

**INF101 - INF131**  
**Algorithmique et programmation en Python**  
Responsable d'UE : Carole Adam

*Polycopié conçu à partir du cours INF204 2016 créé par:  
Lydie du Bousquet, Aurélie Lagoutte, Julie Peyre, Florent Bouchez  
Tichadou, Amir Charif, Grégoire Cattan, Emna Mhiri*

Septembre 2019



# Table des matières

<b>1</b>	<b>Résumé de cours</b>	<b>9</b>
1.1	Quelques définitions . . . . .	9
1.1.1	Programmes, compilateur et interprète . . . . .	9
1.1.2	Algorithme vs Programme . . . . .	9
1.1.3	En Python . . . . .	10
1.2	Variables, expressions, instructions . . . . .	11
1.2.1	Variables . . . . .	11
1.2.2	Identificateur . . . . .	11
1.2.3	Affectation, initialisation . . . . .	11
1.2.4	Types . . . . .	12
1.2.5	Expressions et opérateurs . . . . .	13
1.2.6	Instructions . . . . .	15
1.2.7	Les commentaires . . . . .	15
1.2.8	Outils utiles . . . . .	15
1.3	Entrées / Sorties . . . . .	16
1.3.1	Les entrées . . . . .	16
1.3.2	Les sorties . . . . .	16
1.4	Les expressions booléennes . . . . .	18
1.4.1	Définition : expression booléenne . . . . .	18
1.4.2	Opérateurs booléens . . . . .	18
1.4.3	Élément neutre des opérateurs booléens . . . . .	18
1.4.4	Évaluation fainéante . . . . .	18
1.4.5	Lois de De Morgan . . . . .	19
1.4.6	Tables de vérité . . . . .	19
1.5	Instructions conditionnelles . . . . .	20
1.5.1	Syntaxe basique . . . . .	20
1.5.2	Syntaxe si-alors-sinon . . . . .	20
1.5.3	Cas multiples . . . . .	20
1.5.4	Cas par défaut . . . . .	21
1.5.5	Imbrication . . . . .	22
1.5.6	Nombres aléatoires . . . . .	22
1.6	La boucle 'while' . . . . .	23
1.6.1	While pour filtrer . . . . .	23
1.6.2	While pour rejouer un programme . . . . .	24
1.6.3	While pour répéter, compteur de boucle . . . . .	25
1.6.4	Accumulateurs et drapeaux . . . . .	25
1.6.5	Boucle infinie, Break, Continue . . . . .	26
1.6.6	Imbrication de boucles . . . . .	30
1.7	Les fonctions . . . . .	31
1.7.1	Principe d'une fonction . . . . .	31
1.7.2	Définition d'une fonction . . . . .	32
1.7.3	Appel d'une fonction . . . . .	33
1.7.4	Différence entre valeur de retour et effets de bord . . . . .	35
1.7.5	Portée des variables . . . . .	36

1.7.6	Importation de modules . . . . .	39
1.8	Chaînes de caractères . . . . .	40
1.8.1	Caractères . . . . .	40
1.8.2	Opérateurs sur les chaînes . . . . .	41
1.8.3	Fonctions de manipulation de chaînes . . . . .	42
1.8.4	Parcours de chaînes (nécessite le cours sur les boucles) . . . . .	44
1.9	Listes . . . . .	45
1.9.1	Type liste en Python . . . . .	45
1.9.2	Numérotation des éléments et accès . . . . .	45
1.9.3	Recherche dans une liste . . . . .	46
1.9.4	Affichage . . . . .	47
1.9.5	Modifier et ajouter des éléments . . . . .	48
1.9.6	Supprimer des éléments . . . . .	49
1.9.7	Copie et clone de listes . . . . .	50
1.9.8	Effets de bord avec les listes . . . . .	51
1.10	Boucle inconditionnelle : for . . . . .	53
1.10.1	Boucles conditionnelles vs inconditionnelles . . . . .	53
1.10.2	Boucle for pour répéter n fois, fonction range() . . . . .	53
1.10.3	Parcours de structures itérables . . . . .	54
1.10.4	Très important . . . . .	55
1.11	Fonctions : compléments . . . . .	56
1.11.1	Le mot-clé None . . . . .	56
1.11.2	Fonction avec plusieurs valeurs de retour . . . . .	56
1.11.3	Fonction avec des paramètres optionnels . . . . .	57
1.11.4	Utilisation de Docstring . . . . .	59
1.12	Dictionnaires . . . . .	61
1.12.1	Création . . . . .	61
1.12.2	Utilisation d'un dictionnaire . . . . .	62
1.12.3	Modification d'un dictionnaire . . . . .	62
1.12.4	Parcourir un dictionnaire . . . . .	63
1.12.5	Un exemple détaillé . . . . .	64
1.12.6	Copie de dictionnaires . . . . .	64
1.13	Introduction aux fichiers . . . . .	65
1.13.1	Ouverture d'un fichier texte . . . . .	65
1.13.2	Lecture d'un fichier texte . . . . .	65
1.13.3	Écriture dans un fichier texte . . . . .	66
1.13.4	Fermeture . . . . .	67
<b>2</b>	<b>Exercices de TD</b>	<b>69</b>
2.1	TD1 : les bases . . . . .	69
2.1.1	Échange des valeurs de deux variables . . . . .	69
2.1.2	Affichages . . . . .	69
2.1.3	Entrées, sorties, calculs . . . . .	70
2.1.4	Bonjour . . . . .	70
2.2	TD2 : instruction conditionnelle, booléens . . . . .	71
2.2.1	Un peu de logique . . . . .	71
2.2.2	Expressions booléennes . . . . .	71
2.2.3	Négations d'expressions booléennes . . . . .	71
2.2.4	Comparaison de deux programmes . . . . .	72
2.2.5	Comparaison de deux programmes (bis) . . . . .	72
2.2.6	Bonjour (v2) . . . . .	72
2.2.7	Encore des expressions booléennes . . . . .	72
2.2.8	Ou exclusif . . . . .	73
2.2.9	Maximum de trois nombres . . . . .	73
2.2.10	Année bissextile . . . . .	73
2.2.11	Jours, heures, minutes, secondes . . . . .	73

2.2.12	Quiz . . . . .	73
2.2.13	Multiples . . . . .	74
2.2.14	Dé pipé . . . . .	75
2.3	TD3 : itération conditionnelle while . . . . .	76
2.3.1	Code à trous . . . . .	76
2.3.2	Lecture de code . . . . .	76
2.3.3	Fizz Buzz . . . . .	76
2.3.4	Dé pipé (le retour) . . . . .	77
2.3.5	Voyelle . . . . .	77
2.3.6	Mot de passe . . . . .	77
2.3.7	Accumulateurs simples . . . . .	77
2.3.8	Moyenne, minimum, maximum . . . . .	78
2.3.9	Factorielle . . . . .	78
2.3.10	Séquence croissante . . . . .	78
2.3.11	Nombres premiers (simple) . . . . .	78
2.3.12	Dé pipé : multi-parties . . . . .	79
2.3.13	Suite de Fibonacci . . . . .	79
2.3.14	Syracuse . . . . .	79
2.3.15	Escaliers . . . . .	80
2.3.16	Triangles d'entiers . . . . .	80
2.3.17	Moyenne de classe . . . . .	80
2.3.18	Pyramide ailée . . . . .	80
2.3.19	Nombres premiers (bis) . . . . .	81
2.4	TD4 : chaînes de caractères . . . . .	82
2.4.1	Pyramide alphabétique . . . . .	82
2.4.2	Lecture de mots . . . . .	82
2.4.3	Caractère alphabétique . . . . .	82
2.4.4	Minuscule ou majuscule ? . . . . .	82
2.4.5	Position dans l'alphabet . . . . .	83
2.4.6	Décalage . . . . .	83
2.4.7	Inversion de casse . . . . .	83
2.5	TD5 : fonctions . . . . .	84
2.5.1	Comprendre les fonctions déjà connues . . . . .	84
2.5.2	Lecture . . . . .	84
2.5.3	Calcul de polynome . . . . .	85
2.5.4	Dates . . . . .	85
2.5.5	Somme des chiffres . . . . .	85
2.5.6	Conversions binaire-décimal . . . . .	86
2.5.7	Maximum . . . . .	86
2.5.8	Moyenne pondérée . . . . .	86
2.5.9	Ajoutez des lettres . . . . .	87
2.5.10	Dé . . . . .	87
2.5.11	Factorielle qui dépasse . . . . .	87
2.5.12	Calculatrice . . . . .	87
2.6	TD6 : listes . . . . .	90
2.6.1	Exercice : code à trous . . . . .	90
2.6.2	Création de liste . . . . .	90
2.6.3	Affichage de liste . . . . .	90
2.6.4	Test d'appartenance . . . . .	91
2.6.5	Indice d'apparition . . . . .	91
2.6.6	Comptage d'élément . . . . .	91
2.6.7	Recherche de minimum ou maximum . . . . .	91
2.6.8	Calcul de somme et moyenne . . . . .	91
2.6.9	Diviseurs, nombres premiers . . . . .	91
2.6.10	Jeu de dé et statistiques . . . . .	92

2.6.11	Codage par décalage . . . . .	92
2.6.12	Comptage alphabétique (force brute vs subtile) . . . . .	92
2.6.13	Liste de mots . . . . .	93
2.6.14	Insertion par recherche dichotomique . . . . .	94
2.6.15	Jouons avec des listes triées . . . . .	94
2.6.16	Cartes bancaires et formule de Luhn . . . . .	95
2.6.17	Jeu de démineur (listes de listes) . . . . .	96
2.7	TD7 : itération inconditionnelle for . . . . .	99
2.7.1	Exercices de lecture . . . . .	99
2.7.2	Les bases des boucles for . . . . .	99
2.7.3	Sommes d'entiers . . . . .	100
2.7.4	Suite croissante . . . . .	100
2.7.5	Comptons les moutons . . . . .	100
2.7.6	Règle graduée . . . . .	100
2.7.7	Accumulateurs . . . . .	101
2.7.8	Pile ou face, et statistiques . . . . .	101
2.7.9	Modification de chaînes . . . . .	101
2.7.10	Palindromes . . . . .	102
2.7.11	Zip . . . . .	102
2.7.12	Copie et inversion de listes . . . . .	103
2.7.13	Tri par sélection . . . . .	103
2.7.14	Diviseurs propres et nombres parfaits . . . . .	103
2.7.15	Nombre de Armstrong . . . . .	104
2.7.16	Approximation de la racine carrée . . . . .	104
2.7.17	Approximation de e . . . . .	105
2.7.18	Approximation de l'intégrale d'une fonction . . . . .	105
2.8	TD8 : fonctions avancées . . . . .	106
2.8.1	Affichage d'une liste . . . . .	106
2.8.2	Sauvons la planète... . . . .	106
2.8.3	Le menu . . . . .	106
2.8.4	Les moustiques . . . . .	106
2.8.5	Triangle de Pascal . . . . .	107
2.8.6	Liste d'entiers . . . . .	107
2.8.7	Codage et décodage de texte . . . . .	108
2.8.8	Intervalle avec paramètre optionnel . . . . .	108
2.8.9	Tables de multiplication . . . . .	108
2.8.10	Géométrie . . . . .	109
2.8.11	Nombres rationnels . . . . .	110
2.8.12	Tableau périodique des éléments . . . . .	111
2.9	TD9 : dictionnaires . . . . .	112
2.9.1	Les bases : dictionnaire à clés numériques . . . . .	112
2.9.2	Dictionnaire d'anniversaires . . . . .	112
2.9.3	Annuaire des plaques d'immatriculation . . . . .	113
2.9.4	Personnaliser ses pokémons . . . . .	113
2.9.5	Dictionnaire français-anglais . . . . .	113
2.9.6	Polynômes . . . . .	114
2.9.7	Gestion des notes (simple) . . . . .	114
2.9.8	Gestion des notes d'une classe (bis) . . . . .	115
2.10	Gestion des notes (ter) . . . . .	115
2.10.1	Transformer un dictionnaire en liste triée . . . . .	116
2.10.2	Rendez-vous . . . . .	116
2.11	TD10 : fichiers . . . . .	118
2.11.1	Groupe d'étudiants . . . . .	118
2.11.2	Groupe d'étudiants (bis) . . . . .	118
2.11.3	Analyse de texte . . . . .	118

2.11.4	Table harmonieuse . . . . .	118
2.11.5	Analyse de mesures . . . . .	119
<b>3</b>	<b>Sujets de TP</b>	<b>121</b>
3.1	TP1 : prise en main des outils, lecture de programmes, premiers programmes . . . . .	121
3.1.1	Première prise en main de Idle . . . . .	121
3.1.2	Ecriture et exécution d'un programme sous Idle . . . . .	122
3.1.3	Découverte de Python Tutor . . . . .	123
3.1.4	Exercice de lecture . . . . .	124
3.1.5	Quelques exercices . . . . .	124
3.2	TP2 : importation de modules, nombres aléatoires, boucles while . . . . .	127
3.2.1	Les modules Python . . . . .	127
3.2.2	Prix du gros lot . . . . .	128
3.3	TP3 : Initiation aux fonctions avec turtle . . . . .	131
3.3.1	Marchons au pas de <code>turtle</code> . . . . .	131
3.3.2	A la découverte des fonctions à travers le dessin . . . . .	133
3.3.3	Pour aller plus loin . . . . .	134
3.4	TP 4 : création et manipulation de fonctions, basiques . . . . .	136
3.4.1	Exercices d'observation . . . . .	136
3.4.2	Des petites fonctions . . . . .	137
3.5	TP5 : Manipulation de fonctions (suite), dessin avec Turtle . . . . .	139
3.5.1	Retour aux fonctions numériques . . . . .	139
3.5.2	Turtle . . . . .	140
3.5.3	On travaille en ligne! . . . . .	141
3.6	TP6 : listes . . . . .	142
3.6.1	Consultation de listes . . . . .	142
3.6.2	Avec modifications! . . . . .	143
3.6.3	Exercices sur les fonctions et chaînes de caractères . . . . .	144
3.7	TP7 : retour sur les listes . . . . .	146
3.7.1	Recyclage et Tri sélectif . . . . .	146
3.7.2	Le retour des courses (difficile) . . . . .	146
3.7.3	Course à pied . . . . .	146
3.7.4	Graphiques scientifiques . . . . .	147
3.8	TP semaine 8 : propagation d'une nouvelle . . . . .	149
3.8.1	Contexte du problème . . . . .	149
3.8.2	Exercice préliminaire . . . . .	149
3.8.3	Exercice 1 . . . . .	150
3.8.4	Exercice 2 . . . . .	151
3.8.5	Exercice 3 . . . . .	151
3.8.6	Exercice 4 . . . . .	151
3.8.7	Bonus : visualisation avec matplotlib . . . . .	151
3.9	TP semaine 9 : dictionnaires et traduction . . . . .	152
3.9.1	Exercice préliminaire : manipulation de dictionnaire français-anglais . . . . .	152
3.9.2	Exercice 2 : mini-jeu de traduction . . . . .	152
3.9.3	Exercice 3 : multi-joueurs . . . . .	153
3.9.4	Exercice 4 : traduction de texte . . . . .	153
3.9.5	Exercice 5 : multi-lingue . . . . .	154
3.10	TP 10 : modélisation de population . . . . .	155
3.10.1	Natalité . . . . .	155
3.10.2	Mortalité . . . . .	155
3.10.3	Renards . . . . .	155
3.10.4	Pour aller plus loin . . . . .	156
3.11	TP9 : fichiers et Python Tutor . . . . .	157
3.11.1	Un peu de PythonTutor . . . . .	157
3.11.2	Chaînes de caractères - Compléments . . . . .	157
3.11.3	Travailler avec des fichiers texte - Premiers pas . . . . .	157

3.11.4	Champion de scrabble . . . . .	158
<b>4</b>	<b>Annales</b>	<b>161</b>
<b>5</b>	<b>Annexes</b>	<b>185</b>



# Chapitre 1

## Résumé de cours

### 1.1 Quelques définitions

#### 1.1.1 Programmes, compilateur et interprète

Les ordinateurs permettent d'automatiser des tâches. Mais ce sont des machines sans capacité d'initiative. Un ordinateur fait ce qu'on lui dit de faire. Pour cela, il faut lui parler dans un langage qu'il "comprend". On appelle ça, un **programme**.

De façon générale et naïve, un programme est une suite d'instructions (de commandes) respectant une syntaxe précise, qui sont exécutées de façon séquentielle (c'est-à-dire les unes après les autres). Ces instructions sont élémentaires : des affectations, des instructions conditionnelles, des répétitions (itérations).

Il existe de nombreux langages dans lesquels on peut exprimer des programmes : C, Java... Dans la suite du cours, on utilisera un langage appelé Python.

Une fois que l'on a écrit un programme (c'est un texte écrit dans une syntaxe très précise), on utilise un autre programme qui transforme le texte de ce programme (lisible par des humains) en un texte lisible par la machine (suite d'octets). Ce programme peut être un compilateur ou un interpréteur, en fonction du langage de programmation choisi.

- Le **compilateur** traduit une bonne fois pour toutes un code source en un fichier indépendant exécutable (donc utilisant du code machine ou du code d'assemblage). Par exemple C est un langage compilé.
- L'**interpréteur** est nécessaire à chaque lancement du programme interprété, pour traduire au fur et à mesure le code source en code machine. Python est un langage interprété.

#### 1.1.2 Algorithme vs Programme

Comme dit précédemment, un programme est créé pour demander à l'ordinateur de faire une suite d'actions, en général pour résoudre un problème. Le programme (=le texte) va être lu et traduit par un compilateur ou un interpréteur, qui sont eux-mêmes des programmes. Compilateurs et interpréteurs attendent en entrée un texte sans fautes "d'orthographe" (*syntax error*) et qui résout le problème attendu. C'est parfois compliqué d'écrire directement un texte juste : qui résolve le problème et qui soit juste syntaxiquement (= sans faute d'orthographe). Rappelez-vous vos sujets de dissertations de philo, par exemple.

L'astuce quand on a un problème un peu compliqué est de travailler en 2 temps.

1. D'abord, on essaye de résoudre le problème (en s'autorisant des fautes d'orthographe, des approximations, des abréviations, en s'aidant de schémas), comme quand on rédige le plan de la dissertation avant de rédiger le texte final. C'est ce qu'on appelle un **algorithme**, et c'est indépendant du langage de programmation choisi.
2. Ensuite, on s'occupe de rédiger la solution finale au problème, en respectant la syntaxe exacte d'un langage. La même solution pourra ainsi être écrite dans différents langages de programmation. C'est ce qu'on appelle un **programme**, il n'est compréhensible que par le compilateur ou l'interpréteur du langage de programmation choisi, et il permet de faire faire ce qui est demandé à l'ordinateur.

On trouve des algorithmes partout dans la vie courante. Par exemple, en cuisine, un algorithme s'appelle une recette. En chimie ou en biologie, un algorithme s'appelle un protocole d'expérimentation. On peut aussi donner l'exemple qui consiste à indiquer le chemin à quelqu'un, qui peut se faire en plusieurs langues différentes, et selon plusieurs itinéraires distincts.

**Exemple :**

Donner l'algorithme pour faire une omelette (un seul oeuf). Préciser : les ingrédients nécessaires, les actions à mener (dans l'ordre).

Algo Omelette\_Elémentaire

Début

{ingrédients}

1 oeuf, sel poivre, beurre

{ustensiles}

1 saladier, une fourchette, une poêle, une spatule

{procédure}

Casser l'oeuf dans un saladier

Saler et poivrer

Battre l'oeuf à la fourchette

Dans une poêle, faire chauffer le beurre,

Verser l'oeuf battu dans la poêle,

Cuire doucement jusqu'à l'obtention de la texture souhaitée

{baveuse à bien cuite}

Servir

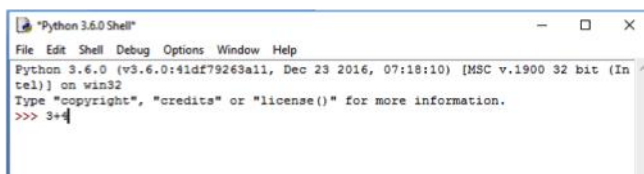
Fin

Dans une recette de cuisine, on a généralement les ingrédients et les ustensiles nécessaires, la liste des actions à mener et parfois des commentaires. On retrouve les mêmes éléments dans les programmes et les algorithmes. On parle de déclarations (pour indiquer ce dont on va avoir besoin), d'instructions (les commandes) et de commentaires (informations pour l'humain qui va lire ou relire le programme).

### 1.1.3 En Python

Python est un langage interprété. On disposera de deux modes d'exécution d'un code Python :

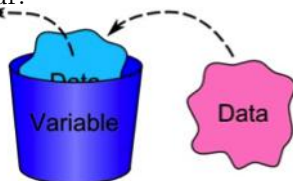
- soit on écrit un ensemble d'instructions dans un fichier puis on l'exécute
- soit on utilise l'interpréteur Python, instruction par instruction, un peu à la façon d'une calculatrice.



## 1.2 Variables, expressions, instructions

### 1.2.1 Variables

Une **variable** est un conteneur d'information qui est identifié par son nom, c'est un endroit pour ranger une valeur.



### 1.2.2 Identificateur

Pour nommer une variable, on utilise un **identificateur** : c'est une suite non vide de caractères respectant un certain nombre de contraintes :

- Doit commencer par un lettre ou le caractère `_`
- Doit contenir seulement des lettres, des chiffres et/ou le caractère `_`
- Ne doit pas être un mot-clé réservé de Python

**Attention** : les identificateurs sont sensibles à la casse, c'est-à-dire que les caractères majuscules et minuscules sont considérés comme différents (ex : `ma_var`  $\neq$  `Ma_Var`).

**Exemples :**

- d'identificateurs valides : `toto`, `prochaine_val`, `max1`, `MA_VALEUR`, `r2d2`, `bb8`, `_mavar`
- d'identificateurs non valides : `2be`, `C-3PO` (le tiret est considéré comme un opérateur 'moins'), `ma var` (l'espace sépare 2 identificateurs distincts)

**Conventions** **Par convention**, pour les variables en Python, on utilise des minuscules. On s'interdira également d'utiliser des accents.

**Standard** : le standard PEP8<sup>1</sup> liste d'autres conventions et bonnes pratiques stylistiques en Python (cohérence de nommage des variables et fonctions, longueur des lignes, style des commentaires, etc).

### 1.2.3 Affectation, initialisation

**Affectation.** Pour mémoriser une valeur dans une variable, on fait une **affectation**, en utilisant le signe `=` (qui est ici très différent de celui utilisé en maths). Dans une affectation le membre de gauche (identificateur) reçoit la valeur qui est à droite (qui doit **d'abord** être évaluée). C'est-à-dire que cette valeur (une fois calculée si nécessaire) est stockée dans cette variable.

```
n = 33
a = 42 + 2 * 5
ch = "bonjour"
euro = 6.55957
```

**Initialisation.** La première affectation d'une variable donnée s'appelle l'**initialisation** de la variable. Comme son nom l'indique, la valeur d'une **variable** peut varier ensuite au cours de l'exécution d'un programme. La valeur antérieure est perdue, remplacée par la nouvelle valeur.

1. Voir ici : <https://www.python.org/dev/peps/pep-0008/#naming-conventions>

```

>>> a = 3 * 7    # l'expression est évaluée, et sa valeur affectée dans a
>>> a
21
>>> b = 7.3      # le séparateur décimal est un point . et non une virgule ,
>>> b
7.3
>>> a = b + 5
>>> a
12.3
>>> a = a * 3    # la nouvelle valeur peut dépendre de l'ancienne valeur
>>> a
36.9

```

**Affectation vs comparaison.** **Attention** : le signe “=” en Python sert **seulement** à faire une affectation. Si on veut savoir si deux nombres sont égaux, on utilise l’opérateur de comparaison “==”.

```

>>> a = 6        # initialisation de a
>>> a
6
>>> b = 9        # initialisation de b
>>> a==b         # comparaison, renvoie un booléen
False
>>> a=b          # affectation, a reçoit la valeur de b
>>> a            # a vaut maintenant 9
9

```

### 1.2.4 Types

Les exemples ci-dessus montrent que les valeurs des variables peuvent être de plusieurs natures (ici entier, réel, chaîne de caractères). En informatique, on parle de **type**. Dans de nombreux langages de programmation, on doit déclarer le type des variables (typage statique).

Ce n’est pas le cas en Python. En Python, le typage est **dynamique**, c’est-à-dire que l’interpréteur déterminera automatiquement le type en fonction de la valeur qui a été affectée à une variable (et il change quand la valeur de la variable change). On peut connaître ce type en écrivant : `type(ma_var)`. Les types de base sont : entier (`int`), réel (`float`), chaîne de caractères (`str`) et booléen (`bool`). On en apprendra plus sur les booléens au chapitre 1.4, et sur les chaînes de caractères au chapitre 1.8.

```

>>> a = 17
>>> type(a)
<class 'int'>

>>> a = "salut"
>>> type(a)
<class 'str'>

>>> a = 3.14
>>> type(a)
<class 'float'>

>>> a = (21==7*3)
>>> a
True
>>> type(a)
<class 'bool'>

```

### 1.2.5 Expressions et opérateurs

Dans les exemples, nous avons montré qu'il était possible de faire des opérations, par exemple en écrivant `42 + 2 * 5`. Cela s'appelle une **expression**, c'est une "formule" qui peut être évaluée (calculée). Une variable ou une constante est aussi une expression, mais dont on n'a pas besoin de calculer la valeur (elle est déjà connue).

```
3
x
3*2.0 - 5
"bonjour"
20 / 3
x > 7
```

Dans ces expressions, on a des **opérandes**, et des **opérateurs**. Les opérateurs que l'on peut utiliser ne sont pas les mêmes selon le type des valeurs qu'on manipule. Quelques opérateurs :

- arithmétiques (sur des nombres, produisent des nombres) : addition `+`, soustraction `-`, multiplication `*`, puissance `**`, division réelle `/`, modulo `%`, division entière `//`
- de comparaison (sur nombres ou chaînes de caractères, produisent un résultat de type booléen) : `==`, `!=`, `<`, `>`, `<=`, `>=`
- logiques (entre des booléens, résultat booléen) : `or` (disjonction), `and` (conjonction), `not` (négation). (Voir le chapitre sur les booléens)

#### Exemples d'opérateurs :

```
>>> 2 + 3      # addition
5
>>> 2 * 3      # multiplication
6
>>> 2 ** 3     # puissance
8
>>> 20 / 3     # division réelle
6.666666666666667
>>> 20 // 3    # division entière
6
>>> 20 % 3     # reste de la division entière (modulo)
2
>>> 2 > 8      # comparaison de 2 entiers, expression booléenne
False
>>> 'a' < 'z'   # comparaison de 2 caractères, expression booléenne
True
>>> 'b' < 'a'   # la comparaison respecte l'ordre alphabétique
False
>>> 2 <= 8 < 15 # comparaison de 3 entiers, expression booléenne
True
>>> (2 <= 8) and (8 < 15) # conjonction, expression booléenne équivalente à la précédente
True
>>> (x % 2 == 0) or (x >= 0) # disjonction, expression booléenne dont la valeur dépend de la
                             # valeur de la variable x (True si x pair ou positif)
>>> "a"+"b"     # concaténation de chaînes de caractères
"ab"
>>> "to"*3      # répétition de chaîne de caractères
"tototo"
```

**Abréviations** Quelques opérateurs permettent d'abrégier les notations des affectations d'une valeur dans une variable qui dépend de son ancienne valeur. Par exemple :

```
>>> a = 3      # affectation
>>> a
3
>>> a += 1     # abréviation de a = a + 1
>>> a
4
>>> a *= 3     # abréviation de a = a * 3
>>> a
12
>>> a -= 5     # abréviation de a = a - 5
>>> a
7
```

**Priorité des opérateurs** Lorsqu'une expression comporte plusieurs opérateurs, afin de savoir dans quel ordre elle est évaluée, on considère la **priorité** des opérateurs. Par exemple en arithmétique vous avez appris que la multiplication est plus prioritaire que l'addition, ainsi  $2 * 3 + 4$  est évalué en  $(2 * 3) + 4 = 6 + 4 = 10$  et non pas en  $2 * (3 + 4) = 2 * 7 = 14$ . Le tableau 1.1 indique les priorités des opérateurs principaux de Python, en ordre décroissant. Quand des opérateurs sont de même priorité, l'expression est évaluée de gauche à droite. En cas de doute et pour améliorer la lisibilité, il est conseillé de parenthéser les expressions pour s'assurer qu'elles soient évaluées dans l'ordre souhaité.

Symbole	Nom
**	Puissance
+ unaire - unaire	Positif Négatif
*	Multiplication
/	Division
//	Division entière
%	Modulo
+ binaire - binaire	Addition Soustraction
<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal
==	Égal
!=	Différent
in	Appartenance
not	Négation booléenne
and	Conjonction booléenne (et)
or	Disjonction booléenne (ou)

TABLE 1.1 – Tableau de priorité des principaux opérateurs Python

```
>>> 8 // 4 // 2      # même opérateur, même priorité, évaluation de gauche à droite
1
>>> 8 // (4 // 2)   # les parenthèses forcent l'évaluation de droite à gauche
4
>>> 1 < 2 < 3 == 7%2 != 0  # les opérateurs de comparaisons sont aussi prioritaires
                               # évaluation de gauche à droite: True == 1 != 0
False
>>> (1<2<3) == (7%2!=0)   # comparaison des résultats booléens des 2 tests
True
```

### 1.2.6 Instructions

Une instruction est une **action** à exécuter par le programme. Un programme informatique est constitué d'une suite d'instructions exécutées dans l'ordre pour atteindre un résultat. Par exemple, une affectation de valeur dans une variable est une instruction. Dans la suite de ce cours nous allons apprendre :

- des types de données plus ou moins complexes (booléens, listes, dictionnaires, fichiers)
- d'autres **instructions**, comme les instructions d'entrées-sorties permettant l'interaction avec l'utilisateur du programme (lecture d'une valeur tapée au clavier, affichage d'une valeur à l'écran : cf chapitre suivant) ;
- des **structures de contrôle** qui permettent d'écrire des programmes plus complexes que de simples séquences d'instructions : exécution sélective / conditionnelle (if), répétition en boucle (for, while).

### 1.2.7 Les commentaires

Les commentaires sont des annotations du programme, qui ne sont pas analysées par l'interpréteur, mais qui servent à mieux comprendre le programme (pour vous, votre binôme, votre professeur...). En Python les commentaires s'écrivent en commençant une ligne par # pour dire à l'interpréteur de l'ignorer.

### 1.2.8 Outils utiles

C'est en programmant qu'on s'améliore ! Pratiquez autant que vous le pouvez, grâce aux outils ci-dessous ou à d'autres que vous pourriez trouver en ligne. Dans cette UE nous utiliserons un certain nombre d'outils :

- CaseIne <https://caseine.org/enrol/index.php?id=86>. Sur cette plateforme vous trouverez l'ensemble des documents du cours ainsi que beaucoup d'exercices d'entraînement auto-évalués. Il faut choisir "Connexion Grenoble" puis vous connecter avec vos identifiants Agalan. Vous devrez d'abord vous inscrire dans votre groupe de TD pour accéder à la suite.
- En TP, nous programmerons sous Idle, que vous pouvez aussi installer chez vous : <https://python.developpez.com/telecharger/detail/id/1925/IDLE>
- PythonTutor permet de visualiser l'exécution d'un programme Python pas à pas (penser à sélectionner Python 3.6 dans le menu déroulant) : <http://pythontutor.com/visualize.html#mode=edit>
- PLM (Programmer's Learning Machine) : <https://plm.telecomnancy.univ-lorraine.fr/#/>. Plateforme d'exercices en ligne. Changer Java en Python en haut à droite. Choisir les thèmes d'exercices en haut à gauche.

D'autres outils utiles :

- PyCharm, une IDE avec un bon débogueur, qui permet de voir la pile d'appels, et interfacée avec GIT (gestionnaire de versions) : <https://www.jetbrains.com/pycharm/> (version Community gratuite)
- Anaconda, une distribution de Python pour des applications scientifiques : <https://www.anaconda.com/distribution/>
- GIT est un gestionnaire de versions (utile par exemple pour un projet en groupe). Voir par exemple ce tutoriel des commandes basiques <https://rubygarage.org/blog/most-basic-git-commands-with-examples>, et cette interface graphique pour GIT <https://www.sourcetreeapp.com/> (outil gratuit pour Windows / Mac).

Des sites pour s'entraîner :

- Le Monde de Reeborg <http://reeborg.ca/reeborg.html> : une initiation graphique interactive à Python, en promenant un robot dans un monde simple.
- CodinGame <https://www.codingame.com/learn> fournit de nombreux exercices (sélectionner le langage Python).

## 1.3 Entrées / Sorties

Dans la plupart des cas on a besoin de pouvoir interagir avec un programme :

- pour lui fournir les données à traiter, en général au clavier -> **entrées**
- pour pouvoir connaître le résultat d'exécution d'un programme, ou pour que le programme puisse écrire ce qu'il attend de l'utilisateur, en général texte écrit à l'écran -> **sorties**

### 1.3.1 Les entrées

Pour gérer les entrées au clavier, on utilise la fonction `input()`. Quand l'ordinateur exécute la fonction `input()`, il interrompt l'exécution du programme, affiche éventuellement un message à l'écran (si demandé), et attend que l'utilisateur entre une donnée au clavier et la valide par un appui sur la touche **Entrée**.

**Saisie textuelle et conversion de type.** La fonction `input()` effectue une saisie **en mode texte** : la valeur saisie est considérée comme une chaîne de caractères. On peut ensuite changer son type, pour le convertir en nombre par exemple (attention cela ne fonctionne que si l'utilisateur a bien saisi un nombre, sinon on déclenche une erreur). Cette conversion est indispensable si on veut ensuite manipuler la valeur numérique.

```
>>> texte = input()
123                # on suppose que l'utilisateur saisit 123 en réponse
>>> texte
'123'              # la variable texte contient la chaîne de caractères '123'
>>> texte + 1      # erreur, addition chaîne + entier interdite
>>> val = int(texte) # conversion de la chaîne en entier
>>> val + 1        # pas d'erreur, val est bien un entier
124
>>> y = int(input()) + 1      # en une seule instruction
```

**Message d'instructions.** La fonction `input()` peut recevoir un paramètre optionnel indiquant le message à afficher. Il est toujours préférable de préciser un message, afin que l'utilisateur comprenne pourquoi le programme s'arrête et ce qui est attendu de lui.

```
>>> x = float(input("Entrez un nombre : "))
Entrez un nombre :      # le programme affiche le message et attend
12.3                    # on suppose que l'utilisateur répond 12.3
>>> x + 2
14.3
```

### 1.3.2 Les sorties

En mode "calculatrice", Python lit-évalue-affiche (comme fait dans les exemples précédents) mais quand on veut demander un affichage au sein d'un programme écrit dans un fichier (script), on utilise la fonction `print()`. Elle se charge d'afficher la représentation textuelle de n'importe quel nombre de valeurs fournies entre les parenthèses et séparées par des virgules (arguments). Par défaut, à l'affichage, ces valeurs sont séparées par un espace et l'ensemble se termine par un retour à la ligne.

```
>>> a = 20
>>> b = 13
>>> print("La somme de", a, "et", b, "vaut", a+b, ".")
La somme de 20 et 13 vaut 33.
```

**Modification de l'affichage** Cependant on peut modifier ce comportement par défaut (séparation par des espaces et retour à la ligne final) en spécifiant les paramètres optionnels `sep` et/ou `end`. **Attention**, ces paramètres fonctionnent uniquement pour la fonction `print` (pas avec `print`). On peut aussi insérer manuellement des sauts de ligne en utilisant `"\\n"` et des tabulations avec `"\\t"` ; ce sont des "caractères spéciaux".



```
>>> print(a,b,sep=";")           # séparateur ; mais on garde le retour à la ligne final
20;13
>>> print("a=",a,"b=",b, sep="\n") # retour a la ligne entre chaque argument et final
a=
20
b=
13
>>> print(a,end="!")             # pas de retour a la ligne final
20!>>>
>>> c = 7
>>> print(a,b,c,sep=";",end=" !\n") # séparateur ; au lieu d'espace, et on rajoute un ! avant
                                     # le retour chariot final (qu'il faut spécifier avec \n)
20;13;7!
>>>
```

**Alignement de l’affichage** Il peut parfois être utile d’aligner les chaînes affichées, notamment quand elles sont de tailles différentes. Par exemple imaginons qu’on veuille afficher une liste d’entiers (colonne de gauche), leur carré (colonne du milieu), et leur cube (colonne de droite). L’affichage standard avec l’instruction `print(x,x**2,x**3)` produira un tableau peu lisible (à gauche ci-dessous).

La méthode `rjust()` permet de justifier l’affichage d’une chaîne de caractères à droite. Elle reçoit un argument qui indique la taille de la colonne dans laquelle on justifie à droite (et donc permet de déduire combien d’espaces il faut insérer à gauche). Ainsi l’instruction `print(str(x).rjust(2), str(x*x).rjust(3),str(x*x*x).rjust(4))` permet de justifier la première colonne (entiers à 1 ou 2 chiffres) sur 2 caractères, la 2e sur 3 caractères, et la 3e sur 4 caractères (taille maximale de 4 chiffres). Elle produit un tableau mieux aligné et plus lisible (à droite ci-dessous).

1	1	1	1	1	1
2	4	8	2	4	8
3	9	27	3	9	27
4	16	64	4	16	64
5	25	125	5	25	125
6	36	216	6	36	216
7	49	343	7	49	343
8	64	512	8	64	512
9	81	729	9	81	729
10	100	1000	10	100	1000

**Formatage** Une autre manière de formater l’affichage consiste à utiliser l’opérateur `%` comme dans les exemples ci-dessous.

```
for i in range(7,11):           # répétition, cf chapitre 10
    print('%2d' % i,'%3d' % i**2) # affichage de l'entier i sur 2 chiffres
                                   # et de son carré sur 3 chiffres

7  49
8  64
9  81
10 100
# affichage du nombre pi (fourni par le module math)
>>> print('%1.7f' % math.pi)     # avec 1 chiffre avant la virgule et 7 chiffres après
3.1415927
>>> print('%1.17f' % math.pi)    # avec 17 chiffres après la virgule
3.14159265358979312
```

**Plus d’informations.** Pour plus de détails sur l’affichage et le formatage des données, voir par exemple : <https://docs.python.org/fr/3/tutorial/inputoutput.html>

## 1.4 Les expressions booléennes

### 1.4.1 Définition : expression booléenne

Une expression qui ne peut prendre que les valeurs **True** (vrai) ou **False** (fausse) est appelée **expression booléenne**. En Python, il s'agit du type `bool` (booléen), vu au chapitre précédent.

```
>>> 1 < 2 < 3      # les entiers 1,2,3 sont-ils en ordre croissant strict
True
>>> 7%2 == 0       # 7 est-il multiple de 2 (reste de la division est nul)
False
```

### 1.4.2 Opérateurs booléens

**OU logique : or.** `expr1 or expr2` vaut vrai si et seulement si au moins une des deux expressions `expr1` et `expr2` est vraie, éventuellement les deux.

**ET logique : and.** `expr1 and expr2` vaut vrai si et seulement si les deux expressions `expr1` et `expr2` sont vraies.

**Négation logique : not.** `not expr` vaut vrai si et seulement si `expr` vaut faux.

L'opérateur de négation est le plus prioritaire, suivi de `and`, suivi de `or`. Par exemple `not a or b and c` est évalué comme `(not a) or (b and c)`.

### 1.4.3 Élément neutre des opérateurs booléens

L'élément neutre `e` d'un opérateur est celui tel que pour toute expression `b`, `b OP e` est égal à `b`.

Pour le ET l'élément neutre est donc `True`, car `True and b` vaut toujours `b`.

Pour le OU l'élément neutre est `False`, car `False or b` vaut toujours `b`.

### 1.4.4 Évaluation fainéante

En Python, les opérateurs `and` et `or` sont fainéants, c'est-à-dire que si l'évaluation de la première opérande permet déjà d'évaluer l'expression, alors la deuxième opérande n'est même pas évaluée.

```
# on suppose que la variable a n'est pas définie
>>> (2 == 1+1) or (a>5)    # l'opérande gauche est vraie donc la disjonction est déjà vraie
True

>>> (3 == 1+1) or (a>5)    # l'opérande gauche est fausse donc il faut évaluer la droite
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> ( 2 == 1+1 ) and (a>5)  # la conjonction nécessite que les 2 opérandes soient évaluées à vrai
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> (3 == 1+1) and (a>5)    # l'opérande gauche est fausse donc la conjonction aussi
False
```

### 1.4.5 Lois de De Morgan

Les lois de De Morgan permettent de simplifier des négations d'expressions contenant une conjonction (and) ou une disjonction (or).

- $\text{not}(\text{expr1 or expr2}) = \text{not}(\text{expr1}) \text{ and } \text{not}(\text{expr2})$
- $\text{not}(\text{expr1 and expr2}) = \text{not}(\text{expr1}) \text{ or } \text{not}(\text{expr2})$

#### Exemples

$\text{not}(\text{ a } > 2 \text{ or } \text{ b } \leq 4 )$  équivaut à  $\text{not}(\text{a} > 2) \text{ and } \text{not}(\text{b} \leq 4)$  ce qui équivaut à  $(\text{ a } \leq 2 ) \text{ and } (\text{ b } > 4 )$   
 $\text{not}(\text{ a } > 2 \text{ and } \text{ b } \leq 4 )$  équivaut à  $\text{not}(\text{a} > 2) \text{ or } \text{not}(\text{b} \leq 4)$  ce qui équivaut à  $(\text{ a } \leq 2 ) \text{ or } (\text{ b } > 4 )$

### 1.4.6 Tables de vérité

Une table de vérité permet d'évaluer et de comparer des expressions booléennes. On commence par lister toutes les variables et toutes les combinaisons possibles de leurs valeurs (ci-dessous abrégées T pour True/vrai et F pour False/faux). Plus il y a de variables dans l'expression, et plus il y aura de combinaisons de valeurs possibles : pour n variables, il y a  $2^n$  combinaisons, donc 4 lignes pour une table de vérité à 2 variables, 8 lignes pour 3 variables, etc.

Par exemple la table de vérité ci-dessous permet d'évaluer le 'ou exclusif' (noté xor) entre 2 variables a et b, défini comme 'soit a soit b est vraie' (**a or b**) 'mais pas les 2 en même temps' (**not (a and b)**).

a	b	a or b	a and b	not (a and b)	a xor b
V	V	V	V	F	F
V	F	V	F	V	V
F	V	V	F	V	V
F	F	F	F	V	F

TABLE 1.2 – Table de vérité pour le ou exclusif

## 1.5 Instructions conditionnelles

En programmation, on peut vouloir effectuer des actions différentes selon qu'une certaine condition est remplie ou pas. Par exemple : faire un traitement différent selon que la valeur d'une variable est positive ou non.

### 1.5.1 Syntaxe basique

L'instruction conditionnelle la plus simple en Python s'écrit comme ceci :

```
if condition :
    suite d'instructions si vrai
```

Dans ce cas, si la condition est vraie, la suite d'instructions est exécutée. Si la condition est fausse, rien n'est fait, le programme continue après cette instruction. La condition doit être une expression booléenne (par exemple le résultat d'une comparaison entre deux nombres, etc).

Attention à l'**indentation**, c'est-à-dire le décalage à droite et l'alignement de la suite d'instructions à l'intérieur du **if**. C'est l'indentation qui détermine dans quel bloc se trouve une instruction, et qui indique donc dans quel cas elle doit être exécutée.

### 1.5.2 Syntaxe si-alors-sinon

Si on veut faire quelque chose quand une condition est vraie, et autre chose quand elle est fausse, on peut rajouter un bloc **else** (sinon). Dans ce cas, si la condition est vraie, la première suite d'instructions est exécutée, alors que si la condition est fausse, c'est la deuxième suite d'instructions qui est exécutée. Le programme continue ensuite à l'instruction suivante. La syntaxe est la suivante :

```
if condition :
    suite d'instructions si vrai
else :
    suite d'instructions si faux
```

**Indentation** En Python, c'est l'**indentation** qui détermine à quel bloc appartient une instruction. Après la ligne **if condition**, les lignes suivantes sont indentées à droite pour signifier qu'elles appartiennent à ce bloc **if**. Si une instruction est alignée à gauche (au même niveau que le **if**), alors elle marque la fin de l'instruction conditionnelle. Pensez donc toujours à bien indenter votre code (ce qui le rend aussi beaucoup plus lisible).

#### Exemple

```
x = 5
if x > 0 :
    print(x, "est plus grand que 0")    # dans le bloc if
    print("il est strictement positif") # dans le bloc if
else :
    print(x, "est négatif ou nul")      # dans le bloc else
print("Fin")                          # ni dans if ni dans else, toujours exécutée,
                                     # après la fin de l'instruction conditionnelle
```

**Attention**, on peut avoir un **if** sans **else**, mais pas l'inverse. Si on ne veut faire aucun traitement quand la condition est fausse, alors il ne faut pas écrire de **else**. On ne peut en aucun cas écrire un bloc **else** vide ! Si on ne veut exécuter une instruction que quand la condition est fausse, il faut écrire un **if** avec la négation de cette condition.

### 1.5.3 Cas multiples

Pour enchaîner les conditions, on dispose également du mot clé **elif** (contraction de **else if**). Les parties **elif** sont optionnelles, comme la partie **else**. On peut mettre plusieurs blocs **elif** pour distinguer plusieurs conditions, autant que nécessaire (par contre il ne peut y avoir qu'un seul **else** qui signifie que toutes les conditions précédentes sont fausses).

**Remarque :** la condition d'un bloc `elif` n'est évaluée que si les conditions précédentes ont échoué. Le programme ne pourra rentrer que dans **un seul** bloc d'une instruction conditionnelle (le premier dont la condition est vraie). Il n'est donc pas nécessaire de spécifier la négation des conditions précédentes dans les conditions suivantes. Par exemple :

```
note = int(input("Quelle est votre note en maths ?"))

# ce qu'il ne faut pas faire
if note < 10:
    print("Vous n'avez pas la moyenne")
# on teste inutilement si note >= 10
elif 10 <= note < 12:
    print("Pas mal...")
# on teste inutilement si note >= 12
elif 12 <= note < 15:
    print("Mention Bien !")
# on teste inutilement si note >= 15
elif note >= 15:
    print("Vous avez la bosse des maths !")

# la version correcte
if note < 10:
    print("Vous n'avez pas la moyenne")
# si le premier test échoue, on sait déjà que note >= 10
# on n'a donc pas besoin de le vérifier dans la condition du elif
elif note < 12:
    print("Pas mal...")
elif note < 15:
    print("Mention Bien !")
# si le 2e et le 3e tests échouent aussi, on sait que note >= 15, un else suffit
else:
    print("Vous avez la bosse des maths !")
```

### 1.5.4 Cas par défaut

**Attention :** en l'absence d'un bloc `else` (cas par défaut) il est possible que le programme ne rentre dans **aucun** des blocs d'une instruction conditionnelle (si toutes les conditions sont fausses). Il est donc en général préférable de toujours prévoir un cas par défaut (bloc `else`). Par exemple :

```
reponse = input("Faites-vous du sport régulièrement ? (oui/non) ")
if reponse == 'oui':
    print("Super, c'est bon pour la santé !")
elif reponse == 'non':
    print("Dommage, vous devriez vous y mettre...")
# en l'absence d'un bloc else, le programme n'affichera rien
# si l'utilisateur répond autrement que par 'oui' ou 'non'
# il faut rajouter un cas par défaut pour gérer les exceptions
else:
    print("Je n'ai pas compris la réponse")
```

**Exemple :** calculer le nombre de racines réelles d'un polynôme du second degré. On sait qu'il y a 3 cas selon que le déterminant est strictement positif, nul, ou strictement négatif.

```
a = 3.2 # coefficient du monome de degre 2
b = 5   # coefficient du monome de degre 1
c = -7.9 # coefficient du monome de degre 0
d = b**2 - 4*a*c
```

```

if d>0 :
    print("Deux racines reelles distinctes")
elif d==0 :
    print("Une seule racine reelle")
else :    # ici on a forcément d < 0
    print("Aucune racine reelle")

```

**Attention** : les `elif` sont suivis d'une condition, par contre le `else` n'est pas suivi d'une condition. Il correspond exactement à la négation de la condition du `if`, ou à la négation du `if` et de tous les `elif` précédents : on y rentre par défaut si on n'a pu rentrer dans aucun bloc précédent de l'instruction conditionnelle.

### 1.5.5 Imbrication

Remarque : on peut **imbriquer** les instructions conditionnelles, c'est-à-dire écrire un `if` dans le bloc d'instructions d'un autre `if`. Cela permet de distinguer des sous-cas, par exemple une fois que je sais que ma variable est positive, je peux vouloir distinguer les valeurs paires ou impaires.

**Attention** à l'indentation. A chaque nouveau bloc imbriqué dans le précédent, on décale d'un cran supplémentaire vers la droite.

#### Exemple

```

print("Testeur de parite")
x = int(input("Entre un entier positif"))
if x>=0 :           # x est positif
    if x%2==0 :     # x est positif ET pair
        print(x,"est pair")
    else :          # x est positif ET non pair (donc impair)
        print(x,"est impair")
else :              # x n'est pas positif (donc negatif)
    print("Erreur !",x,"est negatif")

```

### 1.5.6 Nombres aléatoires

Le module `random` permet de générer des nombres pseudo-aléatoires. Il fournit par exemple les fonctions suivantes qui seront utilisées en TD et TP :

- `randint(a,b)` : renvoie un entier pseudo-aléatoire entre les bornes `a` et `b` (incluses)
- `random()` : renvoie un réel pseudo-aléatoire entre 0 et 1
- `choice(l)` : renvoie un élément au hasard parmi ceux de la liste `l`.

## 1.6 La boucle 'while'

Précédemment, nous avons appris à utiliser les instructions conditionnelles (**if-elif-else**), qui permettent d'exécuter une suite d'instructions uniquement **si** une certaine condition est vérifiée. On appelle ces instructions des **structures de contrôle**. En programmation, il existe d'autres structures de contrôle, les **boucles** (ou itérations), qui permettent de répéter un ensemble d'instructions plusieurs fois. En particulier la boucle *while* permet de répéter un bloc d'instructions **tant que** une certaine condition (une expression booléenne) reste vraie. En Python, la forme générale du **while** est la suivante :

```
while condition :
    instruction_1
    instruction_2
    ...
    instruction_n
# ici la condition est fausse, on sort de la boucle
```

**Indentation :** comme pour le **if**, c'est l'indentation des instructions qui détermine si elles sont dans la boucle (exécutées tant que la condition est vraie) ou en dehors (exécutées après la fin de la boucle, quand la condition devient fausse).

**Condition de boucle :** si la condition est fausse dès le premier essai, alors on n'entre jamais dans la boucle et on passe directement à la suite du programme. Au contraire, si la condition reste toujours vraie, alors on ne sort jamais de la boucle, le programme ne se termine pas, on est coincé dans une **boucle infinie : attention danger !** Pour interrompre un programme qui boucle, il faut taper Ctrl-C.

**Utilité :** les boucles sont très utiles dans plusieurs cas :

- pour filtrer les entrées de l'utilisateur
- pour rejouer un programme
- pour répéter plusieurs fois les mêmes instructions

### 1.6.1 While pour filtrer

On a vu qu'un programme pouvait interagir avec l'utilisateur en lui demandant d'entrer des valeurs. On a parfois besoin de vérifier que les valeurs entrées par l'utilisateur respectent bien certains critères, sous peine que le programme ne fonctionne pas correctement : c'est ce qu'on appelle **filtrer** les entrées.

Ce type de comportement est observé très souvent dans les programmes que nous utilisons tous les jours. Par exemple, un programme qui nous demande de nous identifier avec un mot de passe (par exemple : Skype), va continuer à afficher la page d'accueil tant qu'on n'a pas rentré les bons identifiants.

Considérons un exemple simple, qui demande à l'utilisateur deux entiers positifs A et B, et affiche le quotient de la division de A par B. Pour avoir un résultat correct, nous devons d'abord vérifier que B est différent de 0. Le programme peut être exprimé de la façon suivante :

```
a = int(input('Donnez la valeur de A : '))
b = int(input('Donnez la valeur de B : '))
if b != 0 :
    print('A / B = ', a // b) # division entière
else:
    print('Division par 0 impossible')
```

Ce programme fonctionne correctement, mais si l'utilisateur entre une valeur B nulle, il n'a pas de deuxième chance. Comment le modifier afin qu'il continue de demander la valeur de B à l'utilisateur, **jusqu'à ce que** celle-ci ne soit pas égale à 0 ?

On pourrait répéter manuellement le même code un certain nombre de fois. Par exemple, pour donner à l'utilisateur 3 chances d'entrer une valeur correcte, on peut naïvement écrire :

```

a = int(input('Donnez la valeur de A : '))
b = int(input('Donnez la valeur de B : '))
if b == 0 :
    b = int(input('B ne peut etre nul, veuillez reessayer : '))
    if b == 0 :
        b = int(input('B ne peut etre nul, veuillez reessayer : '))
        if b == 0 :
            b = int(input('B ne peut etre nul, veuillez reessayer : '))
if b != 0 :
    print('A / B = ', a // b)
else:
    print('Division par 0 impossible')

```

Cependant, ce code est très répétitif (ce qui n'est pas bon!) et ne permet de répéter cette action qu'un nombre limité de fois (ici 3 fois). Ce n'est pas le résultat souhaité. A l'aide du `while`, on peut simplement ré-exprimer le programme comme suit :

```

a = int(input('Donnez la valeur de A : '))
b = int(input('Donnez la valeur de B : '))
while b == 0 :
    b = int(input('B ne peut etre nul, veuillez reessayer : '))
# sortie de la boucle quand b est different de 0
# maintenant on peut faire la division sans erreur
print('A / B = ', a // b)

```

Ce programme redemande la valeur de B *tant que* celle-ci est égale à 0 (c'est-à-dire **jusqu'à** ce qu'elle soit non nulle). Si on sort de la boucle, cela veut dire que b n'est plus égal à 0, et on peut alors effectuer notre division sans re-vérifier sa valeur.

### 1.6.2 While pour rejouer un programme

La boucle `while` peut aussi être utilisée pour répéter un programme un nombre indéterminé de fois, selon le choix de l'utilisateur. Par exemple imaginons un programme qui demande à l'utilisateur un caractère et affiche son code ASCII.

```

carac = input("Entre un caractere ?")
code = ord(carac)
print("Le code ASCII de",carac,"est",code)

```

Si on veut maintenant proposer à l'utilisateur de recommencer avec un nouveau caractère, puis un autre, etc, jusqu'à ce qu'il demande d'arrêter, on peut compléter le programme comme ceci :

```

play = 'oui' # on initialise a 'oui' pour jouer au moins une fois
while play=='oui' :
    carac = input("Entre un caractere ?")
    code = ord(carac)
    print("Le code ASCII de",carac,"est",code)
    play = input("veux-tu rejouer ? oui/non ")
# sortie de boucle si l'utilisateur repond autre chose que 'oui'
print("Fin du programme")

```

Ce programme rentre au moins une fois dans la boucle puisqu'on initialise la variable `play` à la valeur 'oui', donc le premier test de la condition réussit. Ensuite, après chaque exécution, le programme demande à l'utilisateur de dire s'il veut continuer ou pas. Tant que l'utilisateur répond 'oui', le programme recommence en demandant un nouveau caractère. Dès que l'utilisateur répond 'non' (ou toute autre chaîne que 'oui') la boucle se termine, et l'instruction suivante est exécutée (affichage du message de fin).



### 1.6.3 While pour répéter, compteur de boucle

La boucle *while* peut également être utilisée pour répéter un bout de code un nombre **déterminé** de fois. Pour ce faire, on utilise une variable compteur qui s'incrémente à chaque itération. Par exemple, si on voulait afficher les 10 premières puissances de 2, on pourrait le faire de la manière suivante :

```
i = 0                # notre variable compteur
while i < 10 :      # pour 10 valeurs de i, entre 0 et 9
    print( 2 ** i ) # afficher 2 puissance i
    i = i + 1       # incrémenter compteur à chaque itération
```

Ici, la valeur du compteur *i* augmente de 1 (on dit que la variable *i* a été **incrémentée**) après chaque affichage. Le premier affichage est effectué lorsque *i* est égal à 0, puis tout de suite après *i* est incrémenté et prend la valeur 1. Puisque *i* est toujours inférieur à 10, la boucle continue et l'affichage est encore effectué pour *i* = 1, etc. L'affichage continue donc jusqu'à ce que *i* devienne supérieur ou égal à 10. En d'autres mots, la variable *i* va « compter » de 0 à 9.

**Attention :** si on oublie d'incrémenter le compteur, il ne deviendra jamais supérieur ou égal à 10, et la boucle ne s'arrêtera jamais, c'est une boucle infinie (mauvais!).

Il est également possible de compter suivant un pas différent de 1. Par exemple, le programme suivant avance par pas de 2 afin d'afficher tous les entiers positifs pairs inférieurs à 100.

```
i = 0
while i < 100 :
    print(i)
    i = i + 2
```

Le pas peut également être négatif : on parcourt en ordre décroissant. Dans ce cas on initialise le compteur à la valeur supérieure, et on précise la borne inférieure comme condition de boucle. Par exemple, pour afficher les 10 premiers entiers strictement positifs dans l'ordre décroissant :

```
i = 10                # borne supérieure
while i > 0 :         # condition sur la borne inférieure
    print(i)
    i = i - 1         # décrémenter le compteur
```

### 1.6.4 Accumulateurs et drapeaux

Lorsqu'on parcourt un ensemble de valeurs, il arrive souvent qu'on ait besoin de garder en mémoire certaines informations sur les valeurs parcourues. Par exemple, on peut avoir besoin de calculer la somme ou le produit de ces valeurs. Ou encore, on pourrait avoir besoin de garder en mémoire la plus grande ou la plus petite valeur que nous ayons rencontrée dans une liste de valeurs. Ceci introduit la notion de variable **accumulateur**.

#### Accumulateurs

Prenons l'exemple du calcul d'une somme d'entiers. On veut écrire un programme qui calcule et affiche la somme des 10 premiers entiers strictement positifs (1 à 10). Comme dans l'exemple précédent, nous avons besoin d'une variable compteur (qu'on appellera *i*), qui va compter de 1 à 10. De plus, nous aurons besoin d'une variable accumulateur (qu'on appellera 'somme'), qui va accumuler progressivement la somme de toutes les valeurs de *i* que l'on rencontre. Généralement, on initialise l'accumulateur à l'élément neutre de l'opération que l'on veut effectuer. Dans notre exemple, l'accumulateur a été initialisé à 0 car c'est l'élément neutre de l'addition ( $0+x=0$ ).

```
i = 1
somme = 0                # initialement, la somme est égale à 0
while i <= 10 :
    somme = somme + i     # chaque valeur de i est rajoutée à la somme (accumulation)
    i = i + 1           # ne jamais oublier de mettre à jour le compteur
# affichage une seule fois, après la fin de la boucle
print('La somme des 10 premiers entiers est : ', somme)
```

## Drapeaux

Parfois, l'information qu'on souhaite garder sur notre séquence n'est pas une valeur numérique mais une propriété. Par exemple, est-ce que tous les nombres parcourus sont impairs? Dans ce cas, on peut utiliser une variable booléenne qui est égale à `True` si la propriété est vérifiée, et à `False` dans le cas où cette propriété est fausse. Cette variable, un accumulateur booléen, est aussi appelée "**drapeau**" (ou flag).

**Exemple :** on veut lire 10 entiers et vérifier qu'ils sont tous impairs. Pour que cette propriété soit vraie, il faut que tous les nombres lus soient impairs. Autrement dit, il faut que : **premier nombre est impair ET deuxieme nombre est impair ET ...** Il s'agit donc d'une accumulation utilisant l'opérateur ET (`and`).

Dans cet exemple, la drapeau `tous_impairs` est initialisé à `True`, car c'est l'élément neutre de l'opération "`and`" : si au moins un élément est pair, le drapeau sera égal à `False`, sinon il restera à `True`. Si le drapeau était une disjonction, c'est-à-dire qu'on utilise un opérateur OU (`or`) entre les éléments accumulés, alors l'élément neutre serait `False` : en effet il suffit que l'une des valeurs soit vraie pour que la disjonction soit vraie.

```
i=0
tous_impairs = True
while i < 10:
    x = int(input('Entrez un entier:'))
    tous_impairs = tous_impairs and x % 2 != 0
    i = i + 1
if tous_impairs:
    print('Tous les nombres entrés sont impairs')
else:
    print('Au moins un nombre entré n'était pas impair')
```

Lorsqu'on utilise l'opérateur '`and`', il suffit que l'une des valeurs ne soit pas impaire pour que notre drapeau soit faux. Une manière équivalente et plus intuitive d'écrire ce type de programme est donc de mettre le booléen `tous_impairs` à `False` dès qu'on tombe sur une valeur paire.

```
i=0
tous_impairs = True
while i < 10:
    x = int(input('Entrez un entier:'))
    if x % 2 == 0:
        tous_impairs = False
    i = i + 1
if tous_impairs:
    print('Tous les nombres entrés sont impairs')
else:
    print('Au moins un nombre entré n'était pas impair')
```

### 1.6.5 Boucle infinie, Break, Continue

#### Boucle infinie

Une boucle peut s'exécuter indéfiniment lorsque la condition du `while` est toujours vérifiée (i.e. ne devient jamais fausse). Cela peut être intentionnel, par exemple en utilisant comme condition le booléen `True`. Le programme suivant continue à afficher tout ce que l'utilisateur rentre au clavier, et ne s'arrête jamais.

```
while True:
    a = input()
    print(a)
```

Mais il arrive également qu'on rentre dans une boucle infinie par erreur, parce qu'on oublie de mettre à jour les compteurs de boucle. Dans l'exemple suivant on oublie d'incrémenter la valeur de `i`, ce qui fait que la condition `i < 10` reste tout le temps vraie. Le programme va donc afficher 1 (2 à la puissance 0) à l'infini, jusqu'à être interrompu par l'utilisateur avec Ctrl-C.

```
i = 0
while i < 10:
    print(2 ** i)
```

### Instruction Break

*break* est une instruction qui permet de sortir d'une boucle immédiatement, indépendamment de la condition du 'while'. Cependant le plus souvent son utilisation n'est pas nécessaire. Par exemple la boucle suivante se terminera dès que *i* est égal à 2, et donc n'affichera que les valeurs 0 et 1. Mais il aurait mieux valu changer la condition du *while* en *i* < 2.

```
# avec break
i = 0
while i < 10:
    if i == 2:
        break
    print(i)
    i = i + 1

# version correcte sans break
i=0
while i<2:
    print(i)
    i+=1
```

Une autre utilisation typique de 'break' consiste à éviter les expressions booléennes compliquées. Par exemple, supposons qu'on veuille lire des entiers au clavier jusqu'à ce qu'on tombe sur un multiple de 2, 3, 5, 7, 11, ou 13. On peut écrire une longue condition booléenne, ou bien la séparer en plusieurs lignes en utilisant 'break'

```
# version 1: longue condition booléenne
a = int(input("Tape un entier"))
while a % 2 != 0 and a % 3 != 0 and a % 5 != 0 and a % 7 != 0 and a % 11 != 0 and a % 13 != 0:
    a = int(input("Retape un entier"))

# version 2: avec break, on teste chaque sous-condition à tour de rôle
a = int(input())
while True:
    if a % 2 == 0:
        break
    if a % 3 == 0:
        break
    if a % 5 == 0:
        break
    if a % 7 == 0:
        break
    if a % 11 == 0:
        break
    if a % 13 == 0:
        break
    a = int(input())
```

Dans certains cas, *break* peut aussi être utilisé pour éviter la répétition de code. Par exemple, supposons que l'on veuille écrire un programme qui lit trois nombres au clavier, et qui continue à afficher la moyenne des trois nombres jusqu'à ce que cette moyenne soit inférieure à 10.

```
# version 1: répétition de code
a = int(input())
b = int(input())
```

```

c = int(input())
moyenne = (a + b + c) / 3
while moyenne >= 10.:
    print(moyenne)
    a = int(input())
    b = int(input())
    c = int(input())
    moyenne = (a + b + c) / 3

# version 2: break évite la répétition
while True:
    a = int(input())
    b = int(input())
    c = int(input())
    moyenne = (a + b + c) / 3
    # dès que la moyenne est <10 on sort de la boucle
    if moyenne < 10:
        break
    print(moyenne)

```

**Comment remplacer break** A noter que l'on essaiera de se passer de l'utilisation de **break** dans la mesure du possible car cela peut rendre le programme difficile à lire et à comprendre. Une autre astuce qui permet d'éviter le cas précédent sans utiliser **break** est d'initialiser la condition de la boucle de sorte à être sûr d'y rentrer une première fois. On préférera cette solution à celle utilisant **break**.

```

moyenne = 100 # On initialise à une moyenne >= 10 pour être sûr
               # qu'on rentre dans la boucle une première fois
while moyenne >= 10:
    a = int(input())
    b = int(input())
    c = int(input())
    moyenne = (a + b + c) / 3
    if moyenne >= 10:
        print(moyenne)

```

De même, dans le cas des expressions booléennes longues, on peut utiliser le caractère `\` pour séparer une expression en plusieurs lignes :

```

a = int(input())
while a % 2 != 0 and \
    a % 3 != 0 and \
    a % 5 != 0 and \
    a % 7 != 0 and \
    a % 11 != 0 and \
    a % 13 != 0:
    a = int(input())

```

Enfin pour les conditions d'arrêt de boucle, on préférera les intégrer dans la condition de la boucle directement plutôt que d'utiliser un **break** dans un **if**.

```

# version avec break (à éviter)
while cond:
    instructions
    if condstop :
        break
    other_instruct

```

```

# réécriture sans break (à préférer)
stop=False
while (not stop) and cond:      # on teste les 2 conditions
    instructions
    if condstop :
        stop=True              # on sortira de la boucle au prochain test
    else :                      # ici stop=False
        other_instruct

```

### Instruction Continue

Comme `break`, l'instruction `continue` permet d'altérer le comportement normal de la boucle. `continue` permet de passer directement à l'itération suivante de la boucle en ignorant toutes les instructions qui restent dans l'itération courante. Par exemple, le programme suivant affiche la somme des 10 premiers nombres entrés au clavier qui ne sont pas multiples de 2 ou de 3. Dans la première version avec `continue`, lorsque la condition du `if` est vérifiée, on repasse directement au début de la boucle pour lire un autre nombre, sans changer la somme, et sans incrémenter `i`. Dans la 2e version, on teste si la condition est fausse avant de comptabiliser `n`, ce qui donnera le même résultat.

```

# version avec l'instruction continue
i = 0
somme = 0 # initialisation de la somme
while i < 10:
    n = int(input())
    if n % 2 == 0 or n % 3 == 0:
        continue # si c'est multiple de 2 ou 3 on passe directement à l'itération suivante
    # donc ici n n'est pas multiple de 2 ni de 3
    somme = somme + n
    i = i + 1

# réécriture sans continue
i=0
somme=0
while i<10:
    n=int(input())
    # on utilise la négation de la condition
    # si n n'est multiple ni de 2 ni de 3 on l'ajoute à la somme
    if n%2!=0 and n%3!=0:
        somme += n
        i += 1

```

### Inconvénients

L'utilisation de `break` et/ou `continue` rend le code plus difficile à lire et analyser, en particulier s'il contient plusieurs niveaux d'imbrications et/ou de longues instructions dans le `while`. De plus ces instructions n'ont pas toujours d'équivalent dans les autres langages de programmation. On essaiera donc autant que possible de se passer de `break` et `continue`.

### 1.6.6 Imbrication de boucles

Il est possible qu'une (ou plusieurs) des instructions d'une boucle `while` soit elle-même une boucle `while`. On parle alors de boucles *imbriquées*. Prenons comme exemple le programme suivant :

```
i = 1
while i <= 3:
    j = 1
    while j <= 2:
        print(i, ", ", j)
        j = j + 1
    i = i + 1
```

Ici,  $i$  est le compteur de boucle de la première boucle `while` (boucle externe) et  $j$  celui de la deuxième boucle `while` (boucle interne). (Attention les 2 compteurs doivent être des variables différentes!) La boucle externe ne passe à l'itération suivante (prochaine valeur de  $i$ ), qu'après que la boucle interne a fini toutes ses itérations. Dans l'exemple, pour chaque valeur de  $i$  (de 1 à 3),  $j$  comptera de 1 à 2. Le programme affichera donc :

```
1, 1
1, 2
2, 1
2, 2
3, 1
3, 2
```

**Exemple d'application** : on veut écrire un programme qui affiche un carré en utilisant le caractère `'*'`. L'utilisateur choisit le côté  $N$  du carré et le programme doit afficher  $N$  lignes contenant chacune  $N$  caractères `'*'`. Par exemple, pour  $N=5$  on affiche :

```
*****
*****
*****
*****
*****
```

**Solution :**

```
N = int(input("Entrez la valeur de N: "))
i = 0                                # compteur de lignes
while i < N:                          # N lignes
    j = 0                            # compteur d'étoiles par ligne
    while j < N:                    # N étoiles par ligne
        print('*', end="")         # pas de retour chariot, étoiles sur la meme ligne
        j = j + 1                  # passage a l'étoile suivante
    # fin de l'itération sur j qui a affiche N étoiles
    print()                        # maintenant, saut de ligne
    i = i + 1                      # on passe à la ligne suivante
# fin de l'itération sur i qui a affiché N lignes
```

**Exercice d'application** Modifier ce programme pour afficher un rectangle de  $N$  lignes et  $M$  colonnes.

## 1.7 Les fonctions

On a pour l'instant développé uniquement des programmes *tout-en-un*. Les fonctions permettent de découper le code en plusieurs morceaux réutilisables. Le but des fonctions est de structurer son code lorsque l'on fait plusieurs fois la même chose (ou presque) :

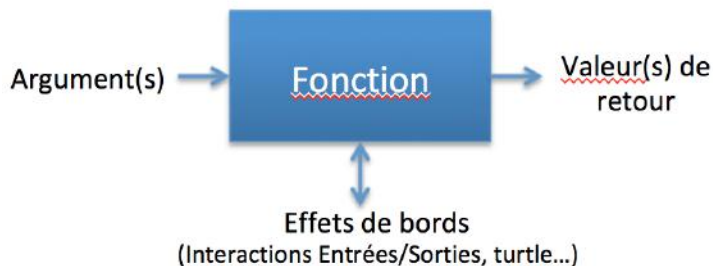
- Pour qu'il soit plus lisible (plusieurs morceaux)
- Pour qu'il soit plus facilement modifiable (pas de duplication de code)
- Pour qu'il soit plus facile à tester (tester chaque morceau séparément)

### Un exemple du TP

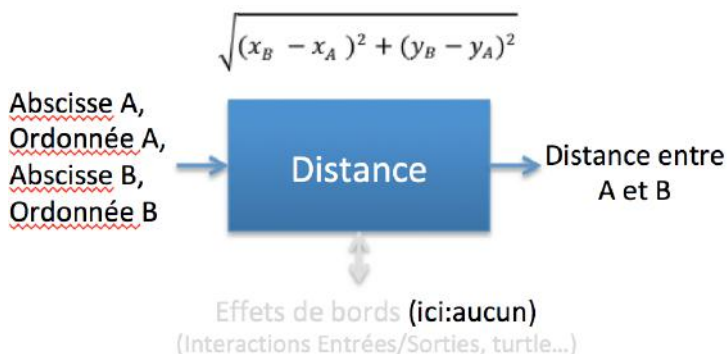
```
import turtle
# fonction qui trace un carre de taille egale a cote
def carre(cote) :
    i = 1                # compteur du nb de cotes
    while i <= 4 :
        turtle.forward(cote)
        turtle.right(90)
        i=i+1
# programme principal
carre(100)                # carre de taille 100
turtle.up()
turtle.forward(130)
turtle.down()
carre(50)                 # carre de taille 50
```

### 1.7.1 Principe d'une fonction

Une fonction est une suite d'instructions, encapsulées dans une «boîte», identifiée par un nom ; elle reçoit zéro, un ou des arguments / paramètres ; elle renvoie zéro, une ou plusieurs valeurs de retour ; et elle crée éventuellement des effets de bord modifiant l'environnement (interactions entrées/sorties, turtle, etc). On peut appeler cette fonction dans un programme pour réutiliser ces instructions plusieurs fois, sur des paramètres différents.



Exemple : fonction géométrique de calcul de distance entre 2 points



Voici le code de cette fonction, qui est à définir au-dessus de votre programme principal. Cette fonction reçoit 4 arguments : `absA`, `ordA`, `absB`, `ordB`. Elle renvoie 1 valeur de retour de type `float`. Elle n'a pas d'effets de bord (en particulier elle n'affiche rien).

```
def distance(absA, ordA, absB, ordB) :
    d=(absB-absA)**2 + (ordB-ordA)**2
    d=d**(1/2)
    return d
```

On peut alors écrire le programme principal suivant, qui appelle cette fonction en lui passant 4 valeurs pour ses 4 paramètres (dans le même ordre) :

```
# prog. principal
print(distance(1, 2, 1, 5))
xA=2
yA=3
z=distance(xA, yA, 0, 0)
print("Distance de (0,0) à A :", z)
```

On remarque que les valeurs passées en paramètres peuvent être soit des constantes (les entiers 1, 2, 1, 5 dans le premier appel), soit des variables (les variables `xA`, `yA` dans le deuxième appel). On remarque aussi que les variables n'ont absolument pas besoin d'avoir le même nom que celui utilisé dans la définition de la fonction. C'est l'ordre qui compte : la première valeur (constante ou variable) est affectée au premier argument, la 2e valeur au 2e argument, etc. Il faut donc passer exactement autant de valeurs lors de l'appel que le nombre de paramètres défini pour cette fonction.

Cette fonction renvoie une valeur, quand on l'appelle il faut donc faire quelque chose de cette valeur : soit on l'affiche directement (comme dans le premier appel), mais alors on ne pourra plus réutiliser cette valeur ; soit on la stocke dans une variable (ici dans la variable `z` pour le 2e appel). Attention, la fonction renvoie une **valeur**, qui est stockée dans la variable `d` dans le corps de la fonction, mais cette variable `d` est **locale** à la fonction, elle n'existe pas dans le programme principal. Il faut donc **affecter** la valeur de retour dans une nouvelle variable, définie dans le programme principal ou dans la fonction appelante.

On peut aussi appeler cette fonction directement depuis l'interpréteur.

```
>>> distance(0, 1, 3, 5)
5.0
>>> distance(1,2,4,7)
5.830951894845301
```

### 1.7.2 Définition d'une fonction

La syntaxe pour définir une nouvelle fonction est la suivante :

```
def nom_fonction(argument1, ..., argumentN) :
    instructions à exécuter
    return valeur de retour
```

**Note** : le `return` est facultatif (dans ce cas la fonction ne renvoie rien), ainsi que les arguments, mais pas les parenthèses.

#### Définition d'une fonction sans paramètres

Voici un exemple de fonction sans paramètre (on note qu'il y a des parenthèses vides après le nom de la fonction, ces parenthèses sont obligatoires). Cette fonction n'a aucune valeur de retour (pas de `return`), par contre elle a un effet de bord, l'affichage d'un message à l'écran.

```
def bonjour() :
    print("bonjour")
```



Ci-dessous, la fonction n'a pas de paramètre, pas d'effet de bord, mais elle renvoie une valeur.

```
def demander_nom():
    nom=input("Quel est ton nom? ")
    return nom
```

La fonction suivante n'a ni paramètre ni valeur de retour, mais a des effets de bord, elle dessine un carré dans la fenêtre `turtle`.

```
import turtle
def carre_standard():
    i = 1 # compteur du nombre de cotes
    while i <= 4 :
        turtle.forward(100)
        turtle.right(90)
        i=i+1
```

### 1.7.3 Appel d'une fonction

**Syntaxe** Une fois définie, on peut appeler une fonction par son nom :

- Depuis le programme principal
- Depuis une autre fonction
- Directement depuis l'interpréteur

On peut appeler soit des fonctions qu'on a soi-même définies, soit des fonctions déjà fournies par Python ou par ses nombreuses bibliothèques (à condition de les avoir importées avant avec `import`). On a par exemple déjà utilisé des fonctions des modules `turtle` et `random`.

La syntaxe pour appeler une fonction est la suivante :

```
nom_fonction(argument1, argument2, ...)
```

**Passage des paramètres** Lors de l'appel d'une fonction, les parenthèses doivent contenir exactement autant de valeurs (constantes ou variables) que la fonction a d'arguments. Ces valeurs sont affectées aux paramètres dans le même ordre. Si la fonction n'a aucun paramètre, les parenthèses sont vides.

Si on appelle une fonction en lui passant trop de valeurs ou pas assez (par rapport à son nombre de paramètres), on déclenche une erreur. Par exemple avec la fonction `distance` définie ci-dessus avec 4 paramètres :

```
>>> distance(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: distance() missing 2 required positional arguments: 'absB' and 'ordB'

>>> distance(1,2,3,4,5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: distance() takes 4 positional arguments but 5 were given
```

**Appel d'une fonction sans paramètres** Pour appeler une fonction sans paramètres, il faut quand même utiliser des parenthèses (mais vides) pour indiquer qu'on veut exécuter cette fonction. Par exemple avec la fonction `bonjour()` définie ci-dessus, directement dans l'interpréteur :

```
>>> bonjour()
bonjour
>>> bonjour
<function bonjour at 0x1048c3048>
```

**Valeur de retour** Si une fonction renvoie une valeur de retour, il faut soit **utiliser** cette valeur immédiatement, ou l'**enregistrer** dans une variable pour pouvoir s'en resservir plus tard. Si on se contente d'appeler la fonction, la valeur est perdue.

```
# definition d'une fonction addition
def addition(x,y):
    s = x+y
    return s

# appel de la fonction depuis l'interpréteur
>>> addition(3,4)      # calcule la somme mais ne sauve pas le résultat
>>> print(s)           # s est une variable locale à la fonction
                        # elle est inconnue ici !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined

# option 1 : afficher directement le résultat
print(addition(3,4))
7

# option 2 : enregistrer le résultat
a = addition(3,4)
print(a)
7
# on peut alors le réutiliser
b = addition(a,6)
print(b)
13
```

**Valeur None** Certaines fonctions n'ont aucune valeur de retour, mais uniquement des effets de bord : elles modifient leur environnement (affichage de valeurs, tracé turtle, etc) mais ne renvoient aucune valeur. On appelle parfois de telles fonctions des **procédures**. Dans ce cas il est inutile d'affecter leur résultat dans une variable (puisqu'il n'y a pas de résultat). Si on le fait quand même, la variable recevra la valeur spéciale **None**, qui signifie "aucune valeur" : la variable n'a pas de valeur, mais s'affiche comme "None".

```
>>> z=distance(2, 3, 4, 5)      # affectation de la valeur dans une variable
>>> print(z)                    # affichage de la variable contenant la valeur de retour
2.8284271247461903

>>> demander_nom()              # effets de bord et valeur de retour
Quel est ton nom? Carole
'Carole'

>>> name                        # on n'a pas sauvé le nom, impossible de l'utiliser
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
>>> name = demander_nom()       # version correcte
Quel est ton nom? Carole
'Carole'
>>> print("Bonjour",name)       # on peut utiliser le nom
Bonjour Carole

# procédure: pas de valeur de retour, pas d'affectation
>>> carre_standard()            # pas d'argument ni valeur de retour
>>> carre(50)                   # pas de valeur de retour mais des effets de bord (turtle)
```

```
>>> x = carre(50)           # si on affecte quand meme
>>> x                       # x n'a pas de valeur
>>> print(x)                # il s'affiche comme la valeur spéciale 'None'
None
>>> x+2                     # on ne peut rien en faire
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

**Appel d'une fonction depuis une autre** On peut appeler une fonction dans le programme principal, mais aussi dans le corps d'une autre fonction. Par exemple, en supposant que la fonction `carre` a été définie précédemment :

```
# fonction qui deplace le curseur sans tracer
# fait appel aux fonctions turtle: up, forward, down
def deplace(distance):
    up()
    forward(distance)
    down()

# fonction qui trace une ligne de carres
# fait appel aux fonctions definies plus haut: carre et deplace
def ligne_carres(nb_carres, cote):
    i=0                      # compte les carres déjà tracés
    while i<nb_carres:
        carre(cote)         # appel a la fonction carre
        deplace(cote+10)    # appel a la fonction deplace
        i=i+1               # au suivant !
```

**Remarque :** on peut même appeler une fonction depuis elle-même, c'est alors une fonction **réursive**. Nous n'en parlerons pas cette année.

#### 1.7.4 Différence entre valeur de retour et effets de bord

On a vu qu'une fonction peut renvoyer une ou plusieurs valeurs de retour, ou aucune ; et qu'elle peut avoir des effets de bord, ou pas. Il faut bien distinguer ces deux concepts.

- Les **valeurs de retour** sont des valeurs renvoyées par la fonction, avec le mot-clé `return`. Ces valeurs peuvent (doivent) être utilisées dans la fonction ou le programme qui appelle cette fonction : on peut les afficher, les affecter dans des variables, etc.
- Les **effets de bord** sont des actions réalisées par la fonction qui modifient son environnement, comme d'afficher des éléments à l'écran (avec des instructions comme `print` ou avec les fonctions du module `turtle` par exemple). Les éléments affichés à l'écran ne sont pas utilisables dans la fonction ou le programme appelant ! Seul l'utilisateur peut les visualiser à l'écran quand il exécute le programme.

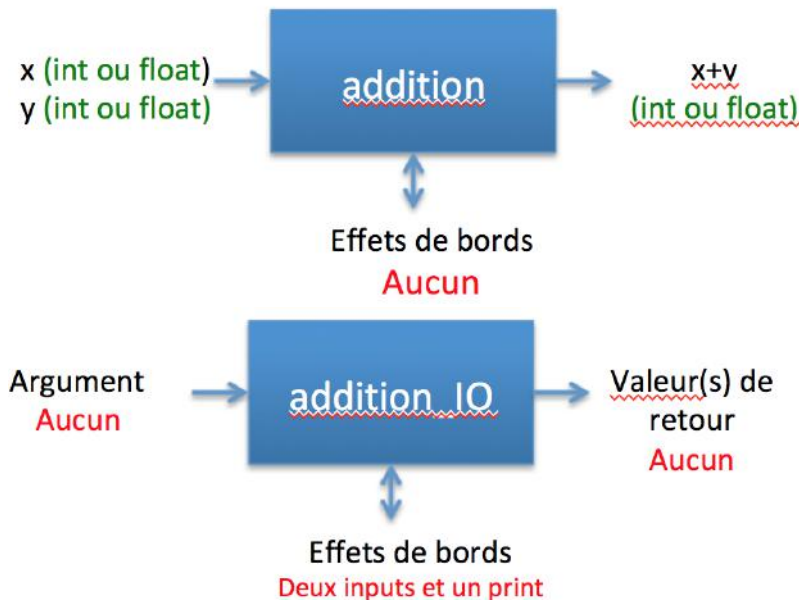
Regardons mieux la différence sur un exemple. On peut écrire plusieurs fonctions d'addition, qui reçoivent les entiers à additionner en paramètre ou via une interaction avec l'utilisateur (effet de bord), et qui affichent le résultat (effet de bord) ou le renvoient (valeur de retour).

```
# fonction d'addition de 2 nombres, qui RENVOIE la somme
def addition(x,y):
    return x+y

# fonction d'addition de 2 nombres, qui AFFICHE la somme
def addition_aff(x,y):
    print(x+y)
```

```
# fonction d'addition, qui demande 2 entiers, et affiche leur somme
def addition_IO():
    x=float(input("x ?"))
    y=float(input("y ?"))
    print(x+y)
```

Les schémas ci-dessous illustrent les entrées, sorties, et effet de bord de ces fonctions d'addition.



L'appel de ces fonctions depuis un programme principal est donc très différent.

```
# programme principal

# appel de la fonction d'addition qui renvoie la valeur
somme = addition(3,7)          # il faut affecter la valeur dans une variable
print("la somme vaut",somme)    # on peut alors afficher cette variable
# on peut aussi afficher directement la valeur de retour sans l'affecter
# mais alors on ne pourra pas la réutiliser (non stockée)
print("la somme de",5,"et",10,"vaut",somme(5,10))

# appel de la fonction qui affiche
# avec cette fonction on ne peut pas stocker la valeur pour la réutiliser
addition_aff(13,25)            # pas d'affectation ! la fonction affichera 38
a = 12
b = 25
addition_aff(a,b)               # la fonction affichera 37
addition_aff(10,a)              # la fonction affichera 22

# appel de la fonction qui demande et affiche
addition_IO()                   # pas d'affectation, pas de parametres
                                # la fonction interagira avec l'utilisateur
```

### 1.7.5 Portée des variables

Chaque fonction a son propre "lot" de variables auquel elle a le droit d'accéder. Une variable créée ou modifiée dans le corps d'une fonction, ou qui contient un argument de la fonction, est dite **locale**, et ne sera pas accessible depuis le programme principal, ni depuis une autre fonction.

L'utilisation de Python Tutor permet de visualiser les variables définies dans les différents environnements (*frames*) : l'environnement global (*global frame*) correspond au programme principal, et chaque fonction a son

propre environnement. Ainsi sur la figure ci-dessous on voit l'environnement `distance` frame contenant les variables locales à la fonction `distance`.

Python 3.6

```

1 def distance(absA, ordA, absB, ordB) :
2     d=(ordB-ordA)**2+ (absB-absA)**2
3     d=d**(1/2)
4     return d
5
6 # prog. principal
7 if __name__=="__main__":
8     xA=2
9     yA=3
10    z=distance(xA, yA, 0, 0)
11    print("Distance de (0,0) à A :", z)

```

Global frame

distance	function distance(absA, ordA, absB, ordB)
xA	2
yA	3

distance

absA	2
ordA	3
absB	0
ordB	0
d	3.6056
Return value	3.6056

### Exemple de variable locale

```

def moyenne(x,y):
    # calcule la moyenne de x et y, et l'affecte dans une variable locale
    resultat=(x+y)/2
    # renvoie la VALEUR de cette variable locale
    return resultat

# programme principal
a=5
b=6
m=moyenne(a,b) # affectation de la valeur de retour dans une variable
print(m)       # affiche 5.5
print(resultat) # provoque une erreur

```

L'exécution de ce programme provoque une erreur : *NameError : name 'resultat' is not defined*, ce qui signifie que la variable `resultat` n'est pas définie dans le programme principal, et donc l'instruction qui essaye d'afficher sa valeur échoue. En effet il s'agit d'une variable locale à la fonction `moyenne`, qui n'existe pas dans le programme principal. Par contre sa **valeur** (renvoyée par la fonction `moyenne`) a bien été sauvée dans la variable `m` lors de l'appel de la fonction.

**Variables du programme principal** Une variable (de type `int`, `float`, `bool` ou `str`) définie dans le programme principal ne peut pas être modifiée par une instruction qui se trouve à l'intérieur d'une fonction. Cela ne provoque pas d'erreur mais cela crée une nouvelle variable locale portant le même identificateur (nom) que l'autre variable. Par exemple :

```

# fonction qui affiche sa variable locale a
def affiche_a():
    a = 1
    print(a)

# programme principal qui définit une variable a
a = 2
# et appelle la fonction ci-dessus
affiche_a() # la fonction affiche la valeur de son a local, soit 1
# affichage de la variable a du programme principal
print(a)   # le programme principal affiche la valeur de son a local, soit 2

```

Un autre exemple :

```
def moyenne(x,y):
    # calcule la moyenne de x et y
    resultat=(x+y)/2
    # crée une nouvelle variable test, locale a la fonction
    test=resultat
    # renvoie la valeur de la variable resultat
    return resultat

# programme principal
a=5
b=6
# variable test, locale au programme principal
test=0
m=moyenne(a,b)
print("m =", m)          # affiche "m=5.5"
print("test =",test)     # affiche "test=0"
```

Ce programme principal initialise à 0 une variable locale `test`. Ensuite, l'appel à la fonction `moyenne` crée une nouvelle variable, aussi nommée `test`, mais locale à cette fonction. Cet appel ne modifie donc **pas** la valeur de la variable `test` du programme principal. L'instruction `print` affiche donc cette valeur non modifiée, c-à-d 0.

**Variables globales** Une variable peut être définie comme globale (mot-clé `global`) pour pouvoir être modifiée partout. Cependant, il faut éviter cette pratique. Cela complique le code inutilement et risque d'entraîner des modifications indésirables ailleurs dans le programme.

**Attention !** Les variables définies dans le programme principal sont en fait accessibles en lecture seule (on peut les lire mais pas les modifier) depuis l'intérieur d'une fonction, mais ce comportement est dangereux car très subtil. On ne l'utilisera donc pas (sauf éventuellement pour des variables "constantes", initialisées une fois au début du programme et jamais modifiées ensuite). On préférera passer en arguments toutes les valeurs nécessaires.

Par exemple, voici 2 manières de définir une fonction qui décale une chaîne de `n` tirets. Dans la première version (incorrecte) on utilise la variable `n` définie dans le programme principal, que la fonction peut lire. Cela fonctionne, mais la fonction est mal définie, si elle a besoin de l'information `n` alors cela devrait être un argument. Dans la deuxième version (correcte), la valeur de `n` est reçue en paramètre, et affectée dans la variable `n` locale à la fonction. C'est la bonne façon de définir cette fonction.

```
# version incorrecte, utilise le n du programme principal en lecture seule
def decalage(s):
    return "-" * n) + s

# depuis l'interpréteur
>>> n=5
>>> print(decalage("toto"))
-----toto

# version correcte, reçoit la valeur en paramètre et l'affecte dans le n local
def decalage(s, n):
    return "-" * n) + s

# depuis l'interpréteur
>>> print(decalage("toto", 5))
-----toto
>>> print(decalage("maison",10))
-----maison
```

**Variables modifiables** Nous avons vu que les variables de types simples définies dans le programme principal ne sont pas modifiables dans les fonctions appelées. Il y a une exception avec les types complexes que nous verrons plus tard. Par exemple le type liste (Chapitre 1.9) permet de stocker une liste de plusieurs valeurs. Ces types complexes fonctionnent un peu différemment des types simples, et en particulier ils peuvent être modifiés par une fonction (ce sera un nouveau type d'effet de bord).

## Résumé

- Une variable créée ou modifiée dans une fonction est **locale**, elle n'existe **que** dans la fonction.
- Une variable simple (de type int, float, str ou bool) du programme principal peut être lue mais ne peut pas être modifiée à l'intérieur d'une fonction. On verra plus tard que c'est différent pour les types complexes comme les listes.
- On passera en argument des fonctions toutes les valeurs nécessaires à leur fonctionnement.
- On évitera les variables globales.
- On affectera la valeur de retour d'une fonction dans une variable pour pouvoir la réutiliser ; les procédures ne renvoient rien et on les appelle donc sans affecter le résultat dans une variable (valeur **None**).

### 1.7.6 Importation de modules

Un grand nombre de fonctions sont déjà définies en Python, et sont rangées dans des modules sur un thème spécifique. Pour utiliser ces fonctions, il faut d'abord importer le module correspondant. Les syntaxes possibles pour importer un module sont les suivantes :

```
import nom_module    # il faut ensuite préfixer les fonctions par le nom du module
from nom_module import nom_fonction # importer une fonction spécifique,
                                   # on n'aura pas besoin de prefixer son nom
from nom_module import * # importer toutes les fonctions du module,
                           # on n'aura pas besoin de prefixer leurs noms

import matplotlib.pyplot as plt    # renommer le module pour simplifier l'appel
```

Le mot clé **as** permet de créer un 'raccourci' pour appeler les fonctions d'un module. C'est pratique quand le nom du module est assez long, puisqu'il faut préfixer l'appel de chaque fonction par le nom du module. Par exemple pour `matplotlib.pyplot` ci-dessus, on pourra maintenant appeler les fonctions en les préfixant uniquement par `plt` au lieu du nom complet du module.

**Modules utiles** Quelques modules intéressants :

- **math** : regroupe les fonctions mathématiques les plus courantes
- **random** : génération de nombres pseudo-aléatoires
- **turtle** : librairie graphique pour l'apprentissage de la programmation (<https://docs.python.org/3/library/turtle.html>)
- **os** : manipulation de fichiers, dossiers, chemins d'accès, permissions...
- **Tkinter** : interface graphique par défaut

## 1.8 Chaînes de caractères

On a déjà utilisé les chaînes de caractères, notamment dans les fonctions `print()` et `input()`. En Python, il existe 3 syntaxes pour les chaînes de caractères :

- avec des guillemets doubles, ce qui permet d'utiliser des guillemets simples (apostrophes) dans le texte :

```
print("je m'appelle toto")
```

- avec des apostrophes, ce qui permet d'utiliser des guillemets dans le texte

```
print('il a dit "bonjour" en arrivant')
```

- avec des guillemets triples, ce qui permet d'utiliser des guillemets et des apostrophes dans le texte :

```
print("""il m'a dit "je m'appelle toto" puis est parti""")
```

### Exemples

```
>>> print("C'est toto")
C'est toto
>>> print('C'est toto')
SyntaxError : invalid syntax
>>> print("Il a dit "hello" !")
SyntaxError : invalid syntax
>>> print('Il a dit "hello" !')
Il a dit "hello" !
>>> print("""C'est toto qui a dit "hello" !""")
C'est toto qui a dit "hello" !
>>> print("""C'est toto qui a dit "hello""")
SyntaxError : ...
```

### 1.8.1 Caractères

Une chaîne est formée de caractères, qui sont représentés chacun par un code ASCII unique (cf tableau ci-dessous). Ainsi les lettres minuscules ont des codes ASCII entre 97 ('a') et 122 ('z'), les lettres majuscules ont des codes ASCII entre 65 ('A') et 90 ('Z'). Le tableau contient aussi des caractères numériques ('0', '1', etc) et divers caractères spéciaux. Pour écrire certains caractères spéciaux dans une chaîne de caractères, il faut utiliser :

- `\n` : retour à la ligne
- `\t` : tabulation

Il n'est pas nécessaire de connaître ces codes ASCII! Les opérateurs `ord()` et `chr()` permettent de les retrouver :

```
>>> ord('a')      # trouver le code ASCII d'un caractere
97
>>> ord('@')
64
>>> chr(99)       # trouver le caractere correspondant a un code ASCII
'c'
>>> chr(123)
'{'
```



### 1.8.2 Opérateurs sur les chaînes

**Concaténation** L'opérateur `+` permet de concaténer plusieurs chaînes de caractères, c'est-à-dire de les coller l'une après l'autre pour former une seule chaîne.

```
>>> nom = input("Ton nom ? ")
Ton nom ? toto
>>> salutation = "bonjour "+nom
>>> print(salutation)
bonjour toto

>>> "abc" + "def"
'abcdef'
```

**Répétition** L'opérateur `*` permet de concaténer plusieurs fois la même chaîne.

```
>>> x = int(input("Combien ? "))
Combien ? 7
>>> print(x*'*')    # affiche x symboles etoile
*****
>>> print('+'*x)    # affiche x symboles +
+++++++

>>> "ta " * 4
'ta ta ta ta'
```

**Comparaison** Les opérateurs de comparaison `<`, `>`, `<=`, `>=`, `==`, `!=` peuvent s'utiliser entre des chaînes de caractères. Il s'agit alors d'une comparaison dans l'ordre de la table ASCII (voir le tableau plus loin) : la comparaison respecte l'ordre alphabétique, les majuscules sont avant les minuscules, les chaînes plus courtes sont avant les chaînes plus longues qui ont le même début.

```
>>> 'abc' < 'a'
False
>>> 'abc' < 'abca'
True
>>> 'abc' < 'z'
True

>>> 'a' < 'A'          # les majuscules sont avant dans la table ASCII
False
>>> 'A' == 'a'         # respect de la casse
False
>>> 'A' < 'a'
True
>>> 'toto' < 'toto aaa'
True

>>> x = 'R'            # affectation
>>> 'a' <= x <= 'z'    # x est-elle une lettre minuscule de l'alphabet?
False
>>> 'A' <= x <= 'Z'    # x est-elle une lettre majuscule?
True
```

### 1.8.3 Fonctions de manipulation de chaînes

**Longueur** La fonction `len()` renvoie la longueur d'une chaîne de caractères.

```
>>> s = "abcde"
>>> len(s)
5
>>> len('toto')
4
```

**Test de casse** On appelle *casse* d'une chaîne de caractère le fait de savoir si elle est en majuscules (haut de casse) ou en minuscules (bas de casse). Python fournit plusieurs fonctions pour tester ou modifier la casse d'une chaîne.

```
>>> 'toto'.isupper()    # est en majuscules ?
False
>>> 'Toto'.isupper()
False
>>> 'TOTO'.isupper()
True
>>> 'toto'.islower()    # est en minuscules ?
True
>>> 'toTo'.islower()
False
>>> 'ToTo azerty'.lower()      # passer en minuscules
'toto azerty'
>>> 'ToTo azerty'.upper()      # passer en majuscules
'TOTO AZERTY'
>>> 'bonjour a tous'.capitalize() # une majuscule au début
'Bonjour a tous'
```

**Découpage et recollage** Il existe différentes fonctions pour découper des chaînes de caractères. La fonction `list()` fournit la liste de tous ses caractères (espaces compris). La fonction `split()` renvoie une liste de chaînes résultant du découpage autour d'un séparateur optionnel fourni en paramètre (ou espace par défaut); le séparateur n'apparaît plus dans les éléments. A partir d'une liste de chaînes, `join()` fait l'opération inverse et les recolle ensemble, séparées par une chaîne. Les exemples ci-dessous illustrent la syntaxe de ces fonctions.

```
>>> s = "bonjour a tous"
>>> list(s)                # liste des caractères de s
['b', 'o', 'n', 'j', 'o', 'u', 'r', ' ', 'a', ' ', 't', 'o', 'u', 's']
>>> s.split()              # découpage de s autour du caractère espace (par défaut)
['bonjour', 'a', 'tous']
>>> s.split('o')           # découpage autour de la lettre 'o'
['b', 'nj', 'ur a t', 'us']
>>> l=['toto','titi','abc','azerty'] # liste de chaînes de caractères
>>> '-'.join(l)
'toto-titi-abc-azerty'
>>> ''.join(ls)            # recollage autour d'une chaîne vide = concaténation
'tototitiabcazerty'
```

# table ASCII

000	NUL (Null Character)	033	!	065	A	097	a
001	SOH (Start of Header)	034	"	066	B	098	b
002	STX (Start of Text)	035	#	067	C	099	c
003	ETX (End of Text)	036	\$	068	D	100	d
004	EOT (End of Transmission)	037	%	069	E	101	e
005	ENQ (Enquiry)	038	&	070	F	102	f
006	ACK (Acknowledgement)	039	'	071	G	103	g
007	BEL (Bell)	040	(	072	H	104	h
008	BS (Backspace)	041	)	073	I	105	i
009	HT (Horizontal Tab)	042	*	074	J	106	j
010	LF (Line Feed)	043	+	075	K	107	k
011	VT (Vertical Tab)	044	,	076	L	108	l
012	FF (Form Feed)	045	-	077	M	109	m
013	CR (Carriage Return)	046	.	078	N	110	n
014	SO (Shift Out)	047	/	079	O	111	o
015	SI (Shift In)	048	0	080	P	112	p
016	DLE (Data Link Escape)	049	1	081	Q	113	q
017	DC1 (XON) (Device Control 1)	050	2	082	R	114	r
018	DC2 (Device Control 2)	051	3	083	S	115	s
019	DC3 (XOFF) (Device Control 3)	052	4	084	T	116	t
020	DC4 (device control 4)	053	5	085	U	117	u
021	NAK (Negative Acknowledgement)	054	6	086	V	118	v
022	SYN (Synchronous Idle)	055	7	087	W	119	w
023	ETB (End of Transmission Block)	056	8	088	X	120	x
024	CAN (Cancel)	057	9	089	Y	121	y
025	EM (End of Medium)	058	:	090	Z	122	z
026	SUB (Substitute)	059	;	091	[	123	{
027	ESC (Escape)	060	<	092	\	124	
028	FS (File Separator)	061	=	093	]	125	}
029	GS (Group Separator)	062	>	094	^	126	~
030	RS (Request to Send)	063	?	095	_	127	DEL
031	US (Unit Separator)	064	@	096	`		
032	SP (Space)						

### 1.8.4 Parcours de chaînes (nécessite le cours sur les boucles)

Les chaînes de caractères sont des structures itérables (comme les listes ou les dictionnaires que nous verrons plus tard). Avec une chaîne `s`, on peut accéder à l'élément à une position `i` donnée (`i` est appelé l'**indice**) avec la notation `s[i]`

On peut aussi parcourir tous les éléments d'une chaîne, par itération sur les indices avec une boucle `while` (cf chapitre 1.6), ou par itération directement sur les éléments avec une boucle `for` (cf chapitre 1.10).

```
>>> s = "bonjour a tous"
>>> s[0]          # le premier caractère de la chaîne
'b'
>>> s[-1]         # le dernier caractère de la chaîne
's'
>>> s[3]          # le caractère à l'indice 3 (le 4e de la chaîne)
'j'

>>> s = 'hello'
>>> i=0           # initialisation du compteur de boucle
>>> while i<len(s): # répétition jusqu'à la longueur de la chaîne
>>>     print(s[i]) # afficher le i-ième caractère de s
>>>     i+=1       # penser à incrémenter le compteur
h
e
l
l
o

>>> for e in s:    # itération directement sur les caractères
>>>     print(e)
h
e
l
l
o
```

## 1.9 Listes

### 1.9.1 Type liste en Python

**Types simples vs types complexes.** On a manipulé jusqu'à maintenant plusieurs types de données : les entiers (int), les réels (float), les booléens (bool), les chaînes de caractères (str). Il s'agit de types simples, c'est-à-dire ne contenant qu'une seule valeur. Mais on a parfois besoin de manipuler des structures de données plus complexes, comme des listes, des ensembles, des tableaux à plusieurs dimensions (matrices), etc.

```
>>> a = 6
>>> type(a)
<class 'int'>
>>> type(3.5)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type('bonjour')
<class 'str'>
```

**Listes.** En Python, une liste est un ensemble **ordonné** d'éléments. Elle est nommée par un identificateur (comme n'importe quelle variable). Une liste peut grandir (ou se réduire) dynamiquement, c'est-à-dire qu'on peut y ajouter ou en enlever des éléments : sa taille n'est pas fixe. Une liste peut contenir des éléments de types différents, y compris d'autres listes. Les éléments sont notés entre crochets, séparés par des virgules.

```
# initialisation de diverses listes
weekend=["Samedi","Dimanche"]           # liste de chaînes de caractères
multiple3 = [3, 6, 9, 12]                # liste d'entiers
romain = [[1,'I'],[2,'II'], [3,'III'],[4,'IV']] # liste de listes
iv = 4
fourreTout = ["Un", 2, 3.0, iv]          # liste contenant plusieurs types d'éléments
vide = []                                 # liste vide
```

**Opérateurs utiles** Certains opérateurs arithmétiques fonctionnent sur les listes : + correspond à une concaténation, \* à une répétition. Cela permet d'initialiser rapidement des listes répétitives.

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l1+l2
[1,2,3,4,5,6]
>>> l = ['C'] + 10*['I']                 # utile pour le TP 'propagation de nouvelle'
>>> l
['C', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I']
>>> len(l)
11
>>> 3 * [2]
[2,2,2]
```

### 1.9.2 Numérotation des éléments et accès

On a dit qu'une liste était un ensemble **ordonné**. Cela veut dire que les éléments sont dans un ordre donné, et qu'on peut accéder à un élément à partir de son numéro, qu'on appelle son **indice**. Les éléments sont indexés (numérotés) à partir de 0, c'est-à-dire que le premier élément a pour indice (position, numéro) 0. En conséquence, pour une liste de n éléments, le dernier élément est à l'indice n-1. Il n'y a pas d'élément à l'indice n.

**Accès à un élément** Pour accéder à l'élément en position i (l'élément d'indice i) d'une liste **maliste**, on écrit **maliste[i]**. Attention, i doit être strictement inférieur à la taille de la liste, sinon on obtient une erreur. Chaque élément est une variable qui peut donc être lue et modifiée.

```
>>> weekend=["samedi","dimanche"]
>>> weekend[0]
'samedi'
>>> weekend[1]
'dimanche'
>>> weekend[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

On peut aussi accéder aux éléments en comptant à partir de la fin de la liste, avec un indice négatif. L'indice -1 correspond au dernier élément, l'indice -2 à l'avant-dernier, etc. Là aussi, si l'indice va trop loin, on déclenche une erreur.

```
>>> weekend[-1]
'dimanche'
>>> weekend[-2]
'samedi'
>>> weekend[-3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

**Longueur d'une liste** Pour connaître la longueur d'une liste, on dispose de la fonction `len()`. Par exemple avec les listes définies ci-dessus.

```
>>> len(weekend)
2
>>> len(romain)
4
```

### 1.9.3 Recherche dans une liste

**Test d'appartenance d'un élément : in** Le mot-clé `in` permet de tester si un élément donné est dans une liste donnée. L'expression `elem in l` est évaluée à `True` si la liste `l` contient l'élément `elem`, et à `False` si elle ne le contient pas.

```
>>> ma_liste=[2,5,8,12,17,25,2,7,2,1]
>>> 2 in ma_liste
True
>>> 3 in ma_liste
False
```

**Compter un élément : count** Par contre `in` ne nous dit pas combien de fois l'élément apparaît, s'il est contenu plusieurs fois dans la liste. Pour cela on dispose de la fonction `count`.

```
>>> ma_liste.count(2)
3
>>> ma_liste.count(31)
0
```

**Position d'un élément : index** Si on veut savoir non seulement si un élément est dans une liste, mais où il se situe dans cette liste (son indice) alors on utilise la fonction `index`. Les indices commencent à 0, le premier élément est en position 0, le 2e en position 1, etc. Si un élément n'est pas dans la liste, la fonction déclenche une erreur. Si un élément est plusieurs fois dans la liste, la fonction renvoie l'indice de sa première occurrence (première apparition).

```
>>> ma_liste.index(5)
1
>>> ma_liste.index(2)
0
>>> ma_liste.index(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 9 is not in list
```

On peut définir une fonction `indice` qui vérifie qu'un élément est dans la liste **avant** de chercher sa position. Par convention, cette fonction renvoie la position -1 pour signifier que l'élément n'est pas trouvé dans la liste.

```
def indice(liste,i):
    if i in liste:
        return liste.index(i)
    else:
        return -1

# dans l'interpréteur
>>> indice(ma_liste,2)
0
>>> indice(ma_liste,9)
-1
```

#### 1.9.4 Affichage

**Affichage standard avec print** La fonction `print` déjà utilisée pour afficher des variables de types simples est aussi capable d'afficher des listes. Par contre le paramètre optionnel `sep` ne sert à rien : il s'agit d'un séparateur entre les paramètres de `print`, or ici il n'y en a qu'un, la liste.

```
>>> l1=[1,2,3]                # crée une liste
>>> print(l1)                  # l'affiche sous forme standard
[1,2,3]
>>> print(l1,sep='-')          # le séparateur ne sert à rien, un seul argument
[1, 2, 3]
>>> print(l1,"toto",sep='-')   # séparateur entre la liste et la chaîne
[1, 2, 3]-toto
```

**Affichage avec boucle while** Si on veut procéder à un affichage personnalisé, il faut parcourir les éléments un par un avec une boucle.

```
# procédure d'affichage d'une liste reçue en paramètre
def afficheET(liste):
    i = 0
    while i<len(liste):
        print(liste[i],"et",end=" ")    # affichage personnalisé sans retour chariot
        i += 1                          # élément suivant
    print()                             # retour à la ligne final
# dans l'interpréteur
>>> prenom = ['toto','titi','yoyo','mumu']
>>> afficheET(prenom)                # pour éviter le et final il faudrait traiter le dernier elmt à part
toto et titi et yoyo et mumu et
```

**Affichage avec boucle for** On verra les boucles `for` au chapitre 1.10. Elles permettent d'itérer directement sur les éléments d'une liste plutôt que sur leur indice, et donc de simplifier l'écriture. Par exemple pour afficher un élément par ligne plutôt que l'affichage standard entre crochets.

```
def afficheFOR(liste):
    for elem in liste:
        print(elem)          # retour à la ligne automatique à chaque élément

# depuis l'interpréteur
>>> l = ['a','b','c']
>>> afficheFOR(l)
a
b
c
```

### 1.9.5 Modifier et ajouter des éléments

**Modification d'une valeur** Pour modifier la valeur d'un élément, il suffit de lui affecter une nouvelle valeur. Attention encore à utiliser un indice existant. On ne peut pas affecter de valeur à un élément qui n'est pas encore dans la liste, là aussi on déclenche une erreur (index out of range = dépassement des limites de la liste). (Ce n'est donc pas comme cela qu'il faut procéder pour ajouter un nouvel élément dans la liste ; on dispose pour cela d'opérateurs dédiés.)

```
>>> weekend[0] = 'saturday'
>>> weekend
['saturday', 'dimanche']
>>> weekend[1] = 'sunday'
>>> weekend
['saturday', 'sunday']
>>> weekend[2] = 'lundi'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

**Ajout d'un seul élément en fin de liste : append** Pour ajouter un seul élément *e* à la fin d'une liste *l*, on utilise la fonction `l.append(e)`. Cette fonction modifie directement la liste (effet de bord), elle ne renvoie rien. Attention donc à ne surtout pas affecter son résultat dans la liste elle-même, sinon on l'écrase !

```
>>> Multiple3 = [3, 6, 9]
>>> Multiple3.append(12)
>>> Multiple3
[3, 6, 9, 12]
>>> Multiple3.append(21)
>>> Multiple3
[3, 6, 9, 12, 21]
>>> x = Multiple3.append(7)
>>> Multiple3          # 7 a bien été ajouté en fin de liste
[3,6,9,12,7]
>>> x                  # mais x n'a reçu aucune valeur
>>> print(x)
None
>>> Multiple3 = Multiple3.append(18)    # à ne SURTOUT PAS faire
>>> Multiple3                          # on a perdu notre liste
>>> print(Multiple3)
None
```

**Insertion d'un élément à un position voulue : insert** Pour insérer un seul élément, non pas en fin de liste mais à une position donnée, on utilise `liste.insert(index,element)`. Si l'indice donné est trop grand (dépasse la taille de la liste), alors l'élément est inséré en fin de liste (pas d'erreur). On peut aussi donner une position négative, qui est alors comptée à partir de la fin de la liste. De même, si cet indice négatif est trop



petit (avant le début de la liste), l'élément est inséré en tout début de liste. Cette fonction modifie directement la liste (effet de bord) mais ne renvoie rien. **Remarque :** l'insertion d'un élément est plus coûteuse en calcul que `append` car elle implique de décaler les autres éléments en mémoire.

```
>> multiple3 = [3, 6, 9, 21]
>> multiple3.insert(3,15)      # insérer en position 3 l'élément 15
>> multiple3
[3, 6, 9, 15, 21]
>> multiple3[3]                # on vérifie
15
>>> multiple3
[3, 6, 9, 15, 21]
>>> multiple3.insert(7,24)      # 7 est trop grand: insertion à la fin
>>> multiple3
[3, 6, 9, 15, 21, 24]
>>> multiple3.insert(-7,12)     # -7 est trop petit: insertion au début
>>> multiple3
[12, 3, 6, 9, 15, 21, 24]
```

**Ajout de plusieurs éléments : `extend`** Pour ajouter plusieurs éléments d'un seul coup à la fin d'une liste, on peut utiliser `liste.extend(liste2)` qui reçoit une deuxième liste en paramètre, et ajoute tous ses éléments à la fin de la liste. Attention, ce n'est pas l'élément `liste2` (un élément de type liste) qu'on ajoute comme un élément de liste, mais bien les éléments qu'elle contient (plusieurs éléments de type entier). Attention, `extend` reçoit un seul paramètre, qui doit être une liste (même si elle contient un seul élément).

```
# avec extend, les éléments de la 2e liste sont ajoutés dans la 1e
>>> multiple3 = [3,6,9,15,21]
>>> multiple3.extend([24,27])
>>> multiple3
[3, 6, 9, 15, 21, 24, 27]
# avec append, c'est la liste qui est ajoutée en fin de liste
>>> multiple3 = [3,6,9,15,21]
>>> multiple3.append([24,27])
>>> multiple3
[3, 6, 9, 15, 21, [24,27]]
# extend reçoit un seul paramètre de type liste
>>> multiple3.extend(24,27)      # erreur, 2 entiers au lieu d'1 liste
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: extend() takes exactly one argument (2 given)
>>> multiple3.extend(24)         # erreur, 24 est un entier et pas une liste
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
# il faut passer une liste, même si c'est un singleton
>>> multiple3.extend([30])
>>> multiple3
[3, 6, 9, 15, 21, 24, 27, 30]
```

### 1.9.6 Supprimer des éléments

**Supprimer un élément à une position donnée : `pop`** La fonction `liste.pop(index)` retire l'élément présent à la position `index` et le renvoie. La fonction `pop` modifie directement la liste (effet de bord), et renvoie l'élément supprimé (valeur de retour); on peut donc l'affecter dans une variable pour le récupérer. Si l'indice donné est hors des limites possibles, l'appel déclenche une erreur. Si on ne fournit pas d'indice (paramètre optionnel), la fonction retire et renvoie le dernier élément de la liste.

```
>>> multiple3 = [3, 6, 9, 15, 21, 24, 27, 24, 24]
>>> a = multiple3.pop(0)
>>> a
3
>>> multiple3
[6, 9, 15, 21, 24, 27, 24, 24]
>>> b = multiple3.pop(27) # erreur, indice 27 trop grand
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
>>> b = multiple3.pop()
>>> b
24
>>> multiple3
[3, 6, 9, 15, 21, 24, 27, 24]
```

**Supprimer un élément de valeur donnée : remove** La fonction `liste.remove(element)` retire l'élément de valeur donnée. S'il est présent plusieurs fois dans la liste, uniquement la première valeur trouvée est supprimée. S'il n'est pas présent, l'appel déclenche une erreur. La fonction `remove` modifie directement la liste (effet de bord) et ne renvoie rien.

```
>>> multiple3
[6, 9, 15, 21, 24, 27, 24, 24]
>>> multiple3.remove(24) # retirer le premier 24 trouvé
>>> multiple3
[6, 9, 15, 21, 27, 24, 24]
>>> multiple3.remove(47) # erreur, tentative de supprimer un élément non présent
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

### 1.9.7 Copie et clone de listes

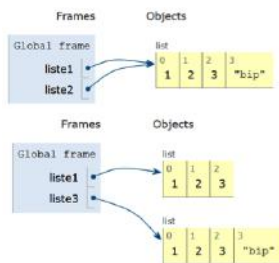
**Renommage de liste** Attention : l'opérateur `=` (affectation) permet juste de donner 2 noms à la même liste. Avec une simple affectation, on donne un deuxième identificateur à la même liste, mais on n'en crée pas de nouvelle. Les deux identificateurs réfèrent toujours à la même liste, et donc les modifications apportées à une des variables après l'affectation s'appliquent également à l'autre variable.

```
liste1=[1,2,3] # création de liste1
liste2=liste1 # liste2 est un autre nom pour liste1
liste2.append("bip") # si on modifie liste2
>>> liste2
[1, 2, 3, 'bip']
>>> liste1 # on modifie aussi liste1
[1, 2, 3, 'bip']
```

**Copie/clonage de surface : list()** La fonction `list()` opère une **copie de surface** : elle crée une nouvelle liste contenant les mêmes éléments que la liste initiale. Il s'agira bien de 2 listes distinctes. Les modifications apportées à une des listes après la copie **n'affecteront pas** l'autre liste.

```
liste1=[1,2,3] # création de liste1
liste3=list(liste1) # copie de liste1 dans liste3
liste3.append("bip") # modification de liste3
>>> liste3
[1, 2, 3, 'bip']
>>> liste1 # liste1 n'a pas été modifiée
[1, 2, 3]
```

**Visualisation** On peut utiliser Python Tutor pour bien visualiser ce qui se passe. Sur la première capture d'écran ci-dessous, on a utilisé une simple affectation `liste2=liste1`, les 2 variables pointent donc vers la même liste : on lui a donné 2 noms. Sur la deuxième capture d'écran on a utilisé `list` pour faire une copie (de surface) de `liste1` dans `liste2`. Les 2 variables correspondent bien à 2 listes différentes.



**Le module copy** Le module `copy` contient plusieurs fonctions génériques permettant de copier des variables de différents types. Pour les utiliser il faut d'abord importer ce module avec l'instruction `import copy`. La fonction `copy.copy()` permet de réaliser une copie *de surface*. La fonction `copy.deepcopy()` permet de réaliser une copie *profonde*, ou récursive.

```
>>> import copy
>>> liste = [1,2,3]
>>> copie = copy.copy(liste)
>>> copie.append(4)
>>> copie
[1, 2, 3, 4]
>>> liste
[1, 2, 3]
```

**Copie de surface vs copie profonde** Si une liste contient d'autres listes, pour la copier correctement il faut utiliser la fonction `deepcopy` du module `copy`, qui va faire une copie *profonde*, récursive, c'est-à-dire qu'elle va aussi faire des copies des listes contenues dans la liste copiée. Pour une liste simple, une copie de surface suffit.

```
>>> liste1
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> surface = copy.copy(liste1)
>>> profonde = copy.deepcopy(liste1)
>>> surface                                     # copie de surface, la liste à l'indice 3 n'est pas clonée
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> profonde
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> surface[3].append('d')                       # si on modifie la liste surface[3]
>>> surface
[1, 2, 3, ['a', 'b', 'c', 'd'], 4]
>>> liste1                                       # alors on la modifie aussi dans liste1 (c'est la même)
[1, 2, 3, ['a', 'b', 'c', 'd'], 4]
>>> profonde[3].append('e')                     # copie profonde/récursive: la liste a été clonée
>>> profonde                                    # si on modifie profonde[3]
[1, 2, 3, ['a', 'b', 'c', 'e'], 4]
>>> liste1                                       # on ne modifie pas liste[3]
[1, 2, 3, ['a', 'b', 'c', 'd'], 4]
```

### 1.9.8 Effets de bord avec les listes

Une fonction peut modifier une liste passée en argument, indépendamment de sa valeur de retour. C'est une nouvelle forme d'effet de bord. Les modifications apportées à la liste dans la fonction sont conservées après la sortie de la fonction, donc dans le programme appelant. **Attention** aux effets de bord non désirés ! Il faut donc

éviter qu'une fonction modifie une liste si ce n'est pas prévu, ou alors bien penser à copier la liste initiale avant de la passer en argument si on ne souhaite pas qu'elle soit modifiée par la fonction.

### Exemple avec oubli de la copie

```
import random

def ajoute_random(l):
    """ Renvoie une liste obtenue a partir de l en ajoutant un entier aleatoire entre 5 et 10"""
    x=random.randint(5,10)
    l.append(x)
    return l
```

Cette fonction modifie la liste reçue en paramètre (effet de bord) **et** renvoie la liste modifiée (valeur de retour). En fait il est inutile de renvoyer la liste, puisqu'on l'a modifiée directement. Ainsi

```
>>> l = [1,2,3]
>>> l2 = ajoute_random(l)
>>> l2
[1, 2, 3, 8]
>>> l
[1, 2, 3, 8]
>>> l.append(5)
>>> l
[1, 2, 3, 8, 5]
>>> l2
[1, 2, 3, 8, 5]
```

On remarque qu'en fait on a ici donné un deuxième identificateur (l2) pour nommer la même liste (l). Et surtout, on a modifié notre liste initiale l alors qu'on ne le souhaitait pas. La version correcte de cette fonction est donc la suivante :

```
import random
def ajoute_random(l):
    """ Renvoie une liste obtenue a partir de l en ajoutant un entier aleatoire entre 5 et 10"""
    x=random.randint(5,10)
    ma_liste=list(l)    # faire une copie de l
    ma_liste.append(x)   # modifier la copie et pas l'original
    return ma_liste     # renvoyer la nouvelle liste, differente de l'original
```

Cette nouvelle version de la fonction crée et renvoie une nouvelle liste. La liste reçue en argument n'est pas modifiée. Il n'y a donc pas d'effets de bord. Attention il faut bien utiliser une fonction de copie et pas une simple affectation (`copie = l`), qui ne crée pas de copie et donc conduit à modifier quand même la liste initiale.

## 1.10 Boucle inconditionnelle : for

### 1.10.1 Boucles conditionnelles vs inconditionnelles

Dans une boucle **while**, c'est la condition qui détermine le nombre de fois que la boucle sera exécutée. Il s'agit d'une boucle **conditionnelle**. La boucle continue **tant que** la condition est vraie, c-à-d **jusqu'à** ce que la condition soit fausse. Le plus souvent on ne sait pas combien de fois la boucle va se répéter.

Si on connaît à l'avance le nombre de répétition, on peut utiliser le **for** qui est une boucle **inconditionnelle**. Au lieu de spécifier une condition de répétition, on spécifie directement le nombre d'itérations ; la boucle se répète exactement ce nombre de fois. On peut aussi parcourir des structures itérables (comme les listes) : la boucle se termine alors à la fin du parcours.

### 1.10.2 Boucle for pour répéter n fois, fonction range()

**Fonction range().** La fonction `range(deb, fin, pas)` reçoit 3 arguments de type entier, et génère une séquence d'entiers compris entre **deb** inclus et **fin** exclus, avec le **pas** choisi. Les paramètres **deb** et **pas** sont optionnels. La borne inférieure **deb** vaut 0 par défaut, et le **pas** vaut 1 par défaut. On peut choisir un pas négatif pour une séquence décroissante d'entiers ; dans ce cas **fin** doit être  $\leq$  **deb**, sinon la séquence est vide.

- `range(a)` : séquence des entiers dans  $[0, a[$ , c'est-à-dire dans  $[0, a-1]$  (borne sup exclue) avec un pas de 1.
- `range(b, c)` : séquence des valeurs  $[b, c[$ , c'est-à-dire dans  $[b, c-1]$  (b incluse, c exclue) avec un pas de 1
- `range(e, f, g)` : séquence des valeurs  $[e, f[$  avec un pas de g

Remarque : comme les arguments ne sont pas nommés, c'est la position des valeurs qui détermine à quel argument elles sont affectées (cf chapitre 1.11). On ne peut donc pas omettre la valeur de début si on veut préciser le pas.

**Utilisation avec for** Quand on veut écrire une boucle inconditionnelle avec n répétitions, le plus simple est donc d'utiliser `range(n)` pour générer une séquence de n entiers.

```
for var in range(n) :
    instructions
```

Plus généralement, si on veut parcourir les entiers entre **deb** et **fin**, avec un pas donné, on peut spécifier les 3 arguments de `range`.

```
for var in range(deb, fin, pas) :
    instructions
```

**Syntaxe.** Comme pour le **if** et le **while**, c'est l'**indentation** des instructions qui détermine si elles sont **dans** le bloc **for** (et donc répétées), ou **après** la fin du bloc (et donc exécutées seulement quand on sort de la boucle).

**Bornes incohérentes** En cas d'incohérence dans les bornes du `range`, la séquence est vide, la boucle est donc ignorée, et l'on passe directement aux instructions suivantes. Par exemple :

```
# de 200 à 210 avec un pas négatif, séquence vide, n'affiche rien
for k in range(200, 210, -2) :
    print(k)

# de 110 à 100 avec un pas positif, séquence vide, n'affiche rien
for k in range(110, 100, 2) :
    print(k)
```

## Exemples

```
# affiche les entiers de 1 à 5 (6 exclus) séparés par une virgule
for i in range(1,6) :
    print(i,end=",")      # affichage de chaque entier suivi d'une virgule
print()                  # retour à la ligne a la fin de la sequence

# affiche les entiers pairs de 0 à 100 (un par ligne)
for i in range(0,101,2) :
    print(i)

# affiche les entiers en ordre décroissant de 10 (inclus) à 0 (exclus)
for i in range(10,0,-1) :
    print(i)
```

**Boucle while vs for** Quand on connaît à l'avance le nombre d'itérations souhaitées, la boucle `for` est beaucoup plus concise. Par exemple, si on veut afficher les entiers de 0 à `n`.

```
# avec un while
i=0          # initialisation compteur
while i<=n :  # condition
    print(i)
    i = i+1   # incrementation compteur
# fin de la boucle quand i dépasse n

# avec un for
for i in range(n+1): # range avec n+1 valeurs a partir de 0
    print(i)
# fin du for apres exactement n+1 iterations
```

### 1.10.3 Parcours de structures itérables

La boucle `for` permet aussi de parcourir des structures : listes, chaînes de caractères, etc. Dans ce cas au lieu d'un compteur de boucle, on utilise une variable qui prend successivement pour valeurs tous les **éléments** contenus dans la structure parcourue (tous les éléments d'une liste, toutes les lettres d'un mot, etc).

```
# parcourt les entiers de la liste, calcule la somme
s = 0
for e in [1, 4, 5, 0, 9, 1] :
    s+=e
print(s)          # affichage en sortie de boucle
# parcourt les lettres de la liste, affiche 1 par ligne
for e in ["a", "e", "i", "o", "u", "y"]:
    print(e)
# parcourt les lettres du mot, affiche 1 par ligne
for e in "python":
    print(e)
```

**Algorithmes sur les listes** La boucle `for` permet ainsi d'écrire des programmes intéressants pour manipuler des listes : trier une liste pour mettre ses éléments dans l'ordre ; chercher l'élément minimum ou maximum d'une liste ; compter le nombre d'occurrences d'un éléments donné dans une liste (combien de fois il apparaît dans la liste) ; etc. (*cf exercices de TD et TP*)

### 1.10.4 Très important

**Attention ! Ne jamais modifier la variable de boucle dans le corps d'une boucle for !**

Quoi qu'il arrive dans le corps de la boucle, la variable de boucle (celle qui parcourt la séquence ou la structure itérable) prend la valeur suivante (l'entier suivant de la séquence, l'élément suivant de la liste ou autre structure itérable) à chaque nouvelle étape de la boucle.

**Exemple (ce qu'il ne faut pas faire)** Ce programme affiche les entiers 1,2,3,4 (un entier par ligne), c'est-à-dire qu'il parcourt la séquence générée par `range(1,5)`. La modification de la variable `i` dans le corps de la boucle est immédiatement écrasée quand `i` prend la valeur suivante de la séquence avant de recommencer les instructions du corps de la boucle. Vous pouvez exécuter ce programme dans Python Tutor pour mieux visualiser.

```
for i in range(1, 5) :  
    print(i)  
    i = i*2
```

**Exemple corrigé (avec un while)** Le programme ci-dessus est mauvais. Il ne faut jamais (jamais !) modifier le compteur de boucle dans un `for`. Dans cet exemple il vaudrait mieux écrire une boucle `while` dans laquelle on peut multiplier `i` par 2 à chaque itération (il faudra initialiser `i` avant le `while`, et la condition du `while` devra spécifier la valeur maximale souhaitée). Cette version fonctionne et affiche les entiers 1,2,4 (un par ligne).

```
i=1  
while i<5:  
    print(i)  
    i=i*2
```

**Exemple corrigé (avec un for)** Avec une boucle `for`, il suffit de se rendre compte qu'on parcourt en fait les puissances de 2, et d'écrire la version suivante, qui itère sur l'exposant, et affiche les 3 premières puissances de 2 (en partant de puissance 0), donc 1, 2, 4.

```
for i in range(3):  
    print(2**i)
```

## 1.11 Fonctions : compléments

### 1.11.1 Le mot-clé None

**Procédures** Il existe une valeur constante en Python qui s'appelle None. Cela correspond à "rien", "aucune valeur". Lorsqu'une fonction n'a pas d'instruction `return`, elle renvoie la valeur None.

```
def dit_bonjour():
    print("Bonjour!")
    print("Bienvenue")
    # pas de return
# programme Principal
test=dit_bonjour()
print("Test vaut", test)
```

```
Affichage obtenu lorsque l'on lance le module:
Bonjour!
Bienvenue.
Test vaut None
```

**Initialisation** Le mot-clé None peut aussi servir à initialiser une variable lorsque l'on ne sait pas encore précisément quelle valeur on souhaite lui attribuer. Attention, None n'est pas une chaîne de caractères en Python, donc il ne faut pas de guillemets!

```
reponse=None # on n'a pas encore de reponse
while reponse!="non":
    x=float(input("Veuillez entrer un nombre:"))
    print("Le carré de ce nombre est:", x*x)
    reponse=input("Voulez-vous recommencer?")
    print("Terminé")
```

### 1.11.2 Fonction avec plusieurs valeurs de retour

On a vu qu'une fonction pouvait ne rien renvoyer. Une fonction peut aussi renvoyer plus d'une valeur. Dans ce cas, quand on l'appelle, il faut affecter le retour dans le bon nombre de variables. Par exemple la fonction ci-dessous calcule et renvoie à la fois le quotient et le reste de la division de a par b. Le programme principal affecte donc son retour dans 2 variables.

**Syntaxe** La syntaxe générale est la suivante :

```
# Syntaxe (dans le corps de la fonction) pour renvoyer N valeurs
return valeur1, valeur2, ... , valeurN

# Syntaxe pour récupérer les N valeurs de retour lors d'un appel
var1, var2, ... , varN = nom_fonction(arguments)
```

**Exemple** Par exemple cette fonction calcule et renvoie à la fois le quotient et le reste d'une division entière.

```
def division(a,b) :
    # renvoie le quotient et le reste
    # de la division de a par b
    quotient=a//b
    reste= a%b
    return quotient, reste

# programme principal
q,r = division(22,5)
print("q=", q, "et r=", r)
```



### 1.11.3 Fonction avec des paramètres optionnels

On a déjà vu plusieurs fonctions avec des paramètres optionnels : `print` (paramètres `sep` et `end`); `range` (paramètres `deb` et `pas`).

**Syntaxe de la définition d'une fonction à paramètres optionnels** Pour rendre un argument optionnel lors de la définition d'une fonction, il faut ajouter après le nom de l'argument le signe `=` suivi de la valeur par défaut. Ci-dessous `arg_opt` est un argument optionnel de la fonction (valeur par défaut précisée), alors que `arg1` n'est pas optionnel (pas de valeur par défaut précisée).

```
# syntaxe standard d'une fonction avec argument optionnel
def nom_fonc(arg1, arg_opt=valeur_par_defaut):
    instructions
```

**Exemple** La carte de MisterPizza comporte de multiples saveurs de pizzas, chacune pouvant être commandée en taille normale au prix de 9 euros, ou en taille maxi au prix de 12 euros. La très grande majorité des clients choisit des pizzas de taille normale, on veut donc que ce soit la valeur par défaut si rien n'est précisé. Dans ce cas on peut utiliser un argument optionnel `taille` qui a pour valeur par défaut `"normale"`.

```
def affiche_pizza(saveur, taille="normale"):
    """ Affiche saveur, taille et prix de la pizza
    """
    print("Pizza", saveur, "taille:", taille)
    if taille=="normale":
        prix=9
    elif taille=="maxi":
        prix=12
    print("Prix", prix, "euros.")
```

**Syntaxe de l'appel d'une fonction à paramètres optionnels** Quand on appelle une fonction à paramètres optionnels, on peut préciser une valeur pour le(s) paramètre(s) optionnel(s), ou bien l'omettre et dans ce cas il(s) reçoit(ven)t leur(s) valeur(s) par défaut (spécifiées dans la définition de la fonction). Si on reprend la fonction de l'exemple ci-dessus, voici comment l'appeler, en précisant ou pas la valeur du paramètre optionnel.

```
>>> affiche_pizza("4 fromages")
Pizza 4 fromages taille: normale
Prix 9 euros.
>>> affiche_pizza("4 fromages", "maxi")
Pizza 4 fromages taille: maxi
Prix 12 euros.
>>> affiche_pizza("Reine", "normale")
Pizza Reine taille: normale
Prix 9 euros.
```

**Définition d'une fonction avec plusieurs paramètres optionnels** MisterPizza souhaite parfois afficher le prix en Francs, lorsque le client est âgé, mais il s'agit d'une situation peu fréquente. On peut donc rajouter un autre paramètre optionnel `afficheF` de type booléen, faux par défaut, qui précise s'il faut afficher le prix en francs ou euros.

```
def affiche_pizza(saveur, taille="normale", afficheF=False):
    """ Affiche saveur, taille et prix de la pizza
    """
    print("Pizza", saveur, ", taille: ", taille)
    if taille=="normale":
        prix=9
    elif taille=="maxi":
```

```

    prix=12
    if afficheF:
        prixFrancs=round(prix*6.55957, 2)
        print("Prix", prix, "euros (", prixFrancs, " F).")
    else:
        print("Prix", prix, "euros.")

```

**Appel d'une fonction avec plusieurs paramètres optionnels** Les valeurs des paramètres sont affectées dans l'ordre où elles sont reçues. Pour appeler la fonction précédente, qui a deux paramètres optionnels, on a donc un problème dans le cas où on veut spécifier le 3e argument (prix en francs) mais pas le 2e (taille normale). En effet la valeur du 2e argument étant omise, c'est la valeur qu'on a donnée pour le 3e (un booléen, True) qui est affectée dans le 2e argument (variable `taille`). Cela déclenche une erreur car la valeur True ne correspond à aucune des 2 valeurs possibles (dans le if), donc la variable `prix` ne reçoit pas de valeur, ce qui empêche de l'afficher ensuite.

```

>>> affiche_pizza("4 fromages")
Pizza 4 fromages taille: normale
Prix 9 euros.

>>> affiche_pizza("4 fromages", "maxi")
Pizza 4 fromages taille: maxi
Prix 12 euros.

>>> affiche_pizza("Reine", "maxi", True)
Pizza Reine , taille: maxi
Prix 12 euros ( 78.71 francs).

>>> affiche_pizza("4 saisons", True)
Erreur car True est pris pour la taille (2eme arg.)

```

**Arguments nommés** Pour résoudre ce problème, il faut nommer les arguments, afin de spécifier de quel argument on reçoit la valeur. En effet, lors d'un appel de fonction, on peut préciser le nom de l'argument concerné par une valeur, comme ceci :

```
>>> affiche_pizza("4 fromages", afficheF=True)
```

Dans cet appel on a 2 types d'arguments :

- La chaîne de caractères "4 fromages" est un argument non nommé, ou argument **positionnel** : c'est sa position qui détermine à quel argument correspond cette valeur (ici la valeur est en 1e position et sera donc affectée dans le 1e argument, à savoir `saveur`)
- `afficheF=True` est un argument **nommé** : c'est le nom qui détermine à quel argument correspond la valeur qui suit (ici la valeur True, en 2e position mais nommée, est donc affectée à l'argument `afficheF`, en 3e position, ce qui permet de ne pas spécifier la valeur du 2e argument, `taille`)

Pour que cela fonctionne, les arguments non-nommés doivent **toujours** être **tous** avant les arguments nommés dans l'appel. L'ordre des arguments nommés peut être fait selon votre préférence, puisque leur position n'est pas déterminante (tant qu'ils sont après les arguments non-nommés).

### Exemples d'appels

```

>>> affiche_pizza("4 fromages", afficheF=True)
Pizza 4 fromages , taille: normale
Prix 9 euros ( 59.04 francs).

>>> affiche_pizza("4 fromages", taille="maxi", afficheF=True)

```

```

Pizza 4 fromages , taille:  maxi
Prix 12 euros ( 78.71  francs).

>>> affiche_pizza("Chorizo", taille="maxi", True)
Erreur: il y a un argument non-nommé après un argument nommé

>>> affiche_pizza("Reine", afficheF=True, taille="maxi")
Pizza Reine , taille:  maxi
Prix 12 euros ( 78.71  francs).

>>> affiche_pizza("Chorizo", True, taille="maxi")
Erreur car taille est définie deux fois (True est pris pour taille car 2eme argument non-nommé)

```

**L'exemple de la fonction print** En fait, nous avons déjà rencontré une fonction avec des arguments optionnels nommés : la fonction `print`

```

>>> print("Mon age est", 18)
>>> print("Mon age est", 18, sep="égal à")
>>> print("Mon age est", 18, end=".")
>>> print("Mon age est", 18, sep=":", end=".")

```

`sep` et `end` sont des arguments optionnels de `print`. Par défaut, `sep` vaut " " (espace) et `end` vaut "\n" (retour à la ligne). Note : `sep` et `end` doivent toujours être nommés car la fonction `print` a un nombre variable d'arguments non-nommés (les valeurs à afficher), donc on ne peut pas déterminer la position de ces arguments.

#### 1.11.4 Utilisation de Docstring

Une docstring est une chaîne de caractères, encadrée par des triples guillemets, placée au tout début d'une fonction, et qui permet de décrire cette fonction.

##### Syntaxe générale

```

def nom_fonction(argument1, argument2, ...):
    """ docstring
    """
    instructions de la fonction

```

On peut l'afficher grâce à la commande `help`, qui prend en paramètre le nom de la fonction (uniquement le nom, pas de parenthèse ni de paramètre) dont on veut obtenir la documentation :

```

help(nom_fonction):
Help on function nom_fonction in module nom_module:
nom_fonction(argument1, argument2, ...)
    docstring de la fonction affichée ici

```

**Exemple** Si on spécifie la documentation au moment de la définition de la fonction, l'utilisateur de notre code pourra alors dans l'interpréteur (ou dans un programme) appeler l'instruction `help` pour afficher cette documentation.

```

def division(a,b) :
    """ Renvoie le quotient et le reste
    de la division de a par b """
    quotient=a//b
    reste= a%b
    return quotient, reste

>>> help(division)

```

```
Help on function division in module __main__:
division(a, b)
    Renvoie le quotient et le reste
    de la division de a par b
```

**Docstring sur une fonction existante** En cas de doute sur le principe d'une fonction existante, vous pouvez utiliser la commande `help` pour afficher sa documentation. Par exemple si on ne se rappelle plus si les bornes sont incluses lors de la génération d'un entier pseudo-aléatoire.

```
>>> import random
>>> help(random.randint)
Help on method randint in module random:
randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

**Docstring et Help sur fonction avec paramètres optionnels** La documentation affichée pour notre fonction précédente montre bien le paramètre optionnel `taille` et sa valeur par défaut.

```
>>> help(affiche_pizza)
Help on function affiche_pizza in module __main__:
affiche_pizza(saveur, taille='normale')
    Affiche saveur, taille et prix de la pizza
```

## 1.12 Dictionnaires

Tout comme une liste, un dictionnaire permet de sauvegarder en mémoire plusieurs valeurs de types quelconques. Cependant, contrairement à une liste, les valeurs d'un dictionnaire ne sont pas stockées de manière ordonnée, mais sont associées à des clés. On accède donc aux éléments par leur clé et pas par leur indice. Chaque clé est **unique**.

### 1.12.1 Création

**Déclaration** La syntaxe générale pour déclarer et initialiser un dictionnaire D contenant certaines paires clé-valeur est la suivante :

```
D = { cle1: valeur1, cle2: valeur2, ..., cleN: valeurN }
```

**Exemple** On veut créer un dictionnaire contenant la note moyenne de chaque étudiant dans une certaine matière. Les clés de ce dictionnaire seront les noms des étudiants, et les valeurs seront leurs notes. Chaque valeur (note) est donc associée à un étudiant (clé). Deux étudiants peuvent avoir la même note, mais chaque étudiant ne peut avoir qu'une seule note (on suppose que chaque prénom est unique dans la classe). Ci-dessous la variable `notes` est un dictionnaire contenant les notes de deux étudiants. Les chaînes de caractères 'nathan' et 'quentin' sont les clés de ce dictionnaire. 12.0 et 15.5 sont les valeurs du dictionnaire. Les éléments du dictionnaire ne sont pas ordonnés : on n'y accède pas par un numéro (indice, position) mais par leur clé.

```
>>> notes = { 'quentin': 15.5, 'nathan': 12.0}
>>> notes
{'nathan': 12.0, 'quentin': 15.5}
```

**Types des clés et valeurs** Les clés d'un dictionnaire ne peuvent être que de certains types. Dans ce cours on se limitera aux entiers et aux chaînes de caractères. Par contre comme pour les listes, les valeurs dans un dictionnaire peuvent être de n'importe quel type, y compris de type dictionnaire.

```
pc_tp = {
    'ram': 16,                # valeur de type int
    'cpu': 3.5,               # valeur de type float
    'portable': False,        # valeur de type bool
    'os': 'windows',          # valeur de type str
    'ports': ['usb3.0', 'jack', 'ethernet', 'hdmi'], # valeur de type list
    'carte_graphique': {      # valeur de type dict
        'vram': 4,
        'nom': 'gtx970',
        'bus': 256
    }
}
```

**Exemple** Ainsi si on veut stocker les moyennes des étudiants dans plusieurs matières, on peut avoir un dictionnaire de dictionnaires, contenant soit un dictionnaire par étudiant (dont les clés seront les matières), soit un dictionnaire par matière (dont les clés seront les étudiants).

```
# un dico par etudiant
notes_e = {
    'nathan': {'maths':15, 'info': 17},
    'quentin': {'info':13, 'bio':18}
}

# un dico par matière
notes_m = {
    'maths': {'nathan':15},
    'info': {'nathan':17, 'quentin':13},
    'bio': {'quentin':18}
}
```

### 1.12.2 Utilisation d'un dictionnaire

**Accès aux valeurs** L'accès à une valeur du dictionnaire se fait non pas par sa position (indice), mais grâce à sa clé. Par exemple dans notre premier dictionnaire de notes, on peut utiliser la chaîne de caractères 'quentin' (la clé) pour accéder à la valeur qui y est associée dans le dictionnaire `notes` (la note de Quentin, 15.5). Les dictionnaires sont aussi appelés **listes associatives**, car ils permettent d'associer à chaque clé une valeur de type quelconque.

```
>>> notes = {'nathan': 12.0, 'quentin': 15.5}
>>> notes['quentin']
15.5
```

**Erreurs de clé** Attention, si on tente d'accéder à une entrée qui n'existe pas dans le dictionnaire, le programme renvoie une erreur de clé (`KeyError`).

```
>>> lettres = {'a': 103, 'b': 8, 'e': 150}
>>> lettres['k']
KeyError: 'k'
>>> lettres['u'] = lettres['u'] + 1
KeyError: 'u'
```

**Vérifier l'existence d'une entrée** Avant d'accéder à une valeur, on prendra donc l'habitude de toujours vérifier l'existence de la clé. Pour cela, on utilise l'opérateur `in` comme dans le cas des listes. **Attention !** L'opérateur `in` vérifie l'existence d'une clé, et non pas d'une valeur.

```
# recherche d'une cle : ok
>>> prix = {'asus': 450, 'alienware': 1200, 'lenovo': 680}
>>> 'asus' in prix
True
>>> 'toshiba' in prix
False
# recherche d'une valeur : echec
>>> 1200 in prix
False
```

### 1.12.3 Modification d'un dictionnaire

**Ajout d'une entrée** Pour rajouter une nouvelle entrée (une paire clé :valeur) dans un dictionnaire existant, il suffit d'utiliser l'opérateur `=` (affectation) pour associer la valeur à la clé, comme suit. On peut donc faire une affectation avec une clé qui n'existe pas encore dans le dictionnaire, pour l'y ajouter.

```
>>> D = {}
>>> D
{}
>>> D['a'] = 1
>>> D
{'a': 1}
```

**Modification d'une entrée** Pour modifier une entrée déjà présente dans le dictionnaire (c'est-à-dire modifier la valeur d'une clé présente dans le dictionnaire), on procède de la même manière, en utilisant aussi l'affectation `=`. L'ancienne valeur de cette clé est écrasée, remplacée par la nouvelle valeur.

```
>>> D
{'a': 1}
>>> D['a'] = 3
>>> D
{'a': 3}
>>> D['a']
3
```

**Suppression d'une entrée** L'opérateur `del` permet de supprimer une association d'un dictionnaire. Comme pour l'accès aux valeurs, il faut que la clé spécifiée existe, sinon on déclenche une erreur de clé.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> del D['a']
>>> D
{'b': 2, 'c': 3}
>>> del D['j']      # erreur, pas de clé 'j'
KeyError: 'j'
```

#### 1.12.4 Parcourir un dictionnaire

**Itération avec for.** La boucle `for` peut être utilisée pour parcourir toutes les **clés** d'un dictionnaire (qui permettent d'accéder aux valeurs associées).

```
# affichage des associations clé:valeur, une par ligne
for key in D:
    print('La clé', key, 'a pour valeur: ', D[key])

# affichage des dates d'anniversaire
dates_naissance={'ingrid':[12,6,1995], 'marc':[27,8,1996], 'brice':[11,10,1995]}
# la variable nom parcourt les clés du dictionnaire (les prénoms)
for nom in dates_naissance:
    # date est la valeur associée au nom, la liste représentant la date de naissance
    date = dates_naissance[nom]
    print(nom, 'fetera son anniversaire le', date[0], '/', date[1])
```

Ce programme affiche les lignes suivantes :

```
ingrid fetera son anniversaire le 12 / 6
marc fetera son anniversaire le 27 / 8
brice fetera son anniversaire le 11 / 10
```

**Liste des clés et des valeurs** La fonction `dict.keys()` permet d'accéder aux clés du dictionnaire `dict`, dans une structure de type `dict_keys`. De même la fonction `dict.values()` permet d'accéder aux valeurs du dictionnaire, dans une structure de type `dict_values`. Ces deux structures ne sont pas indexables, mais on peut les convertir en liste avec la fonction `list()`.

```
>>> dico = {'a':1, 'b':0, 'c':0, 'd':0, 'e':1}
>>> dk = dico.keys()
>>> dk
dict_keys(['a', 'b', 'c', 'd', 'e'])
>>> dk[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_keys' object does not support indexing
>>> dico.values()
dict_values([1, 0, 0, 0, 1])

>>> lc = list(dico.keys())
>>> lc
['a', 'b', 'c', 'd', 'e']
>>> lv = list(dico.values())
>>> lv
[1, 0, 0, 0, 1]
>>> lv[1]
0
```

### 1.12.5 Un exemple détaillé

Dans l'exemple ci-dessus on veut compter chaque lettre dans un mot ou une liste. La fonction reçoit la lettre à compter et le dictionnaire de compteurs, qu'elle met à jour en conséquence. Le programme principal initialise le dictionnaire de compteurs, parcourt la chaîne de caractères, et appelle cette fonction sur chaque lettre lue.

```
# fonction qui reçoit la lettre à compter
# et le dictionnaire de compteurs, qu'elle met à jour
# pas de valeur de retour, modif du dico par effet de bord
def vu_lettre(l,cpts):
    # verification d'existence
    if l in lettres:
        # si existe, incrementation
        cpts[l] = cpts[l] + 1
    else:
        # sinon, initialisation
        cpts[l]=1

# programme principal
mot = "bonjour a tous"
cpts = {}
for lettre in mot:
    vu_lettre(lettre,cpts)
print(cpts)
```

L'exécution de ce programme affiche le dictionnaire de compteurs `'b':1,'o':3,'n':1,'j':1,'u':2,'r':1,' ':2,'a':1,'t':1,'s':1`. Seuls les caractères qui ont été rencontrés ont été ajoutés dans le dictionnaire (y compris l'espace).

### 1.12.6 Copie de dictionnaires

**Affectation** Comme dans le cas des listes, l'affectation d'un dictionnaire vers une variable ne fait que référencer le **même** dictionnaire par un nouvel identificateur. Si on modifie l'un, on modifie aussi l'autre.

```
>>> D = {1: 10, 2: 20, 3: 30}
>>> E = D
>>> E[5] = 50
>>> E
{1: 10, 2: 20, 3: 30, 5: 50}
>>> D
{1: 10, 2: 20, 3: 30, 5: 50}
```

**Copie** Pour créer une copie de surface d'un dictionnaire, on utilise `dict()`.

```
>>> F = dict(D)                # creation d'une copie de D nommee F
>>> F[6] = 60                  # ajout d'une cle dans F
>>> F                          # F a ete modifie
{1: 10, 2: 20, 3: 30, 5: 50, 6: 60}
>>> D                          # mais pas D
{1: 10, 2: 20, 3: 30, 5: 50}
>>> G = F                      # creation d'un 2e nom pour le meme dico
>>> G[7] = 1                   # ajout d'une cle dans G
>>> G                          # G a été modifié
{1: 10, 2: 20, 3: 30, 5: 50, 6: 60, 7: 1}
>>> F                          # mais F aussi
{1: 10, 2: 20, 3: 30, 5: 50, 6: 60, 7: 1}
```



## 1.13 Introduction aux fichiers

Jusqu'à présent, nous avons utilisé `input()` et `print()` pour lire les entrées du programme, et afficher les résultats obtenus. Mais parfois, les données d'entrée sont stockées dans un fichier et on aimerait pouvoir y accéder directement, sans les saisir manuellement au clavier (surtout quand il y en a beaucoup, ou quand on teste plusieurs fois un programme). Souvent, il est aussi utile de sauvegarder les résultats d'un programme dans des fichiers afin de pouvoir y accéder plus tard (par ex : sauvegarde d'une partie dans un jeu vidéo). En Python, il est très facile de lire et d'écrire des données dans des fichiers.

### 1.13.1 Ouverture d'un fichier texte

**Ouvrir un fichier** Avant de commencer la lecture d'un fichier, il faut d'abord l'**ouvrir**. Ouvrir un fichier veut simplement dire que l'on crée une variable qui permet de le manipuler. La fonction `open()` est utilisée pour ouvrir un fichier. Par exemple pour ouvrir un fichier appelé «data.txt», on écrit :

```
>>> f=open('data.txt')
>>> f
<_io.TextIOWrapper name='data.txt' mode='r' encoding='UTF-8'>
```

**Mode d'ouverture** Par défaut, `open()` ouvre un fichier en mode lecture (mode = 'r' pour 'read'), c'est-à-dire qu'on peut lire son contenu mais on ne peut pas le modifier. Il faut donc que le fichier existe. Si on tente d'ouvrir un fichier inexistant en mode «lecture», on déclenche une erreur.

```
>>> f = open('toto')                # le fichier 'toto' n'existe pas
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'toto'
```

### 1.13.2 Lecture d'un fichier texte

Une fois notre fichier texte ouvert, il existe plusieurs manières de lire son contenu.

**Lecture de tout le texte** On peut lire en une seule fois tout le contenu du fichier et le stocker dans une chaîne de caractères, qui contiendra le texte entier.

```
texte = f.read()
```

**Lecture de toutes les lignes** On peut aussi lire en une fois toutes les lignes du fichier et les stocker dans une liste de chaînes de caractères (une chaîne par ligne). Après cette instruction, `lignes` est une liste qui contient toutes les lignes du fichier. On peut y accéder une par une, par exemple `lignes[0]` contient la première ligne, etc.

```
lignes = f.readlines()
```

**Lecture itérative par ligne** Enfin on peut aussi lire le fichier ligne par ligne de manière itérative, avec une boucle `for`. Le programme ci-dessous lit une ligne après l'autre dans le fichier et les affiche au fur et à mesure. On ne stocke donc jamais l'intégralité du fichier en mémoire, seulement une ligne à la fois.

```
for ligne in f:
    print(ligne) # affiche une ligne du fichier
```

**Exemple** Soit le fichier «nombres.txt» qui contient les entiers suivants (un par ligne) : 15 18 30 55 16 3 12 13. Le programme ci-dessous calcule la somme de ces entiers.

```

fichier = open('nombres.txt')          # ouvrir le fichier en lecture
somme = 0                              # initialiser la somme
for nombre in fichier:                 # nombre parcourt les lignes une par une
    # nombre est une chaîne, ne pas oublier de la convertir en entier !
    somme = somme + int(nombre)
# apres la fin de la boucle de calcul, afficher une seule fois la somme
print(somme)
# ne pas oublier de fermer le fichier apres utilisation
fichier.close()

```

### 1.13.3 Écriture dans un fichier texte

**Ouverture en mode écriture** Quand un fichier est ouvert en mode lecture, on ne peut que le lire mais pas le modifier. Pour pouvoir écrire dans un fichier, il faut l'ouvrir en mode écriture. Si le fichier n'existe pas encore il est créé (on ne déclenche donc pas d'erreur). On remarque que le mode vaut maintenant 'w' pour 'write', ce qui indique qu'on peut écrire dans ce fichier.

```

>>> f = open('data.txt','w')
>>> f
<_io.TextIOWrapper name='data.txt' mode='w' encoding='UTF-8'>

```

**Ouverture en mode ajout** Si le fichier ouvert en mode «écriture» n'existe pas, il est **créé** (vide). Mais **attention!** S'il existe déjà, tout son contenu est **effacé**. Pour pouvoir écrire à la suite du texte présent dans un fichier déjà existant, il faut l'ouvrir en mode 'ajout' ('append').

```

>>> f = open('data.txt','a')
>>> f
<_io.TextIOWrapper name='data.txt' mode='a' encoding='UTF-8'>

```

**Écriture** La fonction permettant d'écrire dans un fichier texte est `write()`. Contrairement à `print()`, la fonction `write()` ne saute pas de ligne automatiquement. Elle écrit dans le fichier exactement la chaîne de caractères reçue en argument. Pour sauter une ligne dans le fichier, il faut écrire un saut de ligne manuellement avec le caractère spécial `\n` (voir d'autres caractères spéciaux dans le cours sur les chaînes de caractères).

```

f.write('ce texte sera écrit dans le fichier')
f.write('\n') # ceci permet de passer à la ligne

```

**Argument de write** La fonction `write()` reçoit exactement un argument, qui doit obligatoirement être une chaîne de caractères. Pour écrire un entier ou une valeur d'un autre type, il faut le convertir en chaîne de caractères en utilisant `str()` et la concaténer à la chaîne à écrire.

```

>>> m = 12
>>> f.write("m=",m)          # on ne peut pas passer plusieurs arguments
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() takes exactly one argument (2 given)
>>> f.write(m)               # on ne peut pas écrire un entier
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() argument must be str, not int
>>> s = "m="+str(m)          # creation de la chaine a ecrire
>>> f.write(s)               # ecriture : ok
4

```

**Valeur de retour de write** La fonction `write()` renvoie le nombre de caractères effectivement écrits dans le fichier. Dans l'exemple ci-dessus on a écrit la chaîne "m=12" donc 4 caractères.

### 1.13.4 Fermeture

Une fois la lecture/écriture terminée, il faut fermer le fichier en utilisant la fonction `close()`. Une fois fermé, on ne peut plus lire ou écrire dans le fichier.

```
f = open('fichier.txt')
# lecture iterative ligne par ligne
for ligne in f:
    print('une ligne lue :', ligne)
f.close()

>>> f.write('tata')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

**Exemple** La fonction suivante permet de sauvegarder les 10 premières puissances de 2 dans un fichier texte nommé «`puis.txt`» (une par ligne). Après exécution de ce programme, le fichier texte `puis.txt` contiendra donc les entiers suivants (un par ligne) : 1 2 4 8 16 32 64 128 256 512, c'est-à-dire les puissances de 2 entre  $2^0$  et  $2^9$ .

```
# ouverture en ecriture (creation si inexistant, ecrasement si existe)
fichier = open('puis.txt', 'w')
# 10 iterations pour les 10 puissances
for i in range(0,10):
    # il faut convertir l'entier en chaîne de caractères
    # et passer a la ligne explicitement
    fichier.write(str(2 ** i) + '\n')
# fin de la boucle, on peut fermer le fichier
fichier.close()
```



# Chapitre 2

## Exercices de TD

### 2.1 TD1 : les bases

#### 2.1.1 Échange des valeurs de deux variables

On considère le programme ci-dessous :

```
print("Entrez deux reels :")
x = float(input())
y = float(input())
print("Vous avez saisi ",x," et ", y)
x = y
y = x
print("Après échange des variables : ",x , " et ",y)
```

1. Dire ce qui s'affiche quand on exécute ce programme en donnant 1.23 et -17.5 en entrée.
2. Ce programme fait-il bien ce qu'il est censé faire et sinon comment le corriger ?
3. Compléter ce programme pour lire, échanger et afficher les valeurs de 3 variables.

#### 2.1.2 Affichages

Écrire un programme qui demande à l'utilisateur 2 entiers A et B, et qui les affiche de la manière suivante :

1. Un par ligne
2. Sur la même ligne séparés par une virgule : A,B
3. Avec un message donnant le nom et la valeur : A=... et B=...
4. En damier

```
A B A
B A B
A B A
```

5. Sur la même ligne séparés par des tirets et avec un point d'exclamation final : A-B-A-B-!

### 2.1.3 Entrées, sorties, calculs

Écrire un programme qui demande à l'utilisateur 2 réels  $x$  et  $y$ , calcule leur somme et leur produit, et les affiche dans un message explicite. Exemple d'exécution :

```
x=? 3.2
y=? 7
3.2 + 7 = 10.2
3.2 * 7 = 22.4
```

### 2.1.4 Bonjour

Écrire un programme qui demande à l'utilisateur son prénom et affiche un message de salutations personnalisé. Exemple d'exécution :

```
Comment t'appelles-tu ? Toto
Bonjour Toto !
```

## 2.2 TD2 : instruction conditionnelle, booléens

### 2.2.1 Un peu de logique

Compléter ci-dessous avec le nombre de valeurs de l'entier A pour lesquelles les expressions suivantes sont vraies (et spécifier quelles valeurs), comme dans l'exemple.

A == 8 or A == 9  
-> 2 (8 et 9)

A > 0 and A < 20  
-> 19 (Les entiers compris entre 1 et 19 inclus)

A >= 3 and A < 5  
-> ...

A >= 3 or A < 5  
-> ...

A > 0 and A < 10 and A % 3 == 0 and A % 2 != 0  
-> ...

A <= 0 and A >= 20  
-> ...

### 2.2.2 Expressions booléennes

On considère qu'on a initialisé les variables a,b,c de type entier. Écrire des expressions booléennes représentant les faits suivants :

1. a est impair
2. (au moins) un des 3 entiers a,b,c est impair
3. (exactement) un seul des 3 entiers a,b,c est impair
4. a est multiple de b
5. a,b,c sont compris entre 0 (inclus) et 20 (exclus)
6. a est soit (strictement) négatif, soit positif pair
7. b est nul ou impair

### 2.2.3 Négations d'expressions booléennes

Donner la négation simplifiée (pas d'opérateur not) des expressions booléennes suivantes :

1.  $a < 10$  or  $a > 15$
2.  $a \% 3 != 0$  and  $a >= 20$
3.  $a \% b == 0$  or  $b \% a == 0$
4.  $(a >= b$  or  $a >= c)$  and  $(a \% 2 == 0$  or  $a \% 3 == 0)$

Donner aussi la lecture en français de ces négations.

### 2.2.4 Comparaison de deux programmes

Les deux programmes suivants sont-ils semblables ? Justifier votre réponse.

<pre>rep = input("Etes-vous fumeur : ")  if (rep == "oui") or (rep == "OUI") :     print("Vous devriez arreter") else :     print("Continuez ainsi")     print("Merci de votre participation")</pre>	<pre>rep = input("Etes-vous fumeur : ")  if (rep == "oui") or (rep == "OUI") :     print("Vous devriez arreter") else :     print("Continuez ainsi")     print("Merci de votre participation")</pre>
--	--

### 2.2.5 Comparaison de deux programmes (bis)

Les deux programmes suivants sont-ils semblables ? Justifier votre réponse.

<pre>n = int(input("Entrez n : "))  if n%2 == 1 :     n = 3*n+1 else :     n = n//2  print(n)</pre>	<pre>n = int(input("Entrez n : "))  if n%2 == 1 :     n = 3*n+1 if n%2 == 0 :     n = n//2  print(n)</pre>
---	--

### 2.2.6 Bonjour (v2)

Écrire un programme qui :

1. Demande à l'utilisateur son prénom
2. Demande à l'utilisateur sa langue (français ou anglais)
3. Si l'utilisateur parle français, lui affiche "bonjour ..." (avec son prénom)
4. Si l'utilisateur parle anglais, lui affiche "hello ..." (avec son prénom)
5. Si l'utilisateur saisit une autre langue, affiche un message d'erreur

### 2.2.7 Encore des expressions booléennes

1. Écrire les expressions booléennes qui représentent les assertions suivantes :

- $a$  est un entier pair
- $a$  est un multiple de 3 ou un multiple de 7
- $b$  est un entier strictement négatif multiple de 3
- $c$  est une lettre minuscule de l'alphabet
- $c$  est une lettre de l'alphabet (majuscule ou minuscule)
- $c$  est un caractère numérique
- `rep` n'est ni la lettre 'o' ni la lettre 'n'
- l'entier  $b$  n'est pas compris entre 0 et 20 inclus

2. Écrire la négation des expressions booléennes ci-dessus, et les simplifier autant que possible (c'est-à-dire qu'on ne veut pas utiliser l'opérateur `not`).



### 2.2.8 Ou exclusif

1. Écrire l'expression booléenne correspondant au `ou_exclusif` entre 2 booléens `a` et `b`. On rappelle que le OU EXCLUSIF signifie que exactement l'un des 2 booléens est vrai (mais pas les 2 en même temps).
2. Faire sa table de vérité

### 2.2.9 Maximum de trois nombres

Écrire un programme qui demande trois entiers et qui affiche le plus grand des trois.

- Version 1 : demander d'abord les 3 nombres, puis les comparer entre eux. Combien y a-t-il de cas à considérer (donc de comparaisons faites) ?
- Version 2 : garder en mémoire un maximum temporaire, et lui comparer chaque nouvel entier lu. Dans ce cas combien fait-on de comparaisons ?
- Bonus avec une boucle : lire `n` entiers et afficher le maximum et le minimum.

### 2.2.10 Année bissextile

Écrire un programme qui détermine si une année saisie par l'utilisateur est bissextile ou non en affichant "L'année XXX est bissextile" ou "L'année XXX n'est pas bissextile" selon le cas. On rappelle que : une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100 ; toutefois, elle est considérée comme bissextile si c'est un multiple de 400.

### 2.2.11 Jours, heures, minutes, secondes

1. Écrire un programme en Python qui lit un nombre de secondes (entier) au clavier et puis affiche cette durée en jours, heures, minutes et secondes au format de l'exemple suivant :

Donner un nombre de secondes : 93901

La durée saisie se décompose en : 1 jour(s), 2 heure(s), 5 minute(s), 1 seconde(s)

2. Ajouter la gestion des cas particuliers : si une valeur vaut 0 alors on n'affiche pas cet élément, si elle vaut 1 on l'affiche au singulier, sinon on l'affiche au pluriel.

Donner un nombre de secondes : 93900

La durée saisie se décompose en : 1 jour, 2 heures, 5 minutes

### 2.2.12 Quiz

On veut écrire un programme de quiz (voir fonctionnement attendu ci-dessous).

1. Commencer par écrire un programme qui pose uniquement la première question et teste la réponse de l'utilisateur. Attention : combien y a-t-il de cas à envisager ?
2. Compléter maintenant ce programme pour poser les 3 questions successivement.
3. Compléter ce programme pour compter le nombre de bonnes réponses et l'afficher à la fin.
4. Bonus avec boucle `while` : modifier ce programme pour qu'il propose à l'utilisateur de rejouer à chaque fin de partie ; tant que l'utilisateur accepte, le programme démarre un nouveau quiz avec 3 nouveaux entiers ; quand l'utilisateur refuse, le programme se termine.
5. Bonus avec fonctions : écrire une fonction par question (qui pose la question, lit la réponse, et renvoie le résultat de l'utilisateur (juste ou faux)) ; les appeler dans le programme principal qui gère le quiz.

**Fonctionnement attendu** Le programme demande trois entiers a, b, c à l'utilisateur, puis lui pose successivement les questions suivantes :

- $a < b < c$  ?
- un seul nombre impair parmi a, b, c ?
- a, b, c distincts deux à deux ?

Pour chaque question, le programme doit contrôler la réponse, et doit afficher "réponse correcte, Bravo" ou "réponse incorrecte" selon le cas ; il doit aussi compter le nombre de bonnes réponses et, à la fin, afficher le score sous la forme : "x bonnes réponses sur 3".

**Exemple d'exécution du programme** (conformez vous strictement à ce modèle).

```
Donner trois entiers :
a=? 1 (Entrée)
b=? 4 (Entrée)
c=? 3 (Entrée)

Jeu du "Vrai ou Faux ?" (répondez en tapant V ou F)

Question 1.
1 < 4 < 3 ?
V (Entrée)
réponse incorrecte

Question 2.
y a-t-il un seul nombre impair parmi 1, 4, 3 ?
F (Entrée)
réponse correcte, Bravo

Question 3.
1, 4, 3 sont-ils distincts deux à deux ?
V (Entrée)
réponse correcte, Bravo

2 bonnes réponses sur 3
Le jeu est terminé : Au revoir !
```

### 2.2.13 Multiples

Écrire un programme qui demande à l'utilisateur un entier entre 1 et 100. Si l'entier ne respecte pas ces bornes, le programme se termine sur un message d'erreur. Sinon il :

1. affiche uniquement le premier entier entre 2 et 10 dont x est un multiple (plus petit diviseur)
2. affiche tous les entiers entre 2 et 10 dont x est un multiple

Quelle est la différence entre les 2 versions? *Remarque : on verra bientôt comment écrire des boucles qui permettent d'éviter de telles répétitions de code.*

### 2.2.14 Dé pipé

On utilisera ici le module `random` vu en cours. Écrire un programme qui joue une partie de dés entre 2 joueurs :

- demande les prénoms des 2 joueurs
- pour le premier joueur, on utilise un vrai dé 6 : tirer un nombre au hasard entre 1 et 6 et l'afficher sous la forme “JJJ fait XXX” (en remplaçant JJJ par le prénom et XXX par le tirage)
- pour le 2e joueur, on utilise un dé pipé : tirer un nombre au hasard entre 1 et 3, l'afficher sous la même forme
- déterminer le vainqueur (celui qui a fait le plus gros tirage) et afficher son prénom sous la forme “JJJ gagne”

Extension avec d'autres dé pipés :

- Écrire une instruction qui ne tire que des faces impaires d'un dé 6
- Écrire une instruction qui tire 1 avec une probabilité de 30%, 2 avec une probabilité de 20%, 3 ou 4 avec une probabilité de 15% chacun, 5 ou 6 avec une probabilité de 10% chacun.

## 2.3 TD3 : itération conditionnelle while

### Bases des itérations conditionnelles

#### 2.3.1 Code à trous

Soit le programme incomplet suivant :

```
i = <A>
while <B>:
    print(i)
    i = <C>
```

Donner les expressions <A>, <B>, <C> pour que :

1. Le programme affiche les entiers allant de 0 à 10.
2. Le programme affiche les entiers allant de 10 à 0.
3. Le programme affiche les multiples de 3 strictement inférieurs à 30 (0 compris).
4. Le programme affiche les puissances de 3 strictement inférieures à 300 (1 compris).
5. Le programme affiche un entier saisi par l'utilisateur tant que celui-ci est strictement positif.
6. Le programme affiche une lettre saisie par l'utilisateur tant que celle-ci est une voyelle.
7. Le programme affiche une lettre saisie par l'utilisateur jusqu'à ce que celle-ci soit une voyelle.

#### 2.3.2 Lecture de code

Combien de '\*' les programmes suivants affichent-ils ? Justifier les réponses.

```
# (a)
i = 0
while i <= 15 :
    print('*')
    i = i + 1
```

```
# (b)
i=0
while i>10 :
    print('*')
    i=i+1
```

```
# (c)
i=20
while i<100 :
    if i%2!=0 :
        print('*')
    i=i+1
```

```
# (d)
i = 0
while i < 100 :
    if i % 2 == 0 :
        print('*')
    i = i + 1
```

```
# (e)
i=1
while i<=16 :
    print('*')
    i=i*2
```

```
# (f)
i=0
while i<10 :
    print('*')
    i=i-1
```

Bonus : le(s)quel(s) de ces programmes devrai(en)t être modifié(s), pourquoi, et comment ?

#### 2.3.3 Fizz Buzz

Écrire un programme qui affiche tous les entiers entre 1 et 1000, mais qui remplace les multiples de 3 par "fizz", les multiples de 5 par "buzz", et les multiples à la fois de 3 et 5 par "fizzbuzz". Exemple d'exécution (début) :

```
1
2
fizz
4
buzz
fizz
```

```
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
16
17
...
```

## Boucle while pour filtrer les entrées

### 2.3.4 Dé pipé (le retour)

Écrire un programme qui tire un nombre au hasard jusqu'à obtenir un nombre pair. Il affiche alors le nombre pair tiré. Bonus : afficher aussi le nombre de tirages nécessaires.

### 2.3.5 Voyelle

Écrire un programme qui demande à l'utilisateur de saisir une lettre et qui continue jusqu'à ce que l'utilisateur saisisse une voyelle.

1. Version 1 : avec instruction **break**
2. Version 2 : sans instruction **break**

### 2.3.6 Mot de passe

1. Écrire un programme dans lequel vous définissez un mot de passe (par exemple `mdp = 'abcdef'`), puis qui demande à l'utilisateur de saisir un mot de passe pour accéder. Si l'utilisateur saisit le bon mot de passe, le programme affiche le message 'Acces autorisé' et se termine, sinon il affiche 'Acces refuse' et redemande.
2. Modifier ce programme pour que l'utilisateur n'ait droit qu'à 5 essais pour saisir le mot de passe (après 5 erreurs, accès refusé).
3. Si vous avez utilisé **break** dans la 2e version, réécrire le programme sans l'utiliser.

## Boucle while, accumulateurs et drapeaux

### 2.3.7 Accumulateurs simples

Écrire un programme qui :

1. Calcule et affiche la somme des n premiers entiers
2. Calcule et affiche la somme des n premiers entiers impairs (n entiers impairs, et pas entiers impairs inférieurs à n)
3. Calcule et affiche la somme de n entiers lus au clavier
4. Calcule et affiche le produit de n entiers lus au clavier
5. Lit n paires d'entiers, compare les entiers 2 à 2, et affiche à la fin un booléen indiquant si toutes les paires étaient en ordre croissant (cf exemple pour n=3).

```

Paire d'entiers #1
3
7
Paire d'entiers #2
2
12
Paire d'entiers #3
5
9
Toutes les paires sont en ordre croissant
...
Paire d'entiers #3
9
5
Toutes les paires ne sont pas en ordre croissant

```

### 2.3.8 Moyenne, minimum, maximum

Écrire un programme qui demande à l'utilisateur de saisir des nombres réels (float) correspondant à ses notes d'examen. L'utilisateur donnera une note invalide (non comprise entre 0 et 20) pour indiquer la fin de la saisie (cette note invalide ne sera pas comptabilisée). Après la fin de la saisie le programme doit afficher la moyenne des notes (valides) saisies (*attention* au cas spécial où aucune note valide n'est saisie).

Compléter ce programme pour afficher aussi la meilleure note et la pire note de l'utilisateur.

### 2.3.9 Factorielle

Écrire un programme qui lit un entier  $n$  saisi par l'utilisateur, et qui affiche  $n!$  (factorielle de  $n$ ). Rappel :  $n! = 1 \times 2 \times 3 \times 4 \times \dots \times n$

*Indice* : Attention à la valeur initiale.

Bonus avec une boucle imbriquée : proposer à l'utilisateur de rejouer avec une nouvelle valeur de  $n$ .

### 2.3.10 Séquence croissante

Écrire un programme qui lit une suite d'entiers au clavier jusqu'à ce que l'utilisateur saisisse un nombre strictement négatif. Le programme affichera alors 'Croissante' si la suite d'entiers était croissante, 'Décroissante' si la suite était décroissante, 'constante' si tous les entiers étaient égaux, et 'Ni croissante ni décroissante' si la suite n'est ni croissante ni décroissante. *Indice* : utiliser des drapeaux.

## Boucle while pour rejouer un programme

### 2.3.11 Nombres premiers (simple)

Écrire un programme qui demande à l'utilisateur de saisir un entier positif  $N$ , et qui vérifie si  $N$  est premier. (*Rappel* : un nombre est premier s'il n'a pas de diviseurs autres que 1 et lui même. Ce programme affiche le résultat, puis propose à l'utilisateur de recommencer avec un nouveau nombre, jusqu'à ce que celui-ci réponde 'non'. Exemple d'exécution :

```

Programme de test de nombres premiers
Entre un entier ? 7
7 est premier
Rejouer ? oui
Entre un entier ? 10
10 n'est pas premier
Rejouer ? non
Fin du programme

```

**Question** : combien de diviseurs faut-il tester avant de déterminer si  $N$  est premier ?

### 2.3.12 Dé pipé : multi-parties

On reprend l'exercice du dé pipé vu au TD précédent.

1. Écrire un programme qui oppose 2 joueurs, l'un lance un vrai dé 6, l'autre lance un dé pipé qui l'avantage (par exemple lancer 3 dés puis garder le maximum).
2. Modifier ce programme pour jouer N parties (par exemple 20) et afficher le nombre de victoires du joueur honnête (dé non pipé).
3. Modifier le premier programme pour jouer jusqu'à ce que le joueur honnête gagne, et afficher le nombre de parties qui auront été nécessaires.

## Boucle while, suites et récurrence

### 2.3.13 Suite de Fibonacci

Soit la suite de Fibonacci donnée par

$$\begin{cases} f_0 = 1 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \end{cases}$$

1. Écrire un programme qui lit un entier n au clavier, puis calcule et affiche  $f_n$
2. On peut estimer le nombre d'or en divisant un terme de la suite par le précédent. Plus n grandit et plus l'estimation est précise. Compléter ce programme pour afficher l'estimation du nombre d'or  $f_n/f_{n-1}$  (on affiche uniquement la dernière, pas les valeurs intermédiaires).
3. Bonus plus difficile : définir une constante  $nbor = 1,618033988749895$  (approximation du nombre d'or). Modifier ce programme pour qu'il demande à l'utilisateur une précision souhaitée (distance maximale autorisée avec le nombre d'or, par exemple  $p = 0.01$ ), puis il calcule les termes de la suite et l'estimation du nombre d'or  $estimnbo$  jusqu'à ce que la précision soit atteinte, c'est-à-dire que la distance entre  $nbor$  et  $estimnbo$  est inférieure ou égale à p.

### 2.3.14 Syracuse

Soit la suite récursive donnée par  $u_0 = A$  et par la relation de récurrence :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3 * u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

*NB : la grande question est de savoir si, pour toute valeur initiale A cette suite contient l'élément 1. Les mathématiciens n'ayant pas résolu ce problème, on dit que ce problème est ouvert. On dit aussi que "cette suite passe par 1 quel que soit le point de départ" est la conjecture de Syracuse.*

1. Écrire un programme qui demande la saisie d'un entier strictement positif et détermine pour quel n on a :  $u_n = 1$  (c'est la "durée de vol")
2. Modifier ce programme pour qu'il affiche tous les termes  $u_n$  calculés au fur et à mesure.
3. Modifier ce programme pour qu'il affiche le maximum atteint par la suite (la plus grande valeur d'un terme  $u_n$ ).
4. Cette suite est donc croissante quand un terme est impair (et le suivant est pair), décroissante tant que les termes sont pairs. Bonus : modifier ce programme pour qu'il affiche la longueur de la plus longue descente (suite décroissante), en nombre de termes.





### 2.3.19 Nombres premiers (bis)

Le bout de programme suivant vérifie si un nombre N est premier et met le résultat dans le drapeau `est_premier`.

```
est_premier=True
i = 2
while i * i <= N and est_premier:
    if N % i == 0:
        est_premier=False
    i = i + 1
```

1. Comprendre et expliquer le fonctionnement de ce code
2. L'utiliser pour écrire un premier programme qui affiche les 100 premiers nombres premiers. Bonus : les afficher sur une seule ligne, séparés par des virgules. Bonus : éviter la virgule finale.
3. L'utiliser pour écrire un 2e programme qui :
  - lit deux entiers positifs A et B
  - filtre ces entiers pour qu'ils soient positifs, et que B soit supérieur ou égal à A
  - affiche 'Oui' s'il existe au moins un nombre premier entre A et B inclus (et affiche ce nombre premier), et 'Non' sinon.

## 2.4 TD4 : chaînes de caractères

*Remarque* : les exercices suivants demandent parfois d'écrire des fonctions. Si vous n'avez pas encore vu les fonctions, vous pouvez à la place écrire un programme principal qui fait la même chose.

*Remarque2* : Python dispose déjà de fonctions `isalpha`, `lower`, `upper`, `islower`, `isupper`, `capitalize`, mais on s'interdit de les utiliser ici.

### 2.4.1 Pyramide alphabétique

Écrire un programme qui affiche l'alphabet sous forme d'un triangle (en complétant par des tirets après la fin de l'alphabet).

```

a                                     a
b c                                 b c
d e f                             d e f
g h i j                           g h i j
k l m n o                         k l m n o
p q r s t u                       p q r s t u
v w x y z - -                     v w x y z - -

```

### 2.4.2 Lecture de mots

Écrire les programmes suivants :

1. demande à l'utilisateur de taper un mot au clavier, jusqu'à ce que ce mot commence par 'a' (en majuscule ou en minuscule), afficher alors le nombre d'erreurs faites avant la bonne saisie.
2. demander à l'utilisateur un mot, afficher sa longueur, lui proposer de rejouer ; à la fin, afficher le mot le plus long qui a été lu.
3. demander à l'utilisateur de saisir des mots jusqu'à ce qu'il indique 'stop' ; afficher alors combien de mots ont été lus ('stop' ne compte pas), le mot le plus long, et sa longueur.
4. demander des mots jusqu'à la saisie d'une chaîne vide pour arrêter, et afficher alors lequel des mots lus est le premier dans l'ordre alphabétique.

### 2.4.3 Caractère alphabétique

Écrire un programme / une fonction qui reçoit une chaîne de caractères et teste si c'est une lettre de l'alphabet (majuscule ou minuscule). Cette fonction renvoie un booléen (vrai pour une lettre, qu'elle soit majuscule ou minuscule ; faux pour un autre caractère ou pour une chaîne de plusieurs caractères).

**Remarque** : il faut donc vérifier d'abord si la chaîne est bien de longueur 1, c'est-à-dire ne comprend qu'un seul caractère. Rappel : la fonction `len` donne la longueur d'une chaîne ou d'une liste.

### 2.4.4 Minuscule ou majuscule ?

1. Écrire un programme qui lit au clavier une lettre de l'alphabet, et teste si elle est en minuscule ou en majuscule. Remarque : Python fournit déjà des méthodes `isupper()` et `islower()` sur les chaînes de caractères, mais on demande de ne pas les utiliser.
2. Modifier ce programme pour qu'il propose de rejouer avec une nouvelle lettre.
3. Bonus avec fonctions : à la place d'un programme, écrire une fonction `est_minuscule` qui reçoit la lettre en paramètre et renvoie un booléen indiquant si elle est minuscule (True) ou majuscule (False).

### 2.4.5 Position dans l'alphabet

1. Écrire un programme qui lit un caractère et affiche sa position dans l'alphabet, ou -1 si ce n'est pas une lettre. *Remarque : la lettre b et minuscule ou B en majuscule ont la même position, à savoir 2. Rappel : la fonction prédéfinie `ord` reçoit un caractère et renvoie son code ASCII.*
2. Écrire ensuite un programme qui lit un entier (entre 1 et 26) et un caractère ('m' pour minuscule ou 'M' pour majuscule) et affiche la lettre située à cette position dans l'alphabet, en minuscule/majuscule selon. *Indice : la fonction prédéfinie `chr` fait la conversion inverse, du code ASCII vers le caractère correspondant.*
3. Bonus avec fonction : au lieu d'un programme principal, écrire 2 fonctions qui reçoivent le caractère, ou l'entier pos et un booléen maj, en paramètre, et renvoient le résultat. Écrire un programme principal qui appelle ces fonctions et affiche le résultat.

### 2.4.6 Décalage

Écrire une fonction qui reçoit un caractère `x` et un entier `n` :

- Si ce caractère n'est pas une lettre, il est renvoyé tel quel
- Si c'est une lettre de l'alphabet (majuscule ou minuscule), la fonction renvoie le caractère situé `n` positions après `x` (sans changer la casse : une majuscule reste une majuscule).
- Si on dépasse la lettre 'z' il faut revenir à 'a' ; si on dépasse le 'Z' il faut revenir à 'A'. Par exemple 3 positions après 'y' on veut obtenir 'b' ; 3 positions après 'Y' on veut obtenir 'B'.

### 2.4.7 Inversion de casse

Écrire une fonction qui reçoit un caractère. Si c'est une lettre minuscule, elle renvoie la même lettre en majuscule. Si c'est une majuscule, elle renvoie la même lettre en minuscule. Si ce n'est pas une lettre, elle renvoie le caractère tel quel. On s'interdit d'utiliser les fonctions Python `upper()` et `lower()`.

## 2.5 TD5 : fonctions

### 2.5.1 Comprendre les fonctions déjà connues

Pour chacun des appels de fonction ci-contre :

1. Quel est le nom de la fonction ?
2. Combien y a-t-il d'arguments ?
3. Pour chaque argument (s'il y en a), donnez son type et sa valeur.
4. Y a-t-il une valeur de retour, et si oui laquelle (valeur et type) ?
5. Y a-t-il des "effets de bord" (interactions entrées/sorties, dessin turtle...) provoqués par l'appel ?

```
import math
import random
import turtle

n = int(input("Borne sup ?"))
x = random.randint(0,10)
print("Nombre aleatoire: ", x)

text=input("Quel est votre nom?")
print("Bonjour",text)

turtle.up()
turtle.forward(100)
turtle.down()
turtle.circle(50.5)

z=math.sqrt(5.5)
print("La racine de 5.5 est", z)
```

### 2.5.2 Lecture

Prédire le fonctionnement des 3 programmes ci-dessous.

#### Programme 1

```
def f(x,y):
    c=x**2+y**2
    return c
# programme principal
a=3
b=4
d=f(a,b)
print("d=", d)
d=f(4,5)
print("d=", d)
```

#### Programme 2

```
def est_voyelle(l):
    if l=="a" or l=="e" or l=="i" or l=="o" or l=="u" or l=="y":
        return True
    else:
        return False
def demande_voyelle():
```

```

reponse=""
while not(est_voyelle(reponse)):
    reponse=input("Choisissez une lettre : ")
    if not(est_voyelle(reponse)):
        print("Ceci n'est pas une voyelle.")
    return reponse
# programme principal
l=demande_voyelle()
print("La voyelle que vous avez choisi est:", l)
demande_voyelle()

```

### Programme 3

```

def somme_premiers_entiers(n, modeAffichage):
    i=1
    somme=0
    while i<n:
        somme=somme+i
        if modeAffichage: # pareil que if modeAffichage==True
            print("0+ ... +", i, "=", somme)
        i=i+1
    return somme
# programme principal
s=somme_premiers_entiers(5,False)
print("s=", s)
modeAffichage=True
s=somme_premiers_entiers(5,modeAffichage)
print("s=", s)

```

### 2.5.3 Calcul de polynome

Écrire une fonction qui reçoit 3 entiers  $a, b, c$  et un réel  $x$  et qui calcule et renvoie la valeur du polynôme  $ax^2 + bx + c$ .

Écrire ensuite un programme principal qui :

- demande à l'utilisateur 2 réels  $y$  et  $z$  ; utilise la fonction ci-dessus pour calculer les polynômes  $3y^2 - 4y + 7$  et  $7z^2 - 10z - 5$  ; affiche les valeurs avec un message clair
- tire au hasard 3 entiers  $h_1, h_2, h_3$  (rappel : fonction `randint` du module `random`) entre -10 et 10, et utilise la fonction ci-dessus pour calculer la valeur du polynôme  $9h_1 - 3h_2 + h_3$ , puis affiche la valeur

### 2.5.4 Dates

Écrire une fonction qui reçoit un jour, un mois et une année (3 entiers) et qui teste si la date correspondante est correcte. Cette fonction renvoie donc un booléen. Par exemple le 31/2/2012 n'est pas correct. Le 20/7/2015 est correct.

On pourra écrire une fonction auxiliaire qui teste si un numéro de jour donné est correct pour un mois donné.

### 2.5.5 Somme des chiffres

Écrire une fonction qui reçoit un entier (par exemple 37251) et renvoie la somme de tous ses chiffres (ici :  $3+7+2+5+1=18$ ). On a donc besoin pour commencer de savoir combien de chiffres a ce nombre :

1. Méthode 1 (facile) : on peut mesurer la longueur de la chaîne de caractères correspondante pour trouver le nombre de chiffres.
2. Méthode 2 : on s'interdit de convertir l'entier en chaîne. *Indice* : utiliser l'opérateur modulo pour trouver son nombre de chiffres.

### 2.5.6 Conversions binaire-décimal

Écrire une fonction qui prend en argument un entier `n`, et renvoie une chaîne de caractères représentant son écriture en binaire. Par exemple cette fonction appelée sur l'entier 9 renvoie la chaîne de caractères "1001".

Écrire ensuite la fonction inverse, qui reçoit une chaîne de caractères représentant un entier en binaire, et qui renvoie un entier égal à sa valeur en base 10. Par exemple cette fonction appelée sur la chaîne "1100" renvoie l'entier 12; avec la chaîne "11001" elle renvoie l'entier 25.

Écrire enfin un programme principal qui effectue la boucle suivante :

- propose à l'utilisateur 3 choix : conversion binaire→décimal, décimal→binaire, ou arrêter
- s'il choisit d'arrêter, la boucle se termine
- s'il choisit de faire une conversion décimal vers binaire, le programme lui demande un entier et affiche sa conversion,
- s'il choisit de faire une conversion binaire vers décimal, le programme lui demande une chaîne de caractères et affiche sa conversion en entier décimal
- puis le programme recommence jusqu'à ce que l'utilisateur choisisse d'arrêter

**Remarque :** Python fournit des fonctions prédéfinies permettant de faire ces conversions (mais on ne veut pas les utiliser dans cet exercice).

- La fonction `bin(x)` renvoie une chaîne de caractères représentant l'écriture binaire de l'entier `x`. Par exemple `bin(11)` renvoie la chaîne '0b1011', le préfixe '0b' indiquant qu'il s'agit d'une écriture binaire.
- La fonction `int(chaine,base)` renvoie l'entier décimal représenté par la chaîne de caractères donnée dans la base donnée. Par exemple `int('1010',10)` interprète la chaîne '1010' en base 10 et renvoie donc l'entier décimal 1010; `int('1010',2)` interprète la chaîne '1010' en base 2 (en binaire) et renvoie donc l'entier décimal 10.

### 2.5.7 Maximum

*Note : Il existe déjà une fonction Python pour calculer le maximum, elle s'appelle `max`. Cependant, dans cet exercice, nous vous demandons de ne pas l'utiliser.*

1. Écrire une fonction `maximum` qui prend en arguments deux nombres et qui renvoie le plus grand des deux.
2. Écrire un programme principal qui demande deux entiers à l'utilisateur, puis qui affiche **Le plus grand des deux est: ....** Vous ferez un **appel** à votre fonction `maximum`.
3. Écrire une fonction `maximum3` qui prend en arguments trois nombres et qui renvoie le plus grand des trois. Vous ferez appel à votre fonction `maximum`.
4. Écrire une fonction `maximum3_input` qui ne prend pas d'arguments, qui demande trois nombres à l'utilisateur puis qui renvoie le maximum des 3 nombres. Vous ferez appel à votre fonction `maximum3`.
5. Écrire un programme principal utilisant `maximum3_input`, qui demande trois nombres à l'utilisateur puis qui affiche le plus grand des trois.
6. Écrire une fonction `max_input` qui lit des entiers strictement positifs au clavier (jusqu'à lire un entier nul qui signifie fin de saisie), et en fin de série affiche le maximum lu.

### 2.5.8 Moyenne pondérée

1. Écrire une fonction `moyenne_ponderee` qui prend en arguments quatre nombres que l'on appellera `note1`, `note2`, `coeff1` et `coeff2` et qui calcule la moyenne pondérée de `note1` et `note2`, ayant pour coefficients respectifs `coeff1` et `coeff2`.
2. Écrire un programme principal qui demande deux notes puis deux coefficients à l'utilisateur, puis qui affiche la moyenne pondérée. Vous ferez un appel à votre fonction `moyenne_ponderee`.

3. Bonus : modifier ce programme pour qu'il demande à l'utilisateur le nombre de notes ( $n$ , par exemple 2 ci-dessus), puis la saisie des  $n$  notes et  $n$  coefficients, puis affiche la moyenne pondérée. Indice : utiliser une boucle.

### 2.5.9 Ajoutez des lettres

1. Écrire une fonction `ajoute_suffixe` qui prend en argument deux chaînes de caractères `chaine` et `suffixe`, et un entier `nb_fois` et qui renvoie la chaîne de caractères obtenue à partir de `chaine` en ajoutant `nb_fois` fois la chaîne `suffixe` à la fin. Par exemple, l'appel `ajoute_suffixe("hello", "you", 3)` doit renvoyer `"helloyouyouyou"`.
2. Écrire une fonction `ajoute_b` qui prend en argument une chaîne de caractères `chaine` et un entier `nb_fois` et qui renvoie la chaîne de caractères obtenue à partir de `chaine` en ajoutant `nb_fois` fois la lettre `b` à la fin. Par exemple, `ajoute_b("bonjour", 5)` doit renvoyer `"bonjourbbbb"`. Vous pouvez utiliser votre fonction précédente.
3. Écrire un programme principal qui demande à l'utilisateur un mot, un entier et un suffixe, puis qui teste chacune des deux fonctions avec les bons arguments, puis qui affiche le résultat. Par exemple :  
Donnez un mot: *merci*  
Donnez un nombre: *3*  
La fonction "ajoute\_b" appelée avec ces arguments renvoie : *mercibbb*  
Donnez un suffixe: *la*  
La fonction "ajoute\_suffixe" appelée avec ces arguments renvoie : *mercilalala*

### 2.5.10 Dé

1. Écrire une fonction qui prend un argument un entier et qui teste si la valeur donnée est celle d'un dé (entre 1 et 6). Si oui, la fonction affiche **Valeur correcte** et renvoie `True`, sinon la fonction affiche **Valeur incorrecte** et renvoie `False`.
2. Écrire un programme principal qui demande à l'utilisateur une première valeur de dé puis qui teste si la valeur est correcte grâce à la fonction précédente (avec l'affichage correspondant) ; puis, si la première valeur était correcte, le programme doit demander une deuxième valeur de dé et la vérifier (avec l'affichage correspondant) ; enfin, si les deux valeurs étaient correctes, le programme doit afficher la somme des deux dés.
3. On décide finalement de ne pas afficher **Valeur correcte** si la valeur du dé testée est effectivement correcte (mais on garde l'affichage si la valeur est incorrecte). Quelle partie de votre code doit être modifiée ?

### 2.5.11 Factorielle qui dépasse

Écrire une fonction `dépasse` qui prend en argument un entier  $A$  et qui renvoie le plus petit entier  $n$  tel que  $n!$  soit supérieur ou égal à  $A$ , i.e. le plus petit entier  $n$  dont la factorielle atteint ou dépasse  $A$ . *Rappel :  $n! = 1 \times 2 \times 3 \times \dots \times n$*

### 2.5.12 Calculatrice

Dans cet exercice, nous allons nous interdire d'utiliser les opérateurs mathématiques de Python, ainsi que les fonctions du module `math`. À partir d'une seule fonction de départ, nous allons tenter de recréer tous les opérateurs arithmétiques nous mêmes.

1. Écrire une fonction `soustraction(a, b)` qui reçoit 2 entiers  $a$  et  $b$ , et renvoie leur soustraction ( $a-b$ ). Ici on peut utiliser l'opérateur `-`. C'est notre fonction de départ, à partir de laquelle nous allons recréer les autres. Cette fonction nous suffira pour définir toute l'arithmétique de base pour les entiers et plus encore !
2. En utilisant la fonction `soustraction`, définir la fonction `addition(a, b)`, qui calcule la somme de ses arguments. On rappelle qu'il ne faut utiliser aucun opérateur de Python (y compris l'opérateur `-`) mais uniquement la fonction `soustraction`.

3. En utilisant la fonction *addition*, définir la fonction *multiplication(a, b)* qui calcule le produit de ses arguments.
4. En utilisant les fonctions précédemment définies, écrire une fonction *division(a, b)* qui retourne le quotient de la division de a par b.
5. Bonus avec fonctions avancées : écrire à la place une fonction *division(a, b)* qui retourne deux valeurs : le quotient de la division entière de a par b ainsi que le reste de cette division.
6. En utilisant la fonction *multiplication* définie ci-dessus, définir les fonctions *puissance(a, n)* et *factorielle(a)*.

À présent, nous allons écrire un programme principal qui fonctionne comme une calculatrice. Le programme commence par lire un premier entier A, suivi d'un opérateur. Si (et seulement si) l'opérateur requiert un autre opérande, un autre entier B est lu. Si l'opérateur ne requiert pas d'autre opérande (exemple : factorielle), le résultat est affiché directement. Après l'affichage du résultat, le programme se relance et attend que l'utilisateur exécute une nouvelle opération. Les opérateurs qu'on veut supporter sont les suivants :

- "+" : addition
- "-" : soustraction
- "\*" : multiplication
- "/" : division
- "\*\*" : puissance
- "!" : factorielle

Exemples d'exécution :

```
A=5
+
B=8
13
```

```
A=9
-
B=12
-3
```

```
A=12
/
B=5
(2, 2)
```

```
A=13
/
B=5
(2, 3)
```

```
A=5
*
B=9
45
```

```
A=5
!
120
```



```
A=2
```

```
**
```

```
B=6
```

```
64
```

```
A=5
```

```
$
```

```
Opérateur inconnu, on recommence !
```

```
A=5
```

```
**
```

```
B=5
```

```
3125
```

## 2.6 TD6 : listes

### Basiques

#### 2.6.1 Exercice : code à trous

```
>> Multiple3 = [3, 6, 9, 15, 21]
>> Multiple3[2]

-----
>> Multiple3[____]
21
>> Multiple3.-----
>> Multiple3
[3, 6, 9, 15, 21, 24]

>> Multiple3 = [3, 6, 9, 15, 21, 24, 27]

>> Multiple3.insert( ____, 12)
>> Multiple3.insert( ____, 18)
>> Multiple3
[3, 6, 9, 12, 15, 18, 21, 24, 27]
>> Multiple3._____.([30,33])
>> Multiple3
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33]
>> Multiple3.-----
>> Multiple3
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36]

>> EhEh = ["tra", "la", "la", "la", "lère"])
>> EhEh.-----
>> EhEh
["tra", "la", "la", "la"]
>> EhEh.-----
>> EhEh
["tra", "la", "la"]
```

#### 2.6.2 Création de liste

Écrire un programme qui demande à l'utilisateur de taper des entiers (jusqu'à obtenir 0), ajoute les entiers positifs pairs à une liste appelée `liste_pairs`, les entiers positifs impairs à une autre liste appelée `liste_impairs`, et ne fait rien des entiers négatifs. Le programme affiche ensuite les 2 listes. Par exemple si l'utilisateur tape 1,-5,7,8,13,-4,0 alors le programme affiche : `pairs: [8] impairs: [1, 7, 13]` On remarque que les nombres négatifs et le 0 ne sont pas affichés.

#### 2.6.3 Affichage de liste

Écrire une fonction qui reçoit une liste et l'affiche sous les formats suivants :

- un élément par ligne
- sur une seule ligne, éléments séparés par des point-virgules
- sur plusieurs lignes, avec 10 éléments par ligne (éventuellement moins sur la dernière ligne) séparés par des espaces
- comment ne pas afficher de point-virgule après le dernier élément ?

## Réécriture de fonctions existantes

On se propose ici de réécrire un certain nombre de fonctions classiques de manipulation de listes, disponibles par défaut en Python mais pas dans d'autres langages comme le C par exemple. On s'interdira donc d'utiliser les fonctions Python prédéfinies. *Remarque* : avant d'avoir vu les boucles `for`, il est possible de faire tous ces exercices en utilisant des boucles `while`.

### 2.6.4 Test d'appartenance

Écrire une fonction qui reçoit une liste de nombres et un nombre, et qui teste si ce nombre appartient bien à la liste. Cette fonction renvoie donc un booléen. *Remarque* : Python permet de faire ce test avec le mot-clé `in`, mais on veut ici réécrire cette fonction pour comprendre comment la recherche fonctionne.

### 2.6.5 Indice d'apparition

Écrire une fonction qui reçoit une liste de nombres et un nombre, et qui renvoie l'indice de la première position de ce nombre dans cette liste, ou renvoie `-1` si le nombre n'y apparaît pas. *Remarque* : Python a une fonction `index()`, mais on se propose ici de la réécrire pour comprendre son principe.

### 2.6.6 Comptage d'élément

Écrire une fonction qui reçoit une liste de nombres et un nombre et qui compte combien de fois ce nombre apparaît dans la liste. La fonction renvoie ce compteur. *Remarque* : Python dispose d'une fonction `count` prédéfinie, mais on veut la réécrire.

### 2.6.7 Recherche de minimum ou maximum

1. Écrire une fonction qui reçoit une liste de nombres et qui renvoie son élément minimum.
2. Écrire une fonction qui reçoit une liste de nombres et qui renvoie l'indice de son plus grand élément.

*Remarque* : les fonctions `min` et `max` existent déjà en Python mais on se propose de les réécrire.

### 2.6.8 Calcul de somme et moyenne

1. Écrire une fonction qui reçoit une liste de nombres et renvoie la somme de ces nombres.
2. Écrire une fonction qui reçoit une liste de nombres et renvoie la moyenne de ces nombres.

*Remarque* : la fonction `sum` existe déjà dans Python. Le module `statistics` contient une fonction `mean` qui calcule la moyenne. Mais encore une fois on veut les réécrire.

## Manipulation de listes d'entiers

### 2.6.9 Diviseurs, nombres premiers

1. Écrire une fonction qui reçoit un entier positif et renvoie la liste de ses diviseurs.
2. Écrire une fonction qui reçoit un entier positif et renvoie un booléen qui indique si cet entier est premier. On utilisera la fonction précédente. On rappelle qu'un entier est premier si ses seuls diviseurs sont 1 et lui-même.

### 2.6.10 Jeu de dé et statistiques

1. Écrire une fonction `une_partie` qui reçoit un entier `n`, lance `n` fois un dé à 6 faces, stocke les `n` résultats dans une liste d'entiers, et la renvoie.
2. Écrire une fonction `stats_partie` qui reçoit une liste de `n` tirages, et affiche pour chaque face le pourcentage de tirages qui l'ont obtenue. (Par exemple : 1 - 17.2% ; 2 - 15.5% ; etc)
3. Écrire une fonction `face_gagnante` qui reçoit une liste de `n` tirages, et renvoie la face qui est apparue le plus souvent sur cette partie (la plus grande si égalité).
4. Écrire un programme principal qui demande à l'utilisateur le nombre de tirages par partie (`n`), le nombre de parties à jouer (`p`) ; qui joue `p` parties de `n` tirages ; qui affiche pour chaque partie la face gagnante ; qui affiche pour chaque face combien de parties elle a gagné.

## Manipulation de listes de caractères

Ces exercices utilisent les fonctions sur les caractères et chaînes de caractères.

### 2.6.11 Codage par décalage

1. Écrire une fonction `decal_list` qui reçoit une liste de caractères (représentant un mot) et un entier `n`, et qui renvoie la liste de caractères correspondant à ce mot codé par décalage de chaque lettre de `n` positions. On pourra utiliser la fonction de décalage d'un caractère codée précédemment (exo 2.4.6) ou la réécrire.
2. Écrire une fonction `list_to_string` qui reçoit une liste de caractères et renvoie la chaîne de caractères obtenue en les concaténant dans l'ordre. Par exemple cette fonction transforme la liste `['a', 'z', 'e', 'r', 't', 'y']` en la chaîne de caractères `"azerty"`. **Remarque** : la fonction prédéfinie `join` permet aussi de faire cette concaténation (cf cours 1.8.3), mais on veut la réécrire.
3. Écrire ensuite un programme principal qui demande à l'utilisateur un mot et un entier, code ce mot par décalage, et affiche le résultat. Par exemple le mot `"Bonjour"` décalé de 3 positions doit devenir `"Erqmrxu"`. Il faut donc récupérer la liste des caractères de ce mot (fonction `list`).
4. Modifier ce programme principal pour demander en boucle un mot à l'utilisateur (s'arrête en lisant le mot `"stop"`), puis un entier, et afficher le codage de ce mot par décalage de cet entier.

### 2.6.12 Comptage alphabétique (force brute vs subtile)

1. Écrire une fonction qui reçoit une liste de caractères et une lettre, compte combien de fois cette lettre apparaît dans la liste (que ce soit en minuscule ou en majuscule), et renvoie le résultat.
2. Écrire une fonction qui reçoit une liste de caractères, compte toutes les lettres de l'alphabet avec la fonction précédente, et renvoie la liste des 26 compteurs.
3. Question : combien de fois a-t-on parcouru la liste de caractères ?
4. Modifier ce programme afin de compter toutes les lettres de l'alphabet en un seul parcours de la liste de caractères.
5. Bonus avec des dictionnaires : modifier ce programme pour compter tous les caractères qui apparaissent dans un texte ; il peut y avoir des caractères non alphabétiques, et surtout on ne sait pas à l'avance quels caractères apparaissent ou pas.

### 2.6.13 Liste de mots

1. Écrire une fonction *taille(liste)* qui prend en argument une liste des chaînes de caractères et renvoie la liste des tailles de chaque élément de la liste.
2. Écrire une fonction *lire(n)* qui reçoit en paramètre un entier n, lit au clavier n chaînes de caractères, les stocke dans une liste, et renvoie cette liste. Un exemple d'exécution avec n=5 :

```
Tapez un mot : train
Tapez un mot : cheval
Tapez un mot : voiture
Tapez un mot : avion
Tapez un mot : accordéon
['train','cheval','voiture','avion','accordéon']
```

3. Écrire une fonction *affiche(liste)* qui reçoit en paramètre une liste de chaînes, calcule la liste de leurs tailles, affiche chaque chaîne et sa taille, puis la moyenne des tailles. Par exemple avec la liste précédente :

```
Taille du mot train : 5
Taille du mot cheval: 6
Taille du mot voiture: 7
Taille du mot avion: 5
Taille du mot accordéon: 9
Taille moyenne: 6.4
```

4. Compléter pour afficher aussi les mots plus long que la taille moyenne.
5. Écrire une fonction *nbocc(mot,carac)* qui compte et renvoie le nombre d'occurrences (le nombre d'apparitions) d'un caractère donné dans une chaîne donnée.
6. Écrire une fonction *compteCarac(liste,car)* qui reçoit une liste de chaînes et un caractère, affiche les mots contenant ce caractère, et affiche le nombre total d'occurrences de ce caractère dans tous les mots. Si ce caractère n'est présent dans aucun mot, le programme affiche un message d'erreur à la place. Deux exemples d'exécution (avec la liste précédente de 5 mots et la lettre 'o' puis 'w') :

```
Mots contenant le caractère o :
voiture
avion
accordéon
le caractère o apparait 4 fois.
```

```
Erreur la lettre w n'est présente dans aucun des mots
```

7. Utiliser cette fonction pour trouver le mot d'une liste (reçue en paramètre) où un caractère donné (en paramètre) apparaît le plus de fois. En cas d'égalité, favoriser le mot le plus court contenant autant de fois ce caractère. Renvoyer ce mot. **Version avancée** : renvoyer ce mot **et** le nombre d'occurrences du caractère dedans.
8. Écrire un programme principal qui utilise les fonctions ci-dessus et : lit au clavier le nombre de mots de la liste, puis les mots ; les affiche avec leur taille ; puis lit un caractère et le compte dans les mots ; propose à l'utilisateur de rejouer (même liste, nouveau caractère) jusqu'à ce qu'il refuse ; se termine alors.

## Listes avancées : for, tri, etc

Ces exercices sont réalisables avec une boucle **while**, mais seront plus faciles avec la boucle **for** (voir chapitre 1.10).

### 2.6.14 Insertion par recherche dichotomique

Étant donnée une liste d'entiers triée dans l'ordre croissant, on souhaite insérer un nouvel élément à la bonne position pour garder la liste triée.

1. Ecrire une fonction *insert\_intuitif(liste\_triee, e)* qui prend en argument une liste d'entiers supposée déjà triée et un entier, et qui insère le nouvel élément par la méthode qui vous semble la plus intuitive. Attention la fonction ne renvoie rien mais modifie la liste qui lui est passée en argument.
2. Si on note  $n$  la longueur de la liste, quel est le nombre maximum d'itérations utilisées dans *insert\_intuitif* pour réaliser l'insertion ?

On voudrait utiliser une méthode dichotomique pour insérer un entier  $e$  dans une liste triée. L'idée est de comparer  $e$  avec un élément de la liste qu'on appelle le pivot, idéalement situé au milieu. Si  $e$  est plus grand que le pivot alors il doit être inséré après, sinon il doit être inséré avant. On recommence ensuite la même opération avec la moitié de liste concernée.

Illustrons cela par un exemple. Dans la liste  $[2, 12, 17, 25, 33, 35, 44, 54, 77, 91]$  on souhaite insérer 49. Pour chaque itération on délimite par  $d$  et  $f$  la portion de la liste où l'on sait que 49 doit être inséré.  $p$  représente le pivot.

2	12	17	25	33	35	44	54	77	91
d				p					f

2	12	17	25	33	35	44	54	77	91
				d			p		f

2	12	17	25	33	35	44	54	77	91
				d	p		f		

2	12	17	25	33	35	44	54	77	91
					d	p	f		

2	12	17	25	33	35	44	54	77	91
						d	f		

On voit que 49 doit être inséré entre 44 et 54.

3. Écrire une fonction *insert\_dicho(liste\_triee, e)* qui implémente cette méthode.
4. Supposons que la longueur de la liste est une puissance de 2, notée  $n = 2^k$ . Quel est le nombre d'itérations utilisées dans *insert\_dicho* pour réaliser l'insertion ? Quelle est la méthode la plus rapide ?

### 2.6.15 Jouons avec des listes triées

1. Soit la fonction **InsererDans** ayant deux arguments **e1** et *liste\_triee*, une liste de nombres classée en ordre croissant. La fonction ne doit pas la modifier ! La fonction **InsererDans(e1, liste\_triee)** retourne une liste dans laquelle l'élément **e1** est placé à la bonne position dans *liste\_triee*.

```
> print(InsererDans(1, []))
[1]
> print(InsererDans(4, [1,3,5]))
[1,3,4,5]
```

**Indice** : il faut d'abord cloner la liste *liste\_triee* dans une variable locale, la parcourir pour identifier la position où l'élément doit être inséré, et utiliser la fonction *insert*. Attention au cas où la liste triée est vide.

2. Soit la fonction `FusionListesT` qui reçoit pour arguments deux listes ordonnées, et qui renvoie une liste ordonnées correspondant à la fusion des deux listes, sans perdre aucun élément (il peut donc y avoir des doublons). La taille de la liste retournée doit donc être égale à la somme des tailles des deux listes fournies en argument. On utilisera la fonction `InsererDans` définie ci-dessus.

```
> print(FusionListesT([], []))
[]
> print(FusionListesT([1,3,4], [2,2,7,9]))
[1,2,2,3,4,7,9]
```

3. Soit la fonction `SupprimDoublons` qui reçoit en argument une liste ordonnée, et retourne une nouvelle liste ordonnée dans laquelle les doublons ont été éliminés.

```
> print(SupprimDoublons([1,2,3]))
[1,2,3]
> print(SupprimDoublons([1,1,1]))
[1]
```

### 2.6.16 Cartes bancaires et formule de Luhn

Pour protéger les numéros de cartes bancaires contre les erreurs aléatoires (faute de frappe par exemple), ceux-ci sont générés de façon à respecter la formule de Luhn. Il s'agit d'une somme de contrôle, c'est-à-dire une série d'opérations arithmétiques, dont le résultat doit être un multiple de 10 pour que le numéro soit valide. On modélise un numéro par la liste de ses chiffres de gauche à droite. Voici la description de l'algorithme de Luhn faite par wikipedia ([https://fr.wikipedia.org/wiki/Formule\\_de\\_Luhn](https://fr.wikipedia.org/wiki/Formule_de_Luhn)), en trois étapes :

- L'algorithme multiplie par deux un chiffre sur deux, en commençant par l'avant dernier et en se déplaçant de droite à gauche. Si un chiffre qui est multiplié par deux est plus grand que neuf (comme c'est le cas par exemple pour 8 qui devient 16), alors il faut le ramener à un chiffre entre 1 et 9. Pour cela, il y a 2 manières de faire (pour un résultat identique) :
    - Soit les chiffres composant le doublement sont additionnés (pour le chiffre 8 : on obtient d'abord 16 en le multipliant par 2 puis 7 en sommant les chiffres composant le résultat :  $1+6$ ).
    - Soit on lui soustrait 9 (pour le chiffre 8 : on obtient 16 en le multipliant par 2 puis 7 en soustrayant 9 au résultat).
  - La somme de tous les chiffres obtenus est effectuée.
  - Le résultat est divisé par 10. Si le reste de la division est égal à zéro, alors le nombre original est valide.
1. Commencer par écrire une fonction `changeUnChiffre` qui reçoit un chiffre et le modifie selon l'étape 1 de l'algorithme.
  2. Puis écrire une fonction `changeChiffres` qui reçoit une liste de chiffres, les parcourt et les modifie selon l'étape 1 de l'algorithme (on rappelle qu'on ne change qu'un chiffre sur 2).
  3. Enfin écrire une fonction `resteDivSomme` qui reçoit une liste de chiffres (résultats de l'étape 1), calcule leur somme (étape 2), la divise par 10 et renvoie le reste de cette division (étape 3).
  4. On peut maintenant écrire une fonction `verifie_Luhn(numero)` qui prend en argument un numéro représenté par la liste de ses chiffres, et qui renvoie un booléen indiquant s'il vérifie la formule de Luhn. On utilisera les fonctions précédentes.
  5. Pour générer un numéro valide, il suffit d'ajouter à la fin (à droite) d'un numéro quelconque un chiffre de contrôle. Pour déterminer ce chiffre, on applique la formule de Luhn sur le numéro auquel on a ajouté un chiffre de contrôle égal à zéro. Si le résultat  $S$  est un multiple de 10 il n'y a rien à changer. Sinon, on remplace le chiffre de contrôle par  $10 - (S \% 10)$ . Écrire une fonction `ajoute_chiffre_controle(numero)` qui prends en argument une liste de chiffres et qui lui rajoute un chiffre de contrôle pour la rendre valide.
  6. Écrire une fonction `genere_numero(n)` qui prend un entier positif en argument et qui renvoie un numéro de  $n$  décimales valide généré aléatoirement. On utilisera la fonction `randint` du module `random`, et la fonction définie ci-dessus.

## Listes de listes, ou listes "à 2 dimensions"

### 2.6.17 Jeu de démineur (listes de listes)

Dans cet exercice on veut coder un jeu de démineur (avec un affichage uniquement textuel). Même si cet exercice est d'abord abordé en TD, il vous est conseillé de le tester ensuite sur machine. La grille de démineur (à N lignes et M colonnes) sera représentée par une liste de listes de valeurs. Chaque liste de M éléments contient toutes les valeurs d'une ligne donnée de la grille. La liste de N éléments contient donc les N lignes de la grille. Il s'agit ici de la grille solution, c'est-à-dire la grille contenant la position des mines. Il est par ailleurs nécessaire de stocker aussi une autre grille, indiquant quelles cases ont déjà été dévoilées par le joueur. On choisit pour cela d'utiliser une liste de listes de booléens. La valeur True dans une case signifie que le joueur a déjà dévoilé cette case, alors que la valeur False signifie que cette case n'a pas encore été dévoilée.

1. Écrire une fonction `creerGrille` qui : reçoit en paramètres les entiers N et M, et un paramètre optionnel v (par défaut 0) représentant la valeur d'initialisation de toutes les cellules ; crée une grille de démineur à N lignes et M colonnes (donc une liste de N listes de M entiers), ne contenant que des valeurs v (0 par défaut, ou autre si spécifiée) ; et renvoie cette liste. *Par exemple pour N=5 et M=7 on obtient la liste suivante :*

```
[[0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
```

2. Écrire une fonction `placerMines` qui : reçoit en paramètre une grille de démineur et un entier X ; modifie cette liste pour y placer X mines (valeur 1) à des positions au hasard ; ne renvoie rien. *Par exemple si on demande de placer 7 mines dans la liste précédente, on obtient :*

```
[[0, 0, 1, 0, 0, 0, 0], [0, 0, 1, 1, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1],
 [1, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 0]]
```

3. Écrire une fonction `afficheSolution` qui : reçoit en paramètre une liste `positionsMines` (grille d'entiers indiquant les positions des mines), et affiche cette grille sous la forme d'un rectangle de caractères représentant la solution du démineur, c'est-à-dire dévoilant les positions des mines. On choisit d'afficher avec un '-' (tiret) les cases non minées (valeur 0), et avec une '\*' (étoile) les cases minées (valeur 1). *Par exemple la liste précédente sera affichée sous la forme suivante (à gauche avant placement des mines, à droite après) :*

```
-----      --*----
-----      --***--
-----      -----*
-----      *-----*
-----      -----
```

4. Écrire une fonction `testMine` qui : reçoit la grille des positions des mines, et 2 coordonnées i (numéro de ligne entre 0 et N-1) et j (numéro de colonne entre 0 et M-1) supposées correctes (on n'a pas besoin de les tester) ; vérifie s'il y a une mine sur la case indiquée par ces coordonnées ; et renvoie un booléen indiquant le résultat. *Par exemple :*

```
testMine(0,2) renvoie True
testMine(1,1) renvoie False
```

5. Écrire une fonction `compteMinesVoisines` qui : reçoit la grille des positions des mines, et 2 coordonnées i et j supposées correctes ; compte le nombre de mines sur les cases voisines (attention aux effets de bord, certaines cases ont moins de voisines que d'autres ! 3 voisines dans les coins, 5 voisines sur les bords, 8 voisines au centre) ; renvoie ce compteur. On pourra utiliser une fonction auxiliaire qui calcule et renvoie la liste des cellules voisines d'une cellule donnée.



6. Écrire une fonction `afficheJeu` qui : reçoit en paramètre une liste `positionsMines` (grille d'entiers indiquant les positions des mines), et une liste `casesDevoilees` (grille de booléens indiquant les cases dévoilées) ; affiche la grille de jeu sous la forme d'un rectangle de caractères représentant la grille telle que le joueur la voit pendant la partie (il ne voit pas les positions des mines). On choisit d'afficher avec un '?' (point d'interrogation) les cases non encore découvertes ; avec un '\*' une case découverte minée (se produit quand le joueur perd) ; sur les cases découvertes non minées, on affichera un entier indiquant le nombre de mines sur les cellules voisines (compté avec la fonction précédente). *Par exemple affichage avec seulement 2 cases découvertes dans les coins :*

0??????  
 ???????  
 ???????  
 ???????  
 ???????1

7. Écrire une fonction `getCoords` qui : reçoit en paramètre la grille indiquant les cases déjà dévoilées, et les dimensions N et M ; qui demande à l'utilisateur des coordonnées i et j et les filtre jusqu'à ce qu'elles soient correctes (comprises dans les bornes autorisées et correspondant à une case non encore dévoilée) ; puis qui renvoie ces coordonnées une fois correctes. **Attention** : on ne redemande que la coordonnée incorrecte s'il n'y en a qu'une. *Exemples d'interactions* :

```
A toi de jouer !
Ligne? 10
Ligne < 5 svp ? 18
Ligne < 5 svp ? 4
Colonne? 10
Colonne < 7 svp ? 7
Colonne < 7 svp ? 6

A toi de jouer !
Ligne? 3
Colonne? 4
Case deja devoilee, recommence
Ligne? 3
Colonne? 5
```

8. Écrire un programme principal qui :

- initialise une grille
- demande à l'utilisateur le nombre X de mines à placer et le filtre, puis place les X mines
- initialise la grille de booléens indiquant les cases dévoilées (pour l'instant aucune)
- affiche la grille de jeu
- demande à l'utilisateur un coup (filtre jusqu'à avoir des coordonnées correctes) et dévoile la case correspondante dans la grille de booléens
- vérifie s'il a perdu (touché une mine) : dans ce cas s'arrête et affiche la grille de jeu (y compris la mine touchée), puis la solution. *Par exemple :*

```
Perdu, touche une mine !
0??????
01*????
???????
???????
???????
???????
```

```

La solution etait :
-----
--*--***
-----
---**---
-----*

```

- sinon affiche la grille de jeu, en remplaçant donc le ? de la nouvelle cellule dévoilée par le nombre de mines sur ses cases voisines
- recommence avec un nouveau coup
- s'arrête dès que le joueur perd (en touchant une mine), ou une fois qu'il a dévoilé toutes les cases sauf les mines (dans ce cas il a gagné). *Par exemple :*

```

Coup numero 28
A toi de jouer !
Ligne? 4
Colonne? 6
1?22110
23?2?10
?224320
221??10
?112210
Tu as gagne en 28 coups, bravo !

```

- **Attention :** ce programme principal doit utiliser les fonctions déjà codées ci-dessus.

9. **Bonus :** quand le joueur choisit une case à découvrir qui n'est entourée d'aucune mine, découvrir automatiquement récursivement toutes les cases voisines qui n'ont pas de mines (c-à-d qu'on peut dévoiler automatiquement toutes les voisines d'une case dont on sait qu'elle a 0 mines sur ses voisines, et ainsi de suite pour les autres cases découvertes ainsi qui ont aussi 0 mines voisines).

## 2.7 TD7 : itération inconditionnelle for

### Quelques exercices basiques pour commencer...

#### 2.7.1 Exercices de lecture

Pour les instructions suivantes, indiquer ce qui sera affiché.

```
for i in range(4) :
    print(i)

for j in range(2,5):
    print(j)

for k in range(3,12,3):
    print(k)

for l in range(12, 3):
    print(l)

for m in range(12,3,-2):
    print(m)
```

Expliquer ce qu'affiche le programme suivant.

```
# la liste en arg. doit ne contenir que des nb
def somme_des_positifs(liste) :
    s=0
    for e in liste:
        if e>0:
            s=s+e
    return s

# prog. principal
ma_liste = [2,-4,6,0,-5,1]
for e in ma_liste:
    print(e+1)
t=somme_des_positifs(ma_liste)
print(t)
```

#### 2.7.2 Les bases des boucles for

1. Écrire une boucle for qui affiche les entiers de 1 à 10 (un par ligne)
2. Écrire une boucle for qui affiche les entiers de 1 à 10 en ordre inverse
3. Idem mais en les affichant sur une seule ligne, séparés par une virgule, ligne terminée par un point.
4. Écrire une fonction qui reçoit 3 entiers (deux bornes et un pas) et affiche les entiers entre ces deux bornes incluses, séparés par ce pas. (Par exemple pour les bornes 1 et 20, et un pas de 3, il doit afficher les entiers : 1, 4, 7, 10, 13, 16, 19.)
5. Écrire un programme principal qui demande à l'utilisateur des bornes **binf** et **bsup** et un pas **step**, et appelle la fonction précédente pour afficher la liste des entiers entre ces bornes séparés par ce pas.
6. Écrire une boucle for qui affiche les multiples de 3 et les multiples de 5 compris entre 1 et 50.
7. Idem mais en n'affichant pas les multiples de 15. Par exemple on veut afficher : 3,5,6,9,10,12,18,20,21,etc

### 2.7.3 Sommes d'entiers

Écrire des fonctions qui reçoivent un entier  $n$  et qui :

1. f1 renvoie la somme des  $n$  premiers entiers
2. f2 renvoie la somme des  $n$  premiers entiers impairs
3. f2 renvoie la somme des  $n$  premiers carrés

Écrire un programme principal qui lit un entier au clavier, appelle ces fonctions l'une après l'autre, et affiche leurs résultats (avec un texte clair à chaque fois).

### 2.7.4 Suite croissante

- Écrire une fonction qui reçoit une liste d'entiers et qui renvoie un booléen indiquant si les éléments de cette liste sont dans l'ordre croissant (non strict), c'est-à-dire si pour tout indice  $i$ , l'élément en position  $i$  est inférieur ou égal à l'élément en position  $i+1$ .
- On suppose qu'on a une fonction similaire qui détermine si une liste reçue en paramètre est en ordre décroissant. Écrire une fonction qui détermine si une liste reçue en paramètre est dans le désordre (ni en ordre croissant, ni en ordre décroissant).

### 2.7.5 Comptons les moutons

1. Écrire un programme qui demande à l'utilisateur un nombre de moutons, et qui compte les moutons "à haute voix", par exemple :

```
Combien de moutons ? 3
1 moutons
2 moutons
3 moutons
```

2. Que se passe-t-il si le nombre de moutons rentré par l'utilisateur est 1 ? Est nul ? Est négatif ?
3. Modifier le programme précédent pour qu'il affiche un seul mouton au singulier ("1 mouton" plutôt que "1 moutons"). Attention : on évitera les comparaisons inutiles.

### 2.7.6 Règle graduée

Dans cet exercice on veut utiliser des boucles for, on s'interdit d'utiliser la multiplication de chaînes (par exemple  $n \times '-'$  donne directement une chaîne de  $n$  caractères '-').

1. Écrire un programme qui demande à l'utilisateur un entier  $n$ , puis affiche avec des '-' une règle de longueur  $n$ . Par exemple :

```
Longueur ? 13
-----
```

2. Modifier ce programme pour qu'il affiche une règle graduée, avec un '|' au début, et un '|' à la place de chaque  $x$ -ième '-', où l'intervalle  $x$  est aussi demandé à l'utilisateur. Par exemple :

```
Longueur ? 27
Intervalle ? 10
|-----|-----|-----
```

## Les choses se compliquent un peu...

### 2.7.7 Accumulateurs

1. Écrire une fonction qui utilise une boucle for pour calculer et renvoyer la factorielle d'un entier n reçu en paramètre.
2. Écrire une fonction qui reçoit en paramètre deux entiers n et a, utilise une boucle for pour calculer n à la puissance a (sans utiliser l'opérateur de puissance \*\*), et renvoie le résultat.
3. Que se passe-t-il si l'utilisateur saisit un a négatif? Le résultat est-il correct?
4. Modifier cette fonction pour qu'elle soit capable de calculer les puissances négatives (rappel :  $a^{-1} = 1/a$ )

### 2.7.8 Pile ou face, et statistiques

1. Écrire une fonction qui reçoit un entier T, simule une partie de T tirages Pile ou Face, et renvoie le nombre de Pile obtenus.
2. Écrire une fonction qui reçoit un entier N et un entier T, qui simule N parties de T tirages Pile ou Face, qui construit la liste de N entiers contenant le nombre de Pile obtenus pour chaque partie, et qui renvoie cette liste.
3. A partir d'une liste d'éléments correspondant au nombre de Pile obtenus dans plusieurs parties de Pile ou Face, et d'un entier T représentant le nombre de tirages par partie, on veut construire et afficher un histogramme du nombre de parties ayant obtenu chaque nombre de Pile possible (entre 0 et T inclus). On va procéder en plusieurs étapes :
  - (a) Écrire une fonction qui reçoit la liste de N nombres de tirages Pile par partie (pour les N parties), et qui compte le nombre d'occurrences d'une valeur x. Cela représente le nombre de parties qui se sont conclues par x tirages Pile.
  - (b) Utiliser cette fonction pour construire une liste de compteurs correspondant au nombre de parties qui se sont conclues par chaque nombre de tirages Pile possible (entre 0 si on a fait Face à chaque fois, et T si on a fait Pile à chaque fois). La fonction renvoie la liste de nombre de tirages Pile.
  - (c) Combien de fois parcourt-on la liste des résultats des N parties?
  - (d) Réécrire cette fonction pour compter le nombre de parties conclues par chaque nombre de tirages Pile (liste à T+1 éléments) en un seul parcours de la liste de N parties.
  - (e) Affichage histogramme : écrire une fonction qui reçoit une liste de T+1 entiers et l'affiche sous forme d'un histogramme horizontal à T+1 lignes, sur chacune desquelles apparaît le numéro i de la ligne (de 0 à T inclus) et un nombre d'étoiles proportionnel à la valeur associée (qui correspond au nombre de parties ayant obtenu exactement i Pile).
4. Écrire un programme principal qui demande à l'utilisateur un nombre de parties N et un nombre de tirages par partie T, qui simule les tirages et affiche l'histogramme résultant, en appelant les fonctions précédentes.

## Manipulation de chaînes de caractères

### 2.7.9 Modification de chaînes

1. Écrire une fonction qui supprime tous les espaces d'une chaîne de caractères reçue en paramètre. Elle renvoie le résultat.
2. Écrire une fonction qui reçoit une chaîne de caractères, et renvoie son inverse. Par exemple :

```
>>> inverse_chaine("inf101") == "101fni"
True
```

3. Écrire une fonction qui reçoit une chaîne de caractères, mélange ses lettres, et renvoie le résultat (sous forme d'une chaîne, et pas d'une liste).

### 2.7.10 Palindromes

1. Écrire une fonction qui reçoit un mot, qui détermine si ce mot est un palindrome, et qui renvoie le résultat (un booléen). Un palindrome est un mot qui se lit de la même manière de la gauche vers la droite ou de la droite vers la gauche, comme "kayak" ou "elle".
2. Écrire une fonction qui reçoit une phrase, détermine si la phrase est un vrai palindrome (dans un vrai palindrome, les espaces sont répartis de manière symétrique dans la phrase, par exemple : "rats live on no evil star") ; un pseudo-palindrome (dans un pseudo palindrome, il y a une symétrie entre les lettres mais pas entre les espaces, par exemple "eve reve") ; ou ni l'un ni l'autre. Cette fonction renvoie le résultat sous la forme d'un entier (par exemple vrai palindrome = 2, pseudo-palindrome = 1, pas un palindrome = 0).
3. Écrire un programme principal qui boucle jusqu'à ce que l'utilisateur saisisse "stop". Pour chaque chaîne de caractère saisie :
  - Si c'est le mot "stop", le programme s'arrête en affichant "fin"
  - S'il s'agit d'un mot (ne contenant pas d'espaces), il appelle la première fonction pour déterminer si c'est un palindrome, et affiche le résultat
  - S'il s'agit d'une phrase (contient des espaces), il appelle la deuxième fonction pour déterminer son type, et affiche un message correspondant.
  - Demande ensuite une nouvelle chaîne de caractères et recommence.

### 2.7.11 Zip

1. Écrire une fonction qui reçoit 2 chaînes et renvoie leur "fermeture éclair", c'est-à-dire la liste de toutes les chaînes obtenues en combinant un caractère de la première chaîne suivi d'un caractère de la deuxième chaîne. Par exemple :

```
>>> zip_chaine("abc", "d") == ['ad', 'bd', 'cd']
True
>>> zip_chaine("a", "bcd") == ['ab', 'ac', 'ad']
True
>>> zip_chaine("ab", "cd") == ['ac', 'ad', 'bc', 'bd']
True
```

2. Définir une fonction qui réalise la fermeture éclair de 2 listes lst1 et lst2 reçues en paramètres, c'est-à-dire la liste de toutes les paires composées d'un élément de lst1 et d'un élément de lst2. Par exemple :

```
>>> zip_liste([1], [2,3,4]) == [(1, 2), (1, 3), (1, 4)]
True
>>> zip_liste([1,2,3], [4]) == [(1, 4), (2, 4), (3, 4)]
True
>>> zip_liste([1,2], [3,4]) == [(1, 3), (1, 4), (2, 3), (2, 4)]
True
```

## Manipulation de listes

Vous pouvez aussi refaire les exercices Listes Avancées du TD précédent en utilisant une boucle for au lieu d'une boucle while.

### 2.7.12 Copie et inversion de listes

1. Écrire une fonction qui copie la liste reçue en paramètre et renvoie la nouvelle liste. On demande donc en fait de ré-implémenter la fonction `list` qui existe déjà en Python.
2. Écrire une fonction qui crée une copie en ordre inverse de la liste reçue en paramètre, et renvoie cette copie inversée.

### 2.7.13 Tri par sélection

Écrire une fonction qui reçoit une liste en paramètre, la modifie pour la trier, puis renvoie la liste triée. Cette fonction a donc à la fois une valeur de retour (la liste triée) et des effets de bord (elle modifie la liste reçue en paramètre). On utilisera l'algorithme du tri par sélection qui consiste à répéter en boucle les étapes suivantes :

- Chercher le minimum de la partie droite de la liste (partie non triée).
- Échanger le minimum trouvé avec l'élément à l'indice *i* (indice de début de la partie non triée).
- Décaler l'indice *i*, donc la limite entre partie triée et partie non triée, vers la droite.

L'algorithme commence avec une limite au début de la liste (liste triée vide, liste non triée = liste entière). A chaque étape, la longueur de la partie triée augmente et la longueur de la partie non triée diminue. L'algorithme s'arrête quand la limite entre parties triée et non triée atteint la fin de la liste, et donc que la liste est entièrement triée. Par exemple :

```
>>> tri_selection([1,2,3]) == [1,2,3]
True
>>> tri_selection([3,2,1]) == [1,2,3]
True
>>> tri_selection([3,1,2]) == [2,1,3]
False
```

## Des maths !

### 2.7.14 Diviseurs propres et nombres parfaits

1. Écrire une fonction qui reçoit un entier naturel *n* et renvoie la liste de ses diviseurs.
2. Écrire une fonction qui reçoit un entier naturel *n*, et calcule et renvoie la somme de ses diviseurs propres (ses diviseurs autres que lui-même). Par exemple :

```
>>> sommeDivPropre(0)
0
>>> sommeDivPropre(1)
0
>>> sommeDivPropre(4)
3
>>> sommeDivPropre(6)
6
```

3. Un entier naturel est dit parfait s'il est égal à la somme de tous ses diviseurs propres (par exemple 6). Écrire une fonction qui teste si un entier naturel reçu en paramètre est parfait (en utilisant la fonction précédente). Par exemple :

```
>>> estParfait(6)
      True
>>> estParfait(10)
      False
```

4. Écrire une fonction qui reçoit deux entiers `binf` et `bsup` (optionnels, valeurs par défaut respectivement 2 et 100), et affiche tous les nombres parfaits de l'intervalle `[binf,bsup]`, sur une seule ligne, séparés par un espace. Par exemple :

```
>> parfaits_entre()
Nombres parfaits de [2,100]
6 28
>> parfaits_entre(7)
Nombres parfaits de [7,100]
28
```

### 2.7.15 Nombre de Armstrong

Un nombre est dit de Armstrong s'il est égal à la somme des cubes de ses chiffres.

1. Écrire une fonction qui décompose un entier en la liste de ses chiffres, et renvoie cette liste.
2. Écrire une fonction `est_Armstrong(nombre)` qui prend en argument un nombre et renvoie `True` si le nombre est de Armstrong, `False` sinon.
3. Écrire une fonction qui prend en argument une borne `max` et renvoie la liste des nombres de Armstrong inférieurs à cette borne.
4. Écrire un programme qui génère la liste des nombres de Armstrong inférieurs à 1000.

### 2.7.16 Approximation de la racine carrée

Étant donné un réel positif  $a$ , on définit la suite réelle  $x_n$  de la manière suivante :

$$\begin{cases} x_0 = a \\ x_{i+1} = \frac{x_i^2 + a}{2x_i} \end{cases} \quad \text{pour } i > 0$$

Cette suite converge vers  $\sqrt{a}$ .

1. Écrire une fonction `maRacine` qui reçoit un réel  $a$  et un entier  $n$ , qui calcule l'approximation de la racine de  $a$  par le  $n$ -ième terme de la suite ci-dessus, et renvoie cette valeur.
2. Écrire un programme principal qui demande à l'utilisateur un réel  $a$  et un entier  $n$ , calcule l'approximation de la racine avec la fonction ci-dessus, l'affiche, et affiche son carré pour vérifier si c'est une bonne approximation.
3. Écrire une fonction `precisionRacine` qui a partir d'un réel  $a$  et d'un entier  $n$  reçus en paramètres, calcule la valeur absolue de la différence entre l'approximation (obtenue par la fonction de la première question) et la racine de  $a$  (obtenue par la fonction `sqrt` du module `math`, ou avec l'opérateur de puissance `**`).
4. Écrire une fonction `maRacinePrec` qui reçoit un réel  $a$  et un réel  $p$  représentant la différence maximale souhaitée entre l'approximation de la racine et sa valeur réelle. Cette fonction calcule l'approximation de la racine de  $a$  et sa précision, avec un  $n$  de plus en plus grand, jusqu'à obtenir la précision voulue. Elle renvoie alors la valeur de  $n$  nécessaire pour obtenir cette "bonne" approximation.



### 2.7.17 Approximation de e

Un étudiant e1 place 1 kopec dans une banque, qui le rémunère au taux de 100% à la fin d'une année (taux 1 tous les 1 ans). L'étudiant se retrouvera donc en possession de 2 kopecs au bout d'un an. Un deuxième étudiant e2 choisit de placer son kopec dans une banque lui offrant un taux de rémunération de 50% tous les six mois (taux 0.5 tous les 0.5 ans). Ce dernier se retrouvera donc en possession de 2,25 kopecs à la fin de l'année ( $1.5 * 1.5 * 1$ ).

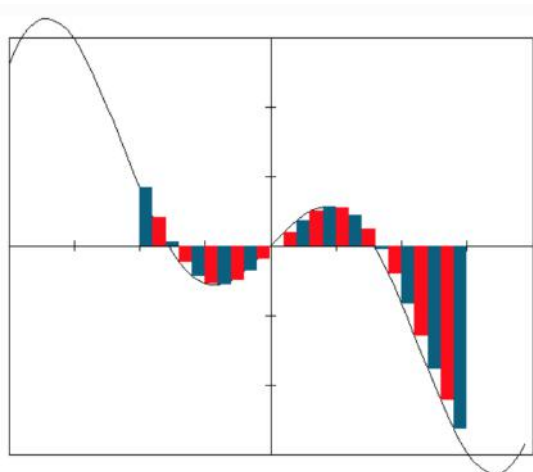
1. Écrire une fonction qui calcule ce que l'étudiant  $e_n$ , qui a placé son kopec dans une banque lui offrant un taux de rémunération de  $1/n$  toutes les  $1/n$  années, possède à la fin de l'année.
2. Cette fonction calcule en fait des valeurs approchées du nombre e (appelé nombre exponentiel, nombre d'Euler, ou constante de Neper), qui vaut environ 2,718281828459045.
3. Écrire un programme principal qui demande à l'utilisateur un entier nmax, et affiche successivement toutes les approximations (calculées avec la fonction ci-dessus) pour toutes les valeurs de n entre 1 et nmax.

```
nmax?10
Approx avec 1 = 2.0
Approx avec 2 = 2.25
Approx avec 3 = 2.3703703703703702
Approx avec 4 = 2.44140625
Approx avec 5 = 2.48832
Approx avec 6 = 2.5216263717421135
Approx avec 7 = 2.546499697040712
Approx avec 8 = 2.565784513950348
Approx avec 9 = 2.5811747917131984
Approx avec 10 = 2.5937424601000023
```

4. Écrire une fonction qui affiche toutes les estimations successives jusqu'à une précision de 0.01 (c-à-d que la distance entre e et son approximation doit être inférieure ou égale à 0.01).

### 2.7.18 Approximation de l'intégrale d'une fonction

On veut calculer l'approximation de l'intégrale<sup>1</sup> de la fonction cosinus sur un intervalle donné, par la méthode des rectangles, pour un nombre n de rectangles reçu en paramètre. Par exemple pour la fonction cosinus sur l'intervalle  $[-2,3]$ , on obtient le graphique ci-dessous. La formule à utiliser pour le calcul est la suivante, où n est la largeur des rectangles, donc plus n est petit, plus le calcul est précis.



h = 0.2000, aire du petit rectangle : -0.52764  
Intégrale de f entre -2.0000 et 3.0000 : -1.58612

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \sum_{i=1}^n f\left(a + i \frac{b-a}{n}\right)$$

1. le package `scipy.integrate` fournit des fonctions de calcul d'intégrales, cf <https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html>

## 2.8 TD8 : fonctions avancées

### Exercices pour débiter

#### 2.8.1 Affichage d'une liste

Écrire une fonction qui reçoit une liste, et un booléen optionnel. Cette fonction affiche la liste à l'endroit si le booléen est vrai (par défaut), à l'envers sinon.

#### 2.8.2 Sauvons la planète...

Écrire une fonction qui reçoit en paramètre un nombre de kilomètres à parcourir, un nombre de passagers (par défaut 1), un booléen indiquant s'il y a des embouteillages (par défaut non), et qui renvoie le meilleur moyen de transport (voiture, vélo, ou tram) selon la situation. On considère que la voiture n'est pas recommandée à moins de 2 passagers, ni pour moins de 10km, ni quand il y a des embouteillages. Le vélo est recommandé jusqu'à 10 km (inclus). On pourra rajouter d'autres paramètres optionnels : niveau de forme, météo, etc.

#### 2.8.3 Le menu

Un restaurant possède trois menus : le **Basique** à 9€, le **Gourmand** à 15€ et le **Complet** à 19€. Pour chacun des menus, le client peut choisir de rajouter une boisson à 4€. De plus, le client peut choisir un supplément fromage et/ou un supplément café, chaque supplément coûtant 1,50€.

1. Écrire une fonction `prix_menu` qui prend comme argument le nom du menu, puis deux arguments optionnels : un booléen `avecBoisson` dont la valeur par défaut est `False`, et un entier `nb_supplement` qui doit valoir 0 par défaut. La fonction doit renvoyer le prix du menu correspondant.
2. Écrire un programme principal qui calcule puis affiche le prix total de l'addition pour la table suivante :
  - Jacqueline a choisi seulement un menu *Basique*,
  - Michel a pris un menu *Gourmand* avec boisson,
  - Johanna a choisi un menu *Basique* avec suppléments fromage et café,
  - et Antoine a choisi un menu *Basique* avec boisson et supplément café.

#### 2.8.4 Les moustiques

Depuis le début de l'année 2017, deux scientifiques Marc et Alice étudient l'évolution d'une population de moustiques sur l'île Chépaou. Ils ont réussi à obtenir l'estimation suivante sur l'évolution de la population : si la population contient  $x$  moustiques au cours d'une année, alors il y aura  $1.09x - 200$  moustiques l'année suivante. Par contre, ils ne sont pour l'instant pas d'accord sur l'estimation de la population en 2017 : ils s'accordent seulement sur le fait que ce nombre est compris entre 8 000 et 12 000. Il faudra donc considérer cette donnée comme une variable.

1. Écrire une fonction `f` qui prend en argument le nombre  $x$  de moustiques à une certaine année, et qui renvoie le nombre de moustiques l'année suivante.
2. Écrire une fonction `nb_moustiques` qui prend en arguments `nb_debut`, le nombre estimé de moustiques en 2017, et un entier `annee_voulue`. La fonction doit renvoyer le nombre de moustiques qu'il y aura en `annee_voulue`.
3. Écrire une fonction `annee_atteindra` qui prend en argument un entier `seuil` et un entier `nb_debut` (qui correspondra au nombre de moustiques en 2017) et qui renvoie l'année à partir de laquelle le nombre de moustiques sera supérieur ou égal à `seuil`.
4. Écrire un programme principal qui demande à Marc son estimation du nombre de moustiques actuellement, puis à Alice la sienne. Votre programme demandera ensuite une année et affichera le nombre de moustiques qu'il y aura cette année-là, selon l'estimation initiale de Marc puis selon celle d'Alice. Enfin, votre programme demandera un seuil et affichera en quelle année on atteindrait ce seuil, en fonction de chacune des deux estimations initiales.

- Marc et Alice se rendent compte que leur formule pour l'évolution de la population de moustiques n'était pas correcte. Ils souhaitent la remplacer par  $1.05x - 150$ . Quelle partie de votre programme doit être modifiée ?

### 2.8.5 Triangle de Pascal

- Écrire une fonction `factorielle` qui prend en argument un entier naturel `n` et qui renvoie la valeur de  $n! = 1 \times 2 \times \dots \times n$ .
- Écrire une fonction `coeff_binomial` qui prend en argument deux entiers naturels `n` et `k` (avec  $k \leq n$ ) et qui renvoie la valeur du coefficient binomial correspondant, donnée par la formule ci-dessous :

$$\binom{n}{k} = \frac{n!}{(n-k)! k!}$$

- Écrire une fonction `triangle_pascal` qui prend en argument un entier `nb_lignes` et qui affiche le triangle de Pascal (voir définition ci-dessous) en s'arrêtant au bout du nombre de lignes indiqué par l'argument. Voici un exemple du triangle de Pascal avec 6 lignes :

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

```

En numérotant les lignes et les colonnes à partir de zéro, le nombre sur la ligne numéro  $n$  et la colonne numéro  $k$  est le coefficient binomial  $\binom{n}{k}$  (c-à-d le nombre de combinaisons de  $k$  éléments parmi  $n$ ). Par exemple, pour la colonne numéro 0, on a toujours  $\binom{n}{0} = 1$  (un seul choix, l'ensemble vide), et pour la colonne 1 on a toujours  $\binom{n}{1} = n$  ( $n$  choix : tous les singletons différents). De même, lorsque  $k = n$ , on a  $\binom{n}{n} = 1$  (un seul choix : l'ensemble complet), donc chaque ligne se termine par un 1.

- Bonus* : Comment faire pour éviter le décalage causé par les nombres à plusieurs chiffres ?

## Fonctions qui manipulent des listes

### 2.8.6 Liste d'entiers

- Écrire une fonction `LireListeEntiers` sans argument qui permet de lire des entiers positifs ou nuls saisis au clavier. La saisie s'arrête lorsque l'on entre un nombre négatif, et la fonction renvoie la liste des entiers saisis.
- Écrire le programme principal qui permet d'appeler la fonction et d'afficher ensuite la liste obtenue avec la fonction `print`.
- Proposer une fonction `LireListeReelsBornes` avec deux arguments facultatifs `bmin` et `bmax` (valeurs par défaut 0 et 100 respectivement), qui permet de lire des réels saisis au clavier. La saisie s'arrête lorsque l'utilisateur saisit une valeur en dehors de l'intervalle délimité par les 2 bornes (incluses). La fonction doit renvoyer la liste des éléments lus. **Notion** : paramètres facultatifs.
- Définir une fonction `MMSListe` qui prend en argument une liste et qui renvoie le minimum, le maximum, et la somme des éléments de la liste. **Notion** : valeurs de retour multiples.
- Dans le programme principal, utiliser `LireListeReelsBornes` pour saisir une liste de réels au clavier. Utiliser `MMSListe` pour obtenir le minimum, le maximum et la somme des valeurs saisies ; afficher sur la console les trois valeurs obtenues.

### 2.8.7 Codage et décodage de texte

1. Soit la fonction `NextElem` qui prend en argument un élément `elm` et une liste d'au moins 2 éléments `liste`. La fonction renvoie l'élément suivant `elm` dans la liste `liste`, ou le premier élément de la liste si `elm` est son dernier élément. La fonction renvoie `None` si `elm` n'est pas dans la liste (cf exemple ci-dessous).

```
> print(NextElem('a', ['a', 'b', 'c']))
b
> print(NextElem('c', ['a', 'b', 'c']))
a
> print(NextElem('d', ['a', 'b', 'c']))
None
```

2. On souhaite faire un programme qui crypte un texte en utilisant le principe de "codage par décalage de lettres". Pour cela, on utilise la fonction `NextElem` pour transformer tous les éléments d'une liste. Ci-après un extrait du programme à compléter. On prévoit 2 codages différents (`Code1` et `Code2`). On remarquera qu'une chaîne de caractères se transforme en liste grâce à la fonction `list()`. Une fois la liste `resultat` obtenue, on l'affichera.

```
Code1 = list("abcdefghijklmnopqrstuvwxyz,;:!?.")
Code2 = list("az.erty,uiop:qsdgf!hjklm;wxcv?bn")
texteAvantCodage = input("entrer un texte en minuscule sans accents: ")
... #partie à compléter
print(texteApresCodage)
```

## Pour réviser les boucles for

### 2.8.8 Intervalle avec paramètre optionnel

1. Écrire une fonction `afficheIntervalle` qui reçoit en paramètre 2 entiers `a` et `b`, et qui affiche les entiers de l'intervalle `[a,b]`.
2. Que se passe-t-il si on appelle cette fonction avec un seul paramètre ?
3. Écrire une nouvelle fonction `afficheIntervalle2` qui reçoit en paramètre 2 entiers, le 2e étant optionnel (valeur par défaut = `None`). Cette fonction affiche les entiers de l'intervalle `[a,b]`, ou uniquement l'entier `a` s'il y a un seul paramètre. On appellera la première fonction.
4. Écrire une troisième fonction `afficheIntervalle3` qui reçoit 2 paramètres optionnels (entiers `a` et `b`), et affiche les entiers de l'intervalle `[a,b]`. Si `b` est absent elle n'affiche que `a`. Si `a` est aussi absent elle n'affiche rien. On appellera la 2e fonction.

### 2.8.9 Tables de multiplication

1. Écrire une fonction qui affiche la table de multiplication d'un entier reçu en paramètre, de la manière suivante.

```
>>> table(3)
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

2. Écrire une nouvelle version de cette fonction pour pouvoir préciser deux paramètres optionnels : la valeur de début (par défaut 1) et la valeur de fin (par défaut 10) des multiplicateurs de la table de multiplication affichée. Par exemple :

```
>>> table(3, 2, 4)
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
```

3. Écrire une troisième version de cette fonction qui reçoit un 3e paramètre optionnel : le pas entre les multiplicateurs considérés (par défaut 1). Par exemple :

```
>>> table(3, 2, 4, 2)
3 x 2 = 6
3 x 4 = 12
```

## Exercices pour réviser

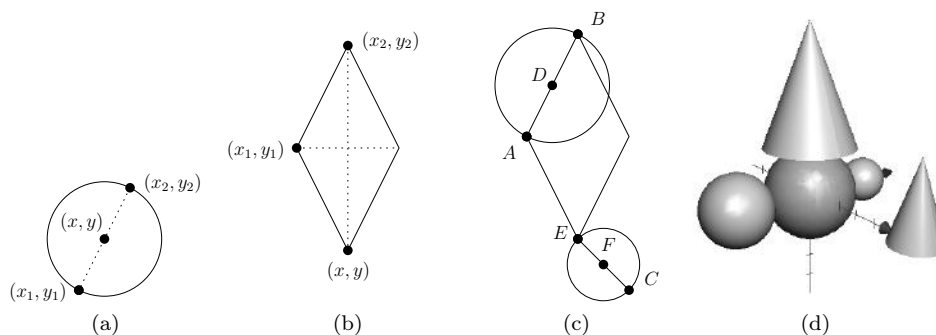
### 2.8.10 Géométrie

#### 1. En 2D

- Écrire une fonction `coord_centre_cercle` qui prend en arguments les coordonnées  $x_1, y_1, x_2, y_2$  de deux points diamétralement opposés sur un cercle (par exemple, comme sur la figure (2.1a)), et qui renvoie les coordonnées  $x, y$  du centre du cercle.
- Écrire une fonction `coord_bas_losange` qui prend en arguments les coordonnées  $x_1, y_1, x_2, y_2$  de deux points placés sur un losange vertical comme à la figure (2.1b) (le premier sur le sommet gauche, le second sur le sommet supérieur) et qui renvoie les coordonnées  $x, y$  du sommet inférieur du losange.
- Écrire un programme principal qui demande les coordonnées des points  $A, B, C$  de la figure (2.1c), puis qui calcule et affiche les coordonnées des points  $D, E, F$ .

#### 2. En 3D

- Écrire une fonction `volume_sphere` qui prend en argument le rayon  $r$  de la sphere (qui vaudra 1 par défaut) et qui renvoie son volume :  $V = 4\pi r^3/3$ .
- Écrire une fonction `volume_cone` qui prend en argument la hauteur  $h$  ainsi que le rayon  $r$  de la base d'un cône de révolution ( $r$  vaudra 1 par défaut) et qui renvoie le volume du cône :  $V = \pi r^2 h/3$ .
- Écrire un programme principal qui permet de calculer le volume total de la figure (2.1d). Votre programme doit demander le rayon de la sphère de gauche, puis le rayon de la sphère du milieu (qui est le même que celui du cône au-dessus), puis le rayon de la sphère de droite, puis le rayon et la hauteur du cône de devant, et enfin la hauteur du cône au-dessus de la sphère. Votre programme affichera le volume total de la figure.



### 2.8.11 Nombres rationnels

Il existe plusieurs manières de représenter un nombre rationnel en Python3. Pour faire simple, nous allons définir chaque nombre rationnel en utilisant deux entiers, représentant le numérateur et le dénominateur. Par exemple :

```
# le nombre x=4/5
x_num = 4
x_den = 5
```

Le but de cet exercice est de définir l'ensemble des opérations de base sur les nombres rationnels (fractions exprimées sous la forme numérateur / dénominateur).

1. Pour commencer, définir la fonction *soustraction*(*x1\_num*,*x1\_den*, *x2\_num*, *x2\_den*) qui prend deux nombres rationnels *x1* et *x2* en paramètre et renvoie le numérateur et le dénominateur de  $x1 - x2$ .

*Rappel* :  $a/b - c/d = (a * d - c * b) / (b * d)$

2. Définir la fonction *addition*(*x1\_num*,*x1\_den*, *x2\_num*, *x2\_den*) en utilisant la fonction soustraction.
3. Définir la fonction *multiplication*(*x1\_num*,*x1\_den*, *x2\_num*, *x2\_den*) et l'utiliser pour définir la fonction *division*(*x1\_num*,*x1\_den*, *x2\_num*, *x2\_den*).

A présent, nous voulons écrire une fonction *vers\_chaine*(*x\_num*,*x\_den*) qui renvoie la représentation en chaîne de caractères du rationnel *x*. Cependant, nous voulons que le nombre soit représenté sous sa forme la plus réduite possible. Par exemple, le nombre  $x = 6/18$  devrait être représenté par la chaîne "1/3".

1. Définir une fonction *pgcd*(*a*, *b*), qui renvoie le plus grand commun diviseur de *a* et *b*.
2. Ecrire la fonction *vers\_chaine*(*x\_num*,*x\_den*) qui retourne une chaîne de caractères décrivant le rationnel *x* sous sa forme la plus simple.

*Rappel* : la fonction *str*() permet de convertir un entier vers une chaîne de caractères. Par exemple : *str*(56) renvoie "56".

Enfin, on veut écrire un programme principal qui lit deux nombres rationnels au clavier et qui affiche leur somme, leur différence, leur produit et leur fraction comme montré dans les exemples d'exécution.

1. Ecrire une fonction *lire*() qui demande à l'utilisateur le numérateur et le dénominateur d'un rationnel et les renvoie après avoir vérifié que le dénominateur est non-nul.
2. Ecrire le programme principal.

Exemples d'exécution:

```
Numérateur: 1
Dénominateur: 2
Numérateur: 1
Dénominateur: 3
1/2 + 1/3 = 5/6
1/2 - 1/3 = 1/6
1/2 x 1/3 = 1/6
1/2 / 1/3 = 3/2
```

```
Numérateur: 3
Dénominateur: 10
Numérateur: 5
Dénominateur: 10
3/10 + 1/2 = 4/5
3/10 - 1/2 = -1/5
3/10 x 1/2 = 3/20
3/10 / 1/2 = 3/5
```

### 2.8.12 Tableau périodique des éléments

Chaque élément chimique a un numéro atomique et des propriétés qui lui sont propres. Par exemple l'hélium (numéro atomique 2) et le néon (numéro atomique 10) sont des gaz nobles.

On veut construire un programme dans lequel on indique par des listes les éléments appartenant respectivement à 3 familles : les gaz nobles, les halogènes, et les "autres non métaux".

#### Squelette du programme

```
##### Code Python #####
# question 2 : parcours simple et definition d'une fonction
def ImprimeFamille
    ...
# question 4 : recherche dans une liste
def isIn
    ...
# prog principal
gaz_noble = [2,10,18,36,54,86]
halogene = [9,17,35,53]
autres_non_métaux = [1,6,7,8,15,16,34]
# question 1 : parcours simple d'une liste
    ...
# question 3 : utilisation d'une fonction
    ...
# question 4
num = int(input("Entrer le numero d'un element chimique : "))
if (num < 1 or num > 103):
    print(num, "n'est pas un numero atomique")
else:
    if isIn(num,gaz_noble):
        print(num,"est un gaz noble")
    elif isIn(num,halogene):
        print(num,"est un halogene")
    elif isIn(num,autres_non_métaux):
        print(num,"est un non métaux")
    else :
        print("Pas encore traite")
```

#### Questions

1. On souhaite afficher les numéros atomiques des gaz nobles. Dans le programme principal, après le commentaire "Question 1", utiliser une itération pour afficher toutes les valeurs de la liste *gaz\_noble*.
2. Après le commentaire "Question 2", écrire la fonction `ImprimeFamille` prenant en argument une liste d'éléments *list\_elem* et affichant le contenu de cette liste sur la console, en utilisant une itération.
3. Après le commentaire "Question 3", utiliser la fonction `ImprimeFamille` pour imprimer les numéros atomiques des halogènes.
4. Après le commentaire "Question 4 ", écrire la fonction `isIn` qui teste si un élément donné par son numéro est dans une famille donnée par la liste de ses éléments.
5. Modifier le programme principal pour qu'il boucle jusqu'à ce que l'utilisateur donne un numéro atomique négatif. Pour chaque numéro positif lu, le programme inscrit sur la console la propriété de l'élément, si celle-ci est connue.

## 2.9 TD9 : dictionnaires

### 2.9.1 Les bases : dictionnaire à clés numériques

Soit le dictionnaire suivant :

```
d = {
    1: [1, 2, 3],
    2: [1, 4, 9],
    'autres': {
        3: None,
        4: [1, 16],
        5: [1, 32]
    }
}
```

1. Que valent les expressions suivantes ?  
 a) `d[1]`      b) `d[2][-1]`      c) `d[d[1][1]]`      d) `d[len(d['autres'][-1])]`
2. On veut supprimer l'élément qui vaut `None`, donner l'instruction qui permet de le faire.
3. On veut rajouter l'entier 81 à la liste associée à la clé 4. Quelle instruction permet de le faire ?
4. On veut mettre les éléments qui sont dans le dictionnaire `d['autres']` dans le dictionnaire `d`, puis supprimer `d['autres']`. Donner le code qui permet de le faire sans utiliser `for`, puis en utilisant `for`.

### 2.9.2 Dictionnaire d'anniversaires

Soit le dictionnaire suivant qui associe à chaque personne (combinaison nom prénom supposée unique) sa date de naissance sous la forme d'une liste de valeurs (jour, mois, année).

```
d = {
    'Michel Durand' : [20, 9, 1963],
    'Sylviane Flinch' : [13, 5, 1977],
    'Lucille Doré' : [30, 4, 2005],
    'Kilian Dupont' : [7, 12, 1999]
}
```

1. Comment récupérer la date de naissance de Kilian Dupont ?
2. Comment ajouter votre date de naissance dans ce dictionnaire ?
3. Comment effacer une date de naissance de ce dictionnaire ?
4. Écrire une fonction `afficheDate` qui reçoit en paramètre ce dictionnaire, demande à l'utilisateur un nom de famille puis un prénom, et affiche la date de naissance sous la forme suivante : `'PRENOM NOM est né(e) le JJ/MM/AAAA'` (en remplaçant les variables en majuscules par leur valeur), ou `'personne inconnue'` si elle n'est pas dans le dictionnaire.
5. Écrire une fonction `listeNatifs` qui reçoit en paramètre ce dictionnaire et une année, et qui renvoie la liste des personnes nées cette année (ou une liste vide s'il n'y en a aucune).



### 2.9.3 Annuaire des plaques d'immatriculation

Soit le dictionnaire suivant qui stocke pour chaque personne (identifiée par la chaîne prénom + nom, supposée unique) son numéro de téléphone (sous forme de chaîne de caractères) et sa plaque d'immatriculation (une autre chaîne de caractères). Les valeurs de ce dictionnaire sont donc de type dictionnaire.

```
d = {
    'Michel Durand' : {
        'tel': '0606060606',
        'plaque': '718 YC 971'
    },
    ...
}
```

1. Comment affiche-t-on à la plaque d'immatriculation de la personne dont le nom est dans la variable `chauffeur` ?
2. Bonus : comment accède-t-on à son numéro de département ? (on suppose la plaque à l'ancien format, avec le numéro de département en dernier, comme dans l'exemple)
3. Écrire une fonction `identifierProprio` qui reçoit en paramètre ce dictionnaire et une plaque d'immatriculation, et qui renvoie le nom et le numéro de téléphone du propriétaire.
4. Comment change-t-on la plaque d'immatriculation d'un chauffeur quand il change de voiture ?
5. Comment change-t-on le nom du propriétaire d'une voiture (d'une plaque d'immatriculation) lorsqu'elle est vendue ? (et donc aussi le numéro de téléphone associé)

### 2.9.4 Personnaliser ses pokémons

Dans un jeu vidéo Pokémon, on souhaite donner des noms personnalisés à certains types de pokémons. Pour ce faire, on définit une fonction `noms_perso(pokemons, noms)`, qui prend en argument une liste de pokémons attrapés, et un dictionnaire faisant correspondre des noms de pokémons originaux à leurs nouveaux noms. La fonction renvoie une nouvelle liste de pokémons, avec les noms modifiés. Les noms qui n'apparaissent pas dans le dictionnaire restent inchangés. Exemple :

```
>>> mes_pokemons = ["pikachu", "bulbizarre", "roucarnage", "lippoutou"]
>>> mes_noms = {"bulbizarre": "jean-jacques", "roucarnage": "gros pigeon"}
>>> noms_perso(mes_pokemons, mes_noms)
["pikachu", "jean-jacques", "gros pigeon", "lippoutou"]
```

### 2.9.5 Dictionnaire français-anglais

1. Écrivez une fonction `ajoute(mot1, mot2, d)` qui prend en argument un mot en français `mot1`, sa traduction en anglais `mot2` et ajoute ces deux mots dans le dictionnaire `d` uniquement si `mot1` n'est pas une clé du dictionnaire.
2. Écrivez une fonction `affiche(d)` qui prend en argument un dictionnaire et affiche à l'écran toutes les valeurs correspondant à ses clés.
3. Écrivez une fonction `supprime(car, d)` qui prend en argument un caractère `car` et un dictionnaire `d` et renvoie un nouveau dictionnaire ne contenant pas les entrées du dictionnaire `d` correspondant à des clés qui commencent par la lettre `car`.
4. Écrivez un programme principal qui demande à l'utilisateur un dictionnaire dont les clés sont 5 mots de la langue française et les valeurs correspondent à la traduction en anglais de chacun de ces mots. Testez les différentes fonctions prédéfinies sur le dictionnaire créé.

### 2.9.6 Polynômes

Dans cet exercice on veut travailler avec des polynômes de degrés quelconques. On peut représenter chaque polynôme avec un dictionnaire, dont les clés correspondent aux puissances de  $x$ , et les valeurs aux coefficients. Par exemple, pour représenter le polynôme  $x^6 + 3x^2$ , on peut utiliser le dictionnaire : `6 : 1, 2 : 3`.

1. Écrire une fonction `evaluer(p, x)` qui prend un polynôme `p` et un nombre `x` en arguments, et renvoie la valeur du polynôme au point `x`.
2. Écrire une fonction `ajoute(p, coeff, puiss)` qui reçoit un polynôme (dictionnaire), un coefficient et une puissance (représentant un monôme), et qui modifie `p` (effet de bord) en lui ajoutant ce monôme. Par exemple `6 : 1, 2 : 3 + 6 : 1` donne `6 : 2, 2 : 3`, alors que `6 : 1, 2 : 3 + 5 : 2` donne `6 : 1, 5 : 2, 2 : 3`. La fonction ne renvoie rien.
3. Écrire une fonction `somme_polynomes(p1, p2)` qui prend deux polynômes (dictionnaires) en arguments et qui renvoie un nouveau dictionnaire représentant la somme des deux polynômes `p1` et `p2`. Attention aux effets de bord indésirables, il ne faut pas modifier `p1` et `p2`.
4. Écrire une fonction `produit_polynomes(p1, p2)` qui prend deux polynômes en argument et renvoie le produit des deux polynômes dans un nouveau dictionnaire. Attention aux effets de bord indésirables, il ne faut pas modifier `p1` et `p2`.

Exemple d'exécution:

```
>>> evaluer({3:1, 1: 2, 0: -1}, 2)
11
>>> somme_polynomes({3:1, 2:1, 0:1}, {4:2, 2:3})
{0: 1, 2: 4, 3: 1, 4: 2}
>>> produit_polynomes({3:1, 2:1, 0:1}, {4:2, 2:3})
{2: 3, 4: 5, 5: 3, 6: 2, 7: 2}
>>> somme_polynomes({2:1, 1:1, 0:1}, {2:1, 1:-1, 0:1})
{0: 2, 2: 2}
```

### 2.9.7 Gestion des notes (simple)

On considère le dictionnaire suivant contenant les notes d'une classe en informatique. Les clés sont les noms des étudiants (supposés uniques), les valeurs sont des listes de notes (à tous les examens d'informatique du semestre). Un étudiant qui a raté un examen aura moins de notes dans sa liste.

```
notes_info = {
    'toto': [12.5, 15, 10, 14],
    'titi': [13, 17, 12, 14.5, 16],
    ...
}
```

1. Écrire une fonction `compteNotes` qui reçoit en paramètres ce dictionnaire et le nom d'un étudiant et qui renvoie le nombre d'examens qu'il a passés.
2. Écrire une fonction `meilleureNote` qui reçoit en paramètres ce dictionnaire et le nom d'un étudiant et qui renvoie sa meilleure note.
3. Écrire une fonction `meilleurSemestre` qui reçoit en paramètre ce dictionnaire, et renvoie le nom et la note de l'étudiant qui a eu la meilleure note du semestre (tous examens confondus).
4. Écrire une fonction `plusAbsent` qui reçoit en paramètre ce dictionnaire, et renvoie le nom et le nombre d'examens passés par l'étudiant qui a été le plus absent (celui qui a le moins de notes dans sa liste).

### 2.9.8 Gestion des notes d'une classe (bis)

On veut gérer les notes d'examens des étudiants d'un groupe. Tous les étudiants du groupe ne suivent pas forcément les mêmes cours, ni le même nombre de cours. Les notes sont organisées dans une liste, dont chaque élément est un dictionnaire contenant le nom de l'étudiant et ses notes dans les matières qu'il a choisies. Exemple :

```
notes_groupe=[
    {'nom': 'lambert', 'notes': {'physique': 12, 'info': 11}},
    {'nom': 'meng', 'notes': {'maths': 13.5, 'info': 18, 'sport': 17}},
    ...]
```

1. Écrire une fonction *moyenne\_etudiant(etudiant)* qui prend en argument un élément de la liste décrite précédemment et qui retourne la moyenne de l'étudiant.
2. Écrire une fonction *meilleur(notes\_groupe)* qui prend la liste du groupe en argument et retourne la meilleure moyenne du groupe ainsi que le nom de l'étudiant qui l'a obtenue.
3. (Difficile) Écrire une fonction *matiere\_difficile(notes\_groupe)* qui retourne le nom de la matière la plus difficile (celle qui a la pire moyenne de groupe).

## 2.10 Gestion des notes (ter)

On travaille avec une liste de dictionnaires de notes pour les étudiants, dont chacun suit un ensemble différent de matières. Le dictionnaire de chaque étudiant a les clés suivantes :

- nom : la valeur est le nom de l'étudiant
- notes : la valeur est un dictionnaire stockant les notes de cet étudiant dans chaque matière (clé = nom de la matière, valeur = la note)
- coeffs : la valeur est un dictionnaire associant chaque nom de matière (clé) avec le coefficient de cette matière pour cet étudiant (valeur) selon son parcours

Par exemple :

```
[{'nom': 'felix', 'notes': {'maths': 20, 'info': 12}, 'coeffs': {'maths': 9,
'info': 4}}, {'nom': 'arthur', 'notes': {'maths': 18, 'physique': 17}, 'coeffs': {'maths': 6,
'physique': 6}}, {'nom': 'roxanne', 'notes': {'maths': 19, 'info': 16}, 'coeffs':
{'maths': 9, 'info': 4}}, {'nom': 'lucien', 'notes': {'maths': 17, 'physique': 19},
'coeffs': {'maths': 3, 'physique': 8}}, {'nom': 'alice', 'notes': {'info': 18,
'physique': 15}, 'coeffs': {'info': 2, 'physique': 8}}, {'nom': 'maxence', 'notes':
{'physique': 8}, 'coeffs': {'physique': 4}}]
```

1. Écrire une fonction qui reçoit un nom d'étudiant, et la liste de dictionnaires, et renvoie le dictionnaire qui est associé à cet étudiant
2. Écrire une fonction qui reçoit un nom d'étudiant et la liste de dictionnaires, et calcule et renvoie sa moyenne en tenant compte des coefficients de chaque matière. On appellera la fonction précédente pour obtenir d'abord le dictionnaire de cet étudiant.
3. Écrire une fonction auxiliaire qui appelle la fonction précédente et crée et renvoie un dictionnaire associant à chaque étudiant sa moyenne.
4. Écrire une fonction qui reçoit la liste de dictionnaires et renvoie le nom du meilleur étudiant (meilleure moyenne générale) et sa moyenne. Cette fonction utilise les précédentes.
5. Écrire une fonction qui reçoit la liste et un nom de matière, et renvoie le nom de l'étudiant qui a la meilleure note dans cette matière, et sa note

6. Écrire une fonction qui reçoit la liste et renvoie le nom de la matière la plus difficile (avec la plus mauvaise moyenne de la classe) ainsi que la moyenne
7. Écrire une fonction qui reçoit la liste de dictionnaires de chaque étudiant, et qui calcule et renvoie une nouvelle liste de dictionnaires, chaque dictionnaire représente une matière, la clé 'nom' doit avoir pour valeur le nom de la matière, et la clé 'notes' doit avoir pour valeur un dictionnaire associant chaque nom d'étudiant (clé) à sa note dans cette matière. *Par exemple :*

```
[{'nom' : 'maths', 'notes' : {'felix':20, 'arthur':18, 'lucien':17,
'roxanne':19}},{'nom' : 'info', 'notes' : {'felix':12, 'roxanne':16,
'alice':18}}, {'nom' : 'physique', 'notes' : {'arthur':17, 'lucien':19,
'alice':15, 'maxence':8}}]
```

8. Écrire une fonction qui reçoit la liste des dictionnaires des étudiants, utilise la fonction ci-dessus pour calculer la liste des dictionnaires des matières, puis parcourt cette liste pour calculer et afficher pour chaque matière le nombre d'étudiants et la moyenne de classe. *Par exemple :*  
*maths, 4 étudiants, moyenne 18.5*

### 2.10.1 Transformer un dictionnaire en liste triée

Écrivez une fonction qui prend en argument un dictionnaire dont les clés sont des entiers, et qui renvoie la liste des valeurs de ce dictionnaire, triée selon l'ordre croissant de leur clé.

### 2.10.2 Rendez-vous

Alice et Bob ont tous les deux un emploi du temps électronique. Chaque événement est modélisé par un dictionnaire qui contient trois clés : "*debut*", "*fin*" et "*titre*". Les valeurs associés aux deux premiers sont des horaires exprimés par un nombre entier de minutes écoulés depuis minuit, la valeur associée au dernier est une chaîne de caractères. Un emploi du temps est une liste qui regroupe les événements d'une journée donnée, **classée dans l'ordre chronologique**, et dont aucun ne se chevauche. Exemple d'emploi du temps :

```
edt = [{"debut": 9*60, "fin": 11*60, "titre": "Réunion"},
       {"debut": 12*60, "fin": 13*60, "titre": "Déjeuner avec John"},
       {"debut": 16*60, "fin": 16.5*60, "titre": "cours de piano"}]
```

1. Écrivez une fonction *espaces\_disponibles(edt, duree)* qui prend en argument un emploi du temps et une durée en minutes (entier), et qui renvoie la liste des indices pour lesquels un événement de durée *duree* pourrait être ajouté avec l'opérateur *insert* sans provoquer de chevauchement.

Exemple d'exécution :

```
>>> espaces_disponibles(edt, 2*60)
[0, 2, 3]
>>> espaces_disponibles(edt, 6*60)
[0, 3]
```

2. Alice souhaite donner rendez-vous à Bob le plus tôt possible dans la journée après son premier événement mais avant son dernier, sans rien avoir à décaler dans les deux emplois du temps. Écrire une fonction *ajouter\_rdv(edtA, edtB, duree, titre)* qui prend en argument les emplois du temps d'Alice et de Bob, une durée en minutes (entier) et un titre (chaîne de caractères), et qui ajoute le rendez-vous dans les deux emplois du temps, si c'est possible. La fonction renvoie un booléen qui indique si le rendez-vous a été ajouté.

Exemple 1 :

```

>>> edtA = [{"debut": 540, "fin": 660, "titre": "Reunion"},
             {"debut": 720, "fin": 780, "titre": "Déjeuner avec John"}]
>>> edtB = [{"debut": 480, "fin": 600, "titre": "Rdv avec Jack"},
             {"debut": 720, "fin": 900, "titre": "Pot de retraite de Joe"}]
>>> ajouter_rdv(edtA, edtB, 1*60, "Rdv Alice Bob")
True
>>> edtA
[{'titre': 'Reunion', 'fin': 660, 'debut': 540},
 {'titre': 'Rdv Alice Bob', 'fin': 720, 'debut': 660},
 {'titre': 'Déjeuner avec John', 'fin': 780, 'debut': 720}]
>>> edtB
[{'titre': 'Rdv avec Jack', 'fin': 600, 'debut': 480},
 {'titre': 'Rdv Alice Bob', 'fin': 720, 'debut': 660},
 {'titre': 'Pot de retraite de Joe', 'fin': 900, 'debut': 720}]

```

Exemple 2 :

```

>>> edtA = [{"debut": 540, "fin": 660, "titre": "Reunion"},
             {"debut": 720, "fin": 780, "titre": "Déjeuner avec John"}]
>>> edtB = [{"debut": 480, "fin": 600, "titre": "Rdv avec Jack"},
             {"debut": 720, "fin": 900, "titre": "Pot de retraite de Joe"}]
>>> ajouter_rdv(edtA, edtB, 2*60, "Rdv Alice Bob")
False

```

3. Alice et Bob n'ont malheureusement pas pu placer leur rendez-vous, mais Alice souhaite vraiment rencontrer Bob. Elle décide de décaler ses autres événements pour rencontrer Bob au premier moment de la journée où il est disponible, mais après 8h. Ecrivez une fonction *imposer\_rdv(edtA, edtB, duree, titre)* qui permettrait de modifier les emplois du temps en conséquence.

Exemple :

```

>>> edtA = [{"debut": 540, "fin": 660, "titre": "Reunion"},
             {"debut": 720, "fin": 780, "titre": "Déjeuner avec John"}]
>>> edtB = [{"debut": 480, "fin": 600, "titre": "Rdv avec Jack"},
             {"debut": 720, "fin": 900, "titre": "Pot de retraite de Joe"}]
>>> imposer_rdv(edtA, edtB, 2*60, "Rdv Alice Bob")
>>> edtA
[{'fin': 720, 'titre': 'Rdv Alice Bob', 'debut': 600},
 {'fin': 840, 'titre': 'Reunion', 'debut': 720},
 {'fin': 900, 'titre': 'Déjeuner avec John', 'debut': 840}]
>>> edtB
[{'fin': 600, 'titre': 'Rdv avec Jack', 'debut': 480},
 {'fin': 720, 'titre': 'Rdv Alice Bob', 'debut': 600},
 {'fin': 900, 'titre': 'Pot de retraite de Joe', 'debut': 720}]

```

## 2.11 TD10 : fichiers

### 2.11.1 Groupe d'étudiants

Écrire un programme qui demande le nombre d'étudiants, puis les prénoms de tous les étudiants d'une classe, et enregistre ces prénoms dans un fichier, un par ligne.

### 2.11.2 Groupe d'étudiants (bis)

Écrire un programme qui lit un fichier `etudiants.txt` (qu'on suppose existant) et affiche les données suivantes :

- Longueur du prénom le plus long dans le fichier
- Longueur moyenne des prénoms dans le fichier
- Prénom le plus représenté (on commencera par créer un dictionnaire pour compter le nombre d'occurrences de chaque prénom) et son nombre d'occurrences
- Nombre de prénoms représentés plusieurs fois

### 2.11.3 Analyse de texte

On veut connaître les lettres de l'alphabet qui sont les plus utilisées dans la langue française. Pour ce faire, on regroupe dans un fichier texte un grand nombre de romans écrits en français (en minuscule) et on souhaite écrire un programme qui permet de compter la fréquence d'apparition de chaque lettre, en ignorant la ponctuation et les caractères spéciaux.

1. Écrire une fonction `lire_fichier(nom_fichier)`, qui prend le nom du fichier contenant le texte en argument, et retourne une chaîne de caractères contenant l'intégralité du texte.
2. Écrire une fonction `liste_spec` qui reçoit un texte et renvoie la liste de ses caractères non alphabétiques (pour simplifier on considère qu'il n'y a pas de lettres accentuées dans le texte).
3. Écrire une fonction `compter_lettres(texte, exceptions)` qui prend un texte en argument et retourne un dictionnaire qui associe à chaque lettre sa fréquence d'apparition dans le texte (le nombre de fois où elle apparaît). L'argument `exceptions` est une liste contenant les caractères spéciaux à ignorer.
4. Écrire une fonction `lettre_rare(frequences)` qui prend le dictionnaire construit dans la question précédente en argument, et retourne la lettre de l'alphabet la plus rare (celle qui apparaît, mais le moins souvent).
5. Écrire le programme principal qui demande à l'utilisateur le nom du fichier, l'ouvre pour le lire, affiche la lettre la plus rare et son nombre d'occurrences, et ferme le fichier.

### 2.11.4 Table harmonieuse

Harmonie a décidé d'inviter ses amis autour d'une seule grande table. Pour que tout soit parfait, elle veut qu'un homme ait au moins une femme à ses côtés et qu'une femme ait au moins un homme à ses côtés. Harmonie a beaucoup d'amis, ils viennent pour la plupart accompagnés. Pour ne pas se torturer l'esprit, elle choisit la méthode informatique pour vérifier que sa table respecte bien ses conditions. Elle dispose de fichiers textes contenant la liste des noms des invités (un fichier pour les hommes et un pour les femmes) et a écrit la disposition de la table dans un autre fichier.

1. Écrire une fonction `read_file(nom)` qui prend en argument le nom d'un fichier texte et renvoie une liste des lignes du fichier.
2. Utiliser cette fonction pour récupérer les fichiers :
  - les prénoms masculins (`homme.txt`)
  - les prénoms féminins (`femme.txt`)
  - la configuration de la table (`table.txt`)

3. Écrire une fonction *genre(prenom, femmes, hommes)* qui prend en argument un prénom, une liste des femmes et une liste des hommes. Cette fonction vérifie si le prénom est masculin ou féminin à l'aide des listes en entrée *femmes* et *hommes*. La fonction retourne **True** si le prénom est féminin, **False** si c'est un prénom masculin.
4. Écrire une fonction *contrainte(table, femmes, hommes)* qui prend en argument la liste des membres de la table et vérifie si la table de Harmonie suit les contraintes qu'elle s'est imposée ou non (un homme a au moins une femme à ses côtés et une femme a au moins un homme à ses côtés). *Attention* : le dernier membre est à côté du premier !
5. Bonus : écrire une fonction qui génère une configuration de table respectant les contraintes (on suppose que c'est possible). Cette fonction doit écrire la configuration dans un fichier *tableAuto.txt* .

### 2.11.5 Analyse de mesures

On considère tout au long de cet exercice des dictionnaires représentant des mesures réelles, de type *float*, faites à divers temps comptés en heures et minutes à partir d'un top de départ "00h00" et sur une durée totale ne pouvant pas dépasser "99h59". Par exemple, { "01h00" :0.5, "02h45" :1.55, "05h00" :2.5, "07h35" :18.5, "10h00" :0.2, "15h11" :4.0 }

Ces mesures sont supposées être les niveaux de concentration d'une protéine tout au long d'une expérience. La chaîne de caractères représentant le temps est normalisée comme dans l'exemple, de sorte que deux temps peuvent être comparés simplement comme des chaînes de caractères.

#### 1. Création d'un fichier de données

Écrire une fonction qui prend en argument le nom d'un fichier à créer, qui demande à l'utilisateur des données représentant les niveaux de concentration d'une protéine à différents temps, et qui écrit un fichier contenant ces informations sous la forme présentée dans la question suivante.

#### 2. Lecture dans un fichier

Ecrire la fonction *chargement* qui prend en entrée le nom d'un fichier dont le contenu a la forme parfaitement normalisée comme ci-dessous, et qui retourne le dictionnaire correspondant.

```
01h00 = 0.5
02h45 = 1.55
05h00 = 2.5
07h35 = 18.5
10h00 = 0.2
15h11 = 4.0
```

Remarquez que, même si le fichier est trié par temps croissants, le dictionnaire ne l'est plus.

INDICATION : la méthode *.split(ch)* appliquée à une chaîne de caractères renvoie la liste des sous-chaînes qui sont séparées par le motif *ch*. Par exemple, "a-c-d-et-r".split('-') renvoie ['a', 'c', 'd', 'et', 'r']

3. Ecrire une fonction *horloge* qui prend en entrée un dictionnaire *d* comme précédemment, et fournit en sortie la liste triée des temps du dictionnaire. **INDICATION** : la méthode *.sort()* appliquée à une liste renvoie une liste triée.
4. Ecrivez la fonction *afficheProfil* qui, à partir d'un dictionnaire représentant un profil, écrit à l'écran, son profil dans l'ordre croissant des temps.
  - Ecrivez la fonction *covariantes* qui prend en entrée deux dictionnaires représentant des mesures effectuées (simul- tanément) sur deux protéines différentes et retourne un booléen disant si elles augmentent et diminuent en même temps (sans pour autant avoir les mêmes mesures). On pourra par exemple d'abord écrire une fonction *listeVariations* qui prend en entrée un dictionnaire et renvoie la liste des variations (attention à l'ordre). Pour ceux qui veulent aller plus loin. Comment modifier votre code pour prendre en compte qu'une variation inférieure à 10% n'est pas significative ?
  - Ecrivez la fonction *cluster* qui prend en entrée :

- d’une part un dictionnaire *ref* donnant les mesures pour une protéine de référence
  - et d’autre part, une liste *l* de dictionnaires représentant chacun les mesures pour une autre protéine (on supposera que les mesures sont simultanées à celles faites pour la protéine de référence)
- et fournit en sortie la liste des dictionnaires covariants avec la protéine de référence.



# Chapitre 3

## Sujets de TP

### 3.1 TP1 : prise en main des outils, lecture de programmes, premiers programmes

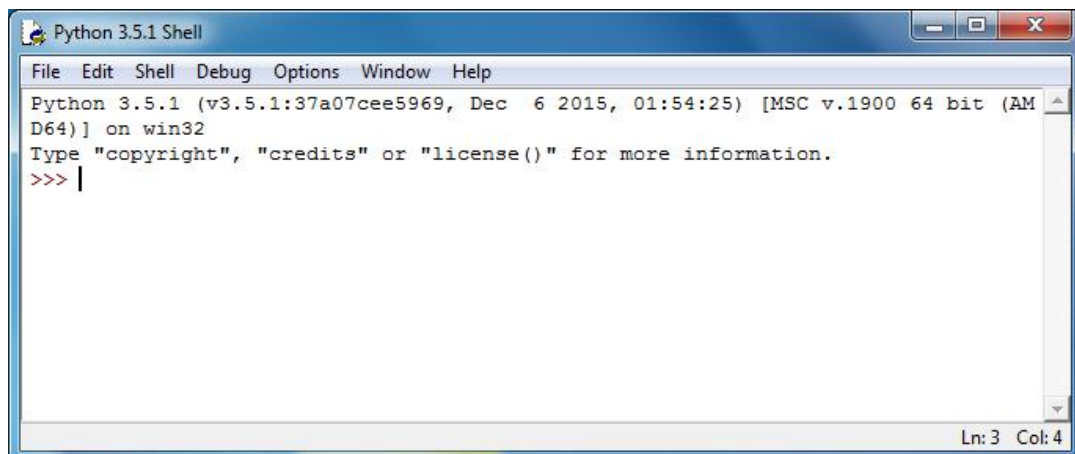
L'objectif de ce TP est d'acquérir la maîtrise des outils qui sont à votre disposition ; pour bien comprendre ce qui est en jeu, faites tous les essais qui vous paraîtront nécessaires.

**CaseIne :** des exercices supplémentaires, des fichiers complémentaires pour certains TP, des diapos de cours, et autres ressources, sont disponibles sur le cours en ligne Python sur CaseIne, accessible avec vos identifiants de connection UGA. <http://caseine.org/course/view.php?id=86>

#### 3.1.1 Première prise en main de Idle

Idle est un environnement de développement - c'est-à-dire un outil permettant de faciliter la programmation - en Python.

Lancer le programme Idle. Idle lance une fenêtre contenant plusieurs menus. Cette fenêtre est un interpréteur de commandes pour Python, vous pouvez y taper directement des lignes en Python. Cela se présente plus ou moins ainsi :



Dans la suite des TP, nous allons écrire nos programmes dans des fichiers et non directement dans cette fenêtre. Mais cette fenêtre reste utilisable directement à tout moment, par exemple pour tester certaines instructions. C'est pratique car nous obtenons directement le résultat (cela peut aussi vous servir de calculatrice puissante). Vous pouvez par exemple taper les lignes suivantes :

```
>>> 3 + 5
>>> "bon" + "jour"
>>> "3" + "5"
>>> 3 + "5"
```

```
>>> 3 + int("5")
>>> a = 3
>>> print(a)
>>> type(a)
>>> a = a + 17
>>> b = a * 2
>>> print("a =", a, "et b =", b)
>>> a == b
>>> type(a == b)
>>> type(a = b)
>>> 17 // 5
>>> 17 / 5
>>> 17 % 5
>>> 17.0 / 5
>>> type(3.3)
```

Une ou plusieurs de ces lignes provoquent des erreurs, c'est normal. Surlignez-les. Avez-vous pris le temps de lire et de comprendre les messages d'erreur affichés ?

### 3.1.2 Ecriture et exécution d'un programme sous Idle

- Dans le menu **File** (comme fichier) choisir **New File**. Idle ouvre une fenêtre nommée "Untitled" dans laquelle on va pouvoir écrire le texte *source* du programme ; cela permet aussi de garder une trace de son travail. Commencez par enregistrer le fichier (*File > Save*) sous le nom `TP1_exo1.py`.

**Attention !** veillez à toujours sauver vos fichiers sur votre répertoire personnel (**home**). Il est conseillé d'y créer un dossier nommé **INF101** pour y ranger tous les TP de l'UE. Vous pouvez aussi copier votre travail sur une clé usb si vous en avez une.

- Ecrire maintenant le texte du programme suivant :

```
print("Premier programme")
nom = input("Donnez votre nom : ")
print("Bonjour", nom)
```

Sauvegardez votre travail. Vous allez maintenant exécuter ce programme : pour cela allez dans le menu **Run** et choisissez **Run Module** (vous pouvez aussi appuyer directement sur la touche **F5**).

Le programme s'exécute alors dans la première fenêtre, celle de l'interpréteur Python (qui est nommée `Python 3.6.1 Shell`).

- Essayez maintenant d'introduire une erreur, c'est-à-dire modifiez le texte de façon à ne pas respecter les conventions du Python ; ensuite sauvegarder, puis demander l'exécution. Que se passe-t-il ?
- Ensuite, modifiez le texte du programme de façon que s'affiche à l'exécution :

```
Premier programme, deuxieme version
Donnez votre nom en majuscules :
PAUL
Bonjour, PAUL, comment vas-tu ?
```

Faites des essais en modifiant le texte du programme jusqu'à obtention de l'affichage requis.

### 3.1.3 Découverte de Python Tutor

Python Tutor est un outil en ligne qui permet de mieux comprendre ce qui se passe quand l'ordinateur exécute un programme (en Python mais aussi en d'autres langages de programmation). Cela deviendra particulièrement utile au fil des semaines quand nous aborderons des notions de plus en plus complexes.

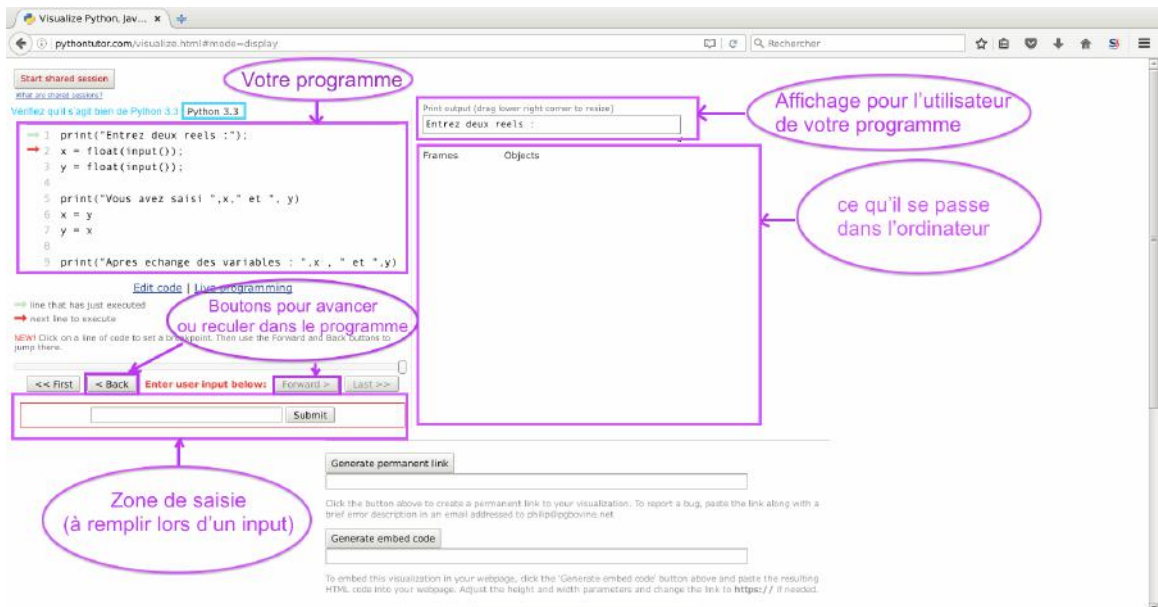
- Pour utiliser Python Tutor, allez sur le site <http://pythontutor.com/> (ajouter ce site à vos marque-pages pour y revenir plus rapidement aux prochaines séances).
- Choisissez Python 3.6 (pas Python 2.7!)
- Cliquez sur *“Start visualizing code”*
- Recopiez le code ci-dessous :

```
# programme pour échanger les valeurs de deux variables
print("Entrez deux reels :")
x = float(input())
y = float(input())

print("Vous avez saisi ",x," et ", y)
x = y
y = x

print("Après échange des variables : ",x , " et ",y)
```

- Cliquez sur “Visualize Execution” puis avancez ligne par ligne dans l'exécution du programme (une zone de saisie apparaît sous la zone où vous avez tapé votre programme quand une saisie est attendue). A droite, vous pouvez voir l'affichage produit par votre programme et surtout une visualisation de ce qui se passe dans la mémoire de l'ordinateur, c'est-à-dire ici les variables existantes et leur valeur.



Appuyez sur **Forward** (resp. **Back**) pour avancer (resp. reculer) d'une ligne dans le programme.

- Corrigez maintenant le programme pour qu'il fasse réellement l'échange entre les deux valeurs des variables (avec une variable temporaire dans un premier temps).

### 3.1.4 Exercice de lecture

*Inspiré d'un exercice du partiel 2016*

On considère le programme ci-dessous, censé déterminer la valeur absolue de la différence de deux entiers saisis au clavier. (Rappel : la valeur absolue de -4 est 4, celle de 7 est 7, celle de -2.5 est 2.5, etc...)

```
# Programme valeur absolue
text = input('Donner un entier : ')
x = int(text)
text = input('Donner un entier : ')
y = int(text)
z = x - y
if (z < 0)
    resultat = -z
else :
    z = resultat
print 'valeur absolue :', resultat
```

1. trouver les erreurs présentes dans ce programme, et corriger chaque ligne fausse
2. que se passe-t-il si vous enlevez les `int(...)` autour des `text` ?
3. écrire ce programme (corrigé) sous Python Tutor et observer son exécution.

### 3.1.5 Quelques exercices

On travaille à nouveau sous Idle.

#### Un peu d'arithmétique

1. Copier et faire exécuter ce programme (le sauver dans un fichier) :

```
print("Calcul de la somme de 2 nombres entiers")
a = int(input("Donner un nombre : "))
b = int(input("Donner un autre nombre : "))
c = a + b
print("\nLa somme est : ", c, "\n")
```

Modifiez le programme précédent pour qu'il demande deux nombres, et affiche la somme, la différence et le produit de ces deux nombres.

2. Ecrire un programme qui demande deux nombres entiers et affiche un message pour dire si les nombres sont égaux ou différents :

```
(si l'on a tapé 3 et 5)
les deux nombres sont differents
```

```
(ou bien, si l'on a tapé 4 et 4)
les deux nombres sont egaux
```

3. Ecrire un programme qui demande deux nombres, et affiche le quotient entier du premier par le deuxième lorsque le deuxième n'est pas nul, ainsi que le reste, précédés du message :

Le quotient de (ici le nombre) par (l'autre nombre) est : (le quotient). Le reste est (le reste).

Lorsque le deuxième nombre est nul, le programme doit afficher le texte : **Erreur, division par zéro impossible.** (Rappel : comme  $14 = 4 \times 3 + 2$ , alors le quotient de 14 par 4 est 3, et le reste est 2.)

## Jeux de dés

1. Écrire un programme `DeuxDes.py` qui lit deux entiers, qui vérifie que ces entiers sont compris entre 1 et 6 (valeurs d'un dé à jouer), et qui, dans le cas de valeurs correctes, affiche les valeurs des deux dés dans l'ordre décroissant (le plus grand d'abord).

Voici un exemple d'affichage du programme :

```
Donner la valeur du premier dé : 4
La valeur du premier de est correcte
Donner la valeur du deuxieme de : 6
La valeur du deuxieme de est correcte
Les des classes en ordre decroissant sont : 6 4
```

2. Écrire un programme `TroisDes.py` :
  - qui lit trois entiers
  - qui vérifie que ceux-ci sont compris entre 1 et 6
  - qui dans le cas de valeurs correctes, affiche les trois dés dans l'ordre décroissant
  - qui, en plus, dans le cas des trois valeurs 4, 2 et 1, affiche "Gagne!", et dans les autres cas corrects, affiche "Perdu!".
  - (dans le cas de valeurs incorrectes le programme ne fait rien)

## Calcul de la moyenne et de la mention d'un étudiant

1. Écrire un programme qui calcule la moyenne d'un module qui comporte quatre disciplines :

- Chimie
- Physique
- Mathématiques
- Informatique

Le programme demandé

- lit les quatre notes de l'étudiant
- calcule la moyenne des notes et l'affiche

Si une note n'est pas comprise entre 0 et 20, le programme affiche "INCORRECT" au lieu d'afficher la moyenne.

2. Écrire une nouvelle version de ce programme mais qui, dès qu'il lit une note incorrecte, affiche "INCORRECT" et ne demande pas les notes suivantes.
3. Écrire un programme qui calcule la moyenne pondérée d'un module comportant quatre disciplines avec les coefficients suivants :

Chimie	3
Physique	4
Mathématiques	2
Informatique	2

Le programme demandé lit les quatre notes de l'étudiant, calcule la moyenne pondérée des notes et l'affiche. Si une note n'est pas comprise entre 0 et 20, le programme affiche "INCORRECT".

4. Écrire un programme qui calcule la moyenne pondérée d'un module comme précédemment et affiche en

	moyenne	< 10	:	AJOURNE	
	10 ≤	moyenne	< 12	:	PASSABLE
plus la mention de l'étudiant. Les mentions sont :	12 ≤	moyenne	< 14	:	ASSEZ BIEN
	14 ≤	moyenne	< 16	:	BIEN
	16 ≤	moyenne		:	TRES BIEN

5. Modifier ce programme pour qu'il donne une 2e chance à l'utilisateur quand il saisit une note incorrecte.

**Bonus si vous connaissez déjà les boucles while**

1. Écrire une nouvelle version de ce programme qui filtre chaque note entrée jusqu'à ce qu'elle soit bien comprise entre 0 et 20
2. Écrire une nouvelle version de ce programme qui :
  - Demande à l'utilisateur combien de disciplines compte le module
  - Pour chacune de ces disciplines, lui demande sa note puis le coefficient de cette discipline
  - Calcule et affiche la moyenne pondérée du module
  - Affiche la mention correspondante

## 3.2 TP2 : importation de modules, nombres aléatoires, boucles while

### 3.2.1 Les modules Python

Beaucoup de problèmes en informatique ont déjà été résolus. Par exemple, comment calculer le logarithme d'un nombre réel, ou, comment établir une connexion internet avec un ordinateur distant ? Un programmeur qui souhaite résoudre un nouveau problème qui utilise ces fonctionnalités ne devrait pas avoir à tout reprogrammer. C'est pourquoi, comme beaucoup d'autres langages de programmation, Python regroupe les fonctions couramment utilisées dans ce que l'on appelle des *modules*.

Chaque module en python contient des fonctions permettant de résoudre des problèmes spécifiques à thème donné. Par exemple, le module 'math' contient des fonctions telles que le cosinus, le sinus, la racine carrée, etc. Le module 'email' contient des fonctions permettant de créer, d'envoyer et de recevoir des emails. Comparée à d'autres langages de programmation tels que C, Python a une bibliothèque de modules très riche. Dans ce TP, nous allons apprendre à utiliser deux de ces modules : **math** et **random**.

#### Le module math

Ouvrez IDLE et tapez la commande suivante :

```
>>> import math
```

Cette directive permet d'accéder à toutes les fonctions du module 'math'. Essayez d'exécuter les instructions suivantes et vérifiez qu'elles donnent le résultat attendu :

```
>>> math.cos(0)
...
>>> math.sin(math.pi / 2)
...
>>> math.tan(math.pi / 4)
...
>>> math.tan(math.pi / 3)
...
>>> math.sqrt(16)
...
>>> math.log(10)
...
>>> math.exp(math.log(89))
...

```

Pour accéder au contenu du module math (fonctions et constantes telles que  $\pi$ ), on a donc besoin de rajouter 'math.' avant la fonction ou la constante à chaque fois. Il est toutefois possible d'importer directement les fonctions qui nous intéressent sans passer par le nom du module :

```
>>> from math import cos, pi, exp, log
>>> cos(pi)
...
>>> x = exp(log(2))
>>> print(x)
...

```

Dans ce cas, seuls 'cos', 'pi', 'exp' et 'log' sont importés, et peuvent être référencés directement. On peut également importer tout le contenu du module en utilisant le caractère spécial '\*', qui veut dire 'tout'.

```
>>> from math import *
>>> x = log(2) + sqrt(exp(1)) * tan(pi/8)
>>> print(x)
...

```

Cependant, on évitera cette méthode d'importation car certains modules peuvent contenir des noms identiques aux noms des variables de notre programme, ce qui risque de créer des conflits.

1) Ecrire un programme qui demande à l'utilisateur 2 nombres  $h$  et  $a$ , correspondant à l'hypoténuse et à l'un des deux angles non-droits (en radians) d'un triangle rectangle. Le programme doit afficher la longueur des deux côtés restants du triangle.

*Indice* : La somme des angles d'un triangle est égale à  $\pi$ .

*Rappels de trigo* :  $\cos(\alpha) = \frac{\text{adjacent}}{\text{hypotenuse}}$ ,  $\sin(\alpha) = \frac{\text{oppose}}{\text{hypotenuse}}$ , et  $\tan(\alpha) = \frac{\text{oppose}}{\text{adjacent}}$

Exemple d'exécution :

Entrez la longueur de l'hypothénuse: 2

Entrez la valeur d'un angle: 1.0472

Les deux côtés sont: 0.9999957585450914 , 1.7320532563670865

1b) Après calcul des autres côtés, vérifier que les valeurs trouvées vérifient bien le théorème de Pythagore.

2) Ecrire un script qui demande à l'utilisateur 3 nombres :  $a$ ,  $b$  et  $c$  en boucle jusqu'à ce que la valeur  $\text{delta} = b^2 - 4ac$  soit supérieure ou égale à 0. Le programme affichera alors les racines du polynôme  $ax^2 + bx + c$ .

### Le module random

Certains types de programme nécessitent l'introduction d'un comportement aléatoire. Par exemple, dans le cas d'un jeu, on veut que le joueur ait une expérience différente à chaque exécution du programme. Le module 'random' de Python permet de générer des nombres aléatoires. En particulier, on s'intéressera à la fonction `randint` de ce module.

1) Importer le module random.

2) Exécuter les instructions suivantes plusieurs fois de suite, qu'observez-vous ?

```
>>> x = random.randint(1, 4)
```

```
>>> print(x)
```

3) Écrire un script qui lit deux entiers  $a$  et  $b$  en vérifiant que  $a < b$  et qui affiche un nombre aléatoire compris entre  $a$  et  $b$  inclus.

### 3.2.2 Prix du gros lot

Vous allez à présent développer un petit jeu dont les règles sont les suivantes :

- Le programme tire un nombre au hasard compris entre 0 et 100 inclus et ne l'affiche pas à l'écran.
- Le joueur doit deviner la valeur de ce nombre en saisissant sa proposition au clavier.
- Le programme affiche selon le cas :
  - "Gagné" si le joueur a deviné le bon nombre. Le programme se termine aussitôt.
  - "Trop grand" si le joueur a saisi un nombre plus grand que le bon prix.
  - "Trop petit" si le joueur a saisi un nombre plus petit que le bon prix.

### Le jeu de base

1. Écrire le programme décrit ci-dessus. Le joueur peut continuer à essayer jusqu'à ce qu'il devine le bon nombre.
2. Modifier pour afficher les nouvelles bornes (inférieures et supérieures) à chaque essai raté. Par exemple si le nombre secret est 77 :

```
Devine mon nombre entre 0 et 100
50
Trop petit
Devine mon nombre entre 51 et 100
80
Trop grand
Devine mon nombre entre 51 et 79
```



3. Modifier le programme pour que le joueur n'ait que 5 essais avant que le programme ne se termine en affichant le message 'Perdu!'. Si le joueur gagne après X essais, le programme doit afficher "Gagné au bout de X essais!".
4. Modifier pour afficher le nombre d'essais restants à chaque tentative.

```
Devine mon nombre entre 0 et 100 (5 essais)
50
Trop petit
Devine mon nombre entre 51 et 100 (4 essais)
```

5. Modifier le programme pour qu'à la fin de chaque partie, le message "Voulez-vous recommencer ? (o/n)" soit affiché. Si l'utilisateur tape 'o', une nouvelle partie du jeu commence, s'il tape 'n', le programme se termine en affichant :
  - Combien de parties le joueur a jouées et combien il en a gagnées (format : 'tu as gagné 2 parties sur 5 jouées')
  - Combien d'essais il a fallu au joueur en moyenne (sur l'ensemble des parties **gagnées**, uniquement s'il y en a au moins une) pour trouver la bonne réponse (format : 'tu as mis en moyenne 3.7 essais pour trouver la solution')

### Une IA pour le prix du gros lot

On veut maintenant écrire un programme qui sait deviner votre nombre secret. Cette fois les rôles sont inversés, vous choisissez un nombre au hasard et c'est à votre programme de le deviner. Vous devez lui répondre 'g' si sa proposition est trop grande, 'p' si sa proposition est trop petite, et 'b' s'il a deviné le bon nombre. Le nombre d'essais n'est pas limité.

1. Écrire une première version "IA aléatoire" qui affiche chaque proposition de l'ordinateur (tirée au hasard entre les bornes inf et sup) et attend la réponse ('p', 'g', ou 'b') de l'utilisateur. Le programme met à jour la borne inf ou sup en fonction de l'indice répondu par l'utilisateur, et les affiche aussi. Par exemple si vous avez choisi 88 :

```
Je cherche un nombre entre 0 et 100
Je propose 50 ? p
Je cherche un nombre entre 51 et 100
Je propose 93 ? g
Je cherche un nombre entre 51 et 92
Je propose 88 ? b
J'ai gagné
```

2. Automatiser le rôle du "maître du jeu". Ce n'est plus l'utilisateur qui choisit un nombre secret, mais le programme qui tire un nombre secret aléatoire, puis essaye de le deviner. Cette fois l'IA n'attend plus la réponse de l'utilisateur, c'est le programme qui vérifie si sa proposition est correcte. Le programme affiche chaque essai au fur et à mesure, l'indice correspondant ('p' ou 'g'), et le nombre d'essais à la fin.
3. Écrire un nouveau programme qui fait jouer n parties (n sera demandé à l'utilisateur) à l'ordinateur (avec un nouveau nombre secret tiré aléatoirement à chaque partie), et affiche son nombre moyen d'essais par partie gagnée à la fin (comme on n'a pas limité le nombre d'essais, le programme finit forcément par gagner).
4. Il y a plusieurs manières de programmer l'IA. Trouver un algorithme plus efficace que l'essai aléatoire et programmer cette "IA intelligente". *Indice* : inspirez vous de votre propre raisonnement quand vous essayez de deviner un nombre.
5. Faire jouer n parties aux 2 IA et comparer leur nombre moyen d'essais. Que se passe-t-il si n est faible ? Si n est grand ?

6. On remarquera qu'en moyenne l'IA "intelligente" est meilleure que l'IA "aléatoire". Mais certains nombres sont plus faciles à deviner que d'autres avec la méthode "dichotomique". Écrire un nouveau programme qui fait jouer  $n$  parties à chaque IA sur chacun des nombres secrets possibles (entre 0 et 100 inclus). On affichera pour chaque nombre secret la moyenne d'essais des 2 IA.
7. Afficher les résultats sous forme d'un histogramme textuel : pour chaque nombre secret (entre 0 et 100) afficher deux lignes de caractères dont la longueur dépend du nombre moyen d'essais, une ligne par IA. Par exemple utiliser le caractère '\*' pour l'IA intelligente, et le caractère '-' pour l'IA aléatoire. Que remarque-t-on ? Par exemple

```
[0] ****
[0] -----
[1] ****
[1] -----
etc
```

### Bonus : affichage graphique

La librairie `matplotlib` permet de créer des graphiques<sup>1</sup>. Utiliser cette librairie pour afficher sur un même graphique les courbes du nombre moyen d'essais de chaque IA, pour toutes les valeurs entre 0 et 100. Penser à afficher les 2 courbes avec des couleurs différentes et à ajouter une légende.

Vous pouvez aussi créer d'autres algorithmes qui jouent à ce jeu avec d'autres méthodes, puis calculer et afficher leurs statistiques sur le même graphique.

---

1. Voir par exemple la page <http://apprendre-python.com/page-creer-graphiques-scientifiques-python-apprendre>

## 3.3 TP3 : Initiation aux fonctions avec turtle

Dans ce TP, vous allez :

- découvrir comment faire des dessins avec le module `turtle`,
- consolider votre pratique des boucles `while` imbriquées,
- vous initier à la notion de fonction.

### 3.3.1 Marchons au pas de turtle

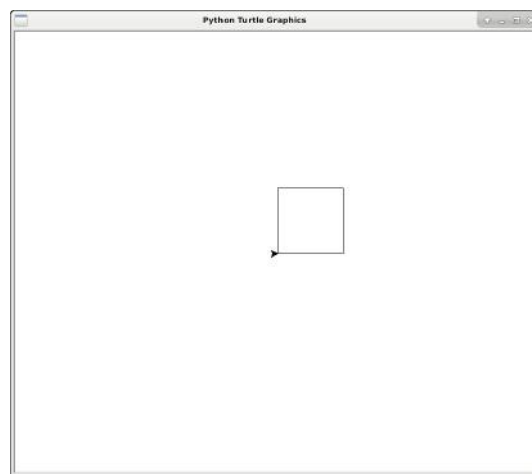
Rappel : pour utiliser les fonctions d'un module, il faut utiliser le mot clé `import` suivi du nom du module.

Le module `turtle` est un module graphique, inspiré de la programmation Logo,<sup>2</sup> qui permet de déplacer une tortue sur l'écran. Quand la tortue se déplace et que le crayon est baissé, sa trajectoire se trace à l'écran. Nous allons l'utiliser pour dessiner des figures.

Quand on utilise une fonction du module `turtle`, cela ouvre une fenêtre graphique, et le curseur est initialement placé au point de coordonnées (0,0) qui est au milieu de la fenêtre. L'orientation des axes (abscisses et ordonnées) est identique à celle conventionnellement utilisée en mathématiques. Sur la figure ci-dessous, le coin inférieur gauche du carré a pour coordonnées (0,0) et son coin supérieur droit (100,100). La taille de la fenêtre peut varier d'un ordinateur à l'autre mais elle peut être changée en utilisant la fonction `setup(x,y)`.

Voici une liste non exhaustive<sup>3</sup> des fonctions disponibles dans cette bibliothèque :

- `reset()` : efface l'écran, recentre la tortue et remet les variables à zéro
- `forward(distance)` : avance d'une distance donnée (en pixels)
- `backward(distance)` : recule d'une distance donnée (en pixels)
- `left(angle)` : pivote vers la gauche d'un angle donné en degrés
- `right(angle)` : pivote vers la droite d'un angle donné en degrés
- `up()` : relève le crayon (pour pouvoir avancer sans dessiner)
- `down()` : abaisse le crayon (pour recommencer à dessiner)
- `goto(x, y)` : va à l'endroit de coordonnées (x, y) (attention à ne pas oublier d'utiliser la fonction `up()` avant d'utiliser `goto(x,y)` car sinon il tracera le parcours effectué)
- `circle(r)` : trace un cercle de rayon r
- `setup(x,y)` : pour fixer les dimensions de la fenêtre graphique : x est la largeur et y la hauteur.
- `color(str)` : change la couleur du dessin
- `speed(int)` : change la vitesse de déplacement de la tortue



2. Si vous voulez retrouver un dessin plus proche de l'original de la tortue Logo vous pouvez utiliser `shape('turtle')` pour changer le curseur en tortue.

3. Il existe d'autres fonctions. Pour connaître la liste de toutes les fonctions disponibles dans le module `turtle`, ainsi que comment les utiliser, visitez la page de la documentation en ligne du module : <https://docs.python.org/3/library/turtle.html>

Nous allons travailler sous `idle`. Pour chaque exercice, vous écrirez votre programme dans un nouveau fichier. **ATTENTION!** ne nommez **jamais** vos fichiers `turtle.py`, sinon c'est votre propre fichier qui est importé au lieu du module `turtle`...

### Carré

Écrivez un programme permettant de tracer un carré de côté 100. A la fin du tracé, le curseur doit être au même endroit et dans la même direction qu'au début du tracé.

*Si vous n'avez pas utilisé de boucle, modifiez votre programme pour utiliser une boucle `while`. Vous ne devez utiliser qu'une fois la fonction `turtle.forward(...)`.*

### Carré bis

Modifiez le programme précédent pour dessiner un carré identique au précédent dont le coin supérieur gauche a pour coordonnées le point  $(-50, 20)$ . A la fin du tracé, le curseur doit être au même endroit  $(-50, 20)$  et dans la même direction qu'au début du tracé.

*Attention : à l'exécution de votre programme, seul le carré doit être tracé ! Si le point  $(-50, 20)$  n'est pas le coin supérieur gauche mais plutôt le coin inférieur gauche de votre carré, essayez de tourner dans l'autre sens.*

### Triangle équilatéral

Écrivez un programme permettant de tracer un triangle équilatéral de côté 100 avec la pointe dirigée vers le bas. A la fin du tracé, le curseur doit être au même endroit et dans la même direction qu'au début du tracé.

*Si vous n'avez pas utilisé de boucle, modifiez votre programme pour utiliser une boucle `while`. Vous ne devez utiliser qu'une fois la fonction `turtle.forward(...)`.*

### Ligne de carrés

On souhaite maintenant obtenir une ligne de 8 carrés comme sur la figure suivante. Les carrés ont une dimension de 50. La distance entre deux carrés successifs est de 10. Pour l'instant, on ne se soucie pas vraiment de la position du curseur au début ou à la fin du tracé.



*Vous devez avoir deux boucles `while` imbriquées, et seuls les carrés doivent être tracés, sans être reliés par des traits.*

Savez-vous comment modifier votre programme pour que le coin supérieur gauche du premier carré (le plus à gauche) ait pour coordonnées  $(-200, 200)$  ? Faites-le.

### Pavage de carrés

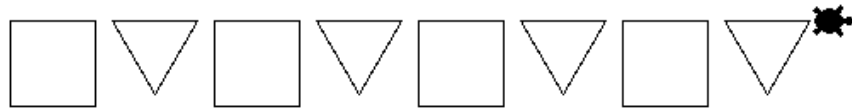
Utiliser une 3e boucle `while` (imbriquées) pour dessiner un pavage : 4 lignes de 8 carrés de côté 50.

### Ligne de carrés et triangles alternés

On souhaite maintenant dessiner une figure très similaire à la précédente mais en alternant carrés et triangles comme ci-dessous. Voici les caractéristiques imposées du dessin :

- on commence par un carré
- les carrés et les triangles ont tous des côtés de 50
- sur la ligne du haut, l'écart entre deux figures consécutives est toujours de 10

- le coin supérieur gauche du premier carré a pour coordonnées (-200,200)



Modifiez maintenant votre programme pour laisser l'utilisateur choisir (avec un `input()`) combien de figures il souhaite tracer en tout. Par exemple, si le tracé ci-dessus s'obtient maintenant en demandant 8 figures.

### 3.3.2 A la découverte des fonctions à travers le dessin

#### Programme mystère

Ecrivez le programme ci à droite dans un nouveau fichier. Exécutez ce programme. Quel résultat obtient-t-on? Ajoutez l'instruction `turtle.circle(115)` à la fin de votre programme et exécutez.

```
import turtle

i = 0
while i < 360 :
    turtle.forward(2)
    turtle.left(1)
    i=i+1
```

Nous avons trouvé comment tracer une figure proche d'un cercle avec le module `turtle` mais pour tracer des cercles, nous n'utilisons pas ce programme car la fonction `circle(r)` existe déjà dans le module `turtle` et que c'est beaucoup plus pratique à utiliser. De la même façon, dans les exercices précédents, il aurait été pratique de disposer d'une fonction pour tracer un carré ou un triangle équilatéral, en spécifiant sa taille. Justement, en programmation, il est possible de définir ses propres fonctions. Une fonction est un ensemble d'instructions regroupées sous un nom et s'exécutant à la demande. Cela a plusieurs avantages :

- cela évite de répéter du code,
- cela permet de limiter les erreurs notamment de copier/coller : une fois qu'on a écrit et testé notre fonction `carre`, on sait que cela marchera à chaque fois qu'on en aura besoin
- cela permet d'améliorer la lisibilité d'un programme (on comprend plus vite ce que fait le programme quand on voit écrit `carre`)

#### une première fonction

Vous allez maintenant écrire votre première fonction en Python. Recopiez la première partie du programme à droite (au dessus de la ligne horizontale).

Que se passe-t-il quand vous exécutez ce programme?

En fait, ici, vous avez *défini comment* tracer un carré, mais vous ne demandez pas à ce qu'un carré soit tracé. Une fonction est une sorte de "sous-programme" qui donne la méthode pour effectuer une tâche, mais ensuite il faut faire **appel** à cette méthode quand on veut que la tâche soit effectuée. Pour cela, on fait suivre notre définition de fonction par un programme principal, écrit à la suite dans le même fichier. Complétez donc le programme précédent en lui ajoutant les lignes à droite, puis exécutez le programme.

```
import turtle

def carre() :
    # trace un carre de taille 100
    i = 1 # compteur du nb de cotes
    while i <= 4 :
        turtle.forward(100)
        turtle.right(90)
        i = i + 1

# programme principal, avec appel de fonction
carre()
turtle.up()
turtle.forward(130)
turtle.down()
carre()
```

En pratique, nous nous trouvons vite limités avec la fonction `carre` que nous venons de définir. Si nous

voulons tracer des carrés de taille 50, et que pour cela il faut modifier la fonction ou en écrire une autre, cela perd beaucoup de son intérêt. Il serait intéressant de pouvoir spécifier une taille au moment de tracer un carré, de la même façon que quand on utilise la fonction `circle` de `turtle` en spécifiant un rayon. Pour cela il faut définir un paramètre à la fonction.

### fonction carre avec paramètre

Écrivez maintenant le code ci à droite et complétez-le avec une programme principal de façon à appeler la fonction pour tracer plusieurs carrés de tailles différentes. Attention, pour appeler cette fonction il faudra spécifier entre les parenthèses la valeur qu'on souhaite donner au paramètre `cote`.

```
import turtle

def carre(cote) :
    # trace un carre de taille determinee : cote
    i = 1 # compteur du nb de cotes
    while i <= 4 :
        turtle.forward(cote)
        turtle.right(90)
        i = i + 1
```

En fait, une fonction peut aussi avoir plusieurs paramètres, ainsi, on aurait pu définir une fonction `carre(x,y,cote)` où `(x,y)` désigneraient les coordonnées du coin supérieur gauche du carré et `cote` sa taille. Dans ce cas, l'appel se fera avec 3 nombres entre parenthèses, le premier correspondant au `x` souhaité, le deuxième au `y` et le troisième au `cote`. Ainsi, l'instruction `carre(-50, 20, 100)` permettrait de tracer le carré demandé dans l'exercice *Carré bis*. Essayez de faire cette modification. Pour la suite, vous choisirez la version de la fonction paramétrée qui vous est la plus pratique.

### fonction pour tracer un triangle équilatéral

En vous inspirant de la fonction pour tracer un carré, écrivez une fonction `triangle(cote)` qui trace un triangle équilatéral de côté `cote` avec la pointe orientée vers le bas. Par la suite, vous pourrez modifier cette fonction pour y ajouter des paramètres `(x,y)` correspondant aux coordonnées du sommet en haut à gauche.

### ligne de carrés et triangles avec fonctions

Dans un même fichier, vous allez définir la fonction `carre`, la fonction `triangle` et les utiliser dans un programme principal pour dessiner la même ligne de carrés et triangles alternés qu'à l'exercice 5 (version avec 8 carrés). Voici à droite la structure de l'ensemble.

```
import turtle

def carre(cote) :
    ...

def triangle(cote) :
    ...

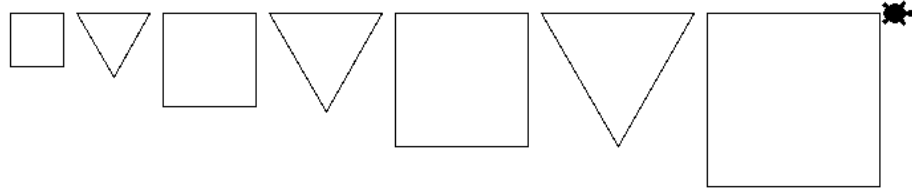
# programme principal : trace ligne carres
# et triangles
...
```

### 3.3.3 Pour aller plus loin

Si vous avez fini l'ensemble de ce qui précède, vous pouvez essayer de réaliser les figures des exercices suivants.

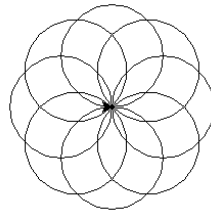
#### Ligne de carrés et triangles alternés

Tracer une ligne de carrés et triangles alternés pour laquelle le premier carré a un côté de 40. La taille augmente de 15 à chaque nouvelle figure tracée.

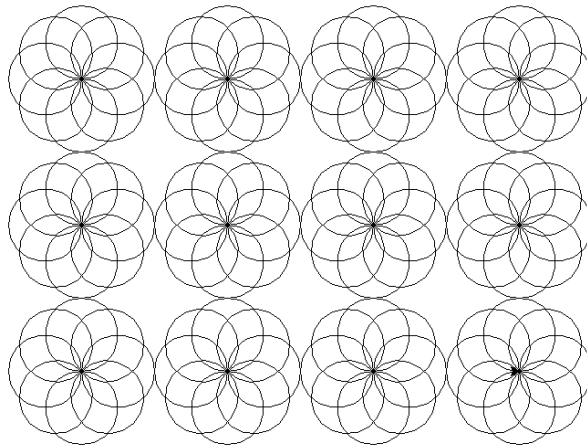


### Rosace et pavage de rosaces

1. Définir une fonction `rosace(x, y, nb, r)` qui trace une rosace de centre le point de coordonnées  $(x, y)$ , composée de `nb` cercles chacun de rayon `r`. Attention : le centre de la rosace est l'endroit où on voit le curseur sur la figure ci-dessous pour une rosace à 8 cercles :



2. Utilisez cette fonction pour écrire une fonction qui trace une ligne de rosaces.
3. Utilisez cette fonction (ligne) pour écrire une fonction qui trace un pavage de rosaces comme ci-dessous :



Si vous n'arrivez pas à définir la fonction, vous pouvez quand même essayer de faire un pavage de rosaces sans utiliser de fonction.

### Bonus : labyrinthe

Créer une fonction qui trace un gros rectangle noir de la taille de la fenêtre, puis fait se déplacer la tortue au hasard en traçant un chemin avec des rectangles blancs pour former un labyrinthe.

### 3.4 TP 4 : création et manipulation de fonctions, basiques

Des fichiers sont à télécharger sur CaseIne : <http://caseine.org/mod/resource/view.php?id=3296>

#### 3.4.1 Exercices d'observation

##### Décrire les appels de fonctions

Tapez et lancez les programmes suivants dans Python Tutor, et comptez combien de fois chacune des fonctions est appelée. Pour chaque appel, décrire quelles sont les valeurs des arguments et la valeur de retour. (Pensez à sauvegarder vos programmes dans un fichier texte avant de les copier-coller dans Python Tutor.)

```
# Programme 1
def est_solution(x,a,b,c):
    # Renvoie True si x est solution de
    #  $ax^2+bx+c=0$ 
    y=a*x**2+b*x+c
    if y==0:
        rep=True
    else:
        rep=False
    return rep

# prog. principal
rep=est_solution(1,1,-2,1)
print(rep)
print(est_solution(5, 2, -20, 50))
print(est_solution(2.5, 1, 2, 3))

# Programme 2
def distance(xA,yA,xB,yB):
    # renvoie la distance entre (xA,yA) et (xB,yB)
    d=(xB-xA)**2+(yB-yA)**2
    d=d**(1/2)
    return d

def appartient_cercle(xA,yA,rayon):
    # teste si le point (xA,yA) appartient
    # au cercle de rayon r centré à l'origine
    if distance(0,0,xA,yA)==rayon:
        return True
    else:
        return False

# prog. principal
d=distance(1,2,2,1)
print(d)
rep=appartient_cercle(1,1,2)
print(rep)
print(appartient_cercle(1,0,1))
```

##### Reconnaître les variables locales

Tapez le programme suivant dans Python Tutor et observez son fonctionnement. Quelles sont les variables locales de la fonction `est_premier` ?

```
def est_premier(N):
    i=2
    a_diviseur=False
    while i<N and (not a_diviseur):
        if N%i==0:
            a_diviseur=True
        i=i+1
    return not a_diviseur

# prog. principal
if __name__=="__main__":
    rep=est_premier(9)
    print("9 est premier ?", rep)
    print("5 est premier ?", est_premier(5))
```

`locals()`

Testez le programme suivant dans IDLE et observez attentivement ce qui s'affiche. Que renvoie l'appel à la fonction `locals()` ?

```
def pente(xA, yA, xB, yB):
    p=(yB-yA)/(xB-xA)
    print("Variables locales de pente :", locals())
    return p

# prog. principal
p1=pente(1,1,2,2)
print("Pente 1: ", p1)
p2=pente(0,2,1,4.5)
print("Pente 2: ", p2)
```

`locals()` appelée dans une fonction renvoie un dictionnaire (voir plus tard le cours sur les dictionnaires) contenant les variables locales de cette fonction et leurs valeurs (si définies avant l'appel à `locals()`). Python



Tutor permet aussi d'observer les variables locales d'une fonction et leurs valeurs, qui apparaissent dans un rectangle séparé (*frame*) des variables du programme principal (*global frame*).

### Incrémentation

Prédire le fonctionnement des deux programmes suivants, et vérifier votre réponse en les lançant dans Python Tutor pas à pas.

<pre># Programme 1 def incremente (a):     a = a+1  # prog. principal if __name__=="__main__":     b=3     incremente(b)     print (b)     a=5     incremente(a)     print (a)</pre>	<pre># Programme 2 def incremente2(a):     return a+1  # prog. principal if __name__=="__main__":     b=1     b = incremente2(b)     print(b)     a=incremente2(b)     print(a)</pre>
--	---

Maintenant, ajoutez l'instruction `a=incremente(a)` à la fin du programme 1 et re-testez dans Python Tutor. Que vaut `a` à la fin du programme ? Traduisez en français et cherchez une explication. Nous expliquerons plus en détails ce que cela veut dire en cours la semaine prochaine.

### 3.4.2 Des petites fonctions

On se place maintenant dans Idle, dans un nouveau fichier. Vous prendrez bien soin de tester vos fonctions au fur et à mesure, soit en faisant des appels dans votre programme principal, soit directement dans l'interpréteur après avoir fait "Run Module" (testez chacune des deux solutions au moins une fois).

#### La bosse des maths

1. Écrire une fonction `valeur_abs` qui prend en argument un nombre  $x$  et qui renvoie sa valeur absolue.
2. Écrire une fonction `signe_différent` qui prend en arguments deux nombres  $x$  et  $y$  et qui renvoie `True` si  $x$  et  $y$  ont des signes différents, `False` sinon (si  $x$  ou  $y$  est égal à 0, la fonction doit renvoyer `False`).
3. Soit  $f$  la fonction mathématique définie sur  $\mathbb{R}$  par  $f(x) = 3x^2 + 2x + 3$ . Écrire une fonction Python `f` qui prend en argument un réel  $x$  et qui renvoie la valeur de  $f(x)$ .
4. Écrire une fonction `nb_racines` qui prend en argument trois réels  $a, b, c$  et qui renvoie le nombre de racines réelles du polynôme  $ax^2 + bx + c$ . On rappelle la formule du discriminant :  $\Delta = b^2 - 4ac$ .

#### Le journal de M. Bizarre

M. Bizarre est rédacteur-en-chef d'un journal, et a souvent des idées farfelues. Suivant les jours de la semaine, il souhaite que les articles de son journal suivent des règles qu'il a inventées.

1. Chaque lundi, il souhaite que les mots soient répétés deux fois, séparés par un espace. Écrivez une fonction `lundi` qui prend en argument un mot et qui transforme ce mot selon la règle du lundi. Par exemple, `lundi("bonjour")` doit valoir `"bonjour bonjour"`.
2. Chaque mardi, il faut que les mots de longueur paire soit répétés 6 fois, en séparant par un tiret, alors que les mots de longueur impaire doivent être répétés 3 fois, en séparant par des virgules. Écrire la fonction `mardi` qui prend en argument un mot et qui renvoie la chaîne de caractères suivant la règle du mardi. *Rappel : la longueur d'une chaîne de caractères est donnée par `len(chaine)`.*
3. Chaque mercredi, il faut que les mots de longueur impaire soit remplacés par le mot `"impair"`. Les mots pairs doivent rester tels qu'ils sont. Écrire la fonction `mercredi` (qui prend en argument un mot et qui renvoie la bonne chaîne de caractères).

4. Chaque jeudi, il faut que les mots soient répétés autant de fois que leur longueur modulo 3 (à la suite, sans espace). Écrire la fonction `jeudi`. Par exemple, `jeudi("merci")` vaut `"mercimerci"`, `jeudi("bonbon")` vaut `"`, `jeudi("comment")` vaut `"comment"`.
5. Le vendredi, il faut que les mots soient écrits normalement. Et heureusement, il n'y a pas de journal le week-end.
6. Écrire une fonction `transforme(mot, num_jour)` qui prend en argument un mot et le numéro du jour (1 pour lundi, 2 pour mardi, etc....) et qui renvoie le mot transformé selon la règle du jour correspondant.

### Des suites

Codez les suites de Fibonacci et de Syracuse (voir TD exercices 2.3.13 et 2.3.14).

## 3.5 TP5 : Manipulation de fonctions (suite), dessin avec Turtle

### 3.5.1 Retour aux fonctions numériques

#### La banque

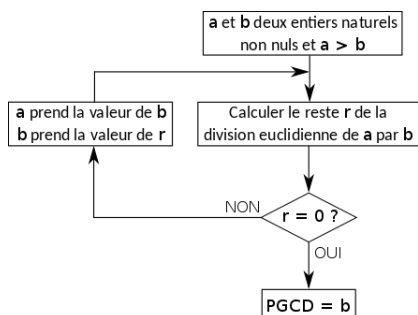
Aline envisage d'ouvrir un compte à la banque Argento, mais elle veut d'abord savoir si cela sera rentable. Sur un tel compte, les intérêts sont de 5% par an, et la banque prélève un coût fixe annuel de 11 euros. Le capital de l'année  $n + 1$  est donc obtenu par la formule  $u_{n+1} = u_n \times 1.05 - 11$ , où  $u_n$  désigne le capital à l'année  $n$ .

1. Écrire une fonction `capital(nb_annees, capital_debut)` qui renvoie le capital en euros qu'Aline aurait sur un tel compte au bout de `nb_annees` en plaçant initialement un capital égal à `capital_debut` (en euros).
2. Écrire une fonction `gagne_argent(nb_annees, capital_debut)` qui renvoie `True` si le capital au bout de `nb_annees` sur un tel compte est supérieur ou égal au capital de début.
3. Écrire une fonction `placement_min(nb_annees, but)` qui calcule le placement minimum nécessaire pour atteindre au moins le capital `but` après `nb_annees` d'économies.
4. Écrire une fonction `duree_min(capital, but)` qui calcule la durée minimum de placement avec un capital de départ donné pour atteindre le capital `but` souhaité. **Attention** : si le capital initial est trop petit, les frais annuels vont coûter plus cher qu'il ne rapporte, et il va diminuer au fil du temps, ce qui risque de causer une boucle infinie. Trouver la valeur minimale du capital nécessaire pour qu'il augmente, et gérer ce cas particulier en affichant un message d'erreur au lieu de rentrer dans la boucle.

#### Un peu d'arithmétique

Dans cet exercice, on ne considérera que des entiers strictement positifs.

1. Écrire une fonction `plus_grand_diviseur_premier(n)` qui renvoie le plus grand diviseur premier de l'entier `n`. Il vous est conseillé de commencer par redéfinir la fonction `est_premier(N)` que l'on a déjà vue.
2. Écrire une fonction `pgcd(a,b)` qui renvoie le plus grand commun diviseur des entiers  $a$  et  $b$ . *Note* : il existe principalement deux algorithmes pour calculer un pgcd, vous pouvez essayer les deux :
  - L'algorithme "naïf" qui parcourt tous les entiers qui sont candidats à être pgcd.
  - L'algorithme d'Euclide, qui est beaucoup plus rapide : voir schéma ci-contre.
  - Bonus : comparer le temps de calcul et le nombre de comparaisons faites par chaque algorithme. Si  $a < b$ , l'algorithme commence par échanger  $a$  et  $b$  puis fonctionne normalement suivant le schéma ; si  $a = b$ , l'algorithme renvoie simplement  $a$ .
3. Écrire une fonction `ppcm(a,b)` qui renvoie le plus petit commun multiple de  $a$  et  $b$ .
4. Écrire une fonction `irreductible(numerateur, denominateur)` qui renvoie `True` si la fraction correspondante `numerateur/denominateur` est irréductible, `False` sinon.



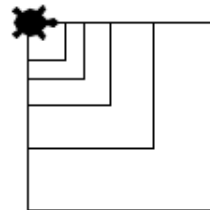
### 3.5.2 Turtle

Dans cette partie, nous allons utiliser la bibliothèque graphique Turtle. N'oubliez pas de l'importer. Vous testerez bien toutes vos fonctions au fur et à mesure, et vous garderez le même fichier pour tous les exercices de cette partie (les exercices sont dépendants les uns des autres, dans l'ordre).

*Note* : À partir de maintenant, nous utilisons l'expression "Écrire une fonction `nom_fonction(argument1, argument2, ...)`" pour dire "Écrire une fonction `nom_fonction` qui prend comme arguments `argument1, argument2, ...`".

#### Carrés imbriqués

1. Écrire une fonction `carre(cote)` qui trace un carré dont la longueur du côté est `cote`, et dont le point de départ et d'arrivée est le coin supérieur gauche (comme la semaine dernière).
2. Écrire une fonction `carres_imbriques(cote_debut, nb_carres)` qui trace des carrés imbriqués comme sur la figure ci-contre : le plus grand carré a pour côté `cote_debut`, la taille du carré est multipliée par  $2/3$  à chaque étape, et le nombre de carrés tracés est `nb_carres`.



`carres_imbriques(100,5)`

#### Se déplacer

1. Écrire une fonction `aller_sans_tracer(x,y)` qui déplace le curseur au point de coordonnées `x,y` sans tracer.
2. Écrire une fonction `descendre_sans_tracer(longueur)` qui descend de `longueur` par rapport à la position actuelle, sans tracer. On supposera qu'au moment de l'appel, le curseur est dirigé vers la droite, et on prendra soin de le remettre dans la même position à la fin.
3. Bonus : écrire une fonction qui fait se promener la tortue au hasard : distance aléatoire, angle aléatoire. Elle pourra laisser des empreintes à chaque pas avec `stamps()` ; dans ce cas on pourra aussi choisir une couleur aléatoire. Enfin on pourra faire en sorte pour ne pas la perdre de vue que quand elle sort de la fenêtre d'un côté, elle y re-rentre aussitôt de l'autre côté (environnement torique).
4. Bonus : écrire une fonction de tracé qui reçoit la couleur en paramètre optionnel (par défaut noir).

#### Polygone

1. Écrire une fonction `polygone(nb_cotes, cote)` qui trace un polygone régulier à `nb_cotes` côtés (triangle, carré, pentagone, hexagone....) et dont la longueur d'un côté est `cote`. Le premier côté tracé doit être horizontal et vous devez tourner dans le sens des aiguilles d'une montre (voir figure ci-contre).
2. Écrire une fonction `diametre_polygone(nb_cotes, cote)` qui renvoie le diamètre d'un polygone régulier à `nb_cotes` de longueur `cote`. Ce diamètre est donné par la formule suivante, avec  $n$  de nombre de côtés et  $c$  la longueur du côté (n'oubliez pas d'importer le module `math`) :

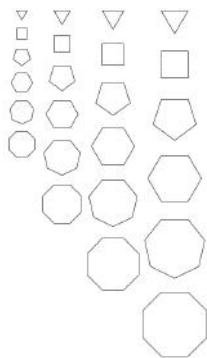


`polygone(6,50)`

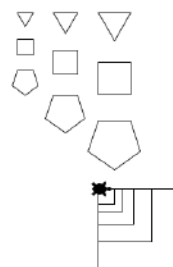
$$\frac{c}{\sin(\pi/n)}$$

### Pavage de polygones

1. Écrire une fonction `colonne_polygone(nb_poly, cote)` qui trace `nb_poly` polygones sur la même colonne, en commençant par un triangle et en augmentant le nombre de côtés de 1 à chaque fois. La longueur du côté des polygones doit toujours être égale à `cote`. Vous vous servirez des fonctions `descendre_sans_tracer` et `diametre_polygone` pour décaler votre curseur de  $d + 5$  vers le bas entre deux polygones successifs, où  $d$  désigne le diamètre du polygone qui vient d'être tracé.
2. Écrire une fonction `pavage(nb_poly, nb_col, cote)` qui dessine un pavage de polygones comme sur la figure ci-contre, en commençant au point de coordonnées  $(-270, 330)$ . Le nombre de polygones par colonne doit être réglé par `nb_poly` et le nombre de colonnes par `nb_col`. De plus, le côté des polygones doit être de `cote` pour la première colonne puis augmenter de 10 lorsque l'on change de colonne. Les points de départ de chaque colonne doivent avoir la même ordonnée (les triangles du haut sont alignés), et leurs abscisses doivent être écartées de  $d + 10$  (vers la droite), où  $d$  est le diamètre du dernier polygone tracé. Pensez à utiliser les fonctions créées précédemment. *Si vous avez du mal à écrire cette fonction, vous pouvez commencer à écrire une fonction `pavage5()` qui ne prend aucun argument, et qui dessine un pavage avec 5 lignes et 5 colonnes, en commençant avec un côté égal à 20.*
3. Écrire un programme principal qui dessine la figure ci-contre, contenant un pavage à 3 colonnes avec 3 polygones par colonne, commençant avec un côté de 20; puis qui dessine 5 carrés imbriqués, dont le premier a pour longueur 100.



pavage(6,4,20)



prog. principal

### 3.5.3 On travaille en ligne !

Quand vous aurez fini les exercices précédemment proposés, vous pouvez travailler en ligne sur le site du PLM : <https://plm.telecomnancy.univ-lorraine.fr/#/>. PLM signifie en anglais "The Programmer's Learning Machine", c'est une plateforme en ligne avec des exercices de programmation.

Pas 1 : Il faut d'abord choisir Python avant de commencer les exercices. Sur le coin haut à droite de votre écran, changez Java en Python.

Pas 2 : Cliquez sur le bouton en haut à gauche et vous aurez la liste de sujets proposés. Vous pouvez commencer par les exercices correspondant à de "boucles tant que" (while) et "fonctions".

## 3.6 TP6 : listes

### 3.6.1 Consultation de listes

Dans cette partie, nous allons d'abord nous concentrer sur l'utilisation de listes existantes. Nous verrons dans la deuxième partie du TP comment créer ou modifier des listes..

*Note* : Les codes python présentés ici sont dans une archive téléchargeable sur Caseine : <http://caseine.org/mod/resource/view.php?id=3326>. Récupérez la et décompressez la dans votre répertoire de TP avant de commencer.

#### Listes sous PythonTutor

Essayez de prévoir le comportement des deux exemples suivants puis testez-les pas à pas sur [www.PythonTutor.com](http://www.PythonTutor.com). (Correspondent aux fichiers `pythontutor1.py` et `pythontutor2.py` de l'archive.)

```
maliste = [1, 3, 9, 6, 4]
print(maliste)

a = maliste[2]
b = maliste[0]
c = maliste[4]
d = maliste[-1]
e = maliste[7]
```

```
maliste = [1, 3, 9, 6, 4]
print(maliste)

i = 0
while i < len(maliste):
    element = maliste[i]
    print("indice", i, " contient: ", element)
    i = i+1
```

Regardez également le fichier nommé `intro-listes.py`. Vous devez être en mesure de comprendre tout ce qui y est fait. Testez le sous Idle. Vous pouvez le modifier à votre guise pour vérifier votre compréhension.

```
#!/usr/bin/env python3

liste_desord = [2, 12, 7, 9, 3, 4]
liste_croiss = [2, 3, 4, 5, 7, 9, 12]
```

```
def parcours (liste):
    i = 0
    while i < len(liste):
        print ("Element a l'indice", i, ":", liste[i])
        i = i+1
```

```
def croissante (liste):
    cr = True
    i = 1
    while i < len(liste):
        cr = cr and (liste[i-1] <= liste[i])
        i = i+1
    return cr
```

```
parcours(liste_desord)
parcours(liste_croiss)
```

```
def parcours_envers (liste):
    i = len(liste)-1
    while i >= 0:
        print ("Element a l'indice", i, "(envers) :", liste[i])
        i = i-1
```

```
print (liste_desord, "est croissante:", croissante(liste_desord))
print (liste_croiss, "est croissante:", croissante(liste_croiss))
```

#### Recherche dans une liste

Dans cet exercice, vous allez devoir écrire plusieurs fonctions “de base” fonctionnant sur des listes d’entiers. Essayez d’avoir un code propre définissant d’abord les fonctions, puis qui teste sur plusieurs exemples vos fonctions dans un programme principal (“main”).

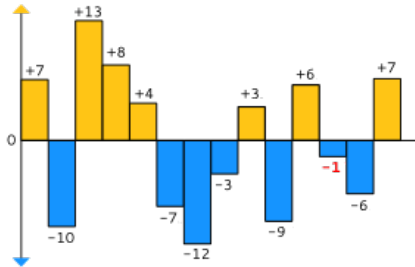
1. Écrivez une fonction `minimum(liste)` qui renvoie l’élément le plus petit de `liste`;

- Écrivez une fonction `contient(liste, elem)` qui renvoie `True` si `liste` contient `elem` et `False` sinon. Pensez à tester les cas limites (p.ex. élément au début ou à la fin).
- Écrivez une fonction `minimum2(liste)` qui renvoie le deuxième élément le plus petit de `liste` (i.e., celui juste au-dessus du minimum);  
*Suggestion* : définissez une fonction auxiliaire `minimum_plus_grand_que(liste, petit)`.

### Températures

*Note* : cet exercice est extrait du site *CodinGame*.

Dans cet exercice, vous devez analyser des enregistrements de température pour trouver la plus proche de zéro.



Dans l'exemple, -1 est la plus proche de 0.

Vous devez écrire une fonction `proche_zero` qui prend en argument une liste de températures (entiers) et renvoie la plus proche de zéro. S'il n'y a aucune température, renvoyer 0. En cas d'égalité entre une température positive et une négative, on considère que c'est la positive qui est la plus proche de zéro (par exemple, si les températures sont 5 et -5, il faut renvoyer 5).

Le fichier `temperature.py` contient un squelette de code prêt à tester votre fonction (note : la fonction `assert` vérifie si son argument est vrai et fait échouer le programme sinon).

### Duels de chevaux de course

*Note* : cet exercice est également extrait du site *CodinGame*. Il est plus complexe que tous les autres exercices vus jusqu'à présent : il est donc nécessaire de prendre un moment pour bien comprendre le problème et réfléchir posément à comment vous allez le résoudre.

À l'hippodrome de Casablanca, on organise des courses de chevaux d'un type particulier : les duels ! Durant un duel, seuls deux chevaux participent à la course. Pour que la course soit intéressante, il faut sélectionner deux chevaux de force similaire. On considère que l'on connaît la "force" des chevaux, donnée par un entier.

Écrire une fonction `plus_proches(liste)` qui, étant donnée une `liste` des forces des chevaux, identifie les forces les plus proches et renvoie la différence entre ces deux forces (un entier positif ou nul).

Le fichier `course.py` contient un squelette de code prêt à tester votre fonction. Vous êtes encouragés à écrire d'autres fonctions auxiliaires si vous le jugez nécessaire.

### 3.6.2 Avec modifications !

Vous allez maintenant modifier des listes, soit directement des éléments (valeurs contenues dans la liste), soit par ajout/suppression d'éléments.

#### Familiarisation

- Essayez vous à modifier les éléments d'une liste dans l'interpréteur Python : que fait `maliste[5] = 12` ? Que se passe-t'il si la liste n'existe pas ? Et si elle est de longueur trop petite ? etc.
- Écrivez un programme qui demande des nombres strictement positifs à l'utilisateur et les stocke dans une liste. La saisie s'arrête dès que l'utilisateur entre "0" (zéro). Affichez ensuite la moyenne de cette liste.

- Écrivez un programme qui inverse l'ordre des éléments d'une liste. Par exemple [1, 3, 9, 6, 4] doit devenir [4, 6, 9, 3, 1].

*Note : il y a plusieurs manières de résoudre ce problème. Vous êtes libres de garder une seule liste du début à la fin, ou de créer des listes temporaires, d'ajouter, supprimer des valeurs, etc.*

### Les mutables : encore du PythonTutor !

Les programmes suivants permettent d'illustrer le comportement "étrange" des listes par rapport aux variables que vous avez l'habitude de manipuler. Il est essentiel à cette étape de bien comprendre la différence de comportement entre données "mutables" (notamment les listes) et "non-mutables" (entiers, flottants, chaînes de caractères...) en Python.

C'est le moment de tester ces programmes sous PythonTutor (ils sont dans l'ordre croissant de difficulté). **Observez bien les "flèches" de PythonTutor qui montrent à quels objets les variables se réfèrent.** Si vous ne comprenez pas le comportement d'un des programmes, faites appel à votre chargée de TP.

```

pythontutor3.py
maliste = [1, 3, 9, 6, 4]

autre = maliste

autre[2] = 42

print (maliste)
print (autre)

pythontutor4.py
def incremente(i):
    i = i+1

def incr_liste(liste):
    i = 0
    while i < len(liste):
        liste[i] = liste[i] + 1
        i = i+1

x = 12
maliste = [1, 3, 9, 6, 4]

incremente(x)
incr_liste(maliste)

print(x)
print(maliste)

pythontutor5.py
maliste = [[1], [3], [9], [6], [4]]

print(maliste)

i = 0
while i < len(maliste):
    element = maliste[i]
    print ("indice", i, " contient: ", element)
    i = i+1

print("Avant:", maliste)
maliste[3]=maliste[2]
maliste[2].append(0)
print("Après:", maliste)

```

### 3.6.3 Exercices sur les fonctions et chaînes de caractères

#### Fonctions de chiffrement

**Position d'une lettre** Écrire une fonction qui reçoit une lettre et renvoie sa position dans l'alphabet. Par exemple la position de 'a' est 1, la position de 'E' est 5.

**Chiffrement d'un mot** Écrire une fonction qui reçoit un mot et renvoie ce mot codé en remplaçant chaque lettre par sa position dans l'alphabet. Les nombres doivent être séparées par des +.



Par exemple "bonjour" est codé comme "2+15+14+10+15+21+18"

**Chiffrement d'un texte** Écrire une fonction qui reçoit un texte et renvoie ce texte codé comme précédemment. Les espaces et autres ponctuations dans le texte ne sont pas modifiés. Attention, on s'abstiendra d'utiliser des lettres accentuées. Par exemple "bonjour a tous !" est codé en "2+15+14+10+15+21+18 1 20+15+21+19!"

**Programme principal** Écrire un programme principal qui opère en boucle :

- demande à l'utilisateur un texte,
- affiche ce texte codé comme ci-dessus,
- puis propose à l'utilisateur de rejouer avec un nouveau texte.

Le programme s'arrête quand l'utilisateur refuse de rejouer.

### Fonctions de comptage et statistiques

**Taille des mots** Écrire une fonction qui lit un texte et renvoie une liste d'entiers contenant la taille de chacun de ses mots.

**Indices** : la fonction `split` permet de séparer une chaîne de caractères (ici le texte) autour d'un séparateur donné (ici l'espace). La fonction `list()` convertit une chaîne de caractères en la liste de ses caractères. La fonction `len()` donne la taille d'une liste.

**Mot le plus long** Écrire une fonction qui lit un texte et renvoie son mot le plus long.

**Mots plus longs que la moyenne** Écrire une fonction qui lit un texte et affiche tous les mots qui sont plus longs que la longueur moyenne de ses mots.

**Indices** : le module `statistics` fournit une fonction `mean` qui fait la moyenne des éléments d'une liste.

## 3.7 TP7 : retour sur les listes

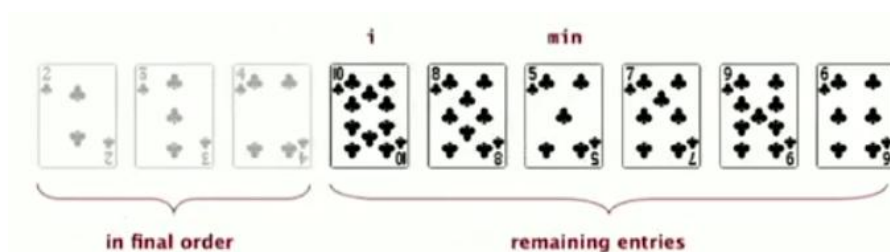
### 3.7.1 Recyclage et Tri sélectif

Reprenez l'exercice de tri vu en TD et écrivez une fonction `tri_insertion` qui trie une liste d'entiers par insertion successive dans une liste triée. C'est le principe en général appliqué lorsqu'on trie sa "main" dans un jeu de cartes.



Source : <http://staff.ustc.edu.cn/>

Un autre façon possible de trier une liste est de prendre le minimum et de le mettre au début, puis de recommencer avec le deuxième plus petit, etc.



Source : <http://x-wei.github.io/>

Écrivez une fonction `tri_selection` qui trie une liste d'entier par sélections successives du plus petit élément de la partie non triée de la liste.

### 3.7.2 Le retour des courses (difficile)

Reprenez le même exercice que pour les duels de chevaux de courses (TP précédent). Cette fois, vous devez apparier les  $n$  chevaux existants pour former  $n/2$  courses intéressantes.

1. Faites une fonction qui étant donnée une liste des forces de chevaux, renvoie une liste des couples telle que la différence des forces entre chaque couple soit la plus petite possible. *Indices : Encore une fois, cet exercice nécessite de bien comprendre et modéliser le problème avant de se lancer dans l'écriture du code. Il est très difficile de le résoudre d'un seul bloc, il est donc fortement recommandé de "découper" le problème en sous-tâches plus simple, et de faire autant de fonctions auxiliaires que nécessaire.*
2. (Vraiment difficile) Avez vous vu que l'ordre dans lequel les paires étaient faites était important ? On peut parfois avoir des courses peu intéressantes dans un ordre alors qu'un autre serait meilleur. Par exemple, pour les 4 chevaux suivants [12, 13, 11, 15], si on commence par faire (12,13), il nous reste (11,15) qui n'est pas très intéressante. Alors que le mieux aurait été (11, 12) et (13, 15). Avez-vous des idées pour améliorer votre algorithme ?

### 3.7.3 Course à pied

#### Initialisation

Écrire une fonction qui crée une liste de 365 éléments correspondant à la distance courue chaque jour de l'année passée par un coureur. Les éléments sont initialisés au hasard entre 0 et 100, avec 2 chiffres après la virgule.

**Indices** : le module random fournit une fonction `uniform(a,b)` qui tire un nombre réel au hasard entre les bornes a et b incluses. La fonction `round(nb,x)` arrondit le nombre nb avec x chiffres après la virgule.

**Premier test** : écrire un programme principal qui appelle cette fonction et affiche la liste obtenue.

### Moyenne par jour de semaine

Écrire une fonction qui reçoit une liste de distances, et un numéro de jour de la semaine (1 pour lundi, 7 pour dimanche). Cette fonction calcule alors la moyenne de kilomètres effectués ce jour de la semaine sur toute l'année. (*Par exemple si l'entier reçu est 2, il faut calculer la moyenne des distances de tous les mardis de l'année.*) **Remarque** : pour simplifier, on considère que l'année commence un lundi.

Écrire ensuite une fonction qui reçoit la liste de distances, calcule la moyenne pour chacun des 7 jours de la semaine, et renvoie une liste de 7 valeurs moyennes.

### Histogramme turtle

Écrire une fonction qui utilise le module turtle pour tracer un histogramme des distances moyennes des 7 jours de la semaine.

N'oubliez pas de tester vos fonctions au fur et à mesure en les appelant dans votre programme principal.

## 3.7.4 Graphiques scientifiques

### Le module matplotlib et pyplot

Le module `matplotlib.pyplot` permet de tracer des courbes facilement. Pour l'utiliser on ajoutera au début du fichier la ligne suivante : `import matplotlib.pyplot as plt`

On pourra ensuite appeler les fonctions de ce module en les préfixant par `plt`. Par exemple :

- la fonction `plt.plot()` reçoit la liste des valeurs de x, puis la liste des valeurs de f(x), et trace la courbe de la fonction f.
- la fonction `plt.show()` affiche le graphique, il faut l'appeler pour voir la fenêtre graphique
- les fonctions `plt.xlabel` et `plt.ylabel` permettent d'ajouter un nom aux 2 axes
- la fonction `plt.title` permet de changer le titre de la fenêtre
- la fonction `plt.axis()` permet de spécifier les bornes des 2 axes

Vous pouvez trouver la liste des fonctions de pyplot sur [https://matplotlib.org/api/pyplot\\_summary.html](https://matplotlib.org/api/pyplot_summary.html)

### Graphique de distance

Utiliser ce module et les fonctions de l'exercice précédent pour afficher sur un même graphique les moyennes de distance par jour de 3 coureurs différents. On pourra changer la couleur, le tracé, et les marqueurs des lignes tracées, en rajoutant une chaîne de caractères en 3e argument de la fonction `plot`.

- Tracés possibles : - ou – ou -. ou :
- Couleurs possibles : b, g, r, c, m, y, k, w
- Marqueurs possibles : o + . x \* ^

Par exemple en ajoutant la chaîne "r-", on trace une ligne pointillée rouge.

### Module numpy

Pour tracer des fonctions plus complexes, on va aussi utiliser le module numpy. Rajouter en haut de votre fichier la ligne : `import numpy as np`

Le module numpy fournit en particulier une fonction `linspace` qui crée une liste de valeurs entre les 2 bornes données, contenant un nombre de valeurs égal au 3e paramètre. Cette fonction va nous servir à préparer l'argument de la fonction `plot`. Plus le 3e argument est grand (plus de valeurs dans la liste), plus la courbe est lissée.

Par exemple pour tracer la fonction cosinus on procède comme suit :

```
x = np.linspace(0, 2*np.pi, 300)
y = np.cos(x)
plt.plot(x, y)
plt.show()
```

Voir aussi la documentation :

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html>

et un cours sur : <http://www.courspython.com/introduction-courbes.html>

### Polynôme du second degré

Écrire une fonction qui reçoit en paramètres des valeurs a,b,c et affiche la courbe correspondant au polynôme :  $a * x^2 + b * x + c$

## 3.8 TP semaine 8 : propagation d'une nouvelle

### 3.8.1 Contexte du problème

On veut modéliser et simuler la propagation d'une nouvelle dans une population donnée.

On considère une population de  $N$  personnes. Pour simplifier, chaque personne est représentée par un numéro unique compris dans l'intervalle  $[1; N]$ . Initialement, seule la personne numéro 1 est au courant de la nouvelle.

Chaque jour a lieu une rencontre aléatoire entre 2 personnes  $X$  et  $Y$  de la population considérée. La nouvelle est propagée différemment d'une personne à l'autre selon leur "statut" respectif relativement à la nouvelle.

Il y a 3 statuts possibles pour une personne :

- **C** : la personne **C**onnaît la nouvelle et la raconte aux personnes qu'elle rencontre
- **I** : la personne **I**gnore la nouvelle et ne peut donc pas la raconter
- **M** : la personne, dite "**M**uette", connaît la nouvelle, mais pense que tout le monde la connaît ; elle ne la raconte donc plus aux personnes rencontrées.

Une rencontre entre deux personnes de n<sup>os</sup>  $X$  et  $Y$  est représentée par le couple  $(X, Y)$ .

Une rencontre  $(X, Y)$  a pour effet un éventuel changement de statut de  $X$  et de  $Y$ , changement qui obéit aux règles de bon sens résumées dans le tableau ci-dessous.

*Remarque* : dans les rencontres du type  $(C, M)$  ou  $(M, C)$ ,  $C$  raconte la nouvelle à son interlocuteur  $M$  qui répond : "je la connaissais", de sorte que  $C$  en est "vexé" et devient  $M$ . De même, dans une rencontre du type  $(C, C)$  chaque  $C$  devient  $M$ .

$X \backslash Y$	C	I	M
C	M, M	C, C	M, M
I	C, C	I, I	I, M
M	M, M	M, I	M, M

TABLE 3.1 – Evolution des statuts à la suite d'une rencontre

On voit que :

1. Le statut d'une personne ne peut évoluer que dans le sens :  $I \rightarrow C \rightarrow M$
2. Lors d'une rencontre, il n'y a aucun changement de statut pour les deux personnes lorsqu'aucune des deux n'a le statut  $C$  avant la rencontre

### 3.8.2 Exercice préliminaire

#### Travail "manuel" de compréhension sur un exemple :

On considère une population de 5 personnes ( $N = 5$ ) et la suite de 9 rencontres :

$$\{(3, 5), (2, 4), (4, 1), (3, 2), (1, 5), (4, 5), (3, 4), (5, 4), (5, 1)\}$$

On considère aussi que, initialement, seule la personne numéro 1 connaît la nouvelle, et qu'une rencontre et une seule a lieu chaque jour à partir du moment où la personne numéro 1 a appris la nouvelle.

Dessinez à la main un tableau représentant l'évolution des statuts des 5 personnes après chacune des 9 rencontres consécutives considérées (utilisez le modèle suivant) :

jour	rencontre	statuts initiaux	statuts après la rencontre				
			pers.1	pers.2	pers.3	pers.4	pers.5
0			C	I	I	I	I
1	3, 5	I, I	...	...	...	...	...
...	...	...	...	...	...	...	...
9	...	...	...	...	...	...	...

Vérifiez qu'à l'issue des 9 rencontres, les statuts des personnes numéro 1 à 5 sont respectivement :  $M, I, I, M, M$ , et qu'ils sont stables (c'est à dire qu'aucun d'eux ne peut changer lors d'une nouvelle rencontre).

### 3.8.3 Exercice 1

#### Simulation d'une suite de rencontres

Il vous est à présent demandé d'écrire un programme `propagation.py` qui, à partir d'un nombre de personnes  $N$  saisi au clavier, effectuera une simulation de la propagation d'une nouvelle parmi cette population de personnes (numérotées de 1 à  $N$ ).

Chaque étape de la simulation (appelée "jour") correspondra à une rencontre entre 2 personnes. Chaque rencontre sera obtenue par un tirage au hasard des numéros des deux personnes entrant en contact l'une avec l'autre. Il va de soi que deux personnes qui se rencontrent sont nécessairement distinctes l'une de l'autre.

Dans l'état initial, la personne numéro 1 connaît la nouvelle (son statut est  $C$ ) dont on veut simuler la propagation. Les personnes n<sup>os</sup> 2 à  $N$  ignorent cette nouvelle (leur statut est  $I$ ).

Les statuts des personnes (des caractères) seront mémorisés dans une liste. Attention : la case de numéro  $X$  ( $0 \leq X \leq N - 1$ ) de la liste `personnes` contiendra le statut ('C', 'I' ou 'M') de la personne numéro  $X + 1$ .

Le tirage aléatoire d'une rencontre est obtenu au moyen de la fonction `random.randint()`. Pour chaque rencontre, il s'agit de tirer au hasard 2 numéros de personnes  $X$  et  $Y$ , tels que  $1 \leq X \leq N$ ,  $1 \leq Y \leq N$  et  $X \neq Y$ .

Le traitement d'une rencontre consistera à déterminer si le statut de l'une ou l'autre des deux personnes est modifié par la rencontre, compte tenu du statut que chacune d'elle avait avant la rencontre. Il faudra pour cela utiliser la table 3.1 présentée en début d'énoncé (page 149). Dans l'affirmative il faudra mémoriser dans le tableau `personnes` cette (ou ces) modification(s) de statut.

La simulation doit s'arrêter dès que plus aucune propagation n'est possible, c'est à dire lorsque plus aucune des  $N$  personnes n'a le statut  $C$ .

#### Exemple d'affichage que devra produire le programme :

```

jour rencontre nature st.1 st.2 st.3 st.4 st.5
0          C      I      I      I      I
1    3, 4    I, I    -      -      -      -      -
2    2, 5    I, I    -      -      -      -      -
3    1, 2    C, I    -      C      -      -      -
4    3, 2    I, C    -      -      C      -      -
5    1, 3    C, C    M      -      M      -      -
6    3, 1    M, M    -      -      -      -      -
7    5, 1    I, M    -      -      -      -      -
8    3, 5    M, I    -      -      -      -      -
9    4, 2    I, C    -      -      -      C      -
10   4, 2    C, C    M      M      M      M      I
Nombre d'ignorants : 1
Nombre de muets : 4

```

### 3.8.4 Exercice 2

On veut maintenant que le programme n'affiche plus que le résultat final de la simulation, c'est à dire le nombres de jours  $NJ$  qu'a duré la propagation de la nouvelle et le nombre d'ignorants  $NI$  restants à la fin de la simulation. Créez une copie du programme précédent appelée `propagation2.py`. Modifiez cette copie pour obtenir le résultat demandé. Le test du programme consistera à lancer successivement plusieurs exécutions avec différentes valeurs de  $N$  (par exemple 100, 1000, etc.).

### 3.8.5 Exercice 3

Il s'agit maintenant de modifier le programme de simulation de façon à ce que pour une même valeur de  $N$  il réalise  $NS$  simulations successives ( $NS$  saisi au clavier) et calcule les moyennes des nombres de jours  $NJ$  et des nombres d'ignorants  $NI$  obtenus à chaque simulation. Créez une copie du programme `propagation2.py` appelée `propagation3.py` et modifiez cette copie de façon à obtenir le résultat attendu. Comparez vos résultats avec ceux obtenus par d'autres étudiants pour une même valeur de  $N$ .

### 3.8.6 Exercice 4

On veut à présent représenter sous forme *semi-graphique* l'évolution des statuts des personnes pendant le déroulement de la simulation. On appelle  $NC$ ,  $NM$  et  $NI$  les nombres de personnes ayant respectivement les statuts  $C$ ,  $M$  et  $I$ . Chaque jour l'état global de la population est représenté par le triplet  $(NC, NM, NI)$ , avec :  $NC + NM + NI = N$ . Initialement (le jour 0), ce triplet est égal à  $(1, 0, N - 1)$ . Cette nouvelle version du programme devra afficher pour chaque rencontre une suite de caractères représentant la répartition de la population en connaisseurs, muets et ignorants. Le caractère '#' sera utilisé pour représenter les connaisseurs, le caractère '\*' pour représenter les muets et le caractère '.' pour représenter les ignorants. Par exemple, pour  $N = 10$ ,  $NC = 3$ ,  $NM = 2$  et  $NI = 5$ , la répartition des statuts des personnes sera représentée par la suite de caractères :###\*... On constate que sont affichés dans l'ordre : les connaisseurs, puis les muets, et enfin les ignorants.

Si on reprend l'exemple de simulation pris dans l'exercice 1, l'affichage obtenu avec cette nouvelle version du programme serait :

jour	rencontre	nature	st.1	st.2	st.3	st.4	st.5	repartition
0			C	I	I	I	I	#....
1	3, 4	I, I	-	-	-	-	-	#....
2	2, 5	I, I	-	-	-	-	-	#....
3	1, 2	C, I	-	C	-	-	-	##...
4	3, 2	I, C	-	-	C	-	-	###..
5	1, 3	C, C	M	-	M	-	-	***..
6	3, 1	M, M	-	-	-	-	-	***..
7	5, 1	I, M	-	-	-	-	-	***..
8	3, 5	M, I	-	-	-	-	-	***..
9	4, 2	I, C	-	-	-	C	-	****.
10	4, 2	C, C	M	M	M	M	I	****.

Créer une copie du programme `propagation.py` appelée `propagation4.py`, puis modifiez cette copie afin que le programme obtenu affiche le résultat souhaité.

### 3.8.7 Bonus : visualisation avec matplotlib

On veut tracer deux graphiques :

- Pour l'exercice 3, un histogramme<sup>4</sup> montrant pour différentes tailles de population de départ les valeurs moyennes de  $NJ$  et  $NI$  en fin de  $NS$  simulations.
- Pour l'exercice 4, un graphique camembert<sup>5</sup> montrant les proportions  $NC$ ,  $NM$  et  $NI$  dans la population totale, et leur évolution au fil des jours de simulation.

4. Voir un exemple à : <https://pythonspot.com/en/matplotlib-histogram/>

5. Voir un exemple à : <https://pythonspot.com/en/matplotlib-pie-chart/>

## 3.9 TP semaine 9 : dictionnaires et traduction

### Contexte du problème

Dans ce TP on va manipuler les dictionnaires, une structure de données qui permet d'associer des clés à des valeurs. On va pour cela travailler sur des dictionnaires de langues, et pour commencer sur un dictionnaire français-anglais, qu'on utilisera pour traduire des mots et des textes.

#### 3.9.1 Exercice préliminaire : manipulation de dictionnaire français-anglais

Créer un fichier `traduc1.py` pour cet exercice.

1. Créer une fonction `ajouteMot` qui reçoit en paramètres un dictionnaire, un mot français, et sa traduction en anglais, et qui permet d'ajouter ce mot dans ce dictionnaire. La fonction modifie le dictionnaire reçu, et ne renvoie rien.
2. Créer une fonction `supprimeMot` qui reçoit en paramètres un dictionnaire et un mot, et qui permet de supprimer ce mot de ce dictionnaire. Cette fonction modifie le dictionnaire reçu, et ne renvoie rien.
3. Créer une fonction `afficheDico` qui reçoit en paramètre un dictionnaire et qui l'affiche, avec sur chaque ligne un mot suivi de sa traduction sous la forme *"chat = cat"*.
4. Créer une fonction `afficheDicoLettre` qui reçoit en paramètre un dictionnaire et une lettre, et qui n'affiche que les mots commençant par cette lettre, sous le même format que ci-dessus.
5. Créer une fonction `afficheDicoLettre` qui reçoit en paramètre un dictionnaire et un entier (une longueur), et qui n'affiche que les mots de la longueur donnée, sous le même format que ci-dessus.

Écrire ensuite un programme principal avec les étapes suivantes :

4. Créer une variable `fr` contenant un dictionnaire associant à quelques mots français leur traduction en anglais.
5. Appeler la fonction `ajouteMot` définie ci-dessus pour ajouter quelques mots dans le dictionnaire, puis l'afficher avec la fonction `afficheDico` pour vérifier.
6. Utiliser la fonction `supprimeMot` définie ci-dessus pour supprimer un mot du dictionnaire, et l'afficher avec `afficheDico` pour vérifier.

#### 3.9.2 Exercice 2 : mini-jeu de traduction

Créer un nouveau fichier `traduc2.py` pour cet exercice.

### Traduction français-anglais

1. Écrire une fonction `traduire` qui reçoit en paramètre le dictionnaire, et un mot français à traduire, et qui renvoie sa traduction en anglais.
2. Écrire une fonction `jouerUnMot` qui reçoit un dictionnaire, choisit au hasard un mot français de ce dictionnaire, et qui demande à l'utilisateur sa traduction en anglais. Cette fonction affiche un message de succès ou d'erreur, et renvoie un booléen indiquant si la réponse de l'utilisateur était correcte ou incorrecte.
3. Écrire un programme principal qui utilise ces deux fonctions pour jouer une partie : chaque tour de jeu correspond à la traduction d'un mot choisi au hasard ; puis le joueur peut choisir de continuer ou d'arrêter ; à la fin de la partie son score est affiché (nombre de traductions correctes sur nombre total de tours).



### Traduction dans les 2 sens

**Remarque :** cette partie est optionnelle et n'est pas nécessaire pour la suite du TP.

Modifier ce programme pour pouvoir demander des traductions dans les 2 sens (soit du français vers l'anglais, soit de l'anglais vers le français). Le sens de traduction pourra être donné comme paramètre supplémentaire optionnel des fonctions définies (valeur par défaut : sens français vers anglais).

Que se passe-t-il quand on demande la traduction en français d'un mot anglais qui a plusieurs traductions en français ?

Ce problème se pose-t-il dans l'autre sens de traduction ?

### 3.9.3 Exercice 3 : multi-joueurs

Créer un nouveau fichier `traduc3.py` pour cet exercice.

1. Écrire une nouvelle version du programme qui demande initialement le nombre de joueurs et le nom de chacun. Un dictionnaire est initialisé avec les scores de chaque joueur.
2. Le programme demande le nombre de tours de la partie, c'est-à-dire le nombre de mots à faire deviner.
3. Les joueurs jouent les mêmes mots (au choix : chaque joueur joue tous ses mots avant de passer au suivant ; ou bien chaque mot est joué par tous les joueurs à tour de rôle).
4. Le score de chaque joueur est calculé en pourcentage de mots correctement traduits par rapport au nombre total de mots de la partie (score entre 0 et 100 %), arrondi à l'entier inférieur.
5. Écrire une fonction qui reçoit le dictionnaire des scores des joueurs et renvoie le nom du meilleur joueur et son score (2 valeurs de retour).
6. Écrire une fonction qui reçoit le dictionnaire des scores des joueurs et affiche le tableau des meilleurs scores : les 3 meilleurs joueurs, un par ligne, avec leur score. En cas d'égalité ils seront affichés par ordre alphabétique mais avec le même numéro de rang. Par exemple :

1.	Tom	99\%
2.	Lea	97\%
2.	Nina	97\%

7. Tester ces fonctions de score, soit en faisant des parties avec au moins 4 joueurs, soit en entrant manuellement des scores dans le dictionnaire.

### 3.9.4 Exercice 4 : traduction de texte

Créer un nouveau fichier `traduc4.py` pour cet exercice.

#### Traduction partielle

1. Écrire une fonction qui reçoit en paramètre une chaîne de caractères (un texte) ainsi qu'un dictionnaire, et qui traduit ce texte de la manière suivante : les mots qui sont dans le dictionnaire sont remplacés par leur traduction, les autres sont laissés tels quels. La fonction renvoie la traduction sous forme d'une chaîne de caractères.

*Rappel : on peut diviser un texte en la liste de ses mots avec la fonction `split`, et on peut reformer une chaîne de caractères à partir d'une liste de mots avec la fonction `join`.*

2. Écrire un programme principal qui demande à l'utilisateur de saisir un texte, et affiche sa (semi-)traduction.

### Traduction interactive

1. Modifier la fonction de traduction ci-dessus pour qu'elle procède de la manière suivante pour chaque mot du texte à traduire :
  - Si le mot est connu (présent dans le dictionnaire) il est traduit
  - Si le mot n'est pas dans le dictionnaire, le programme demande à l'utilisateur la traduction de ce mot
    - Si l'utilisateur connaît la traduction, celle-ci est ajoutée au dictionnaire, et le mot est traduit dans le texte.
    - Si l'utilisateur ne connaît pas la traduction de ce mot, le mot est laissé tel quel dans la traduction du texte (comme dans la version 1) et le dictionnaire n'est pas modifié.

La fonction renvoie finalement le texte (complètement) traduit. On notera que cette fonction a aussi pour effet de bord de modifier le dictionnaire reçu en paramètre.

2. Écrire un programme principal qui demande à l'utilisateur un texte, appelle cette fonction pour le traduire, et affiche ensuite le texte traduit et le dictionnaire modifié.

### 3.9.5 Exercice 5 : multi-langue

1. Créer des dictionnaires pour une ou plusieurs autres langues (espagnol, allemand, japonais, chinois, etc), sur le même modèle que le dictionnaire français-anglais.
2. Créer un dictionnaire de dictionnaires : les clés sont les noms des langues, et les valeurs sont les dictionnaires français-langue étrangère correspondants
3. Écrire une fonction qui reçoit le dictionnaire de dictionnaires, un nom de langue LG2, un mot XX à traduire, et qui affiche sa traduction YY dans la langue demandée sous la forme : **"La traduction de XX de français vers LG2 est YY"**
4. Écrire un programme principal qui demande à l'utilisateur un mot, puis qui affiche la liste des langues disponibles (les clés du dictionnaire de dictionnaires), lui demande d'en choisir une (l'utilisateur doit taper une chaîne de caractères), et appelle la fonction ci-dessus pour afficher la traduction demandée.
5. Compléter ce programme principal pour demander à l'utilisateur un deuxième mot et afficher sa traduction dans toutes les langues disponibles.

## 3.10 TP 10 : modélisation de population

Nous nous intéressons à une population de poules dans un joli coin de campagne. On considère que la nourriture est en quantité infinie par rapport à cette population, et que le temps de reproduction des poules est beaucoup plus court que leur espérance de vie.

### 3.10.1 Natalité

On souhaite donc modéliser la population en supposant que le nombre  $P(t+1)$  de poules à l'instant  $t+1$  vaut le nombre de poules  $P(t)$  au temps  $t$  plus un nombre de nouveaux-nés proportionnel au taux de natalité  $nat\_poules$  (réel entre 0 et 1) et au nombre de poules  $P(t)$ . Soit la formule suivante :

$$P(t+1) = P(t) + nat\_poules * P(t)$$

1. Écrivez une fonction qui accepte en argument le nombre de poules initial, le taux de natalité, et le nombre de pas de temps, et qui renvoie le nombre de poules après le temps écoulé.
2. A quelle fonction ressemble l'évolution de la population ?
3. Que pensez-vous de l'hypothèse des ressources illimitées ?

### 3.10.2 Mortalité

En fait les poules ne sont pas immortelles. On souhaite donc prendre en compte le taux de mortalité  $mort\_poules$  (réel entre 0 et 1), et on modélise la population selon la nouvelle équation suivante :

$$P(t+1) = P(t) + nat\_poules * P(t) - mort\_poules * P(t)$$

1. Écrivez une fonction qui accepte en argument le nombre de poules initial, le taux de natalité, le taux de mortalité, et le nombre de pas de temps, et qui renvoie le nombre de poules après le temps écoulé.
2. Écrivez une fonction qui accepte en argument le nombre de poules initial, les bornes minimum et maximum des taux de natalité et de mortalité, le nombre de pas de temps, et qui renvoie une liste de listes de chaque combinaison `[nat_poules, mort_poules, pop_finale]`.

### 3.10.3 Renards

On désire prendre en compte maintenant la population des renards. Les renards mangent les poules, et leur taux de natalité est proportionnel au nombre de poules, alors que leur taux de mortalité est inversement proportionnel au nombre de poules. Ces correspondances sont exprimées par les formules suivantes en fonction de 2 constantes  $ren1$  et  $ren2$  :

$$nat\_renards = ren1 * P(t)$$

et

$$mort\_renards = ren2 / P(t)$$

La population de renards au temps  $t$  est représentée par  $R(t)$ . On doit donc maintenant définir le taux de mortalité des poules en fonction du nombre de renards comme :

$$mort\_poules = po1 + po2 * R(t)$$

Remarque :  $ren1$ ,  $ren2$ ,  $po1$  et  $po2$  sont en fait de simples coefficients de proportionnalité, ce sont des réels positifs. Par exemple on peut prendre  $ren1=ren2=1$ .  $po1$  est le nombre de poules mourant naturellement par pas de temps, alors que  $po2$  est le nombre de poules mangées par un renard par pas de temps.

On peut modéliser la population des renards par l'équation :

$$R(t+1) = R(t) + nat\_renards * R(t) - mort\_renards * R(t)$$

1. Écrivez une fonction qui accepte en argument le nombre de poules initial, le nombre de renards initial, les coefficients  $po1$ ,  $po2$ ,  $ren1$  et  $ren2$ , et le nombre de pas de temps. La fonction renvoie une liste de listes `[temps, nb_poules, nb_renards]` indiquant le nombre de poules et le nombre de renards à chaque pas de temps.

2. Comment ont l'air d'évoluer les populations en fonctions des paramètres ?
3. Écrivez un programme principal qui demande les coefficients à l'utilisateur et qui affiche chaque pas de temps, le nombre de poules, et le nombre de renards. Observez.

#### 3.10.4 Pour aller plus loin

1. Utiliser `matplotlib.pyplot` pour afficher graphiquement l'évolution des 2 populations au cours du temps, à partir de la fonction précédente.
2. Écrire une fonction qui accepte en argument le nombre de pas de temps et les bornes minimum et maximum de nombre de poules initial, de nombre de renards initial, des coefficients `po1`, `po2`, `ren1` et `ren2`, et qui renvoie une liste de listes `[nb_initial_poules, nb_initial_renards, po1, po2, ren1, ren2, nb_final_poules, nb_final_renards]`.
3. Introduire une population de vipères, qui se font manger par les poules et qui mordent les renards, et modifier les fonctions précédentes avec cette nouvelle population.

## 3.11 TP9 : fichiers et Python Tutor

### 3.11.1 Un peu de PythonTutor

Soit les programmes suivants extraits du fichier `pythontutor.py`, à télécharger.

```
def ajouter_zero(dico, clef):
    copie = dict(dico)
    copie[clef] = 0
    return copie

d = {'a': 12, 'b': 25}
d_avec_zero = ajouter_zero(d, 'c')

print("Avant: ", d)
print("Après: ", d_avec_zero)
```

```
def ajouter_zero_liste(dico, clef):
    copie = dict(dico)
    copie[clef].insert(0, 0)
    return copie

e = {'a': [1, 2, 3], 'b': [4, 5, 6]}
e_avec_zero = ajouter_zero_liste(e, 'a')

print("Avant: ", e)
print("Après: ", e_avec_zero)
```

1. Avant d'exécuter les programmes, prédire ce qui sera affiché. Identifiez les différences entre les deux programmes et leur influence sur le comportement attendu.
2. Copier le contenu du fichier `pythontutor.py` sur PythonTutor et observer son exécution pas à pas.
3. Les programmes se comportent-ils comme prévu ? Si non, que feriez-vous pour corriger ce problème ?

### 3.11.2 Chaînes de caractères - Compléments

Dans cette partie, nous allons voir des fonctions relatives aux chaînes de caractères qui peuvent être particulièrement utiles lorsque l'on travaille avec des fichiers. Pour cette partie, vous travaillerez sur la console d'IDLE. Exécutez les instructions suivantes et observez le résultat :

```
>>> message1 = 'Hello world\n'
>>> print(message1)
>>> message2 = 'Hello world\n\n'
>>> print(message2)
>>> print(message2.strip())
>>> message4 = '    hi    '
>>> print(message4.strip())
```

```
>>> 'HELLO'.lower()
>>> 'hello'.upper()
>>> 'HeLlO'.lower()
>>> 'a b c'.split()
>>> 'a:b:c'.split(':')
>>> ','.join(['a','b','c'])
>>> ':'.join(['a','b','c'])
```

1. Question : À quoi sert chacune des fonctions suivantes : `strip()`, `lower()`, `upper()`, `split()` et `join()` ?  
Pour en savoir plus, n'hésitez pas à consulter la documentation :

Section 4.7.1 *String methods*

<https://docs.python.org/3.6/library/stdtypes.html#string-methods>

2. Complétez les programmes ci-dessous :

<pre>&gt;&gt;&gt; s1='SaLuT' &gt;&gt;&gt;  &gt;&gt;&gt; s2 'salut'</pre>	<pre>&gt;&gt;&gt; s1=' ok \n' &gt;&gt;&gt;  &gt;&gt;&gt; s2 'ok'</pre>	<pre>&gt;&gt;&gt; s1=['yes', 'no', 'maybe'] &gt;&gt;&gt;  &gt;&gt;&gt; s2 'yes-no-maybe'</pre>	<pre>&gt;&gt;&gt; s1='Albus.Severus.Potter' &gt;&gt;&gt;  &gt;&gt;&gt; s2 ['Albus', 'Severus', 'Potter']</pre>
--	--	--	--

### 3.11.3 Travailler avec des fichiers texte - Premiers pas

Nous allons maintenant apprendre à utiliser des fichiers dans nos programmes. Avant de commencer nous allons préparer notre "espace de travail".

1. Créez un dossier appelé "TP8" sur votre bureau. Au long de ce TP, on appellera ce dossier votre "répertoire de travail". Vous devez sauvegarder tous vos programmes, ainsi que les fichiers téléchargés, dans ce même répertoire.

## Proverbe du jour

Le fichier `proverbs.txt`, à télécharger, contient un grand nombre de proverbes en anglais en une seule ligne, séparés par des points ".", ce qui le rend assez difficile à lire. On veut tout d'abord reformatter ce fichier pour le rendre plus lisible.

1. Téléchargez le fichier `proverbs.txt` vers votre répertoire de travail.
2. Ecrire un programme qui crée un nouveau fichier nommé `proverbs_readable.txt` qui contient les mêmes proverbes que le fichier original mais avec un seul proverbe par ligne. Vérifiez que le fichier a bien été créé dans votre répertoire de travail, et qu'il est bien au bon format.
3. Ecrire un programme qui lit l'un des deux fichiers, et affiche un message d'accueil, suivi d'un proverbe aléatoire (utiliser le module `random`). Exemple :

```
Bonjour, voici votre proverbe du jour:
"Actions speak louder than words"
```

### 3.11.4 Champion de scrabble

Dans cette partie, nous allons utiliser les notions vues précédemment combinées avec les dictionnaires pour développer un programme qui sait jouer au célèbre jeu du scrabble.

#### Anagrammes

Pour commencer, on veut créer une fonction qui permet de vérifier si un ensemble de lettres (chaîne de caractères) est un anagramme d'un mot déjà connu. A l'aide des dictionnaires, il est assez facile de faire cette vérification. En effet, si on représente chaque mot par un dictionnaire associant à chaque lettre du mot le nombre de fois où elle apparaît dans le mot, le dictionnaire généré pour un mot et celui généré pour un de ses anagrammes sont identiques.

1. En vous inspirant de l'exercice 4 du TD, écrivez une fonction `mot_vers_dictionnaire(mot)`, qui prend un mot (chaîne de caractères) et renvoie le dictionnaire correspondant.
2. En utilisant la fonction `mot_vers_dictionnaire` écrire une fonction `est_anagramme(lettres, mot)` qui envoie `True` si la chaîne de caractères `lettres` est un anagramme du mot `mot`, `False` sinon.

```
>>> est_anagramme("tusla", "salut")
True
>>> est_anagramme("salus", "salut")
False
```

#### Solution parfaite

En utilisant les fonctions précédemment définies, on peut déjà construire une première version de notre programme. Cette première version va prendre les lettres que l'utilisateur a tirées, et lui afficher les mots français qui peuvent être formés en utilisant TOUS les jetons qu'il a en main.

Le fichier `mots.txt` à télécharger est un fichier texte contenant tous les mots français du petit Larousse. Il contient un mot par ligne, sans accents, et en majuscule.

1. Téléchargez le fichier `mots.txt` vers votre répertoire de travail.
2. Ecrire un programme principal qui demande à l'utilisateur une chaîne de caractères, et qui affiche tous les mots du fichier `mots.txt` qui peuvent être formés à partir de ces lettres. (Utiliser la fonction `est_anagramme`). L'utilisateur peut saisir des majuscules ou des minuscules. Utiliser les fonctions vues dans la première partie du TP pour que le programme marche quelle que soit la casse.

```
Quelles lettres avez vous en main ? taslu
Voici les mots que vous pouvez jouer:
SALUT
TALUS
```

### Solution exhaustive

Comme il n'est pas toujours possible de former un mot en utilisant toutes les lettres disponibles, on aimerait que notre programme soit également capable de nous renvoyer tous les mots que l'on peut former en utilisant seulement une partie des jetons à notre disposition. Cela requiert une méthode un peu plus élaborée que celle des anagrammes.

Si on considère toujours la représentation d'un mot par un dictionnaire, pour vérifier qu'une chaîne *C* permet de former un mot *M*, il suffit de vérifier que chaque lettre du mot *M* apparaît dans la chaîne *C* au moins une fois. Autrement dit, pour chaque lettre *L* du mot *M*, le nombre d'apparitions de *L* dans *C* doit être supérieur ou égal au nombre d'apparitions de *L* dans *M*.

1. Ecrire une fonction `peut_ecrire(lettres, mot)` qui retourne `True` si le mot `mot` peut être écrit en utilisant des lettres de la chaîne de caractères `lettres`, `False` sinon.

```
>>> peut_ecrire("tatxslua", "salut")
True
>>> peut_ecrire("lasautaionna", "salutation") # pas assez de "t"
False
```

2. Ecrire maintenant le programme qui permet d'afficher tous les mots français pouvant être formés par les lettres saisies par l'utilisateur. On n'affichera que les mots de longueur 3 ou plus.

```
Quelles lettres avez vous en main ? taslu
Voici les mots que vous pouvez jouer:
ALU
AULT
LAS
LATS
LUT
SAL
SALUT
SAUT
TALUS
TAS
TAU
USA
```

### Solution optimale

Dans le jeu du scrabble, chaque lettre est associée à un nombre de points. Ce sont ces points qui permettent de calculer le score de chaque joueur à la fin de la partie. On aimerait que notre programme soit capable de choisir, parmi les mots qu'on peut jouer, celui qui a le plus grand total de points.

Pour ce faire, on stocke les points correspondant à chaque lettre dans un fichier texte appelé `points.txt`. Chaque ligne de ce fichier contient une lettre et son nombre de points, séparés par des espaces.

1. Téléchargez le fichier `points.txt` vers votre répertoire de travail.
2. Ecrire une fonction `lire_points()` qui lit le fichier `points.txt` et renvoie un dictionnaire faisant correspondre à chaque lettre son nombre de points (entier).
3. Ecrire une fonction `score_mot(points, mot)` qui prend en argument un dictionnaire faisant correspondre à chaque lettre son nombre de points, et un mot, et qui renvoie le score.
4. Modifier le programme de l'exercice précédent pour qu'il n'affiche que le meilleur mot à jouer, ainsi que le score correspondant.

```
Quelles lettres avez vous en main ? taslu
Voici le meilleur mot que vous pouvez jouer:
SALUT pour un score de: 9
```

### Avec Jokers

Dans le jeu du scrabble, il existe des jetons blancs appelés Jokers. Ces jetons peuvent être utilisés pour remplacer n'importe quelle lettre de l'alphabet. Les jokers valent 0 points. Dans notre programme, le joker sera représenté par le caractère '\_'.

1. Reprenez le programme défini dans l'exercice 4 en modifiant la fonction `peut_ecrire` afin qu'elle prenne en compte les jokers dans la chaîne saisie par l'utilisateur.

```
Quelles lettres avez vous en main ? r_sg
Voici les mots que vous pouvez jouer:
...
GARS
GPRS
GRAS
GRE
GRES
GRIS
GROS
...
```

2. Pouvez-vous modifier le programme de l'exercice 5 de sorte à ce que les jokers ne rentrent pas dans le calcul du score ?



## Chapitre 4

# Annales

# INF101/INF131 : devoir surveillé

Jeudi 26 Octobre 2017

- **Durée 2 heures.**
- **Lisez toutes les questions avant de commencer !** Le sujet fait quatre (4) pages.
- Respectez strictement les consignes de l'énoncé (noms de variables, format d'affichage...)
- Pensez à commenter vos programmes pour les rendre plus lisibles.
- Les exercices sont indépendants, et en ordre croissant de difficulté.
- Aucun document autorisé. L'utilisation d'une calculatrice, d'un téléphone, etc est interdite.
- Le barème est indicatif. La qualité de la rédaction et de la présentation sera prise en compte.

## 1 EXERCICE 1 : BOOLEENS (2.5 points)

On suppose que toutes les variables nécessaires sont bien initialisées. Écrire les expressions booléennes vraies si:

- a est un entier strictement positif et multiple de 3
- a est soit positif pair, soit négatif impair
- c est une lettre de l'alphabet en minuscule ou en majuscule (**remarque** : on interdit d'utiliser la fonction `isalpha` de Python; attention c pourrait être une chaîne de plusieurs caractères)
- x, y, z sont dans l'ordre croissant (non strict) et tous strictement négatifs
- x est un nombre réel non entier

## 2 EXERCICE 2 : CONVERSIONS (2.5 points)

On veut écrire un programme pour aider un coureur à calculer son allure de course. Chaque coureur connaît sa vitesse maximale en km/h (par exemple 21.2 km/h). Son entraîneur lui demande de courir à un certain pourcentage de cette vitesse maximale (par exemple 75%). Sa montre n'affiche pas la vitesse mais l'allure en mn/km (par exemple 5 mn/km correspond à 12 km/h). Écrire un programme qui réalise les opérations suivantes :

1. Demander au coureur de saisir sa vitesse maximale en km/h, qui doit être strictement positive, et inférieure ou égale à 30. Le programme filtre la valeur saisie jusqu'à ce qu'elle respecte cette contrainte (variable `vmax`).
2. Demande au coureur de saisir le pourcentage de cette vitesse auquel il doit courir (entier entre 0 et 100), et filtre cette valeur jusqu'à ce qu'elle soit bien comprise entre 0 et 100 (variable `pc`).
3. Calcule la vitesse (variable `v`) correspondant à ce pourcentage et l'affiche.
4. Calcule l'allure (variable `a`) correspondant à la vitesse `v` et l'affiche. (Attention on veut l'allure en minutes et secondes par kilomètres. Le nombre de minutes doit être un entier, le nombre de secondes un réel qu'on pourra arrondir à 2 chiffres après la virgule avec la fonction `round`).

On demande de respecter **strictement** l'affichage de l'exemple suivant (à l'espace près) :

```
Quelle est ta vitesse maximale en km/h ? 15.4
Quel pourcentage (0-100) ? 180
Quel pourcentage (0-100) ? 80
La vitesse correspondant à 80 pourcents de ton max est : 12.32 km/h
L'allure correspondant à 12.32 km/h est : 4'52''/km
```

**Rappel** : on peut spécifier le séparateur voulu dans l'affichage en précisant `sep=...`

### 3 EXERCICE 3 : PYRAMIDE (4 points)

Écrire un programme qui :

1. demande à l'utilisateur un nombre de lignes (variable `nbl`) et une lettre de départ (variable `dep`). On suppose que l'utilisateur saisit toujours une minuscule, on n'a pas besoin de le vérifier.
2. affiche une pyramide de lettres à `nbl` lignes, sous la forme suivante. Si on dépasse la fin de l'alphabet on veut recommencer à la lettre 'a' après le 'z'.

Par exemple: à gauche pour 4 lignes (`nbl=4`) et départ à 'a' (`dep='a'`); à droite pour 3 lignes (`nbl=3`) et départ à 'u' (`dep='u'`).

a	u
b c d	v w x
e f g h i	y z a b c
j k l m n o p	

Rappels :

- la fonction `ord(caractere)` renvoie le code ASCII du caractère reçu en paramètre. La fonction `chr(code)` renvoie le caractère ayant le code ASCII reçu en paramètre.
- `print` a 2 paramètres optionnels, `sep` qui permet de modifier le séparateur entre les chaînes affichées (espace par défaut), et `end` qui permet de modifier la fin de l'affichage (retour à la ligne par défaut).

### 4 Exercice 4 : mélange de liste (3 points)

1. Écrire une fonction `melange_liste` qui reçoit en paramètre une liste, et renvoie une liste contenant les mêmes éléments dans le désordre.  
**Attention** : on ne veut PAS modifier la liste reçue en paramètre mais renvoyer une nouvelle liste.  
**Indice** : on peut faire une copie de la liste reçue, dans une nouvelle variable qui pourra elle être modifiée.
2. Écrire ensuite un programme principal qui :
  - (a) crée une liste vide
  - (b) demande à l'utilisateur de saisir des entiers, et les ajoute à cette liste. L'utilisateur doit taper -1 pour terminer la saisie (-1 n'est pas ajouté à la liste)
  - (c) affiche la liste ainsi constituée de tous les éléments saisis (dans l'ordre de saisie)
  - (d) mélange cette liste en appelant la fonction écrite ci-dessus
  - (e) affiche la liste mélangée

**Indice** : on utilisera la fonction `randint` du module `random`. **Remarque** : le module `random` contient aussi déjà une fonction `shuffle` qui permet de mélanger une liste, mais on demande de ne **PAS** l'utiliser ici, le but de l'exercice est de la réécrire. (Par ailleurs cette fonction modifie la liste manipulée, alors qu'on demande ici de renvoyer une copie mélangée de la liste de départ sans modifier celle-ci.)

**Rappel de quelques fonctions utiles sur les listes** : `len` (longueur d'une liste) ; `append` (ajouter un élément à la fin d'une liste) ; `pop` (récupérer l'élément à un certain indice et le supprimer de la liste).

## 5 Exercice 5 : message secret (8 points)

### 5.1 Fonctions

Définir les fonctions suivantes permettant de coder un mot de différentes manières selon le jour de la semaine :

1. La fonction `lundi` reçoit un mot, choisit un entier `n` au hasard entre 2 et 7, et renvoie le mot répété `n` fois, séparés par des `+`. Par exemple si le mot est "salut" et que l'entier tiré est 3, le résultat est "salut+salut+salut". On remarquera qu'il n'y a **pas** de `+` à la fin de la chaîne.
2. La fonction `mardi` reçoit un mot, et renvoie ce mot écrit à l'envers. Par exemple si le mot est "bonjour", le résultat est "ruojnob".  
**Rappel** : si `c` est une chaîne de caractères alors `list(c)` permet d'obtenir la liste de toutes ses lettres.
3. La fonction `mercredi` reçoit un mot, choisit un entier `x` au hasard entre 1 et 10, et opère un décalage de `x` positions sur chaque lettre du mot (en rebouclant sur 'a' après le 'z' ou sur 'A' après le 'Z'). Par exemple le mot "Salut" avec un décalage de 7 sera codé en "Zhsba" car 'Z' est situé 7 lettres après 'S' dans l'alphabet en majuscules, 'h' est situé 7 lettres après 'a' dans l'alphabet en minuscules, etc. Pour 'u' et 't' on voit qu'on dépasse le 'z' et on revient donc au début de l'alphabet. On pourra écrire une fonction auxiliaire qui opère le décalage sur une lettre (en considérant bien les majuscules et les minuscules).
4. La fonction `jeudi` reçoit un mot, supprime toutes ses voyelles (majuscules comme minuscules), et renvoie le mot constitué des lettres (consonnes) restantes. Par exemple "salut" devient "slt", "Alice" devient "lc". On pourra écrire une fonction auxiliaire qui teste si une lettre est une voyelle.
5. La fonction `vendredi` reçoit un mot, change les minuscules en majuscules, et vice-versa. Par exemple "Salut" devient "sALUT". On pourra écrire une fonction auxiliaire qui transforme une minuscule en majuscule et vice versa.  
**Remarque** : on **interdit** ici d'utiliser les fonctions `upper`, `lower`, `isupper`, `islower`, déjà définies dans Python; on veut les reprogrammer.
6. La fonction `samedi` reçoit un mot, mélange ses lettres, et renvoie le mot mélangé.  
**Remarque**: on pourra cette fois utiliser la fonction `shuffle` du module `random`. **Rappel** : la fonction `shuffle` reçoit une liste, la mélange, et ne renvoie rien; c'est la liste reçue qui est modifiée pour être mélangée.
7. La fonction `dimanche` reçoit un mot mais ne fait aucun codage et le renvoie tel quel.

### 5.2 Programme principal

8. Définir ensuite un programme principal qui :
  - (a) Opère en boucle en demandant un mot à l'utilisateur, jusqu'à lire le mot "stop".
  - (b) Pour chaque mot saisi par l'utilisateur (sauf "stop"), lui demande un jour de la semaine et affiche le résultat du codage de ce mot selon la fonction correspondant au jour choisi.
  - (c) Recommence ensuite avec le mot suivant.
9. **Question bonus** : Écrire un nouveau programme principal qui, au lieu de lire un seul mot, lit un texte entier. Ensuite le programme demande un jour de la semaine, code chaque mot du texte selon la fonction correspondante, et affiche le texte résultant. De la même manière qu'à la question 8, ce programme s'arrête si le texte saisi est "stop".

Par exemple pour le texte "Bonjour a tous" codé avec la fonction du mardi, on affichera "ruojnoB a suot". (On notera que ce n'est pas le texte qui est à l'envers, mais bien chaque mot qui est à l'envers.)

**Rappel** : la fonction `split()` permet de découper une chaîne de caractères selon un séparateur (ici le séparateur est l'espace entre les mots), et renvoie la liste des mots ainsi obtenue. Par exemple `"Bonne chance a tous".split(" ")` renvoie la liste `["bonne", "chance", "a", "tous"]`.

## INF101/INF131 : examen final

??di ?? Décembre 2017

- **Durée 3 heures.**
- **Lisez toutes les questions avant de commencer !** Le sujet fait ?? pages.
- Respectez **strictement** les consignes de l'énoncé (noms de variables, format d'affichage...)
- Pensez à commenter vos programmes pour les rendre plus lisibles.
- Les exercices sont indépendants.
- Aucun document autorisé. L'utilisation d'une calculatrice, d'un téléphone, etc, est interdite.
- Le barème est indicatif. La qualité de la rédaction et de la présentation sera prise en compte.

## 1 Nombres narcissiques

Exercice similaire à l'exo 40 page 59 du poly de TD sur les nombres de Armstrong.

Un nombre narcissique (ou nombre d'Armstrong de première espèce) est un entier naturel  $n$  non nul qui est égal à la somme des puissances  $p$ -ièmes de ses chiffres en base dix, où  $p$  désigne le nombre de chiffres de  $n$  :

$$n = \sum_{k=0}^{p-1} x_k 10^k = \sum_{k=0}^{p-1} (x_k)^p \quad \text{avec } x_k \in \{0, \dots, 9\} \quad \text{et } x_{p-1} \neq 0.$$

Par exemple :

- Tous les entiers de 1 à 9 sont narcissiques.
- $153 = 1^3 + 5^3 + 3^3$
- $548834 = 5^6 + 4^6 + 8^6 + 8^6 + 3^6 + 4^6$

1. Écrire une fonction `nombreChiffres` qui calcule et renvoie le nombre  $p$  de chiffres d'un entier naturel  $x$  reçu en argument.
2. Écrire une fonction `sommePuissances` qui reçoit en argument un entier naturel  $x$ , et calcule et renvoie la somme des puissances  $p$ -ièmes de ses chiffres, où  $p$  est son nombre de chiffres (qui sera calculé avec la fonction précédente).
3. Écrire une fonction `narcissique` qui reçoit en argument un entier naturel  $x$ , et renvoie un booléen indiquant si ce nombre est narcissique.
4. Écrire une fonction `listeNarcissiques` qui reçoit en argument une borne maximum, et qui construit et renvoie la liste des nombres narcissiques inférieurs à cette borne. On rappelle qu'un entier narcissique doit être non nul.
5. Écrire un programme principal qui demande une borne maximum à l'utilisateur, puis affiche la liste des nombres narcissiques inférieurs à cette borne, sur une seule ligne, séparés par des virgules, avec un retour à la ligne final.

## 2 Estimation du nombre $\pi$

Exo similaire aux exos 15-16-17 de la feuille de TD de la semaine 8 sur les boucles FOR

Il existe plusieurs formules pour calculer le nombre  $\pi$ . On considère en particulier dans cet exercice les deux formules suivantes :

- $\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$  : cette formule de la somme alternée des inverses des nombres impairs, s'approche très très lentement du nombre  $\pi$ .
- $\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \dots$  : cette somme des inverses des carrés des entiers converge plus rapidement que la formule précédente.

1. Écrire une fonction `estimPi1` qui reçoit en paramètre une borne max (l'entier impair constituant le dénominateur de la dernière fraction de la somme), calcule et renvoie l'estimation correspondante de  $\pi$  selon la première formule. Attention la formule permet de calculer  $\pi/4$  et pas  $\pi$ . [Rajouter un exemple ?](#)
2. Écrire une fonction `estimPi2` qui reçoit en paramètre une borne max (l'entier dont le carré constitue le dénominateur de la dernière fraction de la somme), calcule et renvoie l'estimation correspondante de  $\pi$  selon la deuxième formule. Attention la formule permet de calculer  $\pi^2/6$  et pas  $\pi$ . [Rajouter un exemple ?](#)
3. Écrire une fonction qui reçoit en paramètre une précision souhaitée dans l'approximation de  $\pi$  (écart en valeur absolue entre l'estimation et la valeur de  $\pi$ , qu'on pourra obtenir avec `math.pi`), et utilise les fonctions ci-dessus pour calculer les estimations successives de  $\pi$  par les deux formules, jusqu'à ce qu'au moins l'une des deux estimations fournisse la précision souhaitée. La fonction renvoie alors le numéro de la formule la plus proche (1 ou 2), la borne max nécessaire, et l'estimation de  $\pi$  obtenue par cette formule. [Rajouter un exemple ?](#)
4. Écrire un programme principal qui demande à l'utilisateur une précision p, puis affiche quelle formule atteint cette précision en premier, et avec quelle borne max. Le programme doit ensuite proposer à l'utilisateur de rejouer avec une nouvelle précision.

## 3 Jeu de la vie de Conway

Notions : listes de listes

Le jeu de la vie de Conway est un automate cellulaire dont les règles sont extrêmement simples. On considère une grille carrée de cellules qui peuvent être soit vivantes soit mortes. L'évolution de l'état d'une cellule dépend de son nombre de voisines vivantes parmi les 8 cellules qui l'entourent (cf figure 1):

- Une cellule morte qui a exactement 3 voisines vivantes devient vivante (elle naît);
- Une cellule vivante qui a exactement 2 ou 3 voisines vivantes reste vivante, sinon elle meurt.

On choisit ici de représenter cette grille de la manière suivante :

- L'état de chaque cellule est un booléen : True pour vivante, False pour morte.
- La grille est une liste de listes : chaque élément de la liste est une liste contenant les états de toutes les cellules de cette ligne.

Par exemple la grille de gauche sur la figure 1 ci-dessus est représentée par la liste suivante : `[[False, False, False, False], [False, True, True, False, False], [False, True, False, True, False], [False, False, False, False, False], [False, False, False, False, True]]`

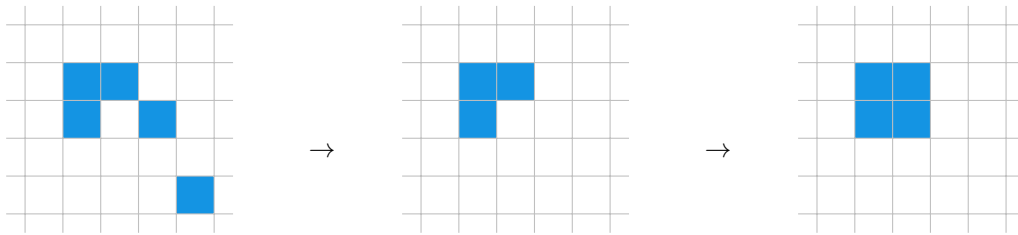


Figure 1: Un exemple d'évolution sur trois étapes d'un extrait de grille dans le jeu de la vie. Les cellules mortes sont en blanc, les cellules vivantes en bleu.

1. Écrire une fonction qui reçoit en argument la taille  $N$  de la grille carrée, et le nombre  $V$  de cellules vivantes initiales. Cette fonction crée une grille de la bonne taille ; l'initialise avec exactement  $V$  cellules vivantes (placées au hasard) et toutes les autres mortes ; et renvoie cette grille. Attention à vérifier que  $V$  n'est pas trop grand (dans ce cas on renvoie une grille dont toutes les cellules sont vivantes).
2. Écrire une fonction qui reçoit en paramètre une grille et l'affiche de la manière suivante : les cellules vivantes sont représentées par le caractère '\*' (étoile), et les cellules mortes par le caractère '.' (point), sans espace entre les valeurs. Par exemple la grille ci-dessus sera affichée sous la forme suivante:

```
.....
.**...
.*.*.
.....
....*
```

3. Écrire une fonction qui compte et renvoie le nombre de voisins vivants (parmi les 8 cellules voisines au maximum) d'une cellule donnée (par son numéro de ligne et son numéro de colonne) d'une grille donnée (attention aux bords !).
4. Écrire une fonction qui reçoit la grille en argument, et crée et renvoie une nouvelle grille contenant le nouvel état de toutes les cellules après une évolution.
5. Écrire une fonction qui reçoit une grille en argument, et calcule et renvoie le nombre de cellules vivantes dans cette grille.
6. Écrire un programme principal qui:
  - Demande à l'utilisateur la taille de la grille, et le nombre initial de cellules vivantes.
  - Initialise une grille avec ces valeurs, et l'affiche.
  - Demande à l'utilisateur un nombre de tours
  - Fait évoluer la grille en affichant à chaque tour le numéro du tour, la nouvelle grille, et le nombre de cellules vivantes.
  - S'arrête après le nombre de tours demandés, ou avant dès qu'il n'y a plus aucune cellule vivante.



## 4 Dictionnaire de cryptage

On s'intéresse dans cet exercice à des dictionnaires dont les clés sont les lettres de l'alphabet en minuscule, et dont les valeurs sont aussi des lettres minuscules de l'alphabet. Un tel dictionnaire associe chaque lettre à une lettre (elle-même ou une autre), chaque lettre n'apparaissant comme valeur que d'une seule clé, réalisant ainsi une permutation de l'alphabet. Cela permet donc de coder un mot ou un texte en remplaçant chaque lettre par une autre. [Ajouter des exemples.](#)

1. Écrire une fonction qui crée et renvoie un dictionnaire qui associe aléatoirement à chaque lettre de l'alphabet une autre lettre. Attention chaque lettre ne doit être utilisée qu'une seule fois. Par exemple si la clé "e" est associée à la valeur "a", alors aucune autre clé ne doit être associée à cette même valeur. Ce dictionnaire constitue donc une permutation de l'alphabet.
2. Écrire une fonction qui reçoit un mot et un dictionnaire dont les clés et les valeurs sont des lettres de l'alphabet. Cette fonction calcule et renvoie ce mot codé par ce dictionnaire, c'est-à-dire dans lequel chaque lettre est remplacée par sa valeur dans le dictionnaire.
3. Écrire une fonction qui reçoit en argument un tel dictionnaire, et qui calcule et renvoie quelle lettre (clé) est associée dans ce dictionnaire avec la lettre la plus lointaine. Pour cela on calculera l'écart en valeur absolue entre la clé et la valeur associée (par exemple si la clé 'e', 5e lettre, est associée à la valeur 'a', 1e lettre, l'écart est de 4; si la clé 'i', 9e lettre, est associée à la valeur 'z', 26e lettre, l'écart est de 17).
4. Écrire une fonction qui reçoit en argument un dictionnaire associant chaque lettre à son codage par une autre lettre, et qui génère et renvoie le dictionnaire permettant de **décoder** les mots codés avec ce dictionnaire. Par exemple à partir du dictionnaire 'a':'e','b': 'c','c': 'd','d': 'a','e': 'b', cette fonction renvoie le dictionnaire inversé 'a': 'd','b': 'e','c': 'b','d': 'c','e': 'a'. Ainsi si on code un mot avec le premier dictionnaire, puis qu'on code le résultat avec le deuxième dictionnaire, on doit retomber sur le mot initial.
5. Écrire un programme principal qui génère un dictionnaire aléatoire, demande à l'utilisateur un mot, le code et affiche le résultat, puis calcule le dictionnaire inverse, décode le mot et affiche le résultat.

## INF101/INF131 : examen 2e session

Lundi 18 Juin 2018

- **Durée 2 heures.**
- **Lisez toutes les questions avant de commencer !** Le sujet fait 2 pages.
- Respectez **strictement** les consignes de l'énoncé (noms des fonctions et variables, format d'affichage...)
- Les exercices sont indépendants et ne sont **pas** dans l'ordre de difficulté. Vous pouvez sauter des exercices et des questions. Vous pouvez supposer existantes les fonctions des questions précédentes et les utiliser, même si vous n'avez pas répondu à la question correspondante.
- Il est inutile de filtrer les entrées lorsque ce n'est pas explicitement demandé (elles seront supposées correctes).
- **Aucun document autorisé. L'utilisation d'une calculatrice, d'un téléphone, etc, est interdite.**
- **Chaque question vaut 1 point.** Le barème est indicatif. La qualité de la rédaction et de la présentation sera prise en compte. Pensez à commenter vos programmes pour les rendre plus lisibles. Attention à l'indentation. Répondez à **chaque exercice sur une feuille séparée.**

## 1 Triplets de Pythagore (5 points)

Un triplet de Pythagore est un ensemble de 3 entiers naturels,  $a < b < c$ , tels que  $a^2 + b^2 = c^2$ . Par exemple,  $3^2 + 4^2 = 9 + 16 = 25 = 5^2$ .

1. Écrire une fonction `triplets` qui calcule et renvoie la liste des triplets de Pythagore tels que  $a$  est inférieur ou égal à une borne maximale reçue en paramètre.
2. Écrire une fonction `tripletsSomme` qui reçoit en paramètre un entier  $s$ , et affiche le triplet de Pythagore tel que  $a + b + c = s$  s'il existe, ou un message d'erreur sinon. On se servira de la fonction précédente. *Indice: quelle est la valeur maximum de  $a$  pour une somme  $s$  donnée ?*
3. Écrire un programme principal qui demande à l'utilisateur une somme, et utilise la fonction précédente pour afficher le triplet de Pythagore dont la somme correspond à celle demandée (s'il existe).

## 2 Codage de mots (4 points)

Dans cet exercice, on appelle 'mot' une chaîne de caractères constituée uniquement de lettres minuscules de l'alphabet.

- Écrire une fonction `codage1` qui reçoit en paramètre un mot, et renvoie la somme des positions de ses lettres dans l'alphabet. Par exemple pour le mot 'salut', on renverra 73 ( $19+1+12+21+20$ ).
- Écrire une fonction `codage2` qui reçoit en paramètre un mot et renvoie le mot obtenu en remplaçant chaque lettre par la suivante (le 'z' devient un 'a'). Par exemple 'salut' devient 'tbmvu'.
- Écrire une fonction `codage3` qui reçoit en paramètre un entier strictement positif, et qui renvoie le mot constitué des lettres dont la position correspond à chaque chiffre, en comptant à partir de 0 (il ne peut donc y avoir que des lettres entre a, position 0, et j, position 9). Par ex : 1203 devient 'bcad'; 2708 devient 'chai'. *Attention au sens !*
- Écrire une fonction `code4` qui reçoit en paramètre un mot, et renvoie un mot composé des mêmes lettres mais mélangées dans un ordre aléatoire. *Indice: on pourra utiliser la fonction `shuffle` du module `random`, sans oublier de l'importer.*

### 3 Démineur (6 points)

On s'intéresse à une grille de démineur carrée, représentée par une liste de listes d'entiers. Une case vide est représentée par l'entier 0, une case minée par l'entier 1. Pour éviter les effets de bord, on stocke une grille de démineur de taille  $n$  dans une liste de  $n+2$  listes de  $n+2$  entiers (les entiers placés sur les bordures de la grille valent nécessairement 0).

1. Écrire une fonction `initGrille` qui reçoit en paramètre un entier  $n$ , calcule et renvoie une liste de listes de taille  $n+2$ , ne contenant que des entiers 0.
2. Écrire une fonction `placeMines` qui reçoit en paramètre une liste de listes et un entier  $x$  (nombre de mines). Cette fonction ajoute exactement  $x$  mines dans la grille, en évitant les bordures. Cette fonction modifie directement la grille reçue en paramètre, elle ne renvoie rien. *Indice: on pourra tirer une position vide au hasard avec `random.randint(inf, sup)`.*
3. Écrire une fonction `affiche` qui reçoit en paramètre une grille d'entiers, et l'affiche sous forme d'un carré, les entiers de chaque ligne séparés par un espace, avec un retour à la ligne entre chaque ligne.
4. Écrire une fonction `lignePlusMinee` qui reçoit en paramètre une grille, cherche la ligne contenant le plus de mines, et renvoie le numéro de cette ligne et le nombre de mines qu'elle contient.
5. Écrire une fonction `ligneNonMinee` qui reçoit en paramètre une grille carrée, cherche la première ligne ne contenant aucune mine, et renvoie son numéro, ou -1 si une telle ligne n'existe pas.
6. Écrire un programme principal qui demande à l'utilisateur la taille de la grille de démineur et le nombre de mines, utilise les fonctions ci-dessus pour initialiser la grille, y placer le bon nombre de mines, et l'afficher sous forme d'un carré. Le programme affiche ensuite un message indiquant le numéro de la ligne la plus minée et son nombre de mines, et le numéro de la première ligne non minée si elle existe.

### 4 Dictionnaire de traduction (5 points)

1. Écrire une fonction `initListeMots` qui ne reçoit aucun paramètre, demande à l'utilisateur de saisir des mots français en terminant par 'stop', enregistre tous ces mots dans une liste (sauf 'stop'), et renvoie cette liste.
2. Écrire une fonction `initDicoAnglais` qui reçoit une liste de mots français, demande à l'utilisateur la traduction de chaque mot en anglais, et initialise un dictionnaire français-anglais (clé = mot français, valeur = mot anglais).
3. Écrire une fonction `supprimerDoublons` qui reçoit en paramètres un dictionnaire français-anglais et une liste de mots français; elle parcourt les mots français pour vérifier qu'il n'y a pas plusieurs mots ayant la même traduction dans le dictionnaire reçu. Si c'est le cas, alors les doublons sont supprimés du dictionnaire (on ne garde que le premier mot ayant cette traduction). Cette fonction ne renvoie rien, elle modifie le dictionnaire reçu en paramètre (effet de bord). *Indice: on a besoin de parcourir la liste des mots plutôt que les clés du dictionnaire, car on n'a pas le droit de modifier le dictionnaire pendant qu'on le parcourt.*
4. Écrire une fonction `inverseDico` qui reçoit en paramètre un dictionnaire français-anglais, et l'inverse pour créer un dictionnaire anglais-français (*on suppose qu'il n'y a pas plusieurs mots ayant la même traduction*).
5. Écrire un programme principal qui utilise les fonctions précédentes pour :
  - Créer un dictionnaire nommé `fr2en` associant des mots français (saisis par l'utilisateur) à leur traduction anglaise (saisie par l'utilisateur).
  - Créer un dictionnaire nommé `en2fr` en inversant le dictionnaire ci-dessus. On prendra soin de supprimer les doublons avant l'inversion.
  - Demander à l'utilisateur de saisir un mot et une langue ('en' ou 'fr'), puis utiliser le bon dictionnaire pour traduire le mot dans l'autre langue, et afficher le résultat.
  - Lui proposer de rejouer avec un autre mot et langue, jusqu'à ce qu'il refuse de continuer.
  - Afficher 'fin du programme' avant de se terminer.

# Projet 2016 : la réussite des alliances

## 1 Organisation du projet

Pour ce projet, vous vous inscrirez en binômes sur CaseInE. Les deux membres d'un binôme doivent être dans le même groupe de TD.

Le projet sera partiellement évalué sur CaseInE mais la note finale sera donnée par votre chargé de TP. CaseInE permettra surtout de valider un certain nombre de fonctions.

Date limite de rendu sur CaseInE : vendredi 28 avril 2017 à 18h59. Vous devrez compléter le fichier `student.py` de l'activité associée, et rendre un rapport au format pdf. Ce rapport, d'un maximum de deux pages donnera le mode d'emploi de la fonction 4.5.2, précisera les extensions que vous avez faites, et présentera de manière synthétique les résultats expérimentaux obtenus : moyennes, maximum et minimum des valeurs obtenues (en précisant le nombre de simulations), graphiques divers que vous aurez générés avec votre programme. Ce compte-rendu sera aussi déposé sur CaseInE, à la même date de rendu que le programme.

Votre programme devra être clair et commenté. Toutes les fonctions (y compris celles non demandées) devront être documentées.

Le rendu et les inscriptions en binômes sur CaseInE seront disponibles aux alentours du 3 avril.

## 2 Le thème du projet : la réussite des alliances

Dans ce projet, nous nous intéressons à une réussite peu connue qui se joue avec un jeu de 32 ou de 52 cartes : la réussite des alliances. On commence par donner les règles de cette réussite.

On mélange les cartes, on obtient un tas de cartes qui va constituer la pioche. On prend les cartes une à une dans la pioche et on les pose côte à côte de gauche à droite. Quand on a placé trois cartes, on considère les cartes 1 et 3 (en les numérotant de gauche à droite). Si elles sont de même couleur ou de même valeur (ce qu'on appellera une alliance), on prend la carte située entre elles (ici la carte numéro 2) et on la pose sur la carte 1 (sinon on ne fait rien et on continue à piocher). C'est ce qu'on va appeler un "saut". On n'a donc potentiellement plus que deux tas.

Par exemple, si les 3 premières cartes piochées sont dans l'ordre un 8 de cœur, un 7 de carreau et un roi de trèfle, aucun saut n'est possible :

8♥ 7♦ R♣

Par contre, si les 3 premières cartes piochées sont dans l'ordre un 8 de cœur, un 7 de carreau et un roi de cœur :

8♥ 7♦ R♥

on devra faire "sauter" le 7 de carreau sur le 8 de cœur avant de piocher la carte suivante, et on obtiendra :

7♦ R♥

On continue alors à piocher des cartes et à les poser une par une sur la droite de la réussite. Dès qu'il y a une alliance entre la dernière carte posée et l'antépénultième, le tas placé entre les deux est décalé et posé sur le tas à sa gauche ("saut"). Ainsi à tout moment de la partie, on a des tas de cartes, posés de gauche à droite. Pour un tas donné, le nombre de cartes du tas importe peu dans la mesure où il n'y a que la carte visible sur le dessus du tas qui compte pour la suite de la partie. On parlera donc de tas, même si le tas en question est composé d'une seule carte.

Par exemple, si on continue la partie suivante, en supposant qu'on pioche successivement l'as de cœur, le 10 de carreau et le 9 de pique, aucun saut ne sera possible, et on aura :

7♦ R♥ A♥ 10♦ 9♠

On pioche donc la carte suivante, un 9 de carreau :

7♦ R♥ A♥ 10♦ 9♠ 9♦

Ici un saut est possible car le tas du 9 de pique est situé entre deux carreaux, et on pose donc le tas du 9 de pique sur le tas du 10 de carreau :

7♦ R♥ A♥ 9♠ 9♦

Parfois, lorsqu'on fait un saut, cela rend d'autres sauts possibles, ce n'est pas le cas ici. Si au moins un autre saut est possible, il faut le faire avant de piocher la carte suivante. Si plusieurs sauts sont possibles, il faut décider lequel on fait en premier. On décide de faire toujours en premier le saut le plus à gauche (ce qui peut en entraîner un autre encore plus à gauche).

Pour illustrer ces sauts en cascade, imaginons par exemple qu'à un moment de la partie, on ait le jeu suivant :

9♣ V♦ 10♥ A♠ V♣ D♣ D♠

Ici, il n'y a toujours aucun saut possible. On pioche la carte suivante, un 7 de trèfle, et on la place à droite :

9♣ V♦ 10♥ A♠ V♣ D♣ D♠ 7♣

Ce 7 de trèfle est un trèfle tout comme la carte situé deux rangs avant lui, donc on prend le tas de la dame de pique qui est entre le 7 de trèfle et la dame de trèfle et on le pose sur le tas de la dame de trèfle. Après ce saut, on obtient :

9♣ V♦ 10♥ A♠ V♣ D♠ 7♣

Ici c'est intéressant car on a deux sauts possibles : on peut faire sauter le tas du valet de trèfle ou le tas de la dame de pique. Mais si on pose d'abord la dame de pique sur le valet de trèfle on ne pourra plus faire l'autre saut, d'où l'idée de toujours commencer par le saut le plus à gauche. Donc on fait sauter la carte (ou le tas de cartes) situé entre l'as de pique et la dame de pique, et c'est donc le tas du valet de trèfle qui sera posé sur l'as de pique :

9♣ V♦ 10♥ V♣ D♠ 7♣

Maintenant, avant de faire l'autre saut qu'on avait repéré, il faut vérifier s'il n'y en a pas un nouveau plus à gauche. On repart du début du jeu à gauche pour chercher les sauts possible. Ici un nouveau saut est apparu entre le valet de carreau et le valet de pique. On pose donc le tas du 10 de cœur sur le valet de carreau :

9♣ 10♥ V♣ D♠ 7♣

Le prochain saut à faire est de poser le tas du 10 de cœur sur le 9 de trèfle :

10♥ V♣ D♠ 7♣

Et maintenant le premier saut à faire est celui que nous avons repéré il y a quelques étapes déjà : on pose le tas de la dame de pique sur le valet de trèfle.

10♥ D♠ 7♣

Ainsi sans piocher d'autre carte après le 7 de trèfle, on est passé d'un jeu à 8 tas à un jeu à 3 tas.

Ici aucun autre mouvement n'est possible. On pioche la carte suivante et on continue la réussite.

La partie s'arrête quand on a épuisé la pioche et qu'on ne peut plus faire diminuer le nombre de tas. On compte alors le nombre de tas restants à la fin de la réussite. On a gagné si le nombre de tas est inférieur à un certain seuil fixé par la règle du jeu. On testera plusieurs seuils dans ce projet. On a toujours au moins 2 tas à la fin (et le dernier n'a qu'une carte), c'est la fin "parfaite" mais c'est suffisamment rare pour qu'on soit plus tolérant pour estimer que la réussite est gagnée, surtout si on joue avec 52 cartes.

### 3 Représentation informatique de cette réussite

Dans ce projet, on va programmer cette règle de réussite et faire plusieurs études pour estimer par exemple la probabilité de gagner une partie.

Pour modéliser cette réussite, on va considérer qu'à tout moment de la partie, la réussite est modélisée par une liste de cartes qui sont les cartes visibles sur les dessus des tas. Ainsi le nombre de tas à la fin de la partie sera tout simplement le nombre d'éléments de cette liste.

Pour coder une carte dans le programme, on va utiliser un dictionnaire à deux clés : `'valeur'` et `'couleur'`.

Pour la valeur de la carte, un as sera représenté par `'A'`, un deux par l'entier 2, un trois par l'entier 3 et un dix par 10. Pour les figures, on écrira respectivement `'V'`, `'D'` et `'R'` pour valet, dame et roi.

Pour la couleur, on aimerait utiliser l'initiale majuscule de chaque couleur (Cœur, Pique, Carreau et Trèfle) mais comme cœur et carreau ont la même initiale, on choisit la lettre `'K'` pour représenter carreau.

Ainsi le dictionnaire `{ 'valeur':7, 'couleur': 'P' }` représentera le 7 de pique.

La liste `[{ 'valeur':7, 'couleur':'P' }, { 'valeur':10, 'couleur':'K' }, { 'valeur':10, 'couleur':'D' }, { 'valeur':10, 'couleur':'K' }]` correspondra à l'état suivant de la réussite :

7♠ 10♦ D♦

La pioche sera également modélisée par une liste de cartes, la carte à l'indice 0 étant la première carte à piocher.

On va maintenant pouvoir écrire les fonctions permettant de faire jouer l'ordinateur à cette réussite.

## 4 Travail demandé

Dans cette partie, on a réparti les différentes fonctions à écrire dans plusieurs catégories. Autant que possible, testez vos fonctions au fur et à mesure que vous les écrivez.

**Attention :** pour toutes les fonctions demandées dans ce projet, quand il y a des listes en argument, ces listes ne doivent pas être modifiées par la fonction, sauf si c'est explicitement demandé. N'hésitez pas à faire une copie de la liste si nécessaire.

### 4.1 Affichage de la réussite

#### 4.1.1 Fonction `carte_to_chaine`

Pour afficher la réussite sur la console Python, on souhaite avoir un affichage plus explicite que `"DK"` par exemple pour la dame de carreau. On va afficher les cartes de façon assez similaire à ce qui est fait dans les exemples de la partie 2.

Pour cela, on doit savoir représenter les caractères spéciaux ♡ ♠ ♦ ♣. En fait, en Python, chaque caractère correspond à un entier qui code ce caractère. Pour savoir quel est le code de la lettre `'a'` par exemple, il faut faire afficher `ord(a)`. Vous trouverez alors que l'entier associé au caractère `'a'` est 97. Réciproquement, pour obtenir le caractère correspondant à un entier donné, il faut utiliser la fonction `chr`. Ainsi par exemple `chr(97)`

retournera 'a'. Voici les codes qui vous seront utiles :

Caractère	Code
♠	9824
♥	9825
♦	9826
♣	9827

Ainsi, par exemple, pour créer la chaîne de caractères permettant d'afficher la dame de carreau, il faudra faire : 'D' + chr(9826)

Ecrire la fonction `carte_to_chaine` qui prend en argument un dictionnaire représentant une carte, et retourne une chaîne de caractères permettant l'affichage de la carte correspondante, sur 3 caractères, avec éventuellement un espace à gauche. Un espace est nécessaire pour toutes les cartes qui ne sont pas un 10. Attention, cette fonction ne produit aucun affichage.

Exemples : (les espaces ont été matérialisés par le caractère `_`)

- `carte_to_chaine({'valeur':7, 'couleur':'P'})` retourne la chaîne `"_7♠"`
- `carte_to_chaine({'valeur':10, 'couleur':'K'})` retourne la chaîne `"10♦"`
- `carte_to_chaine({'valeur':'R', 'couleur':'C'})` retourne la chaîne `"_R♥"`

#### 4.1.2 Fonction `afficher_reussite`

Ecrire une fonction `afficher_reussite` qui prend en argument une liste de cartes correspondant à l'état de la réussite à un moment donné et affiche la réussite. On utilisera la fonction `carte_to_chaine`. On affichera les différentes cartes sur une même ligne, séparées par un espace (en plus de celui éventuellement mis par `carte_to_chaine`).

Exemple :

`afficher_reussite([{'valeur':7, 'couleur':'P'}, {'valeur':10, 'couleur':'K'}, {'valeur':'A', 'couleur':'T'}])` affichera `"_7♠_10♦_A♣"` suivi d'un saut de ligne (2 retours à la ligne).

## 4.2 Entrées / Sorties avec des fichiers

### 4.2.1 Fonction `init_pioche_fichier`

Cette fonction lit une suite de cartes dans le fichier dont le nom est passé en argument, et renvoie la liste de cartes correspondante. On suppose que les valeurs dans le fichier sont correctes, c'est-à-dire qu'elles correspondent toutes à une carte existante, et qu'il y a le bon nombre de cartes (32 ou 52). Dans le fichier, on a la liste des cartes dans l'ordre où elles seront piochées, les cartes étant séparées par des espaces. Pour chaque carte, il est écrit sa valeur puis sa couleur, séparées par un tiret. Par exemple, si on a "7-K", cela correspondra au 7 de carreau et "D-P" représentera la dame de pique. Pour cette question, un fichier d'exemple de pioche `data_init.txt` est fourni sur CaseInE.

### 4.2.2 Fonction `ecrire_fichier_reussite`

On souhaite pouvoir sauvegarder dans un fichier l'ordre des cartes de la pioche d'une réussite donnée, par exemple pour mémoriser le mélange qui a donné le meilleur résultat parmi plusieurs mélanges.

Ecrire pour cela la fonction `ecrire_fichier_reussite` qui prend en argument un nom de fichier `nom_fich` dans lequel il faut écrire la liste des cartes `pioche` fournie en deuxième argument de la fonction. Le fichier ainsi écrit doit pouvoir être lu par la fonction `init_pioche_fichier`. Cette fonction ne renvoie rien.

### 4.3 Générer une pioche mélangée aléatoirement

Ecrire la fonction `init_pioche_alea`. Cette fonction a un seul argument, optionnel, nommé `nb_cartes` qui vaut 32 par défaut, mais qui peut aussi valoir 52. Cette fonction renvoie une liste contenant toutes les cartes du jeu, et les mélange (pour mélanger une liste, vous pourrez utiliser la fonction `shuffle` du module `random`).

### 4.4 Programmer les règles de la réussite des alliances

#### 4.4.1 Fonction `alliance(carte1, carte2)`

Ecrire une fonction `alliance(carte1, carte2)` qui prend en argument deux cartes, et qui renvoie vrai si les cartes ont la même valeur ou la même couleur et faux sinon.

Exemples :

- `alliance({'valeur':7, 'couleur':'P'}, {'valeur':7, 'couleur':'C'})` renvoie vrai
- `alliance({'valeur':7, 'couleur':'P'}, {'valeur':8, 'couleur':'C'})` renvoie faux.

#### 4.4.2 Fonction `saut_si_possible`

Cette fonction prend en argument une liste `liste_tas` correspondant à la liste des cartes visibles sur les tas de la réussite et un entier `num_tas`. La fonction vérifie s'il est possible de faire sauter le tas d'indice `num_tas` sur le tas qui le précède (selon les règles de la réussite). Si le saut est possible, la fonction le fait. La fonction retourne vrai si le saut a été fait et faux sinon. Elle n'affiche rien.

**Attention :** cette fonction modifie donc la liste qui lui est donnée en argument.

#### 4.4.3 Fonction `une_etape_reussite`

Cette fonction prend en argument une liste `liste_tas` correspondant aux cartes visibles des tas de la réussite, une liste `pioche` contenant les cartes restant dans la pioche ainsi qu'un argument optionnel booléen `affiche` valant `False` par défaut. Cette fonction doit effectuer une étape de la réussite, c'est-à-dire :

- tirer la première carte de `pioche` et la placer dans `liste_tas` pour qu'elle corresponde au tas le plus à droite de la réussite,
- faire le saut s'il est possible entre la carte qu'on vient de poser et le tas situé deux rangs plus à gauche (c'est-à-dire faire sauter la carte entre ces deux tas),
- si un saut a été fait, il faudra vérifier si cela a rendu possible d'autres sauts. Pour cela on partira de la gauche de `liste_tas` et on fera le premier saut possible. On recommencera cette action tant qu'un changement aura lieu.
- en plus de tout ça, si l'argument `affiche` vaut vrai, la fonction affichera chacun des états de la réussite pour l'étape en cours. C'est-à-dire qu'à chaque changement (pioche ou saut) il faudra faire un affichage de la réussite. Et si `affiche` vaut faux, rien ne sera affiché.

**Attention :** cette fonction ne retourne rien mais elle modifie les deux listes qui lui sont données en argument (car on a enlevé la première carte de la `pioche` pour la mettre dans `liste_tas`).

Il est recommandé d'utiliser des fonctions auxiliaires pour programmer correctement cette fonction.



## 4.5 Faire une partie

### 4.5.1 Fonction `reussite_mode_auto`

Cette fonction prend en argument une liste de cartes `pioche` correspondant à l'ensemble du jeu de cartes déjà mélangé et un booléen optionnel `affiche` ayant les mêmes spécifications qu'à la question précédente. Si `affiche` vaut vrai, on commence par afficher le contenu de la pioche avant de jouer la réussite en affichant les étapes. Cette fonction joue l'ensemble de la réussite en affichant chaque étape si `affiche` vaut vrai. Le second affichage correspond alors à la première carte de la pioche retournée. La fonction ne doit pas modifier la liste `pioche` (vous pouvez en faire une copie) et elle doit retourner la liste des cartes visibles sur les tas restant à la fin de la réussite.

### 4.5.2 Fonction `reussite_mode_manuel`

On souhaite maintenant laisser l'utilisateur prendre lui-même les décisions comme si c'était lui qui jouait quitte à ce qu'il rate des sauts, mais sans le laisser tricher.

Pour cela, écrire une fonction `reussite_mode_manuel` qui, comme la fonction précédente, prend en argument une liste de cartes `pioche` correspondant à l'ensemble du jeu de cartes déjà mélangé. Ici, il n'y a pas d'argument `affiche` car on veut forcément afficher les étapes pour que le joueur puisse voir où il en est. Par contre, la fonction prendra un argument optionnel `nb_tas_max` (par défaut, réglé à 2) qui fixe le nombre maximum de tas qu'il peut rester à la fin pour que la partie soit déclarée comme gagnante. L'ordinateur proposera un menu au joueur : il lui proposera par exemple de découvrir une carte de la pioche, ou de saisir un saut à faire. Il faudra aussi que le joueur puisse quitter s'il le souhaite. Si le joueur demande à faire un saut non autorisé, l'ordinateur doit refuser de le faire en précisant que le coup est impossible.

Si le joueur abandonne avant d'avoir retourné toutes les cartes, les cartes restant dans la pioche sont découvertes une à une et posées sur la réussite, sans qu'aucun saut ne soit fait. Dans tous les cas, à la fin, la fonction affiche au joueur s'il a gagné ou perdu, et retourne la liste des cartes visibles sur les tas restant à la fin de la réussite.

Cette question est volontairement assez ouverte. A vous de trouver le format qui rendra le jeu le plus agréable, et qui proposera éventuellement d'autres options.

### 4.5.3 Fonction `lance_reussite`

Écrire une fonction `lance_reussite` qui prend en argument une chaîne de caractères `mode` qui vaudra soit `'manuel'` soit `'auto'`, ainsi que trois arguments optionnels `nb_cartes` (par défaut, 32) et un booléen `affiche` (par défaut, False) et un entier `nb_tas_max` (par défaut, 2). Cette fonction doit mélanger aléatoirement un jeu contenant le bon nombre de cartes et lancer la réussite en mode manuel ou en mode auto. L'argument `affiche` ne sert que pour le mode auto, et l'argument `nb_tas_max` ne sert que pour le mode manuel. La fonction doit renvoyer la liste des tas restants à la fin de la partie.

## 4.6 Extensions

Dans cette partie, on propose un certain nombre d'extensions ou d'améliorations possibles. Vous n'êtes pas obligés de toutes les faire mais plus vous réussirez à en faire, meilleure sera votre note. Ici, les instructions sont moins directives que dans la partie précédente, à vous de prendre des décisions pour faire au mieux. Vous pouvez aussi imaginer d'autres extensions.

Pour chaque extension faite, vous ajouterez dans le programme principal du code très bien commenté permettant d'illustrer au mieux cette nouvelle fonctionnalité et vous préciserez ce qu'elle fait et les résultats qu'elle donne dans votre compte-rendu.

#### 4.6.1 Fonction `verifier_pioche`

Jusqu'à maintenant, on n'a jamais vérifié si la liste des cartes donnée comme pioche en entrée des fonctions `reussite_mode_auto` ou `reussite_mode_manuel` correspondait bien à un vrai jeu de cartes non truqué : pas de carte en double, toutes les cartes présentes.

La fonction `verifier_pioche` prend en argument la liste des cartes constituant la pioche et un entier `nb_cartes` qui vaut 32 ou 52 (argument optionnel, 32 par défaut) et elle vérifie si cette pioche contient bien toutes les cartes d'un jeu à `nb_cartes`, sans doublon. Elle renvoie vrai si c'est le cas et faux sinon.

#### 4.6.2 Estimer le nombre moyen de tas

- Ecrire la fonction `res_multi_simulation` qui prend en argument un nombre entier `nb_sim` et en argument optionnel un entier `nb_cartes` correspondant au nombre de cartes dans le jeu (32 par défaut). La fonction génère `nb_sim` mélanges aléatoires d'une pioche d'un jeu de `nb_cartes` cartes, effectue les réussites en mode automatique, sans affichage, et retourne la liste des nombres de tas obtenus à la fin de chaque réussite. (par exemple, si `nb_sim` vaut 3, la fonction retourne [ 5, 3, 12 ] si la première réussite s'est finie avec 5 tas, la deuxième avec 3 tas et la troisième avec 12 tas).
- Ecrire la fonction `statistiques_nb_tas` qui prend en argument un nombre entier `nb_sim` et en argument optionnel un entier `nb_cartes` correspondant au nombre de cartes dans le jeu (32 par défaut). La fonction calcule et affiche avec des messages clairs la moyenne, le minimum et le maximum du nombre de tas obtenus sur `nb_sim` réussites automatiques correspondant à `nb_sim` mélanges aléatoires de la pioche. Vous pouvez également afficher d'autres indicateurs qui vous sembleront pertinents.

Pour cette partie, n'oubliez pas de résumer les résultats numériques obtenus dans votre compte-rendu, en précisant le nombre de simulations et le nombre de cartes du jeu.

#### 4.6.3 Estimer la probabilité de gagner et faire un graphique

Dans les règles officielles de cette réussite, on a gagné si on a seulement deux tas à la fin. Vous avez dû vous rendre compte que cela arrive assez rarement, surtout quand on a 52 cartes. On souhaiterait pouvoir quantifier ce "assez rarement" et voir comment augmentent nos chances de gagner si on autorise plus de tas (par exemple si on décide que finalement une réussite sera gagnante si elle se finit avec au maximum 4 tas). Ces probabilités semblent difficiles à obtenir par un calcul théorique, par contre, il est relativement aisé de les estimer grâce à des simulations.

En faisant un nombre important de simulations de mélange aléatoire, faites en sorte d'estimer la probabilité de gagner pour chaque nombre maximum de tas possiblement autorisé par la règle (entre 2 et 32 pour un jeu de 32 cartes).

Vous pouvez stocker ces probabilités dans une liste. Créez cette liste à partir d'une fonction.

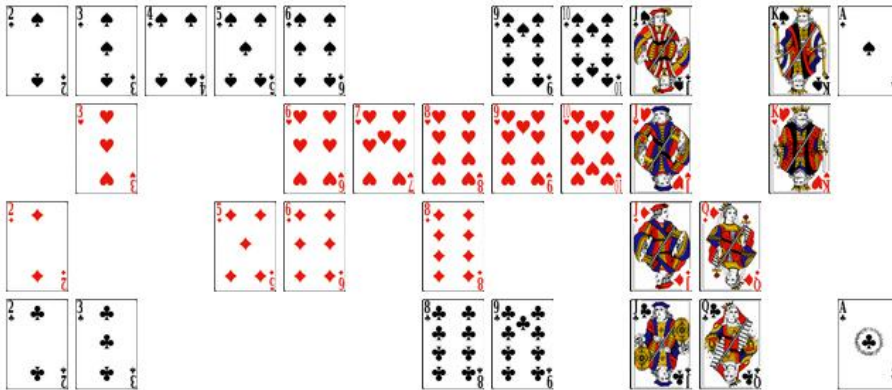
Pour mieux visualiser ces données, vous pouvez faire un graphique en utilisant le module `pyplot` de la librairie `matplotlib`. Pour éviter d'avoir à écrire le nom `matplotlib.pyplot` devant chaque fonction de ce module que vous utiliserez, vous pouvez l'importer ainsi : `import matplotlib.pyplot as plt` et ainsi il suffira d'écrire `plt` devant chaque fonction utilisée. Pour visualiser la fenêtre graphique, après avoir donné à l'ordinateur toutes les instructions pour créer le graphique, il ne faudra pas oublier de faire `plt.show()`. Vous trouverez la documentation de `pyplot` ici :

[http://matplotlib.org/api/pyplot\\_api.html](http://matplotlib.org/api/pyplot_api.html) ou tapez *pyplot tutorial* sans un moteur de recherche

#### 4.6.4 Interface graphique avec Turtle

Récupérez le fichier `affichage.py` ainsi que le répertoire `imgs` (qui contient les images des cartes). Exécutez et observez ce programme d’affichage.

En vous inspirant de ce programme, proposez un affichage graphique de la réussite en cours d’exécution.



#### 4.6.5 Améliorer un mélange en trichant ”un peu”

Si vous avez un peu pratiqué cette réussite, vous vous serez rendus compte que l’on a souvent l’impression qu’à pas grand-chose près, en trichant juste un peu, en échangeant discrètement deux cartes consécutives qui ne sont vraiment pas arrivées dans le bon ordre, le résultat aurait été très différent. On va tester si cette impression est justifiée.

Ecrivez une fonction `meilleur_echange_consecutif` qui prend en paramètre la liste des cartes correspondant à la pioche et cherche la meilleure façon d’améliorer cette réussite en faisant un seul échange, entre deux cartes consécutives. Cette fonction a deux valeurs de retour : elle renvoie la liste des cartes correspondant à la pioche donnée en entrée dans laquelle on a fait le meilleur échange possible entre deux cartes consécutives et elle renvoie aussi le nombre de tas que nous fait gagner cet échange (3 par exemple si la réussite initiale donnait 7 tas à la fin et que celle après avoir fait le meilleur échange donne 4 tas à la fin). Si plusieurs échanges conduisent au résultat optimal (nombre de tas minimum), on choisira de faire l’échange qui est le plus loin dans la pioche (le plus grand indice dans la liste `pioche`).

Etudiez en quoi ça améliore les statistiques de vos parties (nombre moyen, minimum et maximum de tas).

Vous pouvez éventuellement générer de nouveaux graphiques d’estimations de probabilité quand on s’autorise un tel échange de deux cartes dans la pioche avant la partie.

#### 4.6.6 Améliorer un mélange en trichant un peu plus

Sur le modèle de l’extension précédente, à vous d’imaginer d’autres façons de modifier un peu l’ordre de la pioche pour améliorer le score à la réussite. Etudiez si ça améliore beaucoup les résultats.

# INF101 : Projet - Jeu de Uno

Dans ce projet on se propose de réaliser un jeu de Uno. Certaines fonctions sont guidées, mais vous pouvez (et devez) aussi laisser libre cours à votre imagination pour ajouter des fonctionnalités, une interface, etc.

## 1 Partie A - gestion du paquet de cartes et des mains des joueurs

### 1.1 Les cartes du jeu

Le paquet de cartes du Uno contient les 108 cartes, réparties dans 4 couleurs différentes (rouge, bleu, jaune, vert) sauf 8 cartes multicolores. Dans chaque couleur il y a 19 cartes numérotées (une carte 0 et deux cartes pour chaque chiffre de 1 à 9), et 6 cartes spéciales : 2 cartes "+2", 2 cartes "Changement de sens", 2 cartes "Passe ton tour". Les 8 cartes multicolores sont : 4 cartes "Joker", et 4 cartes "+4". On représentera chacune de ces cartes par un entier entre 1 et 108 (qu'on appellera indice de la carte dans le paquet), et le paquet par une liste d'entiers.

### 1.2 Initialisation et mélange du paquet

Écrire une fonction qui initialise le paquet en mélangeant les 108 entiers (entre 1 et 108).

**Fonctions utiles :** range, random.randint

### 1.3 Conversions

Écrire des fonctions de conversion :

- une fonction pour obtenir la couleur d'une carte donnée par son indice (entre 1 et 108),
- une fonction qui permet d'obtenir le numéro d'une carte (entre 0 et 9 pour les chiffres, 10 pour le +2, 11 pour le "change sens", 12 pour le "passe tour") donnée par son indice dans le paquet (entre 1 et 108).
- une fonction qui reçoit l'indice d'une carte et renvoie le texte complet correspondant (par exemple "2 bleu" ou "change tour jaune" ou "Joker")

### 1.4 Pioche

Écrire une fonction qui reçoit une liste (le paquet), y pioche une carte (ce qui doit modifier la liste) et renvoie la carte piochée.

### 1.5 Main du joueur

On représente maintenant la main d'un joueur (les cartes qu'il a en main) par une liste d'indices de cartes (liste d'entiers entre 1 et 108).

Écrire une fonction piocher qui reçoit 3 arguments (la pioche, la main du joueur, le nombre de cartes à piocher) et qui modifie les 2 listes en faisant piocher ce nombre de cartes (qui doivent donc être supprimées de la pioche et ajoutées à la main).

## 1.6 Pioche vide ?

Le jeu ne s'arrête pas quand la pioche est vide, mais on mélange la pile de jeu pour reconstituer la pioche. Écrire une fonction qui reçoit en argument la pioche et la pile de jeu, et mélange la pile de jeu dans la pioche. Attention la dernière carte reste au-dessus du jeu pour savoir quoi jouer, il ne faut pas la remettre dans la pioche.

Cette fonction sera appelée quand la pioche ne contient plus assez de cartes pour piocher le nombre de cartes voulues.

## 2 Partie B - les règles de base du jeu

Les joueurs jouent à tour de rôle une carte de leur main sur la pile centrale. On représente le jeu par une liste, dont la dernière carte est celle visible sur la table.

Les règles du jeu précisent quelle carte peut être jouée (par le joueur dont c'est le tour) selon celle qui est en haut de la pile:

- une carte de la même couleur (n'importe quel numéro, un "+2", "passe tour" ou "change sens") que la carte courante
- une carte identique à la carte courante (même numéro ou type de carte spéciale) de n'importe quelle couleur
- un Joker sur n'importe quelle carte de n'importe quelle couleur, sauf sur un +2 et un +4
- un +4 sur n'importe quelle autre carte

### 2.1 Vérification des règles

Écrire une ou des fonctions pour vérifier si on peut poser une carte sur la carte courante, selon les différentes règles ci-dessus.

### 2.2 Joue ou pioche

Si un joueur ne peut poser aucune carte à son tour de jeu, il doit à la place piocher une carte.

Écrire une fonction qui vérifie si un joueur peut jouer (au moins une de ses cartes est jouable), et éventuellement une fonction qui renvoie l'indice de la (première) carte jouable (ou -1 si aucune).

Un cas particulier se produit si la carte centrale est un +2 ou un +4 : si le joueur ne peut pas jouer il doit piocher le nombre de cartes égal à la somme de tous les +2 et +4 de la séquence en cours. Par exemple si 3 cartes +2 et une carte +4 ont été posées et que le joueur n'a aucune carte à poser dessus, il doit piocher  $3*2+4=10$  cartes. Pour gérer ce cas écrire une fonction qui renvoie le nombre de cartes à

piocher par le joueur selon la carte centrale et sa main. Si ce nombre vaut 0 cela signifie que le joueur peut jouer, sinon il doit piocher.

### 2.3 Choix de la carte à jouer

Écrire une fonction qui demande au joueur quelle carte il veut jouer, filtre jusqu'à ce que la carte choisie soit bien jouable selon les règles, et renvoie la carte choisie.

**Attention** il ne faudra appeler cette fonction que s'il y a bien une carte jouable dans la main du joueur, sinon elle ferait une boucle infinie.

## 3 Partie C - multi-joueur et tours de jeu

On veut maintenant gérer les tours de jeu pour savoir quel joueur doit jouer en prochain. Par défaut le sens de rotation se fait dans le sens des aiguilles d'une montre, mais les cartes "change sens" peuvent modifier la rotation. Il faut donc enregistrer le sens de rotation courant et le modifier selon la carte jouée.

### 3.1 Gestion des joueurs

Il faut une structure de données pour enregistrer les mains de tous les joueurs. L'utilisateur doit aussi pouvoir saisir le nombre de joueurs au départ, et le nombre de cartes dans la main de départ.

On propose d'utiliser un dictionnaire pour stocker pour chaque joueur son nom et le contenu de sa main.

Écrire une fonction d'initialisation en début de partie, qui demande à l'utilisateur le nombre de joueur, lui demande le nom de chaque joueur, lui demande le nombre de cartes dans les mains de départ, puis initialise les mains de chaque joueur avec le bon nombre de cartes et les enregistre dans le dictionnaire.

### 3.2 Sens de rotation et prochain joueur

Écrire une fonction qui détermine le numéro ou le nom du joueur dont c'est le tour à partir du numéro ou du nom du joueur précédent, du sens de rotation courant, et de la dernière carte jouée.

Il faudra donc gérer les effets des cartes spéciales :

- si la dernière carte est un changement de sens, il faut mémoriser le nouveau sens de rotation avant de passer au joueur suivant dans ce sens modifié
- si la dernière carte est un passe-tour, il faut sauter le joueur suivant et passer à celui d'après

### 3.3 Test de victoire et fin de partie

Un joueur gagne quand il n'a plus aucune carte en main. Écrire une fonction qui teste si le joueur courant (reçu en argument) a gagné, et renvoie un booléen.

### 3.4 Tour de jeu

Écrire une fonction qui reçoit en particulier le numéro du joueur dont c'est le tour, et réalise les étapes suivantes qui constituent son tour :

- teste s'il peut jouer ou combien de cartes il doit piocher
- s'il doit piocher : vérifie s'il reste assez de cartes dans la pioche, sinon y mélange la pile de jeu, avant de le faire piocher
- s'il peut jouer : lui demande quelle carte il veut jouer, et la joue (et applique éventuellement les effets de la carte)
- vérifie si le joueur a gagné ou détermine le numéro du prochain joueur, et renvoie cette information

### 3.5 Programme principal

Écrire un programme principal qui gère une partie complète jusqu'à ce qu'un joueur ait gagné, et affiche le nom du gagnant et les mains restantes des autres joueurs.

## 4 Partie D - interface graphique et bonus

En utilisant par exemple le module turtle, fournir une interface graphique qui affiche entre autres :

- la carte centrale sur laquelle on joue
- le joueur courant et sa main
- le sens de rotation
- des instructions éventuelles (tel joueur pioche tant de cartes)
- des indications pour aider le joueur (quelles cartes il peut jouer)

Pour aller plus loin on peut aussi :

- permettre au joueur de jouer en cliquant dans l'interface plutôt qu'en tapant dans la console
- écrire une IA qui joue toute seule, et proposer un menu permettant de définir le nombre de joueurs, de demander leur nom, et de dire lesquels sont joués par l'IA ou par un joueur humain.





## Chapitre 5

## Annexes

# Mémo Python - UE INF101 / INF131 / INF204

## Opérations sur les types

`type()` : pour connaître le type d'une variable  
`int()` : transformation en entier  
`float()` : transformation en flottant  
`str()` : transformation en chaîne de caractères

## Définition d'une fonction

```
def nomFonction(arg) :  
    instructions  
    return v
```

Fonction qui renvoie la valeur ou variable `v`.

## Infini

`float('inf')` : valeur infinie positive ( $+\infty$ )  
`float('-inf')` : valeur infinie négative ( $-\infty$ )

## Écriture dans la console

```
print(a1,a2,...,an, sep=xx, end=yy)
```

- Pour imprimer une suite d'arguments de `a1` à `an`
- `sep` : permet de définir le séparateur affiché entre chaque argument (optionnel, par défaut " ")
- `end` : permet de définir ce qui sera affiché à la fin (optionnel, par défaut : saut de ligne)

## Lecture dans la console

```
res = input(message)
```

- Pour lire une suite de caractères au clavier terminée par `< Enter >`
- Ne pas oublier de transformer la chaîne en entier (`int`) ou réel (`float`) si nécessaire.
- La chaîne de caractères résultante doit être affectée (ici à la variable `res`).
- L'argument est optionnel : c'est un message explicatif destiné à l'utilisateur

## Opérateurs booléens

`and` : et logique  
`or` : ou logique  
`not` : négation

## Opérateurs de comparaison

<code>==</code> : égalité	<code>!=</code> : différence
<code>&lt;</code> : inférieur,	<code>&lt;=</code> : inférieur ou égal
<code>&gt;</code> : supérieur,	<code>&gt;=</code> : supérieur ou égal

## Instructions conditionnelles

```
if condition :  
    instructions
```

```
if condition :  
    instructions  
else :  
    instructions
```

```
if condition1 :  
    instructions  
elif condition2 :  
    instructions  
else :  
    instructions
```

## Opérateurs arithmétiques

<code>+</code> : addition,	<code>-</code> : soustraction
<code>*</code> : multiplication,	<code>**</code> : puissance,
<code>/</code> : division,	<code>//</code> : quotient div entière,
<code>%</code> : reste de la division entière (modulo)	

## Caractères

`ord(c)` : renvoie le code ASCII du caractère `c`  
`chr(a)` : renvoie le caractère de code ASCII `a`

## Chaînes de caractères

`len(s)` : renvoie la longueur de la chaîne `s`  
`s1+s2` : concatène les chaînes `s1` et `s2`  
`s*n` : construit la répétition de `n` fois la chaîne `s`  
exemple : `"ta"*3` donne `"tatata"`  
`list(chaine)` : renvoie la liste des caractères de la chaîne  
`ch.split(arg)` : retourne la liste des sous-chaînes de `ch`, en coupant à chaque occurrence de `arg` (par défaut `arg=" "`)  
`ch.join(liste)` : concatène les chaînes de `liste`, en utilisant `ch` comme séparateur, et renvoie la chaîne résultante  
`ch.upper()` : passe `ch` en majuscules  
`ch.lower()` : passe `ch` en minuscules

## Itération tant que

```
while condition :  
    instructions
```

## Itération for, et range

```
for e in conteneur :  
    instructions
```

```
for var in range (deb, fin, pas) :  
    instructions
```

Itère les instructions avec `e` prenant chaque valeur dans le conteneur (liste, chaîne ou dictionnaire) ; ou avec `var` prenant les valeurs entre `deb` et `fin` avec un `pas` donné.

`range(a)` : séquence des valeurs `[0, a[`  
`range (b,c)` : séquence des valeurs `[b, c[` (`pas=1`, `c > b`)  
`range (b, c, g)` : idem avec un `pas = g`  
`range(b,c,-1)` : valeurs décroissantes de `b` (incl.) à `c` (excl.), `pas=-1` (`c < b`)

## Listes

`maListe = []` : création d'une liste vide  
`maListe = [e1,e2,e3]` : création d'une liste, ici à 3 éléments `e1`, `e2`, et `e3`

`maListe[i]` : obtenir l'élément à l'index `i` (`i ≥ 0`).  
Les éléments sont indexés à partir de 0. Si `i < 0`, les éléments sont accédés à partir de la fin de la liste. Ex : `maListe[-1]` permet d'accéder au dernier élément de la liste

`maListe.append(elem)` : ajoute un élément à la fin  
`maListe.extend(liste2)` : ajout de tous les éléments de la liste `liste2` à la fin de la liste `maListe`  
`maListe.insert(i,elem)` : ajout d'un élément à l'index `i`

`res = maListe.pop(index)` : retire l'élément présent à la position `index` et le renvoie, ici dans la variable `res`  
`maListe.remove(element)` : retire l'élément donné (le premier trouvé)

`len(maListe)` : nombre d'éléments d'une liste  
`elem in maListe` : teste si un élément est dans une liste (renvoie `True` ou `False`)

`l2 = maListe` : crée un synonyme (2ème nom pour la liste)  
`l3 = list(maListe)` : crée une copie de surface (un clone)  
`l4 = copy.deepcopy(maListe)` : crée une copie profonde (récursive)

## Aléatoire

`random.randint(inf,sup)` : entier aléatoire entre bornes `inf` et `sup` incluses  
`random.shuffle(maListe)` : mélange la liste (effet de bord), ne renvoie rien  
`random.choice(maListe)` : renvoie un élément au hasard de la liste

## Dictionnaires

`monDico = {}` : création d'un dictionnaire vide  
`monDico = { c1:v1, c2:v2, c3:v3 }` : création d'un dictionnaire, ici à 3 entrées (clé `c1` avec valeur `v1`, etc)

`e = monDico[c1]` : les valeurs du dictionnaire sont accessibles par leurs clés. Ici, `e` prendra la valeur `v1`. Provoque une erreur si la clé n'existe pas.

`monDico[c3] = v3` : ajoute une nouvelle valeur au dictionnaire (ici `v3`) avec une clé (ici `c3`). Si la clé existe déjà, la valeur associée est modifiée.

`del monDico[C3]` : supprime une association dans le dictionnaire. La clé doit exister.

`c in monDico` : vérifie l'existence d'une clé dans le dictionnaire, renvoie `True` ou `False`.

`dic2 = monDico` : crée un synonyme (2ème nom au dico)  
`dic3 = dict(monDico)` : crée une copie de surface (clone)  
`dic4 = copy.deepcopy(monDico)` : crée une copie profonde (récursive)

## Gestion des fichiers

`f=open('data.txt')` : ouvrir un fichier en lecture seule  
`f=open('data.txt','w')` : ouvre un fichier en écriture (attention s'il existe il est écrasé, sinon il est créé)  
`f=open('data.txt','a')` : ouvre un fichier en écriture (ajoute le texte à la fin)

`texte = f.read()` : lire tout le fichier en une seule fois  
`lignes = f.readlines()` : lire en 1 fois toutes les lignes du fichier et les stocker dans une liste (un élém=une ligne)  
`for ligne in f:`  
    `instructions`  
Lire le fichier ligne par ligne dans une boucle `for`

`f.write(texte)` : écrire dans un fichier (`texte` doit obligatoirement être une `string`).  
Ne saute pas de ligne automatiquement à la fin du texte.  
'\n' code un saut de ligne.

`f.close()` : ferme un fichier