Cahier d'algorithmique et de programmation

Table des matières

Quelques consignes

Ce document contient bien plus que ce que vous devrez savoir faire en Python d'ici la fin de l'année donc pas de panique si vous ne comprenez pas tout !
Au cours de chaque séance, vous aurez des exercices à faire et à me rendre. Donc à chaque séance, vous compléterez un document open office avec :

- NOM Prénom – Classe
- pour chaque exercice :
  - Exercice . . .
  - le numéro des questions (s'il y a lieu) avec la réponse
  - si la réponse contient un programme (du code), vous devrez le formater dans une police à chasse fixe (la police Courrier par exemple).

Le tout sera à me rendre via l'ENT à cette adresse : http://marlioz.elycee.rhonealpes.fr/service/classes/classe-2d9/2d9-maths/blog.do (identification obligatoire).
Pour ceux qui préfèrent et qui ont un compte google, vous pouvez aussi créer un dossier VotreNom-Python et le partager avec moi à l'adresse reymarlioz-at-gmail.com. Dans ce dossier, vous rédigerez un document avec les mêmes consignes que ci-dessus.
Pour ceux qui n'arriveraient pas à installer Python chez eux, il existe une solution « en ligne » : écrire (et exécuter) son code sur la page https://trinket.io. Vous pouvez ensuite copier-coller votre code dans votre traitement de textes.
Enfin, pour ceux qui veulent aller plus loin, le site http://apprendre-python.com est assez complet.

« *Un programme informatique fait ce que vous lui avez dit de faire, pas ce que vous voulez qu'il fasse.* »
TROISIÈME LOI DE GREER

Chapitre 1

Introduction

Au cours de votre scolarité au lycée, conformément aux programmes officiels, vos profs de maths vous parleront souvent d'*algorithmes*; ne prenez pas peur, il s'agit simplement d'une façon « savante » de parler d'une sorte de « recette de cuisine mathématique » ...
Dans ce document, nous allons parler d'algorithmes mais aussi de programmation informatique. Les programmes de maths recommandent de coder ces algorithmes dans un langage informatique respectant les critères suivants :
Un langage de programmation simple d'usage est nécessaire pour l'écriture des programmes. Le choix du langage se fera parmi les langages interprétés, concis, largement répandus, et pouvant fonctionner dans une diversité d'environnements.
Au lycée Marlioz, nous avons choisi de vous faire programmer en langage Python.

1.1 Algorithmes

Définition 1.1 (et Histoire)
Le mot *algorithme* vient du nom de l'auteur persan AL-KHUWARIZMI (né vers 780 - mort vers 850) qui a écrit en langue arabe le plus ancien traité d'algèbre « abrégé de calcul par la complétion et la simplification [a] » dans lequel il décrivait des procédés de calcul à suivre étape par étape pour résoudre des problèmes ramenés à des équations [b].
On définit parfois les algorithmes de la manière suivante : « un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat et cela indépendamment des données ». Le résultat doit donc s'obtenir en un temps fini.

a. Le mot arabe utilisé pour nommer la complétion ou restauration se lit Al-Jabr, ce qui donna naissance à notre mot algèbre.
b. Notamment, les équations du second degré.

Dans ce document, les algorithmes seront toujours présentés comme dans l'exemple 1.1 ci-après. Il ne s'agit que d'un exemple de présentation, libre au lecteur d'en trouver d'autres s'il le souhaite...

Exemple 1.1
Un exemple de présentation : le (faux) algorithme 1 – ou algorithme savoyard (nous expliquerons plus loin pourquoi « faux »).

ENTRÉE : 1,5 kg de pommes de terre;
1 oignon, du sel, du poivre;
120 g de lard fumé 1 reblochon fermier;
10 cl de vin blanc sec et 10 cl de lait;
DÉBUT
    Préchauffer le four à 180 °C;
    Éplucher l'oignon et le tailler dans le lard fumé;
    Faire sauter le mélange oignon - lard fumé pendant 5 min et ajouter ensuite le vin;
    Laver, éplucher et tailler en rondelle les pommes de terre;
    Les mettre dans une casserole, recouvrir d'eau froide, ajouter du gros sel et cuire 5min après ébullition. Égoutter;
    TANT QUE *il reste des pomme de terre* **FAIRE**
        Disposer une couche de pomme de terre;
        Disposer une couche de lardons - oignons;
        Saler poivrer;
    Tailler le reblochon en tranches et les disposer dessus;
    Verser un peu de lait et mettre au four pendant 30 min;
    SI *ce n'est pas assez gratiné* **ALORS**
        Laisser dans le four 5 min. de plus;
RÉSULTATS : Bon appétit !

Algorithme 1 : (faux) Algorithme savoyard - d'après https://www.atelierdeschefs.fr/fr/recette/2275-tartiflette-au-reblochon.php

Remarque 1.1

Attention : il ne faut pas confondre algorithme (mathématique ou informatique) et recette de cuisine ! Dans une recette de cuisine, on a (beaucoup) de liberté : mettre 130 g de lard fumé à la place de 120 g dans la recette précédente ne changera pas fondamentalement le résultat alors que dans un (vrai) algorithme ça peut tout changer !
Dans la suite, on assimilera les « ENTRÉES » de l'algorithme aux « ingrédients » d'une recette, le « DÉBUT » au traitement à effectuer sur les « ENTRÉES » et les « RÉSULTATS » à ce qu'on obtient à la fin.

Exemple 1.2

L'algorithme 2 est un véritable algorithme que vous avez (peut-être) vu en troisième et qui permet de déterminer le pgcd de deux entiers positifs non nuls.
Appliquons cet algorithme aux nombres 1 245 et 420. Pour cela nous allons compléter un tableau dans lequel on va inscrire la valeur de a, de b et de r au cours du déroulement de cet algorithme :

Étape	a	b	r
1	1 245	420	405
2	420	405	15
3	405	15	0

Finalement, le pgcd de 1 245 et 420 vaut 15.

```
1 ENTRÉE : deux nombres entiers strictement positifs notés a et b avec a ≥ b;
2 DÉBUT
3     Calculer le reste r de la division de a par b;
4     TANT QUE r > 0 FAIRE
5         a prend la valeur de b;
6         b prend la valeur de r;
7         r prend pour valeur le reste de la division de a par b;
8 RÉSULTATS : Le pgcd est b
```

Algorithme 2 : Algorithme d'EUCLIDE

1.2 Le langage Python

Dans une première lecture et pour les novices en programmation, ce paragraphe (à part les liens vers l'installation du logiciel) peut être « survolé ».

Un langage informatique permet de donner des instructions simples à l'ordinateur. Il en existe énormément.

Le principe de la programmation est d'écrire un *programme* ou *code* sous la forme d'un texte qui respecte la syntaxe du langage choisi et ensuite d'exécuter ce programme (plus ou moins directement).

On peut distinguer deux types de langages :

les langages interprétés : les lignes du code sont exécutées sucessivement sans tenir compte, a priori, des suivantes (c'est « confortable » car le programme ne « plante » que lorsqu'il rencontre une erreur, mais moins rapide que du langage « machine ») ;

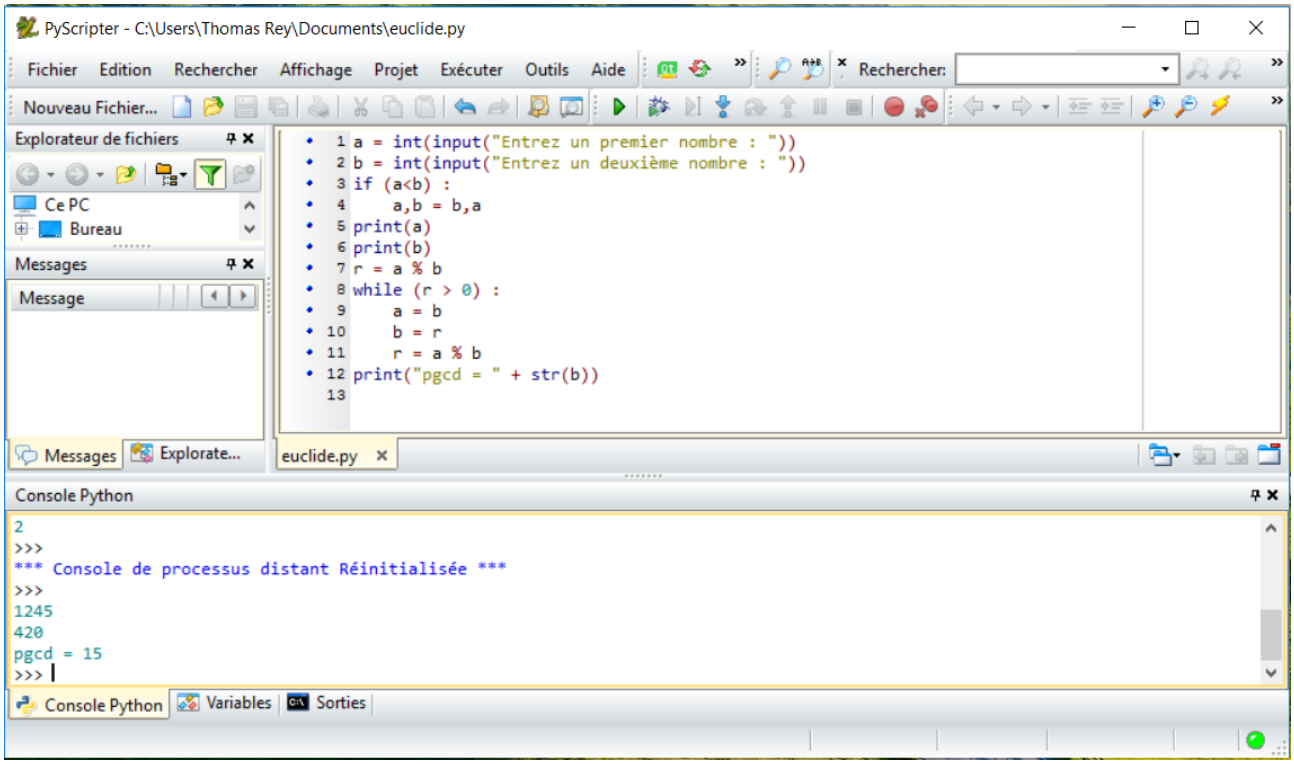
les langages compilés : le code du programme est d'abord transformé entièrement en langage « machine » (on doit « compiler » le programme) et ensuite seulement on peut l'exécuter (s'il y a une erreur, il ne sera pas compilé et donc on ne pourra même pas exécuter la partie du programme qui précède l'erreur).

Dans les deux cas, il faut donc utiliser un programme externe qui nous permet de voir le résultat de notre code : un *interpréteur* dans le premier cas, un *compilateur* dans le second. Ce programme externe peut être assimilé aux couteaux, poële, four, ... nécessaire à l'algorithme 1 savoyard vu précédemment.

Python est un langage *interprété* qu'on peut utiliser à la fois sur les ordinateurs PC munis de Windows ou de Linux, mais aussi sous Mac OSX. Au lycée, nous utiliserons un *environnement de développement* (une application qui permet à la fois d'écrire le code du programme, de l'interpréter et de voir le résultat) appelé EduPython disponible gratuitement ici : https://edupython.tuxfamily.org. Il n'est disponible que sous Windows, mais les utilisateurs [1] de Mac ou de Linux peuvent utiliser Python IDLE disponible ici : https://www.python.org/downloads/ qui fonctionne à peu près sur le même principe.

La programmation de l'algorithme 2 d'EUCLIDE vu précédemment dans EduPython est illustrée sur la copie d'écran suivante :

1. Ces utilisateurs peuvent aussi utiliser EduPython par l'intermédiaire de Wine (pour les experts...).



Enfin, pour terminer cette introduction, rappelons que le but de la programmation en cours de maths n'est pas de faire de vous des experts (sinon, vous auriez dû choisir l'enseignement d'exploration ICN !) mais de vous permettre :

- de prendre conscience de la puissance de calcul d'un ordinateur ;
- d'obtenir facilement des résultats numériques qu'il aurait été difficile d'obtenir à la calculatrice ;
- de comprendre la distinction entre une langue (parlée ou écrite) et un langage informatique ;
- de susciter des vocations ?
- ...

1.3 Quelques exercices

Dans ce document, vous aurez à réaliser des exercices. Lorsque vous devrez écrire un programme, il faudra le *commenter* ; c'est-à-dire écrire dans le code des indications sur ce que vous écrivez (à quoi sert une boucle, à quoi correspond une variable, ...). Pour indiquer à l'interpréteur Python qu'on écrit un commentaire, on le précède du symbole hashtag « # ». Ainsi, sur une ligne, tout ce qui suit un # n'est pas lu par Python.

Exercice 1.1

Installer un environnement de développement sur votre ordinateur familial :

- EduPython si vous avez un PC sous Windows ;
- Python IDLE si vous avez un ordinateur MAc ou PC (Windows ou Linux).

Exercice 1.2

Lancer EduPython et récupérer le programme de l'algorithme d'EUCLIDE dans l'ENT et le commenter. Tester son fonctionnement.

Exercice 1.3

Que fait le programme suivant :

```
1 # Programme sans intérêt
2 # print("mon âge est 15 ans")
3 # fin
```

Remarque 1.2

En informatique, chaque caractère est codé par un nombre. Par exemple le « a » a le code 64, l'espace a le code 32, ...
Il existe cependant plusieurs codages différents pour les lettres accentuées. En France, deux codages principaux sont utilisés : l'UTF-8 et l'iso-latin-1. Dans ces deux codages, les lettres non accentuées ont le même code mais pas les autres symboles ni les lettres accentuées.
Pour indiquer à Python le codage utilisé par votre ordinateur, il est conseillé de commencer tous vos programmes par l'une des deux lignes suivantes :

```
1 # -*- coding: Latin-1 -*-
```

ou

```
1 # -*- coding: Utf-8 -*-
```

Il suffit d'en tester une, si l'interpréteur Python détecte une erreur à chaque lettre accentuée, c'est qu'il faut utiliser l'autre !

« *Le succès est un mauvais professeur. Il pousse les gens intelligents à croire qu'ils sont infaillibles* »
BILL GATES

Chapitre 2

Les variables en informatique

Si on se souvient de l'algorithme 1, on va encore pouvoir faire une analogie avec la cuisine...
En effet, lorsqu'on commence une recette, on prépare les ingrédients et on va les « stocker » dans des récipients différents : un bol pour le lait, une passoire pour les pommes de terre lavées, ... En informatique, il en est de même, les informations utiles au fonctionnement d'un algorithme ou d'un programme sont stockées dans des *variables* qui ne sont pas toutes du même *type* et à qui on doit donner des *noms*.

2.1 Variable

Définition 2.1

Une *variable* est une espace de la mémoire qui contient une (ou plusieurs) information(s). Un variable possède :

un identificateur : c'est son *nom* ; par exemple `a`, `age`, `maVariable`, ...

un type : nombre entier, nombre décimal, chaine de caractères, ... (en Python, on ne précise pas le type lorsqu'on définit une variable) ;

un contenu : ce que contient l'espace en mémoire.

Remarque 2.1

Le nom d'une variable est une suite de caractères (lettres non accentuées, chiffres et le caractère _) dont le premier est *obligatoirement*[a] une lettre minuscule.

a. En fait, une variable pourrait commencer par une lettre majuscule, mais ce n'est pas l'usage. On réserve les « mots » commençant par une lettre majuscule aux *classes* ; mais ceci dépasse le cours de maths de seconde...

Exercice 2.1

Parmi les propositions suivantes, barrer celles qui ne peuvent être des noms de variables :

`a` | `aB` | `a23` | `a%` | `5ps` | `a-b` | `z^k` | `aa_dd` | `âge`

Remarque 2.2

Si dans un bol on verse 10 cl de lait et qu'après réflexion on rajoute 50 % de pommes de terres, il faudra ajouter 5 cl de lait dans le bol. La *variable* `bol` contenait la valeur 10, elle contient désormais 15. Une fois qu'on aura versé le lait dans le plat, la variable `bol` contiendra la valeur 0.
Finalement, une *variable* s'appelle judicieusement ainsi car son contenu peut *varier* !

2.1.1 Types

Dans l'introduction de ce chapitre on a dit qu'on versait le lait dans un bol et les pommes de terre dans une passoire. Si on essaye de faire le contraire, le résultat risque de poser des problèmes...
En informatique c'est parfois la même chose (dans certains langages). Lorsqu'une variable est définie pour contenir des nombres, on ne peut pas par la suite y mettre des lettres. On dit que les variables ont un *type* : il s'agit des informations qu'elles peuvent contenir.
En Python, le *typage* des variables est *dynamique* : une variable qui contenait initialement des nombres peut ensuite contenir du texte. Tout se passe comme si le bol s'aggrandissait lorsqu'on veut y mettre les 2 kg de pommes de terres ou si les trous de la passoire se bouchaient lorsqu'on verse du lait dedans !
Il peut être utile de connaître le type contenu dans une variable pour cela on utilise l'instruction suivante : `type(nomDeLaVariable)`.

Remarque 2.3

Lorsqu'une variable est d'un certain type, il est possible de la convertir en un autre type :

```
1 a = "5" # la variable a est du type String :
2 print(type(a))
3 b = int(a) # la variable b est du type int : on peut
     effectuer des calculs avec...
4 age = 25 # age est de type int
5 phrase = "Mon âge est " + str(age) + " ans"
6 # on a converti age en string pour faire une phrase...
```

2.1.2 Affectations

L'*affectation* consiste à donner une (nouvelle) valeur à une variable. En Python, elle se fait grâce au symbole =.
Dans le programme ci-dessous, on donne quelques exemples d'affectations. À vous de le tester.
Remarques préalables :

- l'instruction `print` permet d'afficher dans la « console » le contenu d'une variable ;
- tout ce qui suit le caractère dièse (#) n'est pas lu : il s'agit de commentaires.

Exemple 2.1

Quelques affectations :

```
1 a = 5
2 b = 3
```

```
3  c = a + b / a  # Priorités opératoires respectées !
4  print(c)
5  msg = "Bonjour, "
6  nom = "Maurice"
7  print(msg + nom) # opération de "concaténation" : on écrit
       les contenus à la suite
8  nom = "Albert"
9  print(msg + nom)
10 c = 2**10     # permet de calculer "2 puissance 10"
11 print(c)
12 a = 1250 % 27 # permet de calculer le reste de la division
       de 1250 par 27
13 print(a)
14 a, b = 5, 3   # "double affectation"
15 print(a, b)   # affichage de deux variables
16 b, a = a, b   # échange des contenus de a et b
17 print(a, b)
```

Remarque 2.4

Attention : le « = » de l'affectation n'a pas la même signification que l'égalité mathématique. On peut le traduire par « prend la valeur suivante ».
Quelques exemples d'utilisations :

Affectation	signification	ex. de valeur avant	valeur après
`i=i+1`	augmenter `i` de 1	`i=5`	`i=6`
`i=i**2`	élever `i` au carré	`i=3`	`i=9`
`i=5*i+2`	multiplier `i` par 5 et ajouter 2	`i=3`	`i=17`

2.2 Quelques exercices

Exercice 2.2

On donne le programme suivant. Compléter le tableau de valeurs des variables pour chaque ligne du programme :

```
1 a = 3
2 b = 5
3 a = b + 7
4 b = a / 2
5 b = a % 5
```

ligne	a	b
1		
2		
3		
4		
5		

Exercice 2.3

Même exercice avec :

```
1 a, b = 3, 5
2 a = a + b
3 b = a - b
4 a = a - b
```

ligne	a	b
1		
2		
3		
4		

L'instruction `input` permet de demander à l'utilisateur de saisir une valeur. Elle s'utilise ainsi :

```
1 a = input("donner une valeur pour a :")
```

Attention, `input` renvoie une chaîne de caractères.

Exercice 2.4

Que fait le programme ci-dessous ?

```
1 a = input("donner une valeur pour a :")
2 b = a + a
3 print(b)
```

Vérifier en le codant. Expliquer. Corriger pour qu'il fasse ce qu'on attendait.

Exercice 2.5

On considère la fonction numérique f définie par $f(x) = 2x^2 - x - 3$.
Écrire un programme qui demande à l'utilisateur un nombre et qui renvoie l'image de ce nombre par la fonction f.

Exercice 2.6

On donne le programme Python suivant :

```
1 a = int(input("Saisir un nombre x :"))
2 b = a + 4
3 b = b * a
4 b = b + 4
5 print("f(x) = " + str(b))
```

1. Faire tourner ce programme « à la main » (sans ordinateur) avec comme valeurs saisies : 3, puis -2 et enfin 4.
2. Expliquer ce que fait ce programme (avec du vocabulaire mathématique...).

Exercice 2.7

La formule qui permet de calculer l'Indice de Masse Corporelle d'un individu est la suivante :

$$\text{IMC} = \frac{Masse}{Taille^2}$$

où la masse est exprimée en kg et la taille en mètres.
Écrire un programme qui demande la masse en kg et la taille en cm d'un individu et qui affiche son IMC.

Exercice 2.8

Écrire un programme qui demande successivement x_A, y_A, x_B et y_B (les coordonnées de deux points dans un repère orthonormé) et qui affiche :

- le carré de la distance entre ces points : AB^2 ;
- la distance entre ces points : AB (en valeur approchée si besoin) ;
- les coordonnées du milieu du segment [AB].

Exercice 2.9

Écrire un programme qui demande à l'utilisateur deux fractions $\frac{a}{b}$ et $\frac{c}{d}$ (sous la forme de deux fois deux entiers non nuls) et qui écrit le résultat (sous forme de fractions) de :

$$\frac{a}{b}+\frac{c}{d}; \qquad \frac{a}{b}-\frac{c}{d}; \qquad \frac{a}{b}\times\frac{c}{d}; \qquad \frac{a}{b}\div\frac{c}{d}$$

Les résultats ne seront pas nécessairement mis sous forme de fractions irréductibles (nous verrons ça plus tard dans l'exercice 3.9).

2.3 Compléments : les listes

Cette partie peut être laissée de côté pour une première lecture. Par contre, il pourra être utile d'y revenir pour la suite de ce cours...
Parfois, on a besoin de stocker plusieurs nombres en mémoire, et il peut être utile de les stocker dans une seule variable qu'on appelera alors *une liste*. Toujours pour faire une analogie culinaire, on peut voir une liste comme un plat à « compartiments » : par exemple une boîte à thé où on stocke plusieurs variétés de thés différents dans des cases différentes.

2.3.1 Définition et remplissage d'une liste

Définition 2.2

Une *liste* est une variable dont le contenu est délimité par des crochets, et dans les crochets, les contenus sont séparés par des virgules.

Pour ajouter un ou des élément(s) à une liste, on peut « additionner » des listes ou bien utiliser la *méthode* `append` :

```
# Les premières notes sont :
notes = [4, 12, 14, 15, 13]
# On obtient deux nouvelles notes : un 15 et un 20
notes = notes + [15, 20]
# on ajoute un 12 :
notes.append(12)
```

Exemple 2.2

Quelques exemples complémentaires dans le code ci-dessous :

```
# création d'une liste de dix "0" :
notes = [0]*10
a=[3,5,6,9,10,15]# nouvelle liste
# suppression de l'élément d'indice 2 (i.e : le "6")
del a[2]
```

2.3.2 Utilisation et modification

Remarque 2.5

À propos des listes :

- attention, les éléments d'une liste sont numérotés en partant de 0 ! (Et non pas en partant de 1) ;
- un élément est utilisé en écrivant le nom de la variable suivi de l'indice de l'élément voulu placé entre crochets ;
- pour modifier un élément d'une liste, on procède comme pour une variable : par affectation (voir exemple suivant).

Exemple 2.3

On donne le code suivant :

```
matieres=['EPS', 'Français', 'Maths']
print(matieres[1])
# notes obtenues ce trimestre :
notes = [5, 10, 14, 12, 18, 17]
print(notes[3])# on affiche ...
# Modifions une note :
notes[3] = notes[3] + 2
print(notes[3])
```

Qu'affiche ce programme ?

Remarque 2.6

En compléments, quelques utilisations des listes sont proposées ci-après.

(a) Pour connaître la longueur d'une liste (son nombre d'éléments) on utilise la fonction `len` :

```
a = ['rouge', 'vert', 'bleu']
nbre_de_coul = len(a)# ici, nbre_de_coul sera égal à 3
```

Attention, si `len(a) = 3`, les indices des éléments de a vont de 0 à 2.

(b) Une liste ne contient pas obligatoirement des éléments de même type :

```
mois = ['janvier', 31, 'février', 28, 'mars', 31, ...]
```

est une définition correcte (mais pas vraiment souhaitable)

(c) On peut trier les éléments d'une liste grâce à `sort` et `reverse`. Tester le programme suivant :

```
a = [2, 18, 4, 5, 31, 0, 13]
a.sort()# trier les éléments de a par ordre croissant
print(a)
a.reverse()# on change l'ordre
print(a)
```

(d) On peut effacer un élément connu :

```
a = [10, 45, 62, 59, 73, 14]
a.remove(45)
print(a)
```

(e) On peut supprimer des éléments connaissant leurs indices :

```
a = [4, 57, 85, 63, 21, 99]
del a[1]# on retire le '57'
# désormais a = [4, 85, 63, 21, 99]
del a[1:3]# on retire les éléments d'indices 1 et 2
print(a)# a = [4, 21, 99]
```

(f) On peut retrouver l'index d'un élément :

```
a = [41, 52, 63, 58, 69, 74, 19]
print(a.index(63))# affiche 2 (63 le 3e de la liste)
b = [10, 52, 62, 10, 45]
print(b.index(10))# affiche 0 : le 1er 10 est retenu
```

(g) On peut compter le nombre d'occurences d'une valeur :

```
a = [4, 6, 4, 7, 4, 6, 1, 10]
print(a.count(4))# Affiche 3
```

(h) On peut insérer un élement dans la liste à une position voulue :

```
1 a = [5, 6, 7, 8]
2 a.insert(2,10)
3 print(a)# a=[5, 6, 10, 7, 8]
```

Remarque 2.7

On peut résumer la remarque 2.6 par le tableau ci-dessous (sans exemple) où `list` et `L` sont deux listes, `i` est un entier et `x` est une variable de type quelconque :

Méthode	Effet
`list.append(x)`	ajoute l élément x en fin de liste
`list.extend(L)`	ajoute en fin de liste les éléments de L
`list.insert(i, x)`	insère un élément x en position i
`list.remove(x)`	supprime la première occurrence de x
`list.pop([i])`	supprime l élément d indice i et le renvoie
`list.index(x)`	renvoie l indice de la première occurrence de x
`list.count(x)`	renvoie le nombre d occurrences de x
`list.sort()`	modifie la liste en la triant
`list.reverse()`	modifie la liste en inversant l ordre des éléments

Chapitre 3

Instructions de base

Pour continuer notre analogie culinaire, il peut arriver que dans une recette on nous demande de plier cinq *fois* la pâte à tarte ou encore de tourner la cuillère *tant que* la sauce est liquide. Ces deux consignes sont appelées en informatique des *boucles*. La première est *bornée* (on sait combien de plis on va faire) la deuxième non (on ne sait pas combien de tours de cuillère seront nécessaires). Enfin, on peut aussi rencontrer une instruction comme « si les blancs ne montent pas, ajouter un peu de sel ». Ce type de consignes est appelé en informatique *instruction conditionnelle*.

3.1 Boucles

3.1.1 Boucle bornée

L'instruction Python permettant d'effectuer une boucle bornée est `for`. Pour l'utiliser, on aura souvent besoin de l'instruction `range` qui crée des listes de nombres :

Instruction	Description
`range(5)`	liste des entiers positifs ou nul strictement inférieurs à 5
`range(1,5)`	liste des entiers entre 1 et 4 (inclus)
`range(1,20,3)`	liste 1, 4, 7, 10, 13, 16, 19

Exemple 3.1

```
1 for i in range(1,10) :
2     print(i)
```

Ce code permet d'afficher la liste des entiers de ... à ... (inclus).

Remarque 3.1

En Python, c'est l'*indentation* (c'est-à-dire le décalage du texte sur la droite) qui définit un bloc d'instructions. Par exemple, pour exécuter deux (ou plus) instructions dans la boucle on écrit :

```
1 for i in range(10):
2     <instruction 1> # DANS la boucle
3     <instruction 2> # DANS la boucle
4 <instruction 3> #  PAS dans la boucle
```

Remarque 3.2

Pour ceux qui ont lu la partie 2.3 (les autres vous pouvez passer...), on peut effectuer une boucle sur n'importe quelle liste :

```
1 coul = ['bleu', 'blanc', 'rouge']
2 for c in coul:
3     print(c)
```

3.1.2 Boucle non bornée

L'instruction Python permettant d'effectuer une boucle non bornée est `while`. Elle est suivie d'un test (est-ce que la sauce est liquide ?) qui est effectué à chaque début de boucle.

Exemple 3.2

Le programme ci-après affiche la liste des multiples de 7 inférieurs ou égaux à 100 :

```
1 mult = 0
2 while  mult <= 100:# le multiple est-il <=100 ?
3     print(mult)
4     mult = mult + 7# multiple suivant
5 print("fin")
```

Remarque 3.3

Comme pour les boucles `for`, l'ensemble du bloc *indenté* est executé dans la boucle (voir la remarque 3.1).

3.2 Instructions conditionnelles

En Python, l'instruction qui permet d'effectuer un test et d'adapter la suite du programme au résultat du test (est-ce que les blancs d'œufs « montent » ?) est constituée du « couple » `if`, `else`. On appelle cela un *branchement conditionnel.*

Exemple 3.3

Le programme ci-dessous constitue le début d'une application de révision des tables de multiplication :

```
1 a = int(input("combien vaut 4 x 7 ? "))
2 if (a==28):
3     print("Bravo !")
4 else:
5     print("Revoir les tables !")
6 print("fin")
```

Remarque 3.4

Comme pour les boucles `for` et `while`, l'ensemble du bloc *indenté* est executé dans le *branchement conditionnel* (voir la remarque 3.1).
Le `else` est facultatif.
Le `else` peut être remplacé par un `elif` suivi d'une autre condition.

Remarque 3.5

L'instruction `if` permet aussi de savoir si un élément est dans une liste :

```
1 a = [12, 15, 14, 10, 5, 20]
2 if (7 in a):
3     print("Oui, tu as eu un 7")
4 else:
5     print("Non, tu n'as pas eu de 7")
```

3.3 Nombres aléatoires

Pour utiliser des nombres aléatoires, il faut utiliser une *bibliothèque*[1] supplémentaire : la bibliothèque `random`. C'est un peu comme si vous achetiez un nouveau livre de cuisine (ou, soyons numériques, que vous mettiez à jour votre application de recettes sur votre tablette). Pour cela, il faut écrire au début du programme l'instruction suivante :

```
1 import random
```

Ensuite l'instruction `random.random()` donne un nombre aléatoire compris entre 0 et 1.
L'instruction `random.randrange(m,M,p)` donne un nombre aléatoire compris entre `m` et `M` avec un écart de `p` entre chaque valeur. Le paramètre `p` est facultatif.

Exemple 3.4

Quelques exemples d'utilisation :

```
1 import random
2 a = random.random()*10 # nombre aléatoire décimal entre 0
      et 10
3 b = random.randrange(1,7) # nombre aléatoire entier entre 1
      et 6 (inclus)
4 c = random.randrange(0,10,2) # nombre aléatoire entier pair
      compris entre 0 et 9 (inclus)
```

1. En programmation, une bibliothèque est une collection d'instructions supplémentaires qui ne font pas partie des bases du langage.

3.4 Quelques exercices

Exercice 3.1

Écrire un programme qui demande un nombre entre 1 et 12 et qui affiche les treize premiers multiples sous la forme (par ex. pour la table de 5) :

```
0
5
10
...
```

Exercice 3.2

Même exercice mais en obtenant le résultat sous la forme :

```
0 x 5 = 0
1 x 5 = 5
2 x 5 = 10
...
```

On pourra utiliser l'instruction `str`.

Exercice 3.3

Écrire un programme qui demande un nombre jusqu'à ce que la réponse soit supérieure ou égale à 100.

Exercice 3.4

Soit f la fonction définie par :

$$f(x) = \begin{cases} 2x+1 & \text{si } x \geq -\frac{1}{2} \\ x+\frac{1}{2} & \text{si } x < -\frac{1}{2} \end{cases}$$

Écrire un programme qui demande une valeur de x, qui calcule $f(x)$ et qui l'affiche.

Exercice 3.5

Écrire un programme qui simule le lancer d'un dé jusqu'à obtenir un 6, qui affiche le résultat de chaque lancer et qui compte le nombre de lancers effectués.

Exercice 3.6

Écrire un programme qui détermine si deux vecteurs (donnés par leurs coordonnées) sont colinéaires.

Exercice 3.7

Écrire un programme qui détermine l'équation réduite d'une droite connaissant les coordonnées de deux points.

Exercice 3.8

Écrire un programme qui demande un nombre entier positif x et qui écrit $\sqrt{x}$ sous la forme $a\sqrt{b}$ avec a et b entiers et b le plus petit possible.

Exercice 3.9

Écrire un programme qui demande le numérateur et le dénominateur d'une fraction (nombres entiers) et qui donne la fraction irréductible correspondante. On pourra ensuite compléter le programme de l'exercice 2.9 fait précédemment.

Exercice 3.10

Écrire un programme qui dresse une table de valeurs de la fonction $f : x \mapsto \frac{1}{2}x^2 - x - 2$ pour les x entiers compris entre -5 et 5 inclus.

Exercice 3.11

Écrire un programme qui :
- demande un nombre entier n ;
- réalise ensuite l'algorithme 3

Ce procédé est appelé suite de Syracuse. La conjecture de Collatz affirme que pour tout nombre n choisi au départ, on aboutit à 1.
voir ici : http://cristal.univ-lille.fr/~jdelahay/pls/053.pdf pour plus d'informations.

```
1 DÉBUT
2     TANT QUE n > 1 FAIRE
3         SI n est pair ALORS
4             n ← n/2;
5         SINON
6             n ← 3 * n + 1;
```

Algorithme 3 : Algorithme de Syracuse

Exercice 3.12

Recherche du maximum approchée d'une fonction sur un intervalle.

« *Informatique : alliance d'une science inexacte et d'une activité humaine faillible* »
LUC FAYARD

Chapitre 4

Fonctions

Pour faire une tarte au fraises, il faut une pâte sablée, de la crème pâtissière et ... des fraises. Dans le livre de recettes, il est indiqué : *pour la pâte sablée, prévoir 250 g de farine, 125 g de beurre, 125 g de sucre, un œuf, un demi sachet de levure et suivre la recette page* ... et de même pour la crème pâtissière. On peut imaginer qu'une personne se charge de la pâte pendant qu'une autre fait la crème et une troisième cueille les fraises.
En informatique, lorsqu'on veut déléguer une partie du travail à un autre « morceau de programme » pour obtenir un résultat, on crée ce qu'on appelle une *fonction*. Une fois cette fonction créée, on va pouvoir l'*appeler* en lui passant les *arguments* (les ingrédients en cuisine) dont elle a besoin. De même qu'un fois qu'on a formé quelqu'un à faire une pâte sablée, il peut en faire plusieurs, des plus grandes, des plus petites, ...
Pour faire une autre analogie, parlons mathématiques... Une fonction numérique f permet d'associer à un nombre x un autre nombre qu'on note $f(x)$. En informatique, une fonction permet de donner à une variable une valeur qui dépend d'autre(s) variable(s).

4.1 Une fonction dans un algorithme

Nous voulons écrire un programme qui permet d'afficher le nouveau prix d'articles pendant les soldes en *fonction* du prix initial et du taux de réduction. Pour cela nous allons créer une fonction `nouveauPrix` qui va prendre *deux arguments* : `prix` et `reduc`.
Cette fonction est ensuite utilisée dans un programme *principal*.

```
1  FONCTION nouveauPrix(prix, reduc) EST DÉFINIE PAR :
2      np ← prix − prix × reduc /100;
3      RETURN np;
4  DÉBUT
5      Afficher "Ancien prix : 39, reduc : 15%, nouveauprix :";
6      Afficher nouveauPrix (39, 15);
7      Afficher "Ancien prix : 1259, reduc : 35%, nouveauprix :";
8      Afficher nouveauPrix (1259, 35);
9      p ← Demander "ancien prix ?";
10     r ← Demander "reduction (en %) ?";
11     Afficher "Nouveau prix : ", nouveauPrix (p,r);
```

Algorithme 4 : fonction nouveauPrix

Remarque 4.1

La fonction est définie par les lignes 1 à 3.
L'instruction RETURN permet de « renvoyer » à l'utilisateur de la fonction un résultat.
Elle peut être utilisée avec des nombres : lignes 6 et 8.
Elle peut aussi être utilisée avec des variables : ligne 11.

4.2 En Python

Observons le code Python correspondant à l'algorithme 4 :

```
1  def nouveauPrix(prix, reduc):
2      np = prix - prix * reduc / 100
3      return np
4
5  print("Ancien prix : 39, reduc : 15\%, nouveauprix :")
6  print(nouveauPrix(39,15))
7  print("Ancien prix : 1259, reduc : 35\%, nouveauprix :")
8  print(nouveauPrix(1259,35))
9
10 p = float(input("Donner l'ancien prix : "))
11 r = int(input("Donner la réduction (en %) : "))
12 print("nouveau prix = " + str(nouveauPrix(p,r)))
```

La fonction est définie par les lignes ... à ...
La ligne 6 permet d'afficher le résultat obtenu par la fonction avec pour paramètres 39 et 15.
Dans la ligne 12, on obtient le résultat obtenu par la fonction avec pour paramètres le contenu de la variable `p` et le contenu de la variable `r`.

Remarque 4.2

Python étant un langage *interprété* (voir page 9), à l'exécution du code, les lignes sont lues les unes après les autres. La définition d'une fonction doit donc intervenir *au début* du code (en tout cas, *avant* qu'on ne l'utilise).
Cette remarque n'est pas valable pour les langages *compilés* dans lesquels on peut définir toutes les fonctions à la fin du code.

Remarque 4.3

Attention, une fonction ne peut pas modifier le contenu d'une variable de type numérique ou chaîne de caractères du programme principal (on dit que ces types sont immuables). Par contre elle le peut pour une liste.

```
1 def fonc(a):
2     a = a + 1
3
4 a = 5
5 fonc(a)
6 print(a) # Affiche 5 !
```

```
1 def fonc(L):
2     n = len(L)
3     L[n-1] = 'fin'# modifie le dernier élément
4 L=[1,2,3,4]
5 fonc(L)
6 print(L)# affiche [1,2,3,'fin']
```

4.3 Quelques exercices

Exercice 4.1

1. Écrire une fonction qui prend un paramètre x et qui renvoie l'image de x par la fonction affine $f : x \mapsto 3x - 7$.
2. Utiliser cette fonction dans un programme qui demande un nombre et affiche l'image de ce nombre par la fonction f.
3. Utiliser la fonction de la question 1 dans un programme qui affiche un tableau de valeurs de la fonction f pour les x entiers compris entre -5 et 5.
4. Même question mais cette fois c'est l'utilisateur qui saisit les bornes de l'intervalle et le pas.

Exercice 4.2

Refaire l'exercice 3.5 en utilisant une fonction qui renvoie le nombre de lancers nécessaires pour obtenir un 6.
Utiliser cette fonction pour simuler 1 000 fois cette expérience et calculer le nombre moyen de lancers nécessaires pour obtenir un 6 sur ces 1 000 répétitions.
Remarque : en probabilité on parle de *loi* pour généraliser les résultats de plusieurs expériences aléatoires « ressemblantes ». Ici, nous venons d'étudier la loi *géométrique*.

Exercice 4.3

1. Écrire une fonction qui prend en paramètres les coordonnées de deux points (donc quatre paramètres) dans un repère orthonormé et qui renvoie la distance entre ces deux points.
2. Utiliser cette fonction dans un programme qui demande les coordonnées de trois points A, B et M et qui détermine si M appartient à la médiatrice de [AB].
3. Utiliser la fonction de la question 1 dans un programme qui demande les coordonnées de trois points et qui détermine si le triangle formé est :
   - rectangle ;
   - isocèle ;
   - équilatéral.

Exercice 4.4

Pour cet exercice, on écrira toutes les fonctions demandées dans un seul programme et on pourra utiliser une fonction déjà écrite dans la définition d'une autre.

1. Écrire une fonction nommée `coord_vect` qui prend pour paramètres `(xA,yA,xB,yB)` les coordonnées de deux points A et B et qui renvoie le couple de coordonnées du vecteur $\overrightarrow{AB}$.
2. Écrire une fonction nommée `colineaire` qui prend pour paramètres `(x1,y1,x2,y2)` les coordonnées de deux vecteurs et qui renvoie `True` ou `False` selon que les vecteurs sont colinéaires ou pas.
3. Écrire une fonction nommée `alignes` qui prend pour paramètres `(xA,yA,xB,yB,xC,yC)` et qui renvoie `True` ou `False` selon que les points sont alignés ou pas.
4. Même question (fonction nommée `parallele`) qui prend pour paramètres les coordonnées de quatre points et qui renvoie `True` ou `False` selon que les droites (AB) et (CD) sont parallèles ou non.
5. Utiliser ces fonctions pour vérifier vos résultats à toutes les questions du dernier DS pour lesquelles on parle de vecteurs colinéaires, points alignés, droites parallèles, ...

Exercice 4.5

Soit f une fonction affine définie par $f(x) = mx + p$. On sait que $f(a) = b$ et $f(c) = d$.

1. Écrire une fonction nommée `fonc_aff` qui prend pour paramètres quatre nombres `(a,b,c,d)` et qui renvoie les coefficients m et p de la fonction affine.
2. Tester cette fonction avec $f(2) = 5$ et $f(-3) = -2$.
3. Même question avec $f(-2) = 7$ et $f(-3) = 1$.
4. Modifier la fonction pour la question précédente ne produise pas d'erreur.

Chapitre 5

Dessiner à l'écran

La bibliothèque `matplotlib` fournie avec EduPython (et qu'on peut aussi installer si on n'utilise pas EduPython) permet de créer un graphique avec le repérage usuel en maths (repère orthonormé avec les axes orientés vers la droite et vers le haut).
Pour utiliser cette bibliothèque, il suffit de l'indiquer au début par l'instruction suivante.

```
1 import matplotlib.pyplot as repere
```

Désormais, un objet `repere` est créé et nous allons « dessiner » dedans.
Ensuite, on place les instructions détaillées ci-après et enfin, on termine par l'instruction `repere.show()` pour provoquer l'affichage de notre figure.

Remarque 5.1

Le nom `repere` est un exemple, vous pouvez appeler votre fenêtre `toto` ou par n'importe quel autre nom.

5.1 Les instructions graphiques

5.1.1 Un premier exemple commenté

Copier et tester l'exemple ci-dessous :

```
1  import matplotlib.pyplot as repere
2
3  # met une légende sur l'axe vertical (y)
4  repere.ylabel('Ordonnées')
5  # met une légende sur l'axe horizontal (x)
6  repere.xlabel('Abscisses')
7  # ligne brisée de sommets (0;2), (1;1), (2;5) et (3,3)
8  repere.plot([2,1,5,3])
9  # place un point (grace au '.') de coordonnées (2;1)
10 repere.plot(2,1,'.')
11 # triangle bleu en (3;1) :
12 repere.plot(3,1,'b^')
13 # affiche le résultat :
14 repere.show()
```

5.1.2 Styles

Dans l'exemple précédent, chaque instruction `plot` prend comme troisième argument une chaîne de caractère définissant un *style* de tracé. Ces styles sont constitués de l'initiale d'une couleur et d'une série de caractères (voir tableau ci-dessous) :

Couleurs		Styles	
b	bleu	–	ligne continue
g	vert	−−	tirets
r	rouge	:	pointillés
c	cyan	.	points
m	magenta	o	billes
y	jaune	x	croix
k	noir	v	triangles
w	blanc	−.	points-tirets

5.1.3 Quelques tracés plus élaborés

Un autre exemple à copier-modifier-tester :

```
1  import matplotlib.pyplot as repere
2
3  # une ligne passant par des points dont les coordonnées sont
       dans 2 listes :
4  x = [20,50,60,100]# les abscisses
5  y = [-10,5,30,15]# les ordonnées
6  # on ajoute l'épaisseur de la ligne, un "marqueur" à chaque
       point, une légende
7  repere.plot(x,y,'r--', linewidth=2, marker='+', label='Bénéfice
       ')
8  repere.legend()# sinon la légende n'apparaît pas
9
10 # affiche le résultat :
11 repere.show()
```

5.1.4 Légendes et textes

Quelques instructions pour illustrer votre graphique :

Instruction	Effet
`repere.clf()`	Efface la fenêtre graphique
`repere.axis(xmin, xmax, ymin, ymax)`	Impose l'échelle du graphique (comme sur vos calculatrices)
`repere.grid(booleen)`	affiche ou non une grille selon si `booleen` vaut `True` ou `False`
`repere.title(texte)`	Ajoute un titre
`repere.xlabel(texte)`	Ajoute la légende texte à l'axe des abscisses
`repere.ylabel(texte)`	Devine !
`repere.text(x,y,texte)`	place `texte` à la position $(x;y)$

Plus de détails sur cette page : http://apprendre-python.com/page-creer-graphiques-scientifique

5.2 Exercices

Pour certains de ces exercices, vous allez programmer des fonctions qui existent déjà (sûrement meilleures en plus :-(...) dans la bibliothèque `matplotlib`. Mais ce n'est pas grave, c'est quand même un bon exercice de les refaire !

Exercice 5.1

Écrire une fonction `segment` qui prend pour arguments les coordonnées de 2 points et une chaîne constituée de l'initiale d'une couleur et d'un style (voir le tableau de la page 32).
Cette fonction tracera le segment joignant les deux points dont on a donné les coordonnées en respectant la couleur et le style de la chaîne.

Exercice 5.2

Écrire une fonction `rectangle` qui prend pour arguments les coordonnées de 2 points et une chaîne constituée de l'initiale d'une couleur et d'un style (voir le tableau de la page 32).
Cette fonction tracera un rectangle aux côtés horizontaux et verticaux dont deux sommets opposés ont pour coordonnées celles passée en arguments. Il faudra bien-sûr respecté la chaîne de couleur et style.

Exercice 5.3

Même exercice que le précédent pour une fonction `rectangleP` qui, en plus du précédent remplit le rectangle avec la couleur proposée.

Exercice 5.4

Écrire une fonction `nuage` qui prend pour arguments deux listes et une chaîne constituée de l'initiale d'une couleur et d'un style (voir le tableau de la page 32).
Cette fonction construira un nuage de points en respectant la couleur et le style de la chaîne.
Par exemple avec les listes `x=[3,4,2]` et `y=[10,13,17]`, il faut placer trois points de coordonnées respectives (3;10), (4;13) et (2;17).

Exercice 5.5

Écrire une fonction qui prend pour paramètres deux listes et une chaîne constituée de l'initiale d'une couleur et d'un style (voir le tableau de la page 32).
Cette fonction tracera le diagramme en barres avec :

- les valeurs de la séries (hauteurs des barres) dans la première liste ;
- les légendes de chaque barre dans la deuxième liste ;
- la couleur et le style des barres dans la chaîne.

Exercice 5.6

Écrire un programme qui trace la courbe d'effectifs cumulés croissants d'une série donnée en paramètre sous la forme d'une liste.

« *La liberté informatique est un enjeu de société, et il est essentiel* »
RICHARD STALLMAN

Chapitre 6

Applications plus élaborées

Ce chapitre regroupe (pour le moment) des idées non développées d'applications de ce qui a été vu précédemment.

6.1 Recherche de solutions

Posons le problème : lorsqu'on doit résoudre un problème, on obtient souvent une équation dont on doit trouver la ou les solutions. Mais pour certaines équations, nous ne pouvons déterminer les solutions (parce que la méthode de résolution nous est inconnue ou même parcqu'il n'en existe pas...). Dans ce cas, on peut utiliser l'ordinateur pour en donner une solution *approchée*. L'objectif de cette partie est d'écrire des programmes permettant de trouver ces solutions approchées.
Supposons donc que nous devons trouver une solution à l'équation $f(x) = 0$ (où f est une fontion numérique monotone sur $[a;b]$ [1]) et que $f(a)$ et $f(b)$ sont de signes contraires. Dans cette situation, un théorème de mathématique nous dit qu'il existe une unique solution à l'équation $f(x) = 0$ sur l'intervalle $[a;b]$.
Nous allons déterminer une valeur approchée de cette solution par deux méthodes différentes.

6.1.1 Le balayage

6.1.2 La dichotomie

Exercice 6.1
Devine un nombre

1. l'ordi choisit un nombre et l'utilisateur cherche
2. l'utilisateur choisit et l'ordi cherche

Exercice 6.2
Résol équation

1. Rappel : une fonction est monotone sur un intervalle I si elle est soit croissante, soit décroissante sur cet intervalle.

6.2 Tracé point par point d'une courbe

6.3 Simulations d'expériences aléatoires

6.3.1 Échantillonnage

Pile ou face 200 fois répétés 50 fois Graphique des résultats (un point par répétition de coord $(i, nbpile)$)

6.3.2 Somme de deux dés

6.3.3 Le jeu du lièvre et de la tortue

6.3.4 méthode de Monte-Carlo pour approcher une aire

Approcher π

6.3.5 Planche de Galton

plus en Première

6.3.6 Paradoxe de Condorcet

Nous disposons de trois toupies. Chaque toupie est divisée en trois secteurs égaux. Chaque secteur porte un numéro. Chaque joueur dispose d une toupie ; il la fait tourner et relève, après l arrêt de la toupie, le nombre inscrit sur le secteur en contact avec la table.
- Le joueur A dispose d une toupie avec les nombres 1, 6 et 8.
- Le joueur B dispose d une toupie avec les nombres 2, 4 et 9.
- Le joueur C dispose d une toupie avec les nombres 3, 5 et 7

Les joueurs s affrontent deux à deux en faisant tourner chacun leur toupie ; le gagnant est celui qui obtient le nombre le plus grand après l arrêt des toupies. Quel est le joueur qui a le plus de chances de gagner ?

6.3.7 Paradoxe du Duc de Toscane

6.3.8 Le bon, la brute et le truand

Chapitre 7

Principales instructions et opérateurs

7.1 Instructions et fonctions

Dans ce paragraphe, on a listé par ordre alphabétique les instructions et fonctions utilisées dans ce document avec pour chacune d'elles une courte explication et des exemples.
Dans la description de chaque instruction ou fonction ce qui est entre crochets est facultatif.

`<nomListe>.append(<valeur>)`

Ajoute `<valeur>` comme dernier élément de la liste `nomListe`.

```
1 a = [3, 4, 6, 10]
2 a.append(50)
3 print(a)
4 # donne : [3, 4, 6, 10, 50]
```

`def <nomFonction>(<paramètres>):`

Permet de définir une nouvelle fonction. Voir le chapitre 4 page 27 pour plus de détails...

`del <nomListe[i]>`

Supprime l'élément de la liste indiqué.

```
1 a = [5, 3, 8, 5]
2 del(a[1])
3 print(a)
4 # affiche : [3, 8, 5]
```

float

Permet de convertir en nombre décimal un nombre qui était considéré comme un entier.

```
a = 5# Python "voit" a comme une variable entière
b = float(a)# Python "voit" b comme une variable float
print(a,b)# Affiche 5 5.0
```

for <variable> in ...:

L'instruction `for` permet de créer une boucle bornée. Tout ce qui suit le : et qui est *indenté* est répété.

```
# Afiicher les entiers de la liste donnée (2,4,6,10):
for i in [2, 4, 6, 10]:
    print(i)
# Additionner les entiers de 1 à 9 :
s = 0
for i in range(1,10):
    s = s + i
    print("somme provisoire = "+str(s))
```

if <condition> : [elif <condition> :] [else:]

Le bloc d'instructions conditionnelles `if`, `elif`, `else` permet de tester une condition et d'exécuter un code différent suivant le résultat du test.
Les blocs `elif` et `else` sont facultatifs.

```
a = int(input("saisir un entier : "))
if a%2 == 1:
    print("le nombre est impair")
elif a%2 == 4:
    print("le nombre est divisible par 4 (donc pair)")
else :
    print("le nombre est pair et non divisible par 4")
```

input(["texte"])

La fonction `input()` renvoie une chaîne de caractères saisie par l'utilisateur du programme.
On peut indiquer dans les parenthèses une chaîne de caractères (la « question » à poser).

```
a = input()
nom = input("Quel est ton nom ? ")
age = int("Quel est ton âge ? ")
```

`import <bibliotheque>`

Permet d'importer une bibliothèque contenant des instructions supplémentaires que celles comprises dans le langage de base Python.

```
1 import lycee# importe les instructions supplémentaires d'
    EduPython
2 # OU BIEN :
3 from lycee import *# On importe tous les modules de la
    bibliothèque lycee
```

Attention, il faut que la bibliothèque soit installée sur l'ordinateur ou bien qu'elle soit dans le même répertoire que le fichier sur lequel on travaille.

`<val> in <nomListe>`

Renvoie un booléen (Vrai/Faux) qui permet de savoir si la valeur `val` appartient à la liste `nomListe`.

```
1 a = [4, 5, 6]
2 print(5 in a)# Affiche True
3 print(3 in a)# Affiche False
```

`int(<expression>)`

La fonction `int()` prend un paramètre et le convertit en entier.
Attention, dans le cas où le paramètre est un `float`, il renvoie la partie avant la virgule (et non pas l'arrondi).

```
1 as = "23"# chaine de caractères
2 an = int(as)# nombre entier
3 p = 3.1415# nombre décimal (float)
4 vapi = int(p)# nombre entier (3)
5 int(2.71)# donne 2
6 int(-3.7)# donne -3
```

`len(<nomListe>)`

Donne le nombre d'éléments de la liste `nomListe`.

```
1 a = [5, 8, 9, 3]
2 print(len(a)) # Affiche 4
```

`len(<chaine>)`

Affiche la longueur de la chaîne `chaine`.

```
1 a = "Cours de maths"
2 print(len(a))# affiche 14
```

`print([<expression>])`

L'instruction `print` permet d'afficher un texte ou le contenu d'une ou plusieurs variables dans la console.

```
1 print("texte à afficher")
2 print(a,b)
3 print("la distance vaut : " + str(dist))
```

`random - randrange`

Permettent de donner des nombres « aléatoires ». Il faut importer la bibliothèque `random`.

```
1 from random import *
2 a = randrange(1,4)# nombre aléa entier entre 1 et 3 inclus
3 b = random()# nbre aléa entre 0 (inclus) et 1 (exclus)
```

`range([debut,] fin[,pas] )`

La fonction `range` renvoie une plage de valeurs sous la forme d'une liste. Cette fonction est très utile pour déterminer les valeurs que prend la variable d'une instruction `for` (voir page 38).

```
1 a = range(1,5)# a est une variable "list"
2 print(list(a))# affiche les valeurs de la liste a (ici :
    1,2,3,4)
```

Attention ceci ne fonctionne que pour un `pas` de type entier.

`return <expression>`

Permet de renvoyer un résultat à la fin d'une fonction (voir le chapitre 4 page 27).

`<nomListe>.remove(<val>) <expression>`

Retire le premier élément de la liste égal à `val`. Attention cette instruction renvoie une erreur si l'élément `val` n'est pas présnt dans la liste. Il peut être intéressant de vérifier avant grâce à l'instruction `in` (voir page 39 ou l'exemple ci-dessous).

```
1 a = [5, 7, 8, 2, 5, 9, 5, 6]
2 # Retirons TOUS les 5 de cette liste :
3 while 5 in a:
4     a.remove(5)
5 print(a)# Affiche [7, 8, 2, 9, 6]
```

`<nomListe>.reverse()`

Permet d'inverser l'ordre de la liste `nomListe`.

```
1 a = [4, 6, 8]
2 a.reverse()
3 print(a)# Affiche [8, 6, 4]
```

`<nomListe>.sort()`

Trie la liste `nomListe` par ordre croissant (si valeurs numériques) ou alphabétique (si chaînes de caractères).

```
1 a = [6, 4, 8, 2, 15]
2 a.sort()
3 print(a)# Affiche [2, 4, 6, 8, 15]
```

`math.sqrt(<expression>)`

Renvoie la racine carrée de `expression`. Attention, il faut avoir chargé la bibliothèque math avec une instruction `import math`.

`str(<expression>)`

Permet de convertir un nombre en chaîne de caractères.

`type(<variable>)`

Permet de connaître le type d'une variable.

`while(<condition>) :`

Permet de créer une boucle non définie (voir page 22)

7.2 Les opérateurs

Un *opérateur* est une opération qu'on peut effectuer sur une ou plusieurs variables.

Sur les variables numériques : Toutes les opérations usuelles sont possibles
- addition (`+`), soustraction (`-`), multiplication (`*`), division décimale (`/`) ;
- test d'égalité (`==`) qui revoie `True` ou `False` ;
- reste de la division euclidienne (`%`) ;
- quotient de la division euclidienne (`//`) ;
- puissance (`**`) ;
- comparaisons (`<`, `<=`, `>`, `>=`, `==`, `!=`) ; le dernier signifie différent.

Sur les chaînes de caractères : la principale opération est la concaténation (`+`) qui consiste à mettre « bout à bout » les deux chaînes.

Sur les conditions : il existe trois opérateurs principaux.
- pour tester si deux conditions sont vérifiées on utilise l'opérateur ET (`and`) ;
- pour tester si une des deux conditions est vérifiée, on utilise l'opérateur OU (`or`) ;
- pour tester le contraire d'une condition, on utilise l'opérateur NON (`not`).