UNIVERSITÉ GRENOBLE ALPES

UFR IM²AG

DÉPARTEMENT LICENCE SCIENCES ET TECHNOLOGIE

LICENCE SCIENCES & TECHNOLOGIES, 1re ANNÉE

UE INF101, INF104, INF131

—

ALGORITHMIQUE ET PROGRAMMATION, EN PYTHON

Responsables d'UE : Carole Adam et François Puitg
carole.adam@univ-grenoble-alpes.fr
francois.puitg@univ-grenoble-alpes.fr

2023-2024

Table des matières

1 INTRODUCTION **1**

1.1 Comment sont notés vos programmes . . . 1
1.2 Bonnes pratiques . . . 2

2 RÉSUMÉ DE COURS **3**

2.1 Quelques définitions . . . 4
2.2 Variables, expressions, instructions . . . 6
2.3 Outils utiles . . . 10
2.4 Entrées / Sorties . . . 12
2.5 Les expressions booléennes . . . 15
2.6 Instructions conditionnelles 'if' . . . 18
2.7 La boucle conditionnelle 'while' . . . 22
2.8 Les fonctions . . . 31
2.9 Chaînes de caractères . . . 43
2.10 Listes . . . 50
2.11 Boucle inconditionnelle : for . . . 60
2.12 Listes avancées . . . 64
2.13 Fonctions : compléments . . . 65
2.14 Dictionnaires . . . 71
2.15 Introduction aux fichiers . . . 77

3 EXERCICES DE TD **81**

3.1 Les bases . . . 82
3.2 Instruction conditionnelle `if` et booléens . . . 85
3.3 Itération conditionnelle `while` . . . 90
3.4 Fonctions . . . 97
3.5 Caractères et chaînes de caractères . . . 103
3.6 Listes . . . 107
3.7 Itération inconditionnelle `for` . . . 113
3.8 Listes avancées : `for`, tri, listes à 2 dimensions, etc . . . 119
3.9 Boucles `for` et maths . . . 123
3.10 Fonctions avancées . . . 126
3.11 Dictionnaires . . . 129
3.12 Fichiers . . . 138

4 EXERCICES DE TP **143**
4.1 TP1 : prise en main des outils, lecture et 1er programmes . . . 144
4.2 TP2 : prix du gros lot . . . 149
4.3 TP3 : initiation aux fonctions avec `turtle` . . . 154
4.4 TP4 : création et manipulation de fonctions basiques . . . 162
4.5 TP5 : manipulation de fonctions (suite), chaînes de caractères . . . 167
4.6 TP6 : listes . . . 170
4.7 TP7 : algorithmes pour trier des listes . . . 175
4.8 TP8 : propagation d'une nouvelle . . . 178
4.9 TP9 : dictionnaires et traduction . . . 182
4.10 TP10 : jeu du démineur . . . 186
4.11 TP11 : révisions . . . 191

Annexes **193**
4.12 Comment sont notés vos programmes . . . 193
4.13 Bonnes pratiques . . . 194

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

Chapitre 1

INTRODUCTION

IMPORTANT — À LIRE — CONSIGNES ET BONNES PRATIQUES

Le but de cette UE **n'est pas** :

- connaître les raccourcis de programation spécifiques à Python ;
- connaître toutes les fonctions et / ou modules Python ;
- programmer en orienté-objet ;
- écrire le code le plus court possible ;
- écrire un code « qui marche ».

Le but de cette UE **est** :

- apprendre à écrire des algorithmes pour résoudre des problèmes ;
- connaître les algorithmes classiques (tri, comptage, recherche...) ;
- savoir implémenter ces algorithmes en respectant la syntaxe d'un langage de programmation, ici Python ;
- apprendre à écrire un code propre, modulaire, commenté, non redondant, respectant les contraintes de l'énoncé ;
- acquérir les premières notions de complexité des programmes.

1.1 Comment sont notés vos programmes

Un programme « qui marche » n'aura pas la note maximale si :

- il est illisible, raturé ;
- il est incompréhensible, non commenté ;
- il contourne les consignes de sorte qu'il ne permet pas au correcteur d'évaluer l'élément souhaité ;
- il utilise des raccourcis de programmation spécifiques à Python ;
- il utilise des éléments (fonctions, modules, constructions, ...) non vus ou hors programme ou non explicitement autorisés ;
- il utilise des éléments explicitement interdits (par exemple `break` ou `continue`) ;

- du code y est dupliqué ou redondant ;
- il est inutilement compliqué ;
- il ne respecte pas la consigne ou les contraintes données (exemple d'exécutions, format d'affichage, texte dans les entrées-sorties, ...).

1.2 Bonnes pratiques

Il est donc vivement conseillé de :

- écrire un code propre ;
- respecter les niveaux d'indentation ;
- commenter les programmes, en particulier les `else` et les sorties de boucle ;
- donner des noms parlants aux variables, en évitant les mots-clés réservés de Python ;
- toujours afficher un message pour accompagner les `input` / `print`.

Chapitre 2

RÉSUMÉ DE COURS

2.1 Quelques définitions

2.1.1 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.

2.1.2 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).

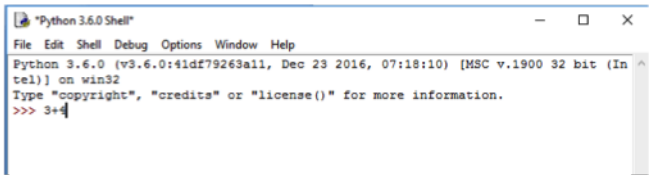
```
Algo Omelette_Elémentaire
Début
   {ingrédients}
      1 oeuf, sel poivre, beurre
   {ustensiles}
      1 saladier, une fourchette, une poêle, une spatule

   {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).

2.1.3 En Python

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

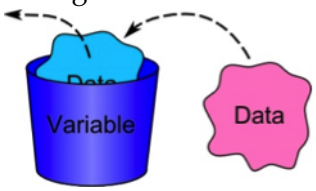
- soit on écrit un ensemble d'instructions dans un fichier puis on l'exécute
- soit on utilise l'interpréteur Python, instruction par instruction, un peu à la façon d'une calculatrice.



2.2 Variables, expressions, instructions

2.2.1 Variables

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



2.2.2 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)

Par convention, pour les variables en Python, on utilise des minuscules. On s'interdira également d'utiliser des accents. On essayera de choisir des noms parlants pour faciliter la lisibilité du code. **Standard** : le standard `PEP8` [1] liste d'autres conventions et bonnes pratiques stylistiques en Python (cohérence de nommage des variables et fonctions, longueur des lignes, style des commentaires, etc).

2.2.3 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
```

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

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.

```
>>> 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
```

2.2.4 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 2.5, et sur les chaînes de caractères au chapitre 2.9.

```
>>> 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'>
```

2.2.5 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) and (8 < 15) # conjonction pour comparer 3 entiers
True
>>> (x % 2 == 0) or (x >= 0)
                # disjonction, expr. bool. dont la valeur dépend de la
```

```
                #   valeur de la variable x (vrai 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'abréger 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 # abrévie a = a + 1
>>> a
4
```

```
>>> a *= 3 # abrévie a = a * 3
>>> a
12
>>> a -= 5 # abrévie 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 $2 * (3 + 4) = 2 * 7 = 14$.

Opérateurs par ordre de priorité décroissante :

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

Évaluation des expressions 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.

```
>>> 8 // 4 // 2      # même op., même prio., éval. de gauche à droite
1
>>> 8 // (4 // 2)    # les parenthèses forcent l'éval. de droite à gauche
4
>>> 8 * 4 // 2 ** 3  # puissance évaluée en 1er, * et // de gauche à droite
4
```

2.2.6 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).

2.2.7 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.

2.3 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 pourrez trouver en ligne. Dans cette UE nous utiliserons un certain nombre d'outils :

- CaseIne : https://moodle.caseine.org/course/view.php?id=86. Sur cette plate-forme 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.
- En TP, nous programmerons sous Idle, un environnement de développement intégré basique que vous pouvez aussi installer sur votre ordinateur :
  https://wiki.python.org/moin/BeginnersGuide/Download
- 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

D'autres outils utiles :

- PyCharm, une IDE avec un bon débugueur, 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://openclassrooms.com/fr/courses/6204541-initiez-vous-a-python-pour-l

- 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 (bien sélectionner le langage Python).

2.4 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**

2.4.1 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'util. saisit 123 en réponse
>>> texte
'123'                  # la var. texte contient la chaîne de car. '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
```

2.4.2 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 `input`). 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 retour ligne final
20;13
>>> print("a=",a,"b=",b, sep="\n")  # retour ligne entre chq arg. + final
a=
20
b=
13
>>> print(a,end="!")                # pas de retour à la ligne final
20!>>>
>>> c = 7
>>> print(a,b,c,sep=";",end=" !\n")
# séparateur ; au lieu d'espace, et on rajoute un ! avant 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
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
```

```
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
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 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

2.5 Les expressions booléennes

2.5.1 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
```

2.5.2 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)`.

2.5.3 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.

2.5.4 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) # opérande gauche vraie donc disj. déjà vraie
True

>>> (3 == 1+1) or (a>5)  # op. gauche 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) # opérande gauche fausse donc conjonction aussi
False
```

2.5.5 Abréviations des expressions booléennes

On peut parfois abréger certaines expressions booléennes composées (conjonctions).

```
>>> 2 <= 8 < 15             # comparaison de 3 entiers, expr. bool.
True
>>> (2 <= 8) and (8 < 15)   # conj., expr. équivalente à la précédente
True
```

De telles abréviations ne sont pas toujours recommandées pour des questions de lisibilité. Par exemple il peut être difficile d'évaluer ce que signifient les expressions suivantes. A votre avis lesquelles sont équivalentes ?

```
>>> 1 < 2 < 3 == 7%2 != 0
>>> (1 < 2 < 3) == (7%2 != 0)
>>> ((1 < 2 < 3) == 7) % 2 != 0
>>> (1<2) and (2<3) and (3==1) and (1!=0)
```

C'est en fait la dernière expression qui correspond au détail de la première, et qui explique pourquoi le résultat de son évaluation est `False`.

On recommandera aussi de toujours bien parenthéser ses expressions booléennes, même si ce n'est pas syntaxiquement nécessaire (revoir les priorités des opérateurs au chapitre précédent), pour les rendre plus lisibles et non ambigues.

```
>>> a or b and c         # and est plus prioritaire que or
>>> a or (b and c)       # équivalent à la précédente

>>> not a and b or c     # not est plus prioritaire
>>> ((not a) and b) or c # équivalent à la précédente
```

2.5.6 Lois de De Morgan

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

- not(expr1 or expr2) = not(expr 1) and not(expr2)
- not(expr1 and expr2) = not(expr 1) or not(expr2)

Exemples
not(a > 2 or b <= 4) équivaut à not(a>2) and not(b<=4) ce qui équivaut à (a <= 2) and (b > 4)
not(a > 2 and b <= 4) équivaut à not(a>2) or not(b<=4) ce qui équivaut à (a <= 2) or (b > 4)

2.5.7 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 2.1 – Table de vérité pour le ou exclusif

2.6 Instructions conditionnelles 'if'

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.

2.6.1 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 exécutées. 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.

2.6.2 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 :                                   # sinon (donc si condition fausse)
    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.

2.6.3 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 autant de conditions que nécessaire. Par contre il ne peut y avoir qu'un seul `else` (cas par défaut), 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 les 2e et 3e tests échouent aussi, on sait que note >= 15, else suffit
else:
    print("Vous avez la bosse des maths !")
```

2.6.4 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 forcement 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.

2.6.5 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")
```

2.6.6 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).
- `randrange(a,b,step)` : renvoie un entier pseudo-aléatoire dans `range(a,b,step)`, ce qui permet en plus de spécifier un pas (par défaut 1). Attention la borne supérieure b est ici **exclue**.
- `random()` : renvoie un réel pseudo-aléatoire entre 0 et 1.
- `choice(l)` : renvoie un élément au hasard parmi ceux de la liste l.

Pour utiliser les fonctions de ce module, il faut l'importer au début du programme avec la commande : `import random` puis préfixer les appels aux fonctions par le nom du module. Exemple :

```
import random
# nombre entier au hasard entre 1 et 10
i = random.randint(1,10)
# nombre réel au hasard entre 0 et 20
x = 20 * random.random()
```

2.7 La boucle conditionnelle '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. On appelle donc cette structure une **boucle conditionnelle**. 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). En Python, la syntaxe générale du `while` est la suivante :

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

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 au clavier pendant l'exécution.

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.

2.7.1 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.

2.7.2 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 voici 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 à '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 répond 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).

2.7.3 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 avec 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
```

2.7.4 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  # chq valeur de i est ajoutée à 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* a 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')
```

2.7.5 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)
```

Altération du fonctionnement de la boucle - Inconvénients

Les instructions `break` et `continue` permettent d'altérer le comportement normal de la boucle `while`. Cependant, leur utilisation 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 d'éviter d'utiliser `break` et `continue`.

Instruction Break

L'instruction *break* 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 moyenne<10 sortie 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)
```

On peut utiliser le caractère \ pour séparer une expression booléenne trop longue 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 directement dans la condition de la boucle.

```
# version avec break (à ÉVITER)
while cond:
    instructions
    if condstop :
        break
```

```
# réécriture sans break (à PRÉFÉRER)
stop=False
# on teste les 2 conditions
while (not condstop) and cond:
    instructions
```

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 ni 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 instr. continue
i = 0
somme = 0  # initialisation somme
while i < 10:
    n = int(input())
    # si n est un multiple de 2 ou de 3
    if n % 2 == 0 or n % 3 == 0:
        # passer à l'itérat. suivante
        continue
    # donc ici n mult. ni de 2 ni de 3
    somme = somme + n
    i = i + 1
```

```
# réécriture sans instr. continue
i=0
somme=0
while i<10:
    n=int(input())
# on utilise la nég. de la cond.
    # si n mult. ni de 2 ni de 3
    if n%2!=0 and n%3!=0:
        # on l'ajoute à la somme
        somme += n
    i += 1
```

2.7.6 Imbrication de boucles

Il peut y avoir une boucle `while` parmi les instructions dans un bloc `while` : on parle alors de boucles *imbriquées*. Prenons comme exemple le programme ci-dessous. 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.

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

```
# affichage produit
    1, 1
    1, 2
    2, 1
    2, 2
    3, 1
    3, 2
```

Exemple d'application : le programme ci-dessous demande à l'utilisateur un entier N et affiche un carré de caractères '*' de N lignes et N colonnes.

```
    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 même lign
            j = j + 1           # passage a l'étoile suivante
        # fin de l'itération sur j qui a affiché 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.

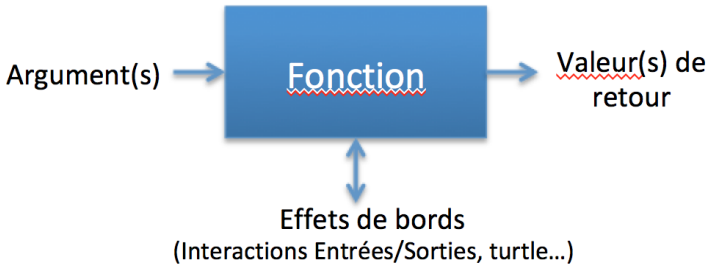
2.8 Les fonctions

On a pour l'instant développé uniquement des programmes *tout-en-un*. Les fonctions permettent de découper le code en plusieurs morceaux réutilisables.

2.8.1 Introduction

Qu'est-ce qu'une fonction ?

Une fonction est une suite d'instructions, encapsulées dans une «boîte», identifiée par un nom ; elle reçoit zéro, un ou plusieurs 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.



Pourquoi écrire des fonctions ?

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 Par exemple en TP on a écrit une fonction qui permet de tracer un carré d'une longueur de côté donnée (en paramètre). On pourra dans notre programme appeler cette fonction plusieurs fois, pour tracer plusieurs carrés, en indiquant seulement la taille de chaque carré à tracer. Cela évite de recopier à chaque fois tout le code qui trace un carré, et rend donc le code plus compact et plus lisible.

Deux types de fonctions On pourra écrire deux types principaux de fonctions, selon ce qu'elles font :

- Des **fonctions** qui calculent et **renvoient** une (ou plusieurs) valeur(s) : par exemple calculer la valeur absolue d'un nombre (valeur entière), calculer la moyenne d'une liste de nombres (valeur réelle), vérifier si une lettre est une voyelle (valeur booléenne), etc
- Des fonctions qui ne renvoient rien mais qui effectuent des actions, qui ont des **effets de bord** sur leur environnement. On les appelle aussi des **procédures**. Par exemple : afficher un message, dessiner un carré, trier une liste, etc.

2.8.2 Les librairies Python

On peut définir soi-même des fonctions, mais on peut aussi utiliser des fonctions déjà fournies par Python ou par ses nombreuses librairies. En effet, 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  # préfixer ensuite les fonctions par le nom du module
from nom_module import nom_fonction # importer une fonction specifique,
                                    # pas besoin de préfixer son nom
from nom_module import *     # importer toutes les fonctions du module,
                             # pas besoin de prefixer leurs noms

import matplotlib.pyplot as plt  # renommage module pour simplifier 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.

Quelques modules utiles (on en a déjà utilisé certains) :

- `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

2.8.3 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
```

Fonction vs procédure Le `return` est facultatif : dans ce cas la fonction ne renvoie rien, c'est une procédure, les instructions qu'elle exécute modifient son environnement et créent des effets de bord. Par exemple `print` affiche un message mais ne renvoie pas de valeur.

Les arguments / paramètres sont facultatifs : une fonction peut ne recevoir aucun paramètre si elle n'a besoin d'aucune information pour fonctionner ; par contre les parenthèses sont obligatoires (même si elles sont alors vides). Par exemple `input` peut être appelée sans aucun paramètre, elle se contente alors de lire une entrée au clavier et de la renvoyer.

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, elle a un effet de bord (affichage du message 'Quel est ton nom ?' et attente d'une saisie par l'utilisateur), et 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 (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
```

Syntaxe de l'appel Une fois définie (par soi-même, ou dans une librairie qu'on a importée), on peut appeler une fonction par son nom :

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

Tant qu'on n'appelle pas une fonction, elle n'est pas exécutée.
La syntaxe pour appeler une fonction est la suivante :

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

Structure d'un fichier Python Votre fichier doit être organisé de la manière suivante :

- Tout en haut, les importations de modules utiles
- Ensuite, les définitions de vos fonctions
- Tout en bas, le programme principal qui appelle ces fonctions
- En n'oubliant pas de commenter raisonnablement le code !

2.8.4 Appel de fonctions

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.

Les paramètres passés peuvent être des constantes (par exemple : 1, 2.3, "bonjour"), des variables, des opérations (la valeur est évaluée avant, puis passée à la fonction), ou des appels à d'autres fonctions (qui sont alors évalués d'abord, pour savoir quelle valeur passer à la fonction). Par exemple :

```
# appel sur une constante
print("bonjour")
nom = input("Ton nom ? ")
# appel sur une constante et une variable
print("bonjour",nom)
# appel sur une opération: l'opération est résolue d'abord,
# puis la fonction print reçoit directement la valeur 10
print("la somme vaut", 3+7)
# appel sur une valeur de retour de fonction
print("bonjour",input("quel est ton nom? "))
```

Dans le dernier appel, la fonction `input` est appelée **d'abord** pour évaluer sa valeur de retour (le nom entré par l'utilisateur). Cette **valeur** (par exemple "toto") est passée en paramètre à la fonction `print` qui affichera alors le message "bonjour toto". On notera quand même que cette dernière syntaxe est moins lisible que la décomposition.

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>
```

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 **récursive** (hors programme de cette UE).

2.8.5 Valeurs de retour

Affectation 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. Si on veut réutiliser la valeur de retour, il **faut** la stocker dans une variable, et surtout pas rappeler la fonction plusieurs fois, sinon on refait les calculs à chaque fois (ce qui est inutile, et peut être coûteux...).

```
# 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 vale
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'u
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
>>> x = carre(50)              # si on affecte quand meme
>>> x                          # x n'a pas de valeur
>>> print(x)                   # il s'affiche comme la valeur spéciale 'Non
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'
```

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.

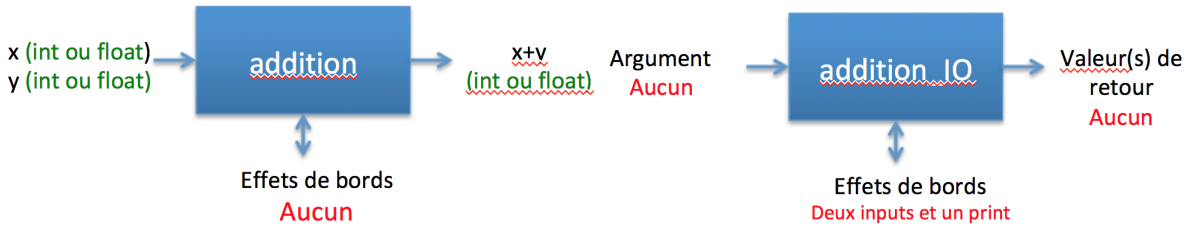
- Les **valeurs de retour** sont des valeurs renvoyées par la fonction, avec le mot-clé `return`. Ces valeurs peuvent (doivent) être utilisées dans la fonction ou le programme qui appelle cette fonction : on peut les afficher, les affecter dans des variables, etc.
- Les **effets de bord** sont des actions 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 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).

```
# Addition de 2 nombres, RENVOIE la somme
def addition(x,y):
    return x+y

# Addition de 2 nombres, AFFICHE la somme
def addition_aff(x,y):
    print(x+y)
```

```
# Addition, demande 2 réels, 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 effets de bord de ces fonctions d'addition.



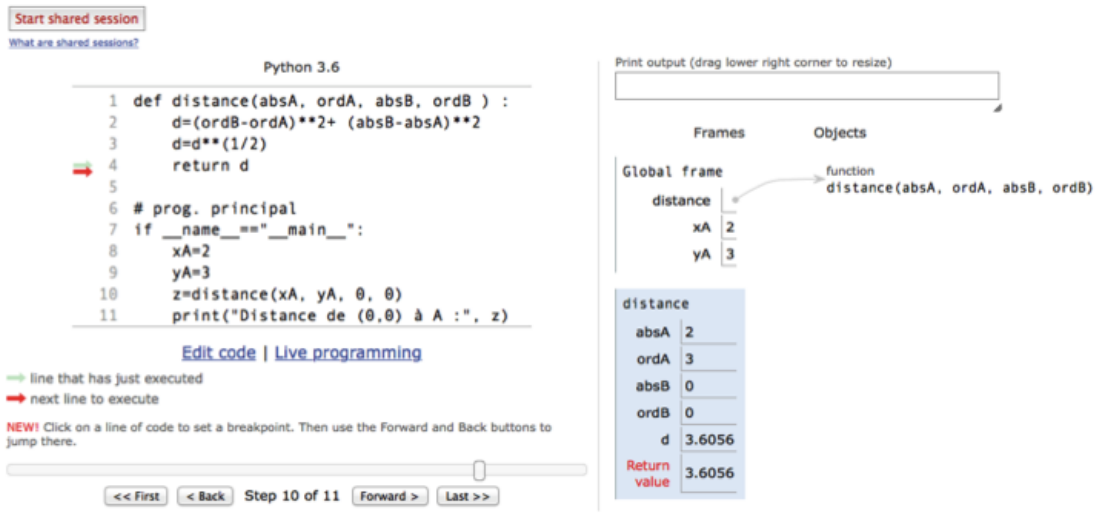
L'appel de ces fonctions depuis un programme principal est donc très différent.

```
# programme principal

# appel de la fonction d'addition qui renvoie la valeur
somme = addition(3,7)          # il faut affecter la valeur dans une var.
print("la somme vaut",somme)   # on peut alors afficher cette var.
# 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 paramètres
                       # la fonction interagira avec l'utilisateur
```

2.8.6 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`.



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 prog. 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 prog. principal en lecture seule
def decalage(s):
    return ("-" * n) + s

# depuis l'interpréteur
```

```
>>> n=5
>>> print(decalage("toto"))
-----toto

# version correcte, reçoit valeur en param. 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 2.10) 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).

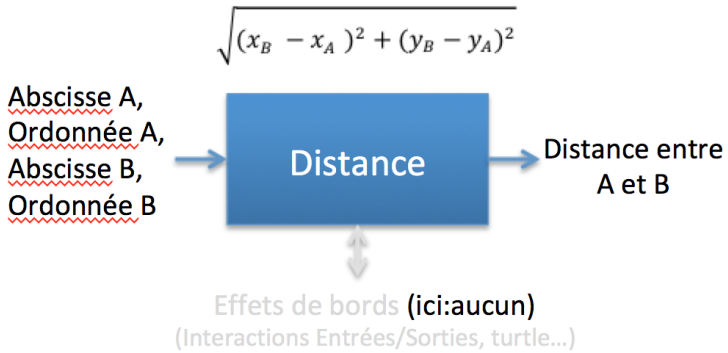
Résumé

- 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 (listes...).
- On passera en argument des fonctions toutes les valeurs nécessaires à leur fonctionnement.
- On évitera en général 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`).

2.8.7 Exemple : fonction géométrique de calcul de distance entre 2 points

Cette section illustre les notions vues ci-dessus sur un exemple.

Définition de la fonction



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
```

Appel dans le programme principal 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.

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

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

Affectation de la valeur de retour Cette fonction renvoie une valeur, quand on l'appelle dans un programme il faut donc **faire quelque chose** de cette valeur : soit on l'affiche directement (comme dans le premier appel, ou dans l'interpréteur), 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.

Erreurs d'appel Si on ne donne pas le bon nombre de paramètres lors de l'appel, on déclenche une erreur.

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

>>> 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
```

2.9 Chaînes de caractères

2.9.1 Type string

Les chaînes de caractères (`string` en Python) sont un type de données, plus complexe que ceux vus jusqu'à présent. Il s'agit d'un type *itérable*, c'est-à-dire dont on peut parcourir les valeurs (ici les différents caractères qui composent la chaîne). On verra d'autres types itérables plus tard (listes, dictionnaires).

Syntaxe 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 doubles dans le texte :

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

- Avec des guillemets triples, ce qui permet de créer de longues chaînes contenant des guillemets, des apostrophes, des sauts de ligne... :

```
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 : ...
```

2.9.2 Caractères

Table ASCII 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.

Fonctions ord() et chr() Ce tableau est à la fin du chapitre, mais 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)
'{'
```

Caractères d'échappement Le caractère \ permet d'utiliser des caractères spéciaux dans une chaîne de caractères :

- \' est une apostrophe mais ne ferme pas la chaîne de caractères (même si entre apostrophes)
- \" est une guillemet double, mais il ne ferme pas la chaîne de caractères
- $\backslash n$: retour à la ligne
- $\backslash t$: tabulation
- \\ : si on veut insérer un caractère \ (*'backslash'*)

Exemples

```
>>> print("il a dit \"bonjour\" en arrivant")
il a dit "bonjour" en arrivant
```

2.9.3 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
```

2.9.4 Fonctions de manipulation de chaînes

On retrouvera ces mêmes fonctions sur les listes.

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

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

Test d'appartenance Le mot-clé `in` permet de vérifier si une chaîne (ou un caractère) est inclue dans une autre. Par exemple :

```
>>> "a" in "toto"
```

```
False
>>> "o" in "toto"
True
>>> "to" in "toto"
True
>>> "abc" in "toto"
False
>>> "abc" in "abcd"
True
>>> "acb" in "abcd"
False
```

Compter On peut aussi vouloir compter le nombre d'occurrences (d'apparitions) d'une chaîne dans une autre, avec la fonction `count()`. Par exemple :

```
>>> "toto".count("o")
2
>>> "toto".count("to")
2
>>> "toto".count("ot")
1
>>> "toto".count("a")
0
```

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

2.9.5 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 2.7), ou par itération directement sur les éléments avec une boucle `for` (cf chapitre 2.11).

```
>>> 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
```

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)						

2.10 Listes

2.10.1 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 car.
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 nouvelle »
>>> l
['C', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I']
>>> len(l)
11
>>> 3 * [2]
[2,2,2]
```

2.10.2 Numérotation des éléments et accès

Une liste est un ensemble **ordonné**. Cela signifie 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-à-d 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 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 utilise la fonction `len()`. Par exemple avec les listes précédentes :

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

2.10.3 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 utiliser 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
```

2.10.4 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 sép. ne sert à rien, 1 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 personnaliser l'affichage, 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 ligne
        i += 1  # élément suivant
    print()     # retour à la ligne final
# appel dans l'interpréteur
>>> prenoms = ['toto','titi','yoyo','mumu']
>>> afficheET(prenoms)
# pour éviter le et final il faudrait traiter le dernier élément à part
toto et titi et yoyo et mumu et
```

Affichage avec boucle for Les boucles `for` (chapitre 2.11) 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 ligne auto à chaque élément

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

2.10.5 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 une 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'une 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]
```

2.10.6 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
[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 elt absent
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

2.10.7 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 liste3. Les 2 variables correspondent bien à 2 listes différentes.

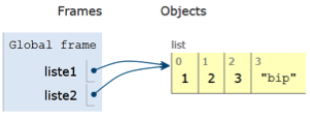


FIGURE 2.1 – Affectation (renommage) : liste2 = liste1

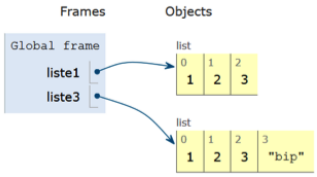


FIGURE 2.2 – Copie de surface : liste3 = list(liste1)

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]
```

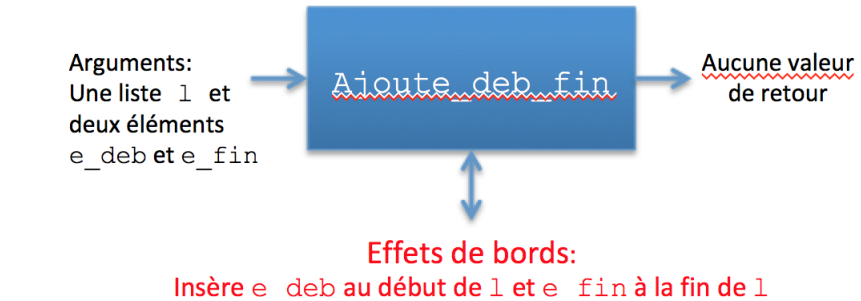
Copie de surface vs copie profonde Si une liste contient d'autres listes, pour la copier correctement il faut utiliser la fonction `deepcopy` du module `copy`, qui va faire une copie *profonde*, récursive, c'est-à-dire qu'elle va aussi faire des copies des listes contenues dans la liste copiée. Pour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

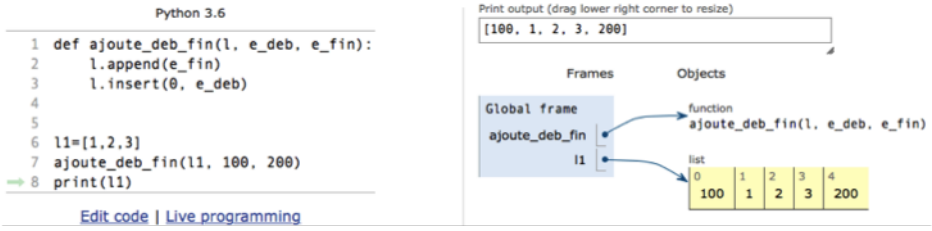
```
>>> liste1
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> surface = copy.copy(liste1)
>>> profonde = copy.deepcopy(liste1)
>>> surface                  # copie de surface, liste d'indice 3 non clonée
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> profonde
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> surface[3].append('d')  # si on modifie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

2.10.8 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.

Par exemple imaginons une fonction qui insère un élément en début et fin de liste.





Exemple avec oubli de la copie

```
import random

def ajoute_random(l):
    """Liste obtenue à partir de l en ajoutant un entier aléatoire 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):
    """Liste obtenue a partir de l en ajoutant un entier aléatoire 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, différente 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.

2.11 Boucle inconditionnelle : for

2.11.1 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.

2.11.2 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 <= 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 inclue, 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 2.13). 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 à 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 depasse 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
```

2.11.3 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)*

2.11.4 Très important : itération et modification

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)
```

Attention : ne jamais modifier la structure pendant qu'elle est parcourue !

Exemple Dans l'exemple ci-dessous, on définit une liste de 5 entiers, puis on écrit une boucle qui affiche chaque élément puis le supprime de la liste.

```
>>> liste = [1,2,3,4,5]
>>> for i in liste :
        print(i)
        liste.remove(i)

# affichage produit
1
3
5

>>> liste
[2, 4]
```

Le comportement obtenu n'est pas ce qu'on attendait : seul un entier sur 2 est affiché et supprimé de la liste. En effet, il ne faut jamais modifier la structure (ici la liste) en même temps qu'on la parcourt.

2.12 Listes avancées

Ces éléments sont hors programme, mais sont ici pour votre culture.

2.12.1 Fonction map

La fonction `map()` permet d'appliquer une fonction à tous les éléments d'une liste, et d'obtenir la liste de tous les résultats. Le résultat est de type `map`, qu'il faut convertir en liste. On peut utiliser en paramètre le nom d'une fonction existante, ou bien une fonction anonyme avec `lambda`. Les fonctions anonymes peuvent aussi servir à combiner plusieurs fonctions. (Voir les exemples ci-dessous.)

```
>>> list(map(len,["alex","cyril","elsa"]))
[4, 5, 4]
>>> list(map(math.sqrt,[1,4,9,16]))
[1.0, 2.0, 3.0, 4.0]
>>> list(map(lambda x:x**2,[1,4,9,16]))
[1, 16, 81, 256]
>>> list(map(lambda x:int(math.sqrt(x)),[1,4,9,16]))
[1, 2, 3, 4]
```

2.12.2 Compréhension de listes

Python offre une syntaxe abrégée pour manipuler les listes : la compréhension de listes. C'est une syntaxe plus concise pour écrire une boucle `for` qui crée une liste. Cela permet par exemple de réécrire la fonction `map`. Quelques exemples :

```
>>> [x**2 for x in range(10)]        # carres des entiers de 0 a 9
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> prenoms = ["alex","cyril","elsa"]
>>> l = [len(x) for x in prenoms]
[4, 5, 4]
>>> ["a" in x for x in prenoms]
[True,False,True]
```

Exercice bonus : essayer d'écrire les fonctions du TD sur les boucles en un minimum d'instructions en utilisant des compréhensions de listes (à éviter en général dans un vrai programme, dans un souci de lisibilité).

2.13 Fonctions : compléments

2.13.1 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)
```

```
Au lancement du module, affiche :
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")
```

2.13.2 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)
```

2.13.3 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) reçoi(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 (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 ensuite n'est pas déterminant, seule compte leur position (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 définie 2 fois
(True pris pour taille car 2ème arg. non-nommé)
```

L'exemple de la fonction print En fait, nous avions 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.

2.13.4 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
```

2.14 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**.

2.14.1 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 = { 'nathan': 12.0, 'quentin': 15.5 }
>>> 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,                                      # val. de type int
    'cpu': 3.5,                                     # val. de type float
    'portable': False,                              # val. de type bool
    'os': 'windows',                                # val. de type str
    'ports': ['usb3.0', 'jack', 'ethernet', 'hdmi'], # val. de type list
    'carte_graphique': {                            # val. de type dict
          'vram': 4,
          'nom': 'gtx970',
          'bus': 256
    }
}
```

Exemple 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}
}
```

2.14.2 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']  # erreur, pas de clé 'k' dans le dico
KeyError: 'k'
>>> lettres['u'] = lettres['u'] + 1 # erreur, pas de clé 'u'
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, avec l'opérateur `in` comme pour les 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
```

2.14.3 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 = {}          # crée un dictionnaire vide
>>> D
{}
>>> D['a'] = 1      # ajout de la nouvelle entrée
>>> 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'
```

2.14.4 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 : valeur associée au nom, la liste représentant la date de naiss.
    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 `keys()` permet d'accéder aux clés d'un dictionnaire, dans une structure de type `dict_keys`. De même la fonction `values()` permet d'accéder aux valeurs d'un 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()`. Par exemple :

```
>>> 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
```

2.14.5 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 (la fonction ne renvoie pas le dictionnaire, mais a pour effet de bord de le modifier). 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 cpts:
        # si existe, incrementation
        cpts[l] = cpts[l] + 1
    else:
        # sinon, initialisation
        cpts[l] = 1

# programme principal
phrase = "bonjour a tous"
# initialiser un dictionnaire vide de compteurs
cpts = {}
for lettre in phrase:
    # appel de la fonction sur chaque lettre vue
    vu_lettre(lettre,cpts)
# affichage du dictionnaire de compteurs
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).

2.14.6 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)                  # création d'une copie de D nommée F
>>> F[6] = 60                    # ajout d'une clé dans F
>>> F                            # F a été 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                        # création d'un 2e nom pour le même dico
>>> G[7] = 1                     # ajout d'une clé 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}
```

2.15 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.

2.15.1 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'
```

2.15.2 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)
# après la fin de la boucle de calcul, afficher une seule fois la somme
print(somme)
# ne pas oublier de fermer le fichier après utilisation
fichier.close()
```

2.15.3 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 $\backslash n$ (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.

2.15.4 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 écriture (création si inexistant, écrasement si existe)
fichier = open('puis.txt', 'w')
# 10 itérations 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 3

EXERCICES DE TD

3.1 Les bases

3.1.1 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("Apres echange 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.

3.1.2 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-!

3.1.3 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
```

3.1.4 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 !
```

3.1.5 Âge

Écrire un programme qui demande à l'utilisateur son année de naissance, puis calcule et affiche son âge.

Exemple d'exécution :

```
Quelle est ton année de naissance ? 1998
En 2020 tu as 22 ans
```

3.1.6 Identité

Écrire un programme qui demande à l'utilisateur son prénom, son nom, puis affiche son identité complète.

Exemple d'exécution :

```
Quel est ton prénom ? Léo
Quel est ton nom ? Dupont
Identité : Léo Dupont
```

3.1.7 Emploi du temps

Écrire un programme qui demande à l'utilisateur son nombre d'heures de cours par semaine, puis le nombre de semaines de cours, puis calcule le nombre d'heures totales et l'affiche exactement comme dans l'exemple ci-dessous :

```
Combien d'heures par semaine ?   6
Combien de semaines de cours ?   12
Total : 72 heures
```

3.1.8 Rectangle

Écrire un programme qui demande à l'utilisateur la longueur et la largeur d'un rectangle, et calcule et affiche son aire exactement comme dans l'exemple ci-dessous :

```
Largeur ?   3
Longueur ?  7
L'aire du rectangle de largeur 3 et de longueur 7 vaut 21
```

3.1.9 Puissance de 2

Écrire un programme qui demande à l'utilisateur un entier n, puis calcule et affiche 2 à la puissance n, exactement comme dans l'exemple d'exécution ci-dessous :

```
Entier ? 3
2 puissance 3 = 8
```

3.2 Instruction conditionnelle `if` et booléens

3.2.1 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
-> ...
```

3.2.2 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

3.2.3 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.

3.2.4 Comparaison de deux programmes

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

```
rep = input("Êtes-vous fumeur : ")

if rep == "oui" or rep == "OUI" :
    print("Vous devriez arrêter")
else :
    print("Continuez ainsi")
    print("Merci de votre réponse")
```

```
rep = input("Êtes-vous fumeur : ")

if rep == "oui" or rep == "OUI" :
    print("Vous devriez arrêter")
else :
    print("Continuez ainsi")
print("Merci de votre réponse")
```

3.2.5 Comparaison de deux programmes (bis)

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

```
n = int(input("Entrez n : "))

if n%2 == 1 :
    n = 3*n+1
else :
    n = n//2

print(n)
```

```
n = int(input("Entrez n : "))

if n%2 == 1 :
    n = 3*n+1
if n%2 == 0 :
    n = n//2

print(n)
```

3.2.6 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

3.2.7 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

3.2.8 Lois de De Morgan

Utiliser des tables de vérité pour prouver les lois de De Morgan (négation des conjonctions et disjonctions, cf cours).

3.2.9 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.

3.2.10 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.

3.2.11 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
   ```

3.2.12 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 !
```

3.2.13 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.*

3.2.14 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.

3.2.15 Encore des expressions booléennes

On suppose que toutes les variables nécessaires sont correctement initialisées : *a* et *b* sont des entiers, *c* et *rep* sont des chaînes de caractères, on suppose qu'on sait que c est de longueur 1. 1. Écrire les expressions booléennes qui représentent les assertions suivantes :

1. `a` est un entier pair
2. `a` est un multiple de 3 ou un multiple de 7
3. `b` est un entier strictement négatif multiple de 3
4. `c` est une lettre minuscule de l'alphabet
5. `c` est une lettre de l'alphabet (majuscule ou minuscule)
6. `c` est un caractère numérique
7. `rep` n'est ni la lettre `'o'` ni la lettre `'n'`
8. 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`).

3.3 Itération conditionnelle `while`

Bases des itérations conditionnelles

3.3.1 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.

3.3.2 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 ?

3.3.3 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

3.3.4 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.

3.3.5 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`

3.3.6 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 autorise' 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

3.3.7 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
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
```

3.3.8 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.

3.3.9 Factorielle

Écrire un programme qui lit un entier n saisi par l'utilisateur, et qui affiche n ! (factorielle de n).
Rappel : n ! = 1 x 2 x 3 x 4 x . . . x n
Indice : Attention à la valeur initiale.
Bonus avec une boucle imbriquée : proposer à l'utilisateur de rejouer avec une nouvelle valeur de n.

3.3.10 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

3.3.11 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 ?

3.3.12 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

3.3.13 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-à-d que la distance entre *nbor* et *estimnbo* est inférieure ou égale à p.

3.3.14 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 matheux 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.

Boucles imbriquées

3.3.15 Escaliers

Écrire un programme qui demande un entier L à l'utilisateur, et qui affiche la forme suivante, constituée de L lignes :

```
a. ****     b. *      c.      *   d. *****   e. *****   f.      *
   ****        **             **      ****       ****         ***
   ****        ***           ***      ***         ***        *****
   ****        ****         ****      **           **       *******
   ****        *****       *****      *             *      ****o****
```

Dans la figure e, la porte (caractère 'o') est en bonus, pour simplifier vous pouvez afficher une '*' à la place. Dans cet exercice on s'interdira d'utiliser l'opérateur de multiplication pour créer les chaînes à afficher, on n'affichera qu'une étoile à la fois.

3.3.16 Triangles d'entiers

Écrire un programme qui demande à l'utilisateur un nombre de lignes, et affiche un triangle d'entiers avec ce nombre de lignes, sous la forme suivante (dans les exemples avec n=4)

```
1                1              1 2 3 4          1 2 3 4
1 2              2 3            1 2 3            5 6 7
1 2 3            4 5 6          1 2              8 9
1 2 3 4          7 8 9 10       1                10
   a)               b)             c)               d)

10 9 8 7           1               1
 6 5 4           1 2             2 3
 3 2           1 2 3           4 5 6
 1           1 2 3 4         7 8 9 10
   e)             f)              g)
```

3.3.17 Moyenne de classe

On veut calculer la moyenne d'un groupe de 30 étudiants. Pour ce faire, on veut écrire un programme qui, pour chaque étudiant du groupe, demande à l'utilisateur de saisir 10 notes. A la fin de la saisie de ces 10 notes, le programme doit afficher la moyenne de l'étudiant. A la fin du programme, la moyenne de tout le groupe doit être affichée.

Bonus : afficher aussi en fin de programme la moyenne et le numéro de l'étudiant qui a la meilleure moyenne.

3.3.18 Pyramide ailée

Écrire un programme qui demande un entier n à l'utilisateur et affiche une pyramide ailée de hauteur n comme sur la figure suivante :

3.3.19 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.

3.4 Fonctions

3.4.1 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,n)
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)
```

3.4.2 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 choisie 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)
```

3.4.3 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

3.4.4 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.

3.4.5 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.

3.4.6 Conversions binaire-décimal

Dans cet exercice on s'intéresse à convertir des entiers entre la base 2 (écriture binaire) et la base 10 (écriture décimale). **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.

Vous pourrez cependant vous servir de ces fonctions pour vérifier vos résultats.

1. Écrire une fonction qui prend en argument un entier n, et renvoie une chaîne de caractères représentant son écriture en binaire. *Par ex. cette fonction appelée sur l'entier 9 renvoie la chaîne de caractères "1001".*
2. 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.
3. Écrire enfin un programme principal qui effectue la boucle suivante :
   - propose à l'utilisateur 3 choix : conversion binaire→décimal, décimal→binaire, ou arrêt
   - 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

3.4.7 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.

3.4.8 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.

3.4.9 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 ?

3.4.10 Factorielle qui dépasse

Écrire une fonction `depasse` 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 :* $n! = 1 \times 2 \times 3 \times \cdots n$

3.4.11 Calculatrice

Dans cet exercice, nous allons nous interdire d'utiliser les opérateurs mathématiques de Python, ainsi que les fonctions du module math. A 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
$
Op. inconnu, on recommence !

A=5
**
B=5
3125
```

3.5 Caractères et 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. Les exercices qui nécessitent des boucles peuvent être réalisés avec des boucles `while`, ou des boucles `for` si vous les avez déjà vues.

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

Caractères

3.5.1 Booléens et caractères

Écrire les expressions booléennes correspondant aux assertions suivantes (on suppose toutes les variables correctement initialisées, x et y sont des chaînes de caractères, n est un entier) :

- La variable x est une lettre majuscule de l'alphabet
- La variable y est une voyelle en majuscule
- Les variables x et y sont des lettres minuscules de l'alphabet
- x est strictement avant y dans l'ordre alphabétique
- x contient entre 3 et 7 lettres (inclus)
- x est une lettre minuscule, et y est la même lettre en majuscule
- n est le code ASCII d'une lettre majuscule de l'alphabet

3.5.2 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 - -
```

3.5.3 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.

3.5.4 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).

3.5.5 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 en 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.

3.5.6 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'.

3.5.7 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()`.

Chaînes de caractères

3.5.8 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 `"bonjourbbbbb"`. 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 arg. renvoie : mercibbb
Donnez un suffixe: la
La fonction "ajoute_suffixe" appelée avec ces arg. renvoie : mercilalala
```

3.5.9 Calculatrice

Écrire un programme qui lit une chaîne de caractères représentant une opération arithmétique (uniquement + * - ou /), et qui affiche le résultat de cette opération.
Par exemple pour la chaîne "3+4" il affiche 7 ; pour la chaîne "2*3" il affiche 6.

3.5.10 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), et affiche alors le nombre d'erreurs faites avant la bonne saisie.

2. Demande à l'utilisateur un mot, affiche sa longueur, lui propose de rejouer ; à la fin (refus de rejouer), affiche le mot le plus long qui a été lu.

3. Demande à l'utilisateur de saisir des mots jusqu'à ce qu'il indique 'stop' ; affiche alors combien de mots ont été lus ('stop' ne compte pas), le mot le plus long, et sa longueur.

4. Demande des mots jusqu'à la saisie d'une chaîne vide pour arrêter ; affiche alors lequel des mots lus est le premier dans l'ordre alphabétique.

3.5.11 Palindrome

Écrire un programme / une fonction qui reçoit une chaîne de caractères et vérifie si c'est un palindrome. Afficher le résultat sous la forme "le mot XXX est un palindrome" ou "le mot XXX n'est pas un palindrome".

3.5.12 Premier mot

Écrire une fonction qui reçoit une chaîne et renvoie le premier mot de cette chaîne. Indice : on considère n'importe quel caractère non alphabétique (espace, ponctuation...) comme séparant les mots.

Par exemple pour la chaîne "bonjour à tous", la fonction renvoie "bonjour".

3.5.13 Verlan

Écrire une fonction qui reçoit une chaîne de caractères et la renvoie à l'envers.

3.5.14 Sans voyelles

Écrire une fonction qui reçoit une chaîne de caractères et renvoie cette chaîne dont on a retiré toutes les voyelles.

3.5.15 Ordre alphabétique

Écrire une fonction qui demande à l'utilisateur de rentrer des chaînes de caractères en ordre alphabétique. La fonction s'arrête dès que l'utilisateur tape 'stop' ou tape un mot qui n'est pas dans l'ordre alphabétique. Elle renvoie alors le nombre de mots saisis (sans compter 'stop' ou le mot qui a causé l'arrêt).

Exemple d'exécution (le mot 'marche' cause l'arrêt car il est avant 'voiture' dans l'ordre alphabétique, la fonction renvoie donc 5, le nombre de mots saisis avant 'marche') :

```
Tape des mots dans l'ordre alphabétique ou stop pour arrêter
avion
bateau
train
velo
voiture
marche
5
```

Plus d'exercices sur les chaînes de caractères dans le TD 3.7 (avec boucle for).

3.6 Listes

Basiques

3.6.1 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"]
```

3.6.2 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.

3.6.3 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
- *Comment ne pas afficher de point-virgule après le dernier élément ?*
- Sur plusieurs lignes, avec 10 éléments par ligne (éventuellement moins sur la dernière) séparés par des espaces
- *Plus difficile* : sur plusieurs colonnes, avec 7 éléments par colonne, par exemple pour l'alphabet :

```
A  H  O  V
B  I  P  W
C  J  Q  X
D  K  R  Y
E  L  S  Z
F  M  T
G  N  U
```

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`*.*

3.6.4 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.*

3.6.5 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.*

3.6.6 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.*

3.6.7 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.*

3.6.8 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 de nombres

3.6.9 Puissances

Écrire une fonction qui reçoit en paramètre un entier n, et qui renvoie la liste des puissances de 2, de la puissance 0 à la puissance n incluse. Par exemple pour $n = 5$, la fonction renvoie la liste $[1, 2, 4, 8, 16, 32]$.

3.6.10 Compter les multiples

Écrire une fonction qui reçoit en paramètre une liste d'entiers, li et un entier x, qui compte combien d'éléments de li sont multiples de x, et qui renvoie la valeur de ce compteur. Par exemple pour $x = 3$ et $li = [3, 2, 5, 9]$ la fonction renvoie 2. Pour la même liste mais $x = 4$, la fonction renvoie 0.

3.6.11 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.*

3.6.12 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 `compteurs_faces` qui reçoit une liste de n tirages, et calcule et renvoie une liste de 6 compteurs indiquant le nombre d'apparitions de chaque face dans cette liste. On s'appliquera à ne parcourir **qu'une seule fois** la liste de tirages, et donc on s'abstiendra d'utiliser `count`.

3. 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). On utilisera la fonction `compteurs_faces`.

4. 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). On utilisera la fonction `compteurs_faces`.

5. 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.

3.6.13 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 3.5.6) 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 2.9.4), 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.

3.6.14 Comptage alphabétique (force brute vs subtile)

1. Écrire une fonction `compte_carac` 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 `compte_alphab` 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. Écrire une nouvelle fonction `compte_alphab2` qui compte toutes les lettres de l'alphabet en un seul parcours de la liste de caractères, et renvoie la liste des 26 compteurs.

5. Bonus avec des dictionnaires : écrire une fonction `compte_alphab_dico` 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. Cette fonction renvoie un dictionnaire dont les clés sont les caractères du texte, et dont les valeurs associées sont les compteurs correspondants.

3.6.15 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 : accordeon
['train','cheval','voiture','avion','accordeon']
```

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 accordeon: 9
Taille moyenne: 6.4
```

4. Compléter pour afficher aussi les mots plus longs que la taille moyenne, sur une seule ligne.

```
Mots plus longs que la moyenne: voiture ; accordeon ;
```

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. Utiliser la fonction `nbocc` pour écrire une fonction *compteCarac(liste,car)* qui reçoit une liste de chaînes et un caractère, affiche les mots contenant ce caractère et le nombre total d'occurrences de ce caractère dans tous les mots. Si ce caractère n'est présent dans aucun mot, la fonction affiche un message d'erreur à la place. Cette fonction ne renvoie rien. Deux exemples d'exécution (avec la liste précédente de 5 mots et la lettre 'o' puis 'w') :

```
Mots contenants le caractère o :
voiture
```

```
avion
accordéon
Le caractère o apparaît 4 fois.

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

7. Utiliser la fonction `nbocc` 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, le compte dans les mots, et affiche le mot qui contient le plus de fois ce caractère.

9. Comment modifier ce programme pour proposer à l'utilisateur de rejouer la dernière étape (même liste, nouveau caractère à compter) jusqu'à ce qu'il refuse ?

3.7 Itération inconditionnelle `for`

Quelques exercices basiques pour commencer

3.7.1 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)
```

3.7.2 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 ex. pour les bornes 1 et 20, et un pas de 3, il doit afficher : 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

3.7.3 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. f3 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).

3.7.4 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).

3.7.5 Comptons les moutons

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

```
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.

3.7.6 Règle graduée

Dans cet exercice on veut utiliser des boucles for, on s'interdit d'utiliser la multiplication de chaînes (par exemple n*'-' 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...

3.7.7 Accumulateurs

1. Écrire une fonction qui utilise un `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 : $a^{-1} = 1/a$)

3.7.8 Pile ou face, et statistiques

1. Écrire une fonction `nb_pile(T)` 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 `liste_nbpile(N,T)` 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 (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 `compte_parties_xpile(liste_pile,x)` 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 écrire une fonction `liste_cpt_parties` qui construit une liste de compteurs correspondant au nombre de parties qui se sont conclues par chaque nombre de tirages Pile possible (T+1 valeurs, entre 0 si on a fait Face à chaque fois, et T si on a fait Pile à chaque fois). La fonction renvoie cette nouvelle liste.

(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

On rappelle que les chaînes de caractères sont un type itérable, c'est-à-dire dont on peut parcourir les éléments (les caractères individuels) à l'aide d'une boucle. On l'a déjà pratiqué avec des boucles `while` en parcourant les indices des éléments, on va maintenant le pratiquer avec les boucles `for` en parcourant soit les indices, soit directement les éléments. On pourra aussi refaire les exercices du TD 3.5 en utilisant `for` plutôt que `while`.

3.7.9 Modification de chaînes

1. Écrire une fonction qui supprime tous les espaces d'une chaîne de caractères reçue, et renvoie le résultat.

2. Écrire une fonction qui reçoit une chaîne de caractères, et renvoie son inverse. Par exemple :

```
>>> inverse_chaîne("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).

3.7.10 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.

3.7.11 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.

```
>>> 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
```

3.7.12 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. Écrire une fonction `genereCode` qui génère et renvoie une liste mélangée de l'alphabet.

3. On souhaite écrire un programme principal qui lit un texte (en minuscules sans accents), puis le crypte en utilisant le principe de "codage par décalage de lettres". Pour cela, on utilise les fonctions précédentes pour générer une liste mélangée de caractères, et décaler les caractères du texte selon l'ordre de cette liste. Les caractères non alphabétiques ne sont pas modifiés. On affichera le résultat obtenu.

Manipulation de listes

Les listes sont aussi un type itérable, c'est-à-dire dont on peut parcourir les éléments (qui peuvent être de n'importe quel type) à l'aide d'une boucle. On l'a déjà pratiqué avec des boucles `while` en parcourant les indices des éléments, on va maintenant le pratiquer avec les boucles `for` en parcourant soit les indices, soit directement les éléments. On pourra aussi refaire les exercices du TD 3.6 en utilisant `for` plutôt que `while`.

3.7.13 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. On ne veut pas utiliser la fonction `reverse()` qui existe déjà en Python.

3.7.14 Liste de mots

1. Écrire une fonction qui reçoit une liste de mots *lm*, un caractère *c*, et qui renvoie une liste des mots de *lm* qui se terminent par *c*.

2. Écrire une fonction similaire mais qui renvoie la liste des mots de *lm* qui commencent par *c*

3. **Bonus (fonctions avancées) :** Écrire une fonction similaire avec un paramètre optionnel booléen permettant de choisir si l'on veut que le mot commence ou termine par la lettre *c*.

4. Écrire une fonction avec les mêmes paramètres qui renvoie la liste des mots de *lm* qui contiennent la lettre *c*.

5. Écrire une fonction similaire avec un paramètre supplémentaire *n* (bonus : optionnel, valeur par défaut = 1) qui renvoie la liste des mots de *lm* contenant au moins *n* fois la lettre *c*.

3.8 Listes avancées : `for`, tri, listes à 2 dimensions, etc

Ces exercices sont réalisables avec une boucle `while`, mais seront plus faciles avec la boucle `for` (voir cours 2.11).

3.8.1 Mélange de liste

1. Écrire une fonction qui reçoit une liste, et en crée une copie mélangée qu'elle renvoie. On ne veut pas modifier la liste reçue. On ne veut pas utiliser la fonction existante `shuffle`, mais la réécrire. On pourra utiliser le module `random`.

2. Écrire une fonction qui reçoit deux listes de même taille, contenant les mêmes éléments dans le désordre, et qui compte le nombre d'éléments qui sont à la même position dans les 2 listes. Cette fonction renvoie ce compteur.

3. Écrire une fonction qui demande à l'utilisateur de saisir un par un des entiers positifs, avec -1 pour finir la saisie (-1 n'est pas ajouté à la liste). La fonction renvoie alors la liste ainsi créée.

4. Écrire un programme principal qui lit une liste saisie par l'utilisateur (avec la fonction ci-dessus), puis qui génère un mélange de cette liste en boucle jusqu'à ce qu'aucun élément ne soit à la même place dans la copie mélangée. Le programme affiche alors le nombre de mélanges qui ont été nécessaires pour atteindre ce résultat.

3.8.2 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. Écrire 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 ?

3.8.3 Jouons avec des listes triées

1. Soit la fonction `InsererDans` ayant deux arguments `el` et *liste_triee*, une liste de nombres classée en ordre croissant. La fonction ne doit pas la modifier ! La fonction `InsererDans(el,` *liste_triee*`)` retourne une liste dans laquelle l'élément `el` 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]
```

3.8.4 Tri par sélection

Écrire une fonction qui reçoit une liste en paramètre, et la modifie pour la trier. Cette fonction ne renvoie rien, par contre elle a 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 (partie triée de la liste : vide, partie 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
```

3.8.5 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[1], 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 :
  - 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 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. https://fr.wikipedia.org/wiki/Formule_de_Luhn

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 renvoie un numéro **valide** de n décimales généré aléatoirement. On utilisera la fonction *randint* du module *random*, et la fonction précédente.

Listes de listes, ou listes "à 2 dimensions"

3.8.6 Jeu d'échecs

On considère une liste `board` de 8 listes de 8 chaînes, représentant un échiquier. Chaque case contient une chaîne indiquant la pièce présente sur la case, la chaîne vide si aucune pièce n'est présente, sinon "pion noir", "tour blanche", "roi noir", etc.

1. Écrire une fonction qui reçoit une liste `board`, une lettre (entre "A" et "H") représentant la colonne, et un chiffre (entre 1 et 8) représentant la ligne, et qui renvoie le contenu de la case correspondante de l'échiquier.
2. Écrire une fonction qui lit et filtre une colonne et une ligne, et renvoie ces 2 valeurs une fois correctes (représentent bien une case du plateau). ***(Cf cours : fonctions à retour multiple.)***
3. Écrire une fonction qui reçoit un échiquier, les coordonnées d'une case, et renvoie 2 chaînes indiquant le type et la couleur de la pièce sur cette case (ou 2 chaînes vides si la case est vide). ***Pensez à la fonction split().***
4. Écrire une fonction qui reçoit un type de pièce, et ses coordonnées (par exemple "cavalier", "A", et 3) et renvoie la liste de toutes les cases d'arrivée possibles de cette pièce. On remarquera qu'on ne reçoit pas l'échiquier en paramètre : on ne veut pour l'instant pas considérer les autres pièces qui pourraient interférer avec le déplacement.

3.9 Boucles `for` et maths

Exercices de révision sur les boucles `for` et les listes, à thème "mathématiques". Ce sont des exos classiques d'examen.

3.9.1 Diviseurs propres et nombres parfaits

1. Écrire une fonction qui reçoit un entier naturel n et renvoie la liste de ses diviseurs (y compris 1 et lui-même).

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)        >>> sommeDivPropre(3)
0                            1
>>> sommeDivPropre(1)        >>> sommeDivPropre(6)
0                            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).

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

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

3.9.2 Nombres de Armstrong

Un nombre est dit de Armstrong s'il est égal à la somme des cubes de ses chiffres. Par exemple 153 est un nombre de Armstrong car $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.

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 et affiche la liste des nombres de Armstrong inférieurs à 1000.

3.9.3 Triangle de Pascal

1. Écrire une fonction `factorielle` qui prend en argument un entier naturel `n` et qui renvoie la valeur de $n! = 1 \times 2 \times \cdots \times n$.

2. 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 :

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

3. 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.

4. *Bonus :* Comment faire pour éviter le décalage causé par les nombres à plusieurs chiffres ?

3.9.4 Approximation de la racine carrée

Étant donné un réel positif a, on définit la suite réelle x_n de la manière suivante. Cette suite converge vers $\sqrt{a}$.

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

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.

3.9.5 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. *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.*

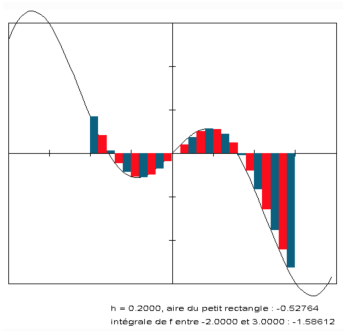
2. Écrire une fonction `estim_incond` qui demande à l'utilisateur un entier nmax, et affiche successivement toutes les approximations (calculées avec la fonction ci-dessus) pour toutes les valeurs de n entre 1 et nmax.

```
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
```

5. Écrire une fonction `estim_cond` 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).

3.9.6 Approximation de l'intégrale d'une fonction

On veut calculer l'approximation de l'intégrale[2] 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.



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

2. Le package `scipy.integrate` fournit des fonctions de calcul d'intégrales
https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html

3.10 Fonctions avancées

Paramètres optionnels

3.10.1 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.

3.10.2 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.

3.10.3 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.

3.10.4 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.

3.10.5 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.

```
>>> 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
```

3.10.6 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 a 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.

5. 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 ?

Retours multiples

3.10.7 Occurrences

- Écrire une fonction qui reçoit une chaîne de caractères, et renvoie le caractère apparaissant le plus souvent dans cette chaîne, ainsi que son nombre d'occurrences.
- Écrire un programme principal qui lit une chaîne au clavier, puis affiche le résultat dans un message clair.

3.10.8 Modulo

- Écrire une fonction qui reçoit 2 entiers *a* et *b*, et calcule et renvoie le quotient et le reste de la division de *a* par *b*
- Écrire un programme principal qui lit 2 entiers au clavier, puis affiche le résultat avec un message explicite.

3.10.9 Liste d'entiers

1. 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.
2. Écrire le programme principal qui permet d'appeler cette fonction et d'afficher la liste renvoyée.
3. Écrire une fonction `LireListeReelsBornes` avec deux arguments facultatifs bmin et bmax (valeurs par défaut 0 et 100 respectivement), qui lit des réels saisis au clavier, jusqu'à ce que l'utilisateur saisisse une valeur hors de l'intervalle délimité par les 2 bornes (inclues). La fonction renvoie la liste des éléments (corrects) saisis.
4. 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.
5. Écrire un programme principal qui lit une liste de réels au clavier ; en calcule le minimum, le maximum et la somme ; et les affiche dans la console.

3.11 Dictionnaires

3.11.1 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'][4])]
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.

3.11.2 Compteurs alphabétiques

Utiliser un dictionnaire pour compter le nombre d'occurrences de chaque caractère dans une chaîne de caractères reçue en argument. Le dictionnaire ne doit contenir que les caractères qui apparaissent dans la chaîne. On ne veut parcourir la chaîne qu'une seule fois.

3.11.3 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).

3.11.4 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 la plaque d'immatriculation de la personne dont le nom (complet) 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)

3.11.5 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"]
```

3.11.6 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 déjà une clé du dictionnaire (dans ce cas afficher un message d'erreur).

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 (on appellera la fonction `ajoute`)
   - Affiche ce dictionnaire (avec la fonction `affiche` ci-dessus)
   - Lit ensuite un mot français (le filtre pour qu'il ne soit pas déjà dans le dictionnaire), demande ensuite sa traduction en anglais, et l'ajoute au dictionnaire
   - Lit une lettre de l'alphabet, et supprime du dictionnaire les mots commençant par cette lettre. Est-il nécessaire de filtrer la lettre lue ?
   - Affiche à nouveau le dictionnaire

3.11.7 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 *lire(n)* qui reçoit un entier n (le degré du polynôme), et qui demande à l'utilisateur les différents coefficients dans l'ordre décroissant des degrés (ordre intuitif d'écriture), et qui renvoie le dictionnaire représentant ce polynôme. Dans l'exemple ci-dessous pour $n = 3$, l'utilisateur saisit le polynôme $2 * x^3 - 5 * x^2 + x - 7$.

```
Coeff de x**3? 2
Coeff de x**2? -5
Coeff de x**1? 1
Coeff de x**0? -7
{3:2, 2:-5, 1:1, 0:-7}
```

2. 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.

3. Écrire une fonction `simplifier(p)` qui reçoit en argument un polynôme p (sous forme d'un dictionnaire), et modifie ce dictionnaire pour le simplifier, en supprimant les clés dont la valeur est 0. Cette fonction ne renvoie rien. *Par exemple la simplification du polynôme 2 :1, 1 :0, 0 :-3 est le dictionnaire 2 :1, 0 :-3.*

4. Écrire une fonction `affiche(p)` qui reçoit un polynôme et l'affiche "joliment". Par exemple le dictionnaire 2 :1, 1 :3, 0 :-3 sera affiché : $x^2 + 3 * x - 3$. On remarque qu'on n'affiche pas les monômes nuls, ni le coefficient multiplicateur ou la puissance si égal(e) à 1. On peut commencer par une version simple sans cas particulier, qui affichera alors : $1 * x^2 + 3 * x^1 - 3 * x^0$.

5. 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.

6. Écrire une fonction *produit_polynomes(p1, p2)* qui prend deux polynômes en arguments 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.

7. Écrire une fonction *derivee_polynome(p)* qui reçoit en argument un polynôme et qui renvoie sa dérivée. Attention on ne doit pas modifier p, il faut créer un nouveau dictionnaire.

```
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}
>>> derivee({5:1,3:2,2:-1,1:7,0:-4})
{4: 5, 2: 6, 1: -2, 0: 7}
```

3.11.8 Acides aminés

En utilisant un dictionnaire, déterminez le nombre d'occurrences de chaque acide aminé dans la séquence AGWPSGGASAGLAILWGASAIMPGALW. Le dictionnaire ne doit contenir que les acides aminés présents dans la séquence.

3.11.9 Séquençage génétique

1. Créez une fonction `compte2(seq)` qui prend comme argument une séquence sous la forme d'une chaîne de caractères et qui renvoie tous les mots de 2 lettres qui existent dans la séquence sous la forme d'un dictionnaire. *Par exemple pour la séquence ACCTAGCCCTA, le dictionnaire renvoyée serait : 'AC' : 1, 'CC' : 3, 'CT' : 2, 'TA' : 2, 'AG' : 1, 'GC' : 1*

2. Créez une nouvelle fonction `compten(seq,n)` qui a un comportement similaire mais qui compte tous les mots de n lettres, et renvoie le dictionnaire résultant.

3. Écrire un programme principal utilisant ces fonctions pour afficher les mots de 2 et 3 lettres et leurs occurrences trouvés dans la séquence d'ADN suivante : ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG

3.11.10 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).

3.11.11 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).

3.11.12 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

3.11.13 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.

3.11.14 Rendez-vous

Alice et Bob ont tous les deux un emploi du temps électronique. Chaque évènement 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 évènements 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 évènement 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 évènement 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 retraite 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 retraite 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 évènements 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 retraite 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 retraite Joe', 'debut': 720}]
```

3.12 Fichiers

3.12.1 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.

3.12.2 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

3.12.3 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.

3.12.4 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 côtes. 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 côtes). *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 .

3.12.5 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 differentes 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.

3.12.6 Fichier CSV

Le format CSV (*Comma-separated values*, valeurs séparées par des virgules) est un format de fichier texte permettant de stocker des données tabulaires, sous forme de valeurs séparées par des virgules. Chaque ligne du texte correspond à une ligne du tableau, et les virgules correspondent aux séparations entre colonnes. Chaque portion de texte entre les virgules correspond donc à une cellule du tableau.

1. Écrire une fonction qui reçoit le nom d'un fichier CSV, qui l'ouvre, lit son contenu, et crée une liste à 2 dimensions représentant son contenu sous forme tabulaire. Ne pas oublier de fermer le fichier après usage.

```
1,2,3
2,4,9              ==> [[1,2,3],[2,4,9],[3,8,27]]
3,8,27
```

2. On suppose maintenant que la première ligne du fichier fournit les noms des colonnes, et les lignes suivantes les valeurs. On veut écrire une fonction qui crée à partir de ce fichier une liste de dictionnaires, chaque dictionnaire représente une ligne de valeurs du fichier, les clés sont les noms présents sur la première ligne.

```
Nom,Naissance,Ville
'Toto',1977,'Ajaccio'     [{Nom:Toto,Naissance:1977,Ville:Ajaccio},
'Alice',2000,'Calvi'  =>   {Nom:Alice,Naissance:2000,Ville:Calvi}
'Bob',1998,'Grenoble'      {Nom:Bob,Naissance:1998,Ville:Grenoble}]
```

3. Écrire une fonction qui reçoit un dictionnaire et une chaîne de caractères, et crée un fichier CSV au format ci-dessus, nommé comme indiqué par la chaîne reçue en paramètre, pour représenter le contenu de ce dictionnaire.

3.12.7 Zoo et devinettes

On suppose qu'on dispose d'une liste de dictionnaires représentant des animaux sous la forme suivante :

```
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0},
 {'nom': 'singe', 'poids': 50, 'taille': 1.75},
 {'nom': 'guepard', 'poids': 50, 'taille': 1.20},
 {'nom': 'lion', 'poids': 200, 'taille': 1.20},
 ...]
```

1. Écrire une fonction qui reçoit une chaîne de caractères (nom d'un critère, par exemple 'poids' ou 'taille') et qui renvoie le nom de l'animal ayant la plus grande valeur de ce critère.

2. Écrire une fonction `devinette()` qui demande à l'utilisateur de penser à un animal, puis qui lui pose des questions pour deviner de quel animal il s'agit. Si plusieurs animaux ont la même valeur d'un critère, il faudra poser plusieurs questions pour raffiner. L'ordre des questions pourra être choisi aléatoirement.

```
Pense a un animal
Quel est le poids de ton animal ? 50
Quelle est la taille de ton animal ? 1.20
C'est : guépard !
```

3. Écrire une fonction `info()` qui répond aux questions de l'utilisateur en suivant le modèle d'exécution ci-dessous.

```
Infos sur les animaux
Choisis un animal parmi : girafe, singe, guépard, lion ? éléphant
Animal inconnu
Choisis un animal parmi : girafe, singe, guépard, lion ? lion
Que veux-tu savoir ? âge
Information inconnue
Que veux-tu savoir ? taille
```

```
taille de lion = 1.20 m
Que veux-tu savoir ? poids
poids de lion = 200 kg
Que veux-tu savoir ? stop
Choisis un animal parmi : girafe, singe, guépard, lion ? stop
Au-revoir
```

Chapitre 4

EXERCICES DE TP

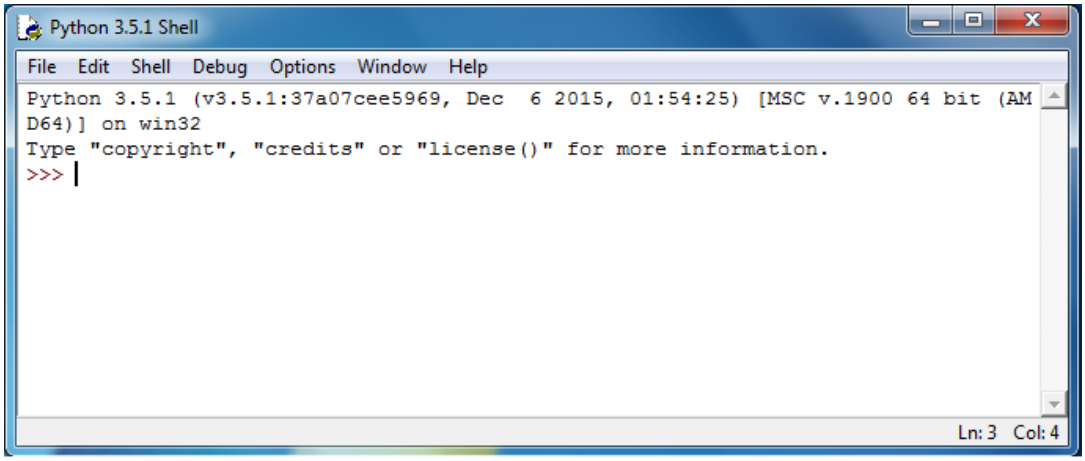
Chaque TP correspond à la mise en application des notions de la semaine correspondante.
CaseIne : des exercices supplémentaires, des fichiers complémentaires pour certains TP, des diapos de cours, et autres ressources, sont disponibles sur le cours en ligne Python sur CaseIne, accessible avec vos identifiants de connection UGA.
http://caseine.org/course/view.php?id=86

4.1 TP1 : prise en main des outils, lecture et 1er programmes

Notions pratiquées : entrées-sorties, instruction conditionnelle if 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.

4.1.1 Première prise en main de Idle

`Idle` est un environnement de développement - c-à-d 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")
>>> 1 + 1
>>> "1" + "1"
```

```
>>> 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 ?

4.1.2 Écriture 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.

- Écrire 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.

4.1.3 Découverte de Python Tutor

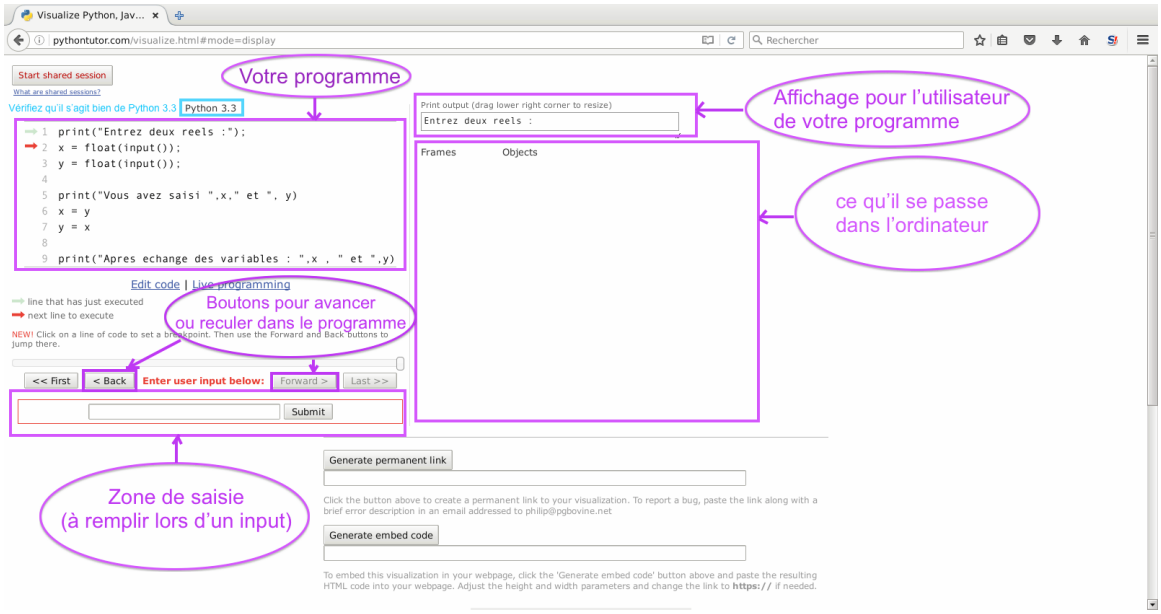
`Python Tutor` est un outil en ligne qui permet de mieux comprendre ce qui se passe quand l'ordinateur exécute un programme (en `Python` mais aussi en d'autres langages de programmation). Cela deviendra particulièrement utile au fil des semaines quand nous aborderons des notions de plus en plus complexes.

- 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("Apres echange 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).

4.1.4 Exercice de lecture

Soit le programme ci-dessous, censé déterminer la valeur absolue de la différence de deux entiers saisis au clavier.

```
# 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.

4.1.5 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. Écrire un programme qui demande deux nombres entiers et affiche un message pour dire si les nombres sont égaux ou différents.

3. Écrire 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). Exemple d'affichage :

   ```
   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

- vérifie que ceux-ci sont corrects (compris entre 1 et 6)
- dans le cas de valeurs correctes, affiche les trois dés dans l'ordre décroissant
- en plus, affiche "Gagné !" si les trois valeurs sont 4 2 1, "Perdu !" sinon (autre cas corrects)
- 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, puis 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 plus la mention de l'étudiant.

		moyenne	< 10	:	AJOURNE
	$10 \leq$	moyenne	< 12	:	PASSABLE
Les mentions sont :	$12 \leq$	moyenne	< 14	:	ASSEZ BIEN
	$14 \leq$	moyenne	< 16	:	BIEN
	$16 \leq$	moyenne		:	TRES BIEN

5. Modifier ce programme pour qu'il donne une 2e chance à l'utilisateur quand il saisit une note incorrecte.

6. **Bonus avec while :** écrire une nouvelle version de ce programme qui filtre chaque note entrée jusqu'à ce qu'elle soit bien comprise entre 0 et 20.

7. **Bonus avec while :** écrire une nouvelle version de ce programme qui ne connaît pas à l'avance le nombre de disciplines du module ni leurs coefficients. Le programme lit les informations nécessaires au fur et à mesure puis affiche la moyenne pondérée.

4.2 TP2 : prix du gros lot

Notions pratiquées : importation de modules, nombres aléatoires, boucles while

4.2.1 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*.

Un module est un fichier Python qui regroupe des variables, des classes et des fonctions, permettant de résoudre des problèmes spécifiques à thème donné. Par exemple, le module 'math' contient des variables comme π, et 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.pi
>>> math.cos(0)
>>> math.sin(math.pi / 2)
>>> math.tan(math.pi / 4)
>>> math.tan(math.pi / 3)
>>> math.log(10)
>>> math.exp(math.log(89))
>>> math.sqrt(16)
```

Pour accéder au contenu du module math (fonctions et constantes telles que π), 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 pour éviter de devoir ensuite les préfixer par le nom du module :

```
>>> from math import cos, pi, exp, log
>>> cos(pi)
...
>>> x = exp(log(2))
```

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.

Exercice de trigonométrie

1. Écrire 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 π.
   Rappels de trigo : $cos(\alpha) = \frac{adjacent}{hypothenuse}$, $sin(\alpha) = \frac{oppose}{hypothenuse}$, et $tan(\alpha) = \frac{oppose}{adjacent}$
   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. Écrire un script qui demande à l'utilisateur 3 nombres : a, b et c en boucle jusqu'à ce que la valeur $delta = b^2 - 4ac$ 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 (cf cours). En particulier, on s'intéressera à sa fonction `randint`.

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 2 entiers a et b, vérifie que $a < b$, et affiche un nombre aléatoire compris entre a et b.

4.2.2 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 mais 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  80
50                                Trop grand
Trop petit                        Devine mon nombre entre 51 et 79
Devine mon nombre entre 51 et 100 ...
```

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 commence, s'il tape 'n', le programme se termine en affichant :

   - Combien de parties le joueur a jouées et combien sont gagnées (format : 'tu as gagné 2 parties sur 5 jouées')
   - Combien d'essais il a utilisé 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 deviner')

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 inférieure ou supérieure en fonction de l'indice répondu par l'utilisateur, et les affiche aussi. Par exemple si vous avez choisi 88 :

```
Je cherche un nbre entre 0 et 100   Je cherche un nbre entre 51 et 92
Je propose 50 ? p                   Je propose 88 ? b
Je cherche un nbre entre 51 et 100  J'ai gagné
Je propose 93 ? g
```

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 ? On peut remarquer que l'IA dichotomique a toujours le même comportement, il n'est donc pas nécessaire de la faire répéter ; par contre il faut répéter l'IA aléatoire pour ”lisser” son comportement.

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 ?

```
[0] ****
[0] ------
[1] ****
[1] -------
etc
```

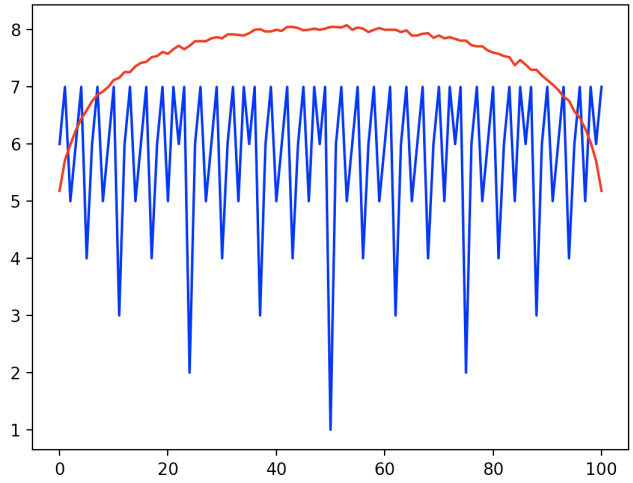
Bonus : affichage graphique

La librairie `matplotlib` permet de créer des graphiques[1]. 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. Par exemple voici le résultat pour 10000 parties. Expliquer ce résultat.

Bonus : stratégies de jeu

Imaginer d'autres algorithmes qui jouent à ce jeu avec d'autres méthodes, puis calculer et afficher leurs statistiques sur le même graphique pour les comparer.

1. Voir par exemple
http://apprendre-python.com/page-creer-graphiques-scientifiques-python-apprendre


8
7
6
5
4
3
2
1
0
20
40
60
80
100

4.3 TP3 : initiation aux fonctions avec `turtle`

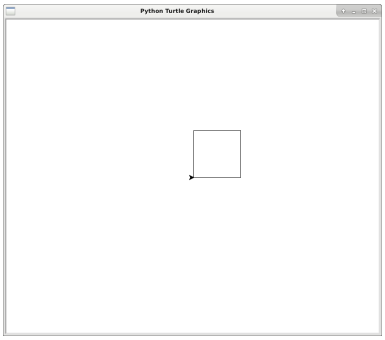
Notions pratiquées : dessin avec le module `turtle`, boucles `while` imbriquées, initiation aux fonctions.

4.3.1 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.

> **ATTENTION !** ne nommez **jamais** votre propre fichier `turtle.py`, sinon c'est lui qui est importé au lieu du module turtle...

Le module `turtle` est un module graphique, inspiré de la programmation `Logo`,[1] 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. Voici une liste non exhaustive[2] des fonctions disponibles dans cette bibliothèque :

- `reset()` : efface l'écran, recentre tortue, remet variables à zéro
- `forward(distance)`, `backward(distance)` : avance (recule) d'une distance donnée (en pixels)
- `left(angle)`, `right(angle)` : pivote vers la gauche (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)`
- `position()`, `xcor()`, `ycor()` : renvoie la position de la tortue, son abscisse, son ordonnée
- `circle(r)` : trace un cercle de rayon `r`
- `setup(largeur,hauteur)` : fixe dimensions de la fenêtre graphique
- `color(str)` : change la couleur du dessin
- `speed(int)` : change la vitesse de déplacement de la tortue

1. 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.

2. 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

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.

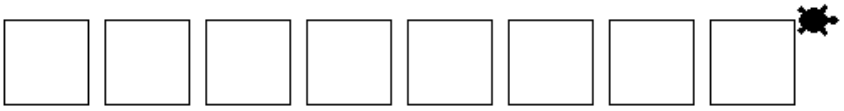
Triangle équilatéral

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.*



Modifiez ce programme pour que le coin supérieur gauche du premier carré (à gauche) ait pour coordonnées (-200,200).

Pavage de carrés

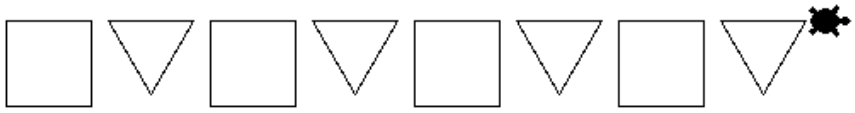
Utiliser une 3^{e}boucle while (imbriquées) pour dessiner un pavage : 4 lignes de 8 carrés de côté 50.

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.

4.3.2 A la découverte des fonctions à travers le dessin

Programme mystère

Écrivez le programme de droite dans un nouveau fichier.
Exécutez ce programme. Quel résultat obtient-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 : éviter de répéter du code (code plus court, moins redondant) ; 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) ; permettre de réutiliser une fonction sans la réécrire ; 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
```

```
# prog. principal, avec appel de fonc.
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 qu'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 de droite et complétez-le avec un 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 carré de taille donnée (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 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.

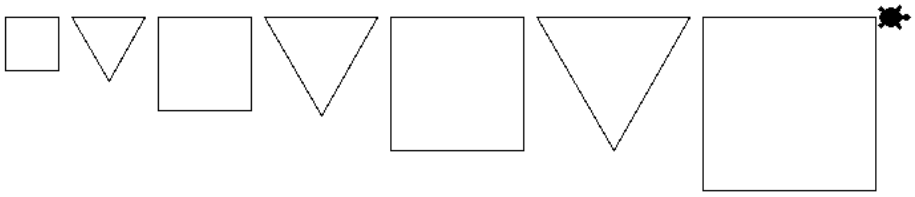
```
import turtle

def carre(cote) :
    ...

def triangle(cote) :
    ...

# programme principal
# trace ligne carres et triangles
...
```

4.3.3 Pour aller plus loin

Si vous avez fini l'ensemble de ce qui précède, vous pouvez essayer de réaliser les figures des exercices suivants.

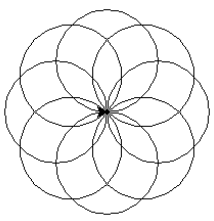
Ligne de carrés et triangles alternés de tailles croissantes

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. L'espacement entre les figures reste le même.



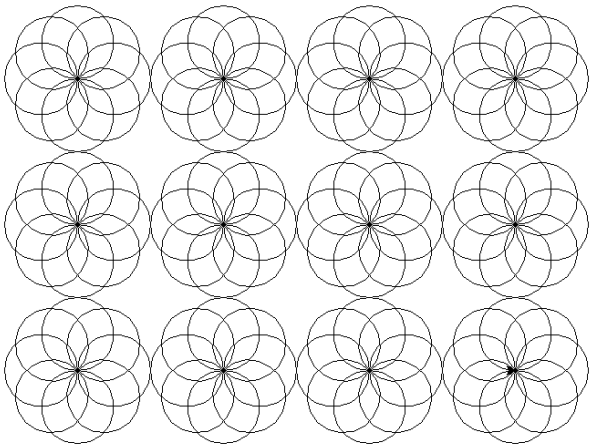
Rosace et pavage de rosaces

1. Définir une fonction `rosace(x, y, nb, r)` qui trace une rosace centrée sur 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 autre fonction qui trace une ligne de `tl` rosaces (paramètre).

3. Utilisez cette fonction (ligne) pour écrire une fonction qui trace un pavage de rosaces avec `tp` lignes (paramètre), comme sur la figure ci-dessous (pour `nb=8`, `tl=4` et `tp=3`) :



4. Bonus : modifier les fonctions précédentes pour choisir aléatoirement une nouvelle couleur pour chaque rosace, ou pour chaque cercle des rosaces.

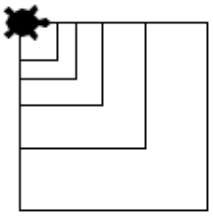
4.3.4 Figures imbriquées

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).

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 au TP3).

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.

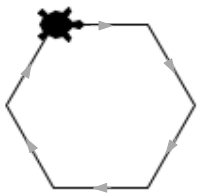
3. Bonus : écrire une fonction qui fait se promener la tortue au hasard : distance aléatoire, angle aléatoire. Elle pourra laisser des empreintes à chaque pas avec `stamp()` ; dans ce cas on pourra changer la forme du curseur avec `turtle.shape('turtle')`, et aussi choisir une couleur aléatoire (avec `random.choice` par exemple qui peut choisir un élément d'une liste, ou bien écrivez votre propre fonction de sélection de couleur). Enfin on pourra faire en sorte pour ne pas perdre la tortue de vue, que quand elle sort de la fenêtre d'un côté, elle y re-rentre aussitôt de l'autre côté (environnement torique). Rappel : `turtle.screensize()` fournit les dimensions de la fenêtre, et `turtle.setup(x,y)` permet de les modifier.

4. Bonus (cours sur les fonctions avancées) : 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 cotés et c la longueur du côté (n'oubliez pas d'importer le module `math`) :



`polygone(6,50)`

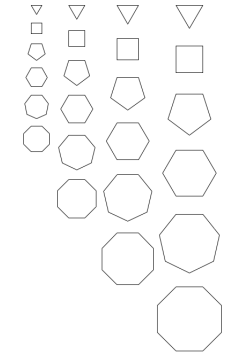
$$\frac{c}{sin\,(\pi/n)}$$

Pavage de polygones

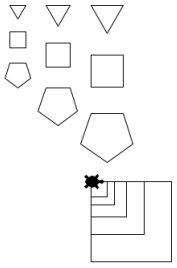
1. Écrire une fonction `colonne_polygone(nb_poly, cote)` qui trace `nb_poly` polygones sur la même colonne, en commençant par un triangle et en augmentant le nombre de côtés de 1 à chaque fois. La longueur du côté des polygones doit toujours être égale à `cote`. Vous vous servirez des fonctions `descendre_sans_tracer` 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 coté de 20 ; puis qui dessine 5 carrés imbriqués, dont le premier a pour longueur 100.



`pavage(6,4,20)`



`prog. principal`

4.3.5 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.
- Attention à empêcher la tortue de sortir de la fenêtre.
- S'assurer que le labyrinthe a une sortie.

Exercices sur Caseine

Faire les exercices sur Caseine sur les boucles `while`.

4.4 TP4 : création et manipulation de fonctions basiques

Notions pratiquées : fonctions.

Des fichiers sont à télécharger sur CaseIne :
http://caseine.org/mod/resource/view.php?id=3296

4.4.1 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 sauver 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 solution de
    # ax^2+bx+c=0
    y=a*x**2+b*x+c
    rep= (y==0)
    return rep
# prog. principal
est_solution(1,2,3,4)
print(rep)
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 dist. 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
    return (distance(0,0,xA,yA)==rayon)
# 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))
```

Fonction `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("Var. loc. 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.

```
# Programme 1
def incremente (a):
    a = a+1

# prog. principal
a=5
incremente(a)
print (a)
b=3
incremente(b)
print (b)
```

```
# Programme 2
def incremente2(a):
    return a+1
# prog. principal
a=3
incremente2(a)
print(a)
b=1
b = incremente2(b)
print(b)
a=incremente2(b)
print(a)
```

Maintenant, rajoutez 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.

4.4.2 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_different` 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 $f(x)$.

Polynôme

1. 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 : $\Delta = b^2 - 4ac$.

2. Écrire une fonction `resolution` qui prend en arguments 3 réels a, b, c, qui résout le polynôme $ax^2 + bx + c$, et qui **affiche** le nombre et la valeurs de ses racines.

3. Écrire un programme principal qui : demande à l'utilisateur 3 réels a, b, c ; tire au hasard 1 entier x et l'affiche ; affiche les racines du polynôme $a * x^2 + bx + c$.

4.4.3 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 un booléen indiquant si le capital au bout de `nb_annees` sur un tel compte est (ou non) supérieur ou égal au capital de début.

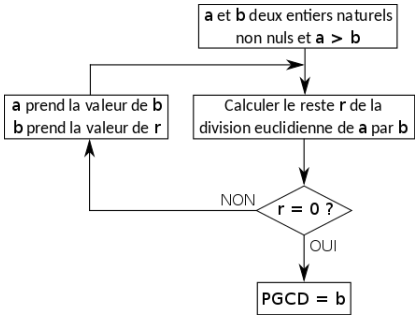
3. Écrire une fonction `placement_min(nb_annees,but)` qui calcule le placement minimum nécessaire pour atteindre au moins le capital `but` après `nb_annees` d'économies.

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.

5. Écrire un programme principal qui teste ces fonctions.

4.4.4 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 (voir schéma), beaucoup plus rapide : 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.
   - Bonus : comparer le temps de calcul et le nombre de comparaisons faites par chaque version. (Le module `time` fournit une fonction `time()` qui renvoie le temps système

en secondes, utilisable pour calculer le temps d'exécution d'un programme. Le module `timeit` [3] permet aussi de chronométrer de courts extraits de code.)

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 calcule un booléen indiquant si la fraction correspondante $\frac{numerateur}{denominateur}$ est irréductible ou non.
5. Difficile : écrire une fonction qui reçoit 2 entiers pgcd et ppcm, et qui affiche tous les couples d'entiers a et b dans l'ordre croissant dont ces nombres sont le pgcd et le ppcm.

4.4.5 Suite de Fibonacci

Soit la suite de Fibonacci donnée par :

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

Écrire des fonctions pour :

1. Afficher tous les termes un par un jusqu'au n-ième
2. Renvoyer le n-ième terme sans rien afficher (sans listes)
3. Bonus avec listes : renvoyer la liste des n premiers termes de la suite
4. Renvoyer la n-ième estimation du nombre d'or ($\frac{f_n}{f_{n-1}}$. Attention : ne calculez pas tous les termes 2 fois !
5. Calculer le nombre d'or avec une précision donnée, renvoyer le n nécessaire pour l'atteindre
6. Bonus : tracer la fonction $\frac{F_n}{F_{n-1}}$ et la droite NBOR avec le module `matplotlib` pour visualiser la convergence de cette estimation.

4.4.6 Suite de Syracuse

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

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

Écrire des fonctions pour :

1. Calculer la durée de vol pour atteindre 1 à partir d'un A donné en paramètre
2. Demander A à l'utilisateur et afficher la durée de vol, en appelant la première fonction
3. Version modifiée qui boucle et redemande une valeur de A jusqu'à ce que l'utilisateur refuse de rejouer
4. Trouver et renvoyer la valeur de A (entre 1 et une borne B reçue en paramètre) qui a la plus longue durée de vol

3. https ://docs.python.org/2/library/timeit.html

5. Afficher les durées de vol pour chaque valeur de A entre 1 et une borne X reçue en paramètre.
6. Calculer la longueur de la plus longue suite décroissante de termes
7. Bonus : renvoyer la liste des n premiers termes pour un A donné en argument (et n donné en argument)
8. **Bonus graphique :** afficher les durées de vol pour chaque A de départ sous la forme d'un histogramme de rectangles avec le module `turtle`. *Bonus : calculer la bonne largeur des rectangles en fonction de la borne max pour que l'histogramme contienne juste dans la fenêtre.*
9. **Bonus graphique :** afficher ce même graphique mais sous la forme d'une courbe avec le module `matplotlib`.

4.4.7 On travaille en ligne !

Quand vous aurez fini, travaillez sur les exercices de PLM (Programmers' Learning Marchine) : https://plm.telecomnancy.univ-lorraine.fr/#/
Sélectionner Python au lieu de Java et choisir la section sur les boucles `while` et les fonctions.

4.5 TP5 : manipulation de fonctions (suite), chaînes de caractères

Notions pratiquées : fonctions, chaînes de caractères.

4.5.1 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(mot)` qui transforme le mot reçu selon la règle du lundi. Par exemple, `lundi("bonjour")` doit valoir `"bonjour bonjour"`. *Rappel : l'opérateur + permet de concaténer des chaînes.*

2. Chaque mardi, les mots de longueur paire sont répétés 6 fois séparées par un tiret, alors que les mots de longueur impaire sont répétés 3 fois séparées par une virgules. Écrire la fonction `mardi(mot)` qui transforme le mot reçu selon cette règle du mardi. *Rappel : la longueur d'une chaîne de caractères est donnée par `len(chaine)`.*

3. Chaque mercredi, les mots de longueur impaire sont remplacés par le mot `"impair"`, les autrtes ne sont pas modifiés. Écrire la fonction `mercredi(mot)`.

4. Chaque jeudi, les mots sont 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"`. *Rappel : l'opérateur modulo en Python s'écrit %*

5. 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.

On complique un peu les choses...

6. Le vendredi, les mots commençant par une majuscule sont privés de leur dernière lettre ; les autres sont privés de leur première lettre. Écrire une fonction `vendredi(mot)`. *Par ex : "Hello" → "Hell" mais "hello" → "ello".*

7. Écrire une fonction `samedi(mot)` qui renvoie le mot écrit à l'envers. Par ex samedi("bonjour") renvoie "ruojnob".

8. Écrire une fonction `dimanche(mot)` qui transforme le mot en ajoutant un espace entre chaque lettre. Par ex dimanche("hello") renvoie "h e l l o".

9. Compléter le programme principal avec ces 2 jours.

10. Inventez d'autres transformations : remplacer les lettres par leur position alphabétique ; changer la casse (min/ maj) ; effacer les voyelles, ou les majuscules ; répéter autant que le nombre de voyelles ; etc

Quelques bonus pour aller plus loin

11. **Bonus :** à partir d'un mot transformé, peut-on deviner le numéro du jour ?

12. **Bonus avec listes :** écrire un programme principal qui lit une phrase complète, la divise en une liste de mots avec `split()`, transforme chaque mot selon le jour, et affiche la phrase résultante.

13. **Bonus avec fichiers :** écrire un programme principal qui lit un texte dans un fichier, le transforme selon le jour, et sauve le résultat dans un nouveau fichier.

4.5.2 Chiffrement de texte

Chiffre de César Écrire une fonction qui reçoit un mot en majuscules, et le renvoie chiffré en décalant chaque lettre de 3 positions (le 'A' devient 'D', le 'B' devient 'E', le 'Z' devient 'C', etc). Par exemple si la fonction reçoit le mot 'PYTHON' elle renvoie 'SBWKRQ'. On pourra commencer par écrire une fonction auxilliaire qui décale une lettre, puis l'appeler dans la fonction de chiffrement.

Chiffrement d'un mot Écrire une fonction qui reçoit un mot (contenant des majuscules et des minuscules) et qui renvoie ce mot chiffré 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"

On pourra commencer par écrire une fonction qui reçoit une lettre (majuscule ou minuscule) et renvoie sa position dans l'alphabet. Par exemple la position de 'a' est 1, la position de 'E' est 5.

Chiffrement d'un texte Écrire une fonction qui reçoit un texte (qu'on suppose non accentué) et qui 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.

4.5.3 Jeu du pendu

Créer un jeu de pendu

- Le programme choisit un mot au hasard dans une liste et donne X erreurs maximum à l'utilisateur pour le deviner, lettre par lettre. Il affiche un tiret par lettre du mot.
- L'utilisateur choisit une lettre, le programme lui répond si elle apparaît ou non dans le mot. Si elle apparaît, le squelette du mot à trouver est affiché avec toutes les occurrences de cette lettre. Si elle n'apparaît pas, le nombre d'erreurs encore autorisées diminue de 1.
- Le jeu se termine quand l'utilisateur a trouvé toutes les lettres du mot (victoire) ou quand il a consommé toutes ses erreurs (défaite).

- Améliorations à faire ensuite : afficher à chaque tour le nombre d'erreurs encore autorisées, et les lettres déjà testées (pas présentes dans le mot) ; afficher le nombre d'occurrences d'une lettre correctement devinée
- Bonus avec fichiers : lire les mots possibles dans un fichier et en choisir un au hasard comme mot secret en début de partie.

Exemple d'exécution

```
Devine mon mot : _ _ _ _ _ _
Tu as droit a 3 erreurs
a
Il n'y a pas de a
Devine mon mot : _ _ _ _ _ _
Tu as droit a 2 erreurs (a)
p
Il y a 1 p
Devine mon mot : p _ _ _ _ _
e
Il n'y a pas de e
Devine mon mot : p _ _ _ _ _
Tu as droit a 1 erreur (a e)
y
Il y a 1 y
Devine mon mot : p y _ _ _ _
Tu as droit a 1 erreur (a e)
u
Il n'y a pas de u
Tu as perdu
Le mot etait : python
```

4.6 TP6 : listes

Notions pratiquées : listes (initiation), types mutables, chaînes de caractères.

4.6.1 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/visualize.html
(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):
    elt = maliste[i]
    print("indice", i, " contient: ", elt)
    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

lst_desord = [2, 12, 7, 9, 3, 4]
lst_croiss = [2, 3, 4, 5, 7, 9, 12]


def parcours(lst):
    i = 0
    while i < len(lst):
        print ("Elt d'indice", i, ":", lst[i])
        i = i+1

parcours(lst_desord)

def parcours_envers(lst):
    i = len(lst)-1
    while i >= 0:
        print("Elt d'indice", i, "(envers) :", lst[i])
        i = i-1
```

```
parcours_envers(lst_desord)


def croiss(lst):
    cr = True
    i = 1
    while i < len(lst):
        cr = cr and (lst[i-1] <= lst[i])  i = i+1
    return cr


print(lst_desord, "est croissante:", croiss(ls
print(lst_croiss, "est croissante:", croiss(ls
```

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").

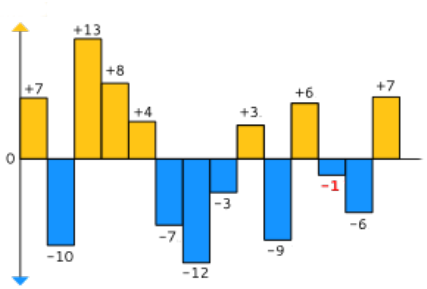
1. Écrivez une fonction `minimum(liste)` qui renvoie l'élément le plus petit de `liste`;
2. Écrivez une fonction `contient(liste, elem)` qui renvoie True si `liste` contient `elem` et False sinon. Pensez à tester les cas limites (p.ex. élément au début ou à la fin).
3. Écrivez une fonction `minimum2(liste)` 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 connait 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.

4.6.2 Le retour des courses (difficile)

Maintenant on veut apparier les n chevaux existants pour former $n/2$ courses intéressantes. *Vous pouvez sauter cet exercice et y revenir à la fin du TP.*

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 simples, 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 [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 ?

4.6.3 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 chargé·e 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):
    elt = maliste[i]
    print("indice", i, " contient: ", elt)
    i = i+1

print("Avant:", maliste)
maliste[3]=maliste[2]
maliste[2].append(0)
print("Apres:", maliste)
```

4.6.4 Listes de mots

Taille des mots Écrire une fonction qui reçoit 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 reçoit un texte et renvoie son mot le plus long.

Mots plus longs que la moyenne Écrire une fonction qui reçoit 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.

Mot comptant le plus de voyelles

- Écrire une fonction qui reçoit un mot (sans espaces) et qui renvoie le nombre de voyelles dans ce mot.
- Utiliser cette fonction pour en écrire une autre qui reçoit un texte et renvoie le mot de ce texte qui compte le plus de voyelles.

Lettre la plus fréquente (plus compliqué) Écrire une fonction qui reçoit un texte et renvoie la lettre qui est présente le plus souvent dans ce texte. Attention à ne parcourir le texte **qu'une seule fois** pour compter les lettres !

4.7 TP7 : algorithmes pour trier des listes

Notions pratiquées : listes, algorithmes de tri, module pyplot

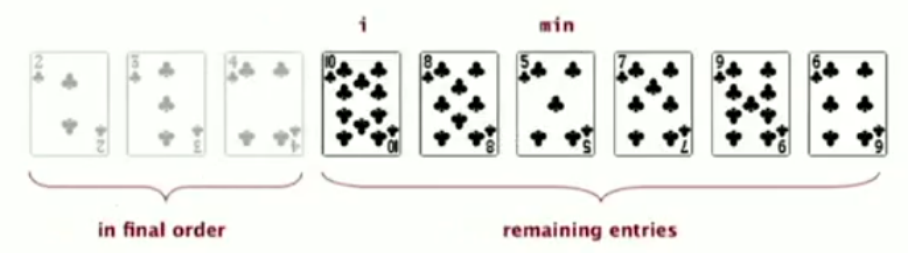
4.7.1 Algorithmes de tri

Voir des animations de différents algorithmes de tri :
https://www.toptal.com/developers/sorting-algorithms

a. Tri insertion. 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 [4].

Source : http://staff.ustc.edu.cn/

b. Tri sélection. 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. Écrivez une fonction `tri_selection` qui trie une liste d'entiers par sélections successives du plus petit élément de la partie non triée de la liste.



Source : http://x-wei.github.io/

c. Tri à bulle Le tri à bulle est une méthode de tri par propagation [5] : on échange progressivement les éléments mal positionnés, ce qui fait remonter les éléments les plus grands vers la fin de la liste.

1. Comprendre l'algorithme du tri à bulle en déroulant un exemple sur papier. En particulier avec le pire cas : une liste en ordre inverse.
2. Écrire une fonction qui reçoit une liste en paramètre et la **modifie** pour la trier avec cet algorithme.
3. Écrire un programme principal qui génère des listes aléatoires d'entiers (module random), et qui les trie avec la fonction écrite ci-dessus.
4. Option : modifier la fonction de tri pour afficher le nombre de comparaisons faites, et le nombre d'échanges faits, pendant le tri de la liste reçue.

4. Voir une démo : https://www.youtube.com/watch?v=3QwCnoa_6FY
5. Voir https://en.wikipedia.org/wiki/Bubble_sort

5. Option : comparer avec les autres algorithmes de tri écrits précédemment.

6. Option : utiliser le module `turtle` pour afficher les éléments de la liste sous forme de rectangles (même largeur, hauteur dépendant de leur valeur), afin d'animer graphiquement le tri.

4.7.2 Course à pied

Cette seconde partie est moins importante. Prenez soin d'avoir terminé et testé vos algorithmes de tri avant de la commencer.

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.

4.7.3 Graphiques scientifiques (bonus, hors programme)

Le module matplotlib et pyplot

Le module matplotlib.pyplot permet de tracer des courbes facilement[6]. 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 f.

- la fonction plt.show() affiche le graphique, il faut l'appeler pour voir la fenêtre graphique

6. Vous pouvez trouver la liste des fonctions de pyplot sur https://matplotlib.org/stable/api/pyplot_summary.ht
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

- 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

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` (Par ex "r–" pour une ligne pointillée rouge).

- Tracés possibles : - ou – ou -. ou :
- Couleurs possibles : b, g, r, c, m, y, k, w
- Marqueurs possibles : o + . x * ∧

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()
```

Polynôme du second degré

Écrire une fonction qui reçoit en paramètres des valeurs a,b,c et affiche la courbe correspondant au polynôme : $a * x^2 + b * x + c$

4.8 TP8 : propagation d'une nouvelle

Notions appliquées : listes, boucles while et for.

4.8.1 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 $[0; N-1]$. 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.

Initialement, seule la personne numéro 0 est au courant de la nouvelle. Puis chaque jour a lieu une rencontre aléatoire entre 2 personnes de numéros X et Y de la population considérée, rencontre représentée par le couple (X, Y).

La nouvelle est propagée différemment d'une personne à l'autre selon leur "statut" respectif relativement à la nouvelle. 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 4.1. 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 4.1 – Évolution 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. Donc la simulation n'évolue plus une fois que plus personne n'a le statut C.

4.8.2 Exercice préliminaire

Travail "manuel" de compréhension sur un exemple :

On considère une population de 5 personnes ($N = 5$), où initialement seule la personne numéro 0 connaît la nouvelle, puis une rencontre et une seule a lieu chaque jour. On considère cette suite de 9 rencontres :

$$\{(2,4),(1,3),(3,0),(2,1),(0,4),(3,4),(2,3),(4,3),(4,0)\}$$

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) :

			statuts après la rencontre				
jour	rencontre	statuts initiaux	pers.0	pers.1	pers.2	pers.3	pers.4
0			C	I	I	I	I
1	2, 4	I, I	...	...	...	...	...
...	...	...	...	...	...	...	...
9	...	...	...	...	...	...	...

Vérifiez qu'à l'issue des 9 rencontres, les statuts des personnes numéro 0 à 4 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).

4.8.3 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 nombres de personnes N saisi au clavier, effectuera une simulation de la propagation d'une nouvelle parmi cette population de personnes (numérotées de 0 à $N-1$), suite à des rencontres **aléatoires**.

Dans l'état initial, la personne numéro 0 connaît la nouvelle (son statut est C) dont on veut simuler la propagation. Les personnes numéros 1 à $N-1$ ignorent cette nouvelle (leur statut est I). Les statuts des personnes (des caractères) seront mémorisés dans une liste.

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. 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 $0 \leq X \leq N-1$, $0 \leq Y \leq N-1$ et $X \neq Y$.

Le traitement d'une rencontre consistera à déterminer si le statut de l'une et/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 4.1 présentée en début d'énoncé (page 178). Dans l'affirmative il faudra mémoriser dans la liste `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. On ne sait donc pas à l'avance combien de jours la propagation va durer.

Exemple d'affichage que devra produire le programme :

```
jour rencontre nature  st.0 st.1 st.2 st.3 st.4
  0                      C    I    I    I    I
  1    2,  3    I, I     -    -    -    -    -
  2    1,  4    I, I     -    -    -    -    -
  3    0,  1    C, I     -    C    -    -    -
  4    2,  1    I, C     -    -    C    -    -
```

```
  5    0,  2    C, C     M    -    M    -    -
  6    2,  0    M, M     -    -    -    -    -
  7    4,  0    I, M     -    -    -    -    -
  8    2,  4    M, I     -    -    -    -    -
  9    3,  1    I, C     -    -    -    C    -
 10    3,  1    C, C     M    M    M    M    I
Nombre d'ignorants :  1
Nombre de muets :  4
```

4.8.4 Exercice 2

On veut maintenant que le programme n'affiche plus **que** le résultat final de la simulation, c'est-à-dire le nombre de jours *NJ* qu'a duré la propagation de la nouvelle, et le nombre d'ignorants *NI* restants à la fin de la simulation. On veut supprimer l'affichage détaillé des rencontres pour pouvoir répéter la simulation un grand nombre de fois et en déduire des statistiques.

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.).

4.8.5 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 pour chaque taille de population.

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 (même taille de population).

4.8.6 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. On rajoutera cette chaîne de caractères dans une nouvelle colonne à la fin de chaque ligne de l'affichage précédent.

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.0 st.1 st.2 st.3 st.4 repartition
  0                    C    I    I    I    I   #....
  1    2,  3    I, I    -    -    -    -    -   #....
```

```
 2    1,  4    I, I    -    -    -    -    -   #....
 3    0,  1    C, I    -    C    -    -    -   ##...
 4    2,  1    I, C    -    -    C    -    -   ###..
 5    0,  2    C, C    M    -    M    -    -   #**..
 6    2,  0    M, M    -    -    -    -    -   #**..
 7    4,  0    I, M    -    -    -    -    -   #**..
 8    2,  4    M, I    -    -    -    -    -   #**..
 9    3,  1    I, C    -    -    -    C    -   ##**.
10    3,  1    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.

4.8.7 Bonus : visualisation avec matplotlib

On veut tracer deux graphiques :

- Pour l'exercice 3, un histogramme[7] 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[8] montrant les proportions NC, NM et NI dans la population totale, et leur évolution au fil des jours de simulation.

7. Voir un exemple à : https://pythonspot.com/en/matplotlib-histogram/
8. Voir un exemple à : https://pythonspot.com/en/matplotlib-pie-chart/

4.9 TP9 : dictionnaires et traduction

Notions pratiquées : dictionnaires.

4.9.1 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("Apres: ", 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("Apres: ", 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-t-ils comme prévu ? Si non, que feriez-vous pour corriger ce problème ?

4.9.2 Exercice préliminaire : manipulation de dictionnaire français-anglais

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.

Créer un fichier `traduc1.py` pour cet exercice.

1. Créer une fonction `ajouteMot` qui reçoit en paramètres un dictionnaire $dfren$, un mot français mfr, et sa traduction en anglais wen, et qui permet d'ajouter ce mot dans le dictionnaire $dfren$. 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 paramètres 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 `afficheDicoLongueur` qui reçoit en paramètres un dictionnaire et un entier (une longueur), et qui n'affiche que les mots de la longueur donnée (longueur du mot français), sous le même format que ci-dessus.

Écrire ensuite un programme principal avec les étapes suivantes :

6. Créer une variable `fr` contenant un dictionnaire associant à quelques mots français leur traduction en anglais.

7. Appeler la fonction `ajouteMot` définie ci-dessus pour ajouter quelques mots dans le dictionnaire, puis l'afficher avec la fonction `afficheDico` pour vérifier.

8. Utiliser la fonction `supprimeMot` définie ci-dessus pour supprimer un mot du dictionnaire, et l'afficher avec `afficheDico` pour vérifier.

4.9.3 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 *(optionnel, non nécessaire pour la suite)*

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 ?

4.9.4 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.

4.9.5 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, elle 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.

4.9.6 Exercice 5 : multi-lingue

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.
6. Bonus : écrire une fonction qui cherche un mot (donné en argument) dans les dictionnaires de toutes les langues, et qui renvoie la langue de ce mot. Si le mot est présent dans plusieurs langues, la fonction renvoie la liste de ces langues.

4.9.7 Bonus en utilisant les fichiers

Se renseigner sur la librairie `pickle` : https://docs.python.org/3/library/pickle.html qui permet de sérialiser des données, c'est-à-dire de les convertir dans un format qui permet de les stocker dans un fichier puis de retrouver leur structure d'origine.

- Enregistrer les dictionnaires de chaque langue dans un fichier dédié.
- Charger le dictionnaire de la langue demandée par l'utilisateur à partir du bon fichier.

4.10 TP10 : jeu du démineur

Notions pratiquées : listes de listes
Dans ce TP on veut coder un jeu de démineur[9] (avec un affichage uniquement textuel).

4.10.1 Structures de données

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]]
```

4.10.2 Compter les mines

Certaines cases contiennent des mines. Chaque case a (au maximum) 8 voisines autour d'elle, et on veut être capable de compter le nombre total de mines sur ces cases voisines.

1. 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
```

9. Si vous ne connaissez pas ce jeu, vous pouvez l'essayer en ligne : http://demineur.hugames.fr/

2. 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 auxilliaire qui calcule et renvoie la liste des cellules voisines d'une cellule donnée.

4.10.3 Affichage

Il y a plusieurs éléments différents à afficher dans le jeu du démineur : soit la solution (avec les positions de toutes les mines), soit le plateau de jeu actuel (avec uniquement les mines connues du joueur). Bien sûr on rappelle qu'une fonction d'affichage ne doit **en aucun cas** modifier la liste affichée.

1. 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) :*

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

2. 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
```

4.10.4 Interaction avec le joueur

1. La fonction de placement des mines ci-dessus boucle jusqu'à avoir placé toutes les mines. On remarque donc qu'elle pourrait entrer dans une boucle infinie si le placement est impossible.

Écrire une fonction `getNbMines` qui reçoit les informations nécessaires pour lire et **filtrer** un nombre de mines à placer (au moins 1, et au maximum une valeur qui assure de ne pas boucler infiniment lors du placement). La fonction renvoie cet entier lu une fois qu'il est correct.

2. 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
```

3. Fin de partie : on suggère d'ajouter une variable qui compte le nombre de cellules encore inconnues au fur et à mesure de la partie, pour savoir quand la terminer.

4. Écrire un programme principal qui appelle les fonctions précédentes :

   - 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 numéro 28
A toi de jouer !
Ligne? 4
Colonne? 6
1?22110
23?2?10
?224320
221??10
?112210
Tu as gagné en 28 coups, bravo !
```

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

4.10.5 Bonus : pose de drapeaux

Dans le jeu de démineur, le joueur peut poser des drapeaux sur certaines cases pour indiquer (et se rappeler) qu'il pense qu'elles sont minées.

- Ajouter une fonction qui pose un drapeau sur une case donnée en paramètre
- Modifier si nécessaire la condition de victoire : le joueur gagne quand toutes les mines sont marquées par des drapeaux
- Modifier le programme principal pour permettre au joueur de choisir à chaque tour s'il veut dévoiler une case ou poser un drapeau.

4.10.6 Bonus : dévoiler récursivement les cases

Dans le vrai jeu de démineur, quand le joueur choisit une case à découvrir qui n'est entourée d'aucune mine, d'autres cases sont découvertes automatiquement.

Écrire une nouvelle fonction qui découvre automatiquement récursivement toutes les cases non minées voisines d'une cellule non minée dévoilée initialement. Concrètement, 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.

4.10.7 Bonus : interface graphique

Utiliser turtle pour réaliser une interface graphique à votre jeu de démineur, avec des cases cliquables (on pourra utiliser le clic gauche pour dévoiler une case, et le clic droit pour poser un drapeau).

4.11 TP11 : révisions

Notions pratiquées : tout le programme

4.11.1 Retour au journal de M. Bizarre

Avec listes

1. Les fonctions écrites précédemment ne transforment qu'un seul mot. Écrire un programme qui les utilise (ainsi que la fonction `split()` qui permet de découper une chaîne) pour transformer tout un texte mot par mot, selon le jour indiqué en paramètre.

2. Bonus, plus difficile : écrire ensuite un programme qui reçoit un texte transformé et qui détermine le jour de la semaine dont on a appliqué la règle. *Il s'agit de trouver pour chaque jour une condition booléenne que vérifie une chaîne ainsi transformée.*

Bonus, avec fichiers

3. Reprendre l'exercice initial, mais le programme doit lire le texte initial dans un fichier "journal.txt", et écrire les résultats de chaque transformation dans un fichier par jour de la semaine (lundi.txt, mardi.txt, etc).

4.11.2 Un menu et des lettres

Écrire les fonctions suivantes, qui reçoivent un caractère (pas forcément alphabétique) en argument et qui :

1. Vérifie s'il est en minuscule ou majuscule

2. Vérifie si c'est une voyelle ou une consonne

3. Calcule sa position dans l'alphabet

Écrire ensuite un programme principal qui affiche le menu en proposant les 4 options codées ci-dessus, et l'option quitter. Tant que l'utilisateur ne choisit pas de quitter, le programme lui demande un caractère, le filtre pour être un seul caractère, affiche le résultat en appelant la bonne fonction, puis ré-affiche le menu.

4.11.3 Dictionnaire

On considère un dictionnaire de randonnées sous la forme suivante :

```
randos = { 1 : {'deniv': 1200, 'distance': 17.8, 'temps': 5.5}, 2: {...}}
```

Les clés sont des numéros identifiants uniques, et les valeurs sont des dictionnaires indiquant pour chaque randonnée son dénivelé en mètres, sa distance en kilomètres, et un temps de parcours en heures.

1. Écrire une fonction qui crée et renvoie un dictionnaire de randonnées en demandant le dénivelé, la distance et le temps à l'utilisateur. L'identifiant unique est généré automatiquement dans l'ordre de saisie.

2. Écrire une fonction qui crée et renvoie un dictionnaire de randonnées aléatoire : pour chaque randonnée, choisir aléatoirement une distance et un dénivelé crédible, puis estimer le temps de parcours (on pourra écrire une fonction auxiliaire), et créer la nouvelle entrée dans le dictionnaire avec l'identifiant suivant.
3. Écrire une fonction qui reçoit en paramètre une chaîne de caractères (valant soit 'distance', soit 'deniv', soit 'temps'), une valeur maximum, et qui renvoie une liste de tous les numéros de randonnées dont la caractéristique nommée est inférieure ou égale à la valeur donnée. Si la clé passée en paramètre n'est pas correcte, renvoyer la liste vide.
4. Modifier la fonction ci-dessus pour ajouter un paramètre optionnel indiquant le sens de comparaison. Cela permettra à l'utilisateur de chercher par exemple toutes les randonnées ayant au maximum 20km, ou à l'inverse toutes les randonnées ayant au minimum 1000m de dénivelé.
5. Écrire une nouvelle fonction de recherche qui renvoie la randonnée **la plus proche** d'une certaine valeur d'un certain critère. Par exemple l'utilisateur peut vouloir la randonnée dont la durée est la plus proche de 3 heures. Il faudra donc comparer les durées de toutes les randonnées du dictionnaire.
6. Écrire une fonction qui reçoit en paramètre le dictionnaire et un numéro de randonnée, et qui affiche son descriptif de manière lisible, sous la forme : "Randonnée numéro XX : XX km, XX m de D+, environ XXhXXmn". *On remarque qu'il faudra convertir le nombre réel d'heures (par exemple 5.5) en nombre d'heures et de minutes (par exemple 5h30mn) : on pourra écrire une fonction auxiliaire pour cela.*
7. Écrire un programme principal qui demande à l'utilisateur son critère de choix, puis sa valeur maximale, puis lui affiche la liste des randonnées correspondant à ce critère (avec la fonction d'affichage précédente), une par ligne.
8. Écrire une fonction qui permet de modifier le dictionnaire pour y ajouter un champ 'chrono' une fois une randonnée réalisée par l'utilisateur. *Toutes les entrées n'auront donc pas forcément de champ chrono, ce qui permettra de reconnaître les randonnées déjà réalisées.*
9. Modifier la fonction de recherche précédente, pour y ajouter un paramètre optionnel `new`, valant par défaut `False`. Si ce paramètre vaut `True` alors la fonction de recherche ne doit renvoyer dans la liste que des randonnées pas encore réalisées par l'utilisateur (champ 'chrono' inexistant).
10. Modifier la fonction de sélection précédente pour qu'elle affiche les randonnées répondant au critère, classées dans l'ordre des valeurs de ce critère. Par exemple si l'utilisateur veut les randonnées de moins de 20km de distance, alors il faut classer les randonnées affichées dans l'ordre croissant de distance.
11. Écrire une fonction qui permet de supprimer le champ 'chrono' d'une randonnée donnée par son numéro.

4.11.4 Plus d'exercices

- En ligne : PLM, Caseine...
- Ou à partir des annales d'examen

Annexes

RAPPELS — CONSIGNES ET BONNES PRATIQUES

Le but de cette UE **n'est pas** :

- connaître les raccourcis de programation spécifiques à Python ;
- connaître toutes les fonctions et / ou modules Python ;
- programmer en orienté-objet ;
- écrire le code le plus court possible ;
- écrire un code « qui marche ».

Le but de cette UE **est** :

- apprendre à écrire des algorithmes pour résoudre des problèmes ;
- connaître les algorithmes classiques (tri, comptage, recherche…) ;
- savoir implémenter ces algorithmes en respectant la syntaxe d'un langage de programmation, ici Python ;
- apprendre à écrire un code propre, modulaire, commenté, non redondant, respectant les contraintes de l'énoncé ;
- acquérir les premières notions de complexité des programmes.

4.12 Comment sont notés vos programmes

Un programme « qui marche » n'aura pas la note maximale si :

- il est illisible, raturé ;
- il est incompréhensible, non commenté ;
- il contourne les consignes de sorte qu'il ne permet pas au correcteur d'évaluer l'élément souhaité ;
- il utilise des raccourcis de programmation spécifiques à Python ;
- il utilise des éléments (fonctions, modules, constructions, …) non vus ou hors programme ou non explicitement autorisés ;
- il utilise des éléments explicitement interdits (par exemple `break` ou `continue`) ;
- du code y est dupliqué ou redondant ;
- il est inutilement compliqué ;

- il ne respecte pas la consigne ou les contraintes données (exemple d'exécutions, format d'affichage, texte dans les entrées-sorties, ...).

4.13 Bonnes pratiques

Il est donc vivement conseillé de :

- écrire un code propre ;
- respecter les niveaux d'indentation ;
- commenter les programmes, en particulier les `else` et les sorties de boucle ;
- donner des noms parlants aux variables, en évitant les mots-clés réservés de Python ;
- toujours afficher un message pour accompagner les `input` / `print`.

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

```
== égalité              != différence
<  inférieur,           <= inférieur ou égal
>  supérieur,           >= supérieur ou égal
```

Instructions conditionnelles

```
if condition :
    instructions

if condition :
    instructions
else :
    instructions

if condition1 :
    instructions
elif condition2 :
    instructions
else :
    instructions
```

Opérateurs arithmétiques

```
+ : addition,          - : soustraction
* : multiplication,    ** : puissance,
/ : division,          // : quotient div entière,
% : 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 élements 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