Chapitre 1

Introduction

On supposera le lecteur familier avec la notion de langage informatique, sans forcément être un expert en algorithmique.

I Pourquoi Python ?

1) Continuité par rapport au collège

Scratch est déjà un langage orienté objet : on y définit des procédures attachés à des lutins et certaines variables peuvent être locales (que pour le lutin) ou partagées.
En Python, on retrouve la gestion de module, où, par exemple, un groupe d'élèves peut travailler de façon indépendante et mettre en commun le travail sans risque que des fonctions nouvellement définies effacent d'autres fonctions.

Scratch utilise des procédures qui peuvent être récursives, mais on se retrouve vite gêné lorsqu'on veut écrire des fonctions récursives car les procédures ne renvoient pas de valeurs. Python permet évidemment de faire de la récursion de façon bien plus lisible.

Scratch utilise des variables alphanumériques ainsi que des tableaux. Quelques fois, le type de la variable change suivant l'utilisation. Par exemple, si on répond à une question le nombre 1 suivit de 18 zéros, il est considéré comme une suite de caractères mais si ensuite on demande de calculer $(reponse+1)-1$ on trouve 0 car la réponse est transformé en nombre et scratch ne gère pas des nombres aussi grands. En revanche, Python peut gérer des entiers de taille quelconque, et, comme Scratch, les opérateurs, comme l'addition, peuvent accepter des variables de types différents. De plus, Python ne se limite pas aux tableaux à une dimension. Pour les tableaux, qui sont un type de variable assez complexe, Scratch est une bonne façon de les voir pour la première fois. Leur édition et leur gestion est simple et visuelle.

2) Atouts et inconvénients du langage

a) Atouts

- Concision et clarté naturelle du langage.
- Des librairies pour tout faire. On peut « s'appuyer sur les épaules des géants ».
- Utilisé dans le monde de l'entreprise, des sciences, de la formation ...
- La calculatrice Numworks fonctionne en Python.
- Python possède en natif les types complexes et ensembles.
- Python possède tous les outils modernes des langages de programmation : décorateurs, itérateurs, surcharge d'opérateurs, gestion des erreurs, listes définies par compréhension, récursivité ...

b) Inconvénients

— Lenteur
— Langage non typé.
— Encore une nouvelle syntaxe à apprendre. Cas de la boucle for et du range.
— Certains modules ont une syntaxe compliquée, particulièrement les modules numériques et graphiques. Les programmes écrits avec cette syntaxe deviennent peu lisibles pour les non spécialistes.

II Exemples

1) Programmation fonctionnelle

Dans Python les fonctions sont des objets comme les autres. On peut écrire des fonctions prenant comme argument une fonction, ce qui est très utile. Quelques exemples :

1. Calcul du n-ième terme d'une suite récurrente.

```
def terme(f, u0, n):
''' renvoie u_n ou u est definie par recurrence'''
    assert n >= 0
    u = u0
    for _ in range(n):
        u = f(u)
    return u
```

2. Simulation d'une expérience aléatoire.
Considérons le problème suivant : on a 3 pièces dont deux normales et une truquée avec deux côtés pile. On choisit une pièce au hasard dans la poche et on la jette en l'air pour voir si le côté pile ou face apparaît.
On peut commencer à écrire une fonction qui simule une pièce normale :

```
from random import *
def piece_normale():
''' renvoie pile ou face de facon equiprobable'''
        if random() > 0.5:
                return 'pile'
        return 'face'
```

Maintenant une fonction qui simule une pièce truquée :

```
def piece_truquee():
''' renvoie pile'''
    return 'pile'
```

On peut maintenant fabriquer notre poche :
`poche = [piece_truquee, piece_normale, piece_normale]`.
L'expérience consiste à faire les deux actions suivantes :

(a) on prend une pièce au hasard : `piece = poche[randrange(3)]`

(b) On lance la pièce : `piece()`

3. Composition de fonctions. Voir chapitre 4, dans les fonctionnalités avancées.
4. Bataille d'algorithmes. Voir le chapitre « Projets ».

2) Exemple de compréhension : crible d'Eratosthène

En utilisant une syntaxe très proche du langage mathématique, on peut fabriquer un ensemble contenant tous les nombres premiers jusqu'à 100 (exclu) avec cette commande : `set(range(2, 100)) - {j for i in range(2, 100) for j in range(i*i, 100, i)}`, ce qui en mathématiques correspond à $\{2;3;\ldots;99\} \setminus \cup_{i=2}^{99}\{i^2; i^2+i; i^2+2i;\ldots\}$.

3) Exemple d'utilisation de Jupyter : Jake Vanderplas, Python Data Science Handbook

Jake Vanderplas a eu la bonne idée de mettre son excellent livre en ligne sous la forme d'un document Jupyter. On y trouve une description détaillé des modules NumPy (calcul scientifique), Pandas (traiter des données), Matplotlib (graphiques), Scikit-learn (apprentissage par la machine).

III Installation

Sur le site du jury du CAPES on trouve des liens vers différents logiciels pour les installer.

— Installation locale :
Il faut installer Python 3 (miniconda ou anaconda) ainsi qu'un éditeur amélioré comme Pyzo ou Atom.
Il est aussi très intéressant d'installer Jupyter qui est un notebook où on peut mélanger du code avec du texte, du Latex, des images, des vidéos. On peut laisser des parties vides à compléter par l'élève.

— Utilisation à distance :
Certains sites permettent de coder en Python sans toucher au contenu de son ordinateur.
Par exemple : Jupyter.org et cocalc.com
Cela peut être intéressant pour partager un document avec une classe et récupérer les documents des élèves.

— Un autre choix serait l'utilisation d'un logiciel installé sur une clé usb.

Chapitre 2

Python comme une calculatrice

Dans ce chapitre nous n'allons pas parler d'algorithmique mais juste montrer comment on peut utiliser ce langage comme une calculatrice puissante.
Dans la fenêtre de commande, on peut accéder aux dernières commandes rentrées en utilisant la flèche du haut. D'autre part la dernière valeur calculée est stockée dans la variable `_` (cela dépend du Python que vous avez installé).

Remarque : La fenêtre de commande est aussi appelée Shell ou interpréteur.

I Valeurs renvoyées et affichage

Toute commande (complète et valide) écrite dans la fenêtre de commande va renvoyer une valeur et peut agir sur l'affichage. Après l'évaluation de la commande, l'interpréteur Python va afficher la valeur renvoyée.
Par exemple, si vous tapez `3 + 2` puis la touche entrée (je ne préciserai plus qu'il faut appuyer sur la touche entrée), Python évalue la commande comme valant 5, et va donc afficher 5. Si vous tapez maintenant `print(3 + 2)`, on va avoir l'impression d'obtenir le même résultat car 5 va s'afficher. Cependant, cette fois-ci aucune valeur n'est renvoyée par cette commande et un affichage est fait.
On peut mieux voir la différence si on met la valeur retournée dans une variable. Comparer par exemple `x = 3 + 5` et `x = print(3 + 5)`. Pour afficher ce que contient la variable `x`, il suffit d'écrire le nom de la variable et appuyer sur entrée.
Lorsqu'une ligne est un peu longue, on peut la couper en rajoutant le caractère `\` à la fin de la ligne.

II Les différents types de nombres

Les opérateurs communs sur les nombres sont `+`, `-`, `*`, `**` (puissance). `-` est à la fois un opérateur binaire comme dans l'expression `2-3` et un opérateur unaire lorsqu'on écrit `-2`. Les priorités opératoires sont les priorités usuelles et on peut utiliser les parenthèses pour modifier l'ordre du calcul.

Les relations disponibles sur les nombres sont `<`, `>`, `<=`, `>=` (sauf pour les complexes) et `==` (test d'égalité), `!=` (non égalité). On peut même enchaîner les relations, par exemple `0 < x <= 1` est équivalent à `0 < x and x <= 1`

1) Nombres entiers, variables

Le premier type d'objet est le type entier. `type(4)`. Tout est stocké dans la mémoire à une adresse obtenue par la fonction `id()`. Essayez par exemple `id(4)`, `id(4) - id(3)`.

Exercice 1 : la suite définie par `id(n)` avec $n \in \mathbb{N}$, est-elle arithmétique ?

En plus des opérateurs sur les nombres, il y a les deux opérateurs, `//` (division entière), `%` (modulo). Attention, pour le modulo, il est prioritaire sur l'addition mais pas sur la multiplication, de sorte que `3 * 7 %2` renvoie bien 1 mais `3 + 7 %2` renvoie 4.

Exercice 2 : Donner les 3 derniers chiffres de 2017^{2018}.

On peut remarquer sur cet exemple que Python travaille sur des nombres arbitrairement grand, et avec des valeurs toujours exactes. Quelque fois l'affichage du nombre prend plus de temps que son calcul. Faire une affectation d'un calcul, va permettre de ne pas l'afficher et d'éviter ce problème. Pour mettre en mémoire des nombres on utilise le symbole `=`. Par exemple `a = 3**10 - 2**10`. Python commence par évaluer la partie droite de l'égalité et modifie la valeur de la variable `a`.
On peut également chaîner les affectations : `a = b = 2**65`. Cela équivaut aux instructions `a = 2**65` et `b = 2**65`, sauf que dans le premier cas, on utilise moins de mémoire car `a` et `b` vont avoir la même adresse.
Enfin il existe une affectation multiple simultanée. Par exemple pour échanger les valeurs de `a` et `b` il suffit d'écrire `a, b = b, a`.
Application : On veut calculer le PGCD de 807 et 210 avec l'algorithme d'Euclide. On écrit `a = 807` puis `b = 210` et ensuite on évalue `a, b = b, a % b; b` jusqu'à ce que `b` soit égal à 0. Le `;` sert à séparer les instructions au sein d'une même ligne. Le pgcd de a et b sera alors le nombre contenu par la variable a.

Exercice 3 : Chiffrement RSA. On suppose que la clé publique est constitué du nombre `n = 253` et de la clé de chiffrement `c = 3`. Pour chiffrer un nombre `x` (compris entre 0 et $n-1$), on calcule `x**c %n` et pour déchiffrer un nombre y, on calcule `y**d %n` où d est la clé secrète.
En chiffrant et déchiffrant des messages, trouvez la clé d sachant que d est compris entre 145 et 150.

Écriture binaire des nombres On peut lire cette section en deuxième lecture seulement. Cependant, les outils que proposent Python sont non seulement très pratiques, mais aussi très utiles dans beaucoup de domaines.
Par exemple, comment envisageriez-vous de faire une boucle sur tous les sous-ensembles d'un ensemble donné ?

Pour avoir l'écriture binaire d'un nombre `x`, il suffit d'écrire `bin(x)`. Pour l'écriture hexadécimale, on écrit `hex(x)`. On peut également rentrer directement des nombres en utilisant cette écriture. Par exemple `a=0b110` équivaut à `a = 6`.

Les opérateurs utilisant l'écriture binaire des nombres sont `|` (ou bit à bit), `&` (et bit à bit), `^` (xor bit à bit, à ne pas confondre avec la puissance ; essayez par exemple `2^3`), `«` (glissade vers la gauche, c'est à dire multiplication par 2), `»` (glissade vers la droite, c'est à dire quotient de la division euclidienne par 2).

Exercice 4 : Supposons que a soit un entier défini sur 32 bits au maximum, pour k entier compris entre 0 et 31, que fait la commande `(a « k) & ((1 « 32) - 1)) | ( a» (32 - k))` ? (le `&` est prioritaire)

2) Nombres flottants

Pour faire du calcul approché, on peut utiliser des nombres flottants sous Python. Ils sont caractérisés par la présence d'un `.`. Attention cependant à ne pas les confondre avec les nombres décimaux. En effet ils sont représentés en mémoire à l'aide d'un système binaire, ce qui fait que les nombres qui tombent justes du point de vue de l'ordinateur sont seulement les nombres x tels qu'il existe $n \in \mathbb{N}$ avec $2^n x \in \mathbb{Z}$.
Par exmple, le nombre 0,1 n'en fait pas partie, sinon il existerait $(n, k) \in \mathbb{N}^2$ tel que $\frac{2^n}{10} = k$, soit $2^n = 10k$, ce qui entraînerait que 5 divise une puissance de 2, d'après Gauss c'est impossible. En essayant `3*0.1-0.3`, on se rend mieux compte qu'une approximation est faite. Donc méfiance !

Les opérateurs sur les flottants sont `+`, `-`, `*`, `**` et `/` (division approchée).

Exercice 5 : Donner une valeur approchée du nombre d'or $\phi = \dfrac{\sqrt{5}+1}{2}$.

Si vous utilisez l'opérateur `/` avec des entiers, Python convertira automatiquement en nombre flottant.

Pour utiliser les fonctions mathématiques usuelles, il faut importer le module math :
`from math import *`, ce qui signifie d'importer toutes les fonctions du module math.
On peut importer uniquement certaines fonctions, en écrivant par exemple :
`from math import sqrt, cos`.
Ou alors, on peut tout simplement signaler qu'on va avoir besoin de certaines fonctions du module math en écrivant `import math`, on accédera ensuite aux différentes fonctions du module en les faisant précéder de `math.`, par exemple `math.pi` pour la constante π. Cette façon de faire évite d'écraser d'autres fonctions qui pourraient avoir le même nom. C'est une fonctionnalité bien utile lorsqu'on travaille à plusieurs sur un même projet.
Les principales fonctions du module mathématiques sont `sin`, `cos`, `tan`, `atan`, `acos`, `asin`, `sqrt`, `exp`, `log` (népérien, malgré son nom). On trouve également dans le module math le nombre `e` et le nombre `pi`. Ces variables ne sont pas protégées, vous pouvez donc les effacer par mégarde.

3) Nombres complexes

Python est un des rares langages à avoir les nombres de type complexe en standard. Pour écrire un nombre complexe, il faut coller la partie imaginaire à la lettre `j`. Par exemple `2+3j` est le nombre complexe de partie réelle 2 et de partie imaginaire 3. Si la partie imaginaire est nulle, on doit écrire `x+0j` où x est la partie réelle de ce complexe.
Si a est une variable contenant un complexe, `a.real` est sa partie réelle, `a.imag` est sa partie imaginaire et `a.conjugate()` est le conjugué du nombre a, tandis que `abs(a)` donne le module du nombre a.

4) Conversion de nombres

`complex()` convertit un entier ou un nombre flottant en complexe.
`int()` convertit un flottant en nombre entier. Attention que `int(-0.3)` renvoie 0, et non -1

comme la partie entière mathématique. Notez en revanche que `-0.3//1` renvoie -1, tandis que `-0.3%1` renvoie 0.7. Pour la partie entière mathématique on peut également utiliser la fonction `floor()` disponible dans le package math (`from math import floor`).
`float()` convertit un nombre entier en nombre arrondi. Par exemple `float(3)` donne `3.`, mais on peut également écrire `0.+3` pour obtenir le nombre flottant.

III Booléens

Il y a un type spécial prenant deux valeurs : `True` et `False` (notez bien la majuscule au début). Lorsque l'on doit évaluer pour un test une expression, elle vaudra `False` pour les objets vides ou nuls et `True` pour les autres. On peut donc (si on ne se rappelle pas l'écriture des deux booléens), remplacer `True` par 1, et `False` par 0.
Les opérateurs sur les booléens sont `not`, `and` et `or`.
Par exemple `a=5 and 7` attribuera la valeur 7 à la variable a, tandis que `a=5 or 7` lui attribuera la valeur 5.

Remarque : On peut également se servir de ces opérateurs pour faire des branchements conditionnels. Par exemple, lorsqu'on écrit `A and B`, si A est évalué à False, cela renverra A, sinon cela renverra B. On peut également écrire sur une ligne : `a= 8 if b else 9`, de sorte que si b est interprété comme True, la variable a contiendra 8 et sinon elle contiendra 9.

Exercice 6 : Comment écrire seulement à l'aide de parenthèses et des opérateurs `or` et `and`, l'expression `A if B else C` ?

IV Les conteneurs

Il existe différents types d'objets pouvant en contenir plusieurs, on va les appeler les conteneurs. Ce sont d'une part les listes, les tuples, les chaînes de caractères et les « range » qui sont indexés (en quelque sorte ordonnés) et d'autre part les ensembles et les dictionnaires qui ne sont pas indexés (pas d'ordre a priori).
Ces conteneurs peuvent être utilisés comme des itérables à l'intérieur d'une boucle `for`. Ils ont également en commun la fonction donnant leur nombre d'éléments : `len()`. Par exemple `len('abcde')` renvoie 5.
On peut aussi tester si un élément `e` est dans le conteneur `k`, en écrivant `e in k`. Par exemple `3 in [1,5,3,7]` renvoie True.
Lorsqu'une addition compatible avec des entiers a été définie, on peut écrire `sum(`ℓ`)` pour additionner les éléments de ℓ. Cela ne marche donc pas si ℓ contient des chaînes de caractères car l'addition de 0 avec une chaîne de caractère renvoie une erreur.
Si un opérateur de comparaison a été défini sur les objets d'une liste ℓ, `min(`ℓ`)` et `max(`ℓ`)` renvoient respectivement le plus petit et le plus grand élément de ℓ qui doit être non vide. On verra donc que cela peut être utilisé pour des listes contenant des chaîne de caractères (ordre lexicographique).

1) Les conteneurs indexés

a) Les listes

(paragraphe à reprendre : temps d'accès, insertion d'éléments)

Les listes sont des objets mutables, c'est à dire modifiables sans copie. Elles sont représentées par des tableaux dynamiques, que l'on peut rallonger dans une certaine mesure. Si la place vient à manquer, la liste entière est recopiée ailleurs avec une place disponible plus grande.
Chaque case de la liste contient l'adresse de son élément. Dans la liste on peut donc trouver des objets de différents types.
Prenons par exemple la suite `[[3,9],5,6,[[8],[]]]`. Elle contient quatre éléments qui sont une liste à deux éléments `[3,9]`, `5`, `6` et `[[8],[]]`. Cette dernière liste a deux éléments qui sont une liste à un élément et une liste vide.

On utilisera les listes pour faire des objets complexes comme les graphes ou les arbres, ou bien pour des objets dont la taille peut être arbitrairement grande.

b) Les tuples

Les tuples sont représentés à l'aide de parenthèses et fonctionnent à peu près comme les listes, sauf qu'ils ne sont pas dynamiques, et en particulier leur taille est fixée une fois pour toute. On peut omettre les parenthèses. Par exemple, `(1,3,2)` et `1,3,2` vont donner le même tuple à 3 éléments.

On a déjà vu qu'on pouvait faire une affectation multiple de la forme `a,b=b,a`, il s'agit en fait d'une affectation avec un tuple de variables. De sorte que les écritures `(x,y)=(1,2)`, `x,y=(1,2)`, `x,y=1,2` et `(x,y)=1,2` sont équivalentes. On peut également faire des affectations à des niveaux plus profonds, comme `(x,y),(a,b)=(1,2),(3,4)`.

Pour différencier les tuples de longueur 1, on rajoute une virgule. Par exemple `(2,)` est un tuple de longueur 1, alors que `2` et `(2)` sont égaux à l'entier 2.

c) Les chaînes de caractères

Comme les tuples, les chaînes de caractères ont une taille fixe. Il s'agit d'une suite finie de caractères. On peut les écrire entre guillemets, quote ou triple guillemets. Par exemple `s="O'Brian"` ou `s='Il dit "bonjour"'`. Les triples guillemets permettent d'inclure des retours à la ligne.
On peut transformer tout objet en chaîne de caractères avec la fonction `str()`. C'est utile pour l'affichage par exemple.

d) Les « range »

Il y a plusieurs syntaxes pour ces objets, suivant le nombre d'arguments qu'on donne à la fonction `range`. La plus fréquente est `range(n)` avec n entier naturel, ce qui représente les entiers de 0 jusqu'à $n-1$, c'est à dire les n premiers entiers naturels. Si n vaut 0, l'objet est vide.
La deuxième syntaxe est `range(a, b)` avec a et b entiers relatifs. Cela représente tous les entiers de a inclus à b exclu. Si $a \geq b$ alors l'objet est vide.
La troisième syntaxe est `range(a, b, c)` avec a, b et c entiers relatifs et c non nul. Si $c > 0$, cet objet génère tous les entiers à partir de a jusqu'à b exclu avec un pas égal à c. Par exemple, `range(1, 99, 2)` représente les entiers impairs de 1 à 97. Si $c < 0$, les entiers sont générés de façon décroissante de la même manière. Par exemple `range(4, 1, -1)` génère les entiers 4, 3 et 2 (1 est exclu).

Ces objets sont à différentier des listes et des tuples dans la mesure où la liste des entiers qu'ils représentent n'est pas stockée en mémoire. Ainsi, on peut parfaitement écrire `range(10 ** 100)` sans provoquer une erreur de dépassement mémoire.

Comme tout objet, on peut les stocker en mémoire. Il est même recommandé de ne pas s'en priver. Par exemple `r = range(100)` peut permettre de gagner en concision dans certains programmes, et on peut même manipuler la variable `r` avec l'indexation et le tranchage !

e) Indexation

Pour tous ces conteneurs : listes, tuples, chaînes de caractères, on peut accéder à un élément en particulier à l'aide des crochets. Le premier élément est indexé par 0 et le dernier par $n-1$ où n est la longueur du conteneur. Par exemple `'1234'[2]` donnera 3. Si on veut accéder au dernier élément, on va utiliser des index négatifs `s[-1]` sera le dernier élément de s, `s[-2]` sera l'avant dernier, etc.

f) Tranchage

Lorsque `k` est un conteneur, on peut fabriquer un nouveau conteneur avec la syntaxe `k[d:f]` ou bien `k[d:f:p]` où d désigne le premier indice du conteneur que l'on considère, f le dernier (non inclus) et p le pas.
Si p est omis il vaut 1. Si d est omis alors il vaut 0 (début de la liste) lorsque $p > 0$ et -1 (fin de la liste) lors que $p < 0$. Si f est omis, cela signifie qu'on s'arrête lorsqu'on dépasse les bornes de la liste.
Par exemple, si `k=[0,1,2,3,4,5,6,7,8,9]`, `k[1:7]` sera `[1,2,3,4,5,6]` et `k[8:2:-2]` sera `[8,6,4]`. Pour renverser un conteneur, on écrira donc tout simplement `k[::-1]`. Par exemple `"bonjour"[::-1]` donnera la chaîne `'ruojnob'`

g) Opérateurs

Il n'y a que deux opérateurs pour les conteneurs qui sont `+` et `*`. L'addition permet de mettre bout à bout deux conteneurs, et la multiplication généralise l'addition. Par exemple `'bonjour'+' '+'toto'` donne `'bonjour toto'` et `[0]*5` donne `[0,0,0,0,0]`. Pour créer un triplet à partir d'un couple, on écrit par exemple `(1,2)+(3,)`
Application : Pour mélanger les lettres d'un mot de neuf lettres, une méthode classique consiste à écrire de gauche à droite puis de base en haut les lettres du mot dans un carré 3×3, puis à lire le mot formé de haut en bas puis de gauche à droite. Avec le mot `s='existence'`, l'opération peut être réalisée de cette manière : `s[::3]+s[1::3]+s[2::3]`

h) Comparaison

Les comparaisons sont possibles et utilisent l'ordre lexicographique. On compare l'élément le plus à gauche et en cas d'égalité on passe au suivant. Par exemple `[1,'a']<[1,'ab']` renvoie True.

i) Tri, transformation en ensemble

On peut utiliser la fonction `sorted()` pour trier un conteneur, mais cette fonction commence par convertir le conteneur en une liste, de sorte que cela renvoie une liste triée. Par exemple `sorted('bac')` renvoie `['a','b','c']`. Une astuce pour trier une chaîne de caractères est d'écrire : `''.join(sorted(s))` où s est la chaîne de caractères.
On peut également écrire `set(C)` pour transformer un conteneur en un ensemble (pour éviter les répétitions).
Par exemple `set('coucou')` est un ensemble à 3 éléments.

j) Méthodes sur les conteneurs indexés

On notera `C` un conteneur indexé et `a` un élément.

— `C.index(a)` va donner le premier indice i tel que `C[i]=a`. Renvoie une erreur si `a not in C`. On peut rajouter deux arguments (début et fin).
Par exemple `C.index(a,debut,fin)` va chercher a entre l'index `debut` et `fin-1`.
Il existe également la méthode `rindex()` (right index) qui commence par la fin du conteneur.

— `C.find(a)` va donner le premier indice i tel que `C[i]=a`. Renvoie `-1` si `a not in C`.

— `C.count(a)` va donner le nombre d'occurence de `a` dans `C`.
Par exemple `'abdae'.count('a')` renvoie 2.

2) Les conteneurs non indexés

a) Ensembles

Les ensembles comme les dictionnaires se notent à l'aide d'accolades. Par exemple `E={3,4,4}` désigne l'ensemble qui contient les éléments 3 et 4. Mais si on écrit simplement `E={}`, `E` sera interprété comme un dictionnaire. Il faut donc écrire `E=set({})`, ou tout simplement `E=set()`.

Les opérateurs sur les ensembles sont `|` (réunion), `&` (intersection), `-` (différence), `^` (différence symétrique). L'explication du choix de ces caractères vient des opérateurs binaires sur les nombres. Si on considère par exemple qu'il n'y a que 4 objets possibles, on pourrait définir un ensemble par un entier compris entre 0 et $2^4 - 1$. La représentation binaire de cet entier indiquerait si l'objet est présent ou non dans l'ensemble.

On peut ajouter ou enlever des éléments dans un ensemble en utilisant les affectations augmentées `|=` et `-=`, ou bien on peut utiliser les méthodes `add()` et `remove()` (ou `discard()`)

On peut comparer des ensembles avec la relation `<=` pour l'inclusion et `<` pour l'inclusion stricte.

b) Dictionnaires

Les dictionnaires sont des listes d'association. Si on veut comparer cela à un objet mathématique, il s'agit d'une fonction. L'ensemble de départ est appelé KEYS et l'ensemble d'arrivée VALUES. On peut les initialiser par exemple comme cela : `E={'chat':'cat', 'chien':'dog'}`. On peut ensuite compléter le dictionnaire en rajoutant des valeurs avec les crochets. Par exemple `E[(1,2)]=2` ou `E['chat']='kitty'` qui va écraser l'ancienne valeur associée à `'chat'`. Les dictionnaires sont optimisés pour la recherche des valeurs associées aux clés. Les clés sont en effet hachées pour accéder rapidement à la valeur.

3) Conteneurs définis par extension

On a fait remarquer au début de cette section que les conteneurs peuvent être utilisés dans une boucle for, en tant qu'itérable. Pour Python 3, l'itérable de référence est `range(n)`.

Pour fabriquer la liste des carrés des entiers inférieurs ou égaux à 10, on écrira alors `[i*i for i in range(11)]`. Pour connaitre les carrés des entiers modulo 7, on peut écrire `[(i*i)%7 for i in range(7)]`, ou bien `[(i,(i*i)%7) for i in range(7)]` ce qui per-

met de les écrire plus facilement dans une table, ou bien on peut encore écrire `{(i*i)%7 for i in range(7)}` pour n'avoir que les nombres qui sont des carrés.

On peut également multiplier les boucles for, par exemple :
`[(i,j) for i in ['Pique','Carreau','Trefle','Coeur'] for j in '23456789dVDRA']`
va permettre de fabriquer un jeu de cartes classique.

On peut également ne prendre que des éléments vérifiant une condition. Par exemple, trouvons les entiers plus petit que 100 qui sont somme de 3 carrés :
Pour raccourcir l'expression suivante on écrit d'abord `r=range(11)` puis
`E={i*i+j*j+k*k for i in r for j in r for k in r if i*i+j*j+k*k<=100}`
Pour trouver maintenant les entiers inférieurs ou égaux à 100 qui ne sont pas somme de 3 carrés, on écrit :
`F={i for i in range(101) if i not in E}`

Exercice 7 : Parmi les entiers n de 1 à 100 quels sont ceux qui ont le moins de carrés modulo n en proportion ? Par exemple pour $n = 1$ le seul carré est 0 donc 100% de carrés, pour $n = 4$, les seuls carrés modulo 4 sont 0 et 1 donc il y a 50% de carrés.

Si on veut également définir des chaîne de caractères par extension, il faut utiliser la méthode `join()`. Par exemple si `s` est une chaine de caractères dont on veut garder uniquement les lettres minuscules, on peut écrire :
`".join([i for i in s if 'a'<=i<='z'])`

Exercice 8 : On suppose que la variable `texte` contient un texte (assez long). Que donne la commande suivante :
`sorted([(texte.count(i)/len(texte),i) for i in set(texte)])[-5:]`

4) Les fonctions et méthodes spécifiques aux listes

a) Méthodes associées

En python coexiste à la fois des fonctions ou procédures que l'on appelle en mettant le nom de la fonction suivi, entre parenthèses, des arguments de cette fonction séparés par une virgule, et des méthodes que l'on appelle en faisant suivre l'objet par un point, puis le nom de la méthode avec les arguments entre parenthèses. Nous avons déjà rencontré la fonction `len()`. Nous allons voir maintenant des méthodes pour les conteneurs. Il faut voir les méthodes comme des fonctions agissant sur l'objet lui-même et le transformant.
Prenons par exemple `ℓ=[1,4,3,5]` et posons `m=ℓ`. Remarquez que `ℓ.reverse()` ne renvoie rien mais a renversé à la fois les valeurs de `ℓ` et `m`. De même `m.sort()` va avoir pour effet de trier les deux listes. `ℓ.pop()` va retirer et renvoyer le dernier élément de la liste, on peut également retirer un élément particulier, par exemple `ℓ.pop(2)` retire et retourne le troisième élément de la liste (et non le deuxième).
Application : On veut tirer au sort des élèves passant au tableau. On va utiliser la fonction `randrange()` qui prend (au moins) un argument entier $n > 0$ et renvoie un entier au hasard entre 0 et $n-1$. Pour charger le module des nombres aléatoires, écrire `from random import *`. Soit `ℓ=['Alice','Bob','Charlie']`. La longueur de cette liste est `n=len(ℓ)`. Pour choisir un élève au hasard, il suffit de faire `ℓ.pop(randrange(len(l)))` autant de fois que nécessaire (on utilise la flèche du haut pour refaire la commande).

Exercice 9 : Mélanger le jeu de carte.

Voici d'autre méthodes utiles pour les listes :
— `ℓ.extend(m)` prolonge la liste `ℓ` avec la liste `m`. Cela fabrique donc la liste `ℓ+m` mais en transformant la liste `ℓ`. Une autre manière de faire la même chose est d'utiliser l'affectation augmentée `+=`, en écrivant `ℓ += m`. Ce genre de procédé est donc différent de `l=l+m` puisque dans ce cas une nouvelle liste est fabriquée.
— `ℓ.append(x)` rajoute l'élément `x` à la fin de la liste `ℓ`.
— `ℓ.remove(i)` enlève l'élément situé à l'index i. On peut également écrire `del ℓ[i]`.
— `ℓ.insert(i,x)` insère l'élément x à la position i.

b) Tranchage

On peut aussi utiliser cet opérateur en affectation de liste.
Par exemple pour `ℓ=[0,1,2,3,4,5,6,7,8,9]` , `ℓ[::2]=l[1::2]` va transformer la liste `ℓ` en `[1,1,3,3,5,5,7,7,9,9]`.

5) Méthodes spécifiques aux chaînes de caractères

`s` désigne une chaîne de caractères.
— `s.isalpha()`, `s.isnum()`, `s.isalnum()` vont tester respectivement si s est contient que des caractères utilisés dans des mots (y compris accentués), que des chiffres, ou bien l'un ou l'autre.
— `s.lower()` renvoie la chaîne de caractère où les caractères de s ont été mis en minuscule.
— `s.upper()` comme le précédent mais en transformant en majuscule cette fois-ci.
Il existe bien sûr beaucoup d'autres fonctions mais on peut déjà faire pas mal de choses avec ces commandes. On peut par exemple se passer aisément des nombres ASCII, en définissant un alphabet : `alphabet='abcdefghijklmnopqrstuvwxyz'`.

Exercice 10 : Supposons que la variable `texte` ne contiennent que des lettres minuscules de a à z. Que fait la commande suivante ?
`''.join(alphabet[(alphabet.index(i)+3)%26] for i in texte)`

V Le type 'type' et la conversion de types

1) Le type 'type'

Nous verrons dans le chapitre de la programmation que nous pouvons définir de nouveaux types, que l'on appelle également « classes ».
Un dernier type est disponible qui le type associé aux types. Il peut prendre les valeurs `int`, `float`, `complex`, `bool`, `str`, `list`, `tuple`, `set`, `dict` et `type` !
On peut tester le type d'une expression ou d'une variable en écrivant `type(2.)==float`.

2) Conversion de types

`str()` transforme à peu près n'importe quoi en une chaîne de caractère. La fonction `repr()` fait (presque tout le temps) la même chose.
`int()` peut transformer une chaîne de caractères qui représentent un nombre en un nombre.
`eval(s)` évalue l'expression contenue dans la chaine de caractère `s`. Par exemple `eval('2**3')`

va renvoyer 8.
`list()` transforme un conteneur en une liste, par exemple une chaîne de caractères en une liste de caractères.
`".join(`ℓ`)` transforme la liste de caractères contenue dans ℓ en une chaîne de caractères. De façon plus générale, si `s`, t_1, ... t_n sont des chaines de caractères, `s.join([`t_1`,...,`t_n`])` renvoie la chaine de caractère t_1`+s+`t_2`+s+...+s+`t_n
`dict()` peut convertir une liste de couples en un dictionnaire.
`set()` converti un conteneur en un ensemble.
`tuple()` transforme un conteneur en un tuple.

VI Exercices

Exercice 11 : On veut simuler le jeu du lièvre et de la tortue. La règle du jeu est la suivante : le lièvre et la tortue font un parcours de 4 cases et se trouvent au début du jeu sur la case départ. On lance autant de fois que nécéssaire un dé à 6 faces, si il tombe sur un nombre de 1 à 5, la tortue avance d'une case, et si il tombe sur 6 le lièvre avance de 4 cases et gagne la partie.
La commande `randint(1,6)` permet de simuler le jet d'un dé à 6 face. On pourrait écrire également `randrange(6)+1`. On rappelle que ces commandes sont dans le module random.
On peut simuler une partie en regardant si un 6 apparait dans le lancé de 4 dés :
`'Lievre' if 6 in [randint(1,6) for j in range(4)] else 'Tortue'`
Simuler le jeu 1 million de fois et calculer la fréquence de la victoire du Lièvre.

Exercice 12 : Un poème médiéval « De Vetula »décrit un jeu où on s'intéresse à la somme de 3 dés. On veut savoir s'il est plus avantageux d'essayer de faire 10 ou 11, ou bien de réaliser 9 ou 12.
Compter le nombre de triplets de dés (dé1,dé2,dé3) dont la somme fait 10 ou 11, et faire de même avec 9 ou 12. Conclure.

Exercice 13 : Dans le livre Recreation in theory of numbers, on peut voir les 3 premiers défis suivants en introduction :

1. Find the divisors, if any, of 16 000 001.
2. One side of a right-angled triangle is 48. Find ten pairs of whole numbers which may represent the other two sides.
3. How many positive integers are there less than and having no divisor in common with 5929. ?

Sauriez-vous rapidement répondre en utilisant Python ?

Exercice 14 : Trouver le premier carré dont l'écriture décimale utilise tous les chiffres de 0 à 9.

Exercice 15 : Une professeure des écoles propose l'exercice suivant à ses élèves :

> Chaque année Isidore prend du poids, mais cette année il a tellement honte
> qu'il donne son poids sous forme d'une énigme.
> Son ancien poids est un nombre à trois chiffres,
> auquel il additionne un nombre de deux chiffres.
> Le tout donne un nombre de trois chiffres :
> _ _ + _ _ _ = _ _ _
> Dans cette addition, chaque chiffre de 0 à 7 est utilisé une seule fois.

Quel peut être le poids d'Isidore ?

Exercice 16 : `premiers=[i for i in range(2,100000) \`
`if len([k for k in range(1,int(i**0.5+1)) if i%k==0])==1]` donne la liste de tous les nombres premiers inférieurs à 100000.
Construire alors une liste contenant les couples $(i, \phi(i))$ où ϕ est la densité des nombres premiers.

Solutions des exercices

Exercice 1 : Non, `id(5)-id(4)` est différent de `id(1001)-id(1000)`.

Exercice 2 : `2017**2018%1000` donne 009.

Exercice 3 : On trouve facilement que $d = 147$. Par exemple, `112**3%253` donne 19 et `19**147%253` donne 112 à nouveau.

Exercice 4 : Il s'agit d'une rotation de k bits vers la droite d'une entier écrit sur 32 bits.

Exercice 5 : On peut se passer de la racine carrée (vu plus loin) en utilisant les puissances non entières : `(5**0.5+1)/2` donne le résultat.

Exercice 6 : `(B and A) or C`

Exercice 7 : Le nombre de carrés modulo n est donné par l'expression :
`len({i*i%n for i in range(n)})`.
On va donc définir la liste :
`[(len({i*i%n for i in range(n)})/n,n) for n in range(1,101)]` et prendre son minimum qui est $14,6\%$ environ pour $n = 96$.

Exercice 8 : La commande va donner les 5 lettres les plus fréquentes dans le texte.

Exercice 9 : Si jeu contient le jeu de carte, on écrit :
`jeu_melange=[jeu[randrange(len(jeu))] for i in range(52)]`

Exercice 10 : Il s'agit du chiffrement de César, décalage de 3 lettres (a devient d, b devient e etc.)

Exercice 11 : On pose `n=1000000` puis on calcule par exemple :
`len([1 for i in range(n) if 6 in [randint(1,6) for j in range(4)] ])/n`

Exercice 12 : On pose `r=range(1,7)` pour abréger la liste des entiers de 1 à 6 inclus, puis la commande `len([(i,j,k) for i in r for j in r for k in r if i+j+k in [10,11]])` fournit une des deux réponses. En retirant la fonction `len()` on affiche tous les triplets correspondants.

Exercice 13 :

1. `n=16000001`, puis `[i for i in range(1,int(n**0.5)) if n%i==0]`
2. `L=[(i,sqrt(48**2+i*i)) for i in range(1,10000) if (48**2+i*i)**0.5%1==0]`
3. On trouve facilement que 7 et 11 sont les seuls diviseurs de 5929, donc la réponse est :
   `len([i for i in range(1,5929) if i%11!=0 and i%7!=0])`

Exercice 14 : Un nombre de 10 chiffres est supérieur ou égal à 10^9, donc on doit chercher à partir de $\sqrt{10}^9$, c'est à dire 31623.
`L=[i for i in range(31623,100000) if len(set(str(i*i)))==10]` contient les entiers dont le carré utilise tous les chiffres de 0 à 9 puisque `str(i*i)` est la chaîne de caractère contenant les chiffres du carré de i, et la fonction `set()` permet de ne garder qu'une fois chacun des chiffres présents.
Le plus petit est 32043, on le trouve par exemple avec `min(L)` ou `L[0]` puisque L est triée déjà.

Exercice 15 : Une façon simple d'y répondre est d'essayer toutes les combinaisons des nombres à 2 chiffres et des nombres à trois chiffres, en ne retenant que ceux qui utilisent tous les chiffres de 0 à 7 :

```
[(i,j,i+j) for i in range(10,100) for j in range(100,1000) \
if set(str(i)+str(j))==set('01234567')]
```

Exercice 16 :

```
L=[ (i,(premiers.index(i)+1)/i) for i in premiers ]
```

Chapitre 3

Formatages des entrées et des sorties

I Formatage des sorties

Avec Python 3, la méthode recommandée pour l'affichage est d'utilisée la méthode `format()`. Par exemple, plutôt que d'écrire `print('La somme de ',2,' et ',3,' fait ',5)` ou bien encore `print('La somme de '+str(2)+' et '+str(3)+' fait '+str(5))`, on va écrire : `'La somme de {} et {} fait {}'.format(2,3,5)`.
Les accolades seront donc remplacées, dans l'ordre, par chacun des arguments de la méthode `format()`. On peut également écrire le numéro de l'argument dans les accolades, ce qui autorise les duplications. Par exemple, dans cette lettre :

```
"Bonjour {0} {1}, ..., veuillez agréer {0} {1} l'expression \
de mes sentiments les meilleurs".format('M.','Dupond')
```

Une troisième façon, plus parlante, d'écrire la même chose est de nommer les arguments. On retrouvera cette façon de procéder dans le chapitre des fonctions. On écrit alors :

```
"Bonjour {civilite} {nom}, ..., veuillez agréer {civilite} {nom} l'expression \
de mes sentiments les meilleurs".format(civilite='M.',nom='Dupond')
```

Enfin et surtout, on peut spécifier un formatage de la sortie, en indiquant le format après le signe `:` dans les accolades, comme `{:.2f}`, ou encore `{0:.2f}` ou `{valeur:.2f}`, ce qui signifie dans chaque cas qu'un nombre flottant va être affiché avec en arrondissant au plus proche et en affichant exactement deux chiffres après la virgule.
Les types de format disponibles sont les suivants : `f` pour les entiers flottants, `d` pour les entiers en notation décimale, `n` pour les nombres en général (y compris complexes), `%` pour afficher un pourcentage, `e` pour les nombres en notations scientifique. Il y a également `b` pour la notation en binaire, `x` pour l'hexadécimal etc.
La syntaxe la plus générale du format est :

```
[[remplissage]alignement][signe][#][0][largeur][,][.precision][type]
```

On a déjà vu les différents types possibles, et la précision (pour les flottants). Voyons le reste.
`,` sert à séparer les chiffres trois par trois pour une question lisibilité (mais c'est une notation américaine)
La largeur essaye d'imposer une taille de la chaîne de caractères en sortie. C'est très pratique pour représenter des nombres sous forme d'un tableau, pour qu'ils soient bien en dessous les uns des autres.
Le 0 qui est mis éventuellement avant la largeur impose de remplir l'espace par des 0 inutiles, mais pratiques dans certains cas de figure. Par exemple, pour afficher tous les bits d'un octets on va utiliser le format `{:08b}`.

`#` sert uniquement pour les types binaires, hexadécimaux, octaux et rajoute le préfixe au début. Le signe peut prendre comme valeur un espace, + , - ou bien être absent. Si il n'y a pas le signe ou si le signe est -, l'écriture est l'écriture usuelle des nombres, si + est indiqué alors les nombres positifs sont écrits avec leur signe, et si le signe est un espace alors le signe + n'est pas mis pour les nombres positifs mais un espace est laissé à la place. C'est assez pratique pour aligner des nombres d'une matrice.
L'alignement prend les valeurs >, < ou ^, ce qui signifie respectivement que l'affichage sera fait à droite (par défaut), à gauche ou centré.
Enfin le caractère de remplissage sert à remplir les espaces pour que la chaîne ait bien la taille demandée.
Par exemple `'{:*^ 20.2f}'.format(3.1415926535)` va afficher `'******* 3.14********'`

II Gestion des entrées

Si on cherche à faire les opérations inverses de ce qui précède, c'est à dire identifier des nombres ou des chaînes de caractères dans une ligne et placer les résultats dans des variables, il existe un module Python dédié à cela, c'est `re`. La syntaxe demande un temps d'apprentissage mais est très proche de ce qui existe dans d'autres langages (Perl, awk de Unix etc.).

En Python 3, on utilise la fonction `input(message)` pour attendre une réponse de la part de l'utilisateur. La réponse est forcément sous forme d'une chaine de caractère. Si on veut lire un nombre entier, il faut alors écrire `reponse=int(input('Donner un nombre entier : '))`. Cette façon de faire est à éviter, il faut privilégier l'écriture de fonction où on passe les informations grâce aux paramètres.
A noter qu'on peut également utiliser des coroutines qui sont des programmes pouvant attendre des envois de données, et il existe également des modules pour gérer les communications entre processus, par exemple dans l'objectif de programmation parallèle.

Chapitre 4

Fonctions et portée des variables

Dans Pyzo, ouvrir un nouveau fichier (menu Fichier) puis écrire dans ce fichier `x=3`. A l'aide du menu Exécuter, vous cliquez sur Exécuter le contenu de l'onglet courant pour exécuter votre fichier, et vous pouvez constater dans le shell que la variable `x` contient bien 3.

I Introduction et syntaxe

1) Pourquoi écrire des fonctions ?

Un des principes fondamentaux de l'informatique est d'éviter les répétitions de code. Avec Scratch, au collège, vous avez peut être vu qu'on pouvait utiliser des « blocs » pouvant prendre des arguments et groupant plusieurs instructions.
Par exemple, pour dessiner un carré de côté donné c, on pouvait créer un bloc nommé `carré`, prenant un argument c et contenant les huit instructions :
avancer d'une longueur égale à c, tourner de 90 degrés,
avancer d'une longueur égale à c, tourner de 90 degrés,
avancer d'une longueur égale à c, tourner de 90 degrés,
avancer d'une longueur égale à c, tourner de 90 degrés.
Ou bien, en continuant à suivre ce principe d'éviter les répétitions, on pouvait créer un nouveau bloc `cote` prenant un paramètre c et contenant les deux instructions : avancer d'une longueur égale à c, tourner de 90 degrés. Ensuite le bloc `carre` pourrait s'écrire simplement en faisant appel quatre fois au bloc `cote`.
En faisant cela, le programme gagne autant en lisibilité qu'en concision.
Malheureusement, dans certaines situations, on a envie d'exécuter une suite d'instructions afin de trouver une valeur. Par exemple, pour le chiffrement de César ou son déchiffrement, ou bien pour le calcul d'une moyenne d'un tableau. Avec Scratch, on pouvait s'en sortir en modifiant la valeur d'une variable depuis l'intérieur du bloc. Cette façon de faire est absolument à proscrire lors de l'écriture de programmes complexes. En effet, il est très difficile de localiser une erreur lorsque l'exécution d'un bloc (qu'on appellera fonction) modifie des variables situées à l'extérieur du bloc.
C'est pourquoi la plupart des langages modernes proposent de renvoyer une valeur après l'exécution d'un bloc. En Python, on renvoie une valeur à l'aide du mot clef **return**.
On peut également faire le rapprochement avec les fonctions mathématiques : dans l'écriture $f(x) = x^2$, la variable x est définie localement et rien ne nous empêche de définir une autre fonction avec la même variable, par exemple $g(x) = x^3$. Dans le même paragraphe, on a pu également utiliser une variable que l'on a nommé x, et qui, bien entendu, n'a rien à voir avec les variables x dans l'écriture des fonctions.

Exercice 1 : Pouvez-vous devinez l'affichage après exécution du code suivant ?

```
x = 3
def f(x):
    return x * x

def g(x):
    return x ** 3

print(x)
print(f(2))
print(g(2))
print(f(g(2)))
print(x)
```

2) Syntaxe et écriture des fonctions en Python

Comme dans l'exemple, pour définir une fonction en python, on écrit :

```
def nom_de_la_fonction(arg1[, arg2[, arg3]]):
    instruction 1
    instruction 2
    ...
    return expression
```

Les « : » annoncent le début d'un bloc qu'on appelle le corps de la fonction et qui est décalé d'une tabulation ou de quatre espaces. La fin du bloc correspond à la fin du fichier ou bien à une ligne non vide qui commence sans indentation. Le mot clef **return** peut apparaître à plusieurs endroits dans le corps de la fonction, mais lors de l'appel de la fonction l'exécution s'arrête au premier **return** rencontré.

Exercice 2 : Que fait la fonction suivante ? Tous les **return** sont-ils utiles ?

```
def f(a, b):
    if a > 0:
        return 'croissant'
    if a < 0:
        return 'decroissant'
    return 'constant'
```

La norme Pep8 recommande d'écrire les noms des fonctions en minuscule, avec des _ pour séparer les mots lorsqu'il y en a plusieurs. Pour les arguments, il faut rajouter un espace après chaque virgule.

Exercice 3 : Trouver un nom de fonction plus approprié et corriger l'écriture de la fonction suivante :

```
def f(a,b):
    return (-b/a)
```

II Premières fonctions

Prenons un exemple simple : le calcul du volume d'un cône dont la base est un disque. Si r désigne le rayon du disque et h la hauteur du cône, le volume V du cône est $V = \frac{1}{3}\pi r^2 h$. La

formule se généralise lorsque la base n'est pas un disque par $V = \frac{1}{3}bh$ où b est l'aire de la base du cône.

On va alors écrire dans notre fichier les quatre lignes suivantes, en remarquant que la tabulation est automatique pour les deux lignes qui suivent la ligne commençant par le mot clef **def**.

```
from math import pi
def volume_cone(rayon, hauteur):
    base = pi * rayon ** 2
    return (1 / 3) * base * hauteur
```

Ensuite, après avoir exécuté cet onglet, on peut calculer le volume d'un cône de rayon 3 et de hauteur 4, en tapant `volume_cone(3, 4)`. A noter que lorsqu'on a tapé le début de l'appel de la fonction : `volume_cone(` une fenêtre d'aide nous indique qu'elle va prendre deux arguments qui s'appellent `rayon` et `hauteur`, d'où l'intérêt de bien nommer les arguments d'une fonction.
Vous pouvez également calculer ce même volume en écrivant `volume_cone(hauteur=4, rayon=3)` mais cela nécessite de connaître le nom des arguments, en revanche on voit que l'ordre des arguments peut ainsi être changé.

Si on veut maintenant améliorer la clarté de notre programme, on peut décomposer le calcul en écrivant une nouvelle fonction qui calcule l'aire d'un disque. On va également rajouter des commentaires à nos fonctions.

```
from math import pi

def volume_cone(rayon, hauteur):
    '''Calcule le volume d'un cone'''
    base = aire_disque(rayon)
    return (1 / 3) * base * hauteur


def aire_disque(rayon):
    '''Calcule l'aire d'un disque'''
    return pi * rayon ** 2
```

On peut remarquer que l'interpréteur accepte que l'on définisse la nouvelle fonction dans un deuxième temps, alors que l'on en a besoin dans la définition de la première fonction. Cela illustre bien le dynamisme du langage. L'inconvénient est que si on se trompe dans l'orthographe du nom de la fonction, Python ne nous le signalera que lors de l'exécution de la fonction. On verra qu'une fonction peut même s'appeler elle-même, c'est ce qu'on appelle la récursivité.
Pour accéder à l'aide d'une fonction, on écrit `?volume_cone` ou `?aire_disque`. Cette aide est donc non seulement utile dans le listing du programme, mais également pour un autre utilisateur qui souhaite avoir des précisions sur la fonction. C'est particulièrement pratique pour des projets écrits en groupe. Il faut donc toujours écrire ce petit bout d'aide. Grâce au triple guillemet l'aide peut prendre plusieurs lignes si vous le souhaitez.

III Portée des variables

Les variables en argument des fonctions ont une portée uniquement locale, c'est à dire qu'elles ne sont pas visibles depuis l'extérieur du corps de la fonction. En pratique ces variables sont renommées avec un nom qui dépend du numéro d'appel de la fonction (numéro unique), ce qui permet justement de faire de la récursivité (fonctions qui s'appellent elles-mêmes).

Les variables dans le corps de la fonction qui sont modifiées sont nécessairement locales, même si une variable du même nom a été définie en dehors. Si on veut vraiment changer la valeur d'une variable définie à l'extérieur de la fonction, il faut utiliser le mot clef **global** pour la déclarer dans le corps de la fonction. Ce genre de programmation est à proscrire (on appelle cela de l'effet de bord) car il est difficile d'imaginer les modifications induites dans l'appel d'une fonction. Voici quelques exemples : ...

IV Conditionner l'entrée dans une fonction

Considérons la fonction suivante basée sur le théorème des valeurs intermédiaires.

```
def dichotomie(f, a, b, precision):
        '''pour f continue, f(a)f(b)<=0
        trouve c entre a et b tel que
        |c-alpha|<=precision avec f(alpha)=0'''
        while abs(b - a) > precision:
                c = (a + b) / 2
                if f(c) * f(a) >= 0:
                        a = c
                else:
                        b = c
        return c
```

Normalement il faudrait vérifier que f est continue et que $f(a) \times f(b) \leq 0$ à l'entrée de la fonction. On ne pas vérifier que f est continue mais pour l'autre condition il suffit de rajouter une ligne en entrée de la fonction qui utilise le mot clef **assert** :

```
def dichotomie(f, a, b, precision):
'''pour f continue, f(a)f(b)<=0,
        trouve c entre a et b tel que
        |c-alpha|<=precision avec f(alpha)=0'''
        assert f(a) * f(b) <= 0
        while abs(b - a) > precision:
                c = (a + b) / 2
                if f(c) * f(a) >= 0:
                        a = c
                else:
                        b = c
        return c
```

Ainsi, si la condition n'est pas vérifiée, Python arrêtera l'évaluation et va afficher une erreur. On peut également vérifier le type des données. On peut le faire avec `type(x) == int` par exemple, mais la façon correcte de le faire est d'utiliser la fonction `isinstance()`, par exemple `isinstance(x, int)`, ce qui renvoie directement un booléen. L'avantage de cette fonction est qu'elle est compatible avec l'héritage. Il faut donc prendre l'habitude de l'utiliser.

V Fonctionnalités avancées

A ne pas lire en première lecture.

1) Fonction dans une fonction

On peut définir une fonction à l'intérieur d'une autre, ce qui est utile lorsque la fonction définie à l'intérieur n'a pas d'intérêt à l'extérieur, ou bien lorsqu'on veut renvoyer une fonction. Voici un exemple bien utile permettant de composer deux fonctions f et g, on remarquera que g ne possède qu'un argument dans ce cas (fonction à une variable), mais on peut facilement généraliser le programme à un nombre quelconque d'arguments :

```
def compose(f, g):
    ''' renvoie la composee des deux fonctions donnees en argument '''
    def h(x):
        return f(g(x))
    return h

def carre(x):
    return x * x

def deux(x):
    return 2 * x

f = compose(carre, deux)
```

2) Nombre quelconque d'arguments

Lorsque `ℓ` est une liste, on peut « déplier » cette liste en écrivant `*ℓ`.
Par exemple, si `ℓ=[2,3]`, on peut écrire `VolumeCone(*ℓ)`, ce qui a pour effet de « mettre à plat » les éléments de la liste ℓ.
À l'inverse, on peut se servir de cette écriture pour récupérer un nombre quelconque d'arguments.

```
def norme(*L):
    '''renvoie la norme du vecteur'''
    return sum([i * i for i in L]) ** 0.5
```

Sous sa forme la plus générale, on peut écrire les arguments d'une fonction `f` de cette façon : `def f(*arg,**karg)`, où `karg` est le dictionnaire des arguments nommés (c'est à dire rentrés sous la forme `rayon=3`), et `arg` la liste des arguments qui ne sont pas nommés. On doit toujours faire précéder les arguments qui ne sont pas nommés. Dans la définition de la fonction on peut également imposer un nombre minimum d'arguments obligatoires, et il faut alors les mettre en premier.

```
import math import pi
def volume_cone(**karg):
    '''Calcule le volume d'un cone
    a partir de la hauteur et de la donnee
    soit du rayon, soit de l'aire de la base'''
    base = karg['base'] if 'base' in karg else pi * karg['rayon'] ** 2
    return (1 / 3) * base * karg['hauteur']
```

Chapitre 5

Ecrire des programmes

L'écriture des programmes peut se faire dans des fichiers séparés. Dans Pyzo, une partie de la fenêtre affiche le contenu de ces fichiers, que l'on peut ré-évaluer à sa guise. Il est possible également d'écrire des programmes avec l'interpréteur mais la moindre faute nous oblige à tout réécrire.

I Structure des programmes

Un programme est constitué par une suite de fonctions et éventuellement certaines commandes. Un des principes fondamental en informatique est de ne jamais écrire deux fois le même code. On doit pour cela décomposer chaque tâche en autant de fonctions et combiner ses fonctions entre elles.
En cas de gros projet, il est préférable de décomposer l'ensemble des fonctions sous formes de différents fichiers, qui pourront être réutilisés pour d'autres projets sous forme d'un module.

II Les tests

Il y a plusieurs syntaxes pour les tests :

```
if a!=0:
        print("c'est un trinome")
```

ou bien :

```
if a!=0:
        print("c'est un trinome")
else:
        print("Ce n'est pas un trinome")
```

Ou bien encore :

```
if a!=0:
        print("c'est un trinome")
        delta=b**2-4*a*c
        if delta>0:
                print("il a deux racines")
        elif delta==0:
                print("il a une racine double")
        else:
                print("il n'a pas de racine reelle")
```

```
else:
        print("Ce n'est pas un trinome")
```

III Les boucles

Les boucles en Python sont les boucles **for** et les boucles **while**.

1) Boucle for

La syntaxe est `for <var1> in <gen>:` suivi d'un bloc d'instructions. `<var1>` est le nom d'une variable, de préférence pas utilisée avant car elle va être effacée, et `<gen1>` est le nom d'un générateur. Chaque conteneur est automatiquement interprété comme un générateur de ses éléments successifs. Pour les dictionnaires, il s'agit des clés. Si on veut faire une boucle sur les valeurs prises par un dictionnaire `d`, il faut écrire : `for v in d.values():`.
Le générateur le plus commun est bien sûr `range()`. A noter que `range(10000000)` n'est pas la liste contenant dix millions de nombres, mais c'est un générateur qui renvoie au fur et à mesure les entiers suivants (c'est une grosse différence avec Python 2). Ainsi, du point de vue de la mémoire c'est beaucoup plus efficace.
A noter également que ce type de boucles, contrairement aux autres langages, n'est pas forcément fini car le générateur peut être infini. Le programme suivant va afficher successivement 0, 1, et 3. Normalement la variable `i` doit prendre toutes les valeurs de 0 à 4 compris, mais lorsque `i` vaut 3 la boucle s'arrête de façon anticipée avec l'instruction **break**. Lorsque `i` vaut 2, l'instruction **continue** évite de lire le code qui suit et retourne directement en début de boucle. L'instruction `i=-1` à la fin de la boucle de sert à rien car c'est le générateur `range(5)` qui fournit les éléments un par un.

```
for i in range(5):
        if i==2:
                continue
        print(i)
        if i==3:
                break
        i=-1
```

2) Boucle while

La boucle fonctionne de façon classique `while <bool>:` suivi d'un bloc. Tant que le booléen `<bool>` est évalué à vrai, la boucle continue.

Chapitre 6

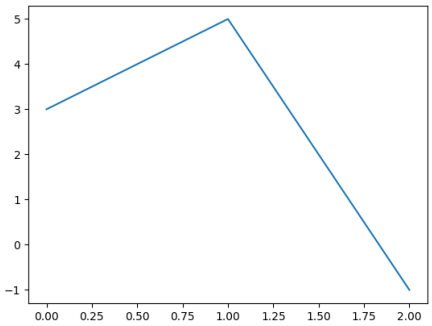
Réaliser des graphiques

I Matplotlib

Une première figure très simple.

```
import matplotlib.pyplot as plt

plt.plot([0,1,2],[3,5,-1])
plt.show()
```

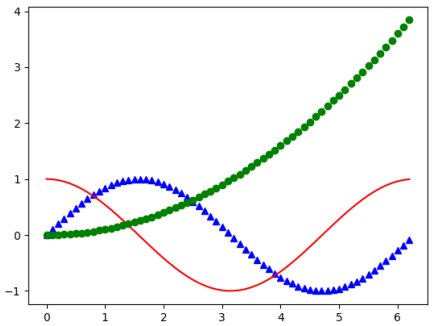


Avec la bibliothèque numpy, on peut facilement tracer des fonctions, comme dans l'exemple ci-dessous :

```
import matplotlib.pyplot as plt
from numpy import *

t=arange(0,2*pi,0.1)

plt.plot(t,sin(t),'b^',t,cos(t),'r-',t,t*
plt.show()
```

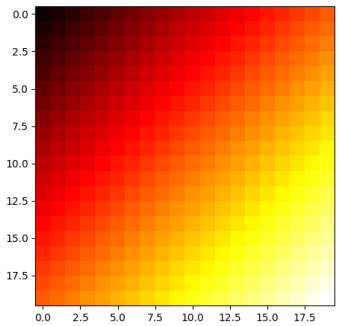


On peut également mettre des tableaux à deux dimensions en couleur. La bibliothèque numpy nous fournit une fonction bien utile (mais pas indispensable) pour créer des tableaux à deux dimensions.

```
import matplotlib.pyplot as plt
from numpy import *

t=zeros(shape=[20,20])
for i in range(20):
    for j in range(20):
        t[i][j]=i+j

plt.imshow(t,cmap='hot')
plt.show()
```



II Module turtle

Exemple spirale de Théodore : $z_0 = 1$ puis $z_{n+1} = f(z_n)$ avec $f(z) = z + i\frac{z}{|z|}$ ou bien le cas plus général $f(z) = az + b\frac{z}{|z|}$, avec a et b complexes non nuls.

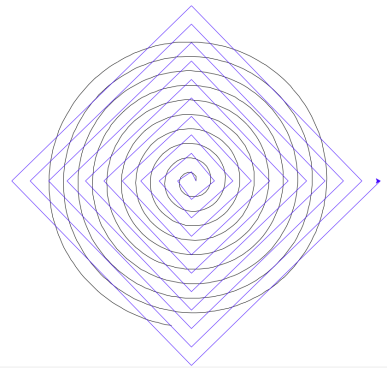
```
from turtle import *

def f(z,a=1,b=1j):
    return 0 if z==0 else a*z+b*z/abs(z)

def go(z,facteur=10):
    goto(z.real*facteur,z.imag*facteur)

def theodore(n,a=1,b=1j):
    '''n est le nombre de pas de la spirale'''
    reset
    z=1
    up()
    go(z)
    down()
    for i in range(n):
        z=f(z,a,b)
        go(z)

theodore(1000)
color('blue')
theodore(40,1j,1j)
```



Chapitre 7

Modules MpMath, NumPy et SciPy

I Module MpMath

Charger le module et écraser les autres fonctions en écrivant `from mpmath import *`.

Ce module permet de calculer avec une précision arbitraire. `mp.prec=100` donne une précision associée à 100 bits, ce qui n'est pas très pratique pour s'y retrouver en décimal. Il vaut mieux passer par `mp.dps=100` pour travailler avec une précision aussi proche que possible de 100 décimales.

Il y a aussi des fonctions très bien faites pour dessiner des courbes de fonctions, dans le plan et l'espace, et même pour les fonctions complexes.

II Modules numpy et scipy

Essentiellement pour faire des calculs rapides, avec des entiers écrits sur 64 bits par exemple ou avec des matrices.

1) Module stats de scipy

from `scipy.stats` **import** $*$

Le module stats contient les lois :

- `norm(a, b)` : loi normale, $\mu = a$, $\sigma = b$.
- `uniform(a, b)` : loi uniforme continue sur $[a, b]$
- `expon(scale = a)` : loi exponentielle de paramètre $\lambda = 1/a$.
- `poisson(a)` : loi de Poisson, $\lambda = a$.
- `randint(a, b)` : loi uniforme discrète sur $[a, b] \cap \mathbb{N}$.
- `bernoulli(p)` : loi de Bernoulli
- `binom(n, p)` : loi binomiale.
- `geom(p)` : loi géométrique.
- `hypergeom(M, n, N)` : Dans une urne M objets, n sont marqués, on en tire N et on compte les marqués.

Et voici les méthodes associées :

rvs() : génère une valeur aléatoire suivant la loi. `rvs(n)` génère n valeurs, on peut préciser la graine (ω) par `rvs(random_state = seed)`

pdf(x) : densité, pour une loi continue.

pmf(k) : loi de probabilité pour une loi discrète.

cdf(k) : fonction de répartition.

ppf(p) : inverse de la fonction de répartition.

mean() : espérance

median() : valeur médiane

std() : écart-type

var() : variance

interval(α) : intervalle de fluctuation avec le niveau de confiance α

Chapitre 8

Pour aller plus loin

I Les décorateurs

II Les itérateurs/générateurs

III Gestion des erreurs

Chapitre 9

Programmation orientée objet

I Opérateurs de classe

- `__init__(self, arg1, arg2, ...)` pour initialiser un objet de la classe.
- `__len__(self)` longueur de l'objet (si utile).
- `__bool__(self)` valeur booléenne de l'objet.
- `__repr__(self)` ou `__str__(self)` affichage de l'objet
- `__int__(self)`, `__float__(self)`, `__complex__(self)`, `__oct__(self)`, `__hex__(self)`, `__trunc__(self)` ... pour les conversions correspondantes.
- `__lt__(self)`, `__gt__(self)`, `__le__(self)`, `__ge__(self)`, `__eq__(self)`, `__ne__(self` correspondent aux opérateurs relationnels `<`, `>`, $\leq$, $\geq$, `==`, `!=`.
- `__add__(self)` et `__radd__(self)` pour l'opérateur d'addition. L'objet de gauche sert à déterminer comment s'effectue l'addition, et si rien n'est prévu par rapport au type de l'objet de droite, on utilise la méthode radd de l'objet de droite.

- `__prod__(self)`, `__rprod__(self)` Par exemple, si on définit une classe de vecteurs, on peut donner à la fois du sens à `v * w` (produit scalaire de deux vecteurs, méthode prod) et à `3 * v` (produit d'un vecteur par un scalaire, méthode rprod).
- `__sub__(self)`, `__truediv__(self)`, `__floordiv__(self)`, `__mod__(self)`, `__pow__(self` `__lshift__(self)`, `__rshift__(self)` `__and__(self)`, `__or__(self)`, `__xor__(self)` `-` (opérateur binaire), `/`, `//`, `%`, `**`, `«`, `»`, `&`, `|`, `^`
- `__neg__(self)`, `__pos__(self)`, `__invert__(self)` opérateurs unaires `-` et `+`, `~`
- `__iadd__(self)`, `__imul__(self)`, ... pour les affectations augmentées `+=`, `*=` etc.
- `__iter__(self)`, `__next__(self)` pour gérer les boucles
- `__call__(self)` lorsque l'objet est utilisé comme une fonction.
- `__getitem__(self)`, `__setitem__(self)`, `__delitem__(self)` pour utiliser l'objet comme un conteneur.

II Créer une classe des fractions

Commençons par rappeler comment on calcule efficacement le pgcd de deux entiers (positifs ou négatifs) à l'aide de l'algorithme d'Euclide :

```
def pgcd(a,b):
    a,b=abs(a),abs(b)
    while b!=0:
        a,b=b,a%b
    return a
```

Voici alors une classe de fractions qui va nous permettre de calculer comme avec les entiers à l'intérieur même des programmes. Nous allons expliquer pas à pas la construction de cette classe qui vous paraitra alors beaucoup moins mystérieuse.

```
class fraction(object):
    def __init__(self, numerateur, denominateur=1):
        if denominateur < 0:
            numerateur, denominateur = -numerateur, -denominateur
        elif denominateur == 0:
            raise ZeroDivisionError(\
            'Le denominateur ne doit pas etre nul')
        d=pgcd(numerateur, denominateur)
        self.num = int(numerateur // d)
        self.den = int(denominateur // d)
    def __repr__(self):
        if self.den != 1:
            return "%d/%d"%(self.num, self.den)
        else:
            return str(self.num)
    def binary_deco(methode):
        def nouvelle_methode(s, o):
            if isinstance(o, int):
                o = fraction(o)
            if isinstance(s, int):
                s = fraction(s)
            return methode(s, o)
        return nouvelle_methode
    @binary_deco
    def __add__(s,o):
        return fraction(s.num*o.den + o.num*s.den, s.den*o.den)
    __radd__ = __add__
    @binary_deco
    def __mul__(self, other):
        return fraction(self.num*other.num, self.den*other.den)
    __rmul__ = __mul__
    def __pow__(self, other):
        return fraction(self.num**other, self.den**other)
    @binary_deco
    def __truediv__(self, other):
        return fraction(self.num*other.den, self.den*other.num)
    @binary_deco
    def __sub__(self, other):
        return fraction(self.num*other.den-other.num*self.den, \
        self.den*other.den)
    __rsub__ = __sub__
    def __neg__(self):
```

```
        return fraction(-self.num, self.den)
    @binary_deco
    def __eq__(self,other):
        return self.num*other.den == self.den*other.num
    __req__ = __eq__
    @binary_deco
    def __gt__(self,other):
        return self.num*other.den > self.den*other.num
    __rgt__ = __gt__
    @binary_deco
    def __lt__(self,other):
        return self.num*other.den < self.den*other.num
    __rlt__ = __lt__
    @binary_deco
    def __ge__(self, other):
        return self.num*other.den >= self.den*other.num
    __rge__ = __ge__
    @binary_deco
    def __le__(self,other):
        return self.num*other.den <= self.den*other.num
    __rle__ = __le__
    def __float__(self):
        return self.num / self.den
```

Donnons un exemple d'utilisation. `x=fraction(1,3)`, `2+x**4`.

Exercice 1 : Dans la même idée, créer une classe de polynômes.

Exercice 2 : Créer une classe de nombres modulaires. Par exemple `a=modulo(3,26)` serait la classe des nombres entiers égaux à 3 modulo 26. Le modulo pourrait être mis par défaut égal à 26. La puissance modulaire est déjà un exercice en soi.

III Classe des décimaux

```
class decimal(object):
    def normalise(a,b):
        while a !=0 and a%10 == 0:
            a = a // 10
            b += 1
        return a,b
    def __init__(self,x):
        alph = set('0123456789.-e')
        s = str(x)
        if not(set(str(x)) <= alph):
            raise TypeError('Format de nombre non reconnu')
        if 'e' in s:
            L = s.split('e')
            self.virgule = int(L[1])
            s = str(L[0])
        else:
```

```
            self.virgule = 0
        if '.' in s:
            L = s.split('.')
            self.entier = int(L[0] + L[1])
            self.virgule -= len(L[1])
        else:
            self.entier = int(s)
        self.entier, self.virgule = \
        decimal.normalise(self.entier, self.virgule)
    def __repr__(self):
        if self.virgule >= 0:
            return str(self.entier)+'0'*self.virgule
        else:
            if self.entier < 0:
                signe = "-"
                s = str(-self.entier)
            else:
                signe = ""
                s = str(self.entier)
            if len(s) > -self.virgule:
                return signe + s[:self.virgule] + '.' + s[self.virgule:]
            else:
                return signe+'0.'+'0'*(-len(s) - self.virgule)+s
    def binary_deco(methode):
        def nouvelle_methode(s,o):
            if type(o) != decimal:
                o = decimal(o)
            if type(s) != decimal:
                s = decimal(s)
            return methode(s,o)
        return nouvelle_methode
    @binary_deco
    def __mul__(self,other):
        a = self.entier * other.entier
        b = self.virgule + other.virgule
        return decimal(str(a) + 'e' + str(b))
    __rmul__ = __mul__
    @binary_deco
    def __add__(self, other):
        m = min(self.virgule, other.virgule)
        a = self.entier * 10**(self.virgule - m) + \
        other.entier * 10**(other.virgule-m)
        b = m
        return decimal(str(a) + 'e' + str(b))
    __radd__ = __add__
    def __pow__(self, other):
        return decimal(str(self.entier ** other)+\
        'e'+str(self.virgule*other))
    @binary_deco
    def __sub__(self,other):
```

```
        a,b = -other.entier, other.virgule
        return self + decimal(str(a)+'e'+str(b))
    __rsub__=__sub__
    def __neg__(self):
        a,b = -other.entier, other.virgule
        return decimal(str(a)+'e'+str(b))
    @binary_deco
    def __eq__(self, other):
        return self.entier == other.entier and \
        self.virgule == other.virgule
    __req__ = __eq__
    @binary_deco
    def __gt__(self, other):
        m = min(self.virgule, other.virgule)
        return self.entier * 10**(self.virgule-m) > \
        other.entier * 10**(other.virgule-m)
    __rgt__=__gt__
    @binary_deco
    def __lt__(self,other):
        m = min(self.virgule, other.virgule)
        return self.entier * 10**(self.virgule-m) < \
        other.entier * 10**(other.virgule-m)
    __rlt__=__lt__
    @binary_deco
    def __ge__(self,other):
        m = min(self.virgule, other.virgule)
        return self.entier * 10**(self.virgule-m) >= \
        other.entier * 10**(other.virgule-m)
    __rge__=__ge__
    @binary_deco
    def __le__(self,other):
        m = min(self.virgule, other.virgule)
        return self.entier * 10**(self.virgule-m) <= \
        other.entier * 10**(other.virgule-m)
    __rle__=__le__
    def __float__(self):
        return float(str(self))
```

IV Création d'une classe pour faire le produit cartésien d'ensembles (ou listes) finis

A noter que le module Itertools fournit des outils pour faire cela. On va néanmoins montrer comment le faire avec les classes.
On va appeler `ens` cette classe.
Par exemple, si `r=ens([1,2,3,4,5,6])` (liste des résultats possibles d'un dé à 6 faces), alors on voudrait pouvoir écrire `len([i for i in r**12 if sum(r)==45])` pour compter le nombre de possibilités (équiprobables) lorsqu'on jette 12 dés où la somme est égale à 45.

```
def prod(l1,l2):
    #lorsque les elements de l1 ou l2
```

```
    # ne sont pas des tuples, on fabrique
    # des tuples
    if len(l1)>0 and type(l1[0])==tuple:
        if len(l2)>0 and type(l2[0])!=tuple:
            return [ i+(j,) for i in l1 for j in l2]
        else:
            return [ i+j for i in l1 for j in l2]

    elif len(l2)>0 and type(l2[0])==tuple:
        return [ (i,)+j for i in l1 for j in l2]
    else:
        return [(i,j) for i in l1 for j in l2]

class ens(object):
    def __init__(self,iterable):
        self.value=iterable
    def __repr__(self):
        return str(self.value)
    def __mul__(self,other):
        return ens(prod(self.value,other.value))
    def __pow__(self,n):
        if n==0:
            return ens([])
        l=self.value
        for i in range(n-1):
            l=prod(l,self.value)
        return ens(l)
    def __iter__(self):
        for i in self.value:
            yield i
```

V Casse-têtes japonais

```
from random import *
def grille_alea(taille):
    C = [i for i in range(1,taille**2+1)]
    L=[]
    for i in range(taille):
        L.append([])
        for j in range(taille):
            L[-1].append(C.pop(randrange(len(C))))
    return L

def colonne(g, i):
    return [g[j][i] for j in range(len(g))]
def produits(grille):
    n = len(grille)
    prodL = [prod(ligne) for ligne in grille]
    prodC =[prod(colonne(grille,j)) for j in range(n)]
```

```
    return [prodL, prodC]

def resoud(g, p):
    n = len(g)
    L = []
    for ligne in g:
        L += ligne
    S = {i for i in range(1,n*n+1)} - set(L)
    L = list(S)
    return resoud2(g, p, L, (0, 0))

def suivant(i, j, n):
    j += 1
    if j == n:
        i += 1
        j = 0
    return (i,j)

def copy(g):
    res = []
    for ligne in g:
        res.append(ligne[:])
    return res

def resoud2(g, p, L, pos):
    i, j = pos
    if L == []:
        return([copy(g)])
    n = len(g)
    res = []
    while i<n and g[i][j] != None:
        i,j = suivant(i,j,n)
    if i == n:
        print('impossible')
        return []
    for c in L:
        g[i][j] = c
        if (p[0][i] == None or None in g[i] \
        or prod(g[i]) == p[0][i])\
        and (p[1][j] == None or None in colonne(g,j) \
        or prod(colonne(g,j)) == p[1][j]):
            L1 = L[:]
            L1.remove(c)
            res += resoud2(g, p, L1,suivant(i,j,n))
    g[i][j] = None
    return res

def probleme(taille, classique=True):
    '''fabrique une liste de probleme a partir d une grille
    tiree au hasard. On remplace une a une les cases
```

```
    par None.
    Classique = True pour d abord retirer les chiffres'''
    g = grille_alea(taille)
    p = produits(g)
    IJ = [(a % (taille +1), a // (taille +1)) for a in range((taille+1)**2-1
    while len(resoud(g,p)) == 1:
        print(g,p)
        (i,j) = IJ[randrange(len(IJ))]
        while classique and taille in (i,j):
            (i,j) = IJ[randrange(len(IJ))]
        IJ.remove((i,j))
        if len(IJ) == taille * 2:
            classique = False
        if i == taille:
            p[0][j] = None
        elif j == taille:
            p[1][i] = None
        else:
            g[i][j] = None
```

VI Classe de fonctions

Dans l'idée de pouvoir demander les variations, les intersections de la courbe avec les axes ou avec la courbe d'une autre fonction, le minimum, le maximum.

Par exemple, pour une fonction trinôme. Pouvoir écrire `f=trinome(a, b , c)`, puis écrire `g = f + f` ou bien `f.min` etc.

Nécessite une classe générale des fonctions, puis des classes héritées : polynôme, trinôme, affine, linéaire, constante ...

VII Classe de variable aléatoire

Dans le même état d'esprit que pour les fonctions (à creuser).

Chapitre 10

Projets

I Bataille d'algorithmes

Programmer un module permettant à des robots de s'affronter suivant des règles à définir. Chaque joueur aurait la possibilité de configurer son robot.

Exemples de jeux.

1. Mini-Yams : un jeu à deux joueurs, on tire au sort qui commence. Chacun son tour on lance un dé et on choisit de placer le résultat sur une des 3 cases. La première case a une valeur multipliée par 3, la deuxième par 2 et la dernière par 1. Le plus gros résultat gagne.
2. Jeu de Singapour : un jeu à deux joueurs, on tire au sort qui commence. Chacun son tour on lance un dé et on choisit de placer le résultat sur une des 3 cases qui représentent le chiffre des centaines, des dizaines et des unités. Le plus grand nombre gagne.
3. Jeu stop ou encore. Chaque joueur joue à tour de rôle. Il s'agit d'être le premier à dépasser un nombre fixé au départ. Par exemple le nombre 100. On part avec 0 (on peut éventuellement mettre un handicap au premier joueur). On lance alors un dé autant de fois que l'on veut à condition de ne jamais faire 6 (excès de vitesse). Lorsqu'on décide d'arrêter, on ajoute tous les nombres obtenus à notre capital. Le premier qui atteint ou dépasse 100 a gagné.

Définir la représentation d'une position. Dans la position il faut également inclure, si le jeu l'impose, l'ordre des joueurs.
Par exemple, pour le premier ou le deuxième jeu : la position peut être une liste contenant deux listes de 3 éléments : `[[5,0,0], [0,3,0]]`, la liste de gauche serait la position du joueur dont c'est le tour, et les zéros représentent les cases vides. L'inconvénient de prendre des listes est que ces objets sont modifiables, et donc une certaine façon de tricher est à surveiller.
D'autre part il ne faut pas oublier de rajouter à la position le tirage du dé, qui ne peut en aucun cas être fait par le joueur (triche trop facile). La position serait donc `[[5,0,0], [0,3,0], 4]` si le tirage du dé est 4. Pour le dernier jeu, une position peut être tout simplement un triplet d'entiers avec, à gauche l'entier correspondant au joueur dont c'est le tour, au milieu la position du second joueur et à droite la valeur cumulée des dés par exemple `(96, 98, 12)`.

II Simulation d'un vol d'oiseaux

III Gérer des chorégraphies de danses de salons

Chapitre 11

Processing avec Python : informatique et arts

Table des matières

1 Introduction **1**
- I Pourquoi Python ? . . . 1
  - 1) Continuité par rapport au collège . . . 1
  - 2) Atouts et inconvénients du langage . . . 1
- II Exemples . . . 2
  - 1) Programmation fonctionnelle . . . 2
  - 2) Exemple de compréhension : crible d'Eratosthène . . . 3
  - 3) Exemple d'utilisation de Jupyter : Jake Vanderplas, Python Data Science Handbook . . . 3
- III Installation . . . 3

2 Python comme une calculatrice **5**
- I Valeurs renvoyées et affichage . . . 5
- II Les différents types de nombres . . . 5
  - 1) Nombres entiers, variables . . . 6
  - 2) Nombres flottants . . . 7
  - 3) Nombres complexes . . . 7
  - 4) Conversion de nombres . . . 7
- III Booléens . . . 8
- IV Les conteneurs . . . 8
  - 1) Les conteneurs indexés . . . 8
  - 2) Les conteneurs non indexés . . . 11
  - 3) Conteneurs définis par extension . . . 11
  - 4) Les fonctions et méthodes spécifiques aux listes . . . 12
  - 5) Méthodes spécifiques aux chaînes de caractères . . . 13
- V Le type 'type' et la conversion de types . . . 13
  - 1) Le type 'type' . . . 13
  - 2) Conversion de types . . . 13
- VI Exercices . . . 14

3 Formatages des entrées et des sorties **19**
- I Formatage des sorties . . . 19
- II Gestion des entrées . . . 20

4 Fonctions et portée des variables **21**
- I Introduction et syntaxe . . . 21
  - 1) Pourquoi écrire des fonctions ? . . . 21
  - 2) Syntaxe et écriture des fonctions en Python . . . 22
- II Premières fonctions . . . 22
- III Portée des variables . . . 23
- IV Conditionner l'entrée dans une fonction . . . 24

V Fonctionnalités avancées . . . 24
1) Fonction dans une fonction . . . 25
2) Nombre quelconque d'arguments . . . 25

5 Ecrire des programmes **27**
I Structure des programmes . . . 27
II Les tests . . . 27
III Les boucles . . . 28
1) Boucle **for** . . . 28
2) Boucle **while** . . . 28

6 Réaliser des graphiques **29**
I Matplotlib . . . 29
II Module turtle . . . 30

7 Modules MpMath, NumPy et SciPy **31**
I Module MpMath . . . 31
II Modules numpy et scipy . . . 31
1) Module stats de scipy . . . 31

8 Pour aller plus loin **33**
I Les décorateurs . . . 33
II Les itérateurs/générateurs . . . 33
III Gestion des erreurs . . . 33

9 Programmation orientée objet **35**
I Opérateurs de classe . . . 35
II Créer une classe des fractions . . . 35
III Classe des décimaux . . . 37
IV Création d'une classe pour faire le produit cartésien d'ensembles (ou listes) finis 39
V Casse-têtes japonais . . . 40
VI Classe de fonctions . . . 42
VII Classe de variable aléatoire . . . 42

10 Projets **43**
I Bataille d'algorithmes . . . 43
II Simulation d'un vol d'oiseaux . . . 43
III Gérer des chorégraphies de danses de salons . . . 43

11 Processing avec Python : informatique et arts **45**