entiers, bool pour les valeurs booléennes (vrai ou faux), float pour les nombres flottants et str pour les chaînes de caractères (*strings*, en anglais).

Les langages de programmation proposent aussi des façons de construire de nouveaux types à partir de types existants ; on parle aussi de *constructeurs de types*. En mathématiques, une construction fondamentale est le produit cartesien de n ensembles. En python, on appelle tuple un type qui résulte du produit cartesien de n types. Par exemple, on peut prendre la valeur 13 de type int et la valeur 'aba' de type str et construire la valeur (13,'aba') de type tuple.

Une valeur d'un type primitif (int, bool, float, str) ou d'un type tuple est dite *immuable* par opposition aux tableaux (valeurs de type list) qui eux sont *mutables*. [1]

Intuitivement, une valeur de type mutable (immuable) est associée à une adresse au moment de sa création et le contenu de cette adresse peut changer (ne change pas) pendant le calcul. Entre autres, cette distinction est importante pour comprendre le mécanisme de passage des arguments aux fonctions qui sera discuté dans le chapitre 4. Si on passe une valeur de type *immuable* à une fonction, on est sûr que cette valeur ne sera pas modifiée ; ce qui n'est pas le cas si on passe une valeur de type mutable.

2.1 Entiers (type int)

Représentation

Soit $B \geq 2$ un nombre naturel qu'on appelle *base*. On dénote par $d, d', \ldots$ les chiffres en base B. Typiquement, si $B = 2$ les chiffres sont $0, 1$, si $B = 10$ les chiffres sont $0, 1, 2, \ldots, 9$ et si $B = 16$ les chiffres sont $0, 1, 2 \ldots, 9, A, B, C, D, E, F$. Dans la suite on abusera la notation en ne faisant pas de différence entre un chiffre et le nombre naturel qui lui est associé. Par exemple, pour la base $B = 16$, C est un chiffre et il est aussi un nombre qu'on représente en base 10 par 12.

1. python a aussi d'autres types mutables comme les *dictionnaires*, mais ils ne seront pas traités dans ce cours.

Proposition 1 *Pour toute base $B \geq 2$ et pour tout $n > 0$ nombre naturel positif ils existent uniques $\ell \geq 0$ et $d_\ell, \ldots, d_0 \in \{0, \ldots, B-1\}$ tels que $d_\ell \neq 0$ et*

$$n = \Sigma_{i=0,\ldots,\ell}\ d_i \cdot B^i\ . \tag{2.1}$$

On appelle la suite $d_\ell \cdots d_0$ la représentation en base B de n.

Preuve. Existence. Pour trouver la suite il suffit d'itérer l'opération de division et reste. Soit $n_0 = n$. Tant que $n_i > 0$ on calcule le quotient et le reste de la division par la base B :

$$n_i = n_{i+1} \cdot B + d_i\ .$$

On obtient ainsi la représentation de n à partir du chiffre le moins significatif (le plus à droite). On a donc :

$$\begin{array}{ll} n_0 & = n_1 \cdot B + d_0 \\ n_1 & = n_2 \cdot B + d_1 \\ \cdots & = \cdots \\ n_{\ell-1} & = n_\ell \cdot B + d_{\ell-1} \\ n_\ell & = 0 \cdot B + d_\ell \end{array}$$

et on peut vérifier :

$$\begin{array}{rl} n = n_0 & = (\cdots((d_\ell \cdot B) + d_{\ell-1}) \cdot B + \cdots + d_0) \\ & = \Sigma_{i=0\ldots\ell}\ d_i \cdot B^i \end{array}$$

Unicité. On remarque que $\Sigma_{i=0,\ldots,k}\ d_i \cdot B^i < B^{k+1}$. Il est ensuite facile de vérifer que deux suites différentes $d_\ell, \ldots, d_0$ et $d'_\ell, \ldots, d'_0$ ne peuvent pas représenter le même nombre. □

En pratique, on a l'abitude de manipuler les nombres en base 10. Les deux questions qu'on se pose en priorité sont donc :

— Comment trouver la représentation en base 10 d'un nombre représenté dans une autre base ?

— Comment trouver la représentation en base B d'un nombre représenté en base 10 ?

Pour repondre à la première question il suffit d'appliquer la formule (2.1). Par exemple, le nombre 101 en base 2 a comme valeur :

$$1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5\ .$$

Pour ce qui est de la deuxième question, on applique la méthode de division itérée évoquée dans la preuve de la proposition 1. Par exemple, pour convertir un *décimal* en *binaire* on itère l'opération de quotient par 2 et reste :

$$\begin{array}{lll} 19 & = 2 \cdot 9 + \mathbf{1} & \text{(bit le moins significatif)} \\ 9 & = 2 \cdot 4 + \mathbf{1} & \\ 4 & = 2 \cdot 2 + \mathbf{0} & \\ 2 & = 2 \cdot 1 + \mathbf{0} & \\ 1 & = 2 \cdot 0 + \mathbf{1} & \text{(bit le plus significatif)} \end{array}$$

Donc :

$$\begin{array}{rl} 19 & = 2 \cdot (2 \cdot (2 \cdot (2 \cdot 0 + \mathbf{1}) + \mathbf{0}) + \mathbf{0}) + \mathbf{1}) + \mathbf{1} \\ & = 2^4 \cdot \mathbf{1} + 2^3 \cdot \mathbf{0} + 2^2 \cdot \mathbf{0} + 2^1 \cdot \mathbf{1} + 2^0 \cdot \mathbf{1} \end{array}$$

La représentation de 19 en base 2 est 10011.

Le type int en python

Le type int permet de représenter les entiers. La grande majorité des langages de programmation prevoient des types qui permettenet de représenter un nombre borné à priori d'entiers (typiquement des entiers sur 8, 16, 32, 64,... bits). Par ailleurs, certains langages disposent de bibliothèques pour la représentation de 'grands entiers' (typiquement des entiers avec de l'ordre de 10^3 chiffres). En python, on ne fait pas de différence entre petits et grands entiers et le type int permet de représenter directement des entiers dont la taille est uniquement limitée par la quantité de mémoire disponible. Sur les valeurs de type int on dispose des opérations arithmétiques suivantes : $+$ pour l'addition, $-$ pour la soustraction, $*$ pour la multiplication, $\backslash\backslash$ pour la division entière, $\%$ pour le reste de la division entière et $**$ pour l'exposant.

2.2 Booléens (type bool)

En python on dénote les valeurs booléennes par True et False. Sur ces valeurs on dispose des opérateurs logiques standard and, or et not dont on rappelle le comportement :

x	y	not(x)	and(x, y)	or(x, y)
False	False	True	False	False
False	True	False	False	True
True	False		False	True
True	True		True	True

Il est facile de montrer que ces opérateurs suffisent à exprimer toute fonction qu'on pourrait définir sur des valeurs booléennes. En particulier, on peut exprimer d'autres opérateurs logiques binaires comme l'implication logique, l'équivalence logique, le ou exclusif,...

Les prédicats de comparaison (égalité $==$, différence $!=$, plus petit que $<$, plus grand que $>$, plus petit ou égal $<=$,...) retournent une valeur booléenne. Typiquement on utilise ces prédicats pour écrire des conditions logiques qui vont déterminer la suite du calcul. Bien sûr, les conditions logiques peuvent être combinées à l'aide d'opérateurs logiques. Par exemple, on peut écrire :

$$(\mathsf{x} == \mathsf{y}) \text{ and } (\mathsf{x} < \mathsf{z} + 5 \text{ or } \mathsf{not}(\mathsf{x} == \mathsf{y} + \mathsf{z})) \ .$$

Remarque 2 *En* python*, ainsi que dans d'autres langages, l'évaluation d'une condition logique se fait de gauche à droite et de façon paresseuse, c'est-à-dire dès qu'on a déterminé la valeur logique de la condition on omet d'évaluer les conditions qui suivent. Ainsi, en* python*, on peut écrire la condition logique :*

$$\mathsf{not}(\mathsf{x} == 0) \text{ and } (\mathsf{y}/\mathsf{x} == 3) \tag{2.2}$$

qui ne produit pas d'erreur même si x *est égal à* 0*. En effet, si* x *est égal à* 0 *alors la première condition* $\mathsf{not}(\mathsf{x} == 0)$ *est fausse et ceci suffit à conclure que la condition logique est fausse. Le problème avec ce raisonnement est qu'en* python *une expression logique peut être vraie, fausse, produire une erreur, produire un effet de bord (par exemple lire une valeur) et même faire boucler le programme. Il en suit que la conjonction en* python *n'est pas commutative. Par exemple, la condition :*

$$(\mathsf{y}/\mathsf{x} == 3) \text{ and } \mathsf{not}(\mathsf{x} == 0) \tag{2.3}$$

n'est pas équivalente à la condition (2.2) ci-dessus.

2.3 Flottants (type float)

Représentation

Le proposition 1 sur la représentation des nombres entiers se généralise aux nombres réels.

Proposition 2 *Soit $B \geq 2$ et $x > 0$ nombre réel. Alors ils existent uniques un nombre entier e (l'exposant) et une suite de chiffres $\{d_i \mid i \geq 0\}$ en base B (la mantisse) tels que (i) $d_0 \neq 0$, (ii) pour tout $i \geq 1$ existe $j > i$ tel que $d_j \neq (B-1)$ et (iii) $x = B^e \cdot (\Sigma_{i \geq 1}\ d_i \cdot B^{-i})$.*

Preuve. On esquisse la preuve pour le cas $B = 2$. On a donc :

$$d_0 = 1, \quad d_i \in \{0, 1\}, \quad \forall i \geq 1 \exists j \geq i (d_i = 0)\ . \tag{2.4}$$

En utilisant les propriétés des series géométriques on vérifie :

$$\begin{array}{ll} (1) & 1 = 2^{-0} \leq \Sigma_{i \geq 0}\ d_i \cdot 2^{-i} < 2 \\ (2) & \Sigma_{i \geq s}\ d_i \cdot 2^{-i} < 2^{-s+1} \qquad (s \geq 1) \end{array}$$

Pour tout $x > 0$ nombre réel positif, il existe unique e nombre entier tel que :

$$2^e \leq x < 2^{e+1}\ .$$

Si on pose $h = x/2^e$ on a $1 \leq h < 2$. Il reste maintenant à montrer que pour un tel h il existe une suite unique $d_0, d_1, \ldots$ avec les propriétés (2.4) et telle que : $h = \Sigma_{i \geq 0}\ d_i \cdot 2^{-i}$.

Les chiffres $d_0, d_1, \ldots$ sont détérminées de façon itérative. Au premier pas, si $h = 2^{-0}$ on termine avec $d_0 = 1$ et $d_i = 0$ pour $i \geq 1$. Sinon, on cherche l'unique d_1 tel que :

$$2^{-0} + d_1 \cdot 2^{-1} \leq h < 2^{-0} + (d_1 + 1) \cdot 2^{-1}\ .$$

A nouveau si $h = 2^{-0} + d_1 \cdot 2^{-1}$ on termine et sinon on cherche d_2 tel que :

$$2^{-0} + d_1 \cdot 2^{-1} + d_2 \cdot 2^{-2} \leq h < 2^{-0} + d_1 \cdot 2^{-1} + (d_2 + 1) \cdot 2^{-2}\ .$$

En continuant de la sorte, on montre l'*existence* de la suite $d_1, d_2, \ldots$ Pour l'unicité, on suppose disposer de deux suites qui correspondent au même nombre h et on montre que dans ce cas une des deux suites doit avoir des chiffres 1 à partir d'un certain indice. □

Nombres flottants

Les langages de programmation disposent de un ou plusieurs types qui permettent de représenter les nombres réels (du moins une partie). En général, la représentation et le calcul sur ces représentations entraînent des *approximations*. Pour assurer la fiabilité et la portabilité des programmes, il est alors important d'établir des *normes* que toute mise en oeuvre doit respecter.

Dans ce cadre, la norme *IEEE 754* est de loin la plus importante. Elle fixe la représentation des nombres en *virgule flottante* sur un certain nombre de bits (typiquement 32 ou 64). La norme reprend la notation avec exposant et mantisse utilisée dans la proposition 2. Par exemple, dans le cas où les nombres sont représentés sur 64 bits (on dit aussi en *double précision*) la norme utilise 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse. Comme en base 2 le chiffre à gauche de la virgule est forcement 1, on utilise les 52

bits pour représenter les chiffres binaires à droite de la virgule. Certaines valeurs de l'exposant sont réservées pour représenter le 0 et d'autres nombres non standards ($+\infty, \ldots$).

Les opérations sur les nombres flottants ne sont pas forcement exactes car le résultat théorique de l'opération n'est pas forcement un nombre flottant. Pour cette raison, la norme *IEEE 754* fixe aussi la façon dans laquelle le résultat d'une opération arithmétique ou d'une opération d'extraction de la racine carrée doit être arrondi pour obtenir un nombre flottant.

Erreur absolue et erreur rélative

Soit $\mathbf{R}$ l'ensemble des nombres réels et $\mathbf{F}$ l'ensemble des nombres réels représentables par la machine. On souhaite analyser la façon dans laquelle $\mathbf{F}$ approxime $\mathbf{R}$. Clairement, pour tout $x \in \mathbf{R}$ on peut définir un nombre $\widetilde{x} \in \mathbf{F}$ (pas forcement unique) qui approxime x.

Définition 1 *On appelle* erreur absolue *la quantité* $|\widetilde{x} - x|$ *et et si* $x \neq 0$ *on appelle* erreur rélative *la quantité :*[2]

$$\left|\frac{\widetilde{x} - x}{x}\right| .$$

En pratique, il est bien plus intéressant de contrôler l'erreur relative que l'erreur absolue ! Par exemple, supposons $x = 100.11$ et $\widetilde{x} = 100.1$ alors l'erreur absolue est 10^{-2} et l'erreur relative d'environ 10^{-4}. D'autre part si $x = 0.11$ et $\widetilde{x} = 0.1$ alors l'erreur absolue est toujours 10^{-2} mais l'erreur relative est d'environ 10^{-1}.

Soit $\mathbf{F}$ fini (ce qui est le cas par exemple en double précision), soient f^- et f^+ le plus petit et le plus grand nombre flottant positif dans $\mathbf{F}$. Soit maintenant $x > 0$ (le cas $x < 0$ est symétrique). Que peut-on dire sur l'erreur absolue et relative d'une approximation de x dans $\mathbf{F}$? On distingue 3 cas.

1. Si $x < f^-$ et on pose $\widetilde{x} = 0$ on a :

   $$|x - \widetilde{x}| = x < f^- \quad \text{(borne sur l'erreur absolue)}, \quad \left|\tfrac{x-\widetilde{x}}{x}\right| = 1 \quad \text{(erreur relative)}.$$

   Il est intéressant de noter que si on avait pris $\widetilde{x} = f^-$ on aurait toujours la même borne sur l'erreur absolue mais une erreur relative qui tend vers $+\infty$ pour x qui tend vers 0.

2. Si $x > f^+$ et on pose $\widetilde{x} = f^+$ on a une erreur absolue qui tende vers $+\infty$ pour x qui tend vers $+\infty$ et une erreur relative qui tend vers 1 pour x qui tend vers $+\infty$.

3. Si $x \in [f^-, f^+]$ on considère la situation où on est en virgule flottante en base B et avec une mantisse qui comporte t chiffres. Dans ce cas, l'erreur absolue est au plus la distance entre 2 nombres consécutifs dans $\mathbf{F}$. Cette distance est de la forme B^{e-t}, où l'exposant e peut varier. La distance n'est donc pas constante mais depend de l''ordre de grandeur' des nombres qu'on est en train de considérer : plus le nombre est grand plus l'erreur absolue est grande. En utilisant la proposition 2, on peut supposer que $x = (d_0, d_1 d_2 \cdots d_t d_{t+1} \cdots) \cdot B^e$ avec $d_0 \neq 0$. On obtient donc la borne suivante sur l'erreur relative :

   $$\left|\frac{x - \widetilde{x}}{x}\right| < \frac{B^{e-t}}{B^e} = B^{-t} . \tag{2.5}$$

2. Ici on prend l'erreur rélative comme une valeur *non-négative*. Dans d'autres contextes, on peut aussi définir l'erreur relative comme le nombre $\epsilon = (\widetilde{x} - x)/x$ ce qui permet de dire que $\widetilde{x} = (1 + \epsilon) \cdot x$.

Il est remarquable que cette borne depend seulement du nombre de chiffres de la mantisse. Par exemple, en double précision on obtient une borne de 2^{-52} sur l'erreur relative.

Remarque 3 *On pourrait penser qu'une erreur relative de 2^{-52} est négligeable ($2^{-52} \approx 2 \cdot 10^{-16}$). En particulier, si on se place dans le cadre de mésures physiques une erreur relative de 2^{-52} est très probablement négligeable par rapport à l'erreur de mésuration. Le problème est que les fonctions mises en oeuvre pour approcher les fonctions mathématiques usuelles (opérations arithmétiques, extraction de racine carrée, fonctions trigonométriques, logarithme,...) induisent aussi des erreurs et que ces erreurs se propagent et peuvent s'accumuler jusqu'à rendre le résultat d'un calcul sur les flottants non-significatif.*

2.4 Vecteurs (type tuple)

Si $v_1, \ldots, v_n$ sont des valeurs ($n \geq 0$), on peut construire un vecteur $(v_1, \ldots, v_n)$. En python on appelle le type de ce vecteur une tuple. Si t est une tuple avec n composantes alors on peut accéder ses composantes avec la notation t[i] où $i = 0, \ldots, n-1$. On remarquera qu'on compte les composantes à partir de 0. Si $i \geq n$ l'interprète lève une exception et arrête le calcul. Le langage python introduit aussi d'autres façons d'accéder les composantes.

- Si on écrit t[i] ou $i = -1, -2, \ldots, -n$ on accède respectivement le dernier, l'avant dernier,..., le premier élément. A nouveau si $i < -n$ on a une erreur.
- Si on écrit t[i : j] on retourne une sous-tuple de t dont les composantes sont comprises entre la position i et la position $j - 1$. Ainsi t[0 : n] retourne exactement la tuple de départ. Il est possible d'omettre le premier et/ou le dernier indice. On peut donc écrire t[i :] ou t[:j] ou t[:]. Dans ce cas, l'indice omis à gauche de : est remplacé par 0 et l'indice omis à droite de : est remplacé par le nombre de composantes de la tuple. A noter qu'à la différence des notations qui accèdent exactement une composante, la notation qui utilise ' :' rend toujours une tuple comme résultat. Par exemple, si $j \leq i$ ou si la tuple n'a pas de composantes aux positions $i, \ldots, j-1$ on obtient la *tuple vide* qu'on écrit (). Au passage, une tuple avec *une seule composante* s'écrit (v,) (notez la virgule) pour la distinguer de la valeur (v).
- Il est aussi possible d'ajouter une troisième composante et d'écrire t[i : j : k]. Dans ce cas, on commence par i et on ajoute k jusqu'à arriver à j (non-compris). [3]

L'opérateur len permet de connaître le nombre de composantes d'une tuple et l'opérateur + permet de prendre deux tuples x et y et de construire une tuple x + y dont les composantes sont celles de x suivies par celles de y.

Digression 3 *En* python *toute valeur peut être vue comme un* objet *d'un certain type (on dit aussi* classe*). Comme evoqué dans la section 1.2, un objet est une combinaison de valeurs et de fonctions qui opèrent sur l'objet. Une fonction qui opère sur un objet s'appelle aussi* méthode *et les langages à objets utilisent une notation avec un 'point' (*dot notation*) pour décrire l'application d'une méthode* m *à un objet* o *avec arguments* $e_1, \ldots, e_n$*, à savoir on écrit :*

$$o.m(e_1, \ldots, e_n)$$

qu'on peut comprendre comme $m(o, e_1, \ldots, e_n)$.

3. Attention, si les indices sont négatifs on a un téléscopage avec la notation qui compte à partir de la dernière position et le comportement n'est pas toujours celui attendu !

Un certain nombres de fonctions disponibles sur les valeurs de type tuple utilisent la notation avec 'point' dont on vient d'expliquer l'origine. Par exemple, si t est une tuple alors :

— t.count(v) retourne le nombre d'occurrences de v dans t,

— t.index(v) retourne le plus petit indice d'une composante de t qui contient v et lève une exception si v n'est pas dans t.

2.5 Chaînes de caractères (type str)

En python, on appelle str (pour *str*) le type des chaînes de caractères. Les chaînes de caractères sont une spécialisation des vecteurs dans laquelle les valeurs des composantes sont les caractères du clavier (qui ont aussi le type str). Plutôt qu'écrire ('a','b','7'), on utilise la notation plus compacte 'ab7'. Les notations décrites dans la section 2.4 pour accéder les composantes des tuples s'appliquent aussi bien aux chaînes de caractères. Les opérateurs len (longueur) et + (concatenation) ainsi que les méthodes count et index sont aussi disponibles. La fonction chr permet de passer d'un entier compris entre 0 et 1114111 à un caractère Unicode et la fonction ord permet de faire l'opération inverse. Par exemple, les 26 caractères minuscules de l'alphabét français correspondent aux entiers compris entre 97 et 122.

2.6 Conversions

En programmation comme en mathématiques on a souvent besoin de transformer ou convertir une valeur d'un certain type dans une valeur d'un autre type. Par exemple, on peut vouloir convertir un nombre flottant (réel) en un entier.

Le langage python dispose d'une fonction type qui peut être appliquée à toute valeur et qui retourne le type de la valeur. On remarquera qu'en python les types ne fournissent pas d'information sur la structure interne d'une valeur. Par exemple, les vecteurs $(1,'a')$ et $('a', 1.0, \mathsf{True})$ ont le même type tuple qui ne dépend pas du type des composantes.

Le langage python propose aussi des fonctions qui permettent (dans une certaine mesure) de convertir une valeur d'un type à un autre.

Une conversion qui est effectuée automatiquement est celle des entiers aux flottants. Ainsi si on écrit 3 + 5.5 le langage convertit automatiquement la valeur entière 3 en flottant. Notez que si le nombre entier est trop grand la conversion entraine une erreur de debordement.

Pour convertir une valeur flottante f à un entier on écrira int(f). La conversion est effectué simplement en tronquant la partie décimale de f.

On convertit une valeur booléenne en un entier ou en flottant en utilisant la convention que True correspond à 1 et False à 0. Dans l'autre direction, on convertit un nombre en un booléen en supposant que 0 correspond à False et tout nombre différent de 0 à True. [4]

De façon plus surprenante, il est possible de convertir (certaines) chaînes de caractères en nombres entiers ou flottants. Pour ce faire, il faut que la chaîne en question puisse être vue comme un nombre. On remarquera que cette possibilité est utilisée chaque fois qu'on veut lire des valeurs numériques avec la fonction input. Pour convertir en entier, la chaîne doit être composée d'une suite de chiffres qui est éventuellement précédé par un signe. Pour convertir en flottant, on pourra en plus séparer la suite de chiffres par un point . (la virgule) et aussi utiliser la notation avec mantisse et exposant comme dans la chaîne '0.99e-43'.

4. Cette convention est bien sûr arbitraire et semble s'inspirer de la représentation des valeurs booléennes par des nombres dans des langages comme C.

Pour convertir une chaîne de caractères en un vecteur on se base sur le fait qu'une chaîne de caractères est rien d'autre qu'un vecteur de caractères. Ainsi `tuple('a77')` donne le vecteur `('a','7','7')`. A noter que la fonction `tuple` appliquée à un nombre ou à un booléen produit une erreur.

Enfin il est possible de convertir toute valeur en une chaîne de caractères. Par exemple, `str(('a',7))` donne "`('a', 7)`". [5]

5. C'est une situation dans laquelle il y a une différence entre les guillemets doubles ou simples. La règle est que si la chaîne contient des guillemets simples alors elle doit être délimitée par des guillemets doubles.

Chapitre 3

Contrôle

On peut voir le corps de chaque fonction comme une suite de *commandes*. Dans une grande partie des programmes étudiés jusqu'à maintenant on a une *liste* de commandes qu'on exécute *une fois* dans l'ordre. On va présenter des opérateurs qui permettent d'exécuter les commandes selon un ordre plus élaboré. Par exemple :

- on exécute une commande seulement si une certaine condition logique est satisfaite,
- on répète l'exécution d'une commande tant qu'une certaine condition logique est satisfaite,
- on arrête l'exécution d'une suite de commandes pour sauter directement à l'exécution d'une commande plus éloignée.

Digression 4 *Pour apprendre à* écrire*, il est aussi important de connaître la* grammaire *que de lire les* classiques*. De la même façon, pour apprendre à* programmer*, il convient de maîtriser les* règles du langage *et en même temps d'étudier un certain nombre d'*exemples classiques*. En essayant de reproduire les 'classiques', vous comprendrez mieux les* règles *du langage et vous développerez votre propre* style *de programmation. Dans ces notes de cours, on va examiner un certain nombre d'algorithmes classiques. Le lecteur est averti que leur programmation correspond au* style *de l'auteur de ces notes. Des variations et des améliorations sont certainement possibles et encouragées !*

3.1 Commandes de base et séquentialisation

Les commandes de base comprennent l'affectation d'une valeur à une variable, la commande d'écriture (print) et de lecture (input), l'appel et le retour de fonction (return). Il est aussi possible de composer les commandes pour obtenir des commandes plus complexes. Le premier opérateur de composition est la *séquentialisation* qui dans de nombreux langages est dénoté par le point virgule :

$$C_1; C_2 \ .$$

L'interprétation de cette commande composée est qu'on exécute d'abord la commande C_1 et ensuite la commande C_2. On notera que l'opération de séquentialisation est *associative* :

$$(C_1; C_2); C_3 \equiv C_1; (C_2; C_3)$$

il est donc inutile de mettre les parenthèses et il suffit d'écrire une liste de commandes. Le langage python se distingue d'autre langages de programmation par le fait qu'il remplace les

points virgules et les parenthèses par une discipline très stricte et un peu fragile qui règle l'indentation des commandes. Ainsi en python pour écrire une suite de commandes de base $C_1; \ldots; C_n$ on écrit une commande par ligne en faisant attention à que chaque commande ait la même indentation (espacement par rapport au début de la ligne). [1] On verra que les choses se compliquent quand on commence à composer ces suites de commandes avec les branchements, les boucles et les déclaration de fonctions.

3.2 Branchement

Un deuxième exemple d'opérateur de composition de commandes est le branchement. En python la forme de base est (notez l'indentation) :

```
if (b) :
    C_1
else :
    C_2
```

L'interprétation est qu'on évalue une *condition logique* b. Si elle est vraie on exécute la suite de commandes C_1 et sinon C_2. Dans la syntaxe de python, on admet aussi une version sans branche else :

```
if (b) :
    C_1
```

qui est équivalente à :

```
if (b) :
    C_1
else :
    pass
```

où pass est une commande python qui ne fait rien d'observable. Par ailleurs, si on veut distinguer parmi plus que 2 situations mutuellement exclusives on peut écrire :

```
if (b_1) :
    C_1
elif (b_2) :
    C_2
...
elif (b_n) :
    C_n
else :
    C_{n+1}
```

ce qui veut dire que si la conditions b_1 est satisfaite on exécute C_1, sinon si la condition b_2 est satisfaite on exécute C_2,..., sinon si la condition b_n est satisfaite on exécute C_n et sinon on exécute C_{n+1}. Bien sûr on pourrait exprimer la même chose en utilisant une imbrication (à droite) de branchements binaires mais si les conditions sont nombreuses le code qui en résulte devient illisible.

1. En python, il est aussi possible d'écrire plusieurs commandes séparées par un point virgule sur la même ligne mais ce style de programmation est plutôt découragé.

Exemple 4 *Pour pratiquer le branchement, on considère la conception d'un programme qui lit trois coefficients a, b, c et imprime les zéros du polynôme ax^2+bx+c. Une solution possible est la suivante :*

```
import math
a=float(input('Coeff a?'))
b=float(input('Coeff b?'))
c=float(input('Coeff c?'))
delta = b*b-(4*a*c)
if (a==0 and b==0) :
    if (c==0):                                  #degre 0
        print('Tout nombre est une solution')
    else:
        print('Pas de solution')
elif (a==0):
    sol1=-c/b                                   #degre 1
    print('L'unique solution est :',sol1)
elif (delta==0):                                #degre 2
    sol1=-b/(2*a)
    print('L'unique solution est :',sol1)
elif (delta<0):
    print('Pas de solution')
else:
        root = math.sqrt(delta)
        sol1=(-b+root)/(2*a)
        sol2=(-b-root)/(2*a)
        print('Les deux solutions sont :',sol1,sol2)
```

La première partie du programme lit les coefficients. Dans la deuxième partie on trouve un certain nombre d'instructions de branchement imbriquées qui nous permettent de distinguer les différentes situations qui peuvent se présenter. Il est fortement conseillé de visualiser d'abord avec un schéma qui peut prendre la forme d'un arbre binaire les différentes possibilités. Une fois qu'on a vérifié la correction du schéma on procédera à son codage en python.

3.3 Boucles

Une boucle permet d'exécuter une commande un nombre arbitraire de fois. L'opérateur while est probablement le plus utilisé pour construire une boucle. Sa forme en python est :

$$\begin{array}{l}\mathsf{while}(b):\\ \quad C\end{array} \tag{3.1}$$

La commande résultante évalue la condition logique b et elle termine si elle est fausse. Autrement, elle exécute la commande C et ensuite elle recommence à exécuter la commande (3.1). Ainsi, d'un point de vue conceptuel la commande (3.1) est équivalente à la commande :

$$\begin{array}{l}\mathsf{if}(b):\\ \quad C\\ \quad \mathsf{while}(b):\\ \quad\quad C\end{array}$$

Exemple 5 *On programme l'algorithme d'Euclide pour le calcul du* pgcd *(exemple 3) en utilisant une boucle* while.

```
a=int(input('Entrer a : '))
b=int(input('Entrer b : '))
while not(b==0):
    aux=b
    b=a%b
    a=aux
print('pgcd =',a)
```

Tant que b *n'est pas* 0*, on remplace* a *par* b *et* b *par* a mod b*. Notez que si on écrit :*

```
a=b
b=a%b
```

on n'obtient pas le résultat souhaité car dans la deuxième affectation la valeur de a *n'est pas la bonne. Pour cette raison dans le code ci-dessous on introduit une variable auxiliaire* aux *qui garde la valeur originale de* b *pendant qu'on remplace* b *par* a mod b*. Il s'agit d'une technique standard pour permuter le contenu de deux variables. Ceci dit,* python *permet aussi d'écrire deux (ou plusieurs) affectations simultanées. Ainsi si on écrit :*

$$x_1, \ldots, x_n = e_1, \ldots, e_n$$

on évalue d'abord les expression $e_1, \ldots, e_n$ *et ensuite seulement on affecte les résultats de l´évaluation aux variables* $x_1, \ldots, x_n$*. Donc dans le cas en question aurait pu écrire :*

```
a,b=b,a%b
```

Boucle for

Pour améliorer la lisibilité du programme, on utilise aussi une boucle dérivée for avec la forme :

$$\begin{array}{l}\mathsf{for}\ i\ \mathsf{in}\ \mathsf{range}(\mathsf{start}, \mathsf{stop}, \mathsf{step}) \\ \quad C\end{array} \tag{3.2}$$

En première approximation, la boucle for est équivalente à :

$$\begin{array}{l} i = \mathsf{start} \\ \mathsf{while}(i < \mathsf{stop}) \\ \quad C \\ \quad i = i + \mathsf{step}\end{array} \tag{3.3}$$

où start, stop et step sont des expressions numériques. L'intuition est que start initialise la variable i, qui est incrémentée de step à chaque itération tant qu'elle est strictement plus petite que stop. Il est possible d'omettre l'expression step et dans ce cas il est entendu que step $= 1$. Il est aussi possible d'omettre l'expression start et dans ce cas il est entendu que start $= 0$. Ainsi for i in range n fait varier i de 0 à $n - 1$.

Exemple 6 *On souhaite lire un nombre naturel* n *et imprimer ses diviseurs propres (différents de* 1 *et* n*). Pour résoudre ce problème, on peut utiliser une boucle* for *qui va parcourir les entiers compris entre* 2 *et* $n/2$*.*

```
n=int(input('Entrer n '))
for i in range(2,n//2+1):
    if (n%i==0):
        print(i)
```

Il est aussi possible d'énumérer une suite d'entiers par ordre décroissant en utilisant une valeur step *négative. En continuant notre exemple, on peut imprimer les diviseurs propres par ordre décroissant de la façon suivante :*

```
for i in range(n//2,1,-1):
    if (n%i==0):
        print(i)
```

3.4 Échappatoires

On présente par ordre décroissant de puissance un certain nombre de commandes qui permettent de s'extraire de la commande en exécution et de sauter à un autre point du contrôle :

- sys.exit() pour terminer l'exécution du *programme*. Pour utiliser cette commande en python il faut avoir importé le module sys.
- return pour terminer l'exécution d'une *fonction*.
- break pour terminer l'exécution de la *boucle* dans laquelle on se trouve.
- continue pour *reprendre l'exécution au début de la boucle* dans laquelle on se trouve. [2]

Exemple 7 *Dans la boucle suivante on va imprimer* $5, 4, 3, 2, 1$. *En particulier le décrément après* continue *n'est jamais exécuté.*

```
x=5
while (x>0):
    print(x)
    x=x-1
    if (x>0):
        continue
    x=x-1
```

La boucle suivante ne terminerait pas sans break.

```
n=10
i=0
acc=0
while (i<n):
    acc=acc+i
    i=i-1
    if (i<-100):
        break
```

Remarque 4 *Il faut utiliser* break *et* continue *seulement si on se trouve dans une boucle.*

2. Si on se trouve dans une boucle for, continue va quand même exécuter la commande d'incrément/décrément. Pour cette raison, la transformation de la boucle for en boucle while décrite dans (3.3) doit être raffinée.

Chapitre 4

Fonctions

Un programme python est composé d'une liste de fonctions qui peuvent s'appeler mutuellement. Les fonctions sont un élément essentiel dans la modularisation et le test d'un programme.

Si une tâche doit être répétée plusieurs fois c'est une bonne pratique de lui associer une fonction ; ceci permet de produire un code plus compact tout en clarifiant le fonctionnement du programme.

Aussi si une tâche est trop compliquée il est probablement utile de la décomposer en plusieurs fonctions. Le code de chaque fonction devrait tenir dans une page (20-30 lignes) et avant de tester le programme dans son intégralité, il convient de s'assurer de la fiabilité de chaque fonction.

4.1 Appel et retour d'une fonction

Comme indiqué dans la section 1.2, une fonction est un *segment de code* identifié par un *nom*. En python, la forme d'une fonction est la suivante :

```
def f(x1,...,xn) :
    corps de la fonction
```

La première ligne spécifie le nom de la fonction (f) et les noms des arguments (x1,..., xn). Ces noms peuvent être vus comme des variables locales à la fonction qui sont initialisées au moment de l'appel de la fonction. Par exemple, dans un appel $\mathsf{f}(\mathsf{e}_1, \ldots, \mathsf{e}_n)$ on évalue les expressions $\mathsf{e}_1, \ldots, \mathsf{e}_n$ et on affecte leurs valeurs aux variables $\mathsf{x1}, \ldots, \mathsf{xn}$. On dit que l'appel d'une fonction est *par valeur*. Si la fonction est censée rendre un résultat alors le corps de la fonction doit contenir des commandes de la forme return e où e est une expression. Quand on exécute une de ces commandes, l'exécution de la fonction en question termine en rendant à la fonction appelante la valeur de l'expression e.

Remarque 5 *En* python *le* NoneType *est un type (immuable) qui contient une seule valeur* None *et aucune opération. Cette valeur peut être utilisée, par exemple, pour rendre totale une fonction partielle ou pour remplir la composante d'une tuple pour laquelle aucune autre valeur significative est disponible. Par défaut, une fonction qui ne rend pas de valeur avec la commande* return *va retourner une valeur* None.

Exemple 8 *Voici un programme simple composé de* 3 *fonctions et d'une variable globale* pi. *Une variable globale est une variable dont la déclaration n'est pas dans le corps d'une fonction. Une variable globale est visible dans toutes les fonctions du programme sauf si elle est cachée par une déclaration locale (plus de détails dans la section 4.2). On remarquera que les appels de fonction peuvent être imbriqués comme dans* imprimer(1.0, circonference(1.0)) ; *Dans ce cas, il faut d'abord exécuter l'appel interne (*circonference(1.0)*) et ensuite celui externe* imprimer$(1.0, 6.28\cdots)$

```
import math
pi = math.pi
def circonference(r):
    return 2 * pi * r
def imprimer(r, c):
    print('La circonference de ', r, 'est ', c)
def main():
    imprimer(1.0, circonference(1.0))
    imprimer(2.0, circonference(2.0))
main()
```

Remarque 6 *Une fonction* python doit *prendre n* arguments *($n \geq 0$) et éventuellement rendre un* résultat. *Par ailleurs, comme effet de bord, elle* peut *aussi* lire *des valeurs (avec* input *par exemple) et* imprimer *des valeurs (avec* print *par exemple). Il faut bien comprendre que :*

— *prendre en argument est différent de lire une valeur.*

— *rendre un résultat est différent d'imprimer une valeur.*

Un argument *est passé par la fonction appelante alors que la* valeur lue *vient de l'écran ou d'un fichier. De même une fonction rend le* résultat *à la fonction appelante alors qu'elle* imprime une valeur *à l'écran ou dans un fichier. Prendre en argument/rendre un résultat est donc une interaction entre deux fonctions alors que lire une valeur/imprimer une valeur est une interaction entre une fonction et l'utilisateur (ou un fichier externe au programme).*

4.2 Portée lexicale

Dans un programme, en particulier dans un programme avec plusieurs fonctions, la même variable peut être déclarée plusieurs fois.

Une bonne pratique consiste à utiliser des noms différentes pour les arguments et les variables locales et si possible à éviter d'utiliser le même nom pour les variables locales et les variables globales. A défaut, la variable locale couvrira la globale.

Une spécificité de python est qu'il impose à chaque fonction de déclarer explicitement les variables *globales* qu'elle souhaite *modifier* (pour la lecture la déclaration n'est pas nécessaire). On a donc deux possibilités :

— on peut *lire* une variable globale dans le corps d'une fonction si la fonction ne contient pas une variable locale ou un paramètre avec le même nom,

— on peut *lire et écrire* une variable globale x dans le corps d'une fonction à condition d'insérer la déclaration global x dans le corps. [1]

Exemple 9 *Voici un programme composé de* 3 *variables globales et* 3 *fonctions. Quels sont les entiers imprimés ? Voici des indications :*

1. Insérer une déclaration global x dans une fonction qui a déjà une variable locale x provoque une erreur.

— *La fonction* f *a une variable locale* x *et une variable globale* y,
— *la fonction* portee *a une variable locale* x *et deux variables globales* y *et* z,
— *la fonction* main *a deux variables globales* x *et* y. .

Si on supprime la ligne (11) *l'interprète lève une exception car dans* (12) *on cherche à modifier la variable* y *qui n'est pas locale à* portee.

```
x=5                  #(1)
y=6                  #(2)
z=7                  #(3)
def f(x):            #(4)
    print(x)         #(5)
    print(y)         #(6)
def portee(x):       #(7)
    x=x+1            #(8)
    f(x)             #(9)
    f(z);            #(10)
    global y         #(11)
    y=y+1            #(12)
    f(y)             #(13)
def main():          #(14)
    global y         #(15)
    y=1              #(16)
    portee(x)        #(17)
    f(x)             #(18)
main()
```

4.3 Décomposition en fonctions

On étudie un exemple qui illustre entre autres comment on décompose un problème en un certain nombre de fonctions.

On souhaite disposer d'un programme qui lit des chaînes de caractères de l'écran. A chaque lecture, le programme se comporte de la façon suivante :

— s'il a lu une chaîne qui n'est ni le mot stop ni un *nombre naturel* alors il imprime un message d'erreur et il termine,
— sinon, s'il a lu le mot stop et auparavant il n'a pas lu au moins un entier alors il imprime un message d'erreur et il termine,
— sinon, s'il a lu le mot stop alors il imprime le plus grand entier et le plus petit entier lus depuis le démarrage du programme et il termine,
— sinon, le programme procède à la prochaine lecture.

On va commencer par faire l'hypothèse qu'un nombre naturel est représenté comme une suite de chiffres en base 10. Pour reconnaître les entrées bien formées on va utiliser le fait que si d est un caractère qui correpond à un chiffre décimal (donc $d \in \{'0', '1', \ldots, '9'\}$) alors $48 \leq \mathsf{ord(d)} \leq 57$.

On peut maintenant réflechir à une décomposition en plusieurs fonctions. Il nous faut une fonction qui s'occupe de l'interaction avec l'utilisateur et il semble une bonne idée de déléguer à une autre fonction la vérification de la bonne formation des données fournies par l'utilisateur.

```
def isnum(x):
    for i in range(len(x)):
        if (ord(x[i])<48) or (ord(x[i])>57):
            return False
    return True
def interaction():
    max=None
    min=None
    while True:
        x=input('entrez un nombre naturel ')
        if ((x=='stop') and (max==None)):
            print('je ne peux pas repondre')
            break
        elif (x=='stop'):
            print('Min=',min,'Max=',max)
            break
        elif not(isnum(x)):
            print('pas un nombre naturel')
            break
        elif (max==None):
            min=int(x)
            max=int(x)
        else:
            if (int(x)<min):
                min=int(x)
            elif (max<int(x)):
                max=int(x)
interaction()
```

4.4 Entre itération et récursion (évaluation de polynômes)

Un programme python est composé d'une liste de fonctions qui peuvent s'appeler mutuellement. En particulier, une fonction peut s'appeler elle même ; il s'agit alors d'un exemple de *fonction récursive* (n fonctions qui s'appellent mutuellement sont aussi des fonctions récursives). On a déjà examiné dans l'exemple 3 la programmation de l'algorithme d'Euclide par une fonction récursive. Les fonctions récursives permettent de programmer aisément les définitions par récurrence qu'on trouve souvent en mathématiques. Aussi, un certain nombre d'algorithmes qui suivent une stratégie diviser pour régner se programment naturellement de façon récursive ; par exemple, la recherche dichotomique et certains algorithmes de tri suivent cette stratégie.

On a vu dans l'exemple 5 que l'algorithme d'Euclide peut se programmer aussi avec une *boucle* ; on dira aussi de façon *itérative*. La récursion utilisée dans la programmation de l'algorithme d'Euclide est de type *terminal* dans le sens qu'après l'appel récursif il ne reste plus rien à faire et la fonction retourne immédiatement. [2] Il se trouve que pour la *récursion terminale* (*tail recursion* en anglais), un compilateur optimisant peut générer automatiquement un programme itératif équivalent.

Dans la suite on pratique la programmation récursive et itérative dans le cadre du problème de l'évaluation de polynômes. Considérons le problème de l'évaluation d'un polynôme de degré n :

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n$$

2. Il s'agit d'une intuition, on ne donnera pas de définition formelle de la récursion terminale.

dans un point x. Un premier algorithme peut consister à calculer les sommes partielles :

$$a_0, \qquad a_0 + a_1 x, \qquad a_0 + a_1 x + a_2 x^2, \cdots$$

en calculant en parallèle les puissances de x :

$$x^0, \qquad x^1 = x \cdot x^0, \qquad x^2 = x \cdot x, \qquad x^3 = x \cdot x^2, \ldots$$

Pour ce calcul, il faut donc effectuer $2 \cdot n$ multiplications ainsi que n sommes. Cependant le coût d'une somme est bien inférieur à celui d'une multiplication et donc on peut considérer que le coût du calcul dépend essentiellement du nombre de multiplications.

Règle de Horner

La règle de Horner est un autre algorithme pour évaluer un polynôme de degré n dans un point qui demande seulement n multiplications. On définit :

$$\begin{array}{lcll} h_0 & = & a_n & \\ h_i & = & h_{i-1} x + a_{n-i} & 1 \leq i \leq n \ . \end{array} \tag{4.1}$$

On remarque que :

$$h_i = a_n x^i + a_{n-1} x^{i-1} + \cdots + a_{n-i+1} x + a_{n-i} \ .$$

Donc $p(x) = h_n$ et on peut calculer $p(x)$ avec seulement n *multiplications* !

Pour mettre en oeuvre la règle on va faire l'hypothèse que le programme lit le degré du polynôme, un point où il faut évaluer le polynôme et les coefficients du polynôme. Pour mémoriser (de façon simple) les $n+1$ coefficients où n est variable on va construire une tuple en faisant attention à placer le coefficient i en position i. On a maintenant deux possibilités : soit on définit une fonction horner_rec qui suit la définition (4.1) soit on réorganise le calcul de façon itérative comme dans la fonction horner_it. Dans le premier cas la fonction à un argument i qui va diminuer à chaque itération. Dans le deuxième cas on introduit une boucle for qui fait le même travail.

```
def horner_rec(i,n,x,a):
    assert (i>=0)
    if (i==0):
        return a[n]
    else:
        return (horner_rec(i-1,n,x,a) * x) + a[n-i]
def horner_it(n,x,a):
    h=a[n]
    for i in range(n-1,-1,-1):
        h=a[i]+x*h
    return h
def main():
    n=int(input('Entrez dégré : '))
    x=float(input('Entrez point : '))
    a=()
    for i in range(n,-1,-1):         #on lit an,...,a0
        print('Entrez coeff ', i)
        coef=float(input())
        a=(coef,)+a                  #le dernier coef lu est le plus à gauche
    print(horner_rec(n,n,x,a))
    print(horner_it(n,x,a))
main()
```

Remarque 7 *La solution proposée utilise la commande* assert. *La commande* assert(b) *évalue la condition logique* b. *Si la condition est fausse le calcul s'arrête et un message d'erreur est émis qui permet d'identifier l'assertion qui n'est pas valide. Avec la fonction* assert, *on a une façon simple et fort utile de documenter et tester un programme.*

4.5 La puissance de la récursion (tour d'Hanoï)

Comme on l'a vu dans la section 1.2, l'appel et le retour de fonction manipulent implicitement une *pile de blocs d'activation*. Il s'avère que dans certaines situations cette structure de données permet une programmation élégante et compacte. Dans cette section on illustre une telle situation avec le problème de la tour d'Hanoï qui est bien connu.

On dispose de 3 pivots et de n disques de diamètre différent qu'on peut enfiler dans les pivots. Au début du jeu, tous les disques sont enfilés sur le premier pivot par ordre de diamètre décroissant (le plus petit diamètre est au sommet).

Une action élémentaire du jeu consiste à déplacer un disque du sommet d'une pile au sommet d'une autre pile en gardant la propriété qu'un disque n'est jamais au dessus d'un disque de diamètre inférieur.

Le problème est de trouver une suite d'actions élémentaires qui permettent de transférer la pile de n disques du pivot 1 au pivot 2 (par exemple).

Pour $n = 1$, une suite est $1 \to 2$. Pour $n = 2$, une suite est $1 \to 3; 1 \to 2; 3 \to 2$. Pour $n = 3$, ça devient déjà plus compliqué, mais heureusement la solution du problème s'exprime naturellement de façon *récursive* :

$$\begin{aligned} \mathsf{Hanoi}(i,j,1) &= i \to j \\ \mathsf{Hanoi}(i,j,n) &= \mathsf{Hanoi}(i,k,n-1); i \to j; \mathsf{Hanoi}(k,j,n-1)\ , \end{aligned}$$

où i, j, k sont 3 pivots *distincts*. Le raisonnement est le suivant : pour déplacer n disques du pivot i au pivot j ($i \neq j$), on peut commencer par déplacer $n-1$ disques du pivot i au pivot k ($k \neq i$ et $k \neq j$). Ensuite, on déplace le disque de diamètre maximal qui se trouve au pivot i au pivot j et on termine en déplaçant $n-1$ disques du pivot k au pivot j. Une possible mise en oeuvre en python est la suivante :

```
def hanoi(n, p1, p2):
    p3=troisieme(p1,p2)
    if (n==1):
        print(p1,'->',p2)
    else:
        hanoi(n-1,p1,p3)
        print(p1,'->',p2)
        hanoi(n-1,p3,p2)
```

On laisse au lecteur le soin de programmer la fonction troisieme qui prend $p1, p2 \in \{1, 2, 3\}$ tels que $p1 \neq p2$ et rend l'entier $p3 \in \{1, 2, 3\}$ différent de $p1$ et $p2$. On notera que chaque appel à la fonction hanoi avec $n > 1$ génère deux appels à la même fonction. En particulier, quand on commence à calculer hanoi(n − 1, ...) on utilise la pile des blocs d'activations pour se souvenir qu'il reste encore à exécuter un print ainsi qu'un deuxième appel hanoi(n − 1, ...). Le lecteur est invité à suivre l'exécution du programme hanoi(3,1,2) avec `http://www.pythontutor.com/`.

4.6 Optimisation (Fibonacci)

La suite de Fibonacci est définie par :

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{autrement.} \end{cases}$$

Il y a des mathématiques non-triviales autour de cette suite... mais ici on s'y intéresse parce que elle illustre un *problème de mise en oeuvre* qu'on rencontre parfois dans les définitions récursives. Une mise en oeuvre directe de la fonction f pourrait être la suivante.

```
def fibo_rec(n):
    if (n==0):
        return 0
    elif (n==1):
        return 1
    else:
        return fibo_rec(n-2)+fibo_rec(n-1)
```

Cette solution est particulièrement inefficace car on recalcule plusieurs fois la fonction f sur les mêmes arguments ! Dans le cas de la suite de Fibonacci, il est facile de concevoir une version itérative dans laquelle on calcule $f(0), f(1), f(2), \ldots$ exactement une fois et au pas $i \geq 2$ on se souvient de la valeur de la fonction f dans $i-1$ et $i-2$.

```
def fibo_it(n):
    x=0
    y=1
    if (n==0):
        return x
    elif (n==1):
        return y
    else:
        for i in range(2,n+1):
            x,y=y,x+y
        return(y)
```

Il est intéressant de noter qu'on peut mettre en oeuvre cette même idée en utilisant une fonction récursive un peu plus générale (fonction fibo_aux).

```
def fibo_aux(n, x, y, i):
    if (i==n):
        return y
    else:
        return fibo_aux(n,y,x+y,i+1)
def fibo_rec_eff(n):
    if (n==0):
        return 0
    else:
        return fibo_aux(n,0,1,1)
```

Digression 5 (mémoïsation) *Il existe aussi une technique générale dite de* mémoïsation *qui permet de transformer automatiquement une fonction récursive. On mentionne l'idée générale sans aller dans les détails car on ne dispose pas des structures de données nécessaires*

(les dictionnaires de python *feraient l'affaire). On associe à la fonction une structure de données dans laquelle on* mémoïse *tous les arguments passés à la fonction ainsi que les valeurs retournées. Chaque fois qu'on appelle la fonction avec un argument on vérifie d'abord dans la structure si la fonction a été déjà appelée avec le même argument et dans ce cas on retourne directement le résultat. Autrement, on effectue le calcul et on mémoïse le résultat dans la structure.*

Chapitre 5

Tableaux dynamiques (type list)

Dans ce chapitre, on introduit une variante *mutable* du (constructeur de) type tuple. Ce type de données est typiquement mis en oeuvre en utilisant une structure de données qu'on appelle *tableau dynamique* mais le nom utilisé par python pour désigner ce type est list. [1] On utilisera *tableau* comme synonyme pour une valeur de type list.

Comme pour les tuples on a la possibilité de créer un tableau et d'accéder directement ses éléments. De plus on peut *modifier* autant de fois qu'on le souhaite les éléments d'un tableau et on peut aussi lui ajouter ou supprimer des éléments.

5.1 Création et manipulation de tableaux

On peut créer un tableau en énumérant ses éléments comme dans :

```
t=[0,True,'abba']
```

On remarquera qu'on distingue un tableau d'une tuple par le fait qu'utilise des crochets [ ] au lieu des parenthèses ( ). Cette méthode de création par énumération n'est pas adaptée si l'on souhaite créer un tableau avec un grand nombre d'éléments ou un tableau dont la taille dépend d'un paramètre n. Par exemple, supposons que l'on veuille créer un tableau avec éléments $0^2, 1^2, 2^2, \ldots, (n-1)^2$.

Une première possibilité est de commencer avec un tableau vide et de lui ajouter n éléments comme dans :

```
t=[]
for i in range(n):
    t.append(i**2)
```

Ici append est une méthode définie sur les tableaux qui permet d'ajouter un nouveau élément à la fin du tableau.

Une deuxième méthode consiste à utiliser un mécanisme puissant connu comme *liste en comprehénsion*. Il s'agit d'imiter un procédé standard en mathématiques pour définir des ensembles. Par exemple, en mathématiques on peut définir :

$$\{i^2 \mid i \in \{0, \ldots, n-1\}\} \ .$$

1. Ce choix est assez discutable car en algorithmique les listes sont une structure de données qui a des propriétés assez différentes de celles des tableaux dynamiques.

On appelle ce procédé *définition d'un ensemble en comprehénsion*. De la même façon, python permet de construire un tableau. Dans le cas en question, on peut écrire :

```
[i**2 for i in range(n)]
```

Une fois qu'on a construit un tableau on peut accéder ses éléments avec les mêmes notations utilisées pour les tuples (voir section 2.4) et on dispose toujours de la fonction len pour calculer le nombre d´éléments d'un tableau et de la fonction + pour concaténer deux tableaux.

Une différence fondamentale avec les tuples est qu'on peut *modifier* les éléments d'un tableau. Ainsi, si t est un tableau on peut écrire par exemple :

```
t[i]=t[i]+1
```

ce qui est strictement interdit si t est une tuple. De plus on peut invoquer les méthodes suivantes sur un tableau t :

t.append(v)	(ajoute v à la fin de t)
t.pop()	(supprime le dernier élément de t et retourne sa valeur)
t.sort()	(trie t par ordre croissant)
t.reverse()	(renverse le tableau)
t.insert(i, v)	(insère v à la place i et décale les éléments suivants vers la droite)
t.pop(i)	(supprime le i-ème élément de t et décale les éléments suivants vers la gauche)
t.remove(v)	(supprime la première occurrence de v dans t et décale les éléments suivants vers la gauche)

Comme l'affectation, ces méthodes *modifient* le tableau t. Cette possibilité de modifier une valeur tableau a des implications importantes lorsque on passe un tableau en argument à une fonction. En effet ce qu'on passe est l'adresse du tableau et non pas son contenu et en utilisant cette adresse la fonction appelée peut modifier le contenu d'un tableau crée par la fonction appelante. Par exemple, dans le programme suivant on va imprimer 3 :

```
def f(x):
    x[0]=3
y=[1,2,3]
f(y)
print(y[0])
```

Voici un autre petit exemple qui permet de comparer les variables de type list aux variables d'un type immuable (ici une variable de type tuple). Dans (1), on va créer une nouvelle tuple différente de celle du main alors que dans (2), on va modifier le tableau du main.

```
def f(t,tab):
    t=t+(1,)              #(1)
    tab=tab.append(1)     #(2)
def main():
    t=()
    tab=[]
    f(t,tab)
    print(t,tab)
```

Remarque 8 *On peut se demander pourquoi on passe les variables de type immuable par valeur et les variables de type tableau par adresse. Une raison est que les tableaux peuvent occuper beaucoup de mémoire et que autant que possible il est préférable de les partager plutôt*

que de les dupliquer. En cas de nécessité, il est possible de passer la copie *d'un tableau* a *en utilisant la notation* a[:]. *Donc si on remplace la ligne* (7) *de l'exemple ci-dessus avec* f(a[:],x) *la fonction* f *va opérer sur une* copie *du tableau et donc à la ligne* (9) *on imprime* 0.[2]

Digression 6 *Un point délicat concerne l'estimation du coût des méthodes évoquées ci-dessus. Comme mentionné au début du chapitre, les valeurs de type* list *de* python *sont en réalité des* tableaux dynamiques. *En général, les éléments d'un tableau sont mémorisés dans un segment de mémoire l'un à coté de l'autre. Le fait que les éléments sont contigus permet d'accéder rapidement chaque élément (on dit en temps constant). Des complications apparaissent si l'on veut modifier la taille d'un tableau pendant l'exécution. Une stratégie populaire consiste à doubler la taille du segment de mémoire alloué au tableau chaque fois que le segment est saturé et à le diviser par deux chaque fois que moins d'un tiers du segment est occupé. Chaque expansion et contraction demande une copie de tous les éléments du tableau. On peut montrer que les opérations d'expansion et contraction sont suffisement rares et considérer que (dans un certain sens) les méthodes* append *et* pop *prennent aussi un temps constant. Les méthodes* reverse *et* sort *prennent un temps linéaire et quasi-linéaire, respectivement dans la taille du tableau. Enfin, les méthodes* insert(i,v), pop(i), remove(v) *sont à utiliser avec modération car elles induisent un coût linéaire dans la taille du tableau dans le pire des cas.*

Digression 7 *Si* t *est un vecteur, une chaîne de caractères ou un tableau on peut écrire en* python *:*

```
for x in t:
    C
```

Dans le terminologie python, *on dit que les types* tuple, str *et* list *sont* itérables. *La notation est équivalent à dire (en supposant que* i *est une nouvelle variable qui ne paraît pas dans* C*) :*

```
for i in range(0,len(t)):
     x=t[i]
     C
```

5.2 Étude de cas (primalité et factorisation)

Soit $n \geq 2$. Si n n'est pas premier alors il y a un premier p tel que $p \mid n$ et $p \leq \sqrt{n}$. Donc tout nombre n qui *n'est pas premier* s'écrit comme :

$$n = i \cdot j$$

où : $2 \leq i \leq \sqrt{n}$ et $i \leq j \leq n/i$. La liste des *premiers inférieurs à un* n donné peut être générée avec un algorithme ancien connu comme *crible d'Ératosthène* :

$$f[i] = \mathsf{True} \text{ pour } i = 2, \ldots, n$$

$$\begin{array}{l} \mathsf{pour}\ i = 2, \ldots, \lfloor \sqrt{n} \rfloor \\ \quad \mathsf{pour}\ \ j = i, \ldots, n/i \\ \quad\quad f[i \cdot j] = \mathsf{False} \end{array}$$

2. Cette méthode marche tant qu'on travaille avec des tableaux qui contiennent des valeurs immuables. Si on veut passer, par exemple, une copie d'un tableau de tableaux il faut utiliser une autre méthode.

Exemple 10 *Pour $n = 15$, on obtient :*

$i\backslash j$	2	3	4	5	6	7
2	4	6	8	10	12	14
3		9	12	15		

Exemple 11 *Voici une fonction* filtre *qui prend en entrée un nombre n, construit un tableau de booléens* f*, y applique le filtre d'Erathostène et construit un tableau* prime *avec le nombres premiers inférieurs ou égaux à n.*

```
import math
def filtre (n):
    f=[True for i in range(n+1)]
    r =int(math.sqrt(n))
    for i in range(2,r+1):
        for j in range(i,(n//i)+1):
            f[i*j]=False
    prime=[]
    for i in range(2,n+1):
        if f[i]:
            prime.append(i)
    return prime
```

On a donc une méthode pour générer un segment initial des nombres premiers. Considérons maintenant le problème de savoir si un nombre n est premier. Par exemple, prenons $n = 15413$. On calcule,

$$\lfloor\sqrt{15413}\rfloor = 124 \ .$$

Avec le crible d'Ératosthène, on peut calculer les premiers inférieurs à 124 :

2	3	5	7	11	13	17	19	23	
29	31	37	41	43	47	53	59	61	67
71	73	79	83	97	101	103	107	109	113

Le nombre n est premier si et seulement si aucun de ces nombres divise n. On a donc une méthode qu'on appelle *essai par division* pour savoir si un nombre n est premier.

On rappelle que tout nombre $n \geq 2$ admet une factorisation unique comme produit de nombres premiers. On peut *itérer* l'essai par division pour trouver une factorisation complète :

1. On trouve p_1 tel que p_1 divise n.
2. Si $n' = n/p_1$ est premier on a trouvé la factorisation et autrement on itère sur le nombre n'.

Par exemple, pour $n = 15400$ on trouve :

$$n = 2 \cdot 2 \cdot 2 \cdot 5 \cdot 5 \cdot 7 \cdot 11 \ .$$

On a donc un algorithme pour *factoriser un nombre*. Voici une mise en oeuvre possible de l'algorithme de factorisation qui utilise la fonction filtre. La fonction factorisation calcule les nombres premiers candidats à diviser m et appelle une fonction auxiliaire fact en lui passant en argument : (i) le nombre m qu'on cherche à factoriser, (ii) le tableau des nombres premiers prime calculé par la fonction filtre ainsi que (iii) l'indice i à partir du quel on peut chercher dans prime un facteur et (iv) une valeur n qui est une borne supérieure à la valeur du plus petit diviseur propre de m. Initialement, l'indice i est 0 et la borne n est la racine carrée de m ; pendant le calcul, i va augmenter alors que n va diminuer. A noter qu'on exploite ici l'évaluation paresseuse des conditions logiques (remarque 2).

```
def factorisation(m):
    assert(m>=2)
    n=int(math.sqrt(m))
    prime=filtre(n)
    fact(m,prime,0,n)
def fact(m,prime,i,n):
    while (i<len(prime)) and (prime[i]<=n) and (m%prime[i]!=0):
        i=i+1
    if (i>=len(prime) or (prime[i]>n)):
        print(m)
    else:
        p=prime[i]
        print(p)
        m=m//p
        if (m>=2):
            n=int(math.sqrt(m))
            fact(m,prime,i,n)
```

Digression 8 *On a présenté des algorithmes pour : (i) décider si un nombre est premier et (ii) calculer la factorisation d'un nombre. Ces algorithmes ne passent pas à l'échelle. Par exemple, on sait qu'il y a environ $m/log(m)$ nombres premiers inférieurs à m (il s'agit d'un resultat majeur qui donne des informations assez précises sur la densité des nombres premiers). Si on cherche à savoir si un nombre n est premier, on risque de faire de l'ordre $m/log(m)$ divisions où $m = \sqrt{n}$. Si par exemple $n \approx 2^{1024}$, on obtient $m \approx 2^{512}$ et $m/log(m) \approx 3 \cdot 10^{151}$; un nombre absolument intraitable de divisions ! L'état de l'art est le suivant : on connaît plusieurs algorithmes efficaces pour savoir si un nombre est premier ; par contre, la recherche d'algorithmes efficaces pour la factorisation est un problème ouvert. Néanmoins, on sait améliorer de façon très significative l'algorithme qu'on vient de présenter.*

5.3 Tri à bulles et par insertion

Le tri d'une suite finie d'éléments selon un certain ordre est une *opération fondamentale*. Dans cette section, on présente 2 algorithmes de tri élémentaires : [3]

1. Tri à bulles (*bubble sort*, en anglais).
2. Tri par insertion (*insertion sort*, en anglais).

Par défaut, on fait l'hypothèse que la suite est mémorisée dans un *tableau*.

Tri à bulles

On peut écrire une fonction bulles(i) qui compare les $i-1$ couples aux positions :

$$(0,1),(1,2),(2,3),\cdots(i-1,i)$$

et les permute si elles ne sont pas en ordre croissant. Le coût est linéaire en $i-1$. A la fin de l'exécution de bulles(i) on est sûr que l'élément le plus grand se trouve à la position i. Pour trier, il suffit donc d'exécuter :

$$\mathsf{bulles(n-1)}, \mathsf{bulles(n-2)}, \ldots, \mathsf{bulles(1)} \ ,$$

3. D'autres algorithmes de tri plus efficaces existent dont le tri par fusion (*merge sort*, en anglais) et le tri rapide ou par partition (*quicksort*, en anglais). Une variante de l'algorithme *merge sort* est utilisé dans la mise en oeuvre de la méthode sort de python.

pour un coût qui est de l'ordre de :

$$\Sigma_{i=1,\ldots,n-1}\ i = \frac{n(n-1)}{2}\ .$$

Voici un exemple de fonction python qui prend en argument un tableau et sa taille et le trie selon le principe décrit ci dessus.

```
def bulles(a,i):
    for j in range(i):
        if (a[j]>a[j+1]):
            a[j],a[j+1]=a[j+1],a[j]
def tri_bulles(a):
    n=len(a)
    for i in range(n-1,0,-1):
        bulles(a,i)
```

Tri par insertion

On peut écrire une fonction ins(i) qui, en supposant les éléments aux positions $i+1, \ldots, n-1$ en ordre croissant, va insérer l'élément en position i à la bonne place. Le coût est linéaire en $n-i$.

Pour trier il suffit donc d'exécuter :

$$\mathsf{ins}(n-2), \mathsf{ins}(n-1), \ldots, \mathsf{ins}(0)\ ,$$

pour un coût qui est de l'ordre de :

$$1+2+\cdots+(n-1) = \frac{n(n-1)}{2}\ .$$

Une mise en oeuvre de l'algorithme est ci-dessous. Le lecteur est invité à analyser en détail la fonction ins qui effectue l'insertion d'un élément dans un tableau.

```
def ins(a, i):
    n=len(a)
    k=a[i]
    j=i+1
    while (j<n and k>a[j]):
        a[j-1]=a[j]
        j=j+1
    a[j-1]=k
def tri_ins(a):
    n=len(a)
    for i in range(n-2,-1,-1):
        ins(a,i)
```

5.4 Tableaux à plusieurs dimensions

On peut représenter des matrices en utilisant des tableaux de tableaux. Dans le code suivant :

— create va créer une matrice $m \times n$ initialisée avec la valeur v ; on remarquera l'utilisation d'une définition par comprehénsion imbriquée,

— copy effectue une copie (profonde) de la matrice ; si la matrice contient des valeurs immuables alors des modifications sur la copie n'auront pas d'effet sur la version originale,
— dim retourne la dimension de la matrice,
— printmatrix imprime la matrice en utilisant un argument optionnel end de la fonction print qui permet d'imprimer une tabulation au lieu d'un retour de ligne,
— mult calcule le produit de deux matrices en utilisant 3 boucles for imbriquées et en effectuant de l'ordre de n^3 multiplications, [4]
— trans calcule la transposée en utilisant aussi une définition par comprehénsion imbriquée.

```
def create(m,n,v):
    return [[v for j in range(n)] for i in range(m)]

def copy(a):
    return [a[i][:] for i in range(len(a))]

def dim(a):
    return (len(a),len(a[0]))

def printmatrix(a):
    (m,n)=dim(a)
    for i in range(m):
        for j in range(n):
            print(a[i][j],end='\t')
        print('')

def mult(a,b):
    (m,n)=dim(a)
    (n1,p)=dim(b)
    assert(n==n1)
    c=create(m,p,0)
    for i in range(m):
        for j in range(p):
            for k in range(n):
                c[i][j]=a[i][k]*b[k][j]+c[i][j]
    return c

def trans(a):
    m,n=dim(a)
    return [[a[i][j] for i in range(m)] for j in range(n)]
```

4. Le produit d'une matrice A de dimension $m \times n$ par une matrice B de dimension $n \times p$ est une matrice C de dimension $m \times p$ dont la composante $c_{i,j}$ est le produit scalaire de la ligne i de A par la colonne j de B.

Chapitre 6

Complexité

On introduit la notion de complexité asymptotique dans le pire des cas d'un algorithme. Il s'agit d'une mesure qui n'est pas très sensible aux détails de la mise en oeuvre et qui permet d'avoir une première estimation de l'efficacité d'un algorithme.

On considère aussi des méthodes probabilistes pour *tester* la correction et l'efficacité d'un programme et on évoque des notions alternatives de complexité (complexité moyenne et complexité amortie).

6.1 O-notation et complexité dans le pire des cas

Définition 2 (O-notation) *Soient $f, g : \mathbf{N} \to \mathbf{N}$ deux fonctions sur les nombres naturels. On dit que f est $O(g)$ si :*

$$\exists\, k, n_0 \geq 0 \;\; \forall\, n \geq n_0 \;\; f(n) \leq k \cdot g(n) \;\; .$$

Exemple 12 *Voici des exemples et des non-exemples.*

3457	*est*	$O(1)$
$25 \cdot n + 32$	*est*	$O(n)$
$7 \cdot n \cdot log(n) + 1$	*est*	$O(n \cdot \log_2(n))$
$n \cdot \sqrt{n} - 50$	*n'est pas*	$O(n \cdot log(n))$
3^n	*n'est pas*	$O(2^n + n^5)$.

Définition 3 (fonction de coût) *Soit A un algorithme qui termine. On associe à A une fonction de côut $c_A : \mathbf{N} \to \mathbf{N}$ telle que, pour tout n, $c_A(n)$ est le coût maximal d'une exécution de l'algorithme A sur une entrée de taille au plus n.*

Typiquement, la taille d'une entrée est le nombre de bits nécessaires à sa représentation et le coût d'une exécution est le *temps* mesuré comme le nombre d'*étapes élémentaires* de calcul. Ce qui constitue une étape élémentaire dépend du modèle de calcul. Par exemple, on peut considérer qu'un accès à la mémoire principale ou la multiplication de deux entiers sur 64 bits prennent un temps borné par une constante. D'autre part, dans certaines applications les données ne peuvent pas tenir en mémoire principale ou alors on est amené à traiter des entiers avec un grand nombre de chiffres. Dans ces cas, le coût de ces opérations sera fonction de la taille de la mémoire nécessaire à l'exécution du programme ou de la taille des entiers à

traiter, respectivement. On remarque que la fonction c_A est bien définie car A termine et il y a un nombre fini d'entrées possibles de taille au plus n. On note aussi que par définition c_A est croissante : si $n \leq n'$ alors $c_A(n) \leq c_A(n')$.

Définition 4 (complexité asymptotique) *Un algorithme A est $O(g)$ si sa fonction de coût c_A est $O(g)$.*

Remarque 9 *La notation O nous donne une* information synthétique sur l'efficacité *d'un algorithme/programme. Mais notez que :*

- *Il s'agit d'une* borne supérieure*.*
- *On considère le* pire des cas*.*
- *On* cache les constantes*. Un algorithme qui prend $3n^2$* msec *est utilisable, un algorithme qui prend $2^{80}n$* msec *ne l'est pas.*
- *Le coût d'une opération élémentaire sur une* vraie machine *peut varier grandement. Par exemple on peut avoir un facteur 10^2 entre un* cache hit *(la donnée est en mémoire* cache*) et un* cache miss *(elle n'y est pas). Les optimisations effectuées par le compilateur peuvent avoir un impact important sur la complexité observée.*
- *Dans les calculs en virgule flottante, on doit aussi se soucier de la* stabilité numérique *des opérations.*

Pour toutes ces raisons, dans les applications, la borne O doit être confortée par une analyse plus fine *et des* tests*.*

6.2 Étude de cas : exposant modulaire

On considère un exemple d'analyse de complexité. On suppose que les entrées sont des nombres naturels représentés en base 2. Donc la taille d'un entier m est approximativement $\log_2(m)$. Le lecteur peut vérifier que l'algorithme standard qui calcule l'*addition* de deux nombres est $O(n)$ et que celui qui calcule la *multiplication* est $O(n^2)$ (n étant la taille de l'entrée). Au passage, on remarquera que pour représenter l'addition et la multiplication de deux nombres sur n bits il faut $n+1$ bits et $2 \cdot n$ bits, respectivement.

Considérons maintenant la situation pour la fonction d'*exponentiation*. Avec n bits on représente les nombres dans l'intervalle $[0, 2^n - 1]$. Si on prend $x \in [0, 2^n - 1]$ on aura $2^x \in [1, 2^{2^n-1}]$ et il faudra environ 2^n bits pour représenter le résultat. Avec une représentation standard des nombres, tout algorithme qui calcule la fonction exponentielle prendra au moins un temps exponentiel. En effet la simple écriture du résultat peut prendre un temps exponentiel.

Soient a et e des nombres naturels. Combien de multiplications faut il pour calculer l'exposant a^e ? Un algorithme possible est de calculer :

$$a_1 = a,\ a_2 = (a_1 \cdot a),\ \cdots,\ a_e = (a_{e-1} \cdot a)\ .$$

Cet algorithme effectue $e-1$ multiplications ce qui est *exponentiel* dans $\log_2(e)$ (à savoir la taille de e !). Mais il y a une autre méthode de calcul dites des *carrés itérés*. Soit

$$e = \Sigma_{i=0,\ldots,k}\ e_i 2^i$$

l'*expansion binaire* de e. Donc $e_i \in \{0,1\}$. On applique les *propriétés de l'exposant* pour dériver :

$$\begin{aligned} a^e &= a^{\Sigma_{i=0,\ldots,k}\ e_i 2^i} \\ &= \Pi_{i=0,\ldots,k}(a^{2^i})^{e_i} \\ &= \Pi_{0\leq i\leq k, e_i=1}(a^{2^i})\ . \end{aligned}$$

On a alors l'*algorithme* suivant :

1. On calcule a^{2^i} pour $0 \leq i \leq k$. En remarquant que

$$a^{2^{i+1}} = (a^{2^i})^2\ .$$

   Ainsi k opérations de multiplication (ou élévation au carré) sont nécessaires.

2. On détermine a^e comme le produit des a^{2^i} tels que $e_i = 1$. Au plus k *multiplications* sont nécessaires.

On arrive ainsi à une situation qui semble contradictoire : le calcul de l'exposant est forcement *exponentiel* mais on peut le calculer avec un nombre *linéaire* de multiplications. Le fait est que les multiplications opèrent sur des données dont la taille peut doubler à chaque itération. Donc à la dernière itération on peut devoir multiplier deux nombres dont la taille est exponentielle en la taille des données en entrée. Mais tout n'est pas perdu ! On peut contrôler la taille des données si l'on passe à l'arithmétique modulaire. L'exposant modulaire :

$$(a^e) \mod m$$

prend en entrée 3 entiers : la base a, l'exposant e et le module m. On suppose $0 \leq a, e \leq m$. Pour représenter l'entrée on a donc besoin d'environ $3 \cdot k$ bits où $k = \log_2(m)$. La *multiplication* de deux nombres de k bits demande $O(k^2)$. Le calcul du *reste* de la division d'un nombre de $2k$ bits (la multiplication de 2 nombres de k bits) par un nombre de k bits (le module) peut aussi se faire en $O(k^2)$. Le calcul de l'exposant modulaire demande au plus $2k$ multiplications et calculs du reste. On doit donc effectuer $O(k)$ opérations dont le coût est $O(k^2)$ ce qui donne $O(k^3)$.

Exemple 13 *On souhaite calculer* $3^{25} \mod 7$. *Dans la suite, toutes les congruences sont modulo* 7. *En base* 2 *la représentation de* 25 *est* 11001 *On calcule, les carrés itérés :*

$$3^{2^0} \equiv 3, \quad 3^{2^1} \equiv 2, \quad 2^{2^2} \equiv 4, \quad 2^{2^3} \equiv 2, \quad 2^{2^4} \equiv 4,$$

et on multiplie les carrés qui correspondent aux 1 *de la représentation en base* 2, *soit :*

$$3^{25} \equiv 3^{16} \cdot 3^8 \cdot 3^1 \equiv 4 \cdot 2 \cdot 3 \equiv 3\ .$$

Digression 9 *La borne* $O(k^3)$ *est une* borne supérieure. *En effet, on peut faire un peu mieux. Par exemple, avec l'algorithme de Karatsuba on peut multiplier deux nombres de* k *chiffres en* $O(k^{1,59})$, *au lieu de* $O(k^2)$. *Par ailleurs, l'algorithme présenté est* pratique *; il est couramment utilisé dans les applications cryptographiques avec* $k \approx 10^3$.

6.3 Tests de correction et de performance

Génération aléatoire

Pour *tester* la correction et l'efficacité d'un programme, il est très utile de générer de façon automatique et aléatoire ses entrées. Par ailleurs, certains programmes dits *probabilistes* ont besoin de nombres aléatoires pendant le calcul. En pratique, tout langage de programmation dispose d'un générateur de nombres (plus ou moins) aléatoires.

En python, on peut importer le module random. Ensuite, en appelant random.randint(n,m), on génère un entier avec une distribution uniforme dans l'intervalle $[n, m]$. Techniquement la fonction randint génère de façon déterministe (SIC !) une séquence de nombres à partir d'un *germe* (*seed*, en anglais). On parle aussi d'une séquence pseudo-aléatoire. Intuitivement, il s'agit d'une séquence qu'un observateur extérieur qui ne connaît pas le germe a du mal à distinguer d'une véritable séquence aléatoire. Par défaut, l'interprète utilise le *temps courant* pour initialiser le germe. Si on a besoin de répéter un calcul en utilisant exactement la même séquence (par exemple, pour tester des programmes exactement dans les mêmes conditions), on appelle random.seed(n) qui initialise le germe avec la valeur n.

Permutations aléatoires

On rencontre souvent le problème suivant : à partir d'un générateur aléatoire de nombres, il faut concevoir un programme qui génère des structures avec une certaine distribution. Ici on considère le problème de générer des permutations avec une distribution uniforme. On va représenter une permutation $p : I_n \to I_n$, où $I_n = \{0, \ldots, n-1\}$, par un tableau p qui contient chaque entier dans I_n exactement une fois.

Premier essai Considérez la fonction `permall` suivante qui génère une permutation d'un tableau. On suppose que randint$(0, n-1)$ nous donne un entier dans I_n avec une *distribution uniforme*.

```
import random
def permall(n):
     t=[i for i in range(n)]
     for i in range(n):
          j=random.randint(0,n-1)
          t[i],t[j]=t[j],t[i]
     return t
```

La fonction `permall` génère-t-elle une permutation avec une distribution uniforme ? La réponse est négative !

Analyse

- Chaque chemin d'exécution demande la génération de n nombres entiers dans I_n.
- On a donc n^n chemins possibles et chaque chemin a probabilité $\frac{1}{n^n}$.
- Comme on a $n!$ permutations, si la distribution était uniforme on devrait avoir $\frac{1}{n!} = \frac{k}{n^n}$ pour $k \in \mathbf{N}$. Soit : $n^n = kn!$
- Contradiction ! Par exemple, en prenant $n = 3$.

Deuxième essai Considérez la fonction `permplace` suivante :

```
import random
def permplace (n):
     t=[i for i in range(n)]
     for i in range(n):
         j=random.randint(0,(n-i-1))+i
         t[i],t[j]=t[j],t[i]
     return t
```

Analyse Une *k-séquence* d'un ensemble X de cardinalité n ($n \geq k$) est une liste de k-éléments différents de X. Il y a $\frac{n!}{(n-k)!}$ k-séquences d'un ensemble de n éléments, car :

$$n \cdots n-k+1 = \binom{n}{k} k! = \frac{n!}{(n-k)!} \ .$$

On suppose que les éléments du tableau `t` sont tous différents. On montre par *récurrence* sur $k = 0, 1, \ldots, n$ que la propriété suivante est satisfaite à la k-ème itération de la boucle `for`.

Proposition 3 *Pour toute k-séquence S de l'ensemble $\{\mathtt{t}[0], \ldots, \mathtt{t}[\mathtt{n}-1]\}$ on a $\mathtt{t}[0] \cdots \mathtt{t}[\mathtt{k}-1] = S$ avec probabilité $\frac{(n-k)!}{n!}$.*

Preuve. Pour $k = 0$, S est la séquence vide et $\mathtt{t}[0] \cdots \mathtt{t}[\mathtt{k}-1]$ est aussi la séquence vide. Par ailleurs $\frac{(n-0)!}{n!} = 1$.

On suppose la propriété vraie pour $k < n-1$. Soit $S = S'v$ une $(k+1)$-séquence. On sait que $\mathtt{t}[0] \cdots \mathtt{t}[\mathtt{k}-1] = S'$ avec probabilité $\frac{(n-k)!}{n!}$. Par ailleurs, on a $\mathtt{t}[\mathtt{k}] = v$ avec probabilité $\frac{1}{n-k}$ puisque l'élément est choisi parmi les $n-k$ qui ne sont pas déjà dans S'. Donc la probabilité que $\mathtt{t}[0] \cdots \mathtt{t}[\mathtt{k}] = S$ est :

$$\frac{(n-k)!}{n!} \frac{1}{n-k} = \frac{(n-(k+1))!}{n!} \ .$$

On peut donc conclure que la fonction `permplace` génère une permutation du tableau avec une probabilité uniforme : chaque n-séquence est générée avec probabilité $\frac{1}{n!}$. □

Mesurer le temps d'exécution

En python, on peut utiliser la fonction clock() du module time pour estimer le temps d'exécution d'un calcul comme dans le schéma suivant :

```
import time
begin = time.clock();
/* calcul */
end = time.clock();
time_spent = end - begin
```

Par exemple, supposons que l'on souhaite comparer les performances de l'algorithme de tri par insertion de la section 5.3 avec celles de l'algorithme optimisé utilisé dans la méthode sort de python.

En supposant une distribution uniforme des entrées, on peut utiliser la fonction permplace pour générer les entrées. Par exemple, supposons que pour des tableaux de taille n on effectue m tests. Pour avoir un sens de comment le temps de calcul varie avec la taille des tableaux on va commencer avec des tableaux de petite taille et ensuite on va doubler la taille à chaque cycle de tests tant que le temps de réponse reste raisonnable. Pour commencer, on peut prendre $m = 10$ et faire varier n entre 2^5 et 2^{12}.

6.4 Variations sur la notion de complexité

Pour l'instant on s'est limité à étudier la complexité dans le *pire des cas*. On rappelle que ceci veut de dire que le coût $c_A(n)$ est le *coût maximal* sur une entrée de taille au plus n. Il y a des situations dans lesquelles le pire des cas n'est pas forcement très significatif.

Une approche alternative consiste à considérer le *cas moyen*. Ceci revient à faire des hypothèses sur la distribution des entrées (comme on l'a fait dans le test de performance des algorithmes de tri) et ensuite à calculer la moyenne (ou espérance) des coûts.

Exemple 14 *Supposons disposer d'un tableau P qui contient les nombres premiers compris entre 2 et p. Si on tire un nombre x compris entre 2 et p^2 combien de divisions faut-il faire pour savoir s'il est premier ? Ici on suppose qu'on considère les nombres premiers du tableau P par ordre croissant. Dans le* pire des cas*, si le nombre est premier et proche de p^2 le nombre de divisions est environ la taille du tableau P. Cependant si on suppose que le nombre est tiré avec probabilité uniforme on s'attend à faire beaucoup moins de divisions. Par exemple, pour les nombres pairs une seule division suffira !*

Une deuxième approche consiste à considérer le coût d'une suite d'opérations au lieu d'une seule opération et à considérer le coût d'une opération comme la moyenne arithmétique des coûts dans le pire des cas des opérations de la suite. Dans ce cas on parle de *complexité amortie*. Notez qu'on ne fait pas d'hypothèse sur la distribution des entrées et plus en général le calcul des probabilités ne joue pas de rôle dans la complexité amortie. On considère toujours *le pire des cas* mais par rapport à une *longue suite d'opérations* plutôt qu'à une seule opération.

Exemple 15 *On considère un tableau de m éléments dans lequel on peut effectuer les opérations suivantes :*

- *lire un élément,*
- *modifier un élément,*
- *ajouter un élément à la fin du tableau.*

On suppose que initialement on alloue un segment de mémoire qui peut contenir $n = 1$ éléments et que chaque fois que le nombre d'éléments m dépasse la capacité du segment n on double la capacité du segment. Le coût d'une opération sans dépassement est 1 et le coût d'une opération avec dépassement est la taille du segment (on imagine qu'il faut copier tous les éléments dans un segment deux fois plus grand). On considère une suite de p opérations dont le coût est $c_1, \ldots, c_p$. Que peut-on dire sur la moyenne arithmétique des coûts, à savoir :

$$\frac{1}{p}(\Sigma_{i=1,\ldots,p}\ c_i)$$

dans le pire des cas pour p qui tend vers ∞ ? Tant qu'on effectue des opérations de lecture et modification la moyenne des coûts est 1. Pour trouver le pire des cas on a intérêt à maximiser le nombre d'opérations d'ajout qui sont potentiellement coûteuses. Considérons donc une suite d'opérations d'ajout où par simplicité $p = 2^k$. On obtient :

$$\Sigma_{i=1,\ldots,2^k}\ c_i < 2^k + \Sigma_{i=0,\ldots,k-1}\ 2^i = 2^k + (2^k - 1) \approx 2p\ .$$

Donc la moyenne arithmétique tend vers 2 et on peut considérer que le coût amorti *de chaque opération est constant (on paye 2 pour chaque opération).*

Chapitre 7

Recherche du zéro d'une fonction

On étudie deux méthodes de calcul numérique connues comme méthode dichotomique et méthode de Newton-Raphson. Ces méthodes permettent sous certaines conditions de calculer une solution *approchée* d'une équation $f(x) = 0$, où f est une fonction sur les nombres réels.

7.1 Solutions approchées

Soient $a, b \in \mathbf{R}$, $a < b$ et $f : [a, b] \to \mathbf{R}$ une fonction *continue*. On cherche à trouver un $x \in]a, b[$ tel $f(x) = 0$. Par le théorème des valeurs intermédiaires, un tel x existe si $f(a)$ et $f(b)$ ont des signes opposés, soit $f(a) \cdot f(b) < 0$.

Digression 10 *Le théorème des valeurs intermédiaires (ou de Bozano) a plusieurs formulations. Une formulation standard dit que l'image d'une fonction continue sur $[a, b]$ contient l'intervalle $[min(f(a), f(b)), max(f(a), f(b))]$. La preuve de ce théorème utilise de façon essentielle la propriété de complétude des nombres réels, à savoir tout ensemble borné non vide admet un sup.*

Remarque 10 *Remarquons au passage que le problème de trouver un zéro d'une fonction est essentiellement équivalent au problème de trouver un point fixe d'une fonction, à savoir un x tel que $f(x) = x$. Par exemple :*

- *si on cherche un zéro de f, on peut aussi bien chercher un point fixe de la fonction g telle que $g(x) = x - f(x)$,*
- *si on cherche un point fixe de la fonction g, on peut aussi bien chercher un zéro de la fonction $f(x) = x - g(x)$*

Notez que ces transformations ne sont pas uniques. Par exemple, dans le premier cas on peut aussi prendre $g(x) = x - \alpha(x) \cdot f(x)$ à condition que $\alpha(x) \neq 0$.

On connaît un certain nombre de méthodes pour calculer une *solution exacte* d'une équation $f(x) = 0$; par exemple, si f est un polynôme de degré au plus 4. Cependant, cette situation est plutôt l'exception que la règle ! Dans ce chapitre, on s'intéresse plutôt à trouver une *solution approchée*. A savoir, plutôt que donner une méthode pour calculer un x tel que $f(x) = 0$, on va décrire une méthode pour calculer une suite $\{x_n \mid n \geq 0\}$ qui va *approcher* une solution dans le sens que $lim_{n \to +\infty} x_n = x$ et $f(x) = 0$.

En général, il y a plusieurs raisons pour s'intéresser aux solutions approchées :

— on ne connaît pas de méthode pour calculer la solution exacte ; par exemple, pour des polynômes de degré supérieur à 4,
— une solution approchée suffit à nos besoins ; par exemple, on veut connaître $\sqrt{2}$ à 10^{-2} près,
— une solution approchée est plus rapide à calculer ; par exemple, on a un grand système d'équations linéaires.

Une façon importante d'évaluer la qualité d'une solution approchée est d'estimer sa *vitesse de convergence* : combien d'itérations faut-il faire pour approcher la solution à une valeur ϵ près ? L'estimation de la vitesse de convergence est essentiellement un problème d'analyse mathématique.

Très souvent les méthodes approchées sont mises en oeuvre en utilisant la représentation des nombres réels comme *flottants*. Dans ce cas, il y a un deuxième problème qui apparaît, au lieu de calculer la séquence $x_0, x_1, \ldots, x_n$ qui tend vers une solution $\overline{x}$ on calcule une séquence $\widetilde{x}_0, \widetilde{x}_1, \ldots, \widetilde{x}_n$ qui non seulement ne converge pas forcement vers $\overline{x}$ mais peut aussi ne pas converger du tout ! Pour cette raison, en pratique, on utilise souvent les *heuristiques* suivantes :

— On calcule une borne supérieure n au nombre d'itérations nécessaires pour obtenir une estimation satisfaisante de la solution dans le modèle mathématique. En règle générale, il est inutile de chercher une précision meilleure que celle offerte par le standard de calcul sur les flottants dont on dispose (dans notre cas *IEEE-754*).
— On borne par une valeur proche de n le nombre d'itérations qu'on effectue dans la mise en oeuvre du modèle mathématique avec les flottants. Ceci évite que la mise en oeuvre boucle.
— On vérifie *a posteriori* la qualité du résultat obtenu $\widetilde{x}$; par exemple, dans notre cas on peut vérifier si $f(\widetilde{x}) \approx 0$.

7.2 Méthode dichotomique

Soit donc $f : [a, b] \to \mathbf{R}$ une fonction continue avec $f(a) \cdot f(b) < 0$. On considère une suite d'intervalles qui contiennent une solution et dont la longueur est divisée par deux à chaque itération tant qu'on ne converge pas sur une solution. Au pas 0, on prend l'intervalle $[a, b]$. Soit $[x_n, y_n]$ l'intervalle au pas n et soit $m = \frac{x_n + y_n}{2}$:

— si $f(m) = 0$ alors $[x_{n+1}, y_{n+1}] = [m, m]$ et on a trouvé une solution,
— sinon, si $f(m) \cdot f(x_n) < 0$ alors $[x_{n+1}, y_{n+1}] = [x_n, m]$,
— sinon, on a forcement $f(m) \cdot f(y_n) < 0$ car $(f(x_n) \cdot f(y_n) < 0)$ et on pose $[x_{n+1}, y_{n+1}] = [m, y_n]$,

Supposons qu'on arrête l'itération au pas n en rendant $(x_n + y_n)/2$ comme resultat et que $\overline{x}$ est une solution. Comme on sait que $\overline{x} \in [x_n, y_n]$, on dérive que l'erreur absolue est bornée par :

$$\frac{(y_n - x_n)}{2} \qquad \text{(borne sur l'erreur absolu)} \tag{7.1}$$

Notez au passage qu'on a :

$$(y_n - x_n) \leq \frac{(b - a)}{2^n}$$

la longueur de l'intervalle est divisée par deux à chaque itération.

On sait aussi que $min(|a|,|b|) \leq |\overline{x}| \leq max(|a|,|b|)$ et donc, en supposant $min(|x_n|,|y_n|) \neq 0$, l'erreur relative est bornée par :

$$\frac{(y_n - x_n)}{2 \cdot min(|x_n|,|y_n|)} \tag{7.2}$$

Une autre façon d'estimer l'erreur relative (une estimation qui n'est pas une borne supérieure mais qui est peut être plus proche de la realité) est de supposer que $|\overline{x}| \approx |x_n + y_n|/2$. Dans ce cas on obtient la valeur :

$$\frac{(y_n - x_n)}{|x_n + y_n|} \tag{7.3}$$

Voici une façon de mettre en oeuvre la méthode dans laquelle :

- la fonction dicho prend en argument le nom d'une fonction f et l'intervalle $[a,b]$ dans lequel chercher un zéro,
- on utilise une fonction sign pour calculer le signe $(1,-1,0)$ d'une valeur,
- on utilise l'estimation (7.3) comme condition d'arrêt,
- on fixe une limite de 200 itérations,
- on imprime un avertissement si on est 'loin' d'un zéro.

```
def dicho(f,a,b):
    prec=2**-52
    eps=10e-6
    nitmax=200
    fa=f(a)
    fb=f(b)
    fm=f((a+b)/2)
    assert sign(fa)*sign(fb)<0
    nit=0
    while ((b-a)/(a+b)>prec and nit<nitmax):
        m=(a+b)/2
        fm=f(m)
        if fm==0:
            return (m,nit)
        elif sign(fa)*sign(fm)<0:
            b,fb=m,fm
        else:
            a,fa=m,fm
        nit=nit+1
    if abs(fm)>eps:
        print('warning')
    return ((a+b)/2,nit)
```

7.3 Méthode de Newton-Raphson

La méthode dichotomique repose sur des conditions assez faibles (continuité, signes opposés) et converge assez rapidement. On va maintenant introduire une méthode qui converge de façon *très* rapide mais dont les conditions d'application sont plus difficiles à vérifier. En pratique, on utilisera une telle méthode si le comportement de la fonction est bien compris d'un point de vue qualitatif et si l'efficacité est un critère particulièrement important (par exemple, la méthode sert à calculer une certaine fonction mathématique dans un module de calcul scientifique). Alternativement, on peut aussi considérer une mise en oeuvre hybride dans laquelle on alterne la méthode dichotomique avec la méthode de Newton-Raphson.

Soit $f : [a, b] \to \mathbf{R}$ une fonction continue et dérivable. On commence par un point x_0 'assez proche' d'un point $\overline{x}$ tel que $f(\overline{x}) = 0$. Au pas i, on détermine x_{i+1} par la formule :

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}} \ . \tag{7.4}$$

En termes géométriques, ceci revient à déterminer x_{i+1} comme le point dans lequel la droite tangente à la fonction dans x_i intersecte l'abscisse. En supposant $f'(x_i) \neq 0$, de l'équation (7.4) on dérive :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \ . \tag{7.5}$$

On remarque au passage qu'on a transformé le problème de trouver un zéro de la fonction f dans le problème de trouver un point fixe de la fonction g suivante (cf. remarque 10) :

$$g(x) = x - \left(\frac{1}{f'(x)}\right) f(x) \ .$$

Voici une mise en oeuvre minimaliste de la méthode dans laquelle :

- on fixe une petite valeur eps et un nombre maximal d'itérations,
- on emet un avertissement si la dérivée est trop petite (inférieure à eps) ou si on dépasse le nombre maximal d'itérations,
- on prend comme critère d'arrêt le fait que $|x_{n+1} - x_n|$ est petit (inférieur à eps) ; comme pour la méthode dichotomique, d'autres critères pourraient être envisagés.

```
def newton(f,fprime,x):
    eps=10e-10
    nitmax=50
    nit=0
    while (nit<nitmax):
        nit=nit+1
        fp=fprime(x)
        if abs(fp)<eps:
            print('derivee proche de 0 !')
        xnext=x-(f(x)/fp)
        if abs(xnext-x)<eps:
            return xnext
        else:
            x=xnext
    print('pas de convergence')
```

Chapitre 8

Solution d'équations linéaires

On développe une méthode pour résoudre un système d'équations linéaires sur un corps. La méthode consiste à transformer le système en une suite de systèmes *équivalents* jusqu'à le mettre en une forme *triangulaire* dont la résolution est aisée.

8.1 Systèmes d'équations linéaires

Soit $\mathbf{K}$ un corps (rationnels, réels, complexes, entiers modulo un nombre premier,...).

Un système d'équations linéaires est constitué de coefficients $a_{i,j}$ et b_i dans $\mathbf{K}$ pour $i = 1, \ldots, m$ et $j = 1, \ldots, n$, $n, m \geq 1$.

On peut représenter le système de façon plus compacte par une matrice A à coefficients dans $\mathbf{K}$ de dimension $m \times n$ et un vecteur b dans $\mathbf{K}^m$.

Résoudre le système revient à déterminer :

$$\{(x_1, \ldots, x_n) \in \mathbf{K}^n \mid \Sigma_{j=1,\ldots,n}\ a_{i,j}x_j = b_i,\ \text{pour } i = 1, \ldots, m\}$$

D'un point de vue qualitatif, on a 3 résultats possibles :

— la solution est *unique* (ceci est le cas si et seulement si la matrice A est *inversible*, ce qui est aussi équivalent à dire que le *déterminant* de A est différent de 0),

— il n'y a pas de solution,

— il y a plusieurs solutions (une infinité si on travaille avec un corps infini).

Exemple 16

$$\left[\begin{array}{cc|c} 1 & 2 & -1 \\ 3 & 4 & 1 \end{array}\right] \qquad \left[\begin{array}{cc|c} 1 & 2 & -1 \\ -2 & -4 & 1 \end{array}\right] \qquad \left[\begin{array}{cc|c} 1 & 2 & -1 \\ -2 & -4 & 2 \end{array}\right]$$

La solution unique du premier système est $(3, -2)$. *Le deuxième système n'a pas de solution car si* $x + 2y = -1$ *alors on devrait voir* $(-2)(x + 2y) = -2x - 4y = -2$ *et* $-2 \neq 1$. *Le troisième système a une infinité de solutions de la forme* $((-1 - 2y), y)$, *pour tout* y.

Forme triangulaire et échellonnée

Définition 5 *Soit* A *une matrice de dimension* $m \times n$. *On dit que* A *est en forme :*

— *triangulaire (supérieure) si* $a_{i,j} = 0$ *pour tout* $j < i$.

— *échellonnée si le nombre de zéros précedant la première valeur non nulle d'une ligne augmente à chaque ligne.*

Remarque 11 *Dans une matrice échellonnée les lignes nulles sont forcement en bas de la matrice. Toute matrice échellonnée est triangulaire mais la reciproque est fausse. Par exemple :*

$$\begin{bmatrix} 0 & 1 \\ 0 & 2 \end{bmatrix}$$

Proposition 4 *Si A est une matrice triangulaire de dimension $n \times n$ et $a_{i,i} \neq 0$ pour $i = 1, \ldots, n$ alors le système $Ax = b$ a une solution unique qui peut être calculée de la façon suivante :*

$$\begin{array}{ll} x_n & = a_{n,n}^{-1}(b_n) \\ x_{n-1} & = a_{n-1,n-1}^{-1}(b_{n-1} - x_n a_{n-1,n}) \\ \ldots & \ldots \\ x_1 & = a_{1,1}^{-1}(b_1 - \Sigma_{j=2,\ldots,n}\ x_j a_{1,j}) \end{array}$$

Proposition 5 *Soit A une matrice échellonnée de dimension $m \times n$ dont les dernières k lignes sont nulles. Alors on a 3 possibilités :*

— *le système n'a pas de solution ssi il y a une ligne i nulle telle que $b_i \neq 0$,*
— *sinon, si $n = (m - k)$ alors la solution est unique et elle calculée comme dans la proposition précedente,*
— *sinon, $n > (m - k)$ et le système a plusieurs solutions (et on exprime $m - k$ variables comme des fonctions affines des autres $n - (m - k)$ variables).*

Transformations

Pour simplifier l'argument on va supposer que :
— on a autant de variables que d'équations ($m = n$),
— on arrête le calcul si la solution n'est pas unique.

Les deux transformations qu'on va utiliser sont :
— permuter deux équations,
— additionner à une équation un multiple d'une autre équation (on appelle ce type transformation *transvection* ou *combinaison linéaire*).

Clairement permuter deux équations dans un système n'affecte pas l'ensemble des solutions et cette propriété est vraie aussi pour la deuxième transformation. La preuve de ce fait est un simple exercice.

Proposition 6 *Pour tout $\alpha \in \mathbf{K}$, l'ensemble des solutions du système :*

$$\begin{array}{l} \Sigma_{j=1,\ldots,n}\ a_{1,j}x_j = b_1 \\ \Sigma_{j=1,\ldots,n}\ a_{2,j}x_j = b_2 \end{array}$$

est égal à l'ensemble des solutions du système :

$$\begin{array}{l} \Sigma_{j=1,\ldots,n}\ a_{1,j}x_j = b_1 \\ \Sigma_{j=1,\ldots,n}(\alpha a_{1,j} + a_{2,j})x_j = (\alpha b_1 + b_2) \end{array}$$

8.2 Élimination de Gauss

Le procédé qui transforme le système en forme triangulaire (ou échellonnée) est connu comme *élimination de Gauss*. Supposons que :

— $a_{i,j} = 0$ pour $j < i$ et $i = 1, \ldots, k-1$,

— $a_{i,i} \neq 0$ pour $i = 1, \ldots, k-1$.

Si $a_{i,k} = 0$ pour $i = k, \ldots, n$ on arrête le calcul (dans ce cas on peut montrer que la solution n'est pas unique).

Sinon, on permute la ligne k avec une ligne $i \in \{k, \ldots, n\}$ telle que $a_{i,k} \neq 0$. On appelle l'élément $a_{k,k}$ obtenu le *pivot*. Si on travaille avec les flottants alors une stratégie populaire pour limiter les erreurs d'arrondi consiste à choisir le pivot le plus grand en valeur absolue.

Ensuite on additionne à la ligne $i = k+1, \ldots, n$ la ligne k multipliée par $-(a_{k,k})^{-1}(a_{i,k})$. L'effet de ces transformations est de rendre 0 tous les coefficients sous $a_{k,k}$. En itérant ces calculs pour $k = 1, \ldots, n-1$, soit on arrête le calcul (solution pas unique) soit on obtient un système équivalent en forme triangulaire dont la solution est unique.

Remarque 12 *Il n'est pas difficile d'adapter la méthode d'élimination pour qu'elle transforme la matrice en forme échellonnée.*

Coût

Analysons le nombre d'opérations arithmétiques qu'il faut faire dans le pire des cas. Pour $k = 1, \ldots, n-1$ on doit :

— chercher un pivot en $O(n-k)$,

— éventuellement permuter en $O(n-k)$,

— additionner un multiple de la ligne k à la ligne i pour $i = k+1, \ldots, n$ en $O((n-k)^2)$.

On a donc :

$$\Sigma_{k=1,\ldots,n-1}\ (n-k)^2 = \Sigma_{i=1,\ldots,n-1}\ i^2\ ,$$

qui est $O(n^3)$. [1]

Programmation

La programmation de la méthode algorithmique qu'on vient de présenter est un excellent exercice. En effet, le programme accomplit une variété de tâches ; il faut :

— chercher un pivot,

— permuter deux lignes,

— effectuer une transvection,

— mettre le système en forme triangulaire,

— résoudre un système triangulaire.

On a donc à programmer, tester et combiner plusieurs fonctions. On considère une première solution minimaliste et on traitera un certain nombre de variations dans les TP. On remarque que les fonctions en question modifient la matrice qu'on leur passe en argument et qu'on ne se soucie pas de calculer la valeur d'expressions dont on sait à priori que le résultat est 0. Ainsi si on imprime la matrice a à la fin du calcul on constatera que les éléments sous la diagonale ne sont pas forcement 0.

1. On peut approximer avec un intégral ou alors se souvenir de la formule : $\Sigma_{i=1,\ldots,n}\ i^2 = (n(n+1)(2n+1))/6$.

```
def pivot(a,i):
    n=len(a)
    for k in range(i,n):
        if (a[k][i]!=0):
            return k
    return -1
def perm(a,i,j):
    n=len(a)
    for k in range(i,n+1):
        a[i][k],a[j][k]=a[j][k],a[i][k]
    return None
def trans(a,i):
    n=len(a)
    for k in range(i+1,n):
        m=-a[k][i]/a[i][i]
        for j in range(i+1,n+1):
            a[k][j]=a[k][j]+ (m* a[i][j])
    return None
def gauss(a):
    n=len(a)
    for i in range(n-1):
        j=pivot(a,i)
        assert(j!=-1)
        if (i!=j):
            perm(a,i,j)
        trans(a,i)
    return None
def solvetri(a):
    n=len(a)
    x=[0 for i in range(n)]
    for i in range(n-1,-1,-1):
        x[i]=a[i][n]
        for j in range(i+1,n):
            x[i]=x[i]-(a[i][j]*x[j])
        assert(a[i][i]!=0)
        x[i]=x[i]/a[i][i]
    return x
```

Chapitre 9

Interpolation et régression linéaire

On fixe n points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ dans $\mathbf{K}^2$ où, pour fixer les idées, $\mathbf{K}$ est le corps des nombres réels et les $x_0, \ldots, x_{n-1}$ sont tous différents. On se propose d'approcher les n points par un polynôme. On se concentre sur deux résultats fondamentaux :

— il existe un polynôme de degré au plus $n-1$ qui passe *exactement* par les n points,

— pour tout $m < (n-1)$, il existe un polynôme $p(x)$ de degré au plus m qui minimise l'*erreur* au sens des *moindres carrés*, à savoir la quantité :

$$\Sigma_{i=0,\ldots,(n-1)} \ (y_i - p(x_i))^2 \ .$$

Dans les deux cas, le polynôme en question peut être calculé de façon efficace par résolution d'un système d'équations linéaires.

9.1 Des points au polynôme

Définition 6 (matrice Vandermonde) *La matrice de Vandermonde V_n pour les points $x_0, \ldots, x_{n-1}$ est définie par :*

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \cdots & \cdots & \cdots & \cdots & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \tag{9.1}$$

Si les points $x_0, \ldots, x_{n-1}$ sont différents, on peut montrer que, pour tout vecteur y de dimension n, le système $V_n a = y$ a une *solution unique* (ce qui est équivalent à dire que la matrice V_n est *inversible* ou que son *déterminant* est différent de 0).

Proposition 7 *Soient (x_k, y_k) des points dans $\mathbf{R}^2$, pour $k = 0, \ldots, n-1$ et avec $x_i \neq x_j$ si $i \neq j$. Alors il existe unique un polynôme $p(x)$ de degré au plus $n-1$ tel que $p(x_k) = y_k$, pour $k = 0, \ldots, n-1$.*

Preuve. L'assertion que $p(x_k) = y_k$ pour $k = 0, \ldots, n-1$ est équivalente à la condition $V_n a = y$, où V_n est la matrice de Vandermonde relative aux points $x_0, \ldots, x_{n-1}$, $y = (y_0, \ldots, y_{n-1})$ et $a = (a_0, \ldots, a_{n-1})$ sont les coefficients du polynôme à déterminer. Comme V_n est inversible, la solution du système est unique et peut être calculée avec la méthode de Gauss. □

Exemple 17 *Si on prend comme points* $\{(-1,0),(1,0),(2,3)\}$*, on obtient le système :*

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 2 & 4 & 3 \end{array}\right]$$

qui se tranforme en la forme triangulaire :

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 3 \end{array}\right]$$

qui a comme solution $(a_0,a_1,a_2)=(-1,0,1)$*. Le polynôme interpolant est donc* $p(x)=x^2-1$*.*

Remarque 13 *La solution du système avec la méthode de Gauss prend* $O(n^3)$*. Ensuite le calcul du polynôme interpolant en un point prend* $O(n)$*.*

9.2 Interpolation de Lagrange

Il est possible de construire le polynôme interpolant de façon plus explicite. Cette méthode est due à Lagrange et elle préférable si le but est d'évaluer le polynôme en un certain nombre de points. [1]

Définition 7 (polynôme interpolant) *Soient* (x_k,y_k) *des points dans* $\mathbf{R}^2$ *pour* $k=0,\ldots,n-1$ *et avec* $x_i\neq x_j$ *si* $i\neq j$*. On définit le polynôme interpolant par :*

$$\ell(x)=\Sigma_{i=0,\ldots,n-1}\ y_i\ \frac{\Pi_{j\neq i}(x-x_j)}{\Pi_{j\neq i}(x_i-x_j)}\ .$$

La proposition suivante est une preuve *directe* (c'est-à-dire, sans passer par la matrice de Vandermonde) de l'existence d'un polynôme interpolant de degré au plus $(n-1)$.

Proposition 8 *Le polynôme* $\ell(x)$ *a degré au plus* $(n-1)$ *et il satisfait* $\ell(x_i)=y_i$ *pour* $i=0,\ldots,(n-1)$*.*

Preuve. Il est clair que le degré est au plus $n-1$ et on vérifie que :

$$\frac{\Pi_{j\neq i}(x-x_j)}{\Pi_{j\neq i}(x_i-x_j)}=\left\{\begin{array}{ll} 1 & \text{si } x=x_i \\ 0 & \text{si } x=x_k\neq x_i\ . \end{array}\right.$$

□

Exemple 18 *Si on prend comme points* $\{(-1,1),(1,1),(2,4)\}$*, on obtient le polynôme :*

$$\ell(x)=1\tfrac{(x-1)(x-2)}{(-1-1)(-1-2)}+1\tfrac{(x+1)(x-2)}{(1+1)(1-2)}+4\tfrac{(x+1)(x-1)}{(2+1)(2-1)},$$

et on peut vérifier que $\ell(-1)=1$*,* $\ell(1)=1$ *et* $\ell(2)=4$*.*

1. Les matrices de Vandermonde sont connues pour être une source d'instabilité numérique quand on travaille sur les flottants.

On programme une fonction qui prend n points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ et calcule en $O(n^2)$ le vecteur $(c_0, \ldots, c_{n-1})$ où :

$$c_i = \frac{y_i}{\Pi_{j \neq i}(x_i - x_j)}$$

Notez que ce vecteur ne dépend *pas* de x ! Le calcul pourrait être un peu optimisé (sans changer sa complexité asymptotique) en notant que pour $i < j$ on a $(x_i - x_j) = -(x_j - x_i)$

```
def lagrange_coef(x,y):
    n=len(x)
    assert(n==len(y))
    Delta = [1 for i in range(n)]
    for i in range(n):
        for j in range(n):
            if i!=j:
                Delta[i]=Delta[i]*(x[i]-x[j])
    c=[y[i] for i in range(n)]
    for i in range(n):
        c[i]=c[i]/Delta[i]
    return c
```

Ensuite on programme une fonction qui a partir du vecteur c évalue en $O(n)$ le polynôme interpolant de Lagrange en un point x.

$$\ell(x) = \Sigma_{i=0,\ldots,n-1}\ c_i(\Pi_{j \neq i}(x - x_j))$$

Soit :

$$\pi_i = \Pi_{j \neq i}(x - x_j)\ .$$

On remarque que si $i < (n-1)$ alors :

$$\pi_{i+1} = \pi_i \frac{(x - x_i)}{(x - x_{i+1})}$$

Ainsi on peut calculer π_{i+1} à partir de π_i en $O(1)$ et on peut calculer π_i pour $i = 0, \ldots, n-1$ en $O(n)$.

```
def interpol(c,x,y,v):
    n=len(c)
    assert(n==len(x) and n==len(y))
    for i in range(n):
        if (v==x[i]):
            return y[i]
    pi=[1 for i in range(n)]
    for j in range(1,n):
        pi[0]=pi[0]*(v-x[j])
    for i in range(1,n):
        pi[i]=(pi[i-1]*(v-x[i-1]))/(v-x[i])
    sum=0
    for i in range(n):
        sum=sum+c[i]*pi[i]
    return sum
```

9.3 Moindres carrés

On vient de voir qu'on peut toujours construire un polynôme de degré $n-1$ qui passe par n points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$. Dans bien de situations, on aimerait calculer un polynôme qui décrit la distribution des points d'une façon *simple*. En d'autres termes, on souhaiterait avoir un polynôme de degré au plus m avec $m << n$ et on est prêt à tolérer le fait que le polynôme ne passe pas exactement par tous les points, d'autant plus si ces points sont le résultat d'observations sujettes à erreur.

Une approche importante à ce problème consiste à définir l'*erreur* du polynôme p comme :

$$\Sigma_{i=0,\ldots,n-1}(p(x_i) - y_i)^2 \tag{9.2}$$

et à minimiser cette quantité en faisant varier p parmi les polynômes de degré au plus m. Cette définition de l'erreur a deux atouts : elle est *raisonnable* et elle a des *propriétés mathématiques* qui permettent de résoudre de façon efficace le problème d'optimisation. En effet, on peut montrer que ce problème a toujours une solution qui peut être calculée de façon efficace avec la méthode de Gauss. On parle souvent de *méthode des moindres carrés*. Pour montrer ce résultat en toute généralité on a besoin de notions d'algèbre linéaire qui ne sont pas encore acquises. De ce fait, on va se limiter à illustrer la méthode dans le cas où on cherche à trouver un polynôme de degré au plus 1 (une droite !) qui approche les points. Dans le domaine de la *statistique*, la droite en question s'appelle aussi *modèle de régression linéaire* ; on utilisera cette terminologie mais on fera complètement l'impasse sur l'interprétation statistique du modèle !

Soit donc $p(x) = ax + b$ la droite qu'on cherche à déterminer. Dans ce cas, l'erreur est une fonction réelle $f(a, b)$ en deux variables :

$$f(a, b) = \Sigma_{i=0,\ldots,n-1}(ax_i + b - y_i)^2 \ . \tag{9.3}$$

Pour que x soit un minimum local d'une fonction dérivable en une variable, il faut que la dérivée dans x soit nulle. Pour les fonctions à deux variables, il faut que les dérivées par rapport à la première et à la deuxième variable soient nulles. Bien entendu ces conditions sont *nécessaires* mais pas suffisantes pour avoir un minimum local ; pour avoir des conditions suffisantes il faudrait analyser aussi les dérivées secondes... mais on négligera cette partie de la preuve. Si on dérive par rapport à b et à a, on obtient les conditions :

$$\frac{\partial f(a,b)}{\partial b} = \Sigma_{i=0,\ldots,n-1}\ 2(ax_i + b - y_i) = 0 \ , \qquad \frac{\partial f(a,b)}{\partial a} = \Sigma_{i=0,\ldots,n-1}\ 2(ax_i + b - y_i)x_i = 0 \ .$$

Soient $\overline{x} = (1/n)(\Sigma_{i=0,\ldots,n-1}\ x_i)$ et $\overline{y} = (1/n)(\Sigma_{i=0,\ldots,n-1}\ y_i)$ les moyennes arithmétiques des points en abscisse et en ordonnée, respectivement. On dénote aussi par $\langle x, y\rangle$ le produit scalaire du vecteur x par le vecteur y. Avec cette notation, on dérive de la première équation :

$$b = \overline{y} - a \cdot \overline{x} \ .$$

En remplaçant b dans la deuxième équation, on a :

$$a(\Sigma_{i=0,\ldots,n-1}(x_i)^2 - \overline{x}(\Sigma_{i=0,\ldots,n}\ x_i)) = \Sigma_{i=0,\ldots,n}\ x_i y_i - \overline{y}(\Sigma_{i=0,\ldots,n}\ x_i) \ .$$

En utilisant la notation vectorielle, on réécrit l'équation de façon plus lisible comme :

$$a \cdot (\langle x, x\rangle - n \cdot \overline{x}^2) = \langle x, y\rangle - n \cdot \overline{x} \cdot \overline{y} \ ,$$

soit :

$$a = \frac{\langle x, y\rangle - n \cdot \overline{x} \cdot \overline{y}}{\langle x, x\rangle - n \cdot \overline{x}^2}$$

et on peut montrer que le dénominateur est positif sauf si toutes les composantes du vecteur x sont égales (ce qui est faux dans le problème en question).

Exemple 19 *Si on prend à nouveau les points* $\{(-1,0),(1,0),(2,3)\}$ *de l'exemple 17, on a :*

$$\overline{x} = \tfrac{2}{3}\ , \quad \overline{y} = 1\ , \quad b = 1 - a \cdot \tfrac{2}{3}\ , \quad a = \tfrac{6}{7}\ ,$$

et donc $b = 3/7$ *et la droite qui approche le mieux les points au sens des moindres carrés est* $y = (6x + 3)/7$.

Chapitre 10

Séries et tables de données (types Series et DataFrame)

On présente deux types de données disponibles dans le module pandas :

— le type des séries (type Series) qui est un tableau à une dimension où on peut associer des noms aux composantes,

— le type des tables de données (type DataFrame) qui est une généralisation en deux dimensions du type des séries dans lequel on peut nommer les lignes et les colonnes.

Comme pour les autres modules évoqués dans le cours, on se limitera à présenter un nombre très limité de fonctionnalités. Par exemple, la documentation [1] mentionne plus que 200 méthodes ou attributs qui relèvent d'une valeur de type DataFrame. Au besoin, il faudra naviguer dans la très volumineuse documentation du module.

Ces types de données sont des modestes variations sur le type des tableaux (type list) et il ne serait pas très compliqué de simuler les opérations qu'on va présenter en utilisant exclusivement le noyau de python. En effet, leur intérêt est éminemment *pratique* ; à savoir, elles proposent des fonctionnalités qu'un utilisateur lambda aurait du mal à programmer correctement. Par exemple, on verra dans les travaux pratiques qu'on peut convertir assez aisément un fichier texte généré par une feuille de calcul (format csv) en une valeur de type DataFrame. [2]

Dans les exemples qui suivent, on suppose avoir importé le module pandas avec la commande :

```
import pandas as pd
```

10.1 Type Series

Pour créer une série on peut passer au constructeur Series un tableau de *valeurs* et (en option) un tableau de *noms* de la même longueur. Par exemple :

```
>>> s=pd.Series([10,True,'philippe',40.5],index=['a','b','c','a'])
>>> s
>>> s
```

1. https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

2. Le module pandas peut aussi traiter, par exemple, des données issues de *pages sur la toile* (langage HTML) ou d'une *base de données* (langage SQL).

```
a          10
b        True
c    philippe
a        40.5
dtype: object
```

On remarquera que les types des valeurs peuvent être différentes et qu'il peut y avoir des répétitions dans les noms.

Il y a deux façons d'accéder les valeurs d'une série : en précisant leur *position* comme on le ferait pour un tableau ou une tuple ou en précisant leur *nom*. Comme les noms ne sont pas uniques, un accès par nom peut retourner une sous-série. Par exemple :

```
>>>s[0]
10
>>> s['a']
a      10
a    40.5
dtype: object
```

Le type des séries est *mutable*. On peut donc affecter une nouvelle valeur à une composante d'une série. Par exemple :

```
>>> s[0]=3
>>> s['a']=4
```

Encore une fois, comme le nom n'est pas unique, la deuxième affectation va modifier toutes les composantes associées au nom 'a'. Il est possible de *filtrer* les éléments d'une série s en utilisant une notation de la forme s[b]. où b est une condition logique sur les valeurs de la série. Pour indiquer une valeur de la série dans la condition logique, on utilise le nom de la série. Ainsi, si on écrit :

```
s1=s[(s<30) or (s>50)]
```

on affecte à s1 la sous-série de s dont les valeurs sont ou bien inférieures à 30 ou bien supérieures à 50. Attention, ceci produit une erreur si la série s contient des valeurs qui ne sont pas comparables avec un entier.

On note au passage que si t est un tableau on peut obtenir le même effet en écrivant :

```
t1=[]
for x in t:
    if (x<30) or (x>50):
        t1.append(x)
```

La méthode describe permet d'obtenir des statistiques sur la série. Par exemple, pour des valeurs numériques on obtient des quantités comme la moyenne arithmétique et l'écart type (ou déviation standard).

10.2 Type DataFrame

Le type DataFrame généralise le type des Series en deux dimensions. Pour créer une valeur de type DataFrame on peut fournir un tableau de tableaux (data) et en option on peut attribuer des noms aux colonnes (columns) et aux lignes (index). Par exemple :

```
>>> df= pd.DataFrame(columns=['tp','partiel','examen','total'],
                     index=['anne','marius','marie'],
                     data=[[10,13,12,11],
                           [13,10,12,11],
                           [11,12,10,13]])
>>> df
        tp  partiel  examen  total
anne    10       13      12     11
marius  13       10      12     11
marie   11       12      10     13
```

Si on omet les noms des colonnes (des lignes), le système les remplace par les positions $(0, 1, \ldots)$. Si df est une valeur de type DataFrame alors df.shape() retourne le tuple composé du nombre de lignes et de colonnes. Par ailleurs, df.columns (df.index) retourne les noms des colonnes (des lignes) et ces noms peuvent être modifiés. Par exemple, la commande :

```
df=df.rename(index={'anne':'annie'})
```

va renommer la ligne anne en annie. Comme pour le type Series, on dispose de deux notations pour accéder les valeurs : par nom (méthode loc) et par position (méthode iloc). Dans notre exemple, df.loc['anne','tp'] est équivalent à df.iloc[0,0] et il est aussi équivalent à df.iloc[0]['tp'] et à df.loc['anne'][0].

Il est aussi possible de sélectionner plusieurs lignes et/ou colonnes en même temps. En continuant notre exemple, df.loc[['anne','marius'],['tp','partiel']] est équivalent à df.iloc[[0,1],[0,1]] et permet de sélectionner les premières 2 lignes et les premières 2 colonnes.

Le type DataFrame étant aussi mutable, on peut utiliser les notations ci-dessus pour modifier les valeurs. Par ailleurs, comme pour les séries on dispose aussi de notations pour filtrer les valeurs d'après un certain prédicat. Par exemple, en continuant l'exemple précédent :

```
df[df.tp>10]
```

est une nouvelle valeur de type DataFrame obtenue de df en supprimant toutes les lignes dans lesquelles la note de tp n'est pas supérieure à 10 ; en l'occurrence on supprime la ligne de anne.

Dans la documentation du type DataFrame on trouvera aussi les fonctions qui permettent d'imiter certaines manipulations typiques d'une feuille de calcul comme, par exemple, ajouter ou supprimer une ligne, ajouter ou supprimer une colonne, fusionner deux tableaux. Notez au passage que l'opération de *fusion* a plusieurs sémantiques possibles et que donc on trouve dans la documentation plusieurs fonctions dédiées à cette tâche.

Bibliographie

[CLRS09] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 2009. Troisième édition. Document disponible en ligne, il existe aussi en français.

[Dem06] Jean-Pierre Demailly. *Analyse numérique et équations différentielles*. Grenoble Sciences, 2006. Document disponible à la bibliothèque MIR UPD.

[FP19] Patrick Fuchs and Pierre Poulain. Introduction à la programmation Python pour les sciences du vivant. Technical report, Université Paris-Diderot, 2019. Document disponible en ligne.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991. Document disponible en ligne.

[Her19] Raphaèle Herbin. Cours d'analyse numérique. Technical report, Université Aix-Marseille, 2019. Document disponible en ligne.

[McK17] Wes McKinney. *Python for data analysis*. O'Reilly, 2017. Document disponible en ligne.

[MES13] MESR. Programme informatique, voies scientifiques. Technical report, Ministère de l'enseignement supérieur et de la recherche, 2013. Document disponible en ligne.

[Sva19] Jules Svartz. Cours d'informatique pour tous. Technical report, Lycée Massena, Nice, 2019. Notes de cours disponibles en ligne.

[VGVdB19] Gaël Varoquaux, Emmanuelle Gouillart, Olaf Vahtras, and Pierre de Buyl. Scipy lecture notes. Technical report, 2019. Nombreuses contributions, document disponible en ligne.

[vR19] Guido van Rossum. Python tutorial. Technical report, Python software foundation, 2019. Document disponible en ligne.

[Wac13] Benjamin Wack. *Informatique pour tous en classes préparatoires aux grandes écoles*. Eyrolles, 2013. Nombreuses contributions.

Index

TP1

La page Moodle du cours

Le calendrier, les modalités de contrôle des connaissances, la bibliographie, la trace du cours, les planches de TP, le rendu des TP, les ressources en ligne, les corrigés des examens, les notes proposées au jury,... sont sur la page MOODLE du cours . Il faut s'y inscrire si ce n'est pas déjà fait. Pour accéder à la page il faut activer votre compte ENT.

Environnement de travail

Les machines du SCRIPT disposent d'un environnement de travail nommé Spyder qui permet d'éditer des programmes python et de les exécuter. Pour démarrer Spyder :

1. cliquez sur Menu en haut à gauche de la fenêtre,
2. cliquez sur Développement,
3. cliquez sur Spyder3,
4. si un message de mise-à-jour apparaît, cliquez sur OK.

Lorsque vous avez ouvert Spyder, vous voyez 3 différentes parties dans la fenêtre :
— la partie de gauche pour éditer le fichier temp.py,
— la partie en haut à droite pour obtenir de l'aide dans Spyder,
— La partie du bas à droite, appelée *console* IPython.
Vous pouvez :
— écrire des commandes dans la console comme vous le feriez avec une calculatrice ; les commandes sont immédiatement exécutées et le résultat est affiché sur la console,
— écrire un programme dans un fichier, le sauver et ensuite l'exécuter. Si votre programme contient des instructions d'entrée sortie (input et print) alors vous pourrez interagir avec le programme via la console (sinon, vous ne verrez rien).

Utilisation de python en ligne de commande

On peut utiliser python comme une calculette : on tape à la console une expression suivie par un retour à la ligne et on regarde le résultat. Par exemple :

TP rédigé en collaboration avec Simon MASSON.
https://moodlesupd.script.univ-paris-diderot.fr/course/view.php?id=12061

```
(3-5)*2
```

En plus des nombres entiers, vous avez les nombres flottants (3.14, -5e-10,...), les valeurs booléennes (True, False), les chaînes de caractères 'Bonjour',... Par ailleurs, vous disposez d'opérateurs arithmétiques $(+, -, *, /, //, \%, \ldots)$ et de prédicats $(<, ==, <=, \ldots)$.

Exercice 1 *1. Quelle est la différence entre la division \ et la division \\ ?*

2. Quelle est la règle utilisée par python *pour calculer la division et le reste ? Testez votre hypothèse sur les nombres négatifs.*

Pour utiliser certaines fonction comme la racine carrée (sqrt), il faut d'abord importer le module math avec la commande import math, à défaut la fonction n'est pas reconnue.

Dès que l'expression devient un peu compliquée, on utilise des variables pour stocker des valeurs intermédiaires. Par exemple :

```
x=(3+5)*2
x*(x-1)
```

Exercice 2 *1. Il ne faut pas confondre les variables avec les mots clefs du langage. Par exemple, que se passe-t-il si on écrit* def=34 *?*

2. Il ne faut pas non plus confondre la commande d'affectation avec le prédicat d'égalité. Quelle est la différence entre : x=3, 3=x, x==3, 3==x *?*

Entrée-sortie standard

Les commandes print et input permettent d'échanger de l'information entre le programme et l'utilisateur. Pour l'instant, on suppose que cet échange d'information se passe sur l'écran (la console), plus tard on verra qu'on peut aussi utiliser des fichiers.

La commande print($\mathsf{e}_1, \ldots, \mathsf{e}_n$) permet d'afficher à l'écran les valeurs des expressions $\mathsf{e}_1, \ldots, \mathsf{e}_n$ en les séparant par un espace et en les faisant suivre par une nouvelle ligne. Par exemple, dans le cas où $n = 2$, $\mathsf{e}_1 = 2 + 3$ et $\mathsf{e}_2 =$ 'Bonjour' + 'Marius', la commande imprimera :

```
5 Bonjour Marius
```

Notez que :

— on utilise les guillemets pour délimiter une chaîne de caractères,

— le symbole + dénote l'addition si on manipule les valeurs numériques et la concaténation si on traite les chaînes de caractères.

La commande input() retourne une chaîne de caractères produite par l'utilisateur qui tape au clavier une suite de caractères suivis par un retour à la ligne. C'est une bonne pratique d'alerter l'utilisateur du fait que le programme attend une entrée. Pour ce faire, on imprime à l'écran un petit message et ensuite on exécute la commande input. Cette pratique est tellement diffuse qu'en python le message peut être intégré directement à la commande input. Par exemple, la commande input('Entrez un mot') est équivalente à print('Entrez un mot') suivie par input().

Édition et interprétation du programme

Dès que le programme fait plus qu'une ligne, il convient de l'écrire dans un fichier plutôt qu'à la console et ensuite de l'exécuter. Vous serez alors confronté à différents types d'erreurs. Erreurs syntaxiques qui sont détectées avant l'exécution du programme : mauvaise utilisation des parenthèses, erreurs de tabulation,...

Exercice 3 *Corrigez le programme suivant :*

```
x=((3+5)*7
   y=x*x
print(y)
```

Erreurs qui sont détectées pendant l'exécution : variable/fonction non déclarée, opérateur incompatible avec les types des arguments, division par zéro,...

Exercice 4 *Corrigez le programme suivant :*

```
x=input('Entrez un entier ')
y=x*x
print(x*x)
print(x/z)
```

Erreurs sémantiques : le programme tourne mais ne produit pas le résultat attendu. Le programme est-il une mise-en-oeuvre fidèle de l'algorithme et l'algorithme est-il correct ?

Exercice 5 *Corrigez le programme suivant qui est censé calculer la moyenne arithmétique des composantes d'un vecteur :*

```
def moyenne(t):
    s=1
    for x in t:
        s=s+x
        return s/len(t)
print(moyenne((2,4,6)))
```

Erreurs de conception : le programme tourne mais est très inefficace.

Exercice 6 *Considérez la fonction suivante qui calcule le pgcd de deux nombres entiers* positifs.

```
def pgcd(a,b):
    while(a!=b):
        if (a>b):
            a=a-b
        else:
            b=b-a
    return a
```

1. *Que se passe-t-il si on appelle* pgcd(1000000000,3) *?*
2. *Ce programme calcule-t-il le pgcd sans l'hypothèse* $a, b > 0$ *?*

Commentaires

python ignore tout ce qui est écrit entre le symbole ♯ et la fin de la ligne. Par exemple, la commande :

```
print('Hello world!') # Dis bonjour !
```

va juste afficher 'Hello world !'. Quand le programme compte beaucoup de fonctions, il est conseillé d'associer un commentaire à chaque fonction qui décrit brièvement son comportement. Pour ce faire on a une notation dédiée, à savoir si on écrit :

```
def produit_scalaire(x,y):
    'calcule le produit scalaire'
    n=len(x)
    assert(n==len(y))
    s=0
    for i in range(n):
        s=s+x[i]*y[i]
    return s
```

la chaînes de caractères 'calcule...' est traitée comme un commentaire et *en plus* elle est associée au nom de la fonction. Ainsi si après la définition on écrit

```
help(produit_scalaire)
```

on obtient comme réponse le commentaire en question (et on sort de l'interaction avec le help en tapant q pour quit). Si on a besoin de plusieurs lignes pour commenter une fonction on entoure le commentaire par des triples guillemets comme dans :

```
    '''calcule
       le produit
       scalaire'''
```

Visualiser l'exécution d'un programme

On peut visualiser l'exécution d'un programme à l'aide du site `http://www.pythontutor.com/visualize.html#mode=edit`. Par exemple, collez le code de la fonction produit_scalaire ci-dessus suivi par la commande print(produit_scalaire((1, 4), (−4, 5))).

Exercice 7 *Le programme suivant est constitué de* 3 *fonctions, dont les fonctions* f *et* g *qui s'appellent mutuellement. Que fait ce programme ?*

```
def f(x):
  if (x>1):
      return g(2*x)
  else:
      return 0
def g(x):
  if (x>1):
      return f(x//3)
  else:
      return 1
def main():
   x=int(input('Entrez nombre'))
   print('La sortie est ', f(x))
main()
```

Alternative à Spyder

Une alternative à l'utilisation de Spyder, consiste à ouvrir une fenêtre terminal en tapant [CTRL]+[ALT]+t. Quelques commandes disponibles dans le terminal :

- pwd pour savoir où vous êtes ; par défaut vous êtes dans le répertoire 'maison',
- ls pour visualiser les fichiers dans le répertoire où vous êtes,
- mkdir TP1 pour créer un nouveau répertoire qui s'appelle TP1,
- chmod 700 TP1 pour empêcher la lecture du repértoire par d'autres utilisateurs,
- cd TP1 pour vous déplacer dans le répertoire TP1 (s'il existe) ; cd sans nom vous ramène au répertoire 'maison'.

Exercice 8 *Ouvrez une fenêtre, visualisez où vous êtes (c'est votre répertoire maison), créez un répertoire* Informatique*, déplacez-vous dans ce répertoire, créez un répertoire* TP1*, déplacez-vous dans ce répertoire, revenez au répertoire maison.*

Placez-vous dans le répertoire TP1 :

- si vous tapez python3, vous avez l'équivalent de la console dont vous sortez avec quit(),
- si vous éditez un fichier avec un éditeur de texte et vous le sauvez, par exemple, comme temp.py, vous pouvez ensuite l'exécuter en tapant dans la fenêtre terminal python3 temp.py.

Il est *indispensable* d'utiliser un éditeur de texte qui connaît la syntaxe de python et vous aide à écrire les programmes correctement ; par exemple, emacs fera l'affaire. Tapez emacs nom.py & pour lancer l'éditeur (notez que les noms de vos fichiers python doivent terminer avec .py), écrivez un programme, sauvez-le avec [CTRL]+[x]+[s], quittez l'éditeur avec [CTRL]+[x]+[c]. Vous pouvez utiliser emacs aussi pour créer des simples fichiers 'texte' ; par exemple pour préparer des fichiers qui seront lus par votre programme.

Wifi à l'Université

Il y a deux réseaux wifi à l'université et vous aurez besoin de votre compte ENT (espace numérique de travail) pour vous y connecter :

- UP7D, à noter que sous serez disconnecté après un certain temps,
- EDUROAM, voir `https://support.wiki.univ-paris-diderot.fr/wifi:eduroam` pour les modalités de configuration de votre machine.

Conseils pour l'installation d'un environnement de programmation python sur votre machine

Pour installer un environnement de programmation sur votre machine lire, par exemple, le chapitre 1, section 2 de `https://python.sdv.univ-paris-diderot.fr/`. A noter qu'en plus d'une distribution standard de python3, on va utiliser les modules suivants qu'il faudra donc installer : numpy pour avoir des tableaux efficaces, scipy pour faire un peu de calcul numérique, matplotlib pour visualiser des fonctions et pandas pour faire un petit peu de manipulation de données.

Exécution à distance

En dernier ressort, vous pouvez utiliser un navigateur (Firefox,...) pour vous connecter à un site qui exécutera le programme pour vous. Par exemple, si l'UFR de Physique a activé votre compte, vous pouvez vous connecter à `https://jupy.physique.univ-paris-diderot.fr:8000/hub/login`.

TP2

Exercice 9 (entiers) *1. Calculez la représentation en base* 2 *de* 324557.

2. *En* python*, des chiffres* 0 *ou* 1 *précédés par* 0b *sont interprétés comme un nombre en base* 2*. Par exemple,* print(0b1101) *renvoie* 13*. Utilisez cette propriété pour vérifier votre calcul.*
3. *Utilisez* input *pour demander un entier à trois chiffres.*
4. *Trouvez comment afficher à l'écran les chiffres contenues dans l'entier. Par exemple, si l'entier est* 123 *on affichera à l'écran* L'entier s'écrit avec les chiffres 1, 2 et 3.
5. *Comment savoir si l'entier est un multiple de* 3 *?*

Exercice 10 (flottants) *1. Les flottants sont représentés en fraction de nombres binaires. Ainsi,* 0.125 *est stocké comme '0.001' :* $\frac{0}{2}+\frac{0}{4}+\frac{1}{8}$*. La représentation en fraction binaire du nombre (décimal)* 0.1 *est infini :* 0.000110011001100 . . . *. Ainsi le nombre* 0.1 *est stocké en prenant une approximation avec* 53 *bits. Additionnez* 0.1 10 *fois. Quel est le résultat ?*

2. *On considère le programme suivant. Quel devrait être le résultat si on disposait d'une arithmétique parfaite sur les réels ? Quel est le comportement observé ?*

```
def id(x) :
    return (1 - (1/(1+x)))*(1+x)
x=1
for i in range(20) :
    print(id(x))
    x=x/10
```

Exercice 11 (booléens) *1. Exprimez les conditions logiques suivantes à l'aide des opérateurs logiques* and, or *et* not *:*

— *si* (t < 0) *alors* (state ==′ solide′).
— *Soit* (state ==′ solide′) *soit* (state ==′ liquide′) *soit* (t > 100) *(il s'agit bien d'un* ou exclusif*).*

2. *Trouvez des valeurs numériques des variables* x *et* y *tels que la condition logique* (x == 0) or (y/x < 3) *n'est pas équivalente en* python *à la condition logique* (y/x < 3) or (x == 0)*. Testez.*

TP rédigé en collaboration avec Simon Masson.

3. *On dit que deux expressions booléennes sont équivalentes quand ces deux expressions s'évaluent toujours vers la même valeur booléenne pour toutes les valeurs possibles des variables qui apparaissent dans ces expressions. Par exemple, si* x *et* y *contiennent des entiers alors* $x > 10 and x < 12$ *est équivalente à* $x == 11$. *alors que* $x > y$ *et* $x! = y$ *ne sont pas équivalentes. En supposant que* x, y, z *varient sur les entiers, dire si les expressions suivantes sont équivalentes :*
   — $x > y$ or $x < y$ *et* $x! = y$,
   — $x! = 3$ and $x! = 4$ and $x! = 5$ *et* $x <= 2$ or $x >= 6$,
   — $x == y$ and $x == z$ *et* $x == z$,
   — $x == y$ and $x == z$ *et* $x == y$ and $y == z$.
   Vérifiez que les expressions qui ne sont pas équivalentes peuvent retourner des booléens différents.
4. *Traduire les expressions suivantes en langage* python *:*
   — *L'entier* n *est divisible par* 5.
   — *Les entiers* m *et* n *sont tels que l'un est multiple de l'autre.*
   — *Les trois entiers* m, n *et* p *sont de même signe.*
   — n *est le plus petit multiple de* 7 *supérieur à* 10^{100}.
   — *Les trois entiers* m, n *et* p *sont distincts deux à deux.*

Exercice 12 (branchement conditionnel) *En* python*, on peut effectuer des branchements conditionnels. Pour cela, on utilise la syntaxe suivante :*

```
if (condition_1):
    commande_1
elif (condition_2):
    commande_2:
...
elif (condition_n):
    commande_n
else:
    commande_n+1
```

où les branches elif *et/ou* else *peuvent être omises. Par exemple, on peut écrire :*

```
if x > 0 :
    print('x est positif')
else :
    print('x est negatif')
```

Écrivez un programme qui demande la saisie de trois notes sur 20 et qui affiche la mention obtenue.

Exercice 13 *Écrivez un programme* python *qui demande l'âge de l'utilisateur et lui indique s'il a droit au tarif réduit (moins de* 26 *ans ou plus de* 60 *ans).*

Exercice 14

1. *Les années bissextiles sont les années dont le millésime est divisible par* 4 *mais non divisible par* 100*, à l'exception des millésimes divisibles par* 400 *qui sont des années bissextiles. Donnez une expression booléenne qui exprime qu'une année est bissextile.*
2. *Écrivez un programme qui lit une année de l'entrée standard avec* input *et déclare si l'année est bissextile (ou pas).*

3. *Écrivez un programme qui lit* 3 *entiers, les interprète comme jour, mois et année et décide si la date en question est possible (par exemple, on ne peut pas avoir* jour==31 *si* mois==4*).*
4. *Écrivez un programme qui lit une date comme dans le question précédente et affiche la date qui suit de* 30 *jours la date reçue en entrée. Par exemple, si la date en entrée est le* 01/01/1985 *la date en sortie sera* 31/01/1985.

TP3

Représentation des nombres

Exercice 15 (complément à 2) *Au niveau de la machine, pour représenter des nombres entiers sur un nombre fixé n de bits (typiquement $n = 32$ ou $n = 64$) on utilise souvent une notation dite* en complément à 2. *Un avantage de cette représentation par rapport à celle avec signe et valeur absolue est qu'on peut utiliser le* même algorithme *(circuit) pour effectuer les opérations d'addition et soustraction. Pour trouver le nombre entier représenté par la suite binaire $d_{n-1} \cdots d_0$ on utilise la formule suivante :*

$$\Sigma_{i=0,\ldots,n-2}\ d_i \cdot 2^i - d_{n-1} \cdot 2^{n-1}\ .$$

1. *Si $n = 3$ quels sont les nombres entiers qu'on peut représenter ?*
2. *Proposez un algorithme pour trouver la représentation de $m+1$ à partir de la représentation de m (bien sûr l'algorithme va échouer si m est le plus grand entier représentable).*
3. *Si $d \in \{0, 1\}$ alors soit $\overline{d}$ son complémentaire : $\overline{0} = 1$ et $\overline{1} = 0$. Montrez que :*

$$\Sigma_{i=0,\ldots,n-2}\ d_i \cdot 2^i + \Sigma_{i=0,\ldots,n-2}\overline{d}_i \cdot 2^i = 2^{n-1} - 1\ .$$

4. *Que faut-il faire pour trouver la représentation de l'opposé d'un nombre représenté en complément à 2 ? Cette opération est-elle toujours possible ?*
5. *On suppose additionner deux représentations en complément à 2 avec l'algorithme avec retenue appris en école primaire. Trouvez des conditions sur les deux retenues les plus à gauche qui assurent que le résultat obtenu représente bien la somme.*

Exercice 16 (mantisse et exposant) *On représente des nombres positifs en virgule* flottante *en base 2 en supposant disposer de 4 bits pour la mantisse . et de 3 bits pour l'exposant qui est représenté en complément à 2 (voir exercice 15).*

1. *Combien de nombres positifs peut-on représenter ?*
2. *Quel est le nombre réel positif plus petit (f^-) et plus grand (f^+) qu'on peut représenter ?*
3. *Quelle est la distance minimale et maximale entre deux nombres consécutifs représentables ?*
4. *Soient $x \in [f^-, f^+]$ et $\widetilde{x}$ un nombre flottant aussi proche de x que possible. Peut-on donner une borne supérieure à l'erreur relative $|(\widetilde{x} - x)/x|$?*
5. *Supposons maintenant qu'on utilise une notation en virgule* fixe *avec 3 chiffres binaires à gauche de la virgule et 4 chiffres binaires à droite de la virgule. Les réponses aux questions ci-dessus changent-elles ?*

Rappel : en base 2 le chiffre à gauche de la virgule est toujours 1 et donc les 4 bits servent à représenter 4 chiffres à droite de la virgule

Cette représentation de l'exposant n'est pas celle adoptée dans la norme IEEE-754.

Tuples et chaînes de caractères

Exercice 17 *1. Quel est le comportement des prédicats ==, < et <= sur les tuples et sur les chaînes de caractères ?*

2. python *permet de 'multiplier' un entier par une tuple ou une chaîne de caractères. Par exemple, si* t *est une tuple on peut écrire* $3 * \mathsf{t}$*. Quel est le résultat ? Quid si on écrit* $0 * \mathsf{t}$*,* $(-3) * \mathsf{t}$ *ou* $(1.5) * \mathsf{t}$ *?*

3. Si t *est une tuple (ou une chaîne de caractères) alors* t.count(v) *retourne le nombre d'occurrences de* v *dans* t *et* t.index(v) *retourne la position de la première occurrence de* v *dans* t*. Testez ces méthodes. Le résultat est-il toujours défini ?*

Exercice 18 (tranches) *Si* t *est une tuple (ou une chaîne) alors* t[start : stop : step] *retourne la sous-tuple* $(\mathsf{t}[\mathsf{start}], \mathsf{t}[\mathsf{start} + \mathsf{step}], \ldots, \mathsf{t}[\mathsf{start} + \mathsf{k} * \mathsf{step}])$ *où l'on suppose que* step *est positif et que* k *est le plus grand nombre naturel tel que* $\mathsf{k} * \mathsf{step} < \mathsf{stop}$*. On peut aussi écrire* t[start : stop] *et dans ce cas il est entendu que* step *est* 1*. Il s'agit donc de découper la tuple et d'en prendre des tranches (*slices*, en anglais).*

1. *Testez l'opérateur de découpage.*
2. *Il est possible d'omettre un ou plusieurs indices. Par exemple, si* t *est la chaîne* 'abcd' *quelle est la valeur de* t[1 : 3] *et de* t[:: 2] *?*
3. *Il est aussi possible d'utiliser un* step négatif*. Par exemple, testez :* t[:: −1]*,* t[len(t) : 0 : −1] *et* t[len(t) :: −1]*. Notez au passage qu'on a une notation pour décrire une tuple* inversée *(la première composante devient la dernière, la deuxième l'avant dernière,...).*
4. *Le résultat de* t[len(t) : −1 : −1] *peu sembler bizarre ! Testez. En effet il y a un télescopage avec la notation utilisée pour compter depuis la fin de la tuple (*t[−1] *est le dernier élément d'une tuple).*

Conversions

Exercice 19 *Si on se restreint aux* 5 *types* int, bool, float, tuple *et* str*, on a* 20 *fonctions de conversion possibles d'un type à l'autre.*

1. *Pour chaque possibilité, énoncez la règle qui régit la conversion (notez que certaines conversions sont interdites et d'autres marchent sous certaines conditions).*
2. *Soient* T1 *et* T2 *deux types,* c_{12} *la fonction de conversion de* T1 *à* T2 *et* c_{21} *la fonction de conversion de* T2 *à* T1*. Soit* v *une valeur de type* T1 *pour laquelle* $c_{12}(v)$ *et* $c_{21}(c_{12}(v))$ *sont définis. Est-ce toujours vrai que* $c_{21}(c_{12}(v)) = v$ *?*

Branchement

Exercice 20 (racines) *Voici un programme qui recherche des racines d'un polynôme de degré au plus* 2 *similaire au programme vu en cours.*

```
import math
a=float(input('Coeff a?'))
b=float(input('Coeff b?'))
c=float(input('Coeff c?'))
```

```
delta = b*b-(4*a*c)
if (a==0 and b==0) :
    if (c==0):                                  #degré 0
        print('Tout nombre est une solution')
    else:
        print('Pas de solution')
elif (a==0):
    print('L'unique solution est :',-c/b)       #degré 1
elif (delta==0):                                #degré 2
    print('L'unique solution est :',-b/(2*a))
elif (delta<0):
    print('Pas de solution')
else:
    root = math.sqrt(delta)
    sol1=(-b+root)/(2*a)
    sol2=(-b-root)/(2*a)
    print('Les deux solutions sont :',sol1,sol2)
```

Dans ce programme on ignore le fait que les nombres réels sont approximés par des flottants. Considérez les polynômes suivants : (i) $x^2 - x - 1$*, (ii)* $x^2 - 0,2x + 0,01$ *et (iii)* $x^2 - (10^{-180})x$*. Dans quels cas le programme prédit correctement le nombre de racines ?*

Exercice 21 *On souhaite concevoir un programme qui reçoit en entrée le nombre de billets de* 50*,* 20 *et* 10 *euros dont on dispose ainsi qu'une somme* s *à payer. Si possible, le programme doit imprimer une façon de payer (exactement) la somme* s *avec les billets dont on dispose. Sinon, le programme imprime un message qui dit que le payement de la somme n'est pas possible. Pour simplifier le problème, on va supposer qu'on dispose d'au moins un billet de* 10 *euros. Dans ce cas, la stratégie* gloutonne *suivante permet de résoudre le problème : on paye autant que possible, c'est-à-dire sans dépasser la somme* s*, avec des billets de* 50*, ensuite avec des billets de* 20 *et enfin avec des billets de* 10*. Le lecteur est invité à vérifier que sans l'hypothèse sur les billets de* 10 *euros, cette stratégie ne permet pas toujours de trouver une solution.*

Boucles

Exercice 22 (recherche de motif) *Un* texte *est une chaîne de* $n \geq 1$ *caractères représentés par une valeur de type* str*. Un* motif *est aussi un suite de* $m \leq n$ *caractères représentés par une valeur de type* str*. Une* occurrence *d'un motif dans un texte est un nombre naturel qui indique une position dans le texte à partir de laquelle les caractères du texte coïncident avec ceux du motif. Par exemple, les occurrences du motif* bra *dans le texte* abracadabra *sont exactement* 1 *et* 8*. On notera que ici* $m = 3$*,* $n = 11$ *et qu'on compte les positions de gauche à droite à partir de* 0*. Les occurrences d'un motif peuvent se 'superposer'. Par exemple, les occurrences du motif* bb *dans le texte* abbba *sont* 1 *et* 2*. Écrire un programme qui lit de l'entrée standard un* motif *et un* texte *et imprime sur la sortie standard toutes les occurrences du motif dans le texte.*

Exercice 23 (suite de Syracuse) *Considérez la séquence suivante sur les entiers :*

$$s_{n+1} = \begin{cases} 1 & \textit{si } s_n \leq 1 \\ 3 \cdot s_n + 1 & \textit{sinon et } s_n \textit{ impair} \\ s_n/2 & \textit{autrement.} \end{cases} \tag{1}$$

Le temps de vol *d'un entier m est le plus petit n tel que $s_0 = m$ et $s_n = 1$. Les tests suggèrent que pour tout entier le temps de vol est* fini *mais personne sait le prouver ! On parle donc de* conjecture de Syracuse. *En supposant la conjecture vraie, votre tâche est d'écrire un programme qui lit un entier m et calcule l'entier qui a un temps de vol* maximal *parmi les entiers p tels que $p \leq m$.*

Exercice 24 (suite de Kaprekar) *Écrire un programme qui :*

1. *lit à l'écran un nombre naturel n de 4 chiffres (décimaux) qui ne sont pas tous égaux et un nombre positif m.*
2. *itère m fois les pas suivants :*
   - *calcule le nombre n^- constitué des 4 chiffres de n ordonnés de façon croissante de gauche à droite.*
   - *calcule le nombre n^+ constitué des 4 chiffres de n ordonnés de façon décroissante de gauche à droite.*
   - *calcule le nombre $n' = (n^+ - n^-)$ sur 4 chiffres.*
   - *remplace n par n'*

Utilisez le programme pour prévoir le comportement asymptotique (à savoir pour $m \to \infty$) de la suite des nombres affichés à l'écran. Dans votre programme, vous pouvez utiliser le fait que si `t` *est une tuple de nombres alors* `tuple(sorted(t))` *est la tuple qui contient les mêmes éléments que* `t` *triés de façon croissante.*

Échappatoires

Exercice 25 (`for` **et** `continue`) *Que fait ce programme ? Transformez-le en un programme qui utilise une boucle* `while` *et qui a le même comportement.*

```
x=0
for i in range(10):
    if (i>7):
        continue
    x=x+1
    print(x)
```

Pour en savoir plus sur ces suites, lire `https://fr.wikipedia.org/wiki/Algorithme_de_Kaprekar`.

TP4

Passage de valeurs immuables

Exercice 26 *Que fait le programme suivant ?*

```
def f(t):
    t=t+(3,)
def g(t):
    return t+(3,)
def main():
    t=(1,2)
    f(t)
    print(t)
    print(g(t))
    print(t)
main()
```

Variables globales et locales

Exercice 27 *Dans le programme suivant on trouve* 10 *appels à la fonction* print. *Pour chaque appel vous devez prévoir combien de fois il sera exécuté et avec quelles valeurs. Rappel : vous pouvez visualiser l'exécution avec* *http://www.pythontutor.com/*.

```
x=5
y=6
def portee(x):
    global y
    x=x+1
    print(x)                                    #(1)
    print(y)                                    #(2)
    y=20
    print(y)                                    #(3)
    y=y+3
    print(y)                                    #(4)
def main():
    x=4
    portee(x)
    print(x)                                    #(5)
    controle()
def controle():
    if (x>0):
        print(x-1)                              #(6)
    else:
```

```
        print(x+1)                              #(7)
    acc=1
    n=2
    for k in range(1,n+1):
        acc=k*acc
        print(acc)                              #(8)
    i=2
    while (i>0):
        acc=acc-i
        i=i-1
        print(acc)                              #(9)
        if (i==1):
            break
        else:
            continue
    print(f(3))                                 #(10)
def f(x):
    if(x==0):
        return 1
    elif (x==1):
        return 2
    else:
        return f(x-1)*f(x-2)
main()
```

Évaluation de polynômes

Exercice 28 (encore Horner) *Écrire un programme qui lit une base B, $2 \leq B \leq 16$, et une chaîne de caractères $d_n \dots d_0$ qui correspond à un nombre en base B et qui imprime à l'écran la valeur du nombre en base 10. Pour améliorer l'efficacité du calcul, notez que la valeur*

$$\Sigma_{i=0,\dots,n} d_i \cdot B^i$$

peut être vue comme l'évaluation d'un polynôme à coefficients $d_n, \dots, d_0$ dans le point B. On peut donc appliquer la méthode de Horner dont on rappelle ci-dessous le code dans sa version itérative. Ici, n *est le degré du polynôme,* x *le point sur lequel il est évalué et* a *est une* tuple *telle que* a[i] *correspond au coefficient* i *du polynôme.*

```
def horner_it(n,x,a):
  h=a[n]
  for i in range(n-1,-1,-1):
      h=a[i]+x*h
  return h
```

Puissance de la récursion

Exercice 29 (encore Hanoï) *On ajoute la contrainte suivante au problème des tours d'Hanoï : le transfert des disques est possible seulement de* façon circulaire*, c'est-à-dire les seuls transferts autorisés sont $1 \to 2$, $2 \to 3$ et $3 \to 1$. Trouvez une méthode de solution pour cette variante et programmez-la. On rappelle ci-dessous une solution pour la version standard du problème :*

```
def hanoi(n, p1, p2):
    p3=6-p1-p2
    if (n==1):
        print(p1,'->',p2)
    else:
        hanoi(n-1,p1,p3)
        print(p1,'->',p2)
        hanoi(n-1,p3,p2)
```

TP5

Recherche dichotomique dans un tableau trié

Exercice 30 *Programmez une fonction qui prend en argument un tableau (ou une tuple) trié par ordre croissant et une valeur et retourne la première position de la valeur dans le tableau ou l'entier* -1 *si la valeur n'est pas dans le tableau (rappel : on compte les positions à partir de* 0*). Réflechissez à comment exploiter l'information que le tableau est trié pour améliorer l'efficacité de votre algorithme !*

Inverse multiplicative

Exercice 31 *Si* p *est un nombre premier alors les entiers modulo* p *forment un corps avec l'addition et la multiplication. En particulier ceci veut dire que pour tout* $n \in \{1, \ldots, p-1\}$ *il existe unique* $m \in \{1, \ldots, p-1\}$ *tel que* $(m \cdot n \equiv 1) \mod p$*. Au passage, pour* $p \geq 2$ *la réciproque est aussi vraie : si tout* $n \in \{1, \ldots, p-1\}$ *a une inverse multiplicative alors* p *est premier.*

Écrire un programme qui lit un nombre premier p *et imprime les inverses multiplicatives de* $1, \ldots, p-1$*. Par exemple, pour* $p = 5$ *le programme imprime :*

```
1 : 1
2 : 3
3 : 2
4 : 4
```

Vous pouvez supposer que p *est petit et que donc une méthode qui énumère toutes les possibilités fait l'affaire. Si* p *est grand et si on a juste besoin de connaître un certain nombre (raisonnable) d'inverses multiplicatives alors une généralisation de l'algorithme d'Euclide permet de trouver une solution de façon efficace.*

Permutations

Une *permutation* sur un ensemble fini admet une représentation naturelle en tant que tableau. Une permutation sur l'ensemble $\{0, 1, \ldots, n-1\}$ est une fonction bijective $p : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$. On représente une telle permutation par un tableau d'entiers de taille n qui contient les entiers $\{0, 1, \ldots, n-1\}$ exactement une fois. Soit id la permutation identité et $\circ$ la composition de permutations. L'ensemble des permutations sur l'ensemble $\{0, 1, \ldots, n-1\}$ est un groupe commutatif. En particulier, chaque permutation admet une permutation inverse par rapport à la composition. Un point fixe d'une permutation p est un élément $i \in \{0, \ldots, n-1\}$ tel que $p(i) = i$.

Exercice 32 (bonne formation) *Programmez une fonction qui prend en argument un tableau et vérifie qu'il représente une permutation (un tableau avec n éléments représente une permutation ssi les valeurs de ses composantes sont exactement les entiers dans l'ensemble* $\{0, 1, \ldots, n-1\}$*).*

Exercice 33 (composition) *Programmez une fonction qui prend en argument deux permutations (représentées par deux tableaux* p *et* q*) et retourne la représentation de la permutation composition '*$p \circ q$*'.*

Exercice 34 (inverse) *Programmez une fonction qui prend en argument une permutation (représentée par le tableau* p*) et retourne la représentation de la permutation inverse de* p*.*

Exercice 35 (nombre points fixes) *Programmez une fonction qui prend en argument une permutation (représentée par le tableau* p*) et retourne le nombre de points fixes de* p*.*

Tableaux de tableaux

Exercice 36 (énumération par diagonale) *On souhaite imprimer les éléments d'une matrice* a *(un tableau de tableaux) avec la contrainte que l'élément* a[i][j] *doit être imprimé avant l'élément* a[k][l] *si* $i + j < k + l$*. Par exemple, si* a *est comme suit :*

8	2	1	4
3	1	9	10
4	5	7	3

en supposant a[0][0] $= 8$*,* a[0][1] $= 2$*, ..., une impression qui respecte la contrainte énoncée (il y en a d'autres) est :* $8, 2, 3, 4, 1, 1, 9, 5, 4, 10, 7, 3$ *; on imprime donc 'par diagonale'.*

Entrée-Sortie sur fichiers

En général pour lire ou écrire des données d'une certaine taille on utilise des fichiers. Ici on fait l'hypothèse que les fichiers se trouvent dans le même répertoire que le programme python qui les manipule. Le nom d'un fichier est une chaîne de caractères. Toute interaction entre programme et fichier se passe de la façon suivante :

— le programme ouvre le fichier (open) en lui associant un nom interne et en précisant les modalités du traitement qu'il souhaite effectuer (lecture, écriture,...)
— le programme interagit avec le fichier,
— le programme ferme (close) le fichier.

Dans la suite on suppose que 'foo' est le nom du fichier, f son nom interne au programme, et m la modalité de traitement. Voici les principales commandes dont on dispose :

f=open('foo',m) On ouvre le fichier foo en lui associant le nom f ; m peut être :
- 'r' pour *read*, si on veut lire le fichier qui existe déjà,
- 'w' pour *write* si on veut écrire dans le fichier ; s'il n'existe pas il est crée et sinon il est écrasé,
- 'a' pour *append* si on veut écrire dans le fichier ; s'il n'existe pas il est crée et s'il existe déjà on ajoute ce qu'on écrit à la fin du fichier.

f.read() On lit tout le fichier comme une chaîne de caractères.

f.readlines() On lit tout le fichier comme un tableau dont les éléments sont les lignes du fichier vues comme chaînes de caractères (avec \n à la fin).

f.readline() Lit la prochaine ligne et avance implicitement le curseur de lecture. Si on est à la fin du fichier on lit la chaîne vide.

f.write(s) On écrit la chaîne s à la fin du fichier.

f.close() Pour clore un fichier.

Remarque En python, on peut considérer qu'un fichier est une tuple qui a autant de composantes que de lignes. Ainsi une façon agréable de programmer un parcours des lignes est d´écrire :

```
for l in f:
    C
```

ce qui est équivalent à :

```
l=f.readline()
while (l!=''):
    C
    l=f.readline()
```

Exemple : trier les lignes d'un fichier

On écrit un programme qui lit les lignes d'un fichier (qui existe déjà) et écrit les lignes triées dans un autre fichier. On pourrait coder les noms des fichiers *en dur* dans le programme ou faire en sorte que le programme interagisse avec l'utilisateur pour qu'il lui fournisse les noms des fichiers. Cependant une façon plus rapide de procéder est de permettre à l'utilisateur de fournir les noms des fichiers quand il lance l'interprétation du programme. Par exemple, supposons que l'utilisateur souhaite trier les lignes du fichier f1 par ordre alphabétique et les écrire dans le fichier f2 et que le programme qui fait ce travail se trouve dans le fichier Sort.py. Dans ce cas, l'utilisateur devrait juste exécuter la commande :

```
python3 Sort.py f1 f2
```

Le programme dans Sort.py est ci-dessous. On remarquera qu'il va chercher les noms f1 et f2 dans le tableau argv du module sys. Il s'agit là d'une *convention* qui veut que les chaînes de caractères qui sont passées au programme (dans notre cas f1 et f2) soient mémorisées dans un tableau qui s'appelle argv. On remarquera que le premier argument se trouve à la position 1 et non pas à la position 0. En effet, la convention en question veut aussi que la position 0 mémorise le nom du fichier python qu'on exécute.

```
import sys
def main():
    inSort = sys.argv[1]
    outSort = sys.argv[2]
    f=open(inSort,'r')
    t=[]
    for ligne in f:
        t.append(ligne)
```

En Spyder, on peut utiliser un bouton *options d'exécution* pour passer les noms f1 et f2 à l'interprète.

```
    f.close()
    t.sort()
    f=open(outSort,'w')
    for ligne in t:
        f.write(ligne)
    f.close()
main()
```

Exercice 37 (lire des valeurs numériques) *On souhaite écrire un programme qui lit un fichier qui contient des flottants et l'imprime à l'écran. On fait l'hypothèse que deux flottants sur la même ligne du fichier sont séparés exactement par un espace. Les flottants qui se trouvent sur la même ligne du fichier seront imprimés sur la même ligne à l'écran et ils seront séparés par un symbole de tabulation pour que les nombres soient alignés (du moins si le nombres ne sont pas trop longs...). Par exemple, si le contenu du fichier a la forme :*

```
12.4 3434 12
13 17.1
19 18 1999
```

la sortie pourrait avoir la forme :

```
12.4 3434.0 12.0
13.0 17.1
19.0 18.0 1999.0
```

Note technique *Si* x *est un flottant et* n *est un nombre positif alors* round(x, n) *est le nombre* x *arrondi à* n *chiffres à droite de la virgule. Cette fonction peut être utilisée pour contrôler les nombres de chiffres qu'on va afficher à l'écran. Attention,* round(0.001,2) *va s'afficher comme* 0.0 *!*

Exercice 38 (extraction de mots) *On va supposer qu'un* mot *est une suite de caractères alphabétiques en minuscules ou majuscules ; pour simplifier, on suppose qu'il n'y a pas d'accents. Un fichier peut contenir des mots ainsi que d'autres caractères (espaces, ponctuation, chiffres,...) Écrire un programme* Extract.py *tel que :*

```
python3 Extract.py f1 f2
```

va extraire tous les mots du fichiers f1 *et les écrire en minuscules, sans répétition, un par ligne et par ordre alphabétique dans* f2.

Il est possible que parmi les méthodes suivantes certaines permettent une simplification de votre programme. Si s *est une chaîne de caractères et* c *un caractère alors :*

- s.lower() *transforme tous les caractères majuscules dans* s *en caractères minuscules. Par exemple,* 'A?b\n'.lower() *donne* 'a?b\n'.
- s.split(c) *produit un tableau dont les composantes sont les chaînes de caractères dans* s *qui sont séparées par* c. *Par exemple,* 'xyz'.split('y') *donne* ['x','z']. *Que se passe-t-il si on a deux occurrences consécutives du caractère* c *?*
- c.join(t) *si* t *est un tableau de chaînes de caractères alors on obtient une chaîne qui résulte de la concaténation des éléments de* t *en interposant entre chaque couple d'éléments le caractère* c. *Par exemple,* 'y'.join(['x','z']) *donne* 'xyz'.

TP6

Complexité dans le pire des cas

Exercice 39 (carrés itérés) *Programmez l'algorithme des carrés itérés sur les entiers modulo.*

Exercice 40 (Euclide et Fibonacci) *On va montrer que la suite de Fibonacci est un bon test pour la complexité de l'algorithme d'Euclide pour le calcul du pgcd.*

1. *Considérez la suite de Fibonacci :* $f(0) = 0$, $f(1) = 1$ *et* $f(n) = f(n-1) + f(n-2)$ *si* $n \geq 2$. *Combien de divisions faut-il pour calculer* $pgcd(f(n), f(n-1))$, $n \geq 3$, *avec l'algorithme d'Euclide ? Testez !*
2. *On cherche à trouver une* borne supérieure *au nombre de divisions effectuées par l'algorithme d'Euclide. Considérons deux pas consécutifs de calcul. En supposant* $a > b$ *et* $a \mod b \neq 0$ *on a :*

$$\begin{aligned} a &= b \cdot q + c, \quad 0 < c < b \\ b &= c \cdot q' + d \quad 0 \leq d < c \ . \end{aligned}$$

Montrez que $a > 2c$. *En supposant que* a *est représenté sur* n *bits et* $a > b > 0$ *donnez une borne supérieure au nombre de divisions nécessaires pour le calcul de* $pgcd(a, b)$.

Exercice 41 (terminaison avec probabilité 1**)** *Considérez le programme :*

```
import random
while True:
    if random.randint(0,1)==0:
        break
```

Dans le pire des cas ce programme boucle, mais en pratique il termine toujours. En combien d'itérations ? Testez ! En moyenne, combien de fois faut-il jouer à pile ou face pour avoir pile ?

Générateurs de permutations

Exercice 42 (générateur de combinaisons) *Adaptez la fonction* permplace *ci-dessous pour qu'elle génère avec probabilité uniforme un échantillon de* k *éléments choisis parmi* n *éléments.*

```
import random
def permplace (n):
     t=[i for i in range(n)]
     for i in range(n):
```

```
        j=random.randint(0,(n-i-1))+i
        t[i],t[j]=t[j],t[i]
    return t
```

Exercice 43 (test de générateurs de permutation) *On considère une façon* alternative *de générer une permutation aléatoire :*

```
import random
def permall(n):
    t=[i for i in range(n)]
    for i in range(n):
        j=random.randint(0,n-1)
        t[i],t[j]=t[j],t[i]
    return t
```

On se focalise sur la génération de permutations de 3 *éléments. Écrire un programme qui génère* $6n$ *permutations avec* permall *et* permplace *et imprime leur distribution. A partir de quel* n *peut-on observer que probablement* permall *ne génère pas une distribution de façon uniforme ? Une façon de mesurer la qualité d'une distribution uniforme est de calculer son écart type par rapport à la moyenne attendue. La fonction* stdev *(*standard deviation*) du module* statistics *prend en entrée une liste de valeurs* $v_1, \ldots, v_n$ *et la moyenne attendue* μ *et calcule pour vous l'écart type des valeurs par rapport à la moyenne, à savoir :*

$$\sqrt{\frac{1}{n}\Sigma_{i=1,\ldots,n}(v_i-\mu)^2} = \sqrt{\frac{1}{n}(\Sigma_{i=1,\ldots,n}v_i^2)-\mu^2}\ .$$

Pouvez-vous proposer un argument mathématique qui montre que la fonction permall *ne génère pas une permutation avec distribution uniforme ?*

Complexité moyenne et complexité amortie

Exercice 44 (test de l'essai par division) *Utilisez la méthode du crible d'Eratosthène (dont on rappelle une programmation possible ci-dessous) pour construire un tableau qui contient les nombres premiers compris entre* 2 *et* $10^{5/2}$.

```
import math
def filtre (n):
    f=[True for i in range(n+1)]
    r =int(math.sqrt(n))
    for i in range(2,r+1):
        for j in range(i,(n//i)+1):
            f[i*j]=False
    prime=[]
    for i in range(2,n+1):
        if f[i]:
            prime.append(i)
    return prime
```

Estimez le nombre moyen d'opérations de division qu'il faut effectuer pour déterminer si un nombre compris entre 10^4 *et* 10^5 *est premier. Il s'agit donc de répéter plusieurs fois l'expérience suivante : tirer au hasard un nombre compris entre* 10^4 *et* 10^5 *et compter le nombre de divisions qu'il faut effectuer pour déterminer s'il est premier. Comparez avec l'estimation du nombre de divisions pour un nombre premier* p *compris entre* 10^4 *et* 10^5 *(on peut exploiter la première expérience pour obtenir cette estimation car l'essai par division nous dit aussi si le nombre est premier...).*

Exercice 45 (simulation d'un tableau dynamique) *Cet exercice est motivé par les tableaux dynamiques de* python *et en particulier par le souhait d'estimer le coût des méthodes* append *et* pop. *On considère un système dont l'état est décrit par un couple* $(i, 2^k)$ *où* $2^k/3 \leq i \leq 2^k$. *Le système peut effectuer les transitions suivantes :*

- $(i, 2^k) \to (i+1, 2^k)$ *avec un coût* 1 *si* $i < 2^k$.
- $(i, 2^k) \to (i+1, 2^{k+1})$ *avec un coût* i *si* $i = 2^k$.
- $(i, 2^k) \to (i-1, 2^k)$ *avec un coût* 1 *si* $i > (2^k/3)$.
- $(i, 2^k) \to (i-1, 2^{k-1})$ *avec un coût* i *si* $i > 0$ *et* $i = (2^k/3)$.

1. *Pour estimer le coût moyen d'une opération, effectuez une simulation du système. A chaque étape, on incrémente le compteur* i *si* $i = 0$ *et sinon on l'incrémente avec probabilité* $1/2$ *et on le décrémente avec probabilité* $1/2$.
2. *Soit* $n = 2^k$. *Une suite de transitions défavorables consiste à faire osciller le système entre l'état* $(n/3 - 1, n/2)$ *et l'état* $(n/2 + 1, n)$. *Dans ce cas, estimez le coût amorti d'une opération pour* n *qui tend vers* $+\infty$.

Optimisation

Exercice 46 (association optimale) *Cet exercice est motivé par le problème de la multiplication de* n matrices *:*

$$M_1 \times M_2 \times \cdots \times M_n$$

Tout ce qu'il faut savoir est que :

- *A chaque matrice* M *on associe une dimension qui est un couple* (m, n) *de nombres naturels* $(m, n \geq 1)$.
- *On peut multiplier une matrice de dimension* (m, n) *par une matrice de dimension* (p, q) *seulement si* $n = p$. *Dans ce cas on obtient une matrice de dimension* (m, q) *pour un coût égal à* $m \cdot n \cdot q$.
- *La multiplication de matrices (avec dimensions compatibles) est une opération associative mais pas commutative.*

On veut calculer le produit $M_1 \times \cdots \times M_n$ *de* n *matrices dont les dimensions sont compatibles et on note* (d_{i-1}, d_i) *la dimension de la matrice* M_i. *Pour obtenir le résultat, il faut effectuer* $n-1$ *multiplications binaires mais dans quel ordre ? Par exemple, si* $d_0 = 10$, $d_1 = 1$, $d_2 = 10$ *et* $d_3 = 1$ *on peut calculer* $(M_1 \times M_2) \times M_3$ *ou* $M_1 \times (M_2 \times M_3)$. *Le premier calcul coûte* 200 *et le deuxième* 20 *! On a donc un problème d'optimisation : il faut trouver une façon d'effectuer les multiplications qui* minimise *le coût du calcul. On dénote par* $c(i, j)$, $i \leq j$, *le*

En théorie des probabilités, on dirait que le système effectue une *marche aléatoire* en dimension 1 avec une *barrière réfléchissante*.

coût minimal pour calculer $M_i \times \cdots \times M_j$. *On remarque que la fonction* $c(i,j)$ *doit satisfaire la condition suivante pour* $1 \leq i \leq j \leq n$:

$$c(i,j) = \begin{cases} 0 & \text{si } i = j \\ min_{k=i,\ldots,j-1}(c(i,k) + c(k+1,j) + (d_{i-1} \cdot d_k \cdot dj)) & \text{si } i < j \end{cases}$$

La valeur qui nous intéresse est $c(1,n)$ *et cette valeur s'exprime récursivement en fonction des valeurs de certains* $c(i,j)$ *où* $(j-i) < (n-1)$.

1. *Explicitez la fonction c dans l'exemple ci-dessus.*
2. *Programmez la fonction c. Si vous suivez à la lettre la condition ci-dessus vous obtiendrez un programme peu efficace dans lequel on recalcule plusieurs fois la fonction c sur les mêmes arguments.*
3. *Modifiez votre programme pour qu'il calcule et mémorise d'abord* $c(i,i+1)$ *pour* $i = 1,\ldots,n-1$, *puis* $c(i,i+2)$ *pour* $i = 1,\ldots,n-2$, *puis ..., puis* $c(i,i+(n-1))$ *pour* $i = 1$ *(à savoir* $c(1,n)$, *la valeur qui nous intéresse !).*

TP7

Visualiser les fonctions

Le module matplotlib permet d'obtenir une représentation graphique de fonctions. Pour commencer il faut importer le module :

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

Ensuite, il faut preparer deux tableaux de flottants de la même taille pour les abscisses et les ordonnées. Par exemple, supposons :

```
xord=[0,1,2,3,4]
yord=[0,1,4,9,16]
```

On dispose ensuite de 3 fonctions qui construisent des représentations différentes :

— plt.scatter(xord,yord) construit une représentation *discrète* du graphe comme *nuage de points*,

— plt.plot(xord,yord) approxime une fonction *continue* en connectant les points par des segments ; évidemment plus les points en abscisse sont nombreux plus on lisse les angles,

— plt.bar(xord,yord) représente la fonction par un histogramme (un diagramme à bâtons).

Les 3 fonctions ci-dessus construisent une représentation graphique mais elles ne l'affichent pas ! Pour ce faire, on appelle plt.show(). En resumant, pour visualiser le graphe ci-dessous comme un nuage de points on écrira :

```
import matplotlib.pyplot as plt
xord=[0,1,2,3,4]
yord=[0,1,4,9,16]
plt.scatter(xord,yord)
plt.show()
```

Notez qu'après l'appel à show, l'exécution du programme se suspend ; une façon de continuer l'exécution est de clore la fenêtre. Ce qu'on vient de décrire est une toute petite partie des possibilités offertes par le module ! Par exemple, on peut superposer plusieurs représentations, jouer avec les couleurs, nommer les axes, sauver les images, ouvrir plusieurs fenêtres,... On réfère le lecteur à la documentation pour plus d'informations.

Exercice 47 (scatter, plot et bar) *1. Écrire un programme qui lit un fichier de $2 \cdot n$ flottants $x_1, y_1, \ldots, x_n, y_n$ et visualise le nuage de points $(\mathsf{x}_1, \mathsf{y}_1), \ldots, (\mathsf{x}_n, \mathsf{y}_n)$. Tout ce qu'on sait sur les flottants dans le fichier est qu'ils sont séparés par un ou plusieurs caractères parmi les suivants : espace, tabulation (\t) ou saut de ligne (\n).*

https://matplotlib.org/3.1.1/users/index.html

2. *Visualisez le polynôme $p(x) = x^3 - 2x + 2$ et déterminez un intervalle qui contient un zéro.*
3. *Écrire une fonction qui lit un fichier qui contient un long texte en anglais (disons au moins* 1000 *caractères) et visualise la distribution des caractères alphabétiques (a-z) dans le texte comme un histogramme (on ne fera pas de différence entre minuscules et majuscules).*

Méthode dichotomique

On rappelle la mise-en-oeuvre de la méthode dichotomique discutée en cours.

```
def dicho(f,a,b):
    prec=2**-52
    eps=10e-6
    nitmax=200
    fa=f(a)
    fb=f(b)
    fm=f((a+b)/2)
    assert sign(fa)*sign(fb)<0
    nit=0
    while ((b-a)/(a+b)>prec and nit<nitmax):
        m=(a+b)/2
        fm=f(m)
        if fm==0:
            return (m,nit)
        elif sign(fa)*sign(fm)<0:
            b,fb=m,fm
        else:
            a,fa=m,fm
        nit=nit+1
    if abs(fm)>eps:
        print('warning')
    return ((a+b)/2,nit)
```

Exercice 48 (applications de la méthode dichotomique) *1. Définissez une fonction inv qui depend d'une variable globale N (qu'on suppose contenir un flottant positif) telle que $inv(x) = 0$ ssi $x = 1/N$. Il est entendu que la définition de inv n'utilise pas la division. Utilisez la méthode dichotomique pour calculer $1/N$.*

2. *Définissez une fonction sq qui depend d'une variable globale N (qu'on suppose contenir un flottant positif) telle que $sq(x) = 0$ ssi $x = \sqrt{N}$. Il est entendu que la définition de sq n'utilise pas la racine carrée. Utilisez la méthode dichotomique pour calculer $\sqrt{N}$.*

Exercice 49 (méthode dichotomique en scipy**)** *La fonction bisect du module* scipy *implémente aussi la méthode dichotomique. Pour chercher un zéro de la fonction* f *dans l'intervalle $[a, b]$ on écrit :*

```
from scipy import optimize
optimize.bisect(f,a,b)
```

On pourra consulter la documentation pour plus d'informations. Vous disposez maintenant de 3 *façons pour calculer la racine carrée : la fonction* sqrt *du module* math*, la fonction* bisect

https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.bisect.html

qu'on vient de mentionner et votre mise en oeuvre de la méthode dichotomique. Comparez les résultats obtenus en faisant varier la valeur N dont on calcule la racine carrée. Par exemple, $N = 2^i$ pour $i = 1, 2, 3, 4, \ldots$

Méthode de Newton-Raphson

On rappelle une mise en oeuvre possible de la méthode de Newton-Raphson.

```
def newton(f,fprime,x):
  eps=10e-10
  nitmax=50
  nit=0
  while (nit<nitmax):
      nit=nit+1
      fp=fprime(x)
      if abs(fp)<eps:
          print('derivee proche de 0 !')
      xnext=x-(f(x)/fp)
      if abs(xnext-x)<eps:
          return xnext
      else:
          x=xnext
  print('pas de convergence')
```

La méthode de Newton-Raphson est aussi disponible dans le module optimize de scipy. Si on écrit :

```
from scipy import optimize
optimize.newton(f,x,fprime)
```

on obtient un zéro de la fonction f calculé avec la méthode de Newton-Raphson avec x comme point initial et fprime comme fonction dérivée (la dérivée peut être omise).

Exercice 50 (test vitesse de convergence)

1. *Utilisez la méthode de Newton-Raphson pour calculer la racine carrée d'un nombre positif (attention au choix du point initial).*
2. *Comparez le nombre d'itérations nécessaires à la convergence de la méthode de Newton-Raphson et de la méthode dichotomique.*

Exercice 51 (approcher la dérivée) *Plutôt que calculer explicitement la dérivée de la fonction dont on cherche le zéro, on peut utiliser l'approximation suivante :*

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

pour h 'petit' relativement à x. Par exemple, $h \approx x/2^{10}$. Modifiez votre mise en oeuvre de la méthode de Newton-Raphson pour qu'elle calcule une approximation numérique de la dérivée. Vérifiez la qualité du résultat obtenu en termes de précision et de nombre d'itérations.

Exercice 52 (un cas pathologique) *On cherche un zéro du polynôme $p(x) = x^3 - 2x + 2$ (c'est le polynôme de l'exercice 47!).*

— *Proposez un intervalle pour la recherche d'un zéro avec la méthode dichotomique et vérifiez la convergence de la mise en oeuvre.*

— *Proposez un point de départ pour la méthode de Newton-Raphson et vérifiez la convergence de la mise en oeuvre.*
— *Que se passe-t-il si on prend comme point de départ soit* 0 *soit* 1 *?*

TP8

Élimination de Gauss : programmation minimaliste

On rappelle ci-dessous la programmation *minimaliste* de la méthode d'élimination de Gauss vue en cours (pour un système avec n équations linéaires et n inconnues).

```
def pivot(a,i):
    n=len(a)
    for k in range(i,n):
        if (a[k][i]!=0):
            return k
    return -1
def perm(a,i,j):
    n=len(a)
    for k in range(i,n+1):
        a[i][k],a[j][k]=a[j][k],a[i][k]
    return None
def trans(a,i):
    n=len(a)
    for k in range(i+1,n):
        m=-a[k][i]/a[i][i]
        for j in range(i+1,n+1):
            a[k][j]=a[k][j]+ (m* a[i][j])
    return None
def gauss(a):
    n=len(a)
    for i in range(n-1):
        j=pivot(a,i)
        assert(j!=-1)
        if (i!=j):
            perm(a,i,j)
        trans(a,i)
    return None
def solvetri(a):
    n=len(a)
    x=[0 for i in range(n)]
    for i in range(n-1,-1,-1):
        x[i]=a[i][n]
        for j in range(i+1,n):
            x[i]=x[i]-(a[i][j]*x[j])
        assert(a[i][i]!=0)
        x[i]=x[i]/a[i][i]
    return x
```

Exercice 53 (vérification) *1. Écrire une fonction qui génère un système de taille n en tirant des nombres entiers de façon aléatoire dans un intervalle $[-M, M]$.*

2. *Écrire une fonction qui prend le système et la solution générée et calcule le vecteur différence $Ax - b$ (si la solution était exacte on devrait obtenir un vecteur de 0).*

Exercice 54 (maxpivot) *Modifiez la fonction pivot pour qu'elle retourne l'indice de la ligne $k \geq i$ qui contient le coefficient $a_{k,i}$ le plus grand en valeur absolue. Cette variante est souvent utilisée pour améliorer la fiabilité du calcul sur les flottants.*

Exercice 55 (forme échellonnée) *Adaptez la méthode d'élimination à un système $m \times n$. Dans ce cas, vous allez transformer la matrice en forme échellonnée. Si la solution du système est unique vous allez la calculer et autrement vous allez imprimer un message qui précise le nombre de solutions (zéro ou une infinité).*

Utilisation de la fonction du module numpy

Exercice 56 (solution avec numpy) *Le module* numpy *contient un type* array *qui est une spécialisation du type* list *dans lequel les éléments du tableau sont* homogènes *(une condition plus forte que avoir le même type* python*) et les méthodes qui modifient la taille du tableau sont interdites. Dans cet exercice, on souhaite juste se servir de la fonction* numpy *qui calcule la solution d'un système d'équations linéaires. Pour ce faire, on commence par importer le module* numpy*.*

```
>>> import numpy as np
```

Il est facile de convertir une valeur de type list *en une valeur de type* array *juste en appliquant la fonction* np.array *comme dans l'exemple suivant :*

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

La fonction np.linalg.solve *permet de résoudre un système d'équations linéaires. La fonction attend en argument une valeur de type* array *d'*array A *et une valeur de type* array b*. Si à la place on lui passe une tableau de tableaux et un tableau, ces valeurs sont converties automatiquement en valeurs de type* array*. En continuant notre exemple, on peut écrire :*

```
>>> b=np.array([5,6])
>>> np.linalg.solve(A,b)
array([-4. ,  4.5])
```

Programmez une fonction qui prend en argument une matrice de coefficients $n \times n + 1$, la convertit dans le format attendu par la fonction de numpy *et retourne la solution calculée.*

Documentation dans `https://docs.scipy.org/doc/numpy/user/basics.html`

Calcul dans les rationnels

Exercice 57 (module fractions**)** *La solution d'un système d'équations linéaires utilise seulement les opérations arithmétiques (somme, soustraction, multiplication, division). Si les coefficients du problème sont rationnels tout le calcul peut être mené dans le corps des rationnels. Le module* fractions *permet de représenter les nombres rationnels sans approximation. Voici un petit résumé. On commence par importer en faisant :*

```
from fractions import Fraction
```

Le rationnel $3/4$ *s'écrit* Fraction(3,4)*. Si on applique les opérations arithmétiques à des nombres de cette forme, le calcul se fait automatiquement de façon exacte. Par exemple,* Fraction(3,4)+Fraction(2,3) *produit comme résultat* Fraction(17,12)*. Encore mieux, si on applique* Fraction *à un flottant on obtient une conversion en rationnel. Attention, le rationnel qui est calculé est celui qui correspond au flottant dans sa représentation IEEE-754 ! Par exemple, on sait que* $1/2$ *est représenté de façon exacte dans ce standard mais* $1/10$ *ne l'est pas. Ainsi,* Fraction(0.5) *vaut* Fraction$(1, 2)$ *mais* Fraction(0.1) *vaut*

$$\text{Fraction}(3602879701896397, 36028797018963968)$$

ce qui n'est pas tout à fait $0, 1$*. En d'autres termes,* Fraction$(1, 10)$ *n'est pas égal à* Fraction$(1/10)$*. Pour preuve :*

```
>>> Fraction(1,10)==Fraction(1/10)
False
```

Vérifiez que si on démarre avec une système d'équations avec des coefficients Fraction *le programme présenté au début du TP va produire une solution* exacte*.*

Génération d'entrées et tests

Exercice 58 (entrées) *On va maintenant considérer différentes façons de générer des entrées pour la fonction* gauss*. Les fonctions à écrire doivent retourner un tableau constitué de* n *tableaux qui contiennent* $n + 1$ *scalaires.*

1. *Écrire une fonction qui lit* $n \times (n + 1)$ *flottants dans un fichier et construit le tableau correspondant pour la la fonction* gauss*. Vous pouvez simplement adapter ce qui a été fait dans le TP5.*
2. *Une matrice tridiagonale est une matrice dont tous les coefficients non nuls sont sur la diagonale principale ou sur les diagonales juste en dessous ou en dessus de la diagonale principale. Ces matrices apparaissent naturellement dans certains problèmes de calcul numérique. Écrire une fonction* trid *qui prend en argument* 5 *valeurs* a, b, c, d, n *et retourne un tableau tridiagonal pour la fonction* gauss*. Par exemple, si* $n = 4$ *le système devrait avoir la forme :*

$$\left[\begin{array}{cccc|c} b & c & 0 & 0 & d \\ a & b & c & 0 & d \\ 0 & a & b & c & d \\ 0 & 0 & a & b & d \end{array}\right]$$

Documentation dans `https://docs.python.org/fr/3/library/fractions.html`

3. *Écrire une fonction* hilb *qui prend en argument une valeur n et génère un tableau* t *pour la fonction* gauss *tel que pour $i = 0, \ldots, n-1$ et $j = 0, \ldots, n$:*

$$\mathsf{t}[i][j] = \frac{1}{i+j+1}$$

Les premières n colonnes de la matrice décrite constituent ce qu'on appelle la matrice de Hilbert. Cette matrice est connue pour être un test de résistance redoutable pour la fiabilité du calcul sur les flottants.

Exercice 59 (norme et erreur) 1. *La norme* 1 *d'un vecteur $x = (x_1, \ldots, x_n)$ est la somme des valeurs absolues des composantes de x :*

$$\|x\|_1 = \Sigma_{i=1,\ldots,n}|x_i| \ .$$

Programmez une fonction qui prend en argument un vecteur de valeurs numériques et retourne sa norme 1*.*

2. *En utilisant la norme* 1*, on peut avoir une version vectorielle de l'erreur relative en la définissant (pour x non nul) comme :*

$$\frac{\|x - \widetilde{x}\|_1}{\|x\|_1} \ .$$

Programmez une fonction qui prend en argument deux vecteurs x et y avec x non nul et retourne l'erreur relative de l'approximation de x par y.

Exercice 60 (test fiabilité) *On se propose de tester l'erreur relative introduite par les calculs sur les flottants. On va travailler avec des matrices de Hilbert de dimension n (un cas particulièrement défavorable !) Les coefficients de ce système étant rationnels, on peut calculer une solution exacte x en utilisant le module* fraction*. Par ailleurs, on peut calculer une solution approchée $\widetilde{x}$ soit avec le programme vu en cours (avec ou sans stratégie max pivot) soit avec la fonction disponible dans* numpy*. On peut ensuite convertir la solution approchée $\widetilde{x}$ en solution rationnelle et calculer l'erreur relative de façon exacte en utilisant à nouveau le module* fraction*. A partir de quel n l'erreur relative devient supérieure à* 1 *?*

Exercice 61 (test efficacité) *On se propose de tester l'efficacité des trois méthodes :*

- *solution 'maison' flottante,*
- *solution exacte rationnelle avec module* fraction*,*
- *solution avec module* numpy*.*

Pour faire ces tests, vous allez générer des matrices aléatoires de dimension n, pour $n = 2^i$, $i = 4, 5, 6, \ldots$ et mesurer le temps d'exécution des trois méthodes en utilisant les techniques vues dans le TP6. Pouvez-vous expliquer le classement final ?

Variations

Exercice 62 (tridiagonale) *Formaliser et programmer une spécialisation de la méthode d'élimination de Gauss aux matrices tridiagonales. La complexité de votre méthode devrait être* linéaire *en la dimension du système ce qui est un progrès très sensible par rapport au coût cubique de la méthode générale. Par exemple, votre méthode devrait traiter rapidement*

des matrices tridiagonales de dimension $n = 1000$ ce qui ne devrait pas être le cas de la méthode générale appliquée à la même matrice tridiagonale. Dans la représentation standard du système, on gaspille une grande partie de la mémoire pour mémoriser des 0 *... Proposez une représentation plus compacte d'une matrice tridiagonale et adaptez votre méthode de solution à cette représentation.*

Exercice 63 (solution dans $\mathbf{Z}_p$) *La méthode d'élimination gaussienne peut être utilisée aussi pour résoudre un système dans le corps des entiers modulo p, où p est un nombre premier. Étant donné un nombre premier p, il est facile de définir les opérations d'addition, soustraction et multiplication modulo p. Pour la division, on utilisera l'exercice 31 du TP5 qui calcule les inverses multiplicatives (en veillant à calculer le tableau des inverses une seule fois !) Vérifiez le bon fonctionnement de votre solution en adaptant l'exercice 53.*

Exercice 64 (calcul de l'inverse) *Une matrice A, $n \times n$, est inversible s'il existe une matrice B, $n \times n$, telle que $A \cdot B = I$, où I est la matrice identité qui a des* 1 *sur la diagonale principale et des* 0 *ailleurs. On sait que A est inversible ssi la méthode de Gauss transforme A en une matrice triangulaire supérieure dans laquelle tous les éléments sur la diagonale principale sont différents de* 0*. On peut réduire le calcul de la matrice inversible à la solution de n systèmes d´équations de la forme $Ax_i = e_i$, pour $i = 1, \ldots, n$, où e_i est le vecteur qui vaut* 1 *à la composante i et* 0 *ailleurs. Si ces n systèmes ont une solution unique x_i alors x_i est la i-ème colonne de la matrice B. Clairement, résoudre n systèmes $n \times n$ a une complexité $O(n^4)$. Cependant dans notre cas les n systèmes partagent la même matrice A ! On peut donc penser qu'il y a une façon de partager les calculs et de réduire la complexité. Écrire un programme qui calcule une inverse (si elle existe) en $O(n^3)$; vous adapterez autant que possible les fonctions présentées au début du TP. Vous devez expliquer comment vous avez organisé le calcul pour obtenir une complexité en $O(n^3)$ et comment vous avez vérifiez la correction de votre mise en oeuvre. Notez que la fonction* inv *du module* numpy *calcule aussi l'inverse. Par exemple :*

```
>>> a=[[0,1],[2,3]]
>>> import numpy as np
>>> np.linalg.inv(a)
array([[-1.5,  0.5],
       [ 1. ,  0. ]])
```

TP9

Interpolation avec la matrice de Vandermonde

Exercice 65 *On rappelle que la matrice de Vandermonde V_n pour les points $x_0, \ldots, x_{n-1}$ est définie par :*

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \cdots & \cdots & \cdots & \cdots & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \tag{2}$$

En supposant $x_i \neq x_j$ pour $i \neq j$, la solution du système linéaire $V_n a = y$ où $y = (y_0, \ldots, y_n)$ permet de calculer les coefficients d'un polynôme de degré au plus $n-1$ qui passe par les points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$.

1. *Programmez une fonction qui prend en argument deux vecteurs de flottants x et y avec la même dimension, construit le système linéaire mentionné ci-dessus, calcule sa solution avec la méthode de Gauss et imprime le vecteur a.*
2. *Écrire un programme qui lit deux vecteurs de flottants et ensuite intéragit avec l'utilisateur pour évaluer le polynôme interpolant dans un ou plusieurs points choisis par l'utilisateur.*

Interpolation avec Lagrange

Exercice 66 *On rappelle de suite les fonctions discutées en cours qui permettent d'évaluer le polynôme interpolant de Lagrange.*

```
def lagrange_coef(x,y):
    n=len(x)
    assert(n==len(y))
    Delta = [1 for i in range(n)]
    for i in range(n):
        for j in range(n):
            if i!=j:
                Delta[i]=Delta[i]*(x[i]-x[j])
    c=[y[i] for i in range(n)]
    for i in range(n):
        c[i]=c[i]/Delta[i]
    return c
def interpol(c,x,y,v):
    n=len(c)
    assert(n==len(x) and n==len(y))
    for i in range(n):
```

```
        if (v==x[i]):
            return y[i]
    pi=[1 for i in range(n)]
    for j in range(1,n):
        pi[0]=pi[0]*(v-x[j])
    for i in range(1,n):
        pi[i]=(pi[i-1]*(v-x[i-1]))/(v-x[i])
    sum=0
    for i in range(n):
        sum=sum+c[i]*pi[i]
    return sum
```

Écrire un programme qui lit deux vecteurs de flottants et ensuite intéragit avec l'utilisateur pour évaluer le polynôme interpolant dans un ou plusieurs points choisis par l'utilisateur. Pour chaque point le programme imprimera l'évaluation obtenue en passant par la matrice de Vandermonde et en suivant la méthode de Lagrange.

Calcul de la regression linéaire

Exercice 67 *Soient* $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$*, n points dans* $\mathbf{R}^2$ *où* $x_i \neq x_j$ *si* $i \neq j$*. Le problème de trouver une droite d'équation* $ax + b$ *qui minimise l'erreur (au sens* moindres carrés*) :*

$$\Sigma_{i=0,\ldots,n-1}(ax_i + b - y_i)^2$$

a toujours une solution unique qui s'exprime par :

$$b \;= \overline{y} - a \cdot \overline{x} \qquad a \;= \frac{\langle x,y\rangle - n\cdot\overline{x}\cdot\overline{y}}{\langle x,x\rangle - n\cdot\overline{x}^2}$$

où $x = (x_0, \ldots, x_{n-1})$*,* $y = (y_0, \ldots, y_{n-1})$ *sont des vecteurs,* $\overline{x} = (1/n)(\Sigma_{i=0,\ldots,n-1}x_i)$*,* $\overline{y} = (1/n)(\Sigma_{i=0,\ldots,n-1}y_i)$ *sont les moyennes arithmétiques des vecteurs et* $\langle x, y\rangle$ *dénote le produit scalaire. Programmez une fonction qui prend en argument les vecteurs* x, y *et retourne les coefficients* (a, b)*.*

Moindres carrés avec numpy

Exercice 68 *La fonction* polyfit *permet d'approcher un nuage de points avec un polynôme de degré borné (comme dans l'exercice précédent, on suit l'approche des* moindres carrés*). Par exemple :*

```
import numpy as np
xord=[0,1,2,3,4,5]
yord=[i**2 for i in xord]
deg=1
a=list(np.polyfit(xord,yord,deg))
```

va mémoriser dans a *les coefficients d'une droite qui est assez proche des points alors que si* deg ≥ 2 *on aura modulo des petites erreurs le polynôme* x^2*.*

https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html

1. *Visualisez* a *pour déterminer dans quel ordre les coefficients sont rendus par la fonction* polyfit.
2. *Définir une fonction qui permet d'évaluer le polynôme rendu par* polyfit *(voir exercice 28).*
3. *Visualisez dans la même image le nuage de points (avec* scatter*) et le polynôme (avec* plot*).*

TP10

Conversion entre format csv et valeurs DataFrame

Vous avez probablement déjà utilisé un logiciel pour manipuler des feuilles de calcul. Parmi ces logiciels on trouve, par exemple, LibreOfficeCalc. Une feuille de calcul se présente comme une matrice avec un certain nombre de lignes et de colonnes. On appelle chaque composante de la matrice une *cellule*. Les cellules contiennent ou bien des données (nombres ou chaînes de caractères) ou bien des formules qui typiquement permettent de calculer une valeur à partir de la valeur d'autres cellules. Par exemple, une feuille de calcul pourrait avoir l'apparence suivante :

	A	B	C	D
1	NOM	CC	$EXAMEN$	$MOYENNE$
2	$Marius$	14	13	13.5
3	$Marie$	13	14	13.5

Cette feuille comporte 4 colonnes, 1 ligne avec les noms des colonnes et 2 lignes qui contiennent les données. Une colonne et une ligne déterminent une cellule. Ainsi la note de CC de Marius est mémorisée à la cellule $B2$. Au lieu d'écrire en dur la moyenne on utilise une formule. Par exemple, si on inspecte la cellule $D2$ on y trouve une formule de la forme AVERAGE(B2,C2) qui indique que la valeur de la cellule $D2$ doit être calculée en effectuant la moyenne arithmétique de la valeur des cellules $B2$ et $C2$. L'avantage d'utiliser une formule au lieu d'une donnée est double : d'une part on n'a pas à faire explicitement le calcul, d'autre part toute modification d'une note de CC ou d'examen est immédiatement répercutée dans le calcul de la moyenne. On peut sauver ce document dans un fichier au format csv, disons test.csv (csv étant un acronyme pour *comma separated values*). Le fichier test.csv est un *texte* qu'on peut lire ; dans notre cas, voici son contenu :

```
NOM,CC,EXAMEN,MOYENNE
Marius,14,13,13.5
Marie,13,14,13.5
```

Note technique On remarquera qu'on utilise ici la notation anglo-saxonne pour les nombres avec décimaux (avec un point au lieu d'une virgule). Pour ce faire, on cherche dans le logiciel une façon de configurer les options de langage (dans LibreOfficeCalc on sélectionne *Outils* et ensuite *Langage*). En effet, un nombre avec virgule est sauvé comme une chaîne de caractères et il est ensuite interprété par python en tant que telle ; on perd ainsi les données numériques.

En python, pour lire le fichier test.csv et le sauver en tant que donnée de type DataFrame il suffit d'exécuter :

```
import pandas as pd
df=pd.read_csv('test.csv')
```

Si on imprime la valeur de df on obtient :

```
      NOM  CC  EXAMEN  MOYENNE
0  Marius  14      13     13.5
1   Marie  13      14     13.5
```

On peut créer une valeur identique en python en écrivant :

```
df1 = pd.DataFrame(columns=['NOM','CC','EXAMEN','MOYENNE'],
                   data=[['Marius',14,13,13.5],
                         ['Marie',14,13,13.5]])
```

Note technique Pour avoir exactement le même format notre valeur DataFrame ne donne pas de noms aux lignes.

Pour sauver cette valeur DataFrame dans un fichier test1.csv on écrit :

```
df1.to_csv('test1.csv',index=False)
```

et on peut vérifier que le fichier test.csv est identique au fichier test1.csv.

Note technique Notez qu'on utilise l'option index=False. A défaut, le fichier test1.csv va contenir une colonne additionnelle avec la numérotation des lignes.

Morale On savait déjà que les conversions entre types ne donnent pas toujours le résultat attendu. La situation empire si on cherche à convertir les données d'un langage (une feuille de calcul) dans les données d'un autre langage (python) car des valeurs qui semblent identiques peuvent être représentées de façon différente. Dans ces cas, la prudence est de mise (voir les notes techniques ci-dessus).

Exercice 69 (csv **et** DataFrame) *Le fichier* input_tp10.csv *disponible sur la page* Moodle *du cours contient des informations sur* 985 *ventes de biens immobiliers. Écrire un programme qui :*

1. *lit le fichier et le convertit en une valeur de type* DataFrame*,*
2. *calcule pour chaque immeuble le prix en euro par* m^2 *(dans le fichier le prix est exprimé en dollars et l'unité de mesure pour la surface est le pied carré, on sait que* 1 *pied carré* $= 0,092903$ *mètre carré et on peut supposer* 1 *dollar*$=0,91$ *euro),*
3. *imprime la moyenne des prix exprimés en euro par* m^2*,*
4. *construit un fichier* output_tp10.csv *qui contient uniquement les informations suivantes sur chaque vente : prix en euro par* m^2*, prix (en euro), surface (en* m^2*), code postal.*
5. *visualise le nuage de points obtenu en mettant en abscisse la surface et en ordonnée le prix en euro et le modèle de régression linéaire associé (c'est à dire, la droite qui approxime le nuage de points d'après le critère des moindres carrés).*

Les données sur les ventes sont-elles toutes significatives/complètes ? Considérez un raffinement du programme qui va éliminer d'abord les données inexploitables.

Note *Ce travail doit s'effectuer en* **binôme**. *Le fichier qui contient le programme et que vous déposerez sur* Moodle *au plus tard le* **Dimanche 05/01/2020 (heure de Paris)** *doit s'appeler* Nom1Nom2.py *où* Nom1 *et* Nom2 *sont les noms de famille des auteurs du programme. Les noms et prénoms des auteurs seront aussi précisés en commentaire au début du fichier. Pour tester votre travail, le fichier en question sera placé dans un répertoire qui contient le fichier* input_tp10.csv *et son exécution sera lancée avec la commande* python3 Nom1Nom2.py. *Suite à cette commande, on s'attend à lire à l'écran la moyenne des prix, à observer un diagramme avec le nuage de points et le modèle de régression linéaire et à pouvoir lire le contenu du fichier* output_tp10.csv *généré (par exemple avec* LibreOfficeCalc*). Pour mémoire :*

- *le TP7 introduit la visualisation de fonctions avec* matplotlib,
- *le TP9 explique comment utiliser la fonction* polyfit,
- *le chapitre 10 de la trace de cours donne les premiers rudiments sur le type* DataFrame.