Premiers pas en Python

PTSI Lycée Eiffel

3 novembre 2014

Cet assez long chapître, central dans notre progression cette année, est consacré à l'apprentissage des bases de la programmation. l'enseignement se fera naturellement dans le langage à votre programme, à savoir Python, même si une bonne partie des concepts présentés ici se retrouvent sous une forme ou une autre dans la majorité des langages. Seule la syntaxe varie réellement d'un langage à l'autre, mais les boucles ou autres listes sont présentes par exemple en C ou en Java sous une forme pas très éloignée de ce que nous verrons en Python. Ainsi, en apprenant à programmer en Python, vous pouvez ensuite nettement plus facilement reprendre votre apprentissage dans un autre langage si nécessaire. Le Python présente par ailleurs, comme nous l'avons déjà signalé dans notre premier chapître, l'avantage d'être relativement simple d'utilisation et syntaxiquement léger, ce qui en fait un bon choix pour l'apprentissage d'un premier langage de programmation.

1 Variables et types

La notion de variables est aussi centrale en mathématiques qu'en informatique, même si elle y prend un sens nettement plus concret et pratiquement physique.

Définition 1. Une **variable** du langage Python permet d'associer un nom à un objet destiné à stocker une valeur d'un certain **type**, qui peut évoluer au cours de l'exécution du programme.

Concrètement, on peut considérer qu'une variable est l'adresse d'une case mémoire de l'ordinateur, le nom de la variable étant en quelque sorte une étiquette collée sur la case, et la valeur prise par la variable le contenu de la case. Le **type** de la variable correspond à une catégorie de donnée numérique qui sera stockée d'une certaine manière par la machine. Ainsi, nous avons vu dans le premier chapître que les nombres entiers et les nombres réels étaient gérés très différemment dans un ordinateur, ce qui explique que les variables contenant des nombres entiers soient considérées comme étant de type différent de celles contenant des réels.

Définition 2. L'**affectation** d'une variable consiste simplement à donner une valeur à une variable donnée. C'est l'opération de base sur les variables dans tout langage de programmation, elle s'effectue simplement en Python à l'aide du signe =, en mettant la variable à gauche du symbole, et la valeur à affecter à droite :

$> a = 3$

permet ainsi de créer une variable et de lui affecter en même temps la valeur 3 (la variable créée sera donc de type entier). On utilisera souvent un seul caractère pour désigner nos variables dans des programmes courts, mais pour un projet plus ambitieux, il est fortement conseillé de leur donner des noms éclairants, il n'y a pas de limitation sur la taille du nom d'une variable (par contre, on évite d'y insérer des accents ou des caractères spéciaux).

En Python, il est complètement inutile de définir une variable avant de l'affecter (contrairement à ce qui peut se faire dans d'autres langages). Il est en particulier inutile de typer la variable, c'est-à-dire de préciser quel type de donnée sera contenu dans la variable, Pyhton s'en chargeant systématiquement à la volée. C'est un gros avantage car cela simplifie énormément la programmation, mais cela peut

aussi être dangereux dans la mesure où on ne peut évidemment pas faire n'importe quoi sur n'importe quel type de variables, et qu'on n'a pas vraiment de contrôle sur le typage en Python.

Liste non exhaustive de types de données en Python.
- int : nombre entier (int est l'abréviation d'*integer*).
- long : nombre entier, mais plus gros !
- float : nombre réel.
- complex : nombre complexe.
- str : chaîne de caractère (str est l'abréviation de *string*).
- bool : booléen (variable ne pouvant prendre que deux valeurs : *true* et *false*).

Nous croiserons plus tard dans l'année d'autres types, notamment des types composés comme les listes, auxquelles nous consacrerons toute une partie du cours.

2 Boucles

2.1 Instructions conditionnelles

Si on se sert uniquement d'affectations de variables en guise d'instructions, on ne fera guère plus que des calculs élémentaires avec Python. Pour écrire des programmes mettant en oeuvre de véritables algorithmes, la première nécessite est de pouvoir introduire des embranchements selon qu'une certaine condition est vérifiée ou non. Pour prendre un exemple mathématique simple, on a un tel embranchement quand on applique l'agoritme de résolution des équations du second degré : après avoir calculé Δ, on distingue des cas selon la valeur obtenue. Tous les langages de programmation disposent de structure permettant de faire de telles distinctions, appelées instructions conditionnelles, et imitant la syntaxe informelle suivante : SI une condition est vérifiée, ALORS on effectue telle action, SINON on fait autre chose (éventuellement rien).

Syntaxe des instructions conditionnelles en Python :
> **if** condition :
> instruction1
> (**elif** condition2 :
> instruction2)
> **else** :
> instruction3

Les mots en gras (if, elif et else) sont des mots-clés du langage Python, qui seront reconnus comme faisant partie d'une instruction conditionnelle dès qu'ils apparaissent dans un programme (en particulier, on ne peut par exemple pas les utiliser comme noms de variables). Contrairement à beaucoup de langages, Pyhton ne fait pas apparaitre d'équivalent au ALORS (souvent exprimé par un *then* dans les langages de programmation) dans ses instructions conditionnelles. La syntaxe est en fait structurée par l'**indentation** du texte : dans l'encadré précédent, les instructions sont décalée vers la droite par rapport aux autres lignes. ce n'est pas une coquetterie, mais bel et bien un élément de syntaxe : tout ce qui est décalé vers la droite par rapport au if est considéré comme faisant partie de l'instruction, ce qui évite de devoir placer un *end* pour en indiquer la fin. Les deux lignes médianes, avec le elif, sont mises entre parenthèses car elles sont facultatives, le elif n'étant nécessaire que si on souhaite distinguer trois cas ou plus. D'ailleurs, le else est lui aussi facultatif ! Si on ne le met pas, on ne fera tout simplement rien dans le cas où la condition derrière le if n'est pas vérifiée. Dernière remarque : les deux points en fin de ligne sont indispensables si on ne veut pas déclencher une erreur de syntaxe.

À quoi peut ressembler la condition apparaissant derrière le if ? Ce sera très souvent une inégalité vérifiée par une variable (du style $\Delta > 0$), ou au contraire une égalité, qui sera dans ce cas indiquée à l'aide d'un double égal (le simple égal étant réservé à l'affectation), par exemple if a==0. Si on

veut au contraire que a soit différent de 0, on le notera a !=0. Plus généralement, une condition est en fait une variable de type booléen, c'est-à-dire quelque chose qui ne peut être que vrai ou faux.

Exemple : L'instruction suivante affiche le plus grand des deux nombres a et b :

```
> if a>b :
>         print a
> else :
>         print b
```

2.2 Instructions répétitives

Le principe des instructions répétitives est de condenser en une seule instruction une série de calculs similaires. Il s'agit en gros de faire la même chose qu'en mathématiques lorsqu'on remplace un calcul de somme par le symbole $\sum$: en Python on n'aura sûrement pas envie de taper cinquante lignes de code pour effectuer cinquante additions, et on ne peut évidemment pas mettre des « petits points » au milieu. Une instruction imitant le principe de la somme mathématique (une variable muette appelée par exemple i qui varie dans un intervalle de valeurs entières, et un calcul dépendant éventuellement de la valeur de i) sera donc fort utile. La plupart des langages de programmation utilisent une sytaxe qui mime vraiment celle de la somme mathématique (en général quelque chose du genre *for i from 1 to n* pour calculer une somme avec i variant de 1 à n par exemple). Python s'éloigne un peu de ce modèle traditionnel avec une syntaxe qui laisse beaucoup plus de liberté dans la manipulation de l'instruction.

Syntaxe des instructions répétitives de type **for** en Python :

```
> for variable in liste :
>         instructions
```

On peut difficilement imaginer plus simple ! Comme dans le cas des instructions conditionnelles, l'indentation est un élément de syntaxe, toutes les instructions indentées par rapport à la première ligne étant répétées à chaque étape de la boucle. La variable apparaissant dans l'instruction est une variable muette qui, comme toujours en Python, n'a pas besoin d'être déclarée auparavant. Plus important, ce qui se trouve après le *in* peut être une liste quelconque. Nous définirons plus précisément ces objets dans une section ultérieure, sachez simplement qu'ils peuvent contenir une collection de variables à peu près quelconques. Au sein d'une boucle *for*, nous emploierons le plus souvent une liste d'entiers (cf définition ci-dessous), mais on peut très bien imaginer par exemple une boucle *for* où la variable parcourt une liste de noms, ou de mots quelconques.

Définition 3. L'instruction **range(debut,fin,pas)** désigne en Python une liste de nombres dont le premier élément vaut **debut** et s'arrêtant avant d'atteindre le nombre **fin** (qui n'est donc pas inclus dans la liste), avec un écart entre deux valeurs successives égal à **pas**.

Exemples : L'instruction **range(2,10,2)** va créer la liste d'entiers [2,4,6,8]. La valeur du pas est facultative (par défaut, il est égal à 1), ainsi que celle de la valeur initiale (par défaut, la liste commence à 0). Ainsi, l'instruction **range(n)** renverra la liste des entiers compris entre 0 et $n-1$, ou si vous préférez une liste de n entiers numérotée à partir de 0.

Exercice 1 : Écrire une boucle calculant la somme des entiers de 1 à 100.

```
> a=0
> for i in range(1,101) :
>         a=a+i
> print a
```

Tous les calculs de sommes ou de produits reprendront cette structure. Attention à mettre les bonnes valeurs dans le *range* pour que i varie effectivement entre 1 et 100.

Exercice 2 : Écrire une boucle permettant de calculer le terme d'indice n de la suite (u_n) définie par $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \dfrac{1}{u_n} + \dfrac{u_n}{2}$.

```
> u=1
> for i in range(n) :
>         u=1/u+u/2
> print u
```

Une seule variable suffit ici à représenter les termes de la suite, puisqu'il suffit de remplacer à chaque étape de l'algorithme la valeur de u par la nouvelle valeur calculée (on n'aura de toute façon plus besoin des valeurs précédentes).

Exercice 3 : Écrire un boucle permettant de compter le nombre de fois où la lettre **e** apparait dans un texte.

```
> a=1
> for i in range(len(texte)) :
>         if texte[i]=='e' :
>                 a=a+1
> print a
```

Rien ne nous interdit bien évidemment de placer une instruction conditionnelle dans notre boucle. Notez ici l'utilisation de la fonction *len* sur les listes que vous avez croisée en TP (et qui permet de déterminer le nombre de caractères d'un texte). La variable appelée texte doit ici être dy type *str* pour que le programme fonctionne. Rappelons que les caractères de la chaîne sont numérotés de 0 à len(texte)-1, ce qui est cohérent avec les conventions utilisées pour la commande *range*. Notons enfin qu'il existe déjà une fonction toute faite en Python effectuant ce travail, mais notre but étant d'apprendre à programmer, nous nous contenterons pendant encore un moment de réinventer la roue !

Les boucles *for* ne sont pas les seuls moyens de créer des instructions répétitives, il existe également (comme dans tous les langages de programmation) une instruction *while* qui permet de répéter des calculs sans imposer le nombre d'étapes dans la boucle, mais en donnant au contraire une condition qui permet à Python de savoir quand arrêter les calculs.

Syntaxe des instructions répétitives de type **while** en Python :

```
> while condition :
>         instructions
```

Encore une syntaxe d'une sobriété remarquable. Les instructions de la boucle seront répétées tant que la condition indiquée sur la première ligne reste vraie. Il faut donc faire très attention lorsqu'on écrit une boucle de ce type de donner une condition qui ne va pas rester vraie tout le temps, sous peine de faire planter notre pauvre Python. En particulier, la condition doit dépendre d'une valeur qui évolue à l'intérieur de la boucle, sinon on testera toujours la même chose. Autre remarque importante : contrairement aux boucles *for* où la variable parcourt « toute seule » les différents éléments de la liste, il n'y a rien dans une boucle *while* qui permette de compter le nombre d'étapes effectuée. Si on souhaite le faire, il faudra donc incrémenter (ajouter une unité) une variable à l'intérieur de la boucle.

Exercice : Que fait le petit programme suivant ?

```
> n=0
> p=1
> while p < 10**12 :
>         n=n+1
>         p=p*n
> print n
```

On est en fait en train de calculer la valeur du plus petit entier pour lequel $n! \geqslant 10^{12}$. En effet, la variable n est incrémentée à chaque passage dans la boucle, et p est toujours égal à $n!$ (on le multiplie successivement par 1, puis 2 etc). On s'arrête lorsque p dépasse 10^{12}, d'où l'explication.

Exercice 2 : Écrire un programme qui permet de jouer au jeu suivant : l'ordinateur tire un nombre au hasard (l'instruction correspondante est donnée) entre 1 et 100, puis le joueur essaie de deviner le

nombre. À chaque essai, le programme doit préciser au joueur s'il a fait une proposition trop haute ou trop basse. Quand le joueur a trouvé le nombre, le programme doit afficher le nombre d'essais nécessaires au joueur pour gagner.

```
> from random import *
> r=randint(1,100)
> a=-1
> n=0
> while a!=r :
>         a=input('Choisissez un entier entre 1 et 100')
>         if a>r :
>                 print('Trop haut !')
>         elif a<r :
>                 print('Trop bas !')
>         n=n+1
> print('Gagne en '+str(n)+' essais')
```

La variable n sert simplement à compter le nombre d'essais (elle est incrémentée à chaque passage dans la boucle), la variable a représente le nombre choisi par le joueur (initialisé à -1 pour pouvoir faire le premier test en début de boucle while, qui sera évidemment toujours faux), et la variable r le nombre tiré par l'ordinateur (il est indispensable de l'initialiser en-dehors de la boucle while, sinon la machine va tirer un nouveau nombre aléatoire à chaque essai !). remarquez que, dans le cas où $a = r$, on ne fait rien à l'intérieur de la boucle. On va de toute façon en sortir immédiatement pour afficher le message final, qui est situé à l'extérieur de la boucle par commodité. Le *str(n)* de la denrière ligne permet simplement d'afficher ce message de façon élégante, c'est un point de détail (la commande transforme en l'occurence le nombre n en chaîne de caractère constituée des caractères affichant ce même nombre).

3 Fonctions et modules

Dans le dernier exemple de la section précédente, nous avons utilisé pour notre programme une fonctions préprogrammée en Python, *randint*, que nous avons été « chercher » au sein du **module random**. En fait, nous utilisons des fonctions Python presque tout le temps, sans forcément le savoir. Ainsi, le simple *print* qui apparait dans presque tous nos programmes est une fonction. Ces fonctions, qu'elles soient préexistantes dans le langage ou non, constituent un aspect essentiel du langage.

Définition 4. Une **fonction** en Python est un objet dépendant d'un ou plusieurs paramètres (ou même d'aucun) et ressortant une valeur (numérique ou non).

3.1 Définition de fonctions en Python

Même si Python dipose déjà de très nombreuses fonctions (et pas seulement des fonctions mathématiques), comme nous le verrons en partie dans le paragraphe suivant, il est souvent intéressant et utile d'écrire soi-même de nouvelles fonctions. La façon la plus simple de le faire est la suivante :

Syntaxe des définitions de fonctions en Python :

```
> def fonction(par1,par2,...) :
>         instructions
>         return valeur
```

Le mot-clé *def* sera comme d'habitude reconnu par le langage comme point de départ d'une définition de fonction. On donne ensuite le nom de la fonction, et ses paramètres, c'est-à-dire les variables nécessaires pour définir les valeurs prises par la fonction. Ces paramètres seront donc des variables qui sont définies au moment de la définition de la fonction (il est inutile de les définir par ailleurs). Ainsi, si on écrit par exemple une fonction calculant les factorielles, elle prendra comme paramètre

un nombre entier n. On a le droit de donner autant de paramètres qu'on le souhaite à l'intérieur d'une fonction, y compris d'ailleurs zéro. Autre spécificité, la dernière ligne du programme qui sera très souvent introduite par un *return* et qui sert simplement à afficher la valeur prise par la fonction. On peut toutefois, si on le souhaite, mettre des commandes *print* (ainsi que toute autre instruction) à l'intérieur de la définition de la fonction.

Exemple : La fonction suivante calcule la factorielle d'un entier n :

```
> def factorielle(n) :
>         a=1
>         for i in range(1,n+1) :
>                 a=a*i
>         return a
```

On fera comme toujours très attention à l'indentation. Pour utiliser ensuite la fonction qu'on vient de définir, on l'appellera en utilisation la notation mathématique : par exemple, l'instruction a=factorielle(5) affectera à la variable a la valeur 120, calculée à l'aide de notre fonction. Profitons de cet exemple pour faire une petite remarque sur le statut assez particulier des variables apparaissant dans les définitions de fonctions. Ici, la variable a est définie à l'intérieur de notre déclaration, et pour cette vue par Python comme une variable **locale**, c'est-à-dire une variable n'ayant d'existence qu'à l'intérieur de cette déclaration. Autrement dit, si cous demandez à Python ce que vaut la variable a après avoir lancé le calcul de **factorielle(10)**, il vous répondra que la variable n'existe pas, elle a déjà été supprimée. Pire, s'il existait, avant le calcul de factorielle(10), une variable également appelée a mais définie à l'extérieur de la fonction (ce qu'on appelle une variable **globale**), elle gardera après l'exécution de la fonction la valeur qu'elle avait auparavant. Autrement dit, il n'y a pas de conflit entre une variable globale et une variable locale portant le même nom. Il est toutefois extrêmement déconseillé d'utiliser des variables globales pour une utilisation locale, pour les soucis évidents de lisibilité.

Il existe une autre façon très légère de définir les fonctions en Python, surtout utilisée pour des définitions de fonction très simples à la volée, car elle est beaucoup plus limitée que ce qu'on a vu plus haut. Il s'agit de ce qu'on appelle les fonctions lambda :

Syntaxe de définition des fonctions lambda en Python :
> fonction = lambda x : valeur

Le mot **lambda** est un mot-clé du langage. La variable x peut être remplacée par toute autre variable, et on donne directement la valeur prise par la fonction (en gros, cela correspond à la notation mathématique $f : x \mapsto f(x)$). On économise ainsi un peu de place, mais les fonctions lambda sont restreintes par le fait qu'elles doivent pouvoir être définies sur une seule ligne, donc par une formule explicite simple. Par exemple, on peut définie en Python la fonction carré par l'instruction **carre = lambda x : x*x**.

Une autre possibilité du langage Python, que vous explorerez plus en détail l'an prochain, est la possibilité d'écrire des fonctions récursives, c'est-à-dire des fonctions qui apparaissent à l'interieur de leur propre définition. Un exemple sera plus parlant qu'un long discours, reprenons le cas du calcul de factorielles :

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

Le principe est de définir les factorielles comme une suite récurrente : $0! = 1$ et $\forall n \in \mathbb{N}$, $n! = n \times (n-1)!$. En pratique, comment se comporte le programme si on lui demande par exemple de calculer **recfact(2)** ? Il se rend compte que $2 \neq 0$, et essaie donc de calculer $2 \times recfact(1)$. Pour cela, il reprend la définition de la fonction, mais avec maintenant $n = 1$. De même, il se ramène au

calcul de recfact(0), qui est explicitement défini comme étant égal à 1, et remonte le calcul. Il est essentiel lorsqu'on écrit une fonction récursive de bien initialiser, c'est-à-dire donner (au moins) une valeur prise par la fonction, à partir de laquelle on calculera les autres. Il faut par ailleurs s'assurer que le calcul finira toujours par se ramener à celui de la valeur explicitement définie. Nous verrons en TP que la récursivité permet souvent d'écrire des programmes plus légers et élégants, mais aussi qu'elle n'est pas adaptée dans toutes les situations.

3.2 Modules

Un module en Python n'est rien d'autre qu'un gros stock de fonctions déjà programmées, en général reliées par un domaine d'application commun. Ainsi, nous avons déjà utilisé plus haut dans ce cours une fonction issue du module random, qui est constituée de nombreuses fonctions ayant toutes un rapport à l'aléatoire. Les fonctions contenues dans un module ne sont pas utilisables par défaut sous Python (il existe des centaines de modules et des milliers de fonctions, et Python en développements constant, leur nombre augmente presque tous les jours), il faut d'abord les importer, ce qui peut se faire de plusieurs façons.

> **import** random (**as** alias)

permet d'importer le module *random* (la précision entre parenthèses est facultative et permet de renommer le module en l'appelant alias plutôt que random, si on trouve son nom vraiment trop pourri). Par contre, la commande n'importe pas les noms des fonctions du module, qui devront donc être précédées du nom du module pour pouvoir être utilisées. Autrement dit, **randint(1,100)** renverra une erreur, alors que **random.randint(1,100)** (ou **alias.randint(1,100)** si on a renommé le module) fera le tirage aléatoire souhaité. Comme la syntaxe module.fonction est un peu lourde, on aimerait bien s'en passer. Eh bien ça tombe bien, car on peut :

> **from** random **import** randint

importera la fonction randint du module random (ceci n'est bien sûr qu'un exemple), qui pourra alors être utilisée telle quelle, sans préciser de random. devant. Par contre, la commande n'importe qu'une seule fonction (ou plusieurs séparées par des virgules, si on le souhaite). Si on souhaite importer toutes les fonctions du module, on remplace simplement le *randint* par un * (dont la signification standard en informatique est en gros « tout ce qu'il y a dedans »).

Liste non exhaustive de modules utiles (et moins utiles) en Python.

- random : sert à faire des tirages de nombres aléatoires.
- mathplotlib : permet de tracer des courbes (utilisé en TP).
- numpy et scipy : permettent de faire du calcul formel et de l'analyse numérique en Python (nous reviendrons plus longuement sur ces énormes modules, qui contiennent chacun des centaines de fonctions, un peu plus tard dans l'année).
- turtle : permet de faire des petits dessins grotesques. Il nous servira d'illustration très bientôt (mais en pratique, il ne sert à rien, contrairement aux précédents).

Pour terminer ce (rapide) aperçu sur les modules, nous allons donner une petite liste de fonctions disponibles dans les modules *random* et *turtle*, ainsi que quelques programmes faisant intervenir les dessins réalisés à l'aide de *turtle*.

Liste non exhaustive de fonctions du module *random*.
- **randint(a,b)** renvoie un entier aléatoire compris entre a et b (tous deux inclus).
- **random()** (pas de paramètre pour cette fonction) renvoie un réel aléatoire compris entre 0 et 1.
- **randrange(debut,fin,pas)** renvoie un nombre aléatoire appartenant à range(debut,fin,pas).
- **choice(liste)** choisit un élément aléatoire dans une liste donnée.
- **shuffle(liste)** replace les éléments de la liste dans un ordre aléatoire.
- **sample(liste,k)** choisit k éléments aléatoires (deux à deux distincts) dans la liste donnée.
- **gauss(m,sigma)** choisit un nombre aléatoire en suivant la loi de probabilité normale d'espérance m et d'écart-type sigma.

Il existe quantité d'autres fonctions dans le module, permettant notamment, à l'instar de la fonction *gauss*, de tirer un nombre suivant une loi donnée, pour toutes les lois de probabilités classiques. Nous reviendrons peut-être plus en détail sur ce module en fin d'année.

Un peu d'amusement pour terminer cette section, avec une illustration des modules, mais également un peu de programmation de fonctions et d'écritures de boucles. Nous allons pour cela travailler avec le module *turtle*, dont l'existence en Python est une réminiscence d'un projet pédagogique des années 80, visant à développer les capacités cognitives d'un enfant en le faisant notamment programmer les déplacements d'une tortue pour faire des dessins géométriques sur un écran d'ordinateur. Passons rapidement sur les ambitions initiales du projet, ce concept de tortue a été repris dans de nombreux langages de programmation, dont Python. Le principe est simple : un curseur est initialement placé au centre de l'écran (c'est la fameuse « tortue »), et on peut le déplacer à l'aide d'instructions simples. Le but est d'arriver à créer un motif donné en écrivant un petit programme. Donnons d'abord la liste des principales fonctions du module :

Liste non exhaustive de fonctions du module *turtle*.
- **forward(k)** avance le curseur de k pixels.
- **backward(k)** recule le curseu de k pixels.
- **position()** donne la position du curseur (dans le repère centré à l'origine, d'unité 1 pixel sur chaque axe).
- **right(theta)** fait pivoter le curseur dans le sens des aiguilles d'une montre d'un angle de theta degrés (on peut régler en radians si on y tient vraiment). Initialement, le curseur est pointé vers la droite.
- **left(theta)** fait pivoter le curseur dans le sens direct d'un angle de theta degrés.
- **heading()** indique la direction dans laquelle pointe le curseur (0 degré correspondant à la direction initiale, donc vers la droite).
- **goto(x,y)** fait déplacer le curseur en ligne droite vers le point de coordonnées (x,y) (en traçant le trait correspondant, sauf si on a auparavant levé le crayon).
- **up()** lève le crayon (tant qu'on ne le baisse pas, les déplacements suivants ne traceront plus rien à l'écran).
- **down()** baisse le crayon.
- **reset()** efface tous les dessins et remet le curseur à sa position initiale.
- **circle(r)** trace un cercle de rayon r dont le centre est situé à r pixels à gauche de la tortue (réflchissez-y, c'est le plus logique pour tracer un cercle dans la continuité des tracés précédents).
- **color(couleur)** permet de changer la couleur des tracés (les couleurs disponibles sont *red*, *blue*, etc).

Exercice 1 : Écrire une fonction **carre(n)** traçant un carré de côté n, puis (en réutilisant la fonction précédente) une deuxième fonction **suitecarres(n,k)** traçant un alignement de k carrés de côté n,

séparés de n pixels à chaque fois.

```
> def carre(n) :
>         for i in range(4) :
>                 forward(n)
>                 right(90)
>
> def suitecarres(n,k) :
>         for i in range(k) :
>                 carre(n)
>                 up()
>                 forward(2*n)
>                 down()
```

Exercice 2 : Tracer une étoile à cinq branches (il s'agit plus d'un exercice de mathématiques pour calculer l'angle dont il faut tourner après chaque trait).

```
> def etoile(n) :
>         for i in range(5) :
>                 forward(n)
>                 right(144)
```

Je vous mettrais volontiers une petite illustration, mais comme au lycée, la fenêtre *turtle* plante systématiquement chez moi en fin de tracé, donc tant pis. Le bon côté, c'est que ça me dissuade de continuer à perdre du temps à faire des tracés de figures géométriques débiles au lieu de taper la suite du cours.

4 Listes

Cette partie du cours est intégralement à l'étude d'un type de données bien particulier en Python : les listes. Pourquoi donc prodiguer un tel intérêt à ce type plutôt qu'à autre ? Car il est très intéressant à double titre : il s'agit du principal type composé utilisé en Python (c'est-à-dire qu'une liste ne contient pas simplement une valeur, mais plusieurs éléments qui sont eux-mêmes des variables, ce qui induit un comportemant et une manipulation plus délicats que pour les types simples croisés jusqu'ici) ; et c'est la porte d'entrée vers une composante très intéressante de Python, celle de la programmation orientée objet (nous ne ferons qu'entrouvrir cette porte cette année).

4.1 Définitions

Définition 5. Une **liste** en Python est une collection de plusieurs variables pouvant être de types différents. Les listes sont délimités par des crochets, et les éléments à l'intérieur de ces crochets séparés par des virgules.

Exemples : L'instruction l=[1,2,3] créée une liste Python constitué de trois variables de type *int*. L'instruction l=[1,3.14,'miaou'] créée une liste Python constituée de trois objets de types différents, ce qui ne pose absolument aucun problème (du moins tant qu'on n'essaye pas de faire des choses incompatibles avec ce qui se trouve à l'intérieur de la liste ; trier la liste donnée dans le deuxième exemple sera peut-être problématique, difficile de savoir s'il faut mettre 'miaou' avant ou après 3.14). On peut également, pourquoi pas, créer des listes dont les éléments sont eux-même des listes (ce qui permettra de donner une représentation aisée des matrices), ou même des listes de listes si on veut se torturer le cerveau.

On peut avoir très simplement accéder aux éléments de la liste (qui, rappelons-le, sont eux-même des variables) à l'aide de l'instruction l[i], qui représente l'élément numéro i de la liste. Attention, comme toujours en Python, les listes sont numérotées à partir de 0. Ces éléments sont par ailleurs modifiables indépendamment les uns des autres (ce ne sera pas le cas, par exemple, des différents caractères d'une chaîne de caractères). Ainsi, l'instruction **l[2]=5** va remplacer la valeur du troisième

terme de la liste (le numéro 2) par 5. Dans le cas de liste de listes, la commande l[i][j] permettra naturellement d'accéder à l'élément numéro j de la liste constituant l'élément numéro i de la liste l (ça va, vous suivez ?).

Remarque 1. Si on tente d'accéder à l[i] pour une valeur de i supérieure au numéro du dernier élément de la liste, on risque de se fâcher avec Python. Par contre, la commande l[-1] est parfaitement comprise par le langage et renvoie le dernier élément de la liste (ce qui est bien pratique quand on veut y accéder sans connaitre la longueur de la liste). De même, l[-2] renverra l'avant-dernier, l[-3] l'antépénultième etc (oui, c'était juste pour caser un mot compliqué).

Remarque 2. Attention à un piège vicieux lié à la nature de type composé des listes. Si vous venez de créer une liste par la commande l=[1,2,3] et que vous effectuez ensuite l'instruction m=l, vous vous attendez logiquement à ce que cela crée une deuxième liste m contenant les mêmes éléments que la liste l. En un sens, c'est le cas, mais ce que vous n'aurez peut-être pas anticipé, c'est que les deux listes ne sont pas du tout indépendantes l'une de l'autre. Si vous décidez un peu plus tard, par exemple, de modifier un élément de m en tapant m[1]=12, vous allez en même temps modifier l'élémént numéro 1 de la liste l (ce qui rend donc la copie de la liste complètement inutile puisque tout se passe comme si vous n'aviez qu'une seule liste avec deux noms différents). Mais pourquoi donc tant de haine ? Cela est lié à la nature des listes, et à la façon dont elles sont gérées par la machine. Une variable, nous l'avons dit il y a un moment déjà est une sorte de case dans la mémoire de l'ordinateur, avec une valeur dedans. Une liste est donc, logiquement, constituée de plusieurs cases (avec plusieurs valeurs). Mais en fait, c'est un peu plus subtil que ça : la liste proprement dite n'est qu'une liste d'adresses indiquant où se trouvent en mémoire les variables constituant la liste. Lorsqu'on écrit l'instruction m=l, on recopie dans m la liste des adresses contenue dans l, mais les valeurs des variables de la liste, elles, ne sont PAS recopiées à un autre endroit de la mémoire. Quand on modifie l'une d'elles, les deux listes ayant les mêmes adresses, elles vont pointer vers la même valeur modifiée. Mais alors, comment fait-on pour « vraiment » recopier une liste ? Eh bien, autrement. On verra ça un tout petit peu plus loin.

4.2 Opérations sur les listes

Par opérations, nous entendons ici de vraies opérations au sens mathématique du terme. Il peut sembler bizarre de vouloir additionner deux listes, par exemple, mais pourtant c'est tout à fait possible en Python.

Définition 6. Le symbole + définit sur les listes Python l'opération de concaténation. La commande **l*k**, où l est une liste et k un entier naturel, créée une nouvelle constituée de la concaténation de k exemplaires de la liste l.

Exemples : Si l=[1,2] et m=[3,4,5], on aura par exemple l+m=[1,2,3,4,5] et l*3=[1,2,1,2,1,2]. Cette opération de multiplication par un entier permet de définir très facilement de grandes listes contenant une même valeur en multiples exemplaires. Ainsi, l'instruction l=[0]*50 créée une liste contenant 50 zéros.

4.3 Fonctions et méthodes

Il existe sur les listes Python un certain nombre de fonctions préprogrammées (vous avez par exemple utilisé en TP la fonction *len* qui renvoie le nombre d'éléments de la liste), mais aussi un type de fonctions un peu particulières appelées méthodes. Ces drôles de bestioles sont une concession faite par Python à ce qu'on appelle la programmation orientée objet, qui constitue une façon un peu différente de concevoir la programmation. En gros, dans la vision traditionnelle, les objets de bases sont les variables (avec leur types), et on écrit ensuite des fonctions, programmes et autres joyeusetés faisant intervenir ces variables (et adaptées, bien évidemment, à certains types de données). En programmation orientée objet, les éléments de bases sont ce qu'on appelle des objets, qui sont censés représenter des concepts ou même des entités physiques (il y a un peu de philosophie là-dessous), mais

surtout qui ont des caractéristiques et des capacités à interagir entre eux. La principale différence est, qu'au lieu de considérer les fonctions comme séparées des variables, en programmation orientée objet, une classe d'objets constitue plus ou moins l'équivalent du type classique, mais contient également toute une liste de fonctions, appelées **méthodes**, destinées à exhiber leurs caractéristiques et à en faciliter la manipulation. En gros, les fonctions font partie de l'objet. Les listes en Python sont définies dans cet esprit, et sont donc fournies avec tout un tas de méthodes directement applicables à l'aide de la drôle de syntaxe suivante, que nous avons déjà aperçue quand on parlait de modules il n'y a pas si longtemps :

Syntaxe d'utilisation des méthodes en Python :
> liste.methode(parametres)

Pour reprendre un exemple vu en TP, la méthode *append*, qui prend comme paramètre un élément z à ajouter à la liste l, sera utilisée via une instruction du type l.append(z) (en gros, la liste n'est pas considérée comme un paramètre de la fonction, et placée avant le nom de la méthode).

Liste non exhaustive de méthodes sur les listes en Python.
- **append(z)** ajoute l'élément z en tant que dernier élément de la liste.
- **reverse()** retourne l'ordre des éléments dans la liste.
- **sort()** trie les éléments de la liste par ordre croissant (si c'est possible).
- **remove(z)** supprime de la liste l'élément z (il s'agit bien de la valeur de l'élément, on ne supprime pas l'élément numéro z). Si l'élément apparait plusieurs fois dans la liste, seule la première occurence sera supprimée. S'il n'apparait pas, bien entendu, il ne se passera rien.
- **index(z)** indixe la position de la liste de l'élément z. Mêmes remarques que pour la méthode précédente.

Une petite remarque à propos de la méthode *remove* : il existe également une fonction (pas une méthode !) *del* permettant cette fois-ci de supprimer un élément de la liste en le repérant par son numéro : **del l[i]** supprimera l'élément numéro i de la liste l.

4.4 Tranchage

Oui, je sais, le terme français n'est pas beau, on peut utiliser le mot anglais *slicing* poru ne pas passer pour un plouc. Le principe est très simple : tout comme on peut accéder à un élément d'une liste via la commande l[i], on peut accéder à toute une partie de la liste via des commandes du style l[i :j]. Plus précisément, ce l[i :j] désigne la « tranche » de la liste commençant à l'élément numéro i (inclus) et s'arrêtant à l'élément numéro j exclus. Pourquoi ces distinctions subtiles ? Imaginez que ce ne sont pas les éléments qui sont numérotés dans la liste, mais les séparateurs (les crochets et les virgules). Ainsi , si l=[7,5,12,8,20] (les petits numéros en bas sont là pour vous aider à comprendre), on aura par exemple l[2,4] qui est égal à la liste [12,8]. On peut également omettre soit l'indice i soit l'indice j pour désigner la tranche de liste commençant au numéro i (et allant jusqu'au bout de la liste) ou terminant au numéro j (et partant du début de la liste). En reprenant la même liste, l[2 :] désignerait la liste [12,8,20] alors que l[:2] désignerait la liste [7,5]. On peut même désigner l'ensemble de la liste par l[:].

Remarque 3. Nous tenons là une solution curieuse mais efficace pour un problème évoqué plus haut, la copie de listes. Nous avions vu que l'instruction m=l possédait de gros défauts, une façon de les régler est de demander à la place **m=l[:]**. Cette fois, les éléments de la liste seront bien recopiés, et pas seulement les adresses.

Exemples : On peut effectuer sur les tranches de listes les mêmes opérations que sur les éléments. En reprenant la liste définie précédemment, l=[7,5,12,8,20], voici ce que produira l'application (non successive) d'une des instructions suivantes :

> l[2 :5]=[] supprime les trois derniers éléments de la liste (qui est désormais égale à [7,5])

> l[2 :4]=[3] remplace une tranche de deux éléments de la liste par un seul (la liste est désormais égale à [7,5,3,20])

> l[3 :]=[4,12] remplace la fin de liste par deux éléments (la liste est désormais égale à [7,5,12,4,12])

> l[2 :2]=[1] ajoute un élément à la liste en milieu de liste (la liste est désormais égale à [7,5,1,12,8,20])

4.5 Compléments

Nous verrons plus tard dans l'année (notamment quand vous en aurez entendu parler en cours de mathématiques) l'intérêt de pouvoir aussi travailler avec des matrices. Mathématiquement, une matrice est un tableau rectangulaire contenant des nombres dans chacune de leurs cases, habituellement désignées par les indices i (pour les lignes) et j (pour les colonnes). En Python, on pourra travailler tout simplement avec des listes ...de listes ! Chaque élément de notre liste sera lui même une liste (toutes de même longueur si on veut modéliser une matrice). Ainsi, on pourra créer une matrice par une instruction du type M=[[1,2],[3,4]]. Pour accéder au terme de la matrice se situant sur la ligne i et la colonne j, on utilisera simplement l'instriction M[i][j] (en faisant toujours attention au fait que les indices en Python commencent à 0).

Deux autres techniques de manipulation des listes peuvent être utiles pour alléger vos programmes Python. La première est la possibilité de décrire une liste « en compréhension », c'est-à-dire en précisant une propriété de ses éléments. Ainsi, Pyhton accepte tout à fait une définition de liste de la forme l=[i for i in range(100)] (ce qui est une façon stupide de recréer la liste range(100)), ou même (plus intéressant) l=[i for i in range (100) if i%2==0], qui va créer une liste constituée de tous les nombres pairs inférieurs à 100 (là aussi, un range avec les bons arguments suffit, mais vous avez compris le principe).

Dernière possibilité : le « mapping » d'une fonction sur une liste. Supposons qu'on dispose d'une fonction f et d'une liste l et qu'on souhaite calculer les images par f de tous les éléments de la liste l. Classiquement, on est obligés de parcourir la liste à l'aide d'une boucle for. On peut en fait l'éviter en utilisant la commande **map(f,l)** qui va créer une liste contenant les valeurs f(l[i]).

4.6 Algorithmes classiques.

Pour terminer cette longue partie de cours consacrée aux listes, voici deux ou trois algorithmes classiques en Python faisant intervenir des listes (au moins pour les deux premiers), qu'il est bon de connaître et de savoir réécrire sans hésitation si besoin.

Exemple : recherche du maximum dans une liste non triée. On dispose d'une liste l et on souhaite en extraire l'élément maximal (on suppose bien entendu que les valeurs de la liste sont toutes de types numériques). La méthode la plus simple pour cela est tout bêtement de parcourir la liste en gardant en mémoire le plus grand élément obtenu depuis le début du parcours, et en le remplaçant par le nouvel élément atteint si celui-ci est plus grand :

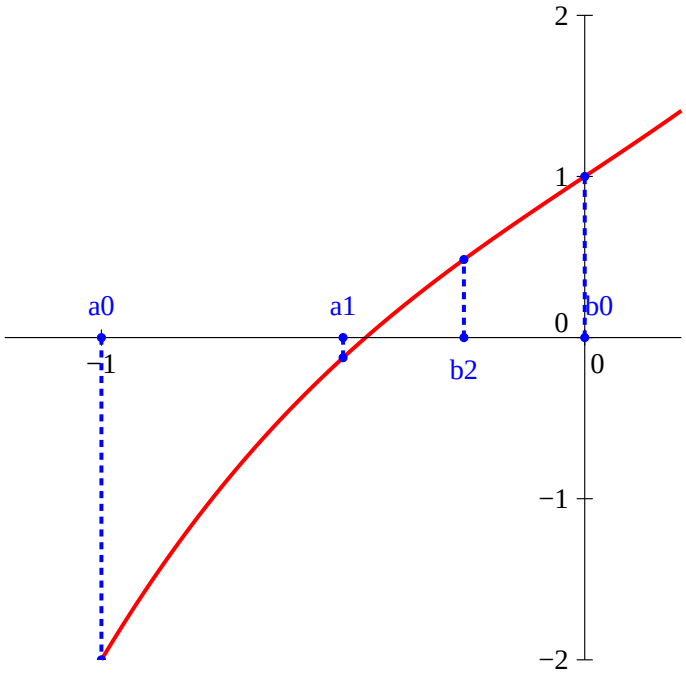
```
> def maximum(l) :
>         a=l[0]
>         for i in range(1,len(l)) :
>                 if l[i]>a :
>                 a=l[i]
>         return a
```

Recherche d'un élément dans une liste triée : On souhaite désormais rechercher la position d'un élément x dans une liste déjà triée (par ordre croissant) l. On suppose pour simplifier que l'élément x appartient effectivement à la liste, et y apparait une seule fois. Une solution naïve consiste à effectuer un parcours de la liste, comme dans l'algorithme précédent, en s'arrêtant dès qu'on tombe sur x. Il est en fait beaucoup plus rapide en moyenne de procéder par **dichotomie** : on prend l'élément situé

au milieu de la liste, on le compare à x et on sait déjà dans quelle moitié de liste se trouve x. On recommence sur la moitié de liste en question, et ce jusqu'à avoir trouvé l'élément x.

```
> def recherche(l,x) :
>         i=0
>         j=len(l)-1
>         while l[i] !=x :
>                 k=(i+j)//2
>                 if l[k]>x :
>                         j=k
>                 else :
>                         i=k
>         return i
```

Une application classique de la dichotomie : une utilisation fréquente de la méthode de dichotomie est la détermination de valeurs approchées de racines de focntions continues. Soit f une fonction continue sur un segment $[a, b]$, telle que $f(a)f(b) < 0$ (ainsi, le théorème des valeurs intermédiaires nous assure l'existence d'au moins un réel α tel que $f(\alpha) = 0$). On souhaite déterminer une valeur approchée de α. Pour cela on peut procéder par dichotomie : on calcule $f\left(\frac{a+b}{2}\right)$, si la valeur est de même signe que $f(a)$, on sait que $\alpha \in \left[\frac{a+b}{2}, b\right]$, sinon $\alpha \in \left[a, \frac{a+b}{2}\right]$, et on recommence sur le nouvel intervalle. Puisque la largeur de l'intervalle de recherche est divisée par deux à chaque étapes, cela permet d'obtenir une valeur approchée à $\frac{b-a}{2^n}$ près au bout de n étapes (on verra beaucoup plus tard dans l'année d'autres méthodes de résolutions approchées d'équations qui peuvent être plus efficaces).



Un programme Python possible pour implémenter cet algorithme est le suivant :

```
> def dichotomie(f,a,b,n) :
>         debut=a
>         fin=b
>         for i in range(n) :
```

```
>               c=(debut+fin)/float(2)
>               if f(c)*f(debut) >0 :
>                       debut=c
>               else :
>                       fin=c
>       return debut
```

5 Chaînes de caractères

Cette partie de cours sera nettement plus courte que la précédente, car le type de données chaîne de caractères ressemble beaucoup aux listes. Qu'est-ce qu'une chaîne de caractères ? Tout simplement une suite de symboles non nécessairement numériques (et même s'ils le sont, on ne pourra faire directement des calculs avec) délimités par des apostrophes (simples ou doubles selon les besoins). Ainsi, la commande c='bonjour' définit une variable c de type string (chaîne de caractère) contenant un mot de sept caractères. Le seul avantage d'utiliser les doubles apostrophes au lieu des simples tient au fait qu'on peut vouloir définir une chaîne de caractères contenant elle-même un symbole apostrophe, par exemple c="L'informatique c'est génial !". Pour accéder aux différents caractères de la chaîne, on procède comme pour les listes à l'aide de la commande c[i]. Toutes les manipulations que l'on a pu voir sur les listes fonctionnent de la même façon sur les chaînes de caractères : manipulations d'indices, tranchage, insertion dans une boucle for, opérations + et *. Ainsi, la commande

```
> for i in c :
>       print(i)
```

affichera successivement tous les caractères de la liste c. Une seule grosse différence entre listes et chaînes en Python :

Remarque 4. Les chaînes de caractères ne sont pas modifiables en Python. Ainsi, une commande du style c[i]='z' ne modifiera pas le i-ème caractère de la chaîne c.

Remarque 5. Il existe quelques caractères spéciaux qu'on peut insérer dans les chaînes de caractères, notamment le caractère \n qui représente un retour à la ligne.

Comme pour les listes, on peut appliquer aux chaînes de caractères quantité de méthodes, donc voici une petite sélection :

Liste non exhaustive de fonctions et méthodes sur les chaînes de caractères en Python.

- **len(c)** renvoie le nombre de caractères de la chaîne c (c'est la seule fonction de la liste, les autres sont des méthodes).
- **index(mot)** recherche si l'argument mot (qui doit lui-même être une chaîne de caractères) est présent dans la chaîne de caractères, et donne la position du mot dans la chaîne.
- **find(mot)** fait la même chose que index, mais ne gère pas bien le cas où le mot n'apparait pas dans la chaîne.
- **count(mot)** compte le nombre d'apparitions du mot dans la chaîne de caractères.
- **replace(mot,autremot)** remplace toutes les occurrences de mot par autremot.
- **lower()** et **upper()** mettent respectivement tous les caractères en minuscules, ou en majuscules.
- **split()** découpe la chaîne en plusieurs chaînes en utilisant les espaces comme séparateurs (on peut mettre autre chose comme séparateur en le donnant tout simplement en argument de cette méthode).
- pour ceux qui aiment faire des choses soignées, on peut également formater des variables à l'intérieur d'une chaîne de caractères. Ainsi, la commande "J'ai eu {} au dernier devoir d'info".format(a) remplacera le {} par la valeur de la variable a, automatqiquement convertie en chaîne de caractères et insérée au milieu de la chaîne.

6 Gestion de fichiers

Jusqu'ici, toutes les variables que nous avons pu manipuler dans nos programmes Python, quel que soit leur type, étaient définies directement sous Python et donc stockées en mémoire par Python. Il peut être utile également de travailler avec Python sur des données contenues dans des fichiers texte sans les recopier intégralement. Pourquoi ? Tout simplement parce que le fichier texte sera trop gros, Python est un outil très performant de manipulation de gros fichiers de données. Ppourquoi seulement des fichiers texte ? Parce que c'est le plus simple à faire dans un premier temps. D'ailleurs, nous nous conterons d'aborder très brièvement ce vaste domaine. Supposons donc que nous ayons un fichier de données au format texte, que nous nommerons par exemple fichier.txt, stocké dans le même répertoire que Python. La première chose à faire est d'ouvrir le fichier avec Python, c'est-à-dire de se donner les moyens d'accéder aux données du fichier.

Les trois façons d'ouvrir un fichier texte en Python.
- **f=open('fichier.txt','r')** créée une variable f ouvrant le fichier texte en mode lecture (on ne pourra pas modifier le fichier, seulement récupérer les caractères qu'il contient).
- **f=open('fichier.txt','w')** créée une variable f et créée un fichier texte nommé fichier.txt dans lequel on pourra écrire, c'est-à-dire insérer des caractères. Attention, cette commande n'est à utiliser que sur un fichier texte vide (ou inexistant). Si jamais le fichier est déjà existant et contient des caractères, ils seront purement et simplement effacés.
- **f=open('fichier.txt','a')** créée une variable f ouvrant le fichier texte en mode ajout, c'est-à-dire qu'on pourra y insérer de nouveaux caractères, mais uniquement à la suite de deux déjà existants.

Tout cela est très restrictif, mais permet de bien maîtriser ce qu'on fait avec son fichier. Si on veut réellement modifier un fichier en profondeur, il faut en fait créer deux variables : une pour l'ancien fichier en mode lecture, et une deuxième pour le nouveau fichier en mode écriture. On gardera ainsi une copie de notre fichier initial. Notons que la variable f constitue une sorte de représentation du fichier texte au sein de Python, c'est sur cette variable qu'on va appliquer toutes les méthodes et autres fonctions permettant de manipuler les données du fichier (en fait, on ne manipule jamais le fichier directement).

Liste non exhaustive de méthodes sur les fichiers en Python.
- **close()** referme le fichier. Si vous ne programmez pas comme un pied, un fichier ouvert doit toujours être refermé à la fin de la manipulation.
- **write(chaine)** écrit la chaîne de caractères dans le fichier (ce qui suppose évidemment qu'il soit ouvert en mode écriture).
- **read()** lit le caractère suivant du fichier (on ne peut jamais revenir en arrière quand on lit un fichier).
- **read(k)** lit d'un seul coup les k caractères suivant (si on effectue une commande du type a=f.read(k), la variable a sera donc une chaîne de caractères).
- **readlines()** lit tout le fichier ligne par ligne. Cette méthode créée donc une liste dont chacun des éléments est une chaîne de caractères.

Exemple de programme manipulant un fichier : Le programme suivant va créer une copie du fichier tagada.txt, en ajoutant un z à la fin de chaque ligne (non, ne cherchez pas, ça n'a rigoureusement aucun intérêt).

```
> f=open('tagada.txt','r')
> g=open('tugudu.txt','w')
> for c in readlines(f) :
>         g.write('c'+'z'+\n)
> g.close()
> f.close()
```

7 Analyse de programmes

La dernière partie de ce chapitre est destinée à vous présenter les rudiments de l'analyse de programmes, c'est-à-dire des techniques permettant de déterminer, de façon tout à fait théorique, si un programme est « bon ». Le premier élément essentiel quand on analyse un programme est d'arriver à prouver qu'il va effectivement faire ce qu'on attend de lui, c'est ce qu'on appelle la preuve de la **correction** du programme. On s'intéressera ensuite à l'efficacité du programme, en cherchant à savoir si notre programme, en plus de donner le résultat correct, fait son travail mieux que d'autres programmes ayant le même but, c'est-à-dire en gros s'il effectue sa tâche plus rapidement. On va quelque peu renverser cet ordre de priorité ici, en commençant par aborder la question de l'efficacité avant de revenir sur la preuve de correction.

7.1 Complexité

La notion de complexité d'un programme informatique vise à mesurer l'efficacité de ce programme, dans deux domaines principaux :

- la complexité spatiale mesure la quantité de mémoire nécessaire à l'exécution de l'algorithme. Il est toujours préférable d'utiliser le moins de variables possibles pour diminuer cette complexité, mais on n'aura à notre niveau pas vraiment de problème ce de point de vue là.
- la complexité temporelle mesure le temps d'exécution de l'algorithme, ou plutôt une estimation de ce temps d'exécution en fonction de la taille des données manipulées. C'est de loin le type de complexité le plus important, celui que nous allons commencer à étudier ici.

Pour mesurer la complexité temporelle, on essaiera tout simplement de compter le nombre d'opérations effectuées par l'algorithme et de le comparer à la taille des données manipulées (le nombre d'éléments de la liste s'il s'agit d'un programme manipulant une liste, par exemple). On classera les algorithmes en différentes catégories selon le nombre d'opérations effectuées. En notant n la taille des données, on dira notamment qu'un algorithme est :

- linéaire si le nombre d'opérations est proportionnel à n (ainsi, avec des données dix fois plus volumineuses, l'algorithme mettra dix fois plus de temps).
- quadratique si le nombre d'opérations est proportionnel à n^2.
- polynomial si le nombre d'opérations est proportionnel à n^k pour un certain entier $k \geqslant 1$.
- exponentiel si le nombre d'opérations est proportionnel à α^n pour un certain réel α.

Ainsi, si un algoritme met une seconde pour trier les données d'un tableau contenant 10 éléments, et qu'on veut lui faire trier un tableau à 100 éléments, il mettra :

- 10 secondes s'il est linéaire.
- 100 secondes s'il est quadratique.
- 1 000 secondes (soit un peu plus d'un quart d'heure) s'il est cubique.
- 1.1^{90} secondes (environ une heure et demie) s'il est exponentiel de base 1.1.
- 2^{90} secondes (soit un peu plus de 39 milliards de millards d'années) s'il est exponentiel de base 2.

Les algorithmes exponentiels sont inutilisables en pratique, et les algorithmes qu'on compare pour des problèmes fréquents de programmation se situent souvent quelque part entre le linéaire et le quadratique. On parle d'ailleurs d'algorithmes sous-linéaires quand le temps d'exécution est proportionnel à $n\ln(n)$, ce qui constitue une grosse amélioration par rapport à un algorithme quadratique.

7.2 Invariants de boucles

Penchons-nous maintenant sur le problème de la correction des algorithmes. Il s'agit en fait de prouver deux choses : d'une part que le programme va toujours sortir un résultat (et ne va pas être pris dans une boucle qui ne termine jamais, par exemple), d'autre part que ce résultat correspond à ce qu'on attendait de lui (qui n'a jamais eu le malheur en tant que programmeur débutant de passer

par une étape de joie intense « C'est bon, ça compile ! » rapidement suivie d'un désespérant « mais ça ne fait pas du tout ce que je voulais » ?). Pour prouver ces deux choses simultanément, on dipose de l'outil technique suivant :

Définition 7. Un **invariant de boucle** pour un algorithme est une propriété qui reste vraie pendant toutes les phases de l'algorithme.

Plutôt que de longs discours, un bon exemple va permettre de clarifier cette notion, et surtout son utilité. Considérons le programme suivant, qui est censé calculer le quotient et le reste de la division euclidienne d'un entier a par un entier b :

```
> def eucl(a,b) :
>         r=a
>         q=0
>         z=b
>         while r>=z :
>                 r=r-z
>                 q=q+1
>         return (a,r)
```

Pour prouver que l'algorithme termine toujours, on peut utiliser le raisonnement suivant : à chaque étape dans la boucle, la valeut de la variable r diminue strictement. Comme il s'agit d'une variable entière, elle finira donc nécessairement par prendre une valeur inférieure à z, ce qui entrainera la fin de la boucle while et l'arrêt du programme. Essayons maintenant de prouver que notre programme calcule bien le quotient et le reste de la division en considérant l'invariant de boucle suivant : $a = z * q + r$. Pour prouver qu'il s'agit bien d'un invariant de boucle, on effectue une sorte de récurrence. Initialement (avant le premier passage dans la boucle), $q = 0$ et $r = a$, donc $a = z * q + r$ est bien vérifiée. Supposons désormais que cette propriété soit vérifiée avant un passage dans la boucle. Ce passage va modifier les valeurs de q et r pour les transformer en $q' = q + 1$ et $r' = r - z$. On peut alors calculer $z * q' + r' = z(q + 1) + r - z = zq + r = a$ par hypothèse. La propriété est donc encore vérifiée en sortie de boucle, et constitue bien un invariant de boucle. Il ne reste plus maintenant qu'à voir ce qui se passe quand on sort de la boucle : on sait qu'alors $a = b * q + r$, et de plus que $r < z$ (condition de sortie de boucle), ce qui prouve qu'on a effectué la division euclidienne de a par b (il faut quand même vérifier aussi que $r \geqslant 0$ en fin de boucle, ce qui est le cas puisque l'avant-dernière valeur prise par r est supérieure ou égale à z, et on diminue ensuite cette valeur de z).

Un deuxième et dernier exemple :

```
> def zoinx(a,n) :
>         x=a
>         p=n
>         z=1
>         while p>0 :
>                 if p%2 == 0 :
>                         x=x*x
>                         p=p/2
>                 else :
>                         z=z*x
>                         p=p-1
>         return z
```

La première étape ici est de comprendre ce qu'est censé faire ce programme. Une bonne façon de faire est de le teste avec des valeurs pas trop grandes des paramètres, en écrivant l'évolution étape

par étape des différentes variables. Avec un peu de chance, on arrivera même à trouver un invariant de boucle à partir de ça. Prenons par exemple $a = 3$ et $n = 5$:

x	3	3	9	81	81
p	5	4	2	1	0
z	1	3	3	3	243

La valeur sortie par le programme est donc 243, soit 3^5. Le programme termine certainement car p prend des valeurs entières strictement décroissantes, donc finira par être négatif et nul. Pour prouver qu'il calcule toujours la valeur a^n, on peut prendre comme invariant de boucle la propriété $x^p \times z = a^n$. Initialement, $x = a$, $p = n$ et $z = 1$, donc $x^p \times z = a^n \times 1$, ça marche. Supposons la propriété vraie avant un passage dans la boucle. On a alors deux possibilités : soit $x' = x^2$, $p' = p/2$ et $z' = z$, ce qui donne $x'^{p'} \times z' = (x^2)^{\frac{p}{2}} \times z = x^p \times z$; soit $x' = x$, $p' = p - 1$ et $z' = z * x$, ce qui donne $x'^{p'} \times z' = x^{p-1} \times z \times x = x^p \times z$. Dans les deux cas, la valeur de notre invariant n'a effectivement pas changé. Plaçons-nous maintenant en fin de boucle : on sait qu'à ce moment-là $p = 0$ (il ne peut pas devenir négatif), donc $x^0 \times z = a^n$, ce qui implqiue $z = a^n$. Le programme a bien calculé la valeur de a^n.