Pascal ORTIZ

Découverte de la programmation

Éléments de cours, 597 exercices

Version du 19 novembre 2023

Table des matières

I. Interface, variables, opération **9**

Le processus de programmation en Python . . . 9
Pouvoir utiliser un interpréteur . . . 12
Environnement de programmation . . . 13
Installer et lancer Jupyter sous Windows 10 ou 11 . . . 16
Feuilles Jupyter Notebook : usage . . . 17
Afficher . . . 23
Afficher un message . . . 23
`print` et saut de ligne par défaut . . . 23
Variables . . . 24
Notion d'expression . . . 25
Syntaxe basique des noms de variables . . . 25
Nom ou variable non reconnus . . . 26
Réaffectation de variable . . . 27
Notions sur les constantes . . . 27
Constantes entières . . . 28
Grands entiers . . . 28
Constantes flottantes . . . 29
Types de base . . . 30
Même valeur et types différents . . . 30
Variables et typage . . . 31
Les fonctions built-in . . . 31
Indentation accidentelle . . . 32
Placer des commentaires . . . 32
Opérations sur les nombres . . . 33
Division flottante . . . 34
Utiliser des variables . . . 34
Exercice type – Volume de la sphère . . . 35
Opérations mixtes entre flottants et entiers . . . 36
Opérateurs « égale à » et « différent de » . . . 36
Comparaison de nombres . . . 37
Comparaison de flottants . . . 37
Confusion entre égalité et affectation . . . 38
Multiples, division entière . . . 39
Quotient et reste de division entière . . . 40
Division entière et quotient exact . . . 42
Traduire « être multiple » en Python . . . 43
Affichage multiple . . . 44
Affichage amélioré . . . 44

Afficher un message complexe . . . 45
Affichage et forcer un passage à la ligne . . . 45
Affichage implicite dans Jupyter Notebook . . . 47
La bibliothèque standard . . . 48
Importer une fonctionnalité standard . . . 49
Le Hasard en Python . . . 49
Fonction `randrange` . . . 50
Fonction `randint` . . . 50
Exercices . . . 52

II. Dessiner avec Matplotlib **64**
1 Cours . . . 64
Utilisation de Matplotlib dans ce cours . . . 64
Installation de Matplotlib . . . 66
Utilisation de Matplotlib sur Google Colab . . . 67
Instruction de visualisation . . . 69
Système de repérage . . . 70
Tracé d'un segment, d'un point . . . 72
Tracé d'un disque . . . 75
Tracé d'un rectangle . . . 77
Tracé d'un secteur circulaire . . . 79
Tracer un polygone . . . 80
Tracer un arc de cercle ou d'ellipse . . . 81
Tracer une ellipse . . . 82
Texte basique . . . 83
Utiliser des fonctions comme raccourci de code . . . 84
Modifier la taille de la figure courante . . . 85
Les couleurs sous Matplotlib . . . 86
Placer une couleur de fond . . . 89
Sauvegarder le dessin dans un fichier png . . . 90
Fonctions prêtes à l'emploi . . . 91
Fonction `addsubplot` et version de Matplotlib . . . 92
Exercices . . . 94

III. Conditions **103**
1 Booléens . . . 103
Opérateurs logiques ET, OU et NON . . . 103
Les booléens en Python . . . 105
Opérateurs `and`, `or` et `not` . . . 106
Expressions booléennes . . . 107
Exercice type – Année bissextile . . . 108
Exercice type – 14 juillet . . . 111
Exercice type – Mois de 31 jours . . . 112
Exercices . . . 114
2 Instructions conditionnelles . . . 118
Instruction `if` . . . 118
Indentation d'une instruction composée . . . 121
Instruction if/else . . . 123
`if/else` pour un booléen : complication inutile . . . 124

Indentation d'une instruction composée avec clause . . . 125
Instructions `if` imbriquées . . . 126
Instruction `if/elif` . . . 127
Suite de `if` vs suite de `if/elif` . . . 128
Les conditions `if/elif` . . . 129
Instruction `if/elif/else` . . . 130
Exercice type – Le plus grand, le plus petit . . . 131
Exercice type – Régler un montant avec des coupures de même valeur 132
Exercices . . . 134

IV. Boucles for, listes, boucles while 147
1 Boucles for . . . 147
Boucle `for` : introduction . . . 147
Répéter une action n fois avec une boucle `for` . . . 149
Affichage sur une même ligne et boucle `for` . . . 150
Itérer sur des entiers consécutifs . . . 151
Calcul d'une somme . . . 151
Exercice type – De 42 en 42 . . . 152
Exercice type – Comptage de 5 en 5 . . . 153
Exercice type – Somme alternée plus/moins . . . 153
Exercice type – Afficher des nombres par paires . . . 154
Exercices . . . 157
2 Listes et matrices . . . 168
Notion de liste . . . 168
Indexation des éléments d'une liste . . . 169
Nombre d'éléments d'une liste . . . 170
Opérations sur les éléments d'une liste . . . 170
Appartenance à une liste . . . 171
Non-appartenance à une liste . . . 171
Modifier les éléments d'une liste . . . 171
Alias de liste . . . 172
Liste de listes . . . 173
Notion de tableau 2D . . . 174
Listes en compréhension . . . 175
Liste en compréhension et boucle `for` . . . 176
Création d'un tableau 2D initialisé . . . 176
Listes en compréhension et la clause `if` . . . 177
Listes en compréhension imbriquées . . . 178
Notion de matrice . . . 178
Les matrices sous le module Numpy . . . 179
Exercice type – Éléments central ou centraux . . . 185
Exercice type – Générer la matrice identité . . . 186
Exercices . . . 188
3 Parcourir, répéter . . . 200
`append` : adjoindre un élément à une liste . . . 200
Parcours complet d'une liste sans indice . . . 201
Parcours complet d'une liste par indices . . . 202
Boucle for sans indice et calcul de somme . . . 203
Boucle for : décompression de la variable de contrôle . . . 203

Exercice type – Somme des positifs . . . 204
Exercice type – Somme sauf des extrémités . . . 204
Boucle for, filtrage et comptage . . . 205
Boucle for et calcul de maximum . . . 206
Parcours de liste : indices voisins de l'indice courant . . . 207
La technique du drapeau dans une boucle for . . . 208
Exercice type – Alternance de parité . . . 209
Liste construite depuis la liste vide . . . 211
Construction de petites listes . . . 212
Exercice type – Liste des N premiers multiples de d . . . 212
Exercices . . . 214
4 Boucles imbriquées . . . 238
Boucles for imbriquées . . . 238
Exercice type – Liste d'entiers sans éléments consécutifs . . . 240
Exercice type – Grille : remplissage par lignes . . . 241
Effectuer plusieurs tests à l'aide d'une boucle for . . . 242
Exercices . . . 243
5 Boucle while . . . 260
Boucle while . . . 260
Boucle for vs boucle while . . . 263
Boucle while et parcours de liste . . . 264
Boucle `for` vs boucle `while` : tableau récapitulatif . . . 265
Exercice type – La petite exponentielle contre le grand produit . . . 265
Exercice type – Premier entier pair . . . 266
Exercices . . . 269

V. Fonctions **279**
1 Syntaxe, vocabulaire . . . 279
Les fonctions en Python . . . 279
Une fonction typique . . . 280
return et fin d'exécution . . . 283
Exercice type – Fonction aire de disque . . . 284
Exercice type – Implémenter la fonction *signe* . . . 284
Fonction renvoyant plusieurs valeurs . . . 286
Fonction sans paramètre . . . 286
Absence de `return` . . . 287
Retour d'une fonction sans instruction `return` . . . 288
Procédure ou pas ? . . . 289
Notion d'effet de bord . . . 290
Instructions `return` multiples . . . 291
Exercices . . . 293
2 Construire, utiliser des fonctions . . . 311
Des instructions à la construction d'une fonction . . . 311
Enchaîner des fonctions . . . 313
De l'usage des fonctions . . . 316
Exercice type – Périodes de l'année . . . 320
Modification par une fonction d'un de ses arguments . . . 322
Boucle for et return . . . 324
Exercice type – Grille en équerres . . . 325

Variables locales . . . 326
Variables globales . . . 328
Constantes placées en variables globales . . . 328
Fonction non appelée . . . 329
Le passage des arguments par affectation . . . 329
Exercices . . . 331

VI. Chaînes de caractères **425**
Les chaînes : premier contact . . . 425
La notion de caractère . . . 426
La notion de chaîne littérale . . . 426
Opérations sur des chaînes . . . 427
Chaîne vide . . . 428
Le caractère *espace* . . . 428
Le caractère *saut de ligne* . . . 429
Concaténation et répétition de chaînes . . . 430
Chaînes égales . . . 432
Recherche de sous-chaînes avec `in` . . . 432
Accès aux caractères d'une chaîne . . . 433
Dépassement d'indice dans une chaîne . . . 434
Boucle for : parcours de chaînes . . . 434
Créer une chaîne à partir de la chaîne vide . . . 435
Modifier une chaîne . . . 435
Exercices . . . 437

Table des exercices

Cliquez sur l'exercice pour y accéder.

Interface, variables, opération	Plus petit multiple supérieur ou égal	Cadrer carré
Volume de la sphère	Nombre d'entiers impairs entre deux entiers donnés (formule)	Logos succès-échec
Demi-carré en mode texte	Nombre de carrés dans une figure (formule)	Danger nucléaire
Echanger deux variables	Nombre de multiples de d entre deux valeurs données	Logo de Tripadvisor
La finance aime Python	Variable définie suivant la parité	Conditions
Factorielle de 10	Répétitions sans boucle	Booléens
Volume du ballon de rugby	Diviser pour carreler	Année bissextile
Formule de Héron	Formule pour arrondir une note au quart de point	14 juillet
Volume d'un tétraèdre	Plus grand entier pair tel que ...	Mois de 31 jours
Volume d'un sac-poubelle	Deux mois plus tôt	Egaux ou opposés
Fromage sur la pizza	Position après des allers et retours	Nombres de même signe
Valeur approchée de $e = 2{,}718...$	Affichages empilés, côte-à-côte	Entiers qui se suivent
Calculs	Reproduire un affichage	Trois entiers consécutifs
Périmètre d'une ellipse	Copier-coller pour Fibonacci	Négatif, positif et nul
Aire d'un segment circulaire	Factorielles	Test de la non-égalité de cinq entiers
Combien d'oranges pour un jus ?	Dessiner avec Matplotlib	Conditions sur un entier
Vulnerant omnes, ultimat necat	Cours	Au moins un pair et au moins un supérieur à 100
Date fournie par le GPS	Quelques marches	Heures d'ouverture
Priorités des opérateurs	Découper un carré	Vacances
Puissances sans parenthèses	Cube	Jours de repos
Formule	Dessiner un signe *plus*	Tous ou uniquement un positifs
Diviseur, multiple	Enveloppes	Exactement un positif
Multiples de 421	Rose des vents	Entier ayant n chiffres
Nombre de tours	Centre de gravité	Entiers dans un rapport de 2 ou 3
Plus petit multiple strictement supérieur	Dessiner un huit	Triangle valide ou pas

Intervalles disjoints ou pas	Nombre d'entiers impairs entre deux entiers donnés	Afficher par paires verticales
Instructions conditionnelles	Raccourcir une suite de conditions	Afficher par triplets verticaux
Le plus grand, le plus petit	Ramener `if/else` à un booléen	Suite de disques
Régler un montant avec des coupures de même valeur	Boucles for, listes, boucles while	Carrés côte-à-côte
Ordonner trois entiers	Boucles for	Dents de scie
Nombre d'éléments distincts	De 42 en 42	Cible
Afficher l'écart entre deux nombres	Comptage de 5 en 5	Dessiner un U
Nombre intermédiaire	Somme alternée plus/moins	Disques en carrés
Date valide (`if\elif`)	Afficher des nombres par paires	Collier de perles
Placer entre deux nombres	Répéter 2030	Carrés concentriques
ensureRange	Afficher des entiers consécutifs	Dessiner un faisceau de demi-droites
Équation du second degré	Afficher deux listes l'une après l'autre	Clôturer un champ carré
Jeu à deux nombres	Itérer 2x+1	Grille carrée
Combien de pairs et d'impairs ?	Somme des n premiers entiers	Grille de Sudoku
Répartion équitable de pièces de monnaie	Somme de n entiers consécutifs	Pyramide maya
Articles dans un magasin	Somme d'entiers impairs consécutifs	Listes et matrices
Catégories en sports	Sommes de puissances consécutives	Éléments central ou centraux
Quelle saison sommes-nous ?	Sommes de puissances de 2 ou de 3	Générer la matrice identité
Catégoriser un triangle	Somme alternée plus/moins en décroissant	Liste de trois éléments
Zones de couleurs	Calcul d'une somme (sigma)	Liste : premier et dernier
Opérateur xor	Plus grand diviseur impair	Liste : k-ème élément
Distance d'un point à un rectangle	Restes de deux entiers	Liste des solutions de l'équation du second degré
Transport en bus	Compter de 0,5 en 0,5	Changer certains éléments d'une liste
Facteur minimal pour dépasser un entier donné	Compter par tiers	Echanger les extrémités d'une liste
Reprographie au format grand livret	Restes de deux entiers	Kubernetes -> K8s
Durée de sommeil	Entier pyramidal	Tester les mois de 31 jours avec une liste
Pierre, papier, ciseaux	Entiers oblongs	Test de présence
Excédent payé	Produit d'entiers	Mois de 31 jours et liste
Arrondir l'heure	Allers et retours	Afficher toutes les dates de l'année
Carré parfait le plus proche	Pomodoro	Lister les non multiples

Liste d'entiers consécutifs décroissants et alternant de signe	Parcours *indice -> élément*	Élément de carré minimal
Liste des entiers impairs suivi des pairs	Valeur et indice de même parité	Ecart maximal de températures
Liste des carrés entre deux nombres	Tester avec une boucle `for`	Alternativement début et fin
Séparer en indices pairs et indices impairs	Tests aléatoires	Affichage en tas binaire (une seule boucle `for`)
Tables de multiplication via des listes en compréhension	Non multiples	Somme des valeurs ayant même parité que l'indice
Décompression de liste via des listes en compréhension	Liste cardinale	Le double du précédent
Liste des diviseurs via des listes en compréhension	Compter les multiples de 10	Nombre de carrés dans une figure (boucle)
Voisins d'un coefficient d'une matrice	Décompte pair/impair	Entiers consécutifs en décroissant
Extraction d'une colonne d'une matrice	De deux en deux dans la deuxième moitié	Dix de plus ou de moins
Trace d'une matrice	Impairs puis pairs	Liste constante
Transposée d'une matrice	Echanger côte à côte	Listes « opposées »
Etre une matrice diagonale	Modifier une liste en multipliant par 10	Suite croissante d'entiers consécutifs
Etre une matrice triangulaire supérieure	Inverser une liste	Suite en zig-zag
Etre une matrice scalaire	Décalages de liste	Liste creuse
Générer une matrice diagonale	Rotation de liste	Liste qui patine
Anti-transposée d'une matrice	Liste : négatif -> opposé	Regroupement par signe
Bords d'une matrice	Inverser indices et valeurs dans une liste	Égaux et côte-à-côte
Implémenter le produit de deux matrices	Un, deux ou cinq	Cumul de dénivelé
Générer un motif de matrice : équerres	Créneaux	Home Sweet Home
Générer une matrice tridiagonale	Disques tangents de couleurs alternées	Concaténer des entiers consécutifs
Insérer une ligne dans une matrice	Carrés bords à bords	Médiane d'une liste de nombres
Insérer une colonne dans une matrice	Somme des multiples de 10	Compter les hausses et les baisses
Inverser l'ordre des colonnes d'une matrice	Somme suivant les indices impairs	Valeurs paires d'une liste
Transformation de Sylvester	Somme de n entiers consécutifs à partir de d	Sous-liste strictement croissante
Remplissage d'une grille en hélice	Calcul depuis la base b	Trier une liste par signe
Parcourir, répéter	Variations des éléments d'une liste de nombres	Indices pairs et indices impairs
Somme des positifs	Calcul du minimum	Subdivision régulière
Somme sauf des extrémités	Indice du minimum	Fonction `linspace`
Alternance de parité	Plus petit entier non nul (hypothèse simplificatrice)	Concaténer deux listes
Liste des N premiers multiples de d	Minimum des entiers d'indices pairs	Somme de deux listes

Intersection de deux segments	Problème de programmation linéaire	Tests aléatoires
Répartition dans l'ordre	Problème de dénombrement	Boucle while
Minimum des entiers pairs (boucle `for`)	Nombres consécutifs dont la somme vaut 54	La petite exponentielle contre le grand produit
Minimum présent une seule fois	Nombres de Harshad	Premier entier pair
Entiers consécutifs mélangés	Somme de sommes	Somme harmonique dépassant 18
Sommer des éléments successifs d'une liste	Tri-casier	ppcm
Élément de rang n de la suite des entiers k répétés k fois	Inversions dans une liste	Restes chinois
Formule pour la suite des entiers k répétés k fois	Ecart minimal d'indices (boucles imbriquées)	Diviser par deux ou retirer 1
Plateau le plus long	Différences successives	Soustraire jusqu'à équilibre
Dupliquer une liste sur une certaine longueur	Damier de nombres	Divisions successives par deux
Pics d'une liste	Pyramide croissante de nombres	Tirage du loto
Compression de liste	Motif carré formé des chiffres 4 ou 2	Calculer le nombre de chiffres d'un entier
Nombre de McNugget	Motif carré et sa diagonale	Nombre de chiffres d'un entier
Dessiner une spirale	Triangle décroissant de nombres	Nombre de zéros qui terminent un entier
Tables de Plouffe	Triangles de 0 et de 1	Inverser l'écriture d'un nombre
Boucles imbriquées	Triangle à lignes alternés	Etre un entier léger
Liste d'entiers sans éléments consécutifs	Triangle : lignes consécutives décroissantes	Placement dans une suite croissante
Grille : remplissage par lignes	Grille : remplissage par colonnes	Insertion dans une suite croissante
Intersection de listes	Sapin	Pas d'impair
Répéter l'affichage du contenu d'une liste	Triangle et son bord	Liste d'entiers en miroir
Remplir une liste avec des 0, des 1 et des 2	Carrés concentriques alternés	Listes « opposées » (boucle `while`)
Somme qui vaut 42	Liste des entiers k répétés k fois	Liste en miroir opposé
Entiers opposés dans une liste	Damier de disques	Alternance de parité, version `while`
Somme nulle	Dessin d'un motif en forme de croix	Suite croissante d'entiers consécutifs (avec `while`)
Etude d'une suite définie par une somme	Dessiner des carrés concentriques alternés	Que des entiers consécutifs
Afficher les effectifs d'une liste d'entiers	Carrés concentriques en dégradé de gris	Que des 81 puis que des 12
Pièces de monnaie	Triangle de Floyd	Déclin (boucle `while`)
Montant irréalisable	Affichage en tas binaire (boucles `for` imbriquées)	Négatif, nul, positif
Multiple de 42 qui soit somme de deux carrés	Générer toutes les permutations	Grenouille dans un enclos
Nombre de rectangles d'aire plafonnée	Tester avec une boucle `for`	Plateau le plus long

Zéros en fin de liste	Nombre de chiffres	Afficher un mouvement
Plus long préfixe commun	Nombre de chiffres de $n!$	Doubler
Eléments absents d'une liste croissante d'entiers	Moyenne harmonique	Moyenne olympique
Fonctions	Somme supérieure à ...	Un élément sur trois
Syntaxe, vocabulaire	Somme multiple de ...	Sommer deux par deux
Fonction aire de disque	Compter les entiers négatifs	Construire, utiliser des fonctions
Implémenter la fonction *signe*	Aucun multiple de 10	Périodes de l'année
Fonction de comparaison de signe	Valeurs non nulles d'une liste	Grille en équerres
Fonctions Celsius <-> Fahrenheit	Positivons ...	Afficher une somme
Conversions de devises	Remplacer les nombres négatifs	Indice de masse corporelle
Nombre d'entiers positifs	Liste des multiples de 2, 3 ou 5	Histoire de zéros
Fonction pour tester une année bissextile	Nombre d'éléments entre deux valeurs	Somme des chiffres
Date valide	Liste d'entiers aléatoires	Entiers narcissiques
Calculer son âge	Sur certaines listes de trois entiers	Plus loin, plus proche
Implémenter la fonction `max`	Extraction de valeurs aux indices impairs	Volume d'un cylindre
Implémenter la fonction *valeur absolue*	Derniers éléments d'une liste	Calcul d'une durée
Fonction copysign	Derniers termes d'une liste	Arrondi au quart de point
Distance entre deux nombres	Indice de la dernière occurrence	Formule de Keith et Craver
Fonction ensureRange	Valeur absolue des éléments d'une liste	Fractions
Le plus proche et entre -1 et 1	Carrés d'une liste	Le moins présent
Le plus proche parmi deux nombres	Minimum et maximum simultanés	Périmètre d'un polygone
Inclusion d'intervalles	Extraire un point dans une liste	Disques disjoints colorés
Segment découpé en parties de même longueur	Réfléchir une liste	Dessiner les tétraminos
Équation du second degré (version fonction)	Nombres entre 1 et 99	Date du jour de Pâques
Fonction de calcul du volume du ballon de rugby	Moitiés	Liste croissante
Volume d'un tétraèdre (version fonctions)	Eléments distincts	Toujours la moyenne
Triangle direct	Liste ordonnée à deux éléments	Pairs d'abord, impairs ensuite
Écart-type d'une liste	Éléments non nuls	Entiers consécutifs croissants ou pas
Somme des inverses des entiers	Objets qui s'accordent	Liste monotone
Nombre d'arrangements	Nombre de bus et division	Suppression alternée d'entiers dans une liste

Suite unimodale	Plus grand diviseur impair : méthode rapide	Regrouper par paires
Suite super croissante	Nombre de zéros qui terminent un entier (version fonction)	Chaînage par paires
Deuxième plus petit élément d'une liste	Décomposer en groupes de 2 ou 3	Séparer suivant la parité des valeurs
Deux plus grands éléments d'une liste	Entier différence de deux carrés	Séparer des entiers selon la parité des indices
Deux plus grands éléments distincts d'une liste	Calcul itératif du ppcm	Pivot
Les plus petits devant	ppcm de plusieurs entiers	Décompression de liste
Nombre d'éléments distincts d'une suite croissante	Somme des termes d'une récurrence double	Séparer les abscisses et les ordonnées
Intersection de deux listes strictement croissantes	Nombres de Perrin	Rectangle à partir d'une de ses diagonales
Liste d'entiers symétrique par rapport à zéro	Sommes d'entiers consécutifs	Reconstruire les sommets d'un rectangle
Eléments majoritaires d'une liste	Somme des carrés	Jeu de cartes
Supprimer les doublons	Nature d'un quadrilatère	Evénement périodique
Ils sont uniques !	Triangle magique	Nom de qui a la meilleure note
Doublons d'une liste croissante	Divisibilité par 7 : le critère de Chika	Moyenne pondérée
Lancers de dé	Puissance de 2 commençant par ...	Séparer des entiers suivant leur nombre de chiffres
Toujours plus	Nombres hautement composés	Liste des diviseurs
Balles dans des boîtes	Billes en rectangle	Statistiques sur des contrôles
FACE quatre fois de suite	Elément le plus proche des éléments d'une liste	Relevé de températures
Suite alternée pile ou face	Elément le plus proche d'une subdivision régulière	Transposer une liste de listes
Suite aléatoire croissante	Comparer des moyennes	Scinder une liste en deux
Jeu de dé : premier coup gagnant	Fonction non injective	Diagonale descendante d'une grille
Tirage du loto (version fonction)	Persistance multiplicative	Matrice de covariance
Jeu du 421	Nombres de Catalan	Triangle de Floyd vertical avec liste de listes
Paradoxe des anniversaires	Nombres de Bernoulli	Transformation du boustrophédon
Mélanger une liste	Fonction binomiale	Rotation horaire d'une matrice carrée
Gagner au jeu de craps	Valeur approchée de π	Remplir une matrice en diagonale
Le dilemme de Monty Hall	Quotient illimité	Somme des carrés des éléments d'une matrice
Mélange de Fisher-Yates	Nombre de Lychrel	Matrice à symétrie verticale
L'algorithme P	Racine carrée d'un nombre complexe	Border une matrice par des zéros
Somme, maximum	Tables de multiplication	Retirer le bord d'une matrice
Diviseurs, nombre premier	Regrouper par signe	Matrice magique

Demi-carré en mode texte de hauteur variable
Générer un motif en forme de croix
Motif triangulaire de nombres
Triangle d'entiers consécutifs
Suite de Prouhet
Pyramide de disques
Carrés emboîtés
Paintball
Déplacement vertical
Signes *plus* emboîtés
Triangles emboîtés
Dessiner une frise
Equerres de disques
Logo Google Drive
Construire un histogramme
Remplissage de bassins
Fizz-buzz en VF
Inspirez, expirez !
Suite de Syracuse
Fibonacci
Multiplication du paysan russe (boucle `while`)
Suite d'entiers
Pyramide de cubes
Sous-listes d'une liste
Plus petit entier non nul (découpage en fonctions)
Minimum des entiers pairs (découpage en fonctions et boucle `for`)
Minimum des entiers pairs (découpage en fonctions et boucle `while`)
Triangle de Floyd vertical
Tranches de termes d'une liste
Zéros qui séparent
Tranches de même signe
Nombre de changements de signe
Répétitions successives
Au plus deux zéros consécutifs
Liste sans doublons (version efficace)
Réduction parallèle de somme
Marques sur un axe
Plus petit nombre unique
Zéros isolés
Indices des minimums des entiers pairs
Plateau le plus long
Somme de tous les autres
Arrondi au demi point
Ecart minimal d'indices (tableau auxiliaire)
La suite de Van Eck
Suite de Mazet-Saias
Numéro de jour de l'année
Le problème des déménageurs
Somme de deux éléments d'une liste croissante
Remplir une grille en zig-zag
Déplacement en zig-zag
Détection de cycle
Le tri par sélection
Vendredi 13
IA pour un Mastermind
Alignement au Puissance 4
Formule de Luhn
Chaînes de caractères
Doubler les lettres
Distance de Hamming
Doubler les voyelles sauf certaines
Changer OU en O et O en OU
Palindrome
Chaînes formées seulement de lettres parmi a ou b
Mots les plus longs
Conjuguer un verbe du premier groupe
Bijou, caillou, etc
Dernières lettres d'un mot
Mots en lignes, en colonnes
Triangle en mode texte
Découper en mots
Immatriculation valide
Afficher un numéronyme
Mot de passe valide
Afficher les occurrences d'un sous-mot
Métathèses
Lettres découvertes
Séparer par des majuscules
Les deux premiers mots
Supprimer les espaces autour d'une chaîne
Nettoyer un message
Séparer par blocs de 3 chiffres
Supprimer la virgule et les zéros inutiles
Chaîne représentant un entier ou pas
Nom valide ou pas
Mot dans une phrase à une position donnée
Découper un texte en lignes
Répéter un motif et tronquer

Chapitre I

Interface, variables, opération

Le processus de programmation en Python

Le cycle de programmation

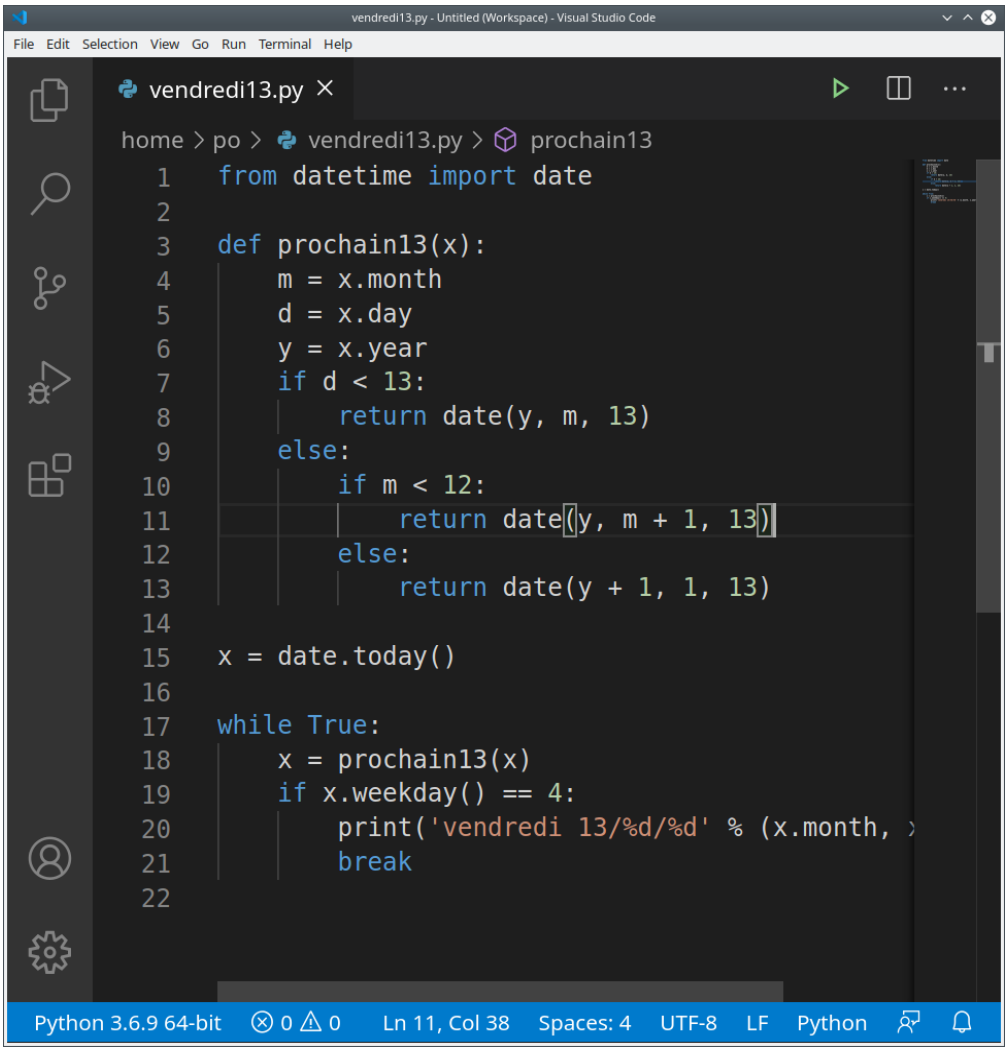
Soit à produire un programme écrit en langage Python et qui résout un problème, par exemple :

PROBLÈME : *calculer et afficher la date du prochain vendredi 13.*

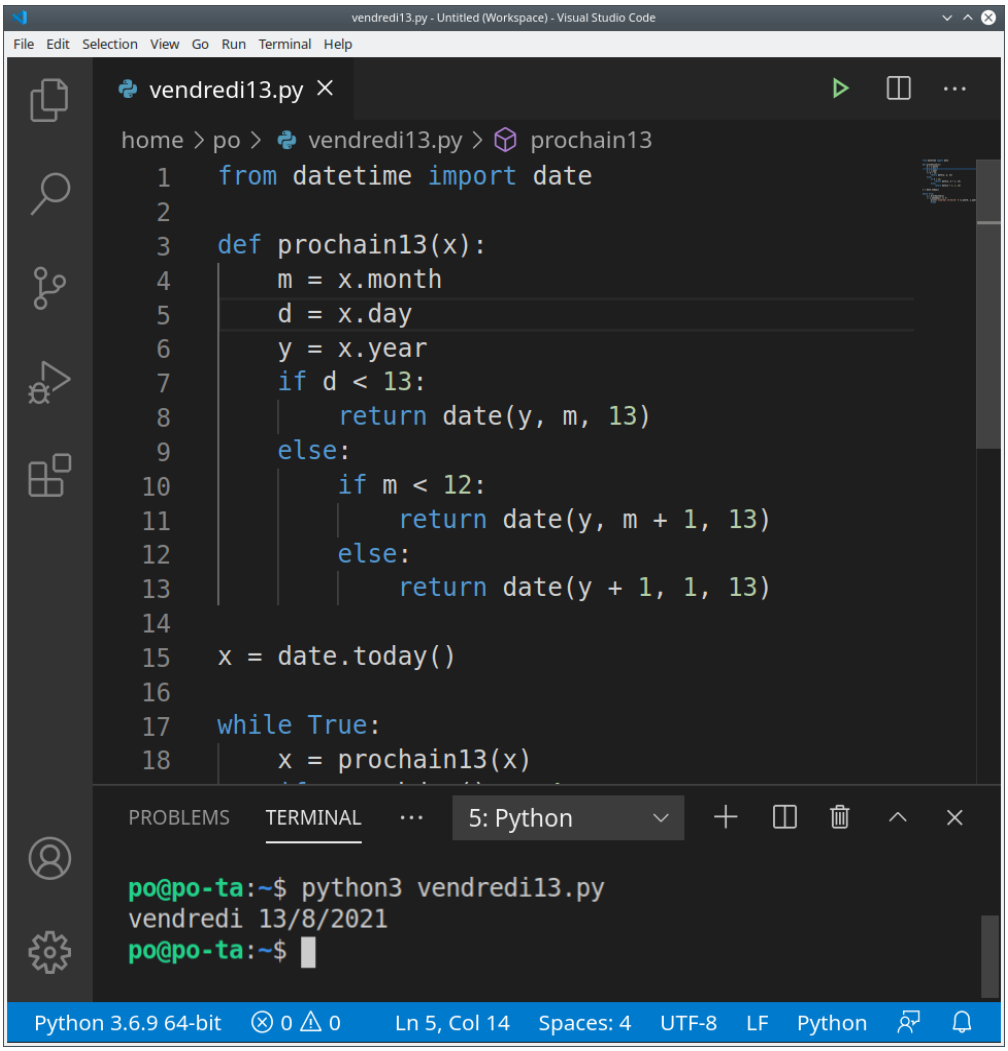
On considère un programmeur qui a les compétences suffisantes pour écrire un tel programme. Le programmeur ouvre un environnement de programmation. Voici les étapes par lesquelles il va passer :

— Étape 1 : *édition du code-source*
  Le programmeur ouvre un logiciel plus ou moins sophistiqué permettant de saisir (au clavier) le code-source dans le langage de programmation Python et censé résoudre le problème posé et il écrit ce code-source, cf. la première image ci-dessous.

— Étape 2 : *exécution du code-source*
  Le programmeur exécute le code qu'il a écrit à l'étape 1 pour voir si ce code réalise bien ce pour quoi il a été écrit, ici trouver le prochain vendredi 13, cf. la deuxième image ci-dessous. Si ce n'est pas le cas, il revient à l'étape 1 et modifie le code-source.



Étape 1 : code Python saisi dans VSCode



Étape 2 : exécution du code précédent. On lit la réponse dans la console en bas

Même pour un nouveau-venu à la programmation, le schéma en deux étapes ci-dessus reste valide.

L'étape 1 est en général la plus longue et la durée et la qualité de la réalisation de cette étape vont dépendre des connaissances plus ou moins approfondies du programmeur.

Programme du vendredi 13

Concernant le problème du prochain vendredi 13, le programmeur pourrait écrire dans un code-source tel que celui-ci :

vendredi13.py

```
1 from datetime import date
2
3 def prochain13(x):
4     m = x.month
5     d = x.day
6     y = x.year
7     if d < 13:
8         return date(y, m, 13)
```

```
    else:
        if m < 12:
            return date(y, m + 1, 13)
        else:
            return date(y + 1, 1, 13)

x = date.today()

while True:
    x = prochain13(x)
    if x.weekday() == 4:
        print(f'vendredi 13/{x.month}/{x.year}')
        break
```

Si vous exécutez le code vendredi13.py en 2021, selon la période de l'année, il affichera

```
vendredi 13/8/2021
```

ou sinon

```
vendredi 13/5/2022
```

Le cycle édition-exécution

Selon que le résultat de l'étape 2 est satisfaisant ou pas, le programmeur retourne à l'étape 1 pour modifier le code-source et ainsi de suite : c'est le cycle dit *édition-exécution.*

Suivant les programmes, ce cycle peut s'étaler entre quelques secondes (pour les petits programmes) et plusieurs mois pour de programmes complexes et volumineux, faits en équipe.

Pouvoir utiliser un interpréteur

Pour coder en Python, il **faut** disposer d'un logiciel appelé un *interpréteur Python* qui puisse exécuter du code écrit en Python. Pour disposer d'un interpréteur, il y a deux situations :

— l'interpréteur est distant, accessible via Internet depuis un site web
— l'interpréteur est local, installé sur l'ordinateur avec lequel vous travaillez

Dans le cadre d'un auto-apprentissage de Python, la solution de l'interpréteur local suppose, la plupart du temps, que c'est **vous** qui vous chargez de l'installation de Python et aussi ... des difficultés qu'elle peut occasionner.

L'interpréteur distant a tous les avantages (et les inconvénients) d'une utilisation via une connexion Internet. Voici quelques avantages :

— vous n'avez rien à installer sur votre machine,
— l'interpréteur fonctionnera quel que soit votre système d'exploitation,
— vous pourrez coder depuis n'importe quel ordinateur connecté à Internet,
— votre interpréteur sera à jour en permanence (avec des nuances),
— votre code est sauvegardé à distance (avec des nuances),
— vous pouvez facilement partager votre code avec d'autres personnes (avec des nuances).

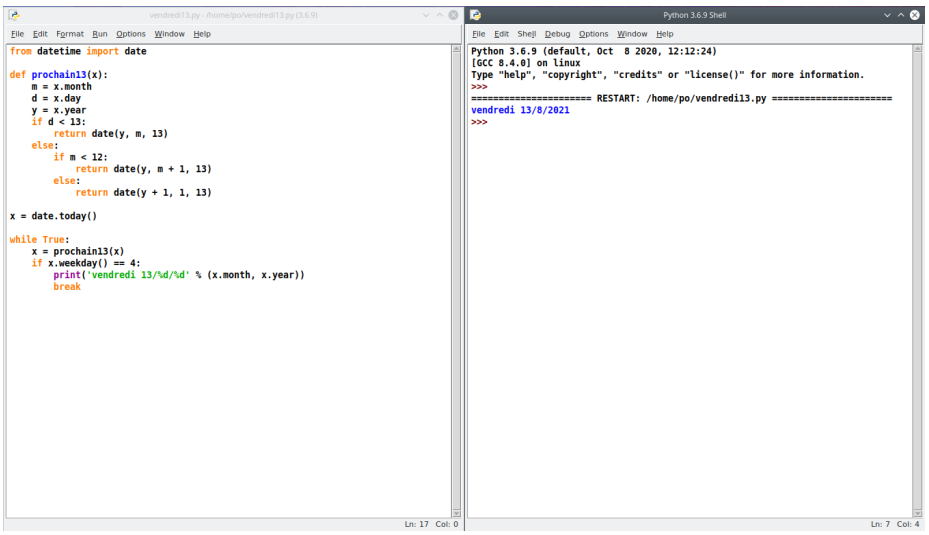
Pour apprendre ou enseigner Python, le choix de l'interpréteur distant est à considérer.

Environnement de programmation

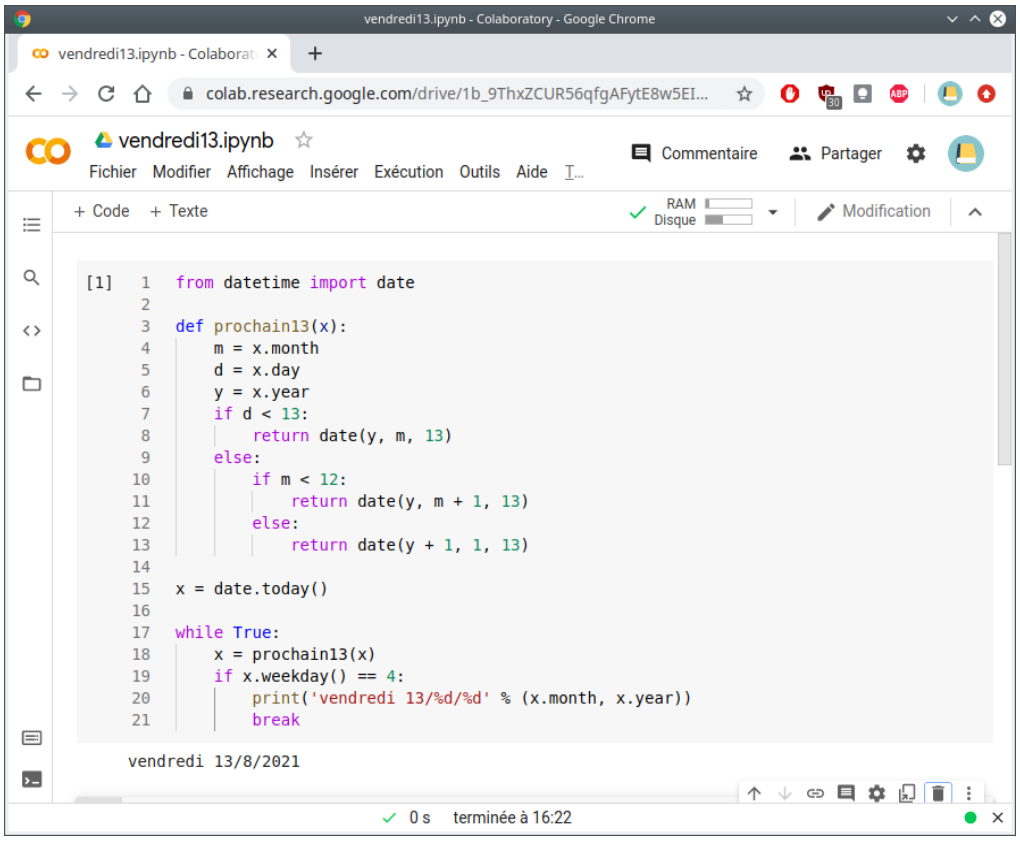
Un *environnement de programmation* pour Python désignera un outil logiciel permettant à la fois d'éditer du code Python et de l'exécuter. Un environnement de programmation peut être être distant ou être installé localement sur votre machine. Par exemple,

— les éditeurs IDLE (cf. la première image ci-dessous) ou VScode sont des éditeurs locaux ;

— des applications web comme Google colab (cf. la deuxième image ci-dessous) ou tutorialspoint proposent un environnement de programmation distant.



Un éditeur local : l'éditeur intégré par défaut IDLE



Un éditeur distant sur Google Colab

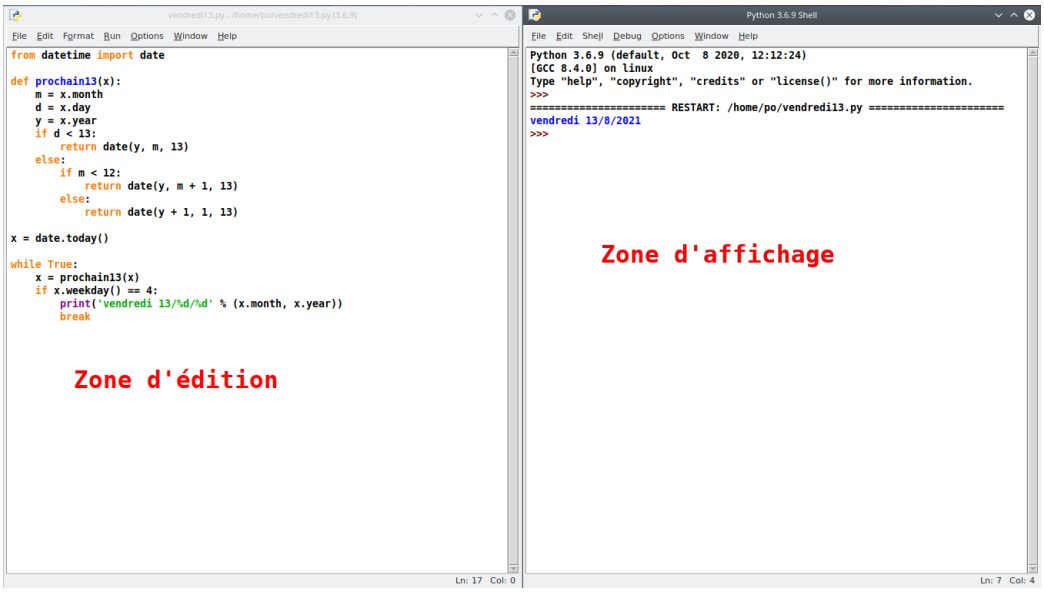
Tout environnement de programmation contient 2 zones bien distinctes (cf. les images ci-après) :

— une zone d'édition de code

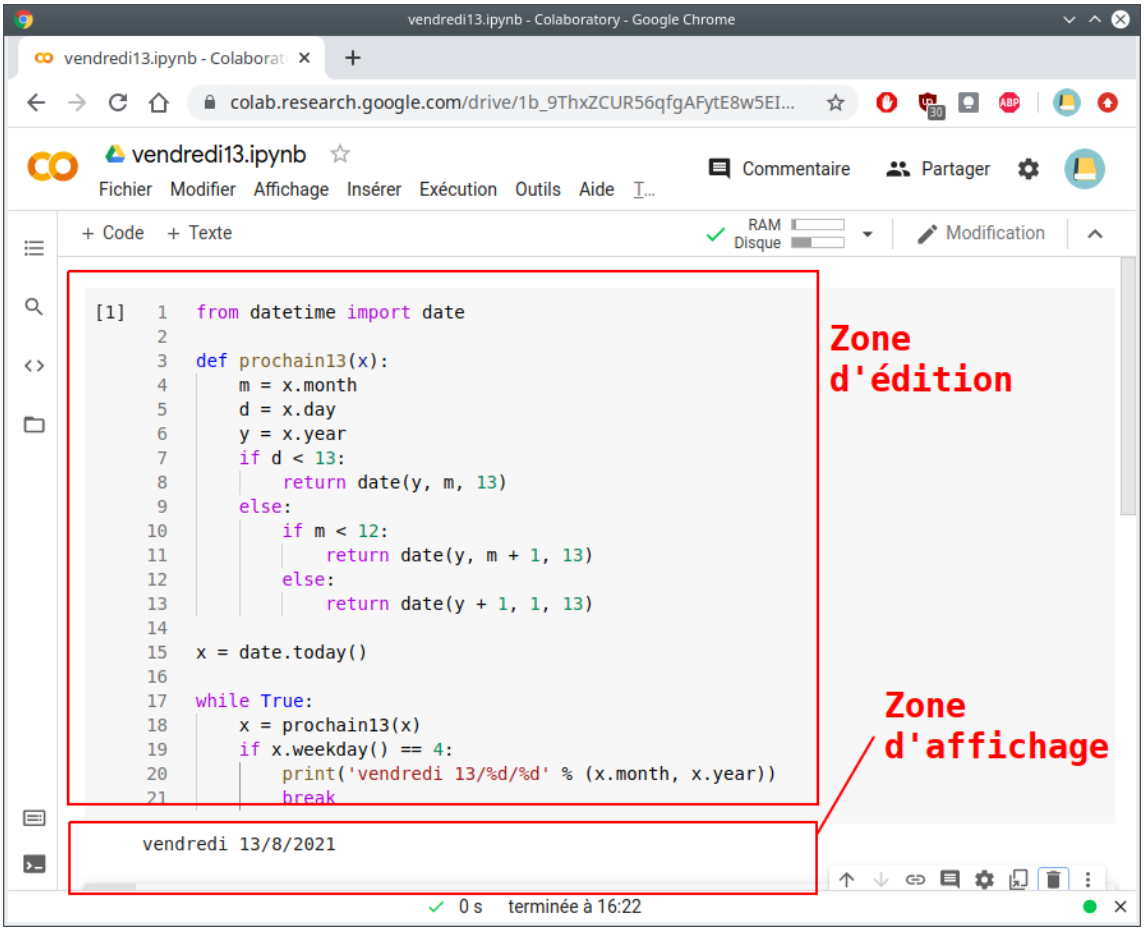
— une zone de sortie

La zone d'édition est la zone de l'éditeur où l'utilisateur écrit son code Python. La zone de sortie est la zone de l'IDE où apparaissent les affichages générés par l'exécution du code de la zone éditeur.

Voici des images de quelques éditeurs intégrés montrant les deux zones :



Zones d'édition de code et d'exécution de code dans l'éditeur IDLE



Cellules d'édition de code et d'exécution de code dans l'éditeur distant JupyterLab

En pratique,

— le programmeur tape son code dans la zone d'édition

— il appuie sur une touche du clavier ou clique sur un bouton ou dans un menu pour exécuter son code

— il lit le résultat de l'exécution dans la zone de sortie.

Quelques environnements de programmation

Voici quelques environnements de programmation en Python gratuits, voire open-source :

— **Jupyter Notebook et JupyterLab** : l'utilisateur écrit en Python dans des cellules de code au sein d'une page web habituelle, contenant du texte, des images, de la vidéo. Cet environnement est utilisé dans les sciences des données et l'IA (cf. les compétitions Kaggle) et peut-être aussi dans l'éducation. JupyterLab est une évolution des feuilles Jupyter Notebook. L'environnement Google Colab utilise JupyterLab. Jupyter Notebook peut s'installer sur sa propre machine, soit via pip soit en utilisant la suite Anaconda.

— **VSCode** : l'éditeur multi-langage et open-source proposé par Microsoft. Selon plusieurs enquêtes, il serait un des éditeurs les plus utilisés, en particulier dans le monde professionnel. Le créateur du langage Python et le créateur de l'extension Python de VSCode travaillent pour Microsoft.

— **IDLE** : il est disponible par défaut quand on utilise la distribution « officielle » téléchargée depuis python.org et donc ne nécessite aucune installation (sauf sous Linux). C'est probablement un des éditeurs les plus simples mais son interface est limitée et, surtout, assez désuète ce qui ne le rend pas très populaire.

— **Spyder** : environnement de programmation en Python destiné aux scientifiques, ingénieurs, analystes de données. Serait utilisé dans l'enseignement.

Aucun environnement de programmation n'est simple à prendre en main quand on en a jamais utilisé et si on débute, il est préférable de choisir un environnement aussi simple que possible (IDLE, Jupyter Notebook) sinon, l'expérience de programmation est altérée. Il y a un certain flottement dans l'usage des termes suivants : éditeur de code, environnement de développement intégré (IDE) et environnement de programmation. Par exemple, VSCode n'est pas considéré comme étant un IDE mais un éditeur.

Installer et lancer Jupyter sous Windows 10 ou 11

L'environnement de travail Jupyter est un module Python et qui s'intalle donc en utilisant le gestionnaire de packages `pip` livré avec Python. Les étapes d'installation sont les suivantes :

— ouvrir ce qu'on appelle une « invite de commande » Windows qui est un terminal où on peut entrer des commandes au clavier ; pour cela,

  — ouvrir la zone de recherche disponible dans la barre des tâches de Windows

  — taper `cmd` dans la barre et valider

  — vous devriez voir apparaître une image du terminal : cliquer dessus pour y accéder.

— devant le prompt, taper la commande `pip install notebook` et valider : cela va installer en moins d'une minute l'environnement Jupyter Notebook. Une fois installé, fermer la ligne de commande.

Ensuite, pour lancer Jupyter Notebook, ouvrir à nouveau une invite de commande, taper `jupyter notebook` et valider. Au bout de quelques secondes (ça peut être plusieurs dizaines de secondes en fait),

dans votre navigateur web, va s'ouvrir la page d'acceuil de Jupyter et qui correspond à un de vos répertoires. De là, vous pouvez créer une feuille Jupyter Notebook ou en ouvrir une qui soit disponible en allant à l'emplacement où se trouve la feuille.

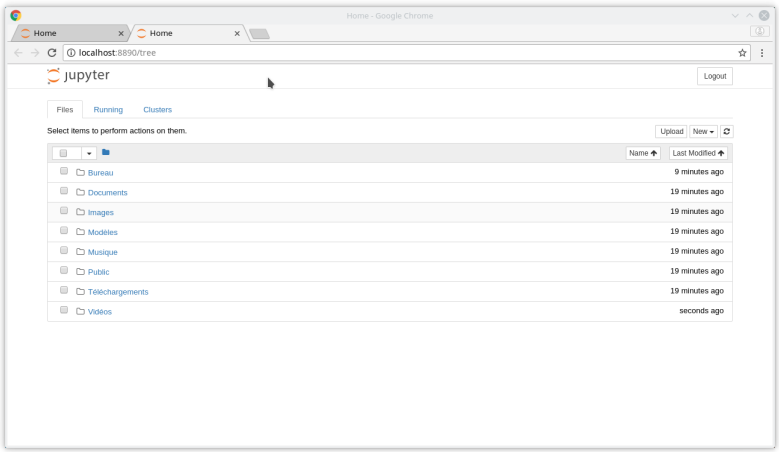
Il existe d'autres méthodes d'installation, comme utiliser Anaconda mais qui n'est pas justifiée pour une utilisation basique de Python. On peut aussi utiliser Jupyter Notebook dans VScode.

Feuilles Jupyter Notebook : usage

Les feuilles de travail Jupyter étant des applications Web, elles utilisent le même format, qu'elles soient utilisées en ligne (Cocalc, Google Colab ou autres) ou en local (Windows, Linux, macOs). Les copies d'écran ci-dessous sont anciennes mais sont encore valides en 2024.

Le dashboard

Après avoir lancé Jupyter, votre nagivateur web s'ouvre sur une page d'accueil permettant d'accéder à des répertoires de votre système de fichiers et d'ouvrir des feuilles `ipynb` : il s'agit du **dashboard**, c'est le terme employé dans la documentation.



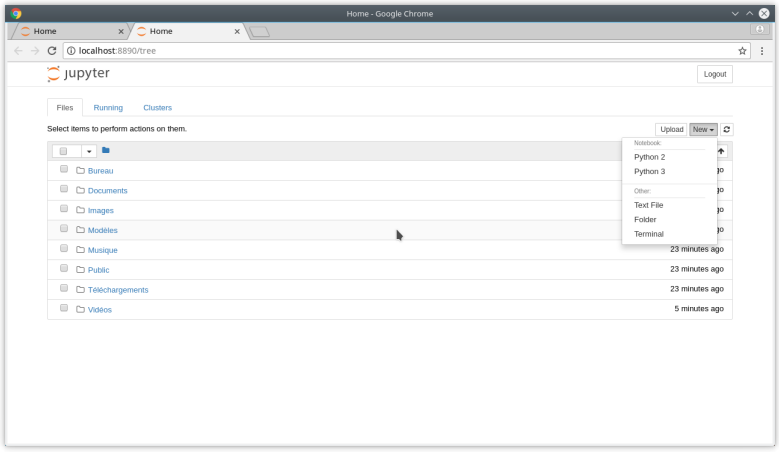
Le dashboard de Jupiter

Sur une installation locale, la page ouverte correspond au répertoire de lancement de Jupyter.

Depuis cette page, on peut naviguer dans une arborescence de répertoires ; on peut aussi créer des répertoires (bouton **New**), lesquels apparaîtront dans votre système de fichiers si votre installation est locale. On peut aussi les renommer (cocher le répertoire et un bouton apparaît), et attention, les **supprimer** (donc par mégarde, vous pouvez supprimer un de vos propres répertoires, sans possibilité de récupération dans la corbeille).

Création d'une nouvelle feuille

Pour créer une nouvelle feuille, naviguez depuis le dashboard jusque dans le dossier où vous voulez que se trouve votre feuille de trvail. Puis, cliquer depuis le dashboard sur le bouton **New** puis choisir le moteur Python 3.



Créer une feuille Jupyter

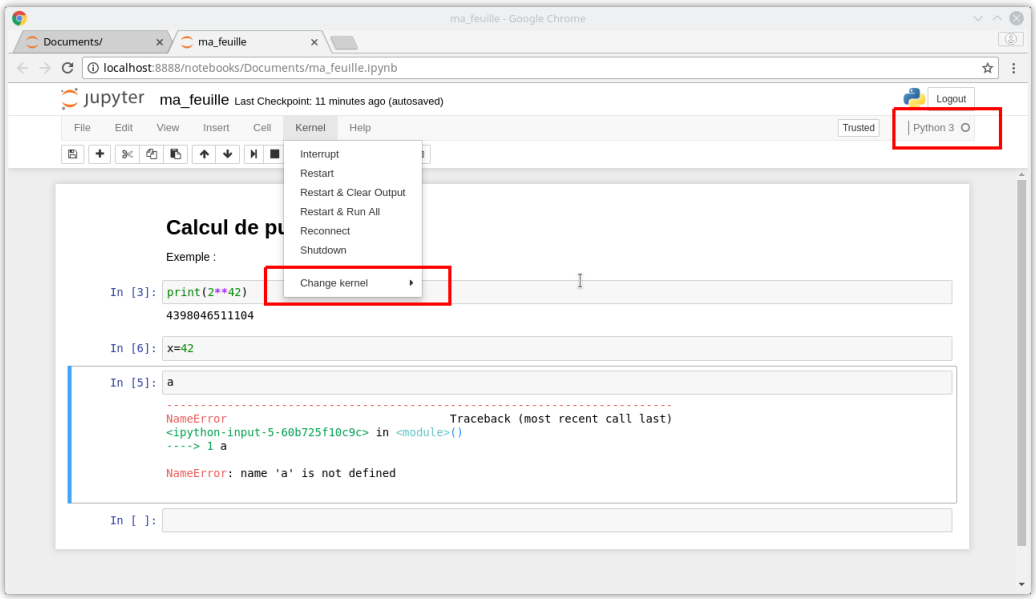
Cela crée dans le répertoire courant affiché par le dashboard, une feuille de travail Jupyter, nommée par défaut **Untitled** et qui s'ouvre automatiquement. Renommez éventuellement la feuille en cliquant sur son nom en haut à gauche. Si on renomme la feuille en **maFeuille**, le nom du fichier créé sera `maFeuille.ipynb` d'extension `ipynb` (ce qui signifie *IPython notebook*).

Ouverture d'une feuille déjà existante

Depuis le dashboard, naviguer vers le répertoire où se trouve ladite feuille et cliquer sur le nom de la feuille (d'extension `ipynb`) ce qui va ouvrir la feuille.

Choix du moteur

En principe, le moteur de votre feuille est Python 3. On peut le vérifier en haut à droite, au niveau de la barre de menus. On peut changer le moteur dans le menu `Kernel > Change Kernel`

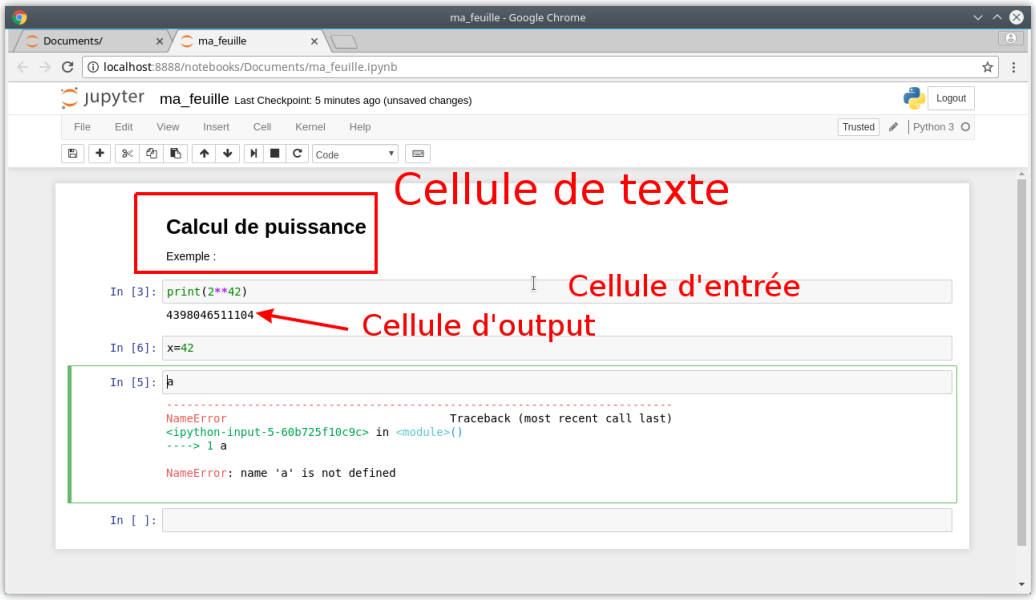


Créer une feuille Jupyter

Description d'une feuille

Une feuille se présente comme une succession de cellules. Il existe 3 catégories de cellules :

— cellule d'input (ou d'entrée) : votre code Python
— cellule d'output : sortie mode en mode texte, ou graphique
— cellule de texte : tout contenu web comme texte, image, vidéo, tableau, math, etc.



Les différentes cellules Jupyter

À la **création** d'une feuille (quand on clique sur **New**), il existe une unique cellule et cette cellule est une cellule de code.

Vous écrivez votre code dans les cellules d'input et on lit les éventuels affichages dans les cellules d'output. Les cellules d'input sont précédées de `In []` et entre crochets figure le numéro dans l'ordre chronologique de l'édition de la cellule.

En tant qu'utilisateur de Python, vous n'êtes pas fondamentalement concerné par les cellules de texte qui sont créées, en langage Markdown, par le créateur de la feuille. Mais par accident, vous pouvez y être confronté ou même volontairement, si vous voulez modifier ou ajouter du texte, voir plus bas.

Entrer du code et l'exécuter

On entre son code dans une cellule d'input comme dans un éditeur de code Python, avec toutes les possibilités habituelles (sélection, suppression, copier-coller, CTRL+Z, coloration syntaxique automatique, etc). La cellule est extensible en hauteur, on peut taper autant de lignes qu'on veut.

Pour valider le code Python d'une cellule, autrement dit pour exécuter ce code, il y a deux possibilités :

— le bouton en triangle de la barre d'outils
— le raccourci clavier MAJ + ENTRÉE.

Si le code est correct, il y a une sortie visible (éventuellement pas de sortie, ça dépend en fait des commandes écrites en entrée) sous la cellule d'entrée, avec éventuellement un message d'erreur concernant le code Python entré.

Il est possible de revenir sur une cellule déjà éditée et de la modifier et de revalider son contenu. La numérotation de la cellule est alors modifiée.

Pour, à la fois, valider une cellule de code **et** ouvrir une cellule de code en dessous, taper Alt + Enter

Réinitialisations

Au fur et à mesure que la feuille est enrichie de code Python exécuté, la mémoire de ce code est enregistrée par la feuille. Pour diverses raisons, il peut être nécessaire de réinitialiser l'état de la mémoire (de faire une purge). Pour cela, cliquer menu Kernel > Restart et accepter la réinitialisation en cliquant sur le bouton coloré.

On peut vouloir nettoyer les cellules de leur sortie (parfois volumineuses). Pour cela menu `Cell > All Output > Clear`

Pour interrompre un code trop long à s'exécuter, deux méthodes :

— menu Kernel > Interrupt.

— bouton carré noir de la barre d'outils

Insertion, suppression

Pour insérer une nouvelle cellule de code, menu Insert.

Pour supprimer une cellule, placer le curseur dans la cellule et aller au menu `Edit > Delete Cells`

Du texte bizarre apparaît

Si par mégarde, vous activez une cellule de texte (en faisant un double clic par exemple), un code bizarre va apparaître (c'est du langage Markdown). Pour remédier à cela, placer le curseur dans la cellule bizarre et TAPER `MAJ+ ENTRÉE`, comme si c'était une cellule de code vous validiez.

Les explications ci-dessus vous montrent aussi comment modifier du texte dans une cellule de texte.

Divers

Les feuilles sont sauvegardées automatiquement à certains intervalles de temps. On peut anticiper la sauvegarde par le menu `File > Save and Checkpoint`

On peut modifier le titre d'une feuille en cliquant sur son nom en haut à gauche.

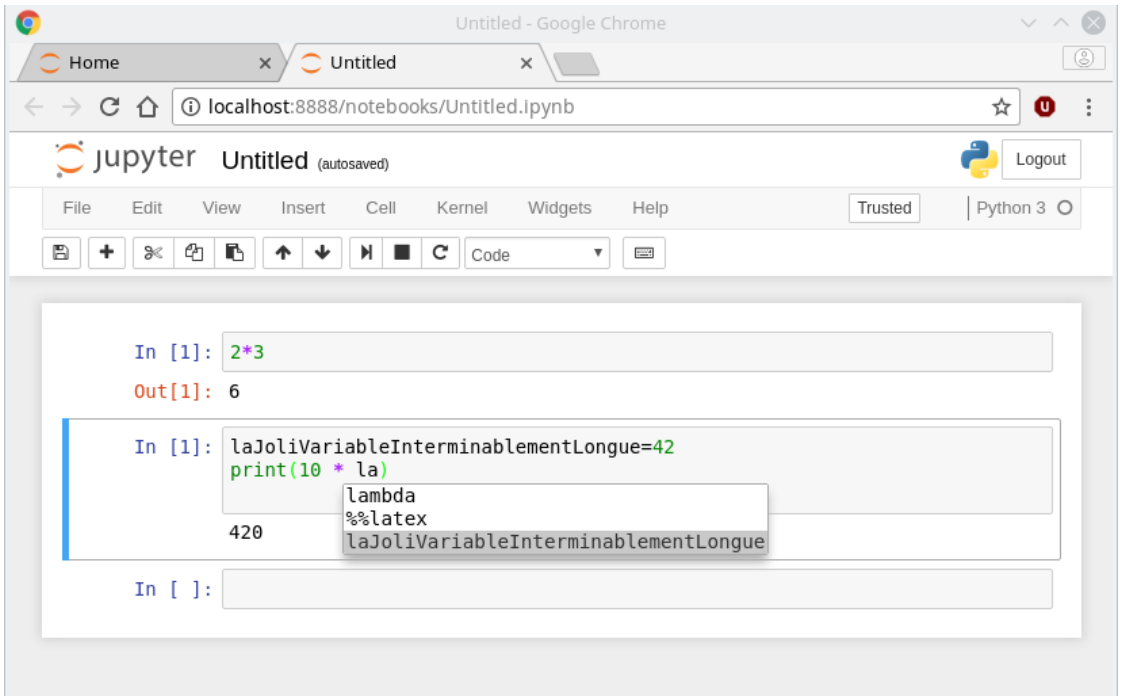
Pour une meilleure lisibilité, on peut cacher les barres de menus et d'outils (menu View).

La complétion syntaxique

Dans une cellule, on tape le code suivant, sans valider :

```
1 laJoliVariableInterminablementLongue=42
2 print(10 * la
```

On veut poursuivre d'écrire le code en écrivant juste derrière le `10 *` la variable à la ligne au-dessus mais comme elle est très longue, on n'a pas envie de la retaper. Pour nous aider, IPython prend en charge la complétion syntaxique. Si on tape les deux ou trois premières lettres de la variable et qu'on appuie sur la touche TAB, le nom de la variable va apparaître, et il suffira de la sélectionner pour qu'elle soit placée dans le code :



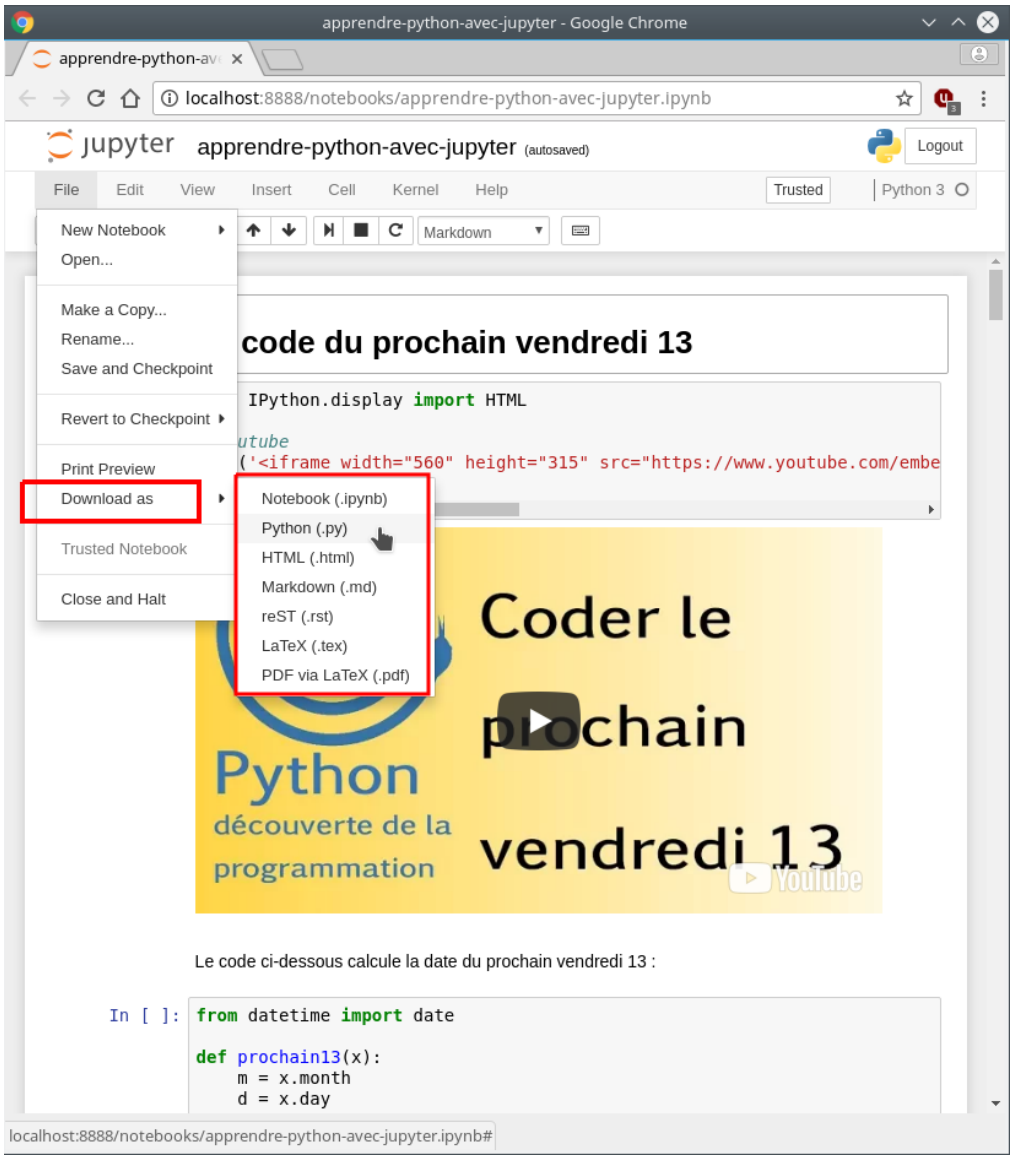
Complétion syntaxique dans Jupyter

Communication entre cellules

Les cellules ne sont pas étanches entre elles. Toute variable déclarée et validée dans une cellule est placée en mémoire et utilisable par les autres cellules, qu'elles soient avant ou après la cellule ou figure cette variable et y compris si la cellule déclarant cette variable est supprimée ou modifiée. Bien sûr, c'est pratique mais cela peut aussi induire des bug dans la mesure où le code d'une cellule dépend de ce qui a été déclaré avant ou ailleurs.

Exportation de la feuille

Il est possible d'exporter une feuille sous différents formats. Pour cela, aller au menu File > Export



Menu d'exportation

On peut exporter dans différents formats dont les formats :

— `ipynb` : c'est le format natif des feuilles Jupyter
— `py` : fichier Python ne contenant que les cellules d'input de la feuille, les autres cellules étant placées en commentaire du code Python.
— `md` : fichier Markdown et format `html`
— `pdf` : le fichier va être créé par Jupyter et il a besoin des programmes latex et pandoc pour générer le pdf (sinon, ça ne marchera pas, surtout si Latex n'est pas installé).

Parcourir la feuille

— Placer le focus dans une cellule, celle à partir de laquelle on va naviguer dans la feuille.
— Taper sur la touche Echap
— Pour descendre page par page, appuyer sur Espace
— Pour remonter page par page, appuyer sur Maj-Espace

Afficher

Soit le code Python dont le contenu est exactement le suivant :

```
5 + 5
2 + 2
```

Lorsqu'on demande à Python d'exécuter ce code, Python calculera $5 + 5 = 10$ puis calculera $2 + 2 = 4$. Ensuite, selon l'environnement de programmation, le programme n'affichera rien ou n'affichera que le résultat du dernier calcul (ici 4).

Pour avoir la **garantie** que le résultat d'une opération sera affiché et correctement affiché, **il faut le demander explicitement** à Python avec une fonction fournie par le langage, la fonction `print` :

```
1 print(5 + 5)
2 print(2 + 2)
```

qui affiche

```
10
4
```

`print` est une fonction *standard* du langage Python pour signifier qu'elle est fournie nativement par le langage Python et immédiatement disponible pour utilisation. Comme c'est une fonction, pour l'utiliser et ainsi obtenir un affichage, il faut placer une paire de parenthèses après le nom `print` de la fonction et placer entre les parenthèses l'objet à afficher.

Afficher un message

On peut afficher un « message » avec la fonction `print`. Dans le code-source, le message est placé entre guillemets :

```
1 print("--->> Bonjour ! <<---")
```

```
2 --->> Bonjour ! <<---
```

Le message peut être plus ou moins élaboré voire fantaisiste.

`print` et saut de ligne par défaut

Soit le code suivant :

```
1 print(42)
2 print(421)
```

A l'exécution, la sortie affichée est

```
1 42
2 421
```

et non

```
1 42421
```

Autrement dit, l'instruction `print(message)` affiche le contenu de `message` immédiatement suivi d'un saut de ligne. Ne pas croire que le saut de ligne dans le code-source explique le saut de ligne dans la sortie, ça n'a pas de rapport.

Variables

Voici un code utilisant une variable, appelée `toto` :

```
1 toto = 42
2 print(toto + 10)
```

```
3 52
```

— Ligne 1 : `toto` désigne une variable qui référence 42. La variable `toto` est comme une *étiquette* qui permet d'accéder à l'entier 42 écrit quelque part en mémoire. L'entier 42 est ce qu'on appelle un *objet* au sens d'une entité présente dans la mémoire de l'ordinateur au moment où Python examine l'instruction `toto = 42`.

— Ligne 2 :

  — d'abord Python regarde s'il dispose d'une variable `toto` référencée en mémoire (c'est le cas, cf. ligne 1) puis il récupère le contenu de cet emplacement-mémoire et effectue le calcul demandé.

  — `toto + 10` est une *expression* et qui a une valeur (ici 52).

  — `print` permet d'afficher la valeur de l'expression.

Un nom de variable est sensible à la casse (lettre en majuscule ou minuscule) :

```
1 year = 2000
2 Year = 3000
3 print(year)
4 print(Year)
```

```
5 2000
6 3000
```

Vocabulaire

— L'instruction `toto = 42` est une *affectation* : on *affecte* 42 à la variable `toto`.

— Quand une variable est définie pour la première fois, la première valeur qu'elle reçoit lorsqu'elle est définie s'appelle l'*initialisation*. Donc, ci-dessus, `toto` est initialisée à 42.

En Python, une variable est définie essentiellement par affectation.

La notion de variable est fondamentale en programmation et est le premier maillon vers l'abstraction. Il est capital d'écrire du code où les données sont placées dans des variables.

Post-it

Un document qui explique très bien le modèle mental de la variable en Python est ICI : une variable Python un **post-it** que l'on peut placer et déplacer sur des objets.

Voir aussi ce message de Ben Finney ainsi que Facts and myths about Python names and values et regarder la vidéo associée.

Notion d'expression

Soit le code suivant :

```
1 d = 4
2 u = 2
3 x = 10 * d + u
4
5 print(10 * x + 1)
```

```
6 421
```

On y définit des variables, par exemple `d` ou `u`, et en combinant ces variables avec des constantes (comme `10`) et des opérateurs comme `*` et `+`, on obtient des « écritures » qui s'appellent des *expressions*, comme `10 * d + u` (ligne 3).

En combinant des expressions à nouveau avec des opérateurs et des constantes, on obtient de nouvelles expressions, comme ci-dessus `10 * x + 1` (ligne 5).

Par définition, une expression a toujours une *valeur*. Par exemple la valeur de l'expression `10 * x + 1` est l'entier 421. En revanche, l'écriture `d = 4` (ligne 1) n'a pas de valeur et n'est pas une expression [1].

Pour afficher une expression `E`, on utilise l'instruction `print(E)` et qui affiche dans une sortie [2] sous forme textuelle la *valeur* de cette expression.

Syntaxe basique des noms de variables

Voici un programme utilisant des noms valides de variables :

```
1 tokyo = 2020
2 tokyo2020 = 32
3 TOKYO = 2020
4 tokyoTokio = 32
```

La plupart du temps, un nom de variable est un mot formé de caractères alphabétiques, (majuscules ou minuscules) ou de chiffres. Il ne peut y avoir aucun élément de ponctuation ni d'espace dans un nom de variable, sinon, la plupart du temps, vous aurez une erreur de type `syntaxError` :

```
1 chou-fleur = 42
```

```
2 SyntaxError: can't assign to operator
```

Par ailleurs, une variable ne peut commencer par un chiffre :

```
1 2020Tokyo = 32
```

```
2 SyntaxError: invalid syntax
```

Les noms de variables sont sensibles à la casse :

1. À la différence de langages comme C ou Java.
2. La sortie dépend de l'environnement d'utilisation (mode fichier, Jupyter, console Python, etc)

```
1 tokyo = 2020
2 print(Tokyo)
```

— Ligne 2 : les variables `Tokyo` et `tokyo` sont différentes. Le 2e nom n'a pas été défini (d'où l'erreur).

Certains noms réservés du langage sont interdits comme nom de variable :

```
1 bronze = 3
2 argent = 2
3 or = 1
```

— Ligne 3 : `or` est un mot-clé du langage (qui signifie **ou** en anglais).

Le seul caractère non-alphanumérique autorisé est le blanc souligné :

```
1 paris_2024 = 42
2 print(paris_2024)
```

— Ligne 3 : `or` est un mot-clé du langage (qui signifie **ou** en anglais).

Pour saisir un blanc souligné, taper simultanément `AltGr + 8`.

Nom ou variable non reconnus

Une variable qui est accédée sans avoir été affectée au préalable n'est pas reconnue :

```
1 R = 10
2 print(2 * pi * R)
```

```
3 NameError: name 'pi' is not defined
```

— Ligne 2 : la variable `pi` est utilisée mais n'a jamais été défini auparavant.

— Ligne 3 : le nom de variable `pi` n'est pas reconnu d'où le message `NameError` qui doit s'interpréter par : « nom inconnu ».

Ce type d'erreur provient parfois d'une erreur de saisie du code (une « coquille ») :

```
1 toto = 42
2 print(totot * 10)
```

```
3 NameError: name 'totot' is not defined
```

Ce phénomène s'applique aussi à des noms (par exemple, de fonctions « officielles » du langage), pas seulement des variables :

```
1 toto = 42
2 printf(toto * 10)
```

```
3 NameError: name 'printf' is not defined
```

— Ligne 2 : le programmeur a saisi `printf` au lieu de `print`.

Réaffectation de variable

Lorqu'un calcul avec une variable x est suivie de la modification de la valeur de la variable x, on dit que x a subi une réaffectation :

```
1 x = 42
2 x = 2 * x + 10
3 print(x)
```

```
4 94
```

— Ligne 2 : l'expression `2 * x + 10` est évaluée et la valeur obtenue (94) est réaffectée à `x`. Ce type d'affectation (ligne 1 vs ligne 2) s'appelle une *réaffectation*. La valeur initiale d'affectation (ici 42) est « perdue ».

Augmenter de 1 la valeur d'une variable (comme `toto` ci-dessous) s'appelle une *incrémentation* :

```
1 toto = 42
2 print(toto)
3
4 toto = toto + 1
5 print(toto)
```

```
6 42
7 43
```

— Ligne 4 : l'expression `toto + 1` est évaluée et la valeur obtenue (43) est réaffectée à `toto`.

La nécessité d'effectuer une réaffectation est extrêmement courante.

Notions sur les constantes

Soit le code Python suivant :

```
1 z = 31
2 a = 42 * z + 81
3 z = "toto"
4 pi = 3.14159
```

Dans ce code,

`31`, `42`, `81`, `"toto"` ou `3.14159`

sont des *constantes* du langage Python, Elles sont dites parfois *littérales* car leur façon de s'écrire dans le code permet de connaître immédiatement leur valeur, sans faire aucune opération. Par contre,

`42 * z + 81`

N'est PAS une constante (c'est juste une *expression* qui utilise des constantes). Les constantes sont des briques élémentaires (parmi d'autres) des « écritures » d'un programme.

Pour afficher une constante présente dans un code Python, on utilise la fonction `print`.

```
1 a=42
2 print(a)
3
4 z="toto"
5 print(z)
6 print("toto")
7
8 pi = 3.14159
9 print(pi)
```

```
10 42
11 toto
12 toto
13 3.14159
```

— Lignes 5 ou 9 : instruction d'affichage du contenu d'une variable dont le contenu (lignes 4 ou 8) est une constante.

— Ligne 6 : on peut aussi obtenir l'affichage de la constante sans l'avoir placée dans une variable.

— Lignes 10-13 : les affichages correspondants.

Constantes entières

Les constantes entières figurant textuellement dans un code , telles que `421`, peuvent être précédées d'un signe `-` pour désigner un entier négatif ou d'un signe `+`, facultatif et en général omis :

```
1 print(-42)
2 print(+421)
```

```
3 -42
4 421
```

Toute constante entière, écrite de la façon habituelle avec des chiffres entre 0 et 9, comme `421` ou `0` est interprétée par défaut comme écrite en base 10.

Attention à ne pas placer involontairement un 0 comme premier chiffre d'une constante entière, comme on le ferait pour certaines dates (par exemple `01` dans la date **19/01/2038**) sinon on obtient un message d'erreur :

```
1 print(042)
```

```
2   File "_.py", line 1
3     print(042)
4               ^
5 SyntaxError: invalid token
```

Grands entiers

On peut effectuer tout type d'opération arithmétique avec des entiers aussi grands qu'on le souhaite. Une opération arithmétique ne peut jamais créer d'« overflow » comme c'est le cas en C ou Java où les entiers sont codés sur un nombre prédéfini de bits.

Exemple d'usage de grands entiers :

```
1 a = 8596125023601452569859
2 b = 7012365895410232252363
3 print(a * b)
```

```
4 60279173948185303803365326980576772175326817
```

Constantes flottantes

La présence d'un point dans une constante numérique donne à la constante le type flottant. Le point est, dans ce cas, l'équivalent de notre virgule décimale :

```
1 print(3.14)
```

```
2 3.14
```

— Ligne 1 : le nombre `3.14` est une constante flottante ; il y a des chiffres de part et d'autre du point

— Ligne 2 : l'affichage est obtenu via la fonction `print`.

Il est parfois possible qu'aucun chiffre ne précède ou ne suive le point :

```
1 print(1.)
2 print(.25)
```

```
3 1.0
4 0.25
```

— Ligne 1 : `1.` est équivalent à `1.0`

— Ligne 2 : `.25` est équivalent à `0.25`

Le but de ce type d'écriture est juste d'alléger la saisie du code.

En France, le séparateur décimal est une virgule. Il faut cependant toujours utiliser un point pour désigner le séparateur d'une constante flottante. Voici un exemple de comportement inattendu :

```
1 pi = 3,14
2 print(pi + 1)
```

```
3 TypeError: can only concatenate tuple (not "int") to tuple
```

— Ligne 1 : on déclare le nombre `pi` sous la forme `3,14` au lieu de `3.14`.

— Lignes 2 : On veut ajouter 1 à `pi` (on devrait obtenir `4,14`) mais le programme est interrompu.

On peut faire précéder une constante flottante d'un signe `-` ou `+` :

```
1 print(-3.14)
2 print(+3.14)
```

```
3 -3.14
4 3.14
```

Types de base

Les objets manipulées par un programme Python ont un *type*, au sens d'un langage de programmation : un type correspond à une catégorie d'objets (entiers, nombres réels, valeurs logiques, etc).

Voici quelques exemples de types usuels :

```
1 print(42)
2 print(4 + 6)
3 print(-5 * 2)
4
5 print(421.5)
6 print(2.7 + 10)
7
8 print(421 > 42)
9 print(421 < 42)
10
11 print("toto")
```

```
12 42
13 10
14 -10
15 421.5
16 12.7
17 True
18 False
19 toto
```

— Lignes 1-3 : type *entier*. Les entiers peuvent être positifs comme négatifs.

— Lignes 5-6 : type *flottant*. Pour simplifier, il s'agit de nombres dit « à virgule », ci-dessus représentés en base 10. Le point utilisé dans le nombre représente notre virgule décimale (« flottant » fait allusion à la notion de virgule « flottante »).

— Lignes 8-9 et 17-18 : type *booléen*. Une expression de type booléen a une valeur de vérite `True` ou `False`.

— Lignes 11 : le type *chaîne*. En première approximation, une chaîne représente une suite de caractères. Dans l'exemple, pour que Python reconnaisse le mot **toto** comme une donnée de type chaîne, on entoure le mot d'une paire de guillemets.

Même valeur et types différents

Des objets peuvent avoir même valeur et des types différents. Par `10.0` et `10` ont même valeur

```
1 a=10
2 b=10.0
3
4 print(a, "==", b, ":", a==b)
```

```
5 10 == 10.0 : True
```

mais ils ont des types différents ce qui se traduit par des comportement différents vis à vis de certains opérateurs.

Par exemple, si on multiplie 10 par un très grand entier N, on ne trouve pas le même résultat que si on multiplie 10. et le grand entier N, comme le code suivant le montre :

```
1 a=10
2 b=10.0
3 N=123456789101234567
4
5 print(a*N == b*N)
```

```
6 False
```

Variables et typage

Un nom de variable n'est pas associé à un type fixé

```
1 a = 2020
2 print(a)
3 a = 3.14
4 print(a)
```

```
5 2020
6 3.14
```

— Ligne 1 : l'étiquette a se réfère d'abord à un entier.

— Ligne 3 : l'étiquette a se réfère ensuite à un flottant.

Une variable n'a pas de type. Ce qui a un type est l'objet placé en mémoire et que la variable référence.

Une variable peut étiqueter un objet de n'importe quel type.

Par exemple, on peut placer une variable sur le nom d'une fonction :

```
1 p = print
2 p("bonjour")
```

```
3 "bonjour"
```

Les fonctions built-in

Python propose, par défaut, des fonctions prêtes à l'emploi (les fonctions dites « built-in » [3]) et qui permettent de faire des opérations courantes en programmation : afficher, trier une liste, trouver le plus grand nombre d'une liste, calculer la longueur d'une chaîne de caractères, etc.

Exemple

```
1 print(max(81, 12, 31, 82, 65))
```

```
2 82
```

3. Que l'on pourrait traduire par *fonctions* **intégrées**.

— Ligne 1 : `max` est une fonction built-in. Elle renvoie le plus grand élément des nombres qu'on lui transmet.

La fonction built-in la plus utilisée est probablement la fonction `print`.

Indentation accidentelle

Le placement d'espaces horizontaux **en début** de ligne n'est autorisé que dans certaines circonstances très strictes.

En cas de non respect de ces règles, on obtient une interruption du programme :

```
1 x = 3000 - (2038 - 38)
2  print("Bonjour !", x)
```

```
3   File "__.py", line 2
4     print("Bonjour !", x)
5     ^
6 IndentationError: unexpected indent
```

Le message `IndentationError` signifie que des espaces en **début** de ligne sont mal placées.

Placer des commentaires

Il est possible d'annoter son code-source par des commentaires personnels. Voici un exemple de code-source commenté :

```
1 # Mon joli code
2
3
4 # Voici un exemple
5 # d'addition
6 a=4+2
7
8     # on va afficher le résultat
9 print(a) # affichage
10 # affiche 6
```

Un commentaire Python est introduit par un caractère dièse [4] comme ceci : `#` (cf. ligne 4 par exemple).

Tout ce qui suit un caractère `#` est un commentaire. Le code `commentaire.py` contient 6 commentaires.

La présence de commentaires n'a pas d'impact sur l'exécution du code [5] et ils sont supprimés du code avant l'exécution de celui-ci. Le code ci-dessus est exactement équivalent au code suivant :

4. Le caractère `#` porte différents noms : dièse, sharp, « pound », « hash » (ce dernier terme est utilisé dans la documentation officielle) ; selon Wikipedia, le terme français correct semblerait être « croisillon ».

5. Un commentaire n'est pas une instruction du langage.

commentaires_retires.py

```
a=4+2
print(a)
```

Les commentaires sont destinés au programmeur lui-même ou à un autre programmeur qui va lire le code afin d'en expliquer certaines parties.

Opérations sur les nombres

Voici quelques opérations usuelles que l'on peut effectuer entre nombres réels (pas seulement entiers).

On peut calculer des expressions mathématiques avec des parenthèses :

```
print(-421)
print(2 * 10 + 15)
print((2 + 3) * (8 + 2))
print(42.1 / 10)
```

```
-421
35
50
4.21
```

— Lignes 4 et 8 : on effectue la division « exacte » de 42.1 par 10.

Certains calculs nécessitent le placement de parenthèses. Par exemple, en mathématiques, on écrit couramment

$$\frac{a}{bc}.$$

En Python, cela NE se traduit PAS par `a / b * c` mais par `a /(b * c)` :

```
x = 100 / 10 * 5
y = 100 / (10 * 5)
print(x, y)
```

```
50.0 2.0
```

En cas de doute dans un calcul, placer des parenthèses même s'il est possible qu'elles ne soient pas indispensables. Ainsi, la fraction

$$\frac{\frac{100.0}{5}}{4}$$

sera traduite par `(100. / 5) / 4` :

```
x = (100 / 5) / 4
y =  100 / 5 / 4
print(x, y)
```

```
5.0 5.0
```

Ne pas utiliser de crochets à la place des parenthèses car les crochets ont un autre sens en Python.

Pour calculer une puissance x^y, on écrit `x ** y` dans le code Python :

```
1 print(10 ** 2)
2 print(6.25 ** 0.5)
3 print(0 ** 0)
```

```
4 100
5 2.5
6 1
```

Les nombres x et y peuvent n'être pas des entiers.
En particulier, `x ** 0.5` représente $\sqrt{x}$, la racine carrée de x (lignes 2 et 5)

Les puissances de 10

Python permet en particulier de manipuler aisément les puissances de 10.

```
1 print(10000000000000)
2 print(10**13)
```

```
3 10000000000000
4 10000000000000
```

— Ligne 1 : écriture, relecture délicates
— Ligne 2 : écriture, relecture immédiates

L'utilisation de puissances de 10 permet d'éviter des difficultés de saisie et de lecture dues à un grand nombre de zéros. Par exemple, pour écrire

— deux millions, utiliser la notation `2 * 10 ** 6`
— deux milliards, utiliser la notation `2 * 10 ** 9`

Division flottante

Pour obtenir la division *exacte*[6] du nombre `a` par le nombre `b`, on utilise tout simplement la syntaxe `a / b` :

```
1 print(366/ 12)
2 print(3.14 / 2)
```

```
3 30.5
4 1.57
```

Les nombres `a` et `b` peuvent aussi bien être des entiers que des flottants. Le résultat `a / b` sera toujours de type flottant.

Utiliser des variables

Quand on veut « récupérer » (autrement dit *sauvegarder*) le résultat d'un calcul, on le place dans une nouvelle variable :

6. Le terme de division *exacte* est abusif puisque la division `a / b` n'a aucune raison d'être *exacte* au sens mathématique du terme mais elle ne sera bien souvent qu'une approximation. Par exemple, $1/3$ n'est pas *strictement* égal à 0.3333333333333333.

```
1 x = 42
2 y = 5
3 z = (x * y + 5) * (x ** 2 + y **2 ) - 100
4 u = (z - 10000) * x
5
6 print(u)
```

```
7 15730470
```

— Lignes 3 : on stocke sous le nom de z le résultat d'un calcul en mémoire

— Lignes 4 : on stocke sous le nom de u le résultat d'un autre calcul

— Ligne 6 : on affiche le contenu de la variable

Une expression utilisant une variable ne modifie pas le contenu de la variable :

```
1 toto = 42
2 print(toto + 1)
3 print(toto)
```

```
4 43
5 42
```

— Ligne 2 : on ajoute 1 à la **valeur** de `toto`. Le résultat (ici 43) est affichée mais ce calcul est « perdu » puisqu'il n'est pas replacé dans une variable.

— Ligne 5 : bien qu'on ait ajouté 1 à `toto`, le contenu de la variable `toto` reste inchangé par rapport à sa valeur initiale.

Exercice type – Volume de la sphère

Le volume de la sphère est

$$V = \frac{4}{3}\pi r^3$$

Écrire en Python une formule (avec des variables) qui calcule le volume V d'une sphère de rayon r. On posera `pi = 3.14`. Tester avec $r = 10$.

Solution

D'abord, écrire la formule de manière purement formelle ne fonctionne pas :

```
1 v = 4/3 * pi * r**3
2 print(v)
```

```
3 NameError                                 Traceback (most recent call last)
4 <ipython-input-32-b6f8fcc9c76e> in <module>()
5 ----> 1 v = 4/3 * pi * r**3
6       2 print(v)
7
8 NameError: name 'pi' is not defined
```

Python n'est pas un outil de calcul formel.

Il faut initialiser `r` mais aussi π :

```
1 pi = 3.14
2
3 r=10
4 v = 4/3 * pi * r**3
5 print(v)
```

```
6 4186.666666666667
```

Pour tester d'autres valeurs, le plus simple est de changer `r` dans le code-source et de relancer l'exécution du code.

Opérations mixtes entre flottants et entiers

Lorsqu'on effectue un ensemble d'opérations arithmétiques qui font intervenir des entiers et des flottants, le résultat est de type flottant :

```
1 x = 1.0
2 y = 42
3
4 print(x * y)
5 print(x - x)
```

```
6 42.0
7 0.0
```

— Lignes 4 et 5 : bien que les valeurs des flottants représentent des nombres entiers, le résultat est de type flottant

Opérateurs « égale à » et « différent de »

Opérateur ==

On peut tester l'égalité de *valeur* de deux objets avec l'opérateur == :

```
1 print(42 == 21 + 21)
2 print(42 == 10 + 10)
```

```
3 True
4 False
```

— Lignes 3 : si les valeurs sont égales l'opérateur == ligne 1 renvoie la valeur logique `True`

— Ligne 4 : si les valeurs comparées ne sont pas égales (42 et 20) l'opérateur == renvoie `False`.

Il faut effectivement écrire deux fois de suite le signe =, sans espace entre les deux.

Quand on lit `True` (ligne 1, ligne 4), il faut comprendre que l'expression `42 == 21 + 21` est un « machin » (un *objet* pour être correct) qui **vaut** `True`. Le code ci-dessous permet peut-être de mieux le comprendre :

```
1 machin = (42 == 21 + 21)
2 print(machin)
```

```
3 True
```

Opérateur !=

Pour tester si deux objets ont des valeurs distinctes, on utilise l'opérateur `!=` (lire : *différent de*). C'est l'opérateur « contraire » de `==` :

```
1 print(42 != 21 + 21)
2 print(42 != 10 + 10)
```

```
3 False
4 True
```

Comparaison de nombres

Le langage Python dispose d'opérateurs permettant d'effectuer des comparaisons de nombres :

```
1 print(42 > 24)
2 print(42 <= 24)
3 print(3.14159 < 2.71828)
4 print(24 >= 2 * 12)
```

```
5 True
6 False
7 False
8 True
```

On peut comparer des nombres au sens des opérateurs mathématiques suivants :

$$< \quad \leq \quad > \quad \geq$$

En Python, on utilise les opérateurs suivants :

<, <=, > et >=

Une comparaison renvoie un booléen `True` ou `False` (*Vrai* ou *Faux*) qui précise si l'inégalité écrite est vraie ou fausse.

Encadrement

On peut utiliser les opérateurs de comparaison en les enchaînant, comme on le fait usuellement en mathématiques :

```
1 print(42 < 81 < 100 <= 512)
2 print(42 < 81 < 100 < 100)
```

```
3 True
4 False
```

Comparaison de flottants

L'arithmétique en nombres flottants est approchée. L'exemple ci-dessous illustre le type de problème rencontré :

```
1 print(6 * 0.7 == 4.2)
```

```
2 False
```

— Ligne 1 : mathématiquement, $6 \times 0,7 = 4,2$

— Ligne 2 : pourtant, Python ne considère pas les deux nombres comme identiques !

De même :

```
1 print(42/6==4.2/0.6)
```

```
2 False
```

Cette limitation provient l'impossibilité de représenter **exactement** en base 2 et avec un nombre limité de bits certains nombre réels.

D'où la conséquence suivante :

Règle fondamentale : on n'utilise *jamais* l'opérateur == pour comparer deux flottants.

Confusion entre égalité et affectation

Le signe = est le signe de l'affectation. Il ne faut pas le confondre avec l'opérateur == d'égalité de valeurs. C'est un source fréquente d'erreur d'inattention, y compris chez les programmeurs non débutants.

Assez souvent, la confusion déclenche un message d'erreur :

```
1 x == 42
2 print(x)
```

```
3 Traceback (most recent call last):
4   File "_.py", line 1, in <module>
5     x == 42
6 NameError: name 'x' is not defined
```

— Ligne 1 : confusion de == avec = : la ligne examine si les valeurs de x et 42 sont égales mais x n'a pas été défini.

— Ligne 6 : la nature de l'erreur est confirmée par le message.

Autre erreur possible :

```
1 x = 42
2 print(x = 421)
```

```
3 Traceback (most recent call last):
4   File "egalite_vs_affectation.py", line 2, in <module>
5     print(x = 421)
6 TypeError: 'x' is an invalid keyword argument for this function
```

— Ligne 2 : confusion de = avec == (une affectation n'est pas une expression donc on ne peut pas l'afficher)

— Lignes 3-6 : l'erreur est plus difficile à interpréter et nécessite des connaissances plus poussées sur la fonction `print` et les fonctions en général.

Mais parfois, il n'y a aucun déclenchement d'erreur :

```
1 x = 42
2 print(x)
3 x == 100
4 print(x)
```

```
5 42
6 42
```

— Ligne 3 : sans doute le programmeur voulait-il écrire une nouvelle affectation `x = 100`. Mais cette erreur de programmation ne déclenche aucune erreur d'exécution.

Multiples, division entière

On rappelle quelques notions d'arithmétique élémentaire. En particulier, la notion de division entière, vue à l'école primaire mais dont le concpet n'est pas toujours maîtrisé et qui est essentielle en programmation.

Multiples

Les nombres qui sont les résultats dans une table de multiplication sont appelés des *multiples*. Par exemple, les 10 premiers multiples de 9 sont :

```
 1 x  9 =  9
 2 x  9 = 18
 3 x  9 = 27
 4 x  9 = 36
 5 x  9 = 45
 6 x  9 = 54
 7 x  9 = 63
 8 x  9 = 72
 9 x  9 = 81
10 x  9 = 90
```

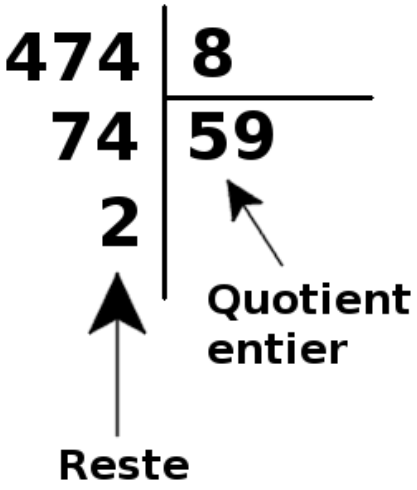
Plus généralement, tout multiple de 9 est de la forme $9 \times k$ où k est un entier. Par exemple, $720 = 9 \times 80$ est un multiple de 9.

Si un nombre n est un multiple de 9, il est équivalent de dire que 9 est un *diviseur* de n ou encore que n est *divisible* par 9.

Réciproquement, soit le problème suivant :

474 est-il un multiple de 8 ?

Pour répondre, il suffit de faire la division à la façon de « l'école primaire » de 474 par 8 :



La division de l'école primaire

Si le reste de cette division vaut 0 autrement dit, si la division tombe juste, c'est que la réponse est *oui* et si le reste est non nul, la réponse est *non*. Ici, la division de 474 par 8 admet 2 pour reste, en sorte que la réponse est *non*.

La division de l'école primaire est en fait appelée *division entière*. La division entière fournit un *reste* et un *quotient*. Par exemple, la division entière de 474 par 8 admet 59 pour quotient et 2 pour reste.

Plus généralement, quand on effectue la division entière de a par b, on obtient un quotient q et un reste r. Noter que r vérifie : $0 \leq r < b$ autrement dit le reste est toujours plus petit que le diviseur. Et bien sûr, on a

$$a = bq + r$$

Noter qu'avec les notations précédentes, le plus grand multiple de b inférieur ou égal à a est exactement $a - r$.

La division entière est différente de la division dite « exacte ». La division exacte ne fournit qu'un quotient, supposé exact. Par exemple, la division exacte de 474 par 8 est 59.25, ce qui s'écrit

$$\frac{474}{8} = 59.25$$

En réalité, très souvent, le quotient obtenu est une valeur approchée : par exemple, on écrit couramment $\frac{474}{7} = 67.71$ bien qu'en fait 67.71 ne soit qu'un approximation de la division de 474 par 7, qui « ne tombe pas juste ».

La notion de multiple donne fréquemment lieu à des questions de programmation.

Quotient et reste de division entière

Le quotient **entier** de l'entier `a` par l'entier `b` est obtenu par l'opération `a // b` (double oblique) :

```
1 print(42 // 10)
```

```
2 4
```

— Ligne 1 : Noter le double slash pour effectuer la division entière, et non `42 / 10` .
— Ligne 2 : noter que la réponse n'est pas **4,2** (il ne s'agit pas de la division exacte).

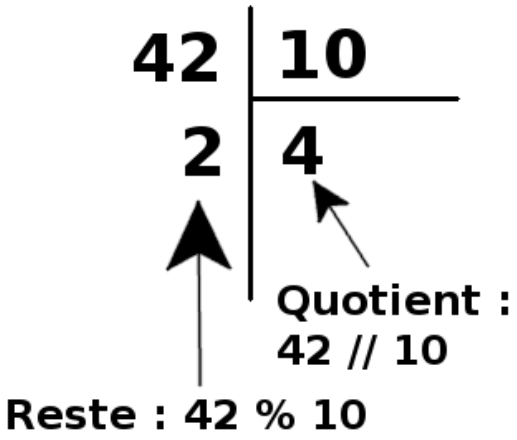
Le reste entier de la division de l'entier `a` par l'entier `b` est noté `a % b` :

```
1 print(42 % 10)
```

```
2 2
```

— Ligne 1 : le calcul du reste se fait avec l'opérateur `%` « pourcent ».

L'opérateur `%` est parfois appelé *opérateur modulo*.



Les opérateurs de division

Division par zéro

Toute forme de division par 0 est interdite et renvoie une erreur :

```
1 print(42 // 0)
```

```
2 Traceback (most recent call last):
3   File "division_par_zero.py", line 1, in <module>
4     print(42 // 0)
5 ZeroDivisionError: division by zero
```

— Ligne 1 : quotient par 0 : interdit

De même :

```
1 print(42 % 0)
```

```
2 Traceback (most recent call last):
3   File "reste_par_zero.py", line 1, in <module>
4     print(42 % 0)
5 ZeroDivisionError: integer division or modulo by zero
```

— Ligne 1 : reste en divisant par 0 : interdit

Division entière et quotient exact

Si b est un entier multiple de l'entier d, la division exacte b/d a même valeur (mathématique) que le quotient entier de b par d. Mais en Python, le quotient exact sera de type flottant tandis que le quotient entier de type entier :

```
1 b = 42
2 d = 6
3
4 print(b // d)
5 print(b % d)
6
7 print(b / d)
```

```
8 7
9 0
10 7.0
```

— Lignes 5 et 9 : `b` est bien un multiple de `d` puisque le reste est nul.
— Lignes 8 et 10 : le quotient entier et le quotient exact ont même valeur mathématique mais pas même type.

Bien que les valeurs des quotients soient exactes, la différence de type peut entraîner des erreurs. Par exemple, la fonction `range`, qui sera vue plus tard, n'admet en arguments que des objets de type entier :

```
1 b = 42
2 d = 6
3
4 print(b // d)
5 print(b % d)
6
7 print(b / d)
8
9
10 print(list(range(b // d)))
11 print("----------------------------")
12
13
14 # Va entraîner une erreur
15 print(list(range(b / d)))
```

```
16 7
17 0
18 7.0
19 [0, 1, 2, 3, 4, 5, 6]
20 ----------------------------
21 Traceback (most recent call last):
22   File "division_entiere_et_quotient_exact.py", line 13, in <module>
23     print(list(range(b / d)))
24 TypeError: 'float' object cannot be interpreted as an integer
```

Il se peut même que les quotients calculés soient différents alors que les valeurs mathématiques soient égales :

```
1 b = 9563862022090661
2 d = 1
3 print(b // d)
4 print(b % d)
5
6 print(b / d)
```

```
7 9563862022090661
8 0
9 9563862022090660.0
```

En conclusion, si on cherche le quotient exact d'un entier b par un de ses *diviseurs*, on utilisera l'opérateur de division entière.

Traduire « être multiple » en Python

Pour savoir si un entier n est pair, on examine le reste r de n dans la division par 2. Si r vaut 0, l'entier n est pair et si le reste vaut 1 alors n est impair. Pour calculer r, on utilise l'opérateur % (modulo) :

```
1 r = 2030 % 2
2 print(r == 0)
3
4 r = 421 % 2
5 print(r == 0)
```

```
6 True
7 False
```

— Lignes 1-2 : le reste 0 montre que 2030 est bien pair

— Lignes 3-4 : le reste 1 montre que 421 est bien impair.

De même, pour savoir si un entier n est multiple de 10, on calcule le reste de n par 10 et on examine si ce reste vaut 0 ou pas :

```
1 r = 2020 % 10
2 print(r == 0)
3
4 r = 421 % 10
5 print(r == 0)
```

```
6 True
7 False
```

D'une façon générale, pour traduire que :

— d est diviseur de a

— a est un multiple de d

on examine si le reste `a % d == 0` vaut `True` ou `False`. On peut aussi examiner si `a % d != 0`.

Affichage multiple

Il est possible d'afficher plusieurs objets sur une même ligne à l'aide d'un seul appel à `print`.

```
1 print("toto", 42+100)
```

```
2 toto 142
```

— Ligne 1 : on affiche la chaîne `toto` et le nombre `42+100`.

— Ligne 2 : les deux objets à afficher (`toto` et le nombre `152`) sont séparés exactement d'**un seul** espace.

Les objets sont affichés sur une même ligne, séparés par un seul espace. Placer plus ou moins d'espacements dans l'écriture de l'appel à la fonction `print` ne modifie nullement l'espacement affiché :

```
1 print("toto"      ,        42+100)
```

```
2 toto 142
```

— Ligne 1 : lors de l'écriture de l'appel dans le code-source, de nombreux espaces ont été placés entre les deux objets à afficher.

— Ligne 2 : l'espacement à l'affichage reste pourtant d'une espace.

La possibilité d'affichage multiple s'applique aussi à des expressions dépendant de variables :

```
1 x = 42
2 y = 10
3 print(x + y, x - y)
```

```
4 52 32
```

Affichage amélioré

Le programmeur a parfois besoin de rendre plus lisible des résultats à afficher dans un terminal ou une zone d'output. Voici un exemple sommaire d'une telle situation :

afficherLisible.py

```
1 a = 100
2 b = 42
3 print(a + b, a - b)
4 print("L'addition -> ", a + b, "La soustraction ->", a - b)
```

```
5 142 58
6 L'addition ->  142 La soustraction -> 58
```

— Ligne 3 et ligne 5 : affichage peu lisible.

— Ligne 4 et ligne 6 : affichage dont le contenu est plus facilement interprétable.

Affichage et débogage

En particulier, pour vérifier (visuellement) la justesse d'un programme ou pour mieux en comprendre le fonctionnement, il est parfois utile de faire des affichages des variables et des valeurs calculées :

```
a = 100
b = 42
print("a = ", a, ",", "b = ", b)
print("---------------------")
print("a + b  = ", a + b)
print("a - b  = ", a - b)
```

```
a =  100 , b =  42
---------------------
a + b  =  142
a - b  =  58
```

L'action de rechercher les erreurs (les « bugs » ou, en français, les « bogues ») que contient un programme s'appelle le *débogage*. Le placement d'instructions d'affichage dans certaines parties bien choisies d'un code-source en cours de réalisation dans le but de mieux comprendre comment il s'exécute et ce que valent les variables pendant l'exécution du code est une forme très fréquente de débogage (bien que très rudimentaire).

Afficher un message complexe

Comme un texte est une suite de lignes, la fonction `print` permet d'afficher des messages plus élaborés :

```
print("Voici les couleurs")
print()
print("+----------+----------+----------+")
print("|   Rouge  |  Vert    |  Bleu    |")
print("+----------+----------+----------+")
```

qui affiche

```
Voici les couleurs

+----------+----------+----------+
|   Rouge  |  Vert    |  Bleu    |
+----------+----------+----------+
```

Noter, qu'en général, ce type de message est compliqué à construire et à maintenir.

Affichage et forcer un passage à la ligne

Pour séparer deux affichages bien distincts d'un même programme, le programmeur veut parfois placer un ou plusieurs sauts de ligne entre les parties à afficher. Par exemple, supposons qu'il veuille afficher

```
un
deux
trois


1
2
3
```

Il est sans effet de passer des sauts de lignes dans le code-source. Ainsi, le code suivant

```
1 print("un")
2 print("deux")
3 print("trois")
4
5
6 print(1)
7 print(2)
8 print(3)
```

affichera

```
un
deux
trois
1
2
3
```

(noter les sauts de lignes dans le code-source aux lignes 4-5).

Une solution au problème est la suivante :

```
1 print("un")
2 print("deux")
3 print("trois")
4 print()
5 print()
6 print(1)
7 print(2)
8 print(3)
```

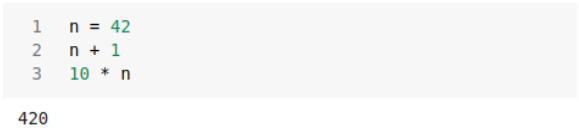
```
9 un
10 deux
11 trois
12
13
14 1
15 2
16 3
```

— Lignes 4-5 : appel à la fonction `print` sans argument (une paire de parenthèses vides).

— Lignes 12-13 : chaque appel `print()` effectue un saut de ligne à l'affichage ce qui « allège » l'affichage final.

Affichage implicite dans Jupyter Notebook

Dans Jupyter Notebook, la valeur d'une expression peut parfois être affichée sans que la fonction `print` ne soit utilisée. Plus précisément, une **expression** placée en dernière ligne d'une cellule de code comme on voit ci-dessous :



provoque l'affichage de la valeur de l'expression. Ainsi, dans le code,

```
n = 42
n + 1
10 * n
```

la ligne 3 provoque l'affichage de 420 et c'est assez logique, cela cadre avec la modalité interactive de l'interface Jupyter Notebook. La ligne `n + 1` ne provoque aucune affichage (car ce n'est pas la dernière ligne).

Une affectation simple n'étant pas une *expression*, elle ne provoque pas d'affichage :

```
n = 42
```

```
# aucun affichage produit
```

L'expression doit être hors-bloc, sinon rien n'est affiché, comme dans l'exemple suivant :

```
if True:
    42
```

```
# aucun affichage produit
```

Si l'expression a pour valeur `None`, rien n'est affiché non plus :

```
[4, 5, 6].append(42)
```

```
# aucun affichage produit
```

Pour forcer un affichage, utiliser la fonction `print` :

```
n = 42
print(n)
10 * n
```

```
42
420
```

La bibliothèque standard

Le langage Python intègre la possibilité, par exemple, de trier une liste d'entiers :

```
1 print(sorted([5,19,14,12,21]))
```

```
2 [5, 12, 14, 19, 21]
```

— Ligne 1 : la liste 5,19,14,12,21 n'est pas triée.

— Ligne 2 : la liste [5, 12, 14, 19, 21] est triée.

À l'inverse de la fonction de tri `sorted`, certaines fonctions, comme la partie entière d'un nombre, ne peuvent être utilisées *directement* par le langage Python mais sont disponibles si on fait appel à ce qu'on appelle la « bibliothèque standard » de Python.

Ainsi, voici comment on peut calculer en Python des parties entières :

```
1 from math import floor
2
3 print(floor(4.21))
4 print(floor(-4.21))
```

```
5 4
6 -5
```

— Ligne 1 : on doit importer la fonction `floor` du « module » `math` de la bibliothèque standard.

— Lignes 3 et 4 : on appelle la fonction de partie entière `floor` importée du module `math`.

La notion de bibliothèque standard

La bibliothèque standard de Python est livrée avec le langage Python mais ne fait pas partie du langage en tant que tel. Elle *étend* les possibilités du langage.

Les fonctionnalités de la bibliothèque standard sont regroupées en entités que l'on appelle des **modules**. Ainsi, les fonctionnalités mathématiques sont disponibles dans le module `math`.

Pour accéder depuis du code Python à une fonctionnalité d'un module de la bibliothèque standard, il faut importer cette fonctionnalité dans le code Python par une instruction d'importation (cf. ligne 1 ci-dessus).

La bibliothèque standard contient beaucoup d'autres modules que le module `math`. Voici quelques exemples de modules disponibles :

`Module`	Fonctionnalité
`math`	Fonctions mathématiques
`random`	Manipuler des nombres aléatoires
`decimal`	Résultats exacts avec les nombres décimaux
`datetime`	Gestion des dates, des durées
`os.path, shutil`	Manipuler le système de fichiers
`Tkinter`	Interfaces graphiques

Un module de la bibliothèque standard est dit *module standard.*

De nombreuses fonctionnalités de la bibliothèque standard concernent des questions spécialisées et/ou techniques et sont réservés à des programmeurs de niveau avancé voire professionnel.

Importer une fonctionnalité standard

Soit le code suivant qui importe la fonction `floor` du module standard math.

```
1 from math import floor
2
3 print(floor(4.21))
```

```
4 4
```

Ligne 1 : on n'importe qu'une seule fonctionnalité, la fonction `floor`.

Pour qu'un code Python puisse utiliser une fonctionnalité, par exemple la fonction `floor`, ce code doit importer (avec une instruction `import`) la fonctionnalité du module ad hoc.

Si on n'importait rien, cela produirait une erreur :

```
1 z = floor(421.2020)
2 print(z)
```

```
3 ----> 1 z = floor(421.2020)
4       2 print(z)
5
6 NameError: name 'floor' is not defined
```

— Ligne 1 : Faute d'importation, le nom `floor` n'est pas reconnu.

On a importé une *fonction* du module `math` mais un module peut, en général, proposer d'autres types d'objets à importer, par exemple des constantes. Ainsi, le module `math` dispose de la constante mathématique π :

```
1 from math import pi
2
3 print(pi)
```

```
4 3.141592653589793
```

Le Hasard en Python

Un nombre *aléatoire* est un nombre *tiré au hasard*. Plus généralement, l'adjectif *aléatoire* se réfère à ce qui est le résultat du *hasard*.

Le module standard `random` gère l'aléatoire en Python.

Le code ci-dessous génère un entier aléatoire entre 1 et 49, comme pour un tirage d'une boule au loto :

```
1 from random import randint
2
3 a = randint(1,49)
4 print(a)
```

```
5 42
```

— Ligne 1 : la bibliothèque standard Python dispose d'un module de création d'objets aléatoires. Ce module s'appelle `random`. Le module `random` « possède » une fonction `randint`. La ligne 1 permet au programme de faire appel à la fonction `randint`.

— Ligne 3 : le tirage est effectué grâce à la fonction `randint`. La fonction `randint` renvoie ici un nombre (aléatoire) entre `1` et `49`, bornes comprises.

— Lignes 4 et 5 : le résultat du tirage aléatoire.

Fonction `randrange`

Dans l'exemple ci-dessous, on effectue un tirage aléatoire de 6 entiers `n` vérifiant $1 \leq n < 50$ (comme au loto en France) :

```
1 from random import randrange
2
3 print(randrange(1, 50))
4 print(randrange(1, 50))
5 print(randrange(1, 50))
6 print(randrange(1, 50))
7 print(randrange(1, 50))
8 print(randrange(1, 50))
```

```
9  14
10 15
11 47
12 45
13 38
14 12
```

La fonction `randrange` est une fonction du module `random`. L'usage de la fonction `randrange` nécessite donc une instruction d'importation, cf. ligne 1 ci-dessus.

Un appel `randrange(a, b)` renvoie un entier aléatoire `n` tel que $a \leq n < b$ où `a` et `b` ont des valeurs entières (on notera que la borne en `b` est *stricte*)

Pour effectuer un tirage par pile ou face, on utilisera `randrange(2)`.

De même, pour simuler un dé, on utilisera `randrange(6)` :

```
1 from random import randrange
2
3 # Tirage de dé à 6 faces
4 de_1_6 = 1+randrange(6)
5 print(de_1_6)
```

```
6 3
```

Fonction `randint`

La fonction `randint` fournie par le module standard `random` génère un entier aléatoire entre deux bornes entières `a` et `b`, bornes incluses.

```
1 from random import randint
2
3 print(randint(1, 49))
```

```
40
```

Un appel `randint(a, b)` se fait toujours avec *deux* arguments, entiers et renvoie un entier aléatoire `n` tel que $a \leq n \leq b$.

Pour effectuer un tirage par pile ou face, on utilisera `randint(1, 2)`.

De même, pour simuler un dé, on utilisera `randint(1,6)` :

```
from random import randint

# Tirage de dé à 6 faces
de_1_6 = randint(1, 6)
print(de_1_6)
```

```
3
```

EXERCICES

Demi-carré en mode texte

Ecrire un code Python qui affiche exactement ceci (les caractères sont des lettres X majuscules) :

```
X
XX
XXX
XXXX
```

Bien sûr, et même si vous avez déjà rencontré cela, aucune boucle `for` n'est à utiliser.

Echanger deux variables

Cet exercice sera plutôt à faire en 2e lecture.

Étant donné deux variables x et y, on veut permuter leurs valeurs. Par exemple si au départ $x = 81$ et $y = 31$ alors, après échange, on doit avoir : $x = 31$ et $y = 81$. On pourra compléter le code ci-dessous :

```
x=81
y=31
print(x,y)

# VOTRE
# CODE
# ICI

print(x,y)
```

Bien sûr votre code doit fonctionner correctement si on donne d'autres valeurs à `x` et `y` aux lignes 1 et 2.

On fera comme lorsqu'on veut échanger deux étiquettes A et B placées sur des objets : on utilise une étiquette de nommage temporaire.

La finance aime Python

En utilisant des puissances de 10, une variable `milliard` et une variable `million`, traduire l'opération suivante :

200 milliards de millions augmenté de 2020 millions et privé de 925 milliards.

Afficher le résultat (on trouvera 199999077020000000).

[De fait, Python est assez utilisé en finances, cf. par exemple Python in finance ou encore Best Python Librariess for Finance]

Factorielle de 10

Calculer la valeur de factorielle 10, c'est-à-dire du produit p des 10 premiers entiers en commençant avec 1 :

$$p = 1 \times 2 \times \cdots \times 10$$

On devra trouver 3628800. Bien sûr, aucune instruction de boucle (`for` ou `while`) n'est à utiliser.

Volume du ballon de rugby

Consulter l'article de wikipedia donnant le volume d'un ellipsoïde de révolution.

On donne la hauteur H d'un ballon de rugby, sa largeur L et on demande de calculer son volume V. On posera `pi = 3.14` et on calculera le volume en litres d'un ballon standard ($H = 29$cm et $L = 19$cm) et on trouvera environ `5,5` litres.

Rappel. 1 litre vaut $1\texttt{dm}^3$.

Formule de Héron

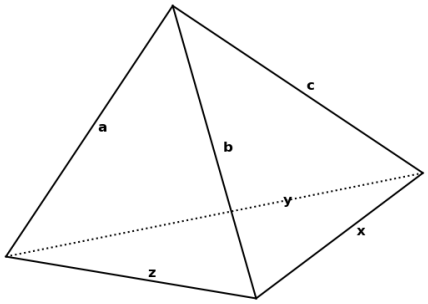
La formule du Héron donne l'aire $\mathcal{A}$ d'un triangle `ABC` connaissant la longueur de ses trois côtés `a` $=$ `BC`, `b` $=$ `CA` et `c` $=$ `AB` :

$$\mathcal{A} = \sqrt{\texttt{p}(\texttt{p}-\texttt{a})(\texttt{p}-\texttt{b})(\texttt{p}-\texttt{c})} \text{ où } \texttt{p} = \tfrac{\texttt{a}+\texttt{b}+\texttt{c}}{2} \text{ est le demi-périmètre du triangle.}$$

Ecrire cette formule en utilisant la variable auxiliaire accompagnant la formule. Appliquer à un triangle de côtés 3, 4, 5 (on trouvera une aire de 6).

Volume d'un tétraèdre

On considère un tétraèdre



et on suppose que les côtés adjacents à un sommet donné sont de longueurs a, b et c. Les longueurs des côtés opposés aux côtés de longueurs a, b et c sont respectivement x, y et z.

On admet que le volume V du tétraèdre est donné par la formule

$$V = \frac{1}{12}\sqrt{4\,a^2b^2c^2 - X^2a^2 - Y^2b^2 - Z^2c^2 + XYZ}$$

où

$$X = b^2 + c^2 - x^2, \quad Y = a^2 + c^2 - y^2, \quad Z = a^2 + b^2 - z^2$$

Ecrire en Python une formule qui donne le volume du tétraèdre. Appliquer au tétraèdre ABCD de dimensions : AB=8, BC=5, CA=6, DA=4, DB=7 et DC=3. On trouvera un volume de `5.59` à 1/1000 près.

La formule utilisée vient de wikipedia, la formule dont le dénominateur est 12.

Volume d'un sac-poubelle

Un sac-poubelle prêt à l'emploi est assimilé à une superposition de deux feuilles de plastique rectangulaires de dimensions d et D avec $d \leq D$. On peut approcher le volume maximal que peut contenir le sac poubelle par la formule suivante :

$$V = d^3 \left[\frac{D}{\pi d} - 0,142 \times \left(1 - \frac{1}{10^{\frac{D}{d}}} \right) \right]$$

d'après la formule trouvée dans Paper bag problem.

Un sac du commerce a pour dimensions 8 dm et 8,5 dm. En appliquant la formule ci-dessus, calculer le volume du sac, en arrondissant aux 10 litres les plus proches (on doit trouver un résultat pas trop éloigné de 100 litres). Pour arrondir un nombre `x` à $1/10^{\texttt{k}}$ près, où `k` est un entier de signe quelconque, on utilisera `round(x, k)`.

Fromage sur la pizza

Une pizza de diamètre D possède une couverture de fromage que l'on assimile à un disque de même centre que la pizza. Le disque de fromage s'arrête à une distance C du bord.

On vous demande de calculer le pourcentage P de fromage par rapport à la pizza. On arrondira le pourcentage à l'entier le plus proche avec la fonction `round` :

`round(3.8)=4` ou `round(2.3)=2`.

Par exemple, si $D = 40$ et $C = 3$ alors on trouvera que $P = 72\%$.

L'aire d'un disque de diamètre d est $\pi d^2/4$ mais en fait aucune formule utilisant $\pi = 3.14$ n'est à connaître. Il vous suffit juste de savoir que si on multiple le diamètre d'une pizza par k, alors sa surface est multipliée par k^2.

Cet exercice est inspiré d'un exercice de Kattis qui est un site de concours de programmation.

Valeur approchée de $e = 2,718...$

Lorsque l'entier n est grand, l'expression $\left(1 + \frac{1}{n}\right)^n$ a une valeur proche de la constante mathématique $e = 2,718...$ qui est la base des logarithmes népériens. Donner la valeur approchée fournie par $n = 100000$.

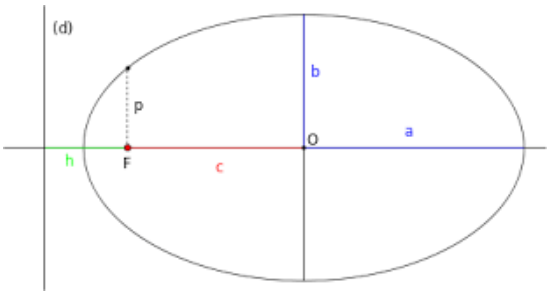
Calculs

Faire effectuer par Python les calculs suivants :

$$a = 10^{2\times 3} \quad b = \frac{666}{74 \times 9} \quad c = 2 \times 3,14159 + 2,71828 + 1,4142 \times 5 \quad d = \frac{1}{1000^{\frac{2}{3}}}$$

Périmètre d'une ellipse

Il n'existe pas de formule donnant une valeur exacte du périmètre d'une ellipse d'axes de longueurs a et b avec $b \leq a$.



Toutefois, il existe plusieurs formules donnant une valeur approchée.
Par exemple, voici l'approximation de Ramanujan :

$$P = \pi\left[3\,a + 3\,b - \sqrt{3\,a^2 + 10\,ab + 3\,b^2}\right]$$

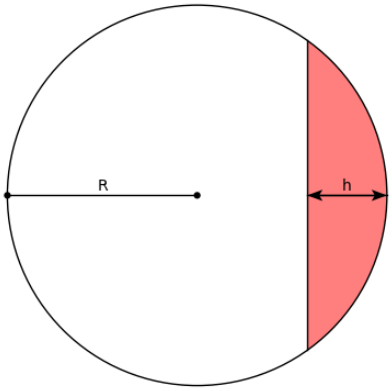
On dispose aussi de cette approximation de Padé :

$$P = \pi(a+b)\frac{135168 - 85760h - 5568h^2 + 3867h^3}{135168 - 119552h + 22208h^2 - 345h^2} \text{ avec } h = \left(\frac{a-b}{a+b}\right)^2$$

Comparer ces approximations pour $a = 2$ et $b = 1$, une approximation très précise étant `P` $=$ `9.688448220547675`.

Aire d'un segment circulaire

On découpe sur un disque de rayon `R` donné une tranche de largeur `h` $\leq$ `R` donnée :



En appliquant la formule donnée sur Wikipedia, calculer l'aire `A` de la zone découpée en fonction de `R` et `h`. On importera du module math la fonction `acos` qui correspond à la fonction `arccos`. Vérifier en particulier que si `R` $=$ `1` et `h` $=$ `2/3` alors `A` $\simeq$ `0.92`.

Combien d'oranges pour un jus ?

On veut remplir un verre d'un volume `V` cl de jus d'orange frais sans pulpe. Le verre sera rempli dans une proportion `p`, par exemple `p=0.75` pour signifier que le verre sera rempli aux 3/4 de son volume. Une orange est assimilée à une sphère de diamètre de `d` cm. Une fois pressée, en retirant

la pulpe et la peau, elle restitue du jus dans une proportion qu'on appelera `taux`, par exemple, `taux = 0.7` signifie que l'orange délivre 70% de son volume en jus. On utilisera `pi = 3.14` et on rappelle que la formule du volume d'une sphère de rayon `R` est

$$V = \frac{4}{3}\pi R^3.$$

En utilisant des variables `V`, `p`, `d` et `taux`, trouver une expression donnant le nombre `N` d'oranges nécessaires pour remplir le verre.

On utilisera que la partie entière d'un flottant positif `z` est donnée par `int(z)`, par exemple, `int(42.81)` vaut `42`.

Application numérique : `V`=50, `p`=90%, `d`=7, `taux`=70% ce qui doit nécessiter `N = 4` oranges.

Vulnerant omnes, ultimat necat

1. Combien y-a-t-il de secondes dans un siècle ? (utiliser des variables `une_heure`, `j`, `an` et `siecle` et supposer qu'une année est constituée de 365 jours 1/4)
2. À partir de quel âge un individu a vécu au moins 1 milliard de secondes ? (réponse : entre 31 et 32 ans)

Date fournie par le GPS

La date initiale à partir de laquelle est mesuré le temps par les GPS est le 6 janvier 1980. La date courante lue par le GPS en utilisation est déterminée à l'aide d'un nombre de **semaines** codé sur **13 bits** compté depuis la date initiale. Autrement dit, le GPS ne peut mesurer que 2^{13} semaines, ensuite il revient à zéro. Indiquer l'année à laquelle le GPS va se réinitiliser. On utilisera qu'une année comporte 365,256 jours.

Pour information, initialement, le nombre de semaines était codé sur 10 bits, donc 1024 semaines, soit un peu moins de 20 ans, ce qui a entraîné deux réinitilisations des GPS (en 1999 et en avril 2019).

Priorités des opérateurs

L'expression `2 + 3 * 5` pourrait a priori s'interpréter `(2 + 3) * 5` ou `2 + (3 * 5)`. Demandons à Python la valeur des trois expressions :

```
x=2 + 3 * 5
y=(2 + 3) * 5
z=2 + (3 * 5)

print (x==y, x==z)
```

```
False True
```

Donc la bonne interprétation avec les parenthèses est `2 + (3 * 5)`.

Faire de même pour les cinq expressions suivantes autrement dit placer toutes les parenthèses utiles dans l'expression pour obtenir une expression non ambiguë et de même valeur :

```
- 1 ** 2
10 ** 2 ** 3
7 * 29 / 7
2 / 1 * 2
```

```
60 / 6 / 2 / 5
```

Puissances sans parenthèses

Si a est un nombre, on considère les deux expressions

$$b = a^{(a^a)} \quad c = (a^a)^a$$

c'est-à-dire que

- le nombre b est a élevé à la puissance a^a, par exemple, si $a = 2$ alors $b = 2^4 = 16$
- le nombre c est a^a élevé à la puissance a, par exemple, si $a = 2$ alors $c = 4^2 = 16$

Est-il possible sous Python de calculer l'expression b ci-dessus **sans** utiliser de parenthèses ?
Même question pour c ?
Calculer b et de c pour les valeurs 2, 3, etc jusqu'à ce que le temps de calcul devienne prohibitif.

Formule

On connaît la formule mathématique suivante $1 + \texttt{x} + \texttt{x}^2 + \texttt{x}^3 = \dfrac{\texttt{x}^4 - 1}{\texttt{x} - 1}$.

1. Introduire une variable `x`, par exemple `x = 42` et traduire les deux expressions (celle de gauche et celle de droite) de la formule ci-dessus par du code Python avec deux variables `gauche` et `droite`. ATTENTION aux parenthèses !
   Afficher un booléen qui indique si les valeurs de `gauche` et `droite` sont égales.
2. Vérifiez l'exactitude de cette formule pour `x = 208066`. Si la formule n'est pas exacte, modifiez votre code en observant que le quotient exact au membre de droite doit être un entier puisque le membre de gauche est un entier.

Diviseur, multiple

1. Vérifier que 29 est un diviseur de 2030.
2. Montrer que 192642212037519549 et 138633362400520509 ont même reste dans la division entière par 2030.

Multiples de 421

Ecrire la liste des 5 premiers multiples de 421. On donnera deux codes :

- un code affichant les nombres les uns en-dessous des autres,
- un code affichant les nombres les uns à côtés des autres.

Nombre de tours

Utiliser des variables pour résoudre le problème suivant :
un coureur fait un tour de circuit en c minutes. Quel est le nombre x de tours **complets** qu'il effectuera en n minutes ? Appliquer avec $c = 7$, $n = 240$.

Plus petit multiple strictement supérieur

On donne un entier $d > 0$ et un entier a et on demande de trouver le plus petit multiple m de d tel que $m > a$. Par exemple, si $d = 10$ alors :

— si $a = 42$ alors $m = 50$
— si $a = 2020$ alors $m = 2030$.

Plus petit multiple supérieur ou égal

On donne un entier $d > 0$ et un entier a et on demande de trouver le plus petit multiple m de d tel que $m \geq a$. Par exemple, si $d = 10$ alors :

— si $a = 42$ alors $m = 50$
— si $a = 2020$ alors $m = 2020$.

Nombre d'entiers impairs entre deux entiers donnés (formule)

On donne deux entiers a et b avec $a \leq b$. On demande de déterminer le nombre d'entiers impairs n entre a et b, c'est-à-dire, vérifiant $a \leq n \leq b$. On pourra utiliser que si A et B sont des entiers tels que $A \leq B$ alors le nombre d'entiers entre A et B, au sens large, est $B - A + 1$. Le résultat s'appuyera sur une formule (à trouver) utilisant des divisions entières par 2.

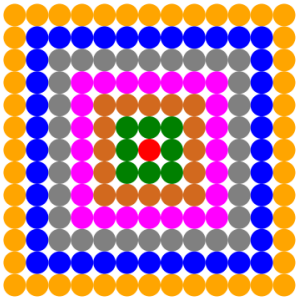
Voici quelques exemples de comportement :

```
Entre 42 et 50 : 4
Entre 42 et 51 : 5
Entre 43 et 50 : 4
Entre 43 et 51 : 5
Entre 42 et 42 : 0
Entre 42 et 43 : 1
Entre 43 et 43 : 1
Entre 43 et 44 : 1
Entre 123456789 et 987654321 : 432098767
```

Nombre de carrés dans une figure (formule)

Cet exercice n'est pas un exercice de dessin

On considère une figure carrée composée de disques alignés, comme ci-dessous :



Le côté du carré est formé de n disques. La figure est formée d'une succession de p carrés concentriques, tracés ici avec des couleurs différentes pour qu'ils soient bien identifiables. Dans le modèle ci-dessus, on a $n = 13$ et $p = 7$.

On demande d'écrire une formule exprimant p en fonction de n (réaliser plusieurs figures pour comprendre le lien entre n et p).

Nombre de multiples de d entre deux valeurs données

On donne deux entiers `a` et `b` avec `a` $\leq$ `b` et un entier `d` $>$ `0`. Le nombre `N` de multiples de `d` tels que `a` $\leq$ `m` $\leq$ `b` est donné par la formule suivante :

$$\texttt{N} = \left\lfloor \frac{\texttt{b}}{\texttt{d}} + \frac{-\texttt{a}}{\texttt{d}} + 1 \right\rfloor$$

où $\lfloor$`x`$\rfloor$ désigne la partie entière de `x` (en Python, c'est la fonction `floor` du module math).

Ecrire un code qui partant de `a`, `b` et `d` calcule le nombre `N` ci-dessus. Par exemple, si `d` $=$ `2030`, si `a` vaut 100 milliards et si `b` vaut 1000 milliards alors on trouvera que `N` $=$ `443349754`.

Variable définie suivant la parité

Soit un entier `n` référençant une variable entière. A l'aide de la division entière, définir une variable `N` valant `n-1` si `n` est impair et valant `n` sinon.

Répétitions sans boucle

Cet exercice réclame de l'astuce !

1. Soit l'entier `N` dont l'écriture en base 10 est formée du chiffre `9` répété 2020 fois : Afficher la valeur `N` en base 10. *Indication* : que se passe-t-il si on ajoute 1 à `N` ?
2. Soit l'entier `N` dont l'écriture en base 2 est formé du chiffre `1` répété 2020 fois. Afficher la valeur `N` en base 10.

Diviser pour carreler

Une pièce mesure a cm de long et b cm de large. On veut carreler la pièce avec des carreaux (carrés) de côté c cm. Combien doit-on acheter de carreaux pour être certain de pouvoir recouvrir complètement la pièce (ignorer l'épaisseur du joint) ?

Par exemple, si `a = 500`, `b = 430` et `c = 50`, le nombre de carreaux à acheter est 90. En effet, il y a besoin d'exactement 10 carreaux dans la longueur (`10` $\times$ `50` $=$ `a`). Pour recouvrir complètement la largeur, 8 carreaux ne suffisent pas (cela ferait une largeur de `8` $\times$ `50` $=$ `400` cm au lieu des 430 cm) et il en faut donc 9 dans la largeur. On a donc besoin de `10` $\times$ `9` $=$ `90` carreaux et il y aura une « chute » non réutilisable de 30 cm de large.

Formule pour arrondir une note au quart de point

1. La fonction standard `round(x)` arrondit un nombre entier ou flottant à l'entier le plus proche. La fonction `int(x)` supprime la partie décimale d'un flottant `x`, par exemple `int(3.14)` vaut 3. Tester ces deux fonctions sur les exemples suivants :

```
22.64
47.1778
42
42.5
```

2. On cherche à arrondir une note au quart de point. Par exemple, une note de 8.33 s'arrondit en 8.25 et une note de 15.66 s'arrondit en 15.75. Une note qui tombe pile entre deux quarts de point est arrondie au quart supérieur. Par exemple, la note de 19.875 est arrondie à 20.

Observation 1 : pour arrondir une note au quart de point, il suffit juste de savoir arrondir la partie décimale de la note (qui est un nombre entre 0 et 1 donc). Par exemple, comme 0.33 s'arrondit 0.25, la note de 8.33 s'arrondit en 8.25.
Observation 2 : pour arrondir au quart de point une note `x` comprise entre 0 et 1, il suffit de

— diviser cette note par 0.25,

— prendre l'entier le plus proche,

— remultiplier le résultat par 0.25.

Par exemple, pour 0.66, on obtient successivement 2.64 puis 3 puis 0.75.
En utilisant ces **deux** observations, écrire une formule utilisant les fonctions `round` et `int` qui partant d'une note `x`, crée une variable `arrondi` qui est l'arrondi de la note `x` au quart de point. Voici quelques exemples de comportement :

```
9 → 9.0
10.5 → 10.5
11.75 → 11.75
12.25 → 12.25
10.375 → 10.5
8.33 → 8.25
5.66 → 5.75
10.14 → 10.25
10.13 → 10.25
19.875 → 20.0
```

Plus grand entier pair tel que ...

On donne un entier $n \geq 0$. On demande de trouver le plus grand entier pair N tel que $N \leq n/2$.
Par exemple,

— si $n = 42$ alors $n/2 = 21$ et donc $N = 20$

— si $n = 2019$ alors $n/2 = 1009.5$ et donc $N = 1008$.

Le calcul de N devrait pouvoir tenir en une ligne.

Deux mois plus tôt

On numérote les mois `m` de 1 à 12 et on recherche le numéro `M` du mois quand on va deux mois en arrière dans le calendrier. Par exemple, si `m = 9` (septembre), deux mois plus tôt c'est le mois de juillet, (donc `M = 7`). Attention, si `m = 1` (janvier), deux mois plutôt c'est `M = 11` (novembre) et si `m = 2`, (février donc) deux mois plutôt c'est `M = 12` (décembre).

On demande de trouver une **formule** apparaissant sous la forme d'une **expression**, utilisant `m` (entre 1 et 12) et des **opérateurs arithmétiques** (+, -, etc) et en particulier **l'opérateur modulo** et qui calcule `M` expliqué ci-dessus et entre (1 et 12).

Il n'est pas attendu d'utiliser la valeur entière des booléens (ce qui n'a pas été vu dans ce cours d'ailleurs). On pourra essayer de traiter un **décalage quelconque, avant ou après la date** (par exemple, 6 mois en arrière ou 3 mois plus tard).

Position après des allers et retours

Cet exercice est à caractère algorithmique et nécessitera un certain temps de recherche (sur feuille) et une analyse préalable. Il faudra utiliser une division entière.

On se donne deux entiers a et b avec $a \leq b$ ainsi qu'un entier $\texttt{N} \geq 0$. Un puce, placée initialement à l'entier a, se déplace pendant N secondes. A l'issue des N secondes elle s'arrête à la position où elle se trouve. La puce se déplace toujours d'un entier à un de ses voisins et le déplacement se fait en exactement une seconde. Quand elle atteint b et que le temps n'est pas écoulé, elle repart vers a et, de même, quand elle atteint a elle repart vers b. On demande de calculer la position c de la puce à l'instant N.

Par exemple, si $a = 4$, $b = 7$ alors si $N = 10$ on aura $\texttt{c} = 6$ et si $N = 2040$ on aura $\texttt{c} = 4$.

On pourra remarquer que la puce passe en a et en direction de b de manière périodique.

Affichages empilés, côte-à-côte

On vous donne un entier x, par exemple $x = 2020$. Ecrire le code d'un programme qui réalise l'affichage de

$$x,\ x+1 \text{ et } x+2$$

avec un seul nombre par ligne puis qui saute une ligne et affiche en ligne de x, $x+1$ et $x+2$. L'ensemble doit être entouré d'un séparateur formé de 25 tirets. Avec l'exemple $x = 2020$, on obtient les 7 lignes suivantes :

```
-------------------------
2020
2021
2022

2020 2021 2022
-------------------------
```

Reproduire un affichage

Soit une variable x, représentant un entier. On cherche à fournir un affichage du calcul de $10x+1$. Si, par exemple, $x = 42$, on veut que l'affichage ait EXACTEMENT la forme suivante

```
x = 42 => 10 * x + 1 = 421
```

ou encore (si $x = 3$) :

```
x = 3 => 10 * x + 1 = 31
```

Écrire un code qui produise ce type d'affichage.

Copier-coller pour Fibonacci

Voici les 12 premiers termes de la suite de Fibonacci :

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144$$

Plus généralement, les deux premiers termes de cette suite sont 1 et encore 1 et, chaque terme de la suite à partir du troisième s'obtient en faisant la somme des deux précédents. Par exemple, ci-dessus, $89 = 34 + 55$.

On cherche à calculer le `n`-ème terme de la suite de Fibonacci. Comme on ne dispose pas encore de la structure de contrôle Python permettant de le faire de manière appropriée (la boucle `for`), on va utiliser une astuce pas très élégante, des copier-coller. On calculera le 200^{e} terme. On trouvera que c'est un entier de 42 chiffres et qui commence par 2805. Pour accélérer la saisie, on fera des copier-coller de copier-coller.

On utilisera quatre variables :

Variable	Initialisation	Rôle
`n`	`n` $= 2$	Rang du terme courant
`x`	`x` $= 1$	Avant-dernière valeur de la suite
`y`	`y` $= 1$	Dernière valeur calculée
`z`	Aucune	`z` $=$ `x` $+$ `y` : valeur suivante

Indications

La répétition sera assurée par des copier-coller. Supposons stockés

— l'avant-dernier terme, disons `x`, de rang `n-1`

— le dernier terme, disons `y`, de rang `n`

Alors le prochain terme sera `z=x+y` et il faut penser à remettre à jour les variables pour l'étape suivante.

Factorielles

On rappelle que la notation $k!$ désigne la factorielle de l'entier positif k c'est-à-dire le nombre

$$1 \times 2 \times \cdots \times k.$$

Par exemple $5! = 1 \times 2 \times 3 \times 4 \times 5 = 6 \times 20 = 120$.

1. Faites calculer $20!$ par Python en écrivant explicitement le produit des 20 entiers.
2. L'instruction suivante :

```
from math import *
```

placée dans un code Python, est une instruction d'importation. Elle permet d'importer dans votre code toutes les fonctionnalités du module `math`. Donc une fois l'instruction validée, votre code peut utiliser librement les fonctions mathématiques comme cosinus, sinus, etc, le nombre π, le nombre $\mathrm{e} = 2,71828\ldots$ (la base du logarithme naturel) ou encore la fonction factorielle.

Après avoir placé dans votre code la ligne ci-dessus, faites calculer directement

— factorielle de 20

— factorielle de 2020

— $\cos \dfrac{\pi}{4}$

— $\dfrac{1}{e}$

Utilisez que :

— la fonction factorielle s'appelle `factorial`

— la fonction cosinus s'appelle `cos`,

— la constante π s'appelle `pi`,

— la constante e s'appelle `e`

(3) La formule de Stirling donne une approximation de $n!$ pour de grandes valeurs de n. Cette formule énonce que $n! \approx \text{sn}$ où :

$$\text{sn} = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Pour $n = 20$, $n = 170$ et $n = 2020$ calculer

— la valeur exacte de $n!$ en utilisant la fonction `factorial`,

— la valeur approchée `sn` donnée par la formule de Stirling.

— l'erreur relative `err` donnée en pourcentage : $\text{err} = 100\dfrac{n! - \text{sn}}{n!}$

Pour $n = 2020$, vous aurez une erreur. Essayez de comprendre cette erreur.

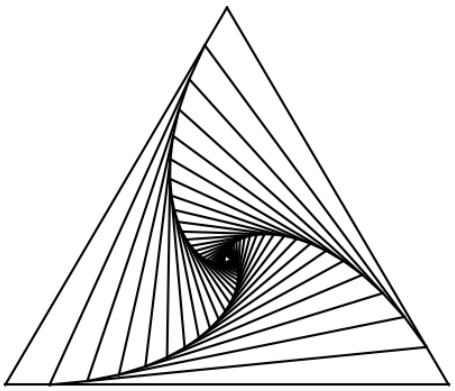
Chapitre II

Dessiner avec Matplotlib

1 Cours

Utilisation de Matplotlib dans ce cours

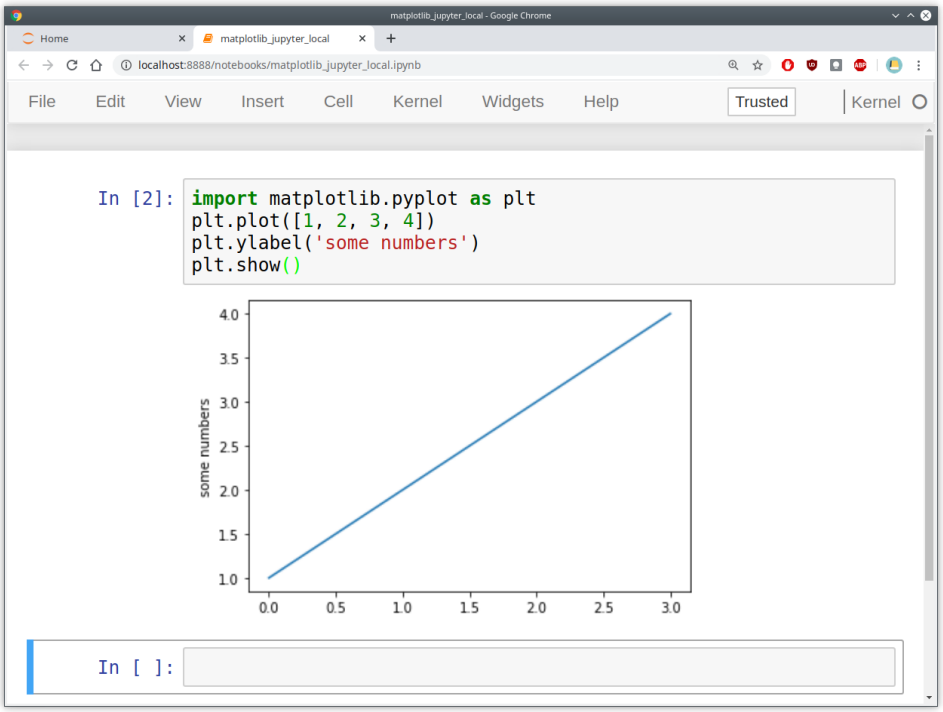
Matplotlib est une extension Python destinée à réaliser des visualisations de données statistiques. Toutefois, elle offre des primitives de dessin (lignes, ellipses, texte, etc) qui peuvent servir à illustrer des notions de programmation Python, pour générer des diagrammes, figures, motifs divers, typiquement ce genre de dessin :



La plupart des exercices de ce cours ne nécessitent que de savoir tracer des segments et des disques.

Environnements d'utilisation

Un environnement courant d'utilisation de Matplotlib est une feuille Jupyter Notebook :



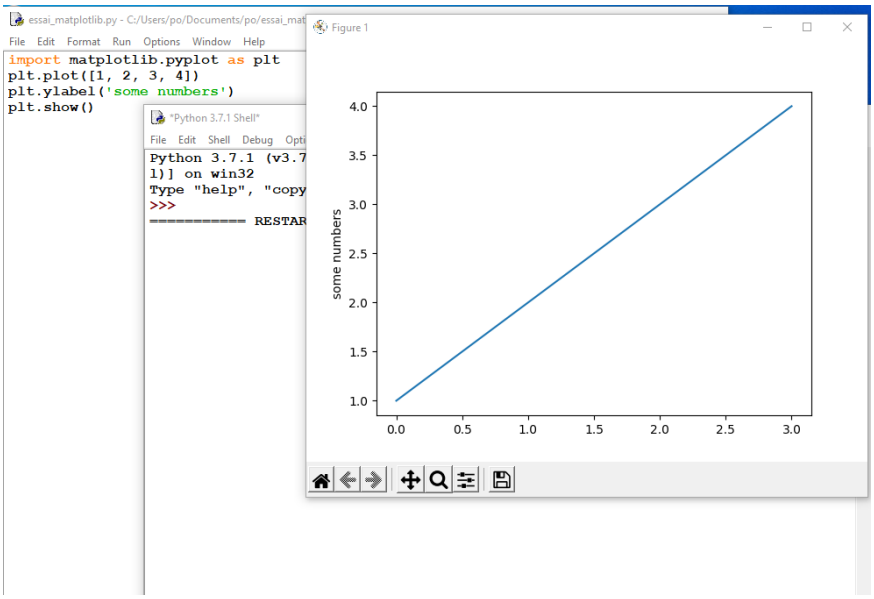
Ces feuilles peuvent s'utiliser :

— soit **en ligne**, par exemple sur Google Colab ou Jupyter.org

— soit **en local** (suite à une installation sur la machine que vous utilisez, soit réalisée par vous-même soit par un administrateur).

En local, on peut accéder à des feuilles Jupyter Notebook soit parce qu'on a installé directement l'environnement Jupyter ou encore on a installé la suite Anaconda.

Toujours en local, mais sans Jupyter Notebook, après installation de Matplotlib, on peut invoquer Matplotlib depuis un environnement classique d'utilisation de Python (comme Spyder, IDLE, VSCode, Pycharm, un éditeur de texte, etc). Les graphiques produits par Matplotlib sont alors visibles dans un logiciel de visualisation qui dépend du système utilisé, par exemple sous Windows, par défaut, c'est une fenêtre Tkinter :



Tout code de dessin dans ce cours importera Matplotlib sous la forme suivante :

```
import matplotlib.pyplot as plt
```

Installation de Matplotlib

Si on veut utiliser Matplotlib sur un site distant comme Google Colab alors Matplotlib est déjà installée.

Pour les 3 systèmes d'exploitation majoritaires, si vous avez déjà installé la suite Anaconda, alors le module Matplotlib est disponible et directement utilisable, dans leurs feuilles Jupyter Notebook ou dans la version de Spyder qu'Anaconda propose.

Pour installer sous Windows 10 :

— taper `cmd` dans Cortana

— ouvrir la ligne de commande proposée

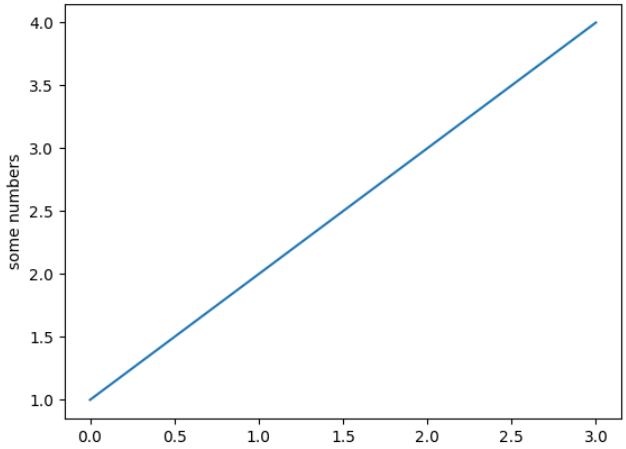
— y taper la commande suivante ci-dessous :

```
pip install matplotlib
```

Une fois la commande exécutée, l'installation de Matplotlib est effective (pas besoin de redémarrer). Vous pouvez la tester en exécutant le code suivant dans un éditeur de texte (par exemple, sous Windows, choisir, `Démarrer > Python > IDLE`) :

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```

ce qui devrait afficher la figure suivante :



Sous Linux, c'est analogue, taper dans un terminal la commande suivante :

```
pip install matplotlib
```

ce qui installera Matplotlib sur votre système. Vous pouvez tester avec le même code que ci-dessus.

D'après la documentation de Matplotlib, sous Linux, il est possible d'installer Matplotlib via votre système de paquets (non testé).

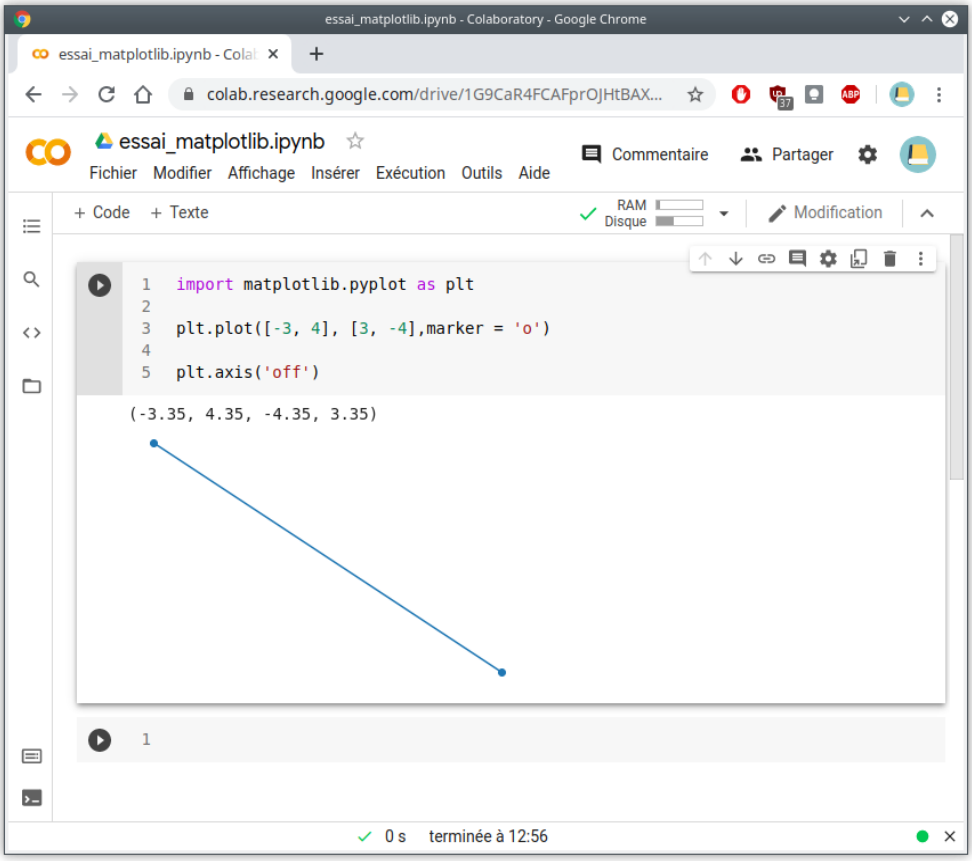
Utilisation de Matplotlib sur Google Colab

Google Colab est un service gratuit mis à la disposition d'un utilisateur ayant un compte Google et permettant de programmer dans des feuilles Jupyter Notebook et exécutées sur l'infrastructure de Google.

Le module Matplotlib est immédiatement disponible sans même que vous ayiez besoin de l'installer. Voici un code simplissime de dessin, à exécuter dans une cellule :

```
import matplotlib.pyplot as plt

plt.plot([-3, 4], [3, -4],marker = 'o')

plt.axis('off')
```

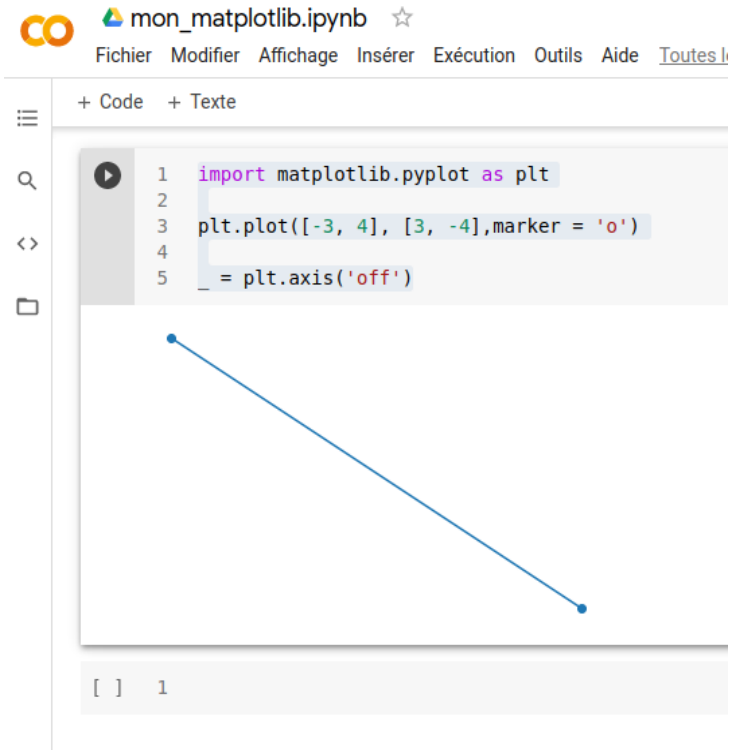
Et voici une vue sur l'interface :



On observera qu'avant de montrer le dessin, la sortie affiche 4 dimensions. Pour les cacher, il suffit de récrire le code comme suit :

```
import matplotlib.pyplot as plt

plt.plot([-3, 4], [3, -4],marker = 'o')

_ = plt.axis('off')
```

qui produit uniquement la figure :



Instruction de visualisation

Dans les feuilles Jupyter Notebook, et en particulier sur Google Colab, il n'est pas nécessaire de placer une instruction d'affichage pour que le résultat graphique soit visible.

Par exemple, le code suivant

```
import matplotlib.pyplot as plt

plt.plot([-3, 4], [3, -4],marker = 'o')

plt.axis('off')
```

affichera la figure demandée. Il n'est toutefois pas pénalisant d'écrire l'instruction d'affichage `plt.show()`, le programme réagira de la même façon que sans l'instruction. Sur ce point, l'éditeur Spyder a le même comportement.

En revanche, le même programme écrit dans d'autres interfaces (comme IDLE, VSCode, etc) ne montrera pas de sortie graphique. Pour l'obtenir, il **faut** écrire une instruction `show`, comme montré dans cet exemple, à la dernière ligne :

```
import matplotlib.pyplot as plt

plt.plot([-3, 4], [3, -4],marker = 'o')

plt.axis('off')
plt.show()
```

Système de repérage

Dans ce cours, Matplotlib est utilisé pour dessiner des motifs, figures géométriques, etc et qui, pour être compris, ne nécessitent pas la présence d'axes donnant les abscisses et ordonnées. Toutefois, pour **construire** ces objets, vous **devrez** utiliser le système d'axes rectangulaires fourni par Matplotlib. Dans la plupart des sorties graphiques de ce cours, le système de repérage sera rendu invisible grâce à la ligne de code :

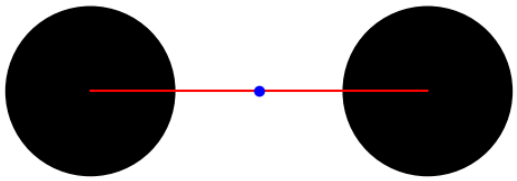
```
plt.axis('off')
```

Vous pouvez rétablir ce système d'axes en désactivant avec un commentaire Python la ligne ci-dessus, ce qui donne :

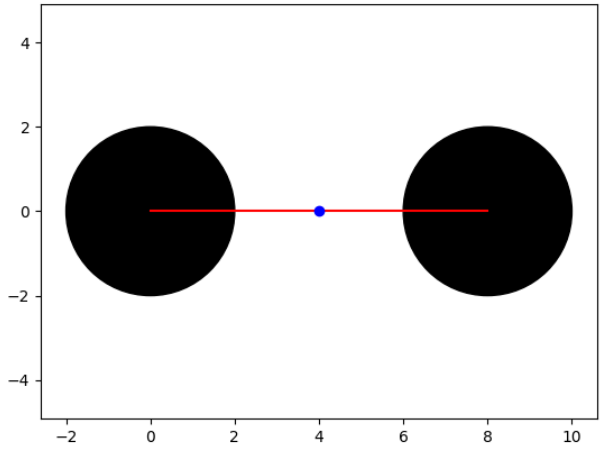
```
# plt.axis('off')
```

ou encore en ne l'écrivant pas.

Voici un dessin rendu sans repère :



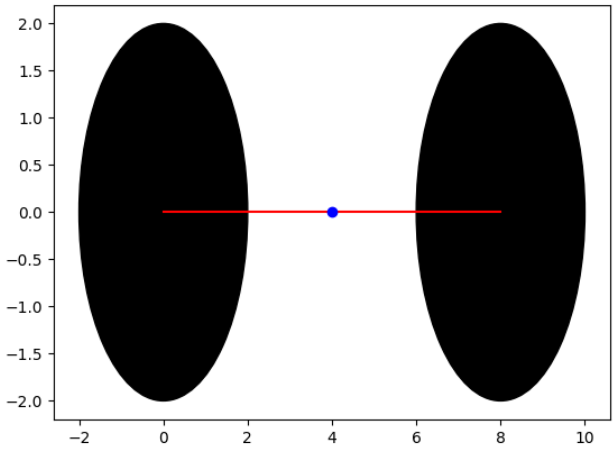
et le même mais avec repère activé :



Par ailleurs, le repère fourni par défaut par Matplotlib N'est PAS orthonomé (autrement dit, il n'a pas les mêmes unités sur les deux axes de coordonnées), ce qui peut entraîner, dans certains cas, des déformations perceptibles. Pour activer le repère orthonormé, on peut placer la ligne suivante dans le code :

```
plt.axis('equal')
```

Pour le dessin précédent, si cette ligne n'est pas présente, voilà le rendu que l'on obtient :



Pour information et test, voici le code utilisé pour générer les dessins ci-dessus :

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

def disk(center, rad, **kwargs):
    circ=plt.Circle(center, rad, lw=0, **kwargs)
```

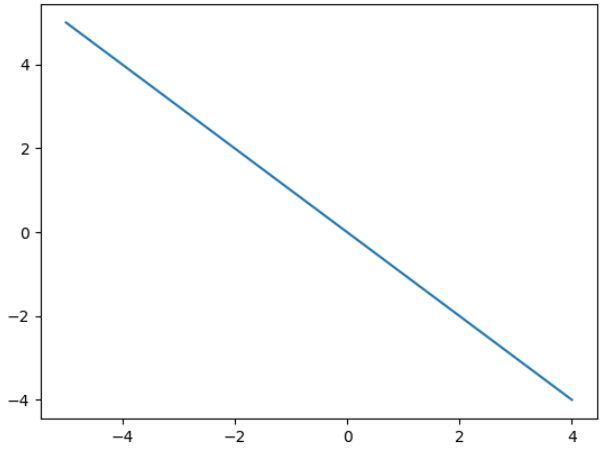
```
    plt.gca().add_patch(circ)

def line(A, B, **kwargs):
    plt.plot([A[0], B[0]],[A[1], B[1]], **kwargs)

def dot(A, **kwargs):
    plt.plot(*A, marker='o',**kwargs)

# --------- Code du dessin  --------------------------


disk((0,0), 2, color="black")
disk((8,0), 2, color= "black")
line((0,0), (8, 0), color="red")
dot((4,0), color="blue")

# -------- FIN du code du dessin --------------------

plt.axis('equal')
# plt.axis('off')
plt.savefig("fonctions_auxiliaires_matplotlib.png")
plt.show()
```

Tracé d'un segment, d'un point

Matplotlib ne dispose pas d'une simple fonction du genre `line(A, B)` qui construirait un segment d'extrémités `A` et `B` données. Pour tracer un segment, il faut utiliser la construction peu intuitive illustrée ci-dessous :

```
1  import matplotlib.pyplot as plt
2
3  # Le point A
4  xA, yA = (-5, 5)
5
6  # Le point B
7  xB, yB = (4, -4)
8
9  plt.plot([xA, xB], [yA, yB])
10
11 # plt.axis('equal')
12 # plt.axis('off')
13
14 plt.show()
```

et qui affiche le dessin suivant :



On remarquera (ligne 9) que l'on donne à la fonction `plot` non pas les points `A` et `B` mais les *abscisses* puis les *ordonnées* de ces points.

Par ailleurs, je n'ai pas activé les lignes suivantes :

— la ligne 11 : on se rend compte que le repère n'est pas orthonormé,
— la ligne 12 : on peut lire les coordonnées des points A et B.

En utilisant des arguments nommés, on peut appliquer différentes **options** telles que :

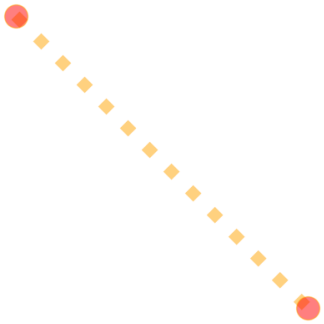
— la couleur avec `color`
— l'épaisseur avec `linewidth` ou encore `lw`
— le style de trait (pointillés variés) avec l'option `linestyle` ou encore `ls`
— le canal de transparence avec `alpha` (un flottant entre 0 et 1)
— les extrémités du segment sous forme de bulles avec `marker='o'`, leur taille avec `markersize`, leur couleur `mfc`

Voici un exemple utilisant toutes ces options :

```
1  import matplotlib.pyplot as plt
2
3  # Le point A
4  xA, yA = (-5, 5)
5
6  # Le point B
7  xB, yB = (4, -4)
8
9  plt.plot([xA, xB], [yA, yB],
10             color="orange", lw=10,
11             alpha=0.5,
12             marker="o", markersize=20, mfc="red", ls='dotted')
13
14 plt.axis('equal')
15 plt.axis('off')
```

```
16 plt.show()
```

et qui produit le dessin suivant :

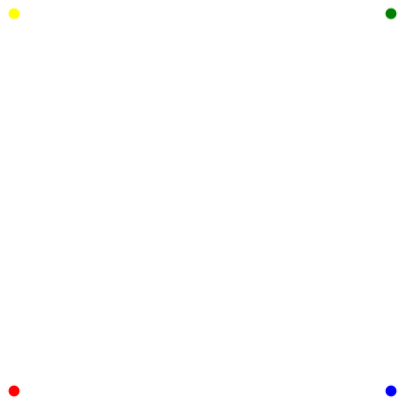


— Lignes 10-12 : les options sont placées entre parenthèses sous forme d'arguments nommés. On peut en utiliser moins si on souhaite.

Le marqueur utilisé ci-dessus permet accessoirement de tracer une « bulle » en un point donné (c'est pratique pour se repérer dans un dessin). Exemple :

```
1 import matplotlib.pyplot as plt
2
3 plt.plot(0, 0, marker="o", color='red')
4 plt.plot(1, 0, marker="o", color='blue')
5 plt.plot(1, 1, marker="o", color='green')
6 plt.plot(0, 1, marker="o", color='yellow')
7
8 plt.axis('equal')
9 plt.axis('off')
10 plt.show()
```

qui produit le dessin suivant :



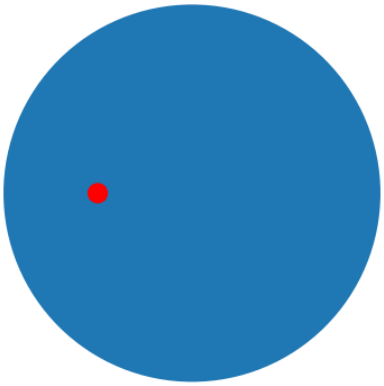
Noter (ligne 2 par exemple) la syntaxe plus simple qui n'utilise plus de listes de coordonnées mais directement la valeur de l'abscisse et de l'ordonnée.

Tracé d'un disque

Sous Matplotlib, on peut tracer assez naturellement un disque de centre et de rayon donnés :

```
1 import matplotlib.pyplot as plt
2
3 fig = plt.figure()
4 ax = fig.add_subplot()
5
6 circ = plt.Circle((2,0), 4)
7 dot = plt.Circle((0,0), 0.2, color="red")
8
9 ax.add_patch(circ)
10 ax.add_patch(dot)
11
12 plt.axis('equal')
13 plt.axis('off')
14 plt.show()
```

qui affiche les deux disques ci-dessous



Commentons le code :

— ligne 3 : il faut créer une figure `fig` ;

— ligne 4 : dans `fig`, on crée un dessin `ax` ; `ax` est un nom de variable assez répandu pour désigner un dessin (`subplot`) et qui fait référence, dans le jargon de Matplotlib à *axe* (qui ne veut pas dire axe ...) ;

— ligne 6 : on crée un disque de centre le point `(2, 0)` et de rayon 4. Par défaut, le cercle est rempli, en la couleur par défaut, un bleu ;

— ligne 7 : on crée un petit disque rouge (une « bulle » : le rayon de `0.2` est petit) centré en l'origine ;

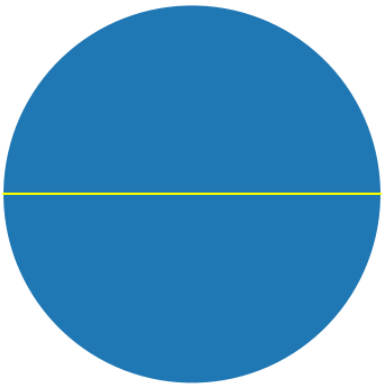
— lignes 9-10 : on insère les disques dans le dessin `ax` (sinon, ils resteraient invisibles) :

— ligne 12 : les axes sont orthonormés, sans cela les disques auraient une forme « ovale ».

On peut, bien sûr, tracer sur le même dessin à la fois des disques et des segments, par exemple :

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot()

circ = plt.Circle((2,0), 4)
ax.add_patch(circ)

A=(-2,0)
B=(6, 0)

plt.plot([A[0], B[0]],[A[1], B[1]], color="yellow")

plt.axis('equal')
plt.axis('off')
plt.show()
```

qui produit le dessin suivant :



Tracé d'un rectangle

Pour tracer un rectangle, on utilise la classe `Rectangle`, cf. cet exemple

```
from matplotlib.patches import Rectangle
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot()

xy=(-2,3)
w=4
h=2

rect = Rectangle(xy, w, h, color="gray")

ax.add_patch(rect)

plt.axis('equal')
plt.axis('off')
plt.show()
```

et qui produit la sortie graphique suivante :

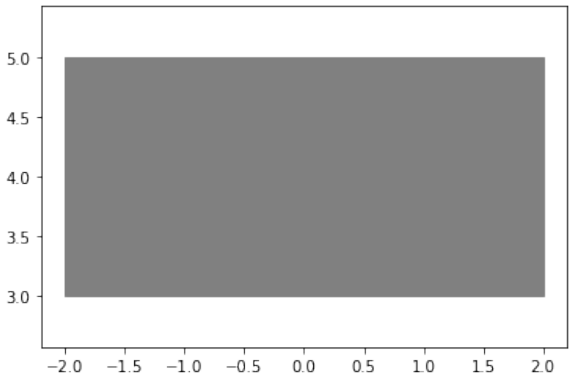
Le principe est que l'on fournit à `Rectangle` :

— le point `xy` du rectangle situé en bas à gauche,
— la largeur du rectangle et qui est la dimension horizontale
— la hauteur du rectangle et qui est la dimension verticale
— d'éventuelles options.

On comprend mieux la génération du rectangle si les axes de coordonnées sont tracés :

```
from matplotlib.patches import Rectangle
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot()

xy=(-2,3)
w=4
h=2

rect = Rectangle(xy, w, h, color="gray")

ax.add_patch(rect)

plt.axis('equal')

plt.show()
```

et qui produit la sortie graphique suivante :



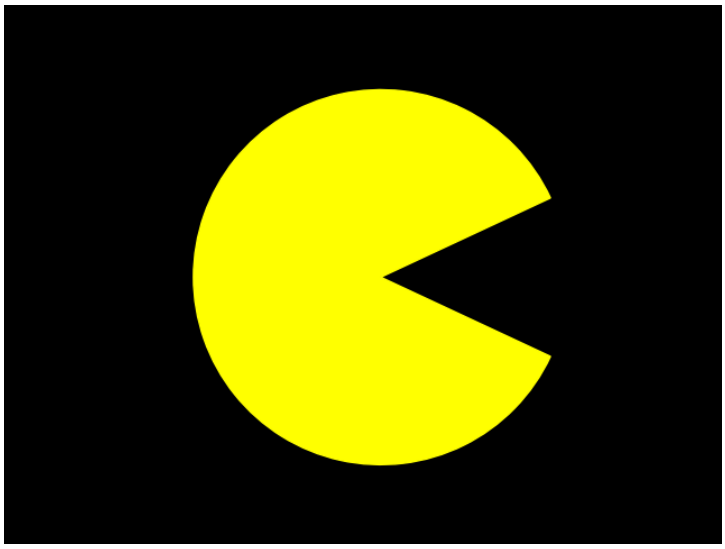
On notera que, par défaut, l'intérieur du rectangle est rempli. Si ce n'est pas souhaité, utiliser l'option `fill=False`.

Tracé d'un secteur circulaire

Pour tracer un secteur circulaire, on utilise la classe `Wedge`, cf. cet exemple :

```
1  import matplotlib.pyplot as plt
2  from matplotlib.patches import Wedge
3
4  fig = plt.figure(facecolor="black")
5  ax = fig.add_subplot()
6
7  centre=(2,0)
8  radius=2
9  start=25
10 end=360-start
11
12 pacman = Wedge(centre, radius, start, end, color="yellow")
13 ax.add_patch(pacman)
14
15 ax.set_facecolor("black")
16 plt.axis('off')
17 plt.axis('equal')
18 plt.savefig("pacman.png", facecolor="black")
19 plt.show()
```

et qui produit la sortie graphique suivante :



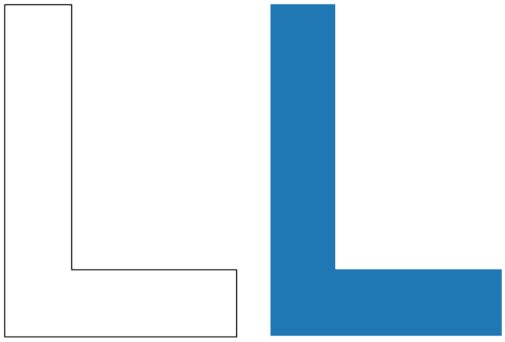
Dans le code ci-dessus, `facecolor` fait référence à la couleur de fond.

Tracer un polygone

On peut tracer un polygone avec le constructeur `Polygon` en donnant une liste de points en coordonnées :

```
from matplotlib.patches import Polygon
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot()

A=(0,0)
B=(0,5)
C=(1,5)
D=(1,1)
E=(3.5, 1)
F=(3.5,0)
polygon = Polygon([A, B, C, D, E, F], fill=False)
ax.add_patch(polygon)

A=(4,0)
B=(4,5)
C=(5,5)
D=(5,1)
E=(7.5, 1)
F=(7.5,0)
polygon = Polygon([A, B, C, D, E, F])
ax.add_patch(polygon)
```

```
plt.axis('equal')
plt.axis('off')
plt.show()
```

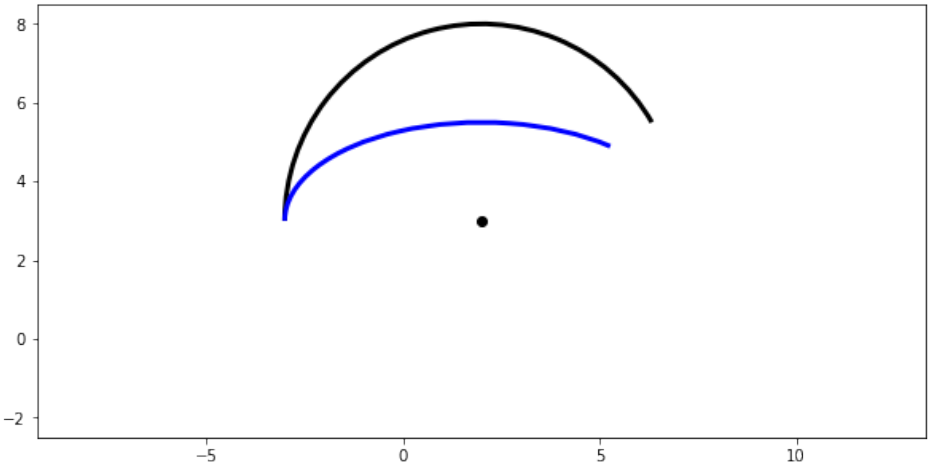


Par défaut, le polygone est rempli par la couleur par défaut. Pour ne pas remplir, placer l'option `fill` à `False`. Pour obtenir juste une ligne polygonale, utiliser l'argument nommé `closed=False`.

Tracer un arc de cercle ou d'ellipse

On peut tracer des arcs d'ellipse avec la classe Arc, en particulier des arcs de cercles. Exemple :

```
import matplotlib.pyplot as plt
from matplotlib.patches import Arc

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot()

A=(2,3)
R=5

arc = Arc(A, 2*R, 2*R,  theta1=30, theta2=180.0,lw=3)
ax.add_patch(arc)

arc = Arc(A, 2*R, R,  theta1=30, theta2=180.0,lw=3, edgecolor="blue")
ax.add_patch(arc)

plt.plot(*A, marker="o", color='black')
plt.axis('on')
plt.axis('equal')

plt.show()
```



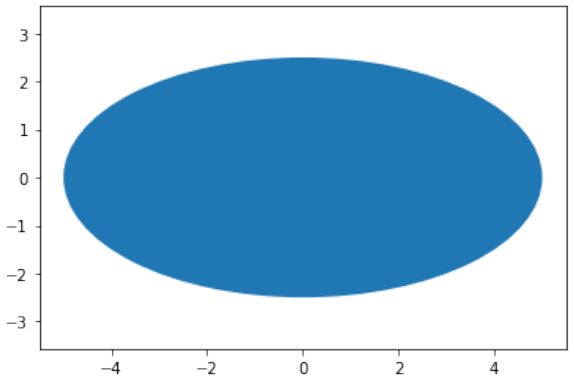
Le premier argument est le centre de l'ellipse ou du cercle. Les deux arguments suivants sont la longueur de l'axe horizontal et la longueur de l'axe vertical. Si ces longueurs sont égales, l'ellipse est un cercle et la longueur commune est son *diamètre* (et pas son rayon).

Ensuite, viennent des arguments nommés qui sont des angles par rapport au premier axe de coordonnées : le début de l'arc `theta1` et la fin de l'arc `theta2`. Les angles sont donnés en degrés.

Tracer une ellipse

On peut tracer une ellipse avec la classe `Ellipse`. Les axes de l'ellipses sont parallèles aux axes de coordonnées. Exemple :

```
1  import matplotlib.pyplot as plt
2  from matplotlib.patches import Ellipse
3
4  fig = plt.figure()
5  ax = fig.add_subplot()
6
7  shape = Ellipse((0,0), 10, 5)
8  ax.add_patch(shape)
9
10 plt.axis('equal')
11 plt.axis('on')
12 plt.show()
```



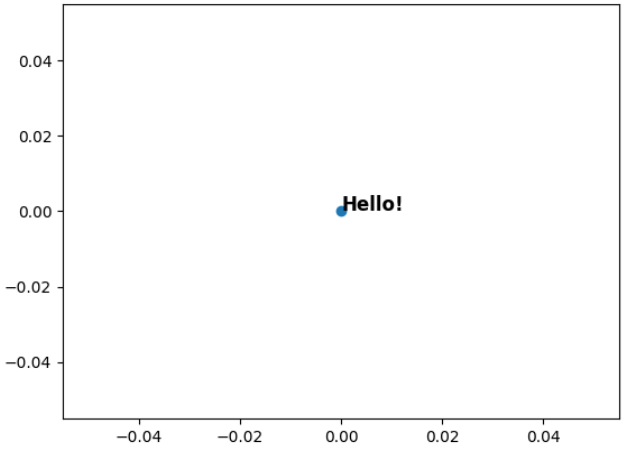
Le premier argument est le centre de l'ellipse. Les deux arguments suivants sont la longueur de l'axe horizontal et la longueur de l'axe vertical. Si ces longueurs sont égales, l'ellipse est un cercle et la longueur commune est son *diamètre* (et pas son rayon).

Texte basique

Pour tracer un texte simple, utiliser la fonction `text` :

```
import matplotlib.pyplot as plt
plt.plot(0, 0, marker = 'o')
plt.text(0, 0, "Hello!", fontsize=12, weight="bold")

plt.show()
```

qui affiche



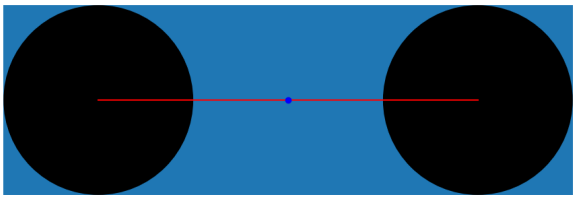
On observera la position de la boîte de texte par rapport au point fourni à la fonction, dans l'exemple c'est le point `(0, 0)`.

Utiliser des fonctions comme raccourci de code

Pour éviter la syntaxe peu naturelle de tracé de segment et d'écrire l'inclusion obligatoire de l'objet de type `Circle` dans le dessin courant, on peut créer ses propres fonctions.
Voici une possibilité :

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

plt.gcf().set_size_inches(12,12)

def disk(center, rad, **kwargs):
    circ=plt.Circle(center, rad, lw=0, **kwargs)
    plt.gca().add_patch(circ)

def line(A, B, **kwargs):
    plt.plot([A[0], B[0]],[A[1], B[1]], **kwargs)

def dot(A, **kwargs):
    plt.plot(*A, marker='o',**kwargs)

def rect(xy, w, h, **kwargs):
    my_rect = Rectangle(xy, w, h,**kwargs)
    plt.gca().add_patch(my_rect)

# --------- Code du dessin  --------------------------

rect((-2, -2),12, 4)
disk((0,0), 2, color="black")
disk((8,0), 2, color= "black")
line((0,0), (8, 0), color="red")
dot((4,0), color="blue")

# -------- FIN du code du dessin --------------------

plt.axis('equal')
plt.axis('off')
plt.show()
```

code qui produit le dessin suivant :



La ligne 4 permet de modifier la taille du dessin (ci-dessus 12 pouces x 12 pouces). Les lignes 1-20 et 28-32 peuvent être incluses dans tout fichier qui utilise Matplotlib pour faire du dessin. Il suffit d'écrire son code dans les lignes 21-27 et le code devient alors nettement plus simple à produire.

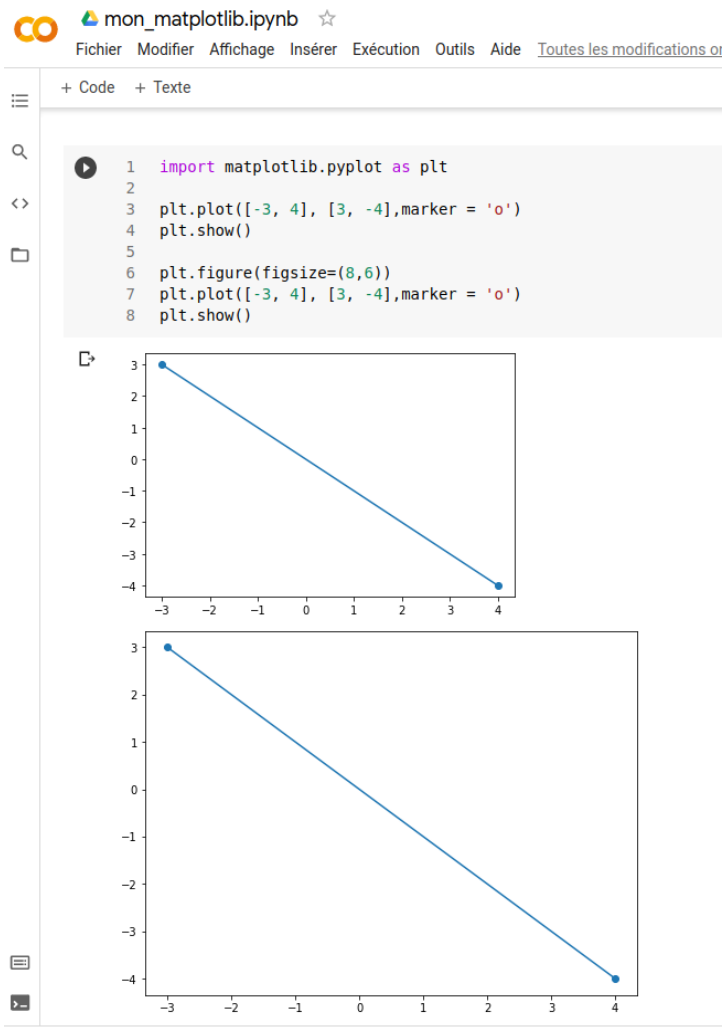
Les paramètres du type `**kwargs` permettent d'utiliser, si on en a besoin, les arguments nommés autorisés (comme `color`) et vus précédemment.

Modifier la taille de la figure courante

Une figure Matplotlib a une taille par défaut (en pouces, 6,4 x 4,8). On peut modifier la taille de cette figure avec l'argument nommé `figsize`, par exemple :

```
import matplotlib.pyplot as plt

plt.plot([-3, 4], [3, -4],marker = 'o')
plt.show()

plt.figure(figsize=(8,6))
plt.plot([-3, 4], [3, -4],marker = 'o')
plt.show()
```

dont voici une vue dans une feuille Jupyter :



Le premier dessin a la taille par défaut.
Le second a pour taille (8, 6).

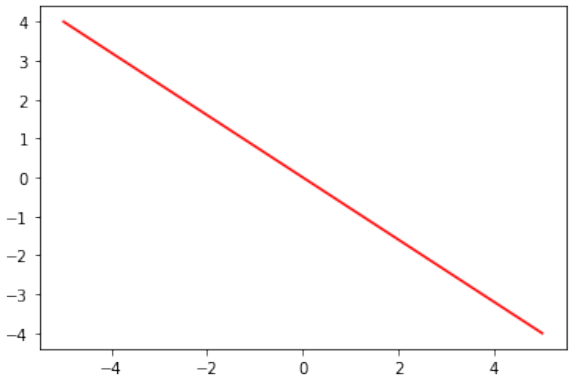
Les couleurs sous Matplotlib

Il existe de nombreuses façons de transmettre des couleurs à des options Matplotlib.

Les couleurs nommées

La façon la plus simple de transmettre les couleurs est d'utiliser une chaîne de caractères donnant le nom anglais d'une couleur (courante), typiquement `yellow` ou `blue` :

```
import matplotlib.pyplot as plt

plt.plot((-5, 5), (4, -4), color="red")

plt.show()
```



Le nom de couleur n'est pas sensible à la casse, autrement dit, il est indifférent d'écrire `"YeLLow"` ou `"yellow"`.

Au-delà des couleurs courantes, on peut utiliser les noms de couleurs utilisées dans des feuilles de style CSS, par exemple, on peut utiliser la couleur `"MediumTurquoise"`.

Les couleurs par défaut

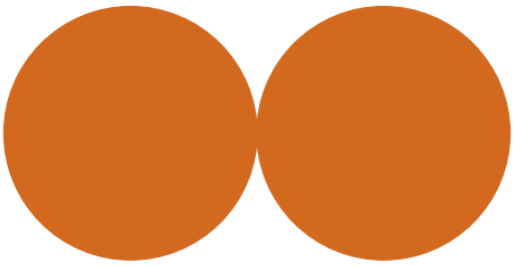
Quand une couleur est prévue et qu'elle n'est pas communiquée, une couleur par défaut existe, le plus souvent un bleu proche du bleu acier, appelé parfois strong blue, utilisable dans du code Matplotlib par la chaîne littérale `"tab:blue"`. Attention, parfois, les couleurs par défaut suivent un cycle :

```
1 import matplotlib.pyplot as plt
2
3 plt.figure(figsize=(15,5))
4
5 for i in range(20):
6         plt.plot(i, 0, marker="o", markersize=30)
7
8 plt.axis('equal')
9 plt.axis('off')
10 plt.show()
```

Les couleurs RGB

On peut aussi transmettre une couleur au format RGB, avec une syntaxe décimale ou hexadécimale, voici un exemple :

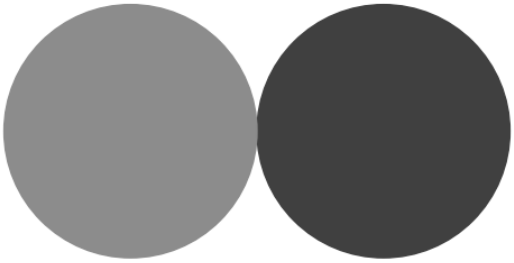
```
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot()


circ1 = plt.Circle((4,0), 2, color="#D2691E")

r=210/255
g=105/255
b=30/255

circ2 = plt.Circle((0,0), 2, color=(r, g, b))

ax.add_patch(circ1)
ax.add_patch(circ2)


plt.axis('equal')
plt.axis('off')
plt.show()
```



Les nuances de gris

Un gris en RGB se code avec une seule couleur, entre le noir, codé par 0 et le blanc, codé par 1. Exemple avec deux disques gris :

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot()


circ1 = plt.Circle((4,0), 2, color="0.25")
circ2 = plt.Circle((0,0), 2, color="0.55")

ax.add_patch(circ1)
ax.add_patch(circ2)


plt.axis('equal')
plt.axis('off')
plt.show()
```



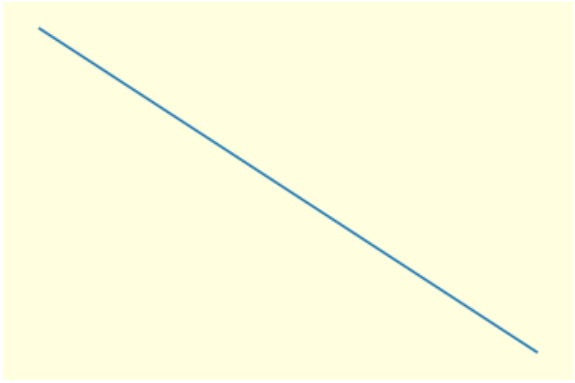
Pour les détails sur le passage des couleurs, on pourra consulter l'aide officielle.

Placer une couleur de fond

Il peut être souhaitable qu'une figure porte une couleur de fond, par exemple, pour qu'elle se distingue bien de son environnement d'affichage. Il existe une option pour cela :

```
import matplotlib.pyplot as plt

fig = plt.figure()
fig.patch.set_facecolor('lightyellow')

plt.plot((-5, 5), (4, -4))
```

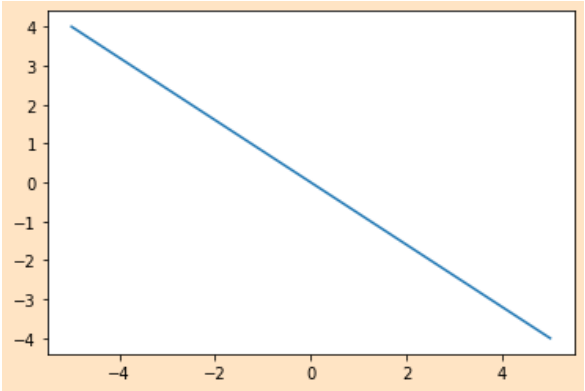
```
plt.axis('off')
plt.show()
```



Si les axes sont activés, cette couleur de fond ne s'applique qu'à l'extérieur du rectangle de repérage :

```
import matplotlib.pyplot as plt

fig = plt.figure()
fig.patch.set_facecolor('bisque')

plt.plot((-5, 5), (4, -4))

plt.axis('on')
plt.show()
```

ce qui est assez compréhensible.



Sauvegarder le dessin dans un fichier png

Il est possible de sauvegarder la production graphique dans un fichier png. Voici un exemple de code qui réalise cette tâche :

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')

plt.savefig("exporter_code_matplotlib.png")
```

ce qui va créer dans le répertoire d'où a été lancé Python un fichier, nommé ici `exporter_code_matplotlib.p`
Il existe différentes options que l'on peut consulter ici. Par exemple, pour modifier la résolution,
on utilisera l'option `dpi` :

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')

plt.savefig("exporter_code_matplotlib.png", dpi=120)
```

Pour ajuster le contenu de l'image par rapport au bord, on pourra jouer sur l'option `bbox_inches` ;
en particulier, l'option `bbox_inches='tight'` semble ajuster serré le contenu de l'image mais
l'ajustement peut aussi nécessiter de jouer sur la taille de l'image via l'option (de figure) `set_size_inches`.

Fonctions prêtes à l'emploi

Voici quelques fonctions qui facilitent l'usage de certaines primitives données par Matplotlib et décrites ci-dessus.

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, Arc, Polygon, Wedge, Ellipse

# Choisir la taille voulue
plt.gcf().set_size_inches(10,5)

def disk(center, rad, **kwargs):
    circ=plt.Circle(center, rad, lw=0, **kwargs)
    plt.gca().add_patch(circ)

def line(A, B, **kwargs):
    plt.plot([A[0], B[0]],[A[1], B[1]], **kwargs)

def dot(A, **kwargs):
    plt.plot(*A, marker='o',**kwargs)

def rect(xy, w, h, **kwargs):
    my_rect = Rectangle(xy, w, h,**kwargs)
    plt.gca().add_patch(my_rect)

def circle_arc(center, R,  theta1, theta2, **kwargs):
    arc = Arc(center, 2*R, 2*R,  theta1=theta1, theta2=theta2, **kwargs)
    plt.gca().add_patch(arc)
```

```
def polygon(L, **kwargs):
    my_polygon = Polygon(L, **kwargs)
    plt.gca().add_patch(my_polygon)

def wedge(centre, radius, start, end, **kwargs):
    my_wedge = Wedge(centre, radius, start, end, **kwargs)
    plt.gca().add_patch(my_wedge)

def ellipse(center, w, h, **kwargs):
    ell=Ellipse(center, w, h,  **kwargs)
    plt.gca().add_patch(ell)

# --------- Code du dessin  -------------------------




# -------- FIN du code du dessin --------------------

plt.axis('equal')
plt.axis('off')
plt.show()
```

Vous écrivez votre code entre les lignes pointillés. Si vous le souhaitez, vous pouver supprimer les fonctions au-dessus des pointillés et qui ne vous sont pas utiles.

Fonction `addsubplot` et version de Matplotlib

Sous certaines versions de Matplotlib, l'usage de `add_subplot` peut échouer comme dans l'exemple ci-dessous :

```
1 import matplotlib.pyplot as plt
2
3 fig = plt.figure()
4 ax = fig.add_subplot()
5
6 circ = plt.Circle((2,0), 4)
7 dot = plt.Circle((0,0), 0.2, color="red")
8
9 ax.add_patch(circ)
10 ax.add_patch(dot)
11
12 plt.axis('equal')
13 plt.axis('off')
14 plt.show()
```

```
15 'NoneType' object has no attribute "add_patch"
```

car `fig.add_subplot()` ligne 4 renvoie `None`. Comme indiqué dans ce message, cela se produit pour des versions de Matplotlib strictement antérieure à 3.1 (2019). Pour y remédier, c'est très

simple, il suffit de changer la ligne 4 par

`ax = fig.add_subplot(1,1,1)` voire `ax = fig.add_subplot(111)`

qui fonctionne aussi avec les versions plus récentes de Matplotlib.
Pour connaître votre version de Matplotlib, lancez le code suivant :

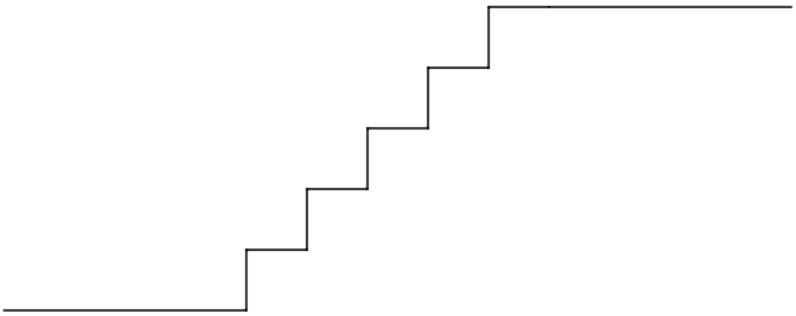
```
import matplotlib as mpl
print(mpl.__version__)
```

```
3.0.2
```

EXERCICES

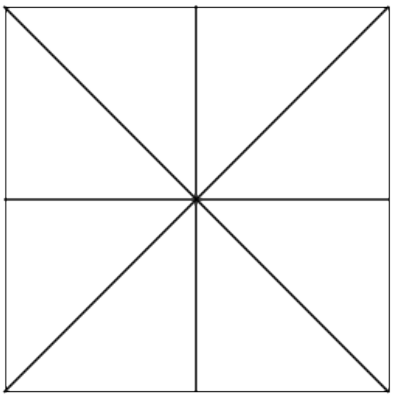
Quelques marches

Imaginez une maison. Dessinez le sol avec un escalier à 5 marches suivi du sol de l'étage (ne pas dessiner la maison). Utiliser une variable pour désigner la hauteur et le profondeur de chaque marche. Vous pouvez aussi introduire des variables x et y pour désigner les coordonnées de la marche courante.



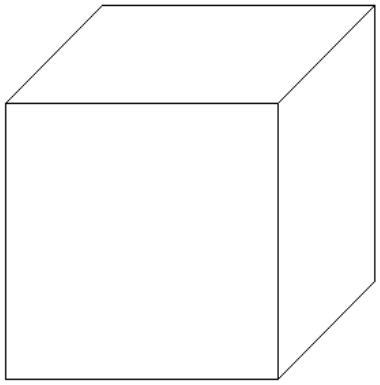
Découper un carré

Dessiner un carré découpé en huit secteurs égaux.



Cube

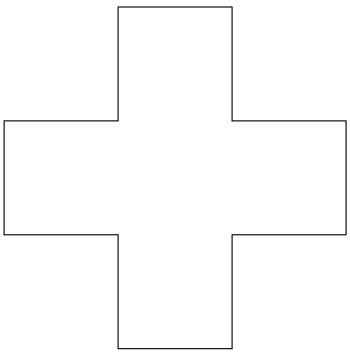
Dessiner un cube



dont la représentation dépendra de deux variables `c` (le côté du carré de la face avant) et `h` la « hauteur » apparente la face supérieure. **Si possible**, faites en sorte que tous les angles aigus de la représentation soient à 45°.

Dessiner un signe *plus*

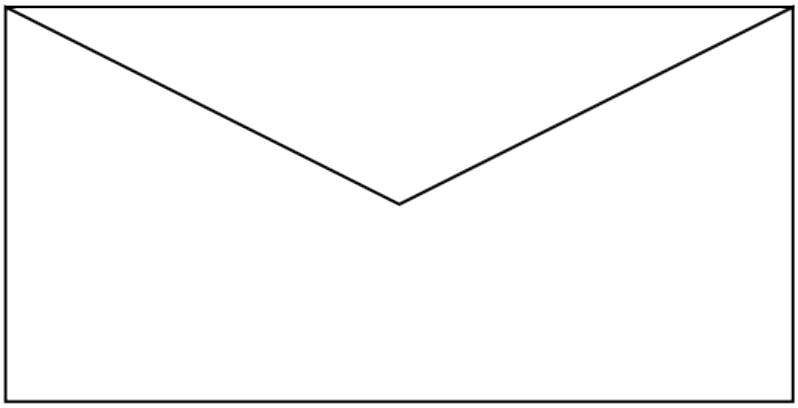
Dessiner une croix sous la forme d'un signe +. La zone centrale de la croix sera un carré de côte de côté c, chaque branche de la croix aura une longueur et une largeur valant encore c.



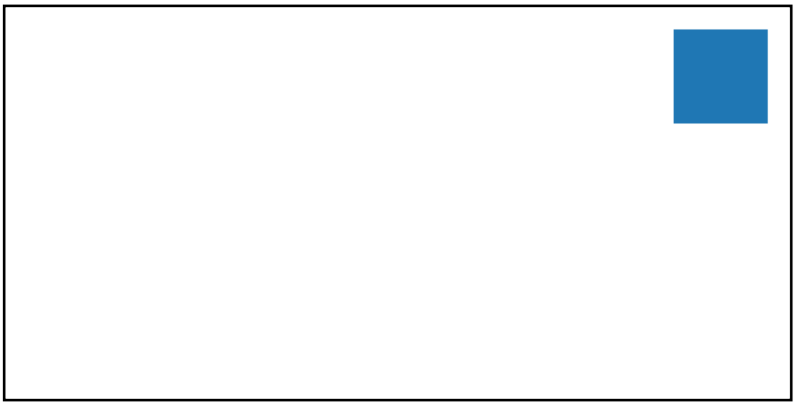
Enveloppes

Dans cet exercice, on s'efforcera d'utiliser des variables représentant les dimensions utiles au dessin.

(1) Dessiner une enveloppe vue de dos (avec le rabat) :

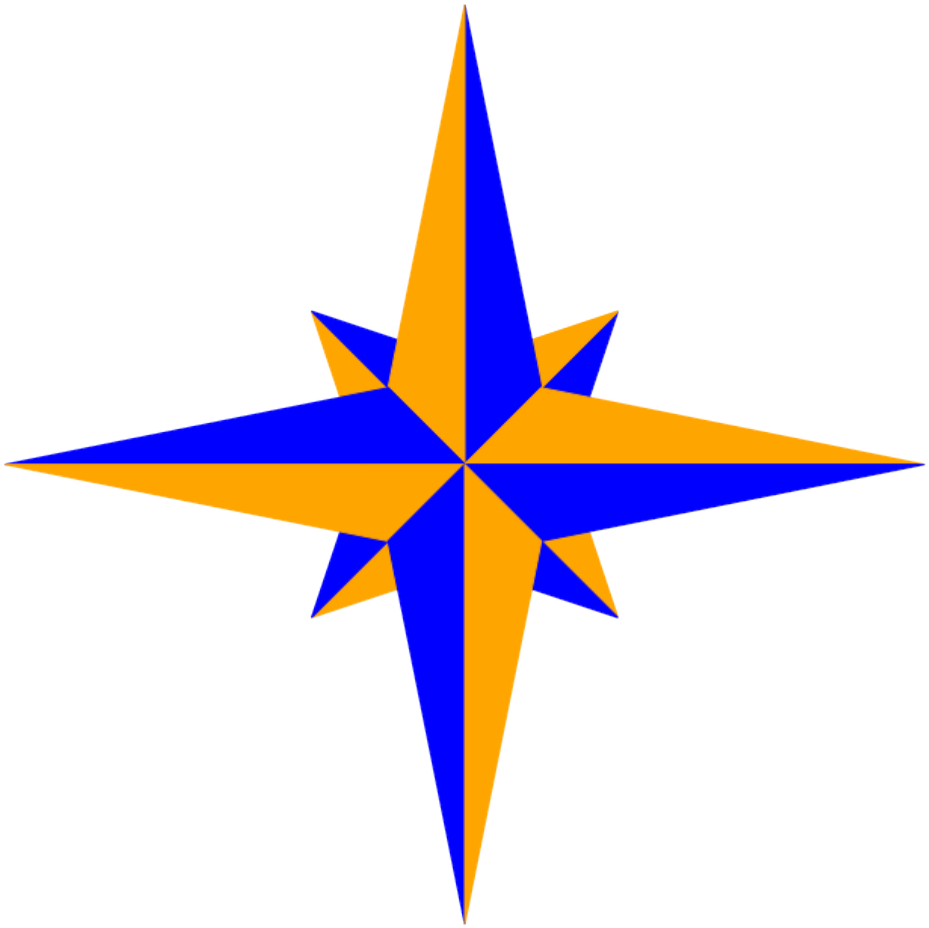


(2) Dessiner une enveloppe timbrée :



Rose des vents

Dessiner une rose des vents. Nommer les points nécessaires avec des variables. Il n'y a aucun calcul à faire, simplement bien observer les alignements et procéder par masquage. Utiliser l'outil `Polygon`. Les couleurs sont orange et bleu.



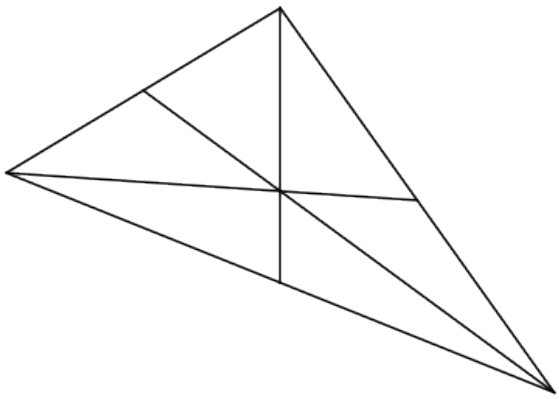
Centre de gravité

Soient les points $\texttt{a} = (50, 50)$, $\texttt{b} = (-200, -100)$ et $\texttt{c} = (300, -300)$. L'abscisse d'un point `m` vaut `m[0]` et l'ordonnée `m[1]`, par exemple :

```
a=(50, 50)
print(a[0])
print(a[1])
```

1. Construire le triangle `abc`.
2. Rappelons qu'une *médiane* d'un triangle est une droite qui joint un sommet et le milieu du côté opposé. On sait que les trois médianes se coupent en le centre de gravité du triangle (g ci-dessous).
   Construire les médianes du triangle `abc`. On utilisera que si on a deux points représentés respectivement par les couples `x` et `y` alors le milieu est représenté par le couple :

```
m=((x[0]+y[0])/2,(x[1]+y[1])/2)
```



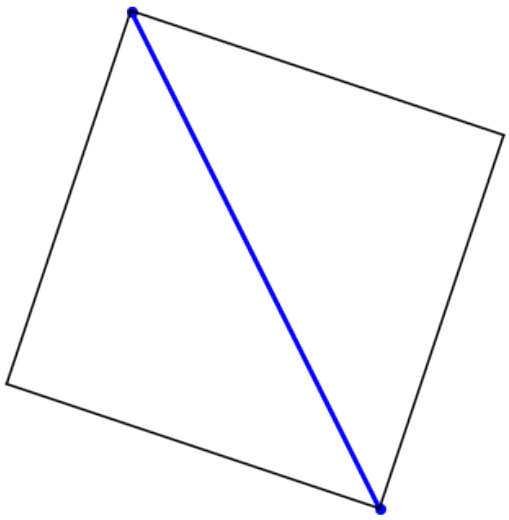
Dessiner un huit

Dessiner le chiffre 8 en plaçant deux ellipses noires ayant une intersection commune et en les creusant avec deux ellipses blanches.

Cadrer carré

On donne deux points distincts A et B, par exemple

et on cherche à construire la carré de diagonale le segment AB, comme ci-dessous :

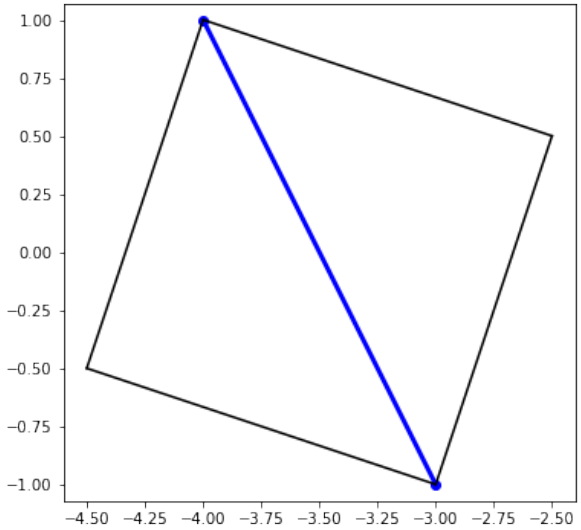


Si `M` est un point, on note `xM` et `yM` ses coordonnées. Les points A et B sont donnés par leurs coordonnées. On admet que l'autre diagonale `CD` du carré est telle que

$$\texttt{C} = (\texttt{xM} - \texttt{v}, \texttt{yM} + \texttt{u}) \text{ et } \texttt{D} = (\texttt{xM} + \texttt{v}, \texttt{yM} - \texttt{u})$$

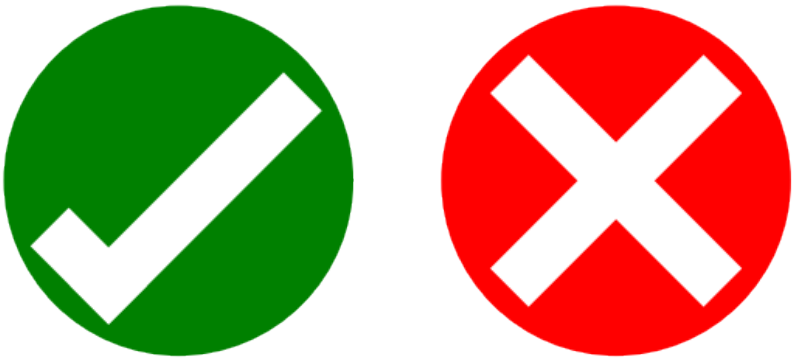
où `M` est le milieu du segment `AB` et $(\texttt{u}, \texttt{v}) = \dfrac{1}{2}\overrightarrow{AB}$.

Connaissant les point `A` et `B`, on demande de dessiner le carré comme illustré ci-dessus où on avait choisi `A = (-4,1)` et `B = (-3,-1)` comme on le voit ci-dessous :



Logos succès-échec

Dessiner côte-à-côte les deux logos suivants :



La croix et la coche ne sont pas dessinées comme des polygones mais comme des lignes épaisses (utiliser `line` et son option `lw`). Utiliser des variables. On pourra utiliser une petite distance pour décaler la coche par rapport au centre du disque.

Danger nucléaire

Dessiner le logo « danger nucléaire » :

Les secteurs angulaires mesurent 60°. On aura besoin de `Circle` et de `Wedge`.

Logo de Tripadvisor

Dessiner le logo de Tripadvisor :

C'est un logo assez difficile à dessiner. Quelques conseils :

— faites un croquis à main levée,
— donnez des noms aux points utiles,

— définissez toutes les variables utiles, elles sont très nombreuses.

La couleur verte est un vert perse, poser `vert = "#00AF87"`.
Suggestion de plan de dessin à suivre :

— Tracer le disque de fond
— Dessiner les pointes gauche et droite des yeux à l'aide d'un triangle noir pour chaque oeil (fonction `polygon`)
— Dessiner le front noir et vert avec deux quarts de disque superposés (fonction `wedge`)
— Dessiner la pointe du nez (fonction `polygon`)
— Dessiner les yeux (chaque oeil nécessite trois disques) avec la fonction `disk`.

Voici un lien vers une description géométrique du logo

Chapitre III

Conditions

1 Booléens

Opérateurs logiques ET, OU et NON

Les notions d'*assertion logique* et d'*opérateur logique* existent indépendamment de tout langage de programmation. L'énoncé $2k+1 \geq 100$ où k est un entier, est un exemple d'assertion logique [1] (elle est vraie ou fausse selon la valeur de k, vraie si $k = 60$, fausse si $k = 25$).

L'opérateur ET

À partir de deux propositions logiques p et q, on peut former la proposition logique `p ET q`. Par exemple, si on a les deux propositions logiques :

`p : 5 > 3`, `q : 4 > 7`

alors

— `p` est *vraie*

— `q` est *fausse*

— `p ET q` est une proposition *fausse*.

Règle : `p ET q` est vraie uniquement si p est vraie et si q est également vraie.
Cette règle est conforme à l'usage habituel de la conjonction de coordination ET.
Voici donc la table de vérité de l'opérateur `ET` :

p	q	p ET q
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

Ne pas accorder trop d'importance à cette table, en pratique assez peu utile. Une expression de la forme `p ET q` s'appelle une *conjonction* et p et q s'appellent les *opérandes* de la conjonction.

1. Au lieu d'*assertion logique*, on rencontre parfois les terme de *proposition* ou de *prédicat*.

La notion d'opérateur OU

À partir de deux propositions logiques p et q, on peut former la proposition logique p OU q. Par exemple, si

p : 5 > 3, q : 4 > 7

alors :

— p est vraie

— q est fausse

— p OU q est une proposition vraie.

Règle : p OU q est fausse uniquement si p est fausse *et* si q est également fausse.

Une expression de la forme p OU q s'appelle une *disjonction*. Les assertions p et q s'appellent les *opérandes* de la disjonction.

Voici la table de vérité de l'opérateur OU :

p	q	p OU q
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

Ne pas accorder trop d'importance à cette table, en pratique assez peu utile.

Le OU est non exclusif

Le OU logique n'a pas valeur exclusive comme dans *fromage ou dessert* et qui signifie

soit fromage soit dessert mais pas les deux.

Autrement dit, dès que l'une des deux assertions p ou q sont les deux vraies alors l'assertion p OU q est vraie.

Par exemple, si

p : 5 > 3, q : 4 < 7

alors :

— p est vraie,

— q est vraie,

— la proposition p OU q est vraie.

L'opérateur logique NON

La proposition NON p est la *négation* de p. Si

p : 5 > 3

alors :

— p est vraie

— NON p est fausse.

Règle : toute proposition logique p admet une négation q, notée NON p. La valeur logique de q est

— vraie si p est fausse,

— fausse si p est vraie.

Les booléens en Python

Les valeurs de vérité *Vrai* et *Faux* en Python sont représentées par deux constantes du langage :

— `True` pour Vrai

— `False` pour Faux

`True` et `False` sont des mots-clés du langage Python 3. Les booléens `True` et `False` appartiennent, comme les entiers ou les flottants, à un type, le type `bool`.

Opérateurs de comparaison et booléen

Certaines expressions ont pour valeur des booléens. C'est le cas, par exemple, de toute expression représentant une comparaison [2]. Par exemple, l'expression Python `42 > 45` vaut `False` :

```
1 print(42 > 45)
2 print(42 == 40 + 2)
```

```
3 False
4 True
```

— Lignes 3-4 : l'assertion « 42 est strictement supérieur à 45 » vaut `False`.

— Ligne 1 : l'assertion « 42 a même valeur que 40 + 2 » vaut `True` (elle est vraie).

Voici un exemple d'utilisation :

```
1 n = 5 * 12 + 6 ** 3 - 42 + 17
2 print(n > 256)
3 print(n)
```

```
4 False
5 251
```

— Ligne 1 : n est un entier dont la valeur explicite n'apparaît pas dans le code.

2. Les opérateurs de comparaison numérique sont : == < > <= > !=

— Ligne 2 : On teste pour savoir si l'entier n est plus grand que 256.
— Ligne 4 : On lit que non (cf. `False`), n n'est pas plus grand que 256.
— Lignes 3 et 5 : le calcul explicite de n confirme que l'assertion `n > 256` est fausse.

En pratique, les comparaisons utilisent souvent des variables. Par exemple, on se donne un entier k et on cherche à savoir si $2k + 1 \geq 100$. En Python on écrira :

```
1 k=10
2 print(2*k+1 >= 100)
3
4 k=60
5 print(2*k+1 >= 100)
```

```
6 False
7 True
```

Opérateurs `and`, `or` et `not`

L'opérateur logique ET est représenté en Python par le mot-clé et opérateur `and` :

```
1 p = (5 > 3)
2 q = (4 > 7)
3 print(p)
4 print(q)
5 print(p and q)
```

```
6 True
7 False
8 False
```

— Lignes 5 et 8 : `p and q` vaut `False` puisque q vaut `False`.

L'opérateur logique OU est représenté en Python par le mot-clé et opérateur `or`.

```
1 p = (5 > 3)
2 q = (4 > 7)
3 print(p)
4 print(q)
5 print(p or q)
```

```
6 True
7 False
8 True
```

— Lignes 5 et 8 : `p or q` vaut `True` puisque p vaut `True`.

L'opérateur NON est représenté en Python par le mot-clé et opérateur `not`.

```
1 p = (5 == 2 + 3)
2 print(p, not p)
3
4 q = (3 > 4)
5 print(q, not q)
```

```
True False
False True
```

Expressions booléennes

De la même façon qu'il existe des expressions arithmétiques comme

```
x*(x+1) - 4*x*y + 1,
```

il existe des *expressions booléennes*, comme

```
p and (q or r) or (not p and q),
```

qui s'obtiennent en combinant des opérateurs logiques avec des expressions.
Évaluer une expression booléenne, c'est déterminer si elle vaut True ou False.
Par exemple

```
(1 == 2 and 2 == 3) or 1 == 1
```

est un expression booléenne dont la valeur est `True`.
Bien se rendre compte que dans de nombreuses situations, les parenthèses figurant dans l'expression sont indispensables. Ainsi, comparer :

```
a=1
b=2
print((not a) == b)
print(not a == b)
```

```
False
True
```

Pour conclure, donnons deux exemples typiques d'expressions booléennes.

Exemple 1 : être positif

Il est fréquent que les expressions booléennes utilisent des variables. Par exemple, si `x` est une variable, on peut définir une expression booléenne qui exprime si oui ou non `x` représente un nombre positif : `x >= 0`. On peut référencer ces expressions booléennes elles-mêmes par une variable. Ainsi :

```
x = 2**19 - 10**6
estPos = (x >= 0)

print(estPos)
print(x)

vrai = True
print(estPos == vrai)
```

```
9 False
10 -475712
11 False
```

— Ligne 2 : on a défini une « variable booléenne » `estPos` et qui ici permet de tester le caractère positif de n'importe quel nombre `x` défini au préalable. L'usage des parenthèses n'est pas obligatoire, autrement dit, l'écriture `estPos = x >= 0`, bien que moins lisible, est licite.

— Ligne 4 : l'affichage (ligne 10) montre que `x` est négatif donc la valeur de la variable `estPos` est bien `False` (ligne 9).

— Ligne 6 : on peut placer la constante `True` dans une variable. Idem pour la constante `False`.

— Ligne 7 : si `b` est une variable booléenne, l'expression `b == True` a toujours même valeur que `b`, autrement dit, en pratique, il n'est que rarement justifié d'écrire une expression de la forme `b == True`.

Exemple 2 : comparer des heures

Cet exemple est traité en vidéo ICI.

On donne deux moments de la journée, désignés par `M` et `MM`, en heures et minutes, par exemple

`M = 14 h 52 min` et `MM = 14 h 51 min`.

et on demande de dire si, oui ou non, `M` est un moment de la journée strictement antérieur à `MM`. Dans l'exemple ci-dessus, la réponse est *non*.

Chaque moment sera donné sous la forme de deux entiers `h`, `m` représentant les heures et les minutes pour `M` et de même `hh`, `mm` pour `MM`.

Pour exprimer que `M` est antérieur à `MM`, on compare les heures : $h < hh$. Si elles sont distinctes, on peut conclure, sinon c'est que l'expression booléenne $h == hh$ est `True`, et il faut comparer les minutes ($m < mm$) pour conclure. D'où l'expression booléenne suivante :

```
(h < hh) or (h == hh and m < mm)
```

Cette expression est vraie si et seulement si `M` est strictement antérieur à `MM`. Voici un exemple d'utilisation :

```
1 h=14
2 m=52
3
4 hh=14
5 mm=51
6
7 print((h < hh) or (h == hh and m < mm))
```

```
8 False
```

Exercice type – Année bissextile

Cet exercice est résolu en vidéo : ICI.
Une année bissextile est une année multiple de 4, non multiple de 100 à moins qu'elle ne soit multiple de 400. Ainsi,

— 2021 n'est pas bissextile puisque 2021 n'est pas un multiple de 4 ;

— 2024 est bissextile car elle est multiple de 4 et non multiple de 100 ;

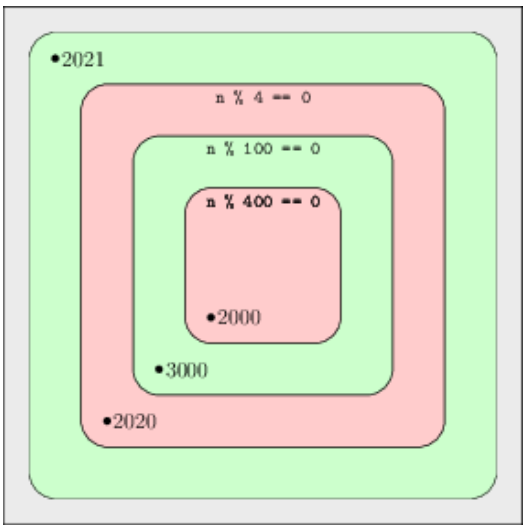
— 3000 n'est pas bissextile car elle est multiple de 100 sans être multiple de 400 ;

— 2000 était bissextile car 2000 est multiple de 400.

Soit `n` une année. En considérant que les restes de `n` par 4, 100 ou 400 sont nuls ou pas, écrire un booléen `estBissext` qui vaut `True` si `n` est bissextile et `False` sinon.

Solution

Si on fait un dessin de toutes les années en distinguant les multiples de 4, de 100 et de 400, on voit que les années bissextiles se répartissent dans les deux zones rouges disjointes ci-dessous :



On distingue une couronne rouge et le disque rouge au centre. La couronne rouge représente les années `n` multiples de 4 ET qui ne sont pas multiples de 100. Donc la couronne rouge représente le booléen

```
n % 4 == 0 and n % 100 != 0
```

Le disque au centre représente les années `n` multiples de 400, donc le booléen `n % 400 == 0`.
Une année est bissextile exactement si elle est dans la couronne OU dans le disque central, ce qui se traduit par le OU booléen suivant :

```
estBissext = (n % 4 == 0 and n % 100 != 0) or n % 400 == 0
```

On pouvait aussi remarquer que `n` est bissextile si elle est multiple de 4 et ne fait pas partie des années telle 1900 ou 3000 qui sont multiples de 100 et pas de 400 (la couronne verte du dessin),

autrement dit si : n n'est pas multiple de 100 ou est multiple de 400. D'où le booléen suivant [3] :

```
estBissext = (n % 4 == 0) and (n % 100 != 0 or n % 400 == 0)
```

D'où le code suivant qui traite les quatre cas possibles :

```
n=2025
p = (n % 4)==0
q = (n % 100)==0
r = (n % 400)==0
estBissextile = p and (not q or r)
print(n, "->", estBissextile)
print("------------------------------")

n=2020
p = (n % 4)==0
q = (n % 100)==0
r = (n % 400)==0
estBissextile = p and (not q or r)
print(n, "->", estBissextile)
print("------------------------------")

n=3000
p = (n % 4)==0
q = (n % 100)==0
r = (n % 400)==0
estBissextile = p and (not q or r)
print(n, "->", estBissextile)
print("------------------------------")

n=2000
p = (n % 4)==0
q = (n % 100)==0
r = (n % 400)==0
estBissextile = p and (not q or r)
print(n, "->", estBissextile)
print("------------------------------")
```

```
2025 -> False
------------------------------
2020 -> True
------------------------------
3000 -> False
------------------------------
2000 -> True
------------------------------
```

3. Cette façon de coder le caractère bissextile est largement « idiomatique » ; c'est ainsi que la bibliothèque standard le code : `_is_leap`.

Alternative

Il aurait aussi été possible d'écrire un booléen `estNonBissextile` traduisant qu'une année `n` est non bissextile en disant qu'elle n'est pas multiple de 4 (traduction : `not p`) ou alors (traduction `or`) qu'elle est multiple de 100 et pas multiple de 400 (traduction : `q and not r`) puis en prenant la négation de `estNonBissextile` :

```
n=3000
p = (n % 4)==0
q = (n % 100)==0
r = (n % 400)==0
estNonBissextile =(not p) or (q and not r)
estBissextile =not estNonBissextile

print(n, "->", estBissextile)
```

Exercice type – 14 juillet

Une date est codée avec deux variables `j` et `m` où `j` représente le jour du mois et `m` le numéro du mois. Par exemple, le 2 novembre est codé avec `j = 2` et `m = 11`. On vous donne deux variables `j` et `m` représentant une date valide et vous devez construire une variable booléenne notée `avant` et qui vaut `True` si la date donnée tombe le 14 juillet ou les jours d'avant et `False` si elle tombe à partir du 15 juillet. Ci-dessous, quelques exemples d'exécution du programme demandé :

```
(10, 4) -> True
(15, 6) -> True
(10, 7) -> True
(14, 7) -> True
(20, 7) -> False
(2, 11) -> False
(25, 12) -> False
```

Solution

Pour savoir si une date `(j, m)` est antérieure au 14 juillet, on commence par regarder si le mois `m` vérifie `m < 7` et si c'est le cas, la date est bien antérieure ; sinon, on regarde si `m` vaut 7 et dans ce cas, la date sera antérieure selon que le jour sera ou pas, antérieur au 14. Dans tous les autres cas, la date n'est pas antérieure. D'où le code :

```
j = 10
m =4
avant = m<7 or m==7 and j<=14
print((j,m), "->", avant)

j = 15
m =6
avant = m<7 or m==7 and j<=14
print((j,m), "->", avant)
```

```
10
11 j = 10
12 m =7
13 avant = m<7 or m==7 and j<=14
14 print((j,m), "->", avant)
15
16 j = 14
17 m =7
18 avant = m<7 or m==7 and j<=14
19 print((j,m), "->", avant)
20
21 j = 20
22 m =7
23 avant = m<7 or m==7 and j<=14
24 print((j,m), "->", avant)
25
26 j = 2
27 m =11
28 avant = m<7 or m==7 and j<=14
29 print((j,m), "->", avant)
30
31 j = 25
32 m =12
33 avant = m<7 or m==7 and j<=14
34 print((j,m), "->", avant)
```

```
35 (10, 4) -> True
36 (15, 6) -> True
37 (10, 7) -> True
38 (14, 7) -> True
39 (20, 7) -> False
40 (2, 11) -> False
41 (25, 12) -> False
```

Exercice type – Mois de 31 jours

Cet exercice est résolu en vidéo : ICI.
On donne un numéro de mois `m` entre 1 et 12. Créer une variable booléenne `est_mois_31` (prenant comme valeur `True` ou `False`) qui teste si `m` est le numéro d'un mois ayant 31 jours [a] comme janvier (numéro 1) ou juillet (numéro 7) mais pas février (numéro 2).

a. Les mois ayant 31 jours sont : janvier (1), mars (3), mai (5), juillet (7), août (8), octobre (10), décembre (12).

Solution

On dispose de la liste des numéros de mois ayant 31 jours, donc il suffit de tester les uns après les autres le mois donné `m` avec les mois possibles :

```
1 m = 4
2 est_mois_31 = (m == 1 or m == 3 or m == 5 or
3                m == 7 or m == 8 or m== 10 or m== 12)
4
5 print(est_mois_31)
```

```
6 False
```

— Lignes 2-3 : une expression placée entre des parenthèse peut être exceptionnellement écrite sur plusieurs lignes, cela évite d'écrire de longues lignes.

On peut aussi essayer de donner une condition plus synthétique. Un mois de 31 jours placé avant août est de numéro 1, 3, 5 ou 7 et donc impair ; à l'inverse, un mois de 31 jours placé à partir d'août est de numéro 8, 10 ou 12 et donc pair. D'où le code suivant :

```
1 m = 9
2 est_mois_31 = ((m <= 7 and m % 2 == 1) or (m >= 8 and m % 2 == 0))
3 print(est_mois_31)
```

```
4 False
```

On pouvait aussi remarquer qu'il y a plus de mois à 31 jours et donc il est (légèrement) plus court d'écrire de passer par la négation :

```
1 m = 4
2 est_mois_31 = not (m==2 or m==4 or m==6 or m==9 or m==11)
3 print(est_mois_31)
```

```
4 False
```

EXERCICES

Egaux ou opposés

Ecrire une variable booléenne `egaux_ou_opp` qui à partir de deux entiers `x` et `y` teste si les deux nombres sont égaux ou opposés. Exemples de comportement :

x	y	egaux_ou_opp
5	6	False
42	42	True
42	-42	True

Nombres de même signe

Ecrire un booléen `memeSigne` qui vaut `True` si les nombres donnés `a` et `b` sont de même signe, et `False` dans le cas inverse. On considérera que 0 est aussi bien positif que négatif. Exemples de comportements :

```
42 17 -> True
42 0 -> True
-42 17 -> False
42 -17 -> False
-42 -17 -> True
-42 0 -> True
0 -42 -> True
0 0 -> True
```

Entiers qui se suivent

On vous donne deux variables `x` et `y` représentant des entiers. Définir une variable booléenne `ok` valant `True` si

— les entiers sont entre 1 et 4 ET

— ou bien sont consécutifs

— ou bien l'un vaut 1 et l'autre vaut 4.

Sinon, `ok` vaudra `False`.

Exemples

```
5 2 -> False
2 3 -> True
3 2 -> True
4 4 -> False
7 8 -> False
4 1 -> True
3 1 -> False
```

Trois entiers consécutifs

On donne trois entiers `a`, `b` et `c`. Construire un booléen `sontConsecutifs` valant True si les trois entiers, lorsque on les range dans l'ordre croissant, sont consécutifs.

Voici quelques exemples de comportements du programme :

```
5 6 7 -> True
6 5 7 -> True
5 8 7 -> False
7 6 5 -> True
```

Négatif, positif et nul

On donne trois entiers `a`, `b` et `c`. Construire un booléen `ok` qui vaut True si parmi les trois entiers il y a un entier strictement positif, un entier strictement négatif et le nombre zéro, sinon `ok` vaudra `False`. Voici quelques exemples de comportements :

```
a = -4 b = 3 c = 0 -> True
a = -2 b = 0 c = 3 -> True
a = 4 b = 5 c = 2 -> False
a = 0 b = 0 c = 0 -> False
a = -1 b = -2 c = 4 -> False
a = 0 b = 0 c = 0 -> False
```

Test de la non-égalité de cinq entiers

On donne cinq entiers a, b, c, d et e. Soit la condition C suivante : *les 5 entiers ne sont pas tous égaux*. Par exemple

— si a=5, b=5, c=5, d=4 et e=4 alors C est True

— si a=5, b=5, c=5, d=5 et e=5 alors C est False

Ecrire une variable booléenne qui vaut True si la condition C est vraie et False sinon.

Conditions sur un entier

Dans cette question, `x` est supposé être un entier. Créer une variable booléenne `cond` (prenant comme valeur `True` ou `False`) qui teste si `x` est ou bien multiple de 10 ou bien à la fois impair, non multiple de 3 et compris, au sens large, entre 42 et 421. Ainsi `x = 2020` passe le test, `x = 301` aussi, `x = 420` également mais pas `x = 81`.

Au moins un pair et au moins un supérieur à 100

On donne trois entiers positifs `a`, `b` et `c`. Construire un booléen `ok` qui vaut `True` si parmi les trois nombres, il y a au moins un nombre pair et aussi s'il y a au moins un nombre supérieur ou égal à 100. Sinon, `ok` vaut `False`

Exemples :

```
7 42 142 -> True
42 60 71 -> False
111 125 2015 -> False
80 111 31 -> True
7 41 142 -> True
```

Heures d'ouverture

Dans l'exercice, on supposera que

— les jours de la semaine sont codés à l'aide d'un entier `j` avec `j` = 1 pour lundi, `j` = 2 pour mardi, et ainsi de suite jusqu'à dimanche (`j` = 7)

— une heure est codée par une entier entre 0 (inclus) et 24 (exclu).

Un magasin est ouvert du lundi au vendredi de 8h à 18h et le samedi de 9h à 12h et il est fermé le reste du temps.

On donne un jour `j` et une heure `h` et on demande d'écrire un booléen `estOuvert` et qui vaut `True` si le magasin est ouvert le jour `j` à l'heure `h` et `False` sinon.

Voici quelques exemples de comportements du programme :

j	h	Ouvert ?
2	17	True
6	10	True
2	19	False
6	17	False

Vacances

On donne deux entiers `j` et `m` représentant une date valide où `m` représente le numéro du mois (entre 1 et 12) et `j` le jour du mois (entre 1 et 31). Ainsi,

— la donnée `j = 25` et `m = 4` représente le 25 avril

— le 31 septembre est représenté par `j = 31` et `m = 9`.

Imaginons que les vacances d'été aient lieu à partir du 6 Juillet et jusqu'au 2 septembre. On donne le jour `j` et le mois `m` d'une date et on vous demande de construire une variable booléenne `enVacances` valant True si la date correspond à un jour de vacances d'été et `False` sinon. Exemples :

```
j = 1 et m = 5 -> False
j = 6 et m = 7 -> True
j = 7 et m = 7 -> True
j = 14 et m = 7 -> True
j = 2 et m = 8 -> True
j = 31 et m = 8 -> True
j = 3 et m = 9 -> False
j = 4 et m = 9 -> False
j = 7 et m = 9 -> False
j = 8 et m = 12 -> False
```

Jours de repos

À un jour donné, Léa se repose si le jour n'est pas un jour ouvrable (lundi à vendredi) ou si elle est en vacances. On vous donne

— un booléen `ouvrable` qui vaut `True` si le jour donné est un jour ouvrable et `False` sinon,

— un booléen `vacances` qui vaut `True` si le jour donné est un jour de vacances et `False` sinon,

et on vous demande de définir un booléen `repos` dépendant des deux variables précédentes et qui vaut `True` si Léa se repose le jour donné et `False` sinon.

Tous ou uniquement un positifs

On donne trois entiers `a`, `b` et `c`. Écrire un booléen `ok` qui vaut `True` lorsque exactement un des trois entiers est strictement positif ou alors si les trois sont tous strictement positifs.

Voici quelques exemples de comportements du programme :

```
-5 12 3 -> False
0 1 5 -> False
-2 -8 -3 -> False
1 2 3 -> True
-2 6 -8 -> True
2 6 -8 -> False
-1 0 0 -> False
1 0 0 -> True
0 0 0 -> False
```

Exactement un positif

On donne deux entiers `a` et `b`. Construire un booléen `unSeulPositif` valant `True` si

— soit `a`

— soit `b`

est strictement positif et `False` sinon.

Voici quelques exemples de comportements du programme :

```
-5 12 -> True
1 5 -> False
-2 -8 -> False
1 -2 -> True
0 6 -> True
0 -6 -> False
0 0 -> False
```

Entier ayant n chiffres

On vous donne un entier $x > 0$ et un autre entier $n > 0$. Créer une variable booléenne `avoir_n_chiffres` (prenant comme valeur True ou False) qui teste si l'entier x admet exactement n chiffres dans son écriture en base 10. Par exemple, si $x = 421$ et $n = 2$, la condition vaut False et si $n = 3$, la condition vaut True. Pour trouver la condition, on pourra observer, en utilisant des puissances de 10, quel est le plus petit et le plus grand entier ayant n chiffres.

Entiers dans un rapport de 2 ou 3

On donne deux nombres entiers positifs `a` et `b` et on demande d'écrire une variable booléenne `double_ou_triple` qui vaudra `True` exactement si un des deux nombres est le double ou le triple de l'autre. Voici quelques exemples accompagnés d'une brève explication :

a	b	double_ou_triple	Explication
5	15	True	$b = 3a$
20	10	True	$a = 2b$
42	30	False	$a \neq 2b$ $a \neq 3b$ $b \neq 2a$ $b \neq 3b$
0	42	False	Idem
0	0	True	$b = 2a$

Triangle valide ou pas

Il n'existe pas de triangle de côtés de longueur 42, 81 et 32. En effet, le côté d'un triangle a toujours une longueur valant strictement moins que la somme des deux autres et, ici, on a $81 > 42 + 32$. Et inversement, si on se donne trois nombres positifs tels que chacun soit strictement inférieur à la somme des deux autres, alors il existe bien un triangle ayant ces nombres pour longueur de côtés.

On donne trois entiers `a`, `b` et `c` et on demande d'écrire, de préférence en une ligne de code, un booléen `estValide` qui vaut `True` si et seulement s'il existe un triangle de côtés de longeurs `a`, `b` et `c`, et `False` sinon.

Voici quelques exemples de comportement :

```
(-3, 4, 10) -> False
(42, 81, 32) -> False
(42, 81, 50) -> True
```

Intervalles disjoints ou pas

On donne deux intervalles réels `I` $=]$`a, b`$[$ et `J` $=]$`c, d`$[$ tels que : `a` $<$ `b` et `c` $<$ `d` (pour éviter les ambiguïtés, les extrémités des intervalles sont exclues). Ecrire un booléen `inter` qui vaut `True` si les intervalles `I` et `J` s'intersectent en au moins un point et `False` sinon. Voici un exemple de comportements du programme :

```
(2, 4), (3, 6) : True
(2, 3), (1, 4) : True
(5, 6), (1, 4) : False
(2, 4), (1, 2) : False
```

Le booléen `inter` peut s'écrire en une seule ligne et ne dépend que de `a`, `b`, `c` et `d`. On essayera plutôt de traduire que les intervalles sont **disjoints**, autrement dit d'intersection vide.

2 Instructions conditionnelles

Instruction `if`

L'instruction `if` permet de coder un schéma logique du type

si ... alors ...

Illustration

On veut coder un programme qui :

— définit une variable `x` référant un entier,
— affiche la valeur de `x`,
— teste si `x` est strictement négatif et, si c'est le cas, affiche :
  — le message : **x est négatif**,
  — l'opposé de `x` (c'est-à-dire `-x`),
— se termine en affichant le message **Fin du programme**.

Voilà ce que pourrait afficher le programme si la variable `x` dans le code-source vaut `10` :

```
-10
x est négatif
10
Fin du programme
```

Le programme sera intitulé `instruction_if.py`.

Implémentation en Python

Le programme `instruction_if.py` doit afficher un message et un nombre **si** une certaine condition est vérifiée. Pour cela, en Python, on utilise

— une instruction `if` pour exprimer l'idée de condition *si ... alors ...*
— un booléen pour exprimer la condition *si*.

Le problème posé admet la solution suivante :

`instruction_if1.py`

```
1 x = -10
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

```
9 -10
10 x est négatif
11 10
12 Fin du programme
```

Examinons maintenant le même code que `instruction_if1.py` où on a remplacé `x = -10` par `x = 42` en ligne 1 :

`instruction_if2.py`

```
1 x = 42
2 print(x)
```

```
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

```
9 42
10 Fin du programme
```

L'édition d'une instruction `if`

On reprend le code ci-dessus :

instruction_if1.py

```
1 x = -10
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

— Lignes 4-6 : une instruction conditionnelle. Elle utilise le mot-clé `if`.

— Ligne 4 : la condition testée est `x < 0`. Cette condition est toujours suivie du séparateur : (*deux-points*). La ligne qui s'étend du mot-clé `if` jusqu'au séparateur s'appelle l'*en-tête* de l'instruction `if`.

— Lignes 5-6 : le *corps* de l'instruction `if` est le bloc situé aux lignes 5 et 6. Ce corps, la plupart du temps :

  — commence **sous** l'en-tête (il faut donc sauter une ligne) ;

  — est indenté par rapport à l'en-tête.

Exécution d'une instruction `if`

On reprend les codes ci-dessus :

instruction_if1.py

```
1 x = -10
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

ainsi que

instruction_if2.py

```
1 x = 42
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

— Ligne 1 : très souvent, la condition de l'instruction `if` dépend d'une variable définie **avant** l'instruction `if`. Ici, la variable s'appelle x.

— Ligne 4 : lorsque l'exécution du code arrive face à une condition `if`, la condition `if` est testée : vaut-elle `True` ? vaut-elle `False` ?

— Lignes 5-6 :

  — si la condition vaut `True` (cf. `instruction_if1.py`), les instructions du **corps** de `if` sont exécutées

  — si la condition vaut `False` (cf. `instruction_if2.py`), les instructions du corps de `if` sont ignorées et l'exécution passe directement à la suite du code (ligne 8)

— Ligne 8 : Une fois l'instruction `if` exécutée, l'exécution continue normalement.

Indentation d'une instruction composée

Soit le code d'une instruction `if` typique :

```
1 x = -100
2
3 if x < 0:
4     print("x est negatif")
5     print(-x)
6
7 print("Fin du programme")
```

Le programme ci-dessus contient trois instructions :

— Ligne 1 : une instruction simple (affectation)

— Lignes 3-5 : l'instruction `if` qui est une instruction composée.

— Ligne 7 : une instruction simple (instruction d'affichage)

Ces trois instructions sont indentées avec la même valeur d'indentation qui est de 0 espace par rapport à la marge gauche du code.

On observe toutefois (lignes 4-5) que certaines lignes du code sont indentées.

Règles fondamentales de l'indentation

L'instruction `if` (lignes 3-5) est une instruction *composée* et son indentation doit respecter deux règles fondamentales :

— Règle 1 : si le **corps** de l'instruction `if` est placé sur plusieurs lignes, ce corps est indenté par rapport à la ligne d'en-tête ;

— Règle 2 : toutes les instructions dans le corps de l'instruction sont à la même indentation. Autrement dit, si par exemple la 1re instruction du corps de l'instruction `if` (ici ligne 4) est indentée de 4 espaces par rapport à l'en-tête, toutes les autres instructions du corps de l'instruction `if`, par exemple ici la ligne 5, seront aussi indentées de 4 espaces par rapport à l'en-tête.

Non-respect des règles

Règle 1

Le code ci-dessous ne respecte pas la règle 1 :

```
1 x = -100
2
3 if x < 0:
4 print("x est negatif")
5 print(-x)
6
7 print("Fin du programme")
```

— Lignes 4-5 : ces deux lignes devraient être indentées.

et l'exécution va générer un message d'erreur d'indentation au niveau de la ligne 4 :

```
1   File "if_erreur1_indentation.py", line 4
2     print("x est negatif")
3         ^
4 IndentationError: expected an indented block
```

— Lignes 8-11 : Le message d'erreur.

— Lignes 8 : Le nom du fichier-source [4] et le numéro de ligne à l'origine de l'erreur sont fournis.

— Lignes 9-10 : La ligne à l'origine de l'erreur et un marqueur (le signe circonflexe ^) signalant où l'erreur est rencontré dans la ligne.

— Ligne 11 : Le type d'erreur, ici `IndentationError` (*erreur d'indentation*) et une description sommaire de la raison de cette erreur.

Règle 2

Le code ci-dessous ne respecte pas la règle 2 :

`if_erreur2_indentation.py`

```
1 x = -100
2
3 if x < 0:
4      print("x est negatif")
5     print(-x)
6
```

4. En fait, cela dépend de l'interface d'utilisation de Python. Pour une feuille IPython/Jupyter, aucun nom de fichier n'est mentionné.

```
7 print("Fin du programme")
```

— Lignes 4-5 : ces deux lignes devraient être à un même niveau d'indentation. Or la ligne 4 est indentée de 5 espaces et la ligne suivante de 4 espaces au lieu de 5.

et l'exécution va générer un message d'erreur d'indentation au niveau de la ligne 5 :

if_erreur2_indentation.py

```
1   File "if_erreur2_indentation.py", line 5
2     print(-x)
3             ^
4 IndentationError: unindent does not match any outer indentation level
```

— Ligne 10 : Le message explique que l'indentation n'est pas uniforme. L'indentation de la ligne 4 est acceptée pas celle de la ligne 5.

Noter que la règle d'indentation s'applique aussi à une instruction `if` emboîtée :

```
1 n = 3000
2 if n % 10 == 0:
3     q = n // 10
4     if q % 10 == 0:
5         print("n est multiple de 100")
```

— Lignes 4-5 : instruction composée emboîtée.

Instruction if/else

L'instruction if/else permet de coder un schéma du type

si ... alors ... sinon ...

Soit un programme qui à partir d'un nombre x affiche *plus* si $x > 42$ et qui sinon affiche *moins*. En Python, cela donne :

if_else.py

```
1 x=10
2 if x > 42:
3     print("plus")
4 else:
5     print("moins")
6
7 print("FIN")
```

```
8 moins
9 FIN
```

— Ligne 2 : comme dans une instruction `if` simple, la condition est testée

— Lignes 2-5 : l'instruction `if/else`

— Ligne 4 : si la condition N'est PAS vérifiée,

  — la partie sous le `if` N'est PAS examinée

  — le bloc sous le `else` est exécuté

— Ligne 7 : l'exécution du programme continue alors normalement.

Inversement, si la condition suivant le `if` est vérifiée la partie `else` n'est pas examinée :

if_else_bis.py

```
1 x=50
2 if x > 42:
3     print( "plus")
4 else:
5     print("moins")
6
7 print("FIN")
```

```
8 plus
9 FIN
```

— Ligne 2 : ici, la condition `x > 42` est vérifiée donc seule la ligne 3 est examinée

— Lignes 4-5 : ces lignes ne sont pas examinées durant l'exécution.

Le mot `else` doit être compris comme *sinon*. Une instruction `if/else` propose une **alternative**. Ainsi, le programme `if_else.py` doit être compris comme codant l'alternative suivante :

— si $x > 42$ alors le programme affiche **plus**

— sinon, le programme affiche **moins**

Le mot `else` est un mot-clé du langage Python. Dans le cadre d'une instruction `if/else`, le mot `else` doit toujours être suivi du signe deux-points (**:**). Sous la ligne figurant `else` se trouve un bloc indenté. La ligne contenant `else` ainsi que le bloc indenté forment ce qu'on appelle une *clause* `else` de l'instruction `if`. Le *corps* de la clause `else` est le bloc indenté. Ce corps peut être constitué de n'importe quel type d'instructions.

C'est une erreur courante chez les débutants de vouloir placer une condition après le `else`, par exemple `else x<42`. Une clause `else` est toujours suivie immédiatement du signe `:`.

Quand on écrit une instruction if/else, prendre garde que le mot if et le mot else soient à la même indentation et rien d'autre ne doit être à la même indentation entre `if` et `else`.

Le terme d'instruction `if/else` est un abus de langage. En fait, il s'agit d'une instruction `if` contenant une clause `else`.

`if/else` pour un booléen : complication inutile

On rencontre souvent des situations où on doit créer une variable booléenne, disons `flag`, qui vaut `True` si une condition est vraie et `False` sinon.

Par exemple, on se donne un entier `n` et `flag` doit évaluer si `n` est pair ou pas. Un code courant est alors le suivant :

```
n=42

if n%2==0:
    flag=True
else:
```

```
    flag=False

print(flag)
```

```
True
```

Mais ce code est maladroit et à éviter. En effet, `flag` a toujours la même valeur que la condition `n%2==0`, donc un code équivalent est

```
n=42
flag = n%2==0

print(flag)
```

```
True
```

C'est un idiome à connaître et valable dans deux nombreux langages de programmation.

Indentation d'une instruction composée avec clause

Une instruction composée est constituée au minimum d'un en-tête et d'un corps, typiquement :

```
1 if 42 > 421:
2     print("hello!")
3     print("OK!")
```

— Ligne 1 : l'en-tête

— Lignes 2-3 : corps composé de deux instructions.

Mais, certaines instructions composées peuvent être composées aussi d'une, voire de plusieurs, « clauses ».

Le cas suivant d'une instruction if/else est typique de la structure générale d'une instruction composée avec clause :

```
1 x=50
2 if x > 42:
3     print( "plus")
4     if x > 100:
5         print("plus plus")
6 else:
7     print(x)
8     print("moins")
9
10 print("FIN")
```

— Lignes 2-8 : une instruction `if`

— Lignes 6-8 : une clause else de l'instruction `if`

— Lignes 3-5 : le corps de l'instruction if. Ce corps est constitué de deux instructions : une instruction simple (ligne 3) et une instruction composée (if, lignes 4-5)

— Lignes 6-8 : le corps de la clause else.

Les clauses font partie de l'instruction composée. Chaque clause contient un en-tête placé sur une seule ligne et qui se termine toujours par le signe deux-points. Le signe deux-points est suivi d'un ensemble d'instructions appelé « corps » de la clause.

Règles d'indentation

Soit une instruction composée contenant une ou plusieurs clauses. Alors,

- l'en-tête de l'instruction composée et chacune des clauses sont sur des lignes différentes et à la même indentation ;
- les instructions figurant dans le corps de l'instruction composée sont à la même indentation ;
- les instructions figurant dans le corps de n'importe quelle clause de l'instruction composée sont à la même indentation ;
- le corps de l'instruction composée ou de l'une de ses clauses peut contenir des instructions de n'importe quel type, simples ou composées. Ces instructions suivent les règles générales de l'indentation en Python.

Instructions `if` imbriquées

Le corps d'une instruction `if` peut être n'importe quelle instruction, en particulier une nouvelle instruction `if`.

Exemple :

```
n = 3000
if n % 10 == 0:
    q = n // 10
    if q % 10 == 0:
        print(n, "est multiple de 100")
```

```
3000 est multiple de 100
```

Une instruction `if` est une instruction multiple. Le corps d'une instruction `if` peut être de taille quelconque et contenir tout type d'instructions.

Dans certains cas, deux instructions `if` imbriquées s'écrivent plus simplement avec **une seule** instruction `if` et une expression `and`. En nous basant sur l'exemple précédent, le code

```
n = 3000
if n % 10 == 0:
    if (n // 10) % 10 == 0:
        print(n, "est multiple de 100")
```

```
3000 est multiple de 100
```

sera plutôt ré-écrit ainsi :

```
n = 3000
if n % 10 == 0 and (n // 10) % 10 == 0:
        print(n, "est multiple de 100")
```

```
3000 est multiple de 100
```

Instruction `if/elif`

L'instruction `if/elif` permet de coder un schéma du type

si ... alors ... sinon si ...

Voici un exemple typique d'utilisation d'une instruction `if/elif`

if_elif.py

```
1 m = 2
2 print(m)
3
4 if m == 1:
5     print("Or")
6 elif m == 2:
7     print("Argent")
8 elif m == 3:
9     print("Bronze")
10
11 print("bonjour !")
```

```
12 2
13 Argent
14 bonjour !
```

L'exécution est la suivante :

- Au départ, `m` vaut 2 et la valeur de `m` est affichée
- Le programme passe à la ligne 4 et teste (à cause du `if`) si `m` vaut 1.
- Comme ce n'est pas le cas, le programme passe à la ligne 6 (à cause du `elif`) et teste si `m` vaut 2.
- Comme c'est le cas, la ligne 7 est exécutée et **Argent** est affiché.
- Les lignes 8-9 sont alors ignorées.
- Finalement, l'exécution passe à la ligne 11.

Le mot `elif` est un mot-clé du langage Python.
Une clause `elif` (lignes 6-7 ou encore 8-9) est constituée de deux parties :

- un en-tête placé sur une ligne et terminé par deux-points
- un corps, le plus souvent indenté par rapport à l'en-tête.

L'en-tête contient obligatoirement une condition qui sera testée à l'exécution.
Le mot-clé `elif` d'une instruction `if` est toujours aligné verticalement avec le mot-clé `if`.
Les clauses `elif` constituent des blocs de code qui sont à la même indentation que le corps de l'instruction `if`.
`elif` est un mot-valise construit de la manière suivante :

elif = else + if.

Il faut comprendre `elif` comme voulant signifier *sinon si*.

On dit que les parties du code contenant un bloc `elif` sont des *clauses* `elif`. Parler d'instruction `if`/`elif` est un abus de langage, il s'agit en fait d'une instruction `if` contenant une ou plusieurs clauses `elif`.

Suite de `if` vs suite de `if`/`elif`

Comparons une suite d'instructions `if`/`elif` à une suite d'instructions où `elif` est remplacé par `if`.

Par exemple, comparons les codes suivants :

if_elif.py

```
1 m = 1
2 print(m)
3
4 if m == 1:
5     print("Or")
6 elif m == 2:
7     print("Argent")
8 elif m == 3:
9     print("Bronze")
10
11 print("bonjour !")
```

suite_if.py

```
12 m = 1
13 print(m)
14
15 if m==1:
16     print("Or")
17 if m==2:
18     print("Argent")
19 if m==3:
20     print("Bronze")
21
22 print("bonjour!")
```

Les deux codes affichent la même chose :

```
1 1
2 Or
3 bonjour !
```

Pourtant, les deux exécutions de code ne sont pas équivalentes :

— code `if_elif.py` : les conditions lignes 6 et 8 ne sont même pas examinées car les lignes 4-5 sont exécutées ;

— code `suite_if.py`, les trois conditions (lignes 15, 17 et 19) sont **toujours** testées ce qui n'est pas pertinent puisque les différentes conditions testées s'excluent mutuellement.

Il se pourrait d'ailleurs que dans certains cas, les affichages ne soient pas identiques. Pour cela comparer les deux codes suivants.

Premier code

```
1 m=0
2 if m>=0:
3     print("m est positif")
4 if m<=0:
5     print("m est negatif")
```

```
6 m est positif
7 m est negatif
```

— Lignes 2-3 : l'affichage est exécuté car `0 >= 0` est `True`

— Lignes 4-5 : l'affichage est exécuté car `0 <= 0` est aussi `True`

— Lignes 6-7 : deux lignes sont affichées

Deuxième code :

```
1 m=0
2 if m>=0:
3     print("m est positif")
4 elif m<=0:
5     print("m est negatif")
```

```
6 m est positif
```

— Lignes 2-3 : l'affichage est exécuté car `0 >= 0` est `True`

— Ligne 4 : la condition n'est pas testée car la condition ligne 2 a été testée

— Ligne 6 : une seule ligne est affichée.

Les conditions `if/elif`

Exclusion mutuelle ?

Soit le code `if_elif.py` :

`if_elif.py`

```
1 m = 2
2 print(m)
3
4 if m == 1:
5     print("Or")
6 elif m == 2:
7     print("Argent")
8 elif m == 3:
9     print("Bronze")
10
11 print("bonjour !")
```

Les différentes conditions `if`/`elif` du code `if_elif.py` (ici `m == 1`, `m == 2`, `m == 3`) s'excluent mutuellement ce qui est le cas le plus fréquent en pratique ; mais ce n'est pas obligatoire. Par exemple, soit l'extrait de code suivant qui teste le signe d'un nombre `x` :

```
1 if x>=0:
2     print("positif")
3 elif x<= 0:
4     print("negatif")
```

Ici, les conditions `x>=0` et `x<=0` ne sont pas incompatibles puisque `x=0` vérifie les deux. Si on exécute le code avec `x=0`, on obtient :

```
1 x=0
2 if x>=0:
3     print("positif")
4 elif x<= 0:
5     print("negatif")
```

```
6 positif
```

Comme la condition ligne 2 est vérifiée, la ligne 4 n'est pas testée.

Cas où aucun corps `if`/`elif` n'est exécuté

Il se peut d'autre part que le corps d'AUCUNE instruction `if` ou `elif` ne soit exécuté, par exemple :

```
1 m=5
2 if m==1:
3     print ("Or")
4 elif m==2:
5     print("Argent")
6 elif m==3:
7     print("Bronze")
8
9 print("bonjour !")
```

```
10 bonjour !
```

— La ligne 2 est testée.
— Comme la condition est fausse, la ligne 3 est sautée
— L'instruction `elif` ligne 4 est testée. A nouveau, elle est fausse
— Donc, l'instruction `elif` de la ligne 6 est testée. Comme elle est encore fausse, la ligne 7 est sautée et aucun nom de médaille n'est affiché.
— L'exécution passe à la ligne 9.

Instruction `if`/`elif`/`else`

L'instruction `if`/`elif`/`else` correspond au schéma suivant :

si ... alors ... sinon si ... alors sinon ....

Il est possible d'associer à une instruction `if`

— un ou plusieurs `elif`,

— un `else` *unique*.

Voici un exemple basique

```
1 m=5
2 if m==1:
3     print("Or")
4 elif m==2:
5     print("Argent")
6 elif m==3:
7     print("Bronze")
8 else :
9     print("Sans médaille")
10
11 print("FIN")
```

```
12 Sans médaille
13 FIN
```

— Ligne 1 : l'exécution de l'instruction `if`/`elif`/`else` aux lignes 2-9 dépend de la valeur initiale de `m`.

— Lignes 2-7 : les conditions ligne 2, ligne 4 et ligne 6 sont testées dans cet ordre tant qu'elles sont fausses. Le corps de la première (éventuelle) condition vraie est exécuté.

— Lignes 8-9 : si aucune des conditions n'est vraie alors le corps du `else` (ligne 9) est exécuté. C'est le cas ici puisque `m = 5` (cf. la ligne 1).

La clause `else` est unique et doit apparaître **après** les clauses `elif`.

A la différence d'une instruction `if` simple ou d'une instruction `if`/`elif`, dans une instruction `if`/`elif`/`else`, comme dans une instruction `if`/`else`, au moins un des corps de l'instruction sera exécutée.

Exercice type – Le plus grand, le plus petit

Cet exercice est corrigé en vidéo : ICI
On donne deux entiers `a` et `b`. Ecrire un code qui calcule le plus petit et le plus grand des deux entiers `a` et `b`. Les valeurs calculées seront placées dans des variables `mini` et `maxi`.

Solution

Il suffit de comparer les deux nombres pour savoir qui est `mini` et qui est `maxi`, d'où le code :

```
1 a = 42
2 b = 17
3
4 if a < b:
5     mini=a
6     maxi=b
7 else:
8     mini=b
```

```
    maxi=a

print(a, b)
print("mini =", mini, "maxi =", maxi)
```

```
42 17
mini = 17 maxi = 42
```

Il est intéressant de remarquer que des variables `mini` et `maxi` sont créées à la fin du code sans que pour autant on ait **en tout début de code** (comme pour `a` et `b`) une définition par affectation de ces variables : dans le corps d'une instruction `if` on peut placer n'importe quel type d'instruction qu'on aurait placée en début de code.

On pouvait aussi écrire le code suivant qui évite le `else` :

```
a = 42
b= 17

maxi=a
mini=b

if a < b:
    mini=a
    maxi=b

print(a, b)
print("mini =", mini, "maxi =", maxi)
```

Exercice type – Régler un montant avec des coupures de même valeur

On vous donne la valeur `b` d'un billet en euros, par exemple `b = 10` et un montant `m`, nombre entier représentant un montant en euros à régler, par exemple, `m = 8181`. Déterminer le nombre minimum `N` de billets de `b` euros pour être en mesure de régler le montant `m`.
Par exemple,

— si `b = 10` et `m = 8181` alors `N = 819`

— si `b = 10` et `m = 800` alors `N = 80`

Solution

Il y a deux cas selon que le montant peut être réglé de manière exacte avec des billets de montant `b`. Par exemple, si `b=10` et `m=8181`, le montant ne peut être réglé de manière exacte (sans rendu de monnaie). Le nombre exact de billets est clairement 819 car avec 818 billets on peut régler jusqu'à 8180 euros (et il manque 1 euro) et avec un billet de plus, on peut régler le montant (et il restera neuf euros) ; ce dernier nombre (819) est une unité de plus que le quotient entier de `m` par `b`. D'où le code :

```
b= 10
m=8181
N=m//b
r=m%b
```

```
5 if r!=0:
6     N=N +1
7 print("b = ", b, "m =", m, "->", N)
8 print("->", N)
9
10 print()
11
12 b= 10
13 m=800
14 N=m//b
15 r=m%b
16 if r!=0:
17     N=N +1
18 print("b = ", b, "m =", m, "->", N)
19 print("->", N)
```

```
20 b =  10 m = 8181 -> 819
21 -> 819
22
23 b =  10 m = 800 -> 80
24 -> 80
```

— Lignes 16 et 17 : s'il est possible de payer *exactement* la somme `m` avec des billets `b` alors la condition ligne 16 est fausse et c'est `N` qui donne la réponse attendue.

On peut aussi utiliser `else` :

```
if m%b!=0:
    N=m//b +1
else:
    N=m//b
```

EXERCICES

Ordonner trois entiers

On donne trois entiers `a`, `b` et `c` et on demande d'écrire un programme qui affiche ces trois entiers dans l'ordre croissant.

Nombre d'éléments distincts

On donne trois variables `a`, `b`, `c` référant des entiers et on demande de donner le nombre `N` de valeurs distinctes parmi les trois.

Voici quelques exemples d'affichages possibles

```
a = 1, b = 1, c = 1 -> N = 1
a = 2, b = 4, c = 3 -> N = 3
a = 2, b = 2, c = 3 -> N = 2
```

Afficher l'écart entre deux nombres

On donne deux entiers `a` et `b` et on demande d'afficher un message comme ci-dessous :

```
L'écart entre a et b est de e
```

où a, b et e sont remplacées par les valeurs adaptées. Voici quelques exemples de comportement :

```
82 , 42 -> L'écart entre 82 et 42 est de 40
1970 , 2038 -> L'écart entre 1970 et 2038 est de 68
42 , 42 -> L'écart entre 42 et 42 est de 0
42 , -10 -> L'écart entre 42 et -10 est de 52
```

Nombre intermédiaire

Cet exercice est corrigé en vidéo : ICI

On donne trois entiers a, b et c, par exemple $a = 42$, $b = 100$ et $c = 10$. On demande de déterminer et d'afficher le nombre qui est encadré par les deux autres. Dans l'exemple précédent, on a $c \leq a \leq b$ donc le nombre demandé est $a = 42$.

Dans un deuxième temps, on pourra essayer de chercher un code qui minimise le nombre de comparaisons.

Date valide (`if\elif`)

On donne une date en utilisant deux variables `jour` et `mois` deux entiers strictement positifs et on demande d'écrire un booléen `estDateValide` qui dise si la date `jour/mois` définit une date valide du calendrier. Par exemple :

Jour, mois	Valide	Explication
`18, 3`	True	18 mars
`31, 12`	True	31 décembre
`35, 3`	False	un mois n'a pas 35 jours
`31, 11`	False	pas de 31 novembre

Pour simplifier, on supposera que le mois de février a 28 jours.

Indication : utiliser une condition testant si le mois est en 31, 30 ou 28 jours.

Placer entre deux nombres

On donne deux entiers `a` et `b`. On donne un troisième entier `x`. Ecrire un programme qui détermine l'entier `y` suivant :

— si `x` est plus petit que `a` et `b` alors `y` est le plus petit des deux entiers `a` et `b`

— si `x` est plus grand que `a` et `b` alors `y` est le plus grand des deux entiers `a` et `b`

— dans les autres cas, `y` vaut `x`

Voici quelques exemples de comportement du programme en fonction de a, b et x :

```
42 10 33 -> 33
10 42 33 -> 33
42 10 0 -> 10
10 42 0 -> 10
42 10 30 -> 30
10 42 60 -> 42
```

ensureRange

On donne trois nombres `a`, `b` et `x`. On demande d'écrire une variable `z` valant l'élément le plus proche de `x` situé dans l'intervalle d'extrémités `a` et `b`. Voici quelques exemples de comportements :

```
a = 0, b = 5, x = 2 -> 2
a = 5, b = 0, x = 2 -> 2
a = 0, b = 5, x = 5 -> 5
a = 5, b = 0, x = 5 -> 5
a = 0, b = 5, x = 6 -> 5
a = 5, b = 0, x = 6 -> 5
a = 0, b = 5, x = -1 -> 0
a = 5, b = 0, x = -1 -> 0
a = 0, b = 0, x = 5 -> 0
a = 0, b = 0, x = -1 -> 0
a = 0, b = 0, x = 0 -> 0
```

Équation du second degré

L'exercice consiste à écrire un code permettant de résoudre dans l'ensemble des réels une équation du second degré $ax^2 + bx + c = 0$. On supposera que a, b, c sont des entiers, que a est non nul et que l'inconnue x est un nombre réel.

Rappels : résolution de l'équation du second degré

On rappelle la résolution de l'équation. On calcule $\Delta = b^2 - 4ac$. Il y a alors trois cas :

— *1er cas* : $\Delta > 0$. L'équation admet deux solutions réelles, données par les formules

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a} \quad et \quad x_2 = \frac{-b + \sqrt{\Delta}}{2a}.$$

— *2e cas* : $\Delta = 0$. L'équation admet une seule solution réelle, $x = \frac{-b}{2a}$, donnée par n'importe laquelle des deux formules du 1er cas.

— *3e cas* : $\Delta < 0$. L'équation n'admet aucune solution réelle.

(1) Écrire un code Python qui détermine le nombre de solutions d'une équation du second degré.

(2) Affiner le code précédent pour résoudre complètement dans les réels l'équation $ax^2 + bx + c = 0$ et afficher les solutions éventuelles sous forme de nombres flottants (et pas de fractions). Tester les trois équations suivantes :

Equation	Solutions
$6x^2 - 5x + 1 = 0$	$1/2$ et $1/3$
$4x^2 - 12x + 9 = 0$	$3/2$
$6x^2 + 7x + 7 = 0$	Aucune

Jeu à deux nombres

Dans cet exercice, on appellera *combinaison* la donnée de deux entiers a et b. Ainsi, on peut parler de :

— la combinaison 42, 17 ;

— la combinaison 81, 81.

À partir d'une combinaison, on a les règles suivantes :

— Si a=b, la combinaison rapporte 10 points.

— Si a et b sont consécutifs (comme a=5 et b=4), alors la combinaison rapporte 3 points.

— sinon la combinaison ne rapporte rien.

Écrire un code Python qui, étant donné une combinaison de deux entiers a et b, affiche le nombre de points que rapporte la combinaison.

Combien de pairs et d'impairs ?

On donne 4 entiers `a`, `b`, `c` et `d`. Construire des variables `pairs` et `impairs` qui comptent le nombre d'entiers pairs et le nombre d'entiers impairs parmi les 4.

Exemples de comportement :

```
a = 15, b = 10, c = 12, d = 13
pairs = 2
impairs = 2

a = 18, b = 10, c = 14, d = 12
pairs = 4
impairs = 0

a = 15, b = 10, c = 11, d = 13
```

```
pairs = 1
impairs = 3
```

On pourra introduire un variable `pairs` valant initalement 0 et référençant le nombre d'éléments pairs.

Répartion équitable de pièces de monnaie

Soit un ensemble de pièces de monnaie constitué de `N1` pièces de 1 euro et `N2` pièces de 2 euros. On se demande s'il est possible de répartir ces pièces entre deux personnes **A** et **B** en sorte que chacune reçoive exactement le même montant (voir les exemples ci-dessous).

1. Définir un booléen `possible` valant `True` ou `False` selon que la répartition est possible ou non. Voici quelques exemples de comportement (et qui traduisent en fait tous les cas possibles) :
   - si `N1 = 81` et `N2 = 42` alors la répartition est impossible
   - si `N1 = 42` et `N2 = 100` alors la répartition est possible
   - si `N1 = 42` et `N2 = 11` alors la répartition est possible
   - si `N1 =0` et `N2 = 42` alors la répartition est impossible.
2. Lorsque la répartition est possible, déterminer
   - les nombres `A1` et `B1` de pièces de 1 euro reçues par A et B
   - les nombres `A2` et `B2` de pièces de 2 euros reçues par A et B
   - un booléen `verif` qui vérifie la validité de la répartition.

   Voici quelques exemples :
   - si `N1 = 42` et `N2 = 100` alors la répartition est possible avec `A1 = 21`, `A2 = 50`, `B1 = 21` et `B2 = 50` puisque le montant total est de 242 euros et A a reçu 21+100 = 121 euros, B en a reçu 21+100=121 euros et qui en est la moitié ;
   - si `N1 = 42` et `N2 = 11` alors la répartition est possible avec `A1 = 22`, `A2 = 5`, `B1 = 20` et `B2 = 6` puisque le montant total est de 64 euros et A a reçu 22+10 = 32 euros, B a reçu 20+12 euros qui en est la moitié.

   L'idée de exercice est inspirée d'un problème de Codeforces.

Articles dans un magasin

(*D'après un exercice du Cours de Caml de mon collègue Thierry Montaut*)

Un article est vendu :

- 10 euros pièce quand on en achète 10 ou moins
- 8 euros entre le 11^{e} et le 50^{e}
- 6 euros à partir du 51^{e}.

Soit `N` le nombre d'articles achetés. Ecrire une variable `prix` donnant le prix de la commande en fonction de `N`.

Exemples :

- pour N=42, on trouvera un prix de 356 euros. En effet $10 \times 10 + 32 \times 8 = 100 + 256 = 356$
- pour N=120, on trouvera un prix de 840 euros. En effet $10 \times 10 + 40 \times 8 + 70 \times 6 = 100 + 320 + 420 = 8$

Catégories en sports

Le tableau ci-dessous définit les catégories de participants en athlétisme en fonction de l'âge, donné en années (d'après Wikipédia) :

Catégorie	**Début (années)**	**Fin (années)**
Eveil		9
Poussin	10	11
Benjamin	12	13
Minime	14	15
Cadet	16	17
Junior	18	19
Espoir	20	22
Sénior	23	39
Vétéran	40	

Écrire une programme qui à partir d'un âge donné sous forme d'entier positif détermine une variable chaîne de caractères représentant la catégorie correspondante.

Quelle saison sommes-nous ?

Le tableau ci-dessous donne les dates de début des quatre saisons :

Saison	Début
printemps	20 mars
été	21 juin
automne	22 septembre
hiver	21 décembre

Une date est donnée par deux variables `j`, le jour du mois et `m`, le numéro de mois entre 1 et 12. On donne une date et on demande de déterminer la saison correspondant à cette date. On définira une variable nommée `saison`.

Voici quelques exemples de comportement :

```
j =  1, m =  1 : saison = hiver
j =  1, m =  3 : saison = hiver
j = 13, m =  3 : saison = hiver
j = 20, m =  3 : saison = printemps
j = 21, m =  3 : saison = printemps
j = 20, m =  6 : saison = printemps
j = 21, m =  6 : saison = été
j = 16, m =  7 : saison = été
j =  1, m =  9 : saison = été
j = 21, m =  9 : saison = été
j = 22, m =  9 : saison = automne
j = 23, m =  9 : saison = automne
j = 11, m = 11 : saison = automne
j = 21, m = 12 : saison = hiver
j = 25, m = 12 : saison = hiver
```

```
j = 31, m = 12 : saison = hiver
```

Catégoriser un triangle

On donne un triplet de nombres entiers positifs, `A`, `B`, `C`, représentant les angles en degrés d'un éventuel triangle, par exemple

```
A = 42, B = 68, C = 70
```

On rappelle que la somme des angles d'un triangle vaut 180°.

Un triangle est dit *isocèle* s'il admet deux angles égaux et on dit qu'il est *équilatéral* si ses trois angles sont égaux. Un triangle est *rectangle* si un de ses angles vaut 90°

Ecrire une variable `natureTriangle` qui à partir des angles aux sommets `A`, `B` et `C` d'un éventuel triangle s'évalue en la chaîne suivante :

— `"invalide"` s'il existe aucun tel triangle,

— `"equilatéral"` si le triangle `ABC` est équilatéral,

— `"isocèle"` si le triangle `ABC` est isocèle,

— `"rectangle"` si le triangle `ABC` est rectangle,

— `"rectangle-isocèle"` si le triangle `ABC` est rectangle-isocèle,

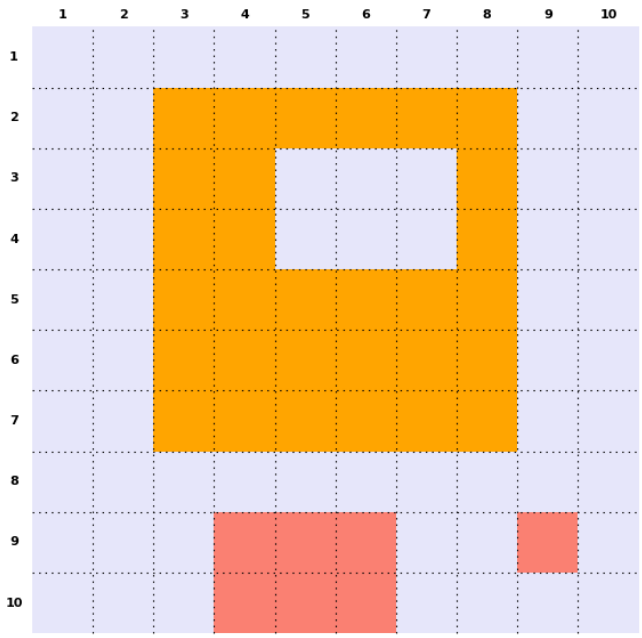
— `"quelconque"` sinon.

Voici quelques exemples de comportement :

```
42 42 33 → invalide
40 60 80 → quelconque
40 40 100 → isocèle
50 40 90 → rectangle
45 45 90 → rectangle-isocèle
60 60 60 → équilatéral
```

Zones de couleurs

Soit la figure ci-dessous :



qui montre un quadrillage avec des zones de couleur lavande, saumon ou orange. On donne une case par son numéro de ligne `lig` et son numéro de colone `col` et on veut connaître la couleur de la case. Ecrire un code Python qui définisse une variable `couleur` valant une chaîne de caractères parmi :

`"lavande"`, `"saumon"`, `"orange"` ou `"aucune"`

selon la position de la case.

Voici quelques exemples de comportements :

```
lig = 8, col = 3 : lavande
lig = 10, col = 10 : lavande
lig = 3, col = 6 : lavande
lig = 5, col = 5 : orange
lig = 10, col = 6 : saumon
lig = 9, col = 9 : saumon
lig = 42, col = 81 : aucune
```

Cet exercice est directement inspiré d'un exercice de France-IOI intitulé zones de couleurs.

Opérateur xor

Étant donné deux assertions `p` et `q`, construire une expression `xor_p_q` qui vaut

— `True` si **exactement** une des deux assertions `p` ou `q` vaut `True`

— `False` sinon.

Par exemple, dans le code ci-dessous :

```
p = 2 * k + 1 >= 100
q = k % 10 == 0
```

l'affichage sera le suivant :

```
k = 42, p = False, q = False -> xor_p,q = False
k = 81, p = True, q = False -> xor_p,q = True
k = 20, p = False, q = True -> xor_p,q = True
k = 420, p = True, q = True -> xor_p,q = False
```

On pourra envisager différents types de solution :

— utilisant des opérateurs booléens
— utilisant `if/else`
— n'utilisant ni booléens ni `if/else`

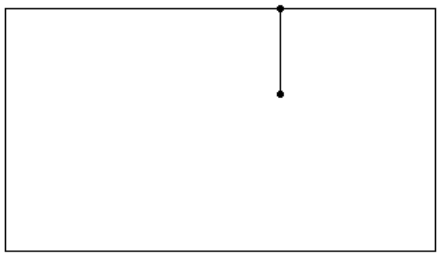
Généraliser à trois assertions `p`, `q`, `r` : `xor_p_q_r` vaut `True` si et seulement si exactement une seule des trois assertions vaut `True`.

Envisager une solutions basée sur la propriété suivante : dans un contexte arithmétique, `True` est converti en 1 et `False` en 0, par exemple, une expression comme `4*True - 3*False + 10` s'évalue en 14.

Distance d'un point à un rectangle

Dans le plan rapporté à deux axes perpendiculaires, on donne un rectangle $ABCD$, donné par ses 4 cotés d'équations **x = gauche**, **x = droite**, **y = bas** et **y = haut**. Soit un point M de coordonnées (x, y) intérieur au rectangle $ABCD$.

(1) Écrire un programme qui calcule la distance la plus courte de M aux côtés du rectangle $ABCD$.
(2) Faites une représentation graphique, en dessinant le segment représentant la distance minimale trouvée



Transport en bus

Un bus peut contenir p passagers. Déterminer le nombre `k` de bus nécessaires au transport de n passagers. Par exemple, si `p=50` alors si `n = 500` on aura `k = 10` et si `n = 542` on aura `k = 11`.

Ecrire un code qui, partant de `p` et de `n`, détermine la variable `k` puis vérifier sur différents exemples. On utilisera une instruction `if`, avec ou sans `else`, au choix.

Facteur minimal pour dépasser un entier donné

Soient des entiers `n`, `d` > 0. On cherche le plus petit entier `N` ≥ 0 tel que `Nd` $\geq$ `n`. Exemples avec `d` $= 10$:

— si n $= 42$ alors N $= 5$ puisque $40 <$ n mais $50 \geq$ n ;
— si n $= 40$ alors N $= 4$ puisque $30 <$ n mais $40 \geq$ n.

Ecrire un code qui détermine N en fonction de n et d.

On pourra remarquer que N se déduit facilement du quotient entier de n par d.

Reprographie au format grand livret

Une feuille au format A3 peut contenir côte à côte deux feuilles au format A4. Reproduire un ouvrage de n pages (format A4) en mode livret consiste à reproduire les pages en en plaçant 4 par feuille, chaque feuille étant au format A3 les feuilles étant imprimées, autant que possible, en recto-verso. Par exemple, une reprographie de 42 pages nécessite dans ce format 11 feuilles A3, les 10 premières feuilles sont complètement remplies (cela fait 40 pages) et la onzième n'est remplie qu'à moitié (il y aura deux pages blanches de taille A4 sur cette dernière feuille).

On donne un nombre n de pages et on demande le nombre N de feuilles pour effectuer une impression de ces n pages. Par exemple, si $n = 42$ alors $N = 11$.

Durée de sommeil

On donne une heure de la journée, qui correspond à l'heure d'endormissement, et une autre heure dans la journée, qui correspond à l'heure de réveil. On suppose que la personne ne dort jamais plus de 24 heures. Chacune des heures de la journée est donnée par deux variables, une pour le nombre d'heures et l'autre pour le nombre de minutes. Par exemple, `he = 23` et `me = 48` traduit une heure d'endormissement de `23 h 48`. Les variables seront `he` et `me` pour le moment d'endormissement, `hr` et `mr` pour le moment de réveil.

On demande d'écrire un programme qui calcule la durée de sommeil avec deux variables `hs` (nombre d'heures) et `ms` (nombre de minutes). Voici quelques exemples de comportement du programme :

```
Dort : 23 h 48
Lève : 7 h 42
Sommeil : 7 h 54
----------------
Dort : 22 h 48
Lève : 11 h 52
Sommeil : 13 h 4
----------------
Dort : 23 h 48
Lève : 23 h 59
Sommeil : 0 h 11
----------------
Dort : 2 h 48
Lève : 9 h 42
Sommeil : 6 h 54
----------------
Dort : 18 h 59
Lève : 7 h 0
Sommeil : 12 h 1
```

On fera le calcul de la manière suivante : on regarde si minuit est entre les deux heures ou pas :

— si c'est le cas, on rajoute à l'heure de réveil l'écart à minuit ; par exemple, si le coucher est

à 22h48 et le lever à 11h52, l'écart à minuit est de 1 heure et 12 minutes qui rajouté à 11h52 donne 13 h 4 min ;

— sinon, on regarde si on peut faire la différence des minutes et, si ce n'est pas le cas, on retient 60 minutes ; par exemple, si le coucher est à 2 h 48 et le lever à 9 h 42, l'écart en minutes est 60+42-48=54 et l'écart d'heures est 8-1-2=5 donc la durée de sommeil est 5h 54 min.

Pierre, papier, ciseaux

Enzo et Laura jouent au jeu pierre-papier-ciseaux. On dispose du choix de chacun des joueurs dans des variables `enzo` et `laura`, le choix étant donné par une chaîne de caractères minuscules, par exemple `"ciseaux"`. On demande d'écrire un programme qui définit puis affiche une variable `quiGagne` contenant le nom de la personne qui a gagné ou la chaîne **Match nul** si personne n'a gagné. Par exemple, si

```
enzo = "papier"
laura = "ciseaux"
```

le programme doit afficher

```
Laura gagne
```

ou encore si

```
enzo = "papier"
laura = "papier"
```

le programme doit afficher

```
Match nul
```

Les règles sont rappelées dans le tableau ci-dessous :

choix	choix	vainqueur
papier	ciseaux	ciseaux
ciseaux	pierre	pierre
pierre	papier	papier

Excédent payé

Soit à régler un montant de `m` euros avec des billets de 20 euros. Par exemple,

— si `m = 85` alors il faut 5 billets de 20 euros

— si `m = 120` alors il faut 6 billets de 20 euros

On demande de calculer le nombre `exc` représentant l'excédent payé en fonction de `m`. Par exemple,

— si `m = 85` alors `exc = 15`.

— si `m = 120` alors `exc = 0`.

Arrondir l'heure

(1) On donne un entier $n \geq 0$. Construire une variable `N` valant le multiple de 5 **le plus proche** de n. Vous raisonnerez en fonction la valeur du reste de la division de `n` par 5.
Par exemple :

- si $n = 42$ alors $N = 40$
- si $n = 15$ alors $N = 15$
- si $n = 64$ alors $N = 65$
- si $n = 90$ alors $N = 90$
- si $n = 0$ alors $N = 0$

(2) Cette question fait appel à la question précédente. Si vous n'avez pas réussi à la résoudre, vous pourrez utiliser le code ci-dessous qui donne le multiple de 5, appelé `m`, le plus proche de l'entier n :

```
m = round(n/5)*5
```

Vous allez devoir écrire un code qui arrondit une heure donnée aux 5 minutes les plus proches. Une heure de la journée sera codée par deux nombres entiers h et m, où h est le nombre d'heures ($0 \leq h < 24$) et m le nombre de minutes ($0 \leq m < 60$). Par exemple, 14 h 05 est codée par $h = 14$ et $m = 5$ ou encore 4 h sera codée par $h = 4$ et $m = 0$.
On se donne une heure de la journée, représentée par deux nombres entiers h et m. Écrire une code qui calcule l'heure arrondie à 5 minutes près. Plus précisément, votre code devra construire deux variables `H` et `M` correspondant à l'heure arrondie. Voici quelques exemples de comportements attendus :

```
14h 53m -> [14, 55]
18h 31m -> [18, 30]
02h 10m -> [2, 10]
01h 02m -> [1, 0]
09h 58m -> [10, 0]
23h 58m -> [0, 0]
23h 57m -> [23, 55]
```

On fera attention de ne pas écrire des nombres de minutes ou d'heures avec un zéro initial (comme 02), Python 3 considérant cette syntaxe comme une erreur.

Carré parfait le plus proche

On appelle *carré parfait* un entier qui a la propriété d'être le carré d'un entier. Par exemple, 100 est un carré parfait car $100 = 10^2$. En revanche, 42 n'est pas un carré parfait.

On donne un entier `k` $\geq$ `0` et on cherche le carré parfait le plus proche de `k`. Par exemple,

- le carré le plus proche de 42 est 36,
- le carré le plus proche de 43 est 64,
- le carré le plus proche de 100 est 10.

Un entier $k \geq 0$ étant donné, écrire un code calculant le carré parfait le plus proche de k.

On utilisera la partie entière d'un nombre `x` $\geq$ `0` et qui, en Python, est donnée par `int(x)`. Par exemple, la partie entière de `3,14` est `3`.

Cet exercice est inspiré de cette question.

Nombre d'entiers impairs entre deux entiers donnés

On donne deux entiers a et b avec $a \leq b$. On demande de déterminer le nombre d'entiers impairs n entre a et b, c'est-à-dire, vérifiant $a \leq n \leq b$. On pourra utiliser que si A et B sont des entiers

tels que $A \leq B$ alors le nombre d'entiers entre A et B, au sens large, est $B - A + 1$. On prendra en compte les 4 cas suivant la parité de a et b.

Voici quelques exemples de comportement :

```
Entre 42 et 50 : 4
Entre 42 et 51 : 5
Entre 43 et 50 : 4
Entre 43 et 51 : 5
Entre 42 et 42 : 0
Entre 42 et 43 : 1
Entre 43 et 43 : 1
Entre 43 et 44 : 1
Entre 123456789 et 987654321 : 432098767
```

Raccourcir une suite de conditions

Soit `x` une variable référençant un entier et soit le code suivant

```
if 0 <= x < 10:
    ok = True
elif x > 100:
    ok = (x %2) == 0
else:
    ok= False
```

Remplacer le code prédédent par un code totalement équivalent (donnant quelque soit la valeur de `x` la même valeur pour la variable `ok`) mais tenant sur une seule ligne.

Ramener `if/else` à un booléen

Soit le code suivant où `i` et `j` sont des entiers que l'on peut choisir librement :

```
from random import randrange
i=randrange(0,5) # juste un exemple
j=randrange(0,5) # juste un exemple
if i < j:
    if i % 2 ==0 and j == i + 1:
        v= True
    else:
        v=False
else:
    if j % 2 == 0 and i == j + 1:
        v= True
    else:
        v= False
print(i, j, v)
```

Simplifier ce code sous la forme suivante

```
from random import randrange
i=randrange(0,5) # juste un exemple
j=randrange(0,5) # juste un exemple
```

```
if i < j:
    v= COMPLETER
else:
    v= COMPLETER
print(i, j, v)
```

puis encore sous la forme :

```
from random import randrange
i=randrange(0,5) # juste un exemple
j=randrange(0,5) # juste un exemple
v= COMPLETER
print(i, j, v)
```

ou le COMPLETER sera remplacé par une expression booléenne.
Bien sûr, les codes produits doivent rester équivalents au code initial

Chapitre IV

Boucles for, listes, boucles while

1 Boucles for

Boucle `for` : introduction

Le code suivant présente un exemple d'utilisation d'une boucle for :

for.py

```
1 for i in range(0, 4):
2     print("Bonjour !")
3     print("---------")
```

```
4 Bonjour !
5 ---------
6 Bonjour !
7 ---------
8 Bonjour !
9 ---------
10 Bonjour !
11 ---------
```

Les lignes 1-3 constituent une instruction `for`. On observe (lignes 4-11) que le message `Bonjour !` suivi du séparateur `---------` est affiché 4 fois. Si on changeait la valeur 4 à la ligne 1 en la valeur 5 le message serait répété 5 fois.

Ici, la boucle `for` a permis de répéter une action un certain nombre de fois.

Déroulement d'une instruction `for`

- À la ligne 1, la présence de `range(0, 4)` permet d'itérer sur les entiers de 0 à 4, l'entier 4 étant exclu : 0, 1, 2 et 3. Les entiers vont être générés les uns à la suite des autres.
- Quand la boucle `for` commence, l'indice `i` vaut le premier élément généré (ici 0).
- L'exécution entre ensuite dans la ligne 2, effectue l'action d'affichage puis idem à la ligne 3
- Une fois les deux affichages effectués, l'exécution revient à la ligne 1. Comme pour l'instant `i = 0`, il reste encore des entiers à générer, donc, le principe même de l'instruction `for` veut que `i` passe à l'élément suivant, ie `i = 1` puis les instructions d'affichage des lignes 2-3 s'exécutent à nouveau.

— L'exécution se poursuit ainsi jusqu'à ce que `i` passe de 2 à 3. Lorsque `i` vaut 3, un affichage se fait encore et lorsque l'exécution repasse ligne 1, tous les entiers ont été générés : la conséquence est que la boucle s'arrête.

Édition d'une boucle `for`

On reprend le code `for.py`, on détaille la syntaxe d'une boucle `for` et le vocabulaire associé :

`for.py`

```
1 for i in range(0, 4):
2     print("Bonjour !")
3     print("---------")
```

— Ligne 1 : `for` et `in` sont des mots-clés de Python et sont obligatoires à toute boucle `for`.

— Ligne 1 : `i` est la *variable de contrôle* de la boucle `for`. D'autres noms fréquents de variables sont `j`, `k` mais tout nom de variable comme `toto` conviendrait. Ici, la variable de contrôle parcourt la succession des entiers générés par `range(0,4)`.

— Ligne 1 : le séparateur *deux-points* (**:**) à la fin est obligatoire

— Ligne 1 : `range` est une fonction built-in du langage Python qui sert à générer des entiers consécutifs.

— Ligne 1 : la partie qui commence à `for` jusqu'aux deux-points est appelé l'*en-tête* de la boucle `for`

— Lignes 2-3 : le *corps* de la boucle. Les instructions du corps de la boucle `for` sont celles qui vont être répétées tant que `i` est généré. Ce n'est pas le cas ici mais en général, le corps de la boucle dépend de la variable `i`. Le corps de la boucle `for` est dans l'immense majorité des cas, situé une ligne **sous** l'en-tête, autrement dit un saut de ligne dans le code-source est effectué après l'en-tête.

— Lignes 2-3 : le corps de la boucle `for` est toujours indenté et suit la règle générale d'indentation en Python.

— Lignes 1-3 : les trois lignes constituent une seule et même **instruction**, dite « instruction `for` ».

Le terme de *boucle* vient du fait que lorsque l'exécution est parvenue à la fin du corps de la boucle, l'exécution **retourne** à l'en-tête de la boucle.

Action qui dépend de la variable de la boucle

Dans le code `for.py`, l'action effectuée à chaque étape du parcours de la liste **ne** dépendait **pas** de la variable `i`. En fait, assez souvent, l'action effectuée dans le corps de la boucle **dépendra** de la variable de contrôle. Voici un exemple :

```
1 for i in range(0, 5):
2     print("Message n°", i+1, ":", "Bonjour !")
```

qui affiche

```
1 Message n° 1 : Bonjour !
2 Message n° 2 : Bonjour !
3 Message n° 3 : Bonjour !
4 Message n° 4 : Bonjour !
5 Message n° 5 : Bonjour !
```

On observe que chaque message affiché est différent du précédent par le numéro qu'il affiche : en effet, le corps de la boucle (ligne 2) dépend de `i` qui varie à chaque tour de boucle.

Répéter une action n fois avec une boucle `for`

La boucle for et la fonction range permettent typiquement de répéter une action `n` fois où `n` est fixé avant le démarrage de la boucle.

Par exemple, soit à afficher un carré de côté 10 comme ci-dessous :

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

L'affichage consiste en la répétition 10 fois de l'affichage d'une ligne de la forme

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

où les 10 étoiles sont séparées par un espace.

Le code Python pourrait être :

```
z="* * * * * * * * * *"
for i in range(10):
    print(z)
```

De même, soit à afficher les 10 premiers multiples de 5 en commençant par 75 :

$$75, 75+5=80, 75+10=85, \text{ etc.}$$

Donc, un code répondant à la question est le suivant :

```
1 d = 75
2 for i in range(10):
3     print(d + 5 * i)
```

```
4 75
5 80
6 85
7 90
8 95
9 100
10 105
11 110
12 115
13 120
```

— Ligne 2 : typiquement, on utilise range(10) pour exprimer qu'une action va être exécutée 10 fois. Le nombre de répétitions (ici 10) est connu avant d'effectuer la boucle `for`.

— Ligne 3 : l'action (d'afficher le multiple) est à effectuer 10 fois.

Règle : Si un programme *répète* une action et si le nombre de répétitions à effectuer est connu à l'avance, le programme utilise une boucle `for` avec la syntaxe suivante :

```
for i in range(n):
    # action a coder ICI
```

Cela fait partie des idiomes de la programmation impérative et, en particulier, de la programmation en Python.

Affichage sur une même ligne et boucle `for`

Soit à réaliser avec une boucle for l'affichage suivant :

```
----------------
0 1 2 3 4 5 6 7 8 9
----------------
```

Pour éviter le saut de ligne après l'affichage de chaque entier et pour séparer chaque nombre du précédent d'une espace, on utilise l'argument nommé `end = " "` :

```
1 print("----------------")
2 for i in range(10):
3     print(i, end = " ")
4 print()
5 print("----------------")
```

```
6 ----------------
7 0 1 2 3 4 5 6 7 8 9
8 ----------------
```

— Ligne 7 : chaque entier est séparé du précédent par un espace. Aucun saut de ligne n'est effectué lors de l'exécution de la boucle.

— Ligne 4 : permet d'effectuer un saut de ligne entre le dernier chiffre affiché (ligne 7) et la ligne formée de pointillés (ligne 8)

Omettre l'appel à `print` en sortie de boucle provoque un affichage incorrect [1] :

```
1 print("----------------")
2 for i in range(10):
3     print(i, end = " ")
4 print("----------------")
```

```
5 ----------------
6 0 1 2 3 4 5 6 7 8 9 ----------------
```

Pour bien comprendre la rôle de `end=" "` dans `print(i, end=" ")`, il suffit d'observer le résultat du code ci-dessous où le `end=" "` est remplacé par, par exemple, la chaîne `end="xxx"` :

1. En fait, cela dépend de l'interface d'utilisation.

```
print("----------------")
for i in range(10):
    print(i, end="xxx")
print()
print("----------------")
```

```
----------------
0xxx1xxx2xxx3xxx4xxx5xxx6xxx7xxx8xxx9xxx
----------------
```

Itérer sur des entiers consécutifs

La fonction built-in `range` permet de générer automatiquement des entiers consécutifs, comme les entiers 6, 7, 8 et 9 :

```
r=range(6, 10)
for i in r:
    print(i)
```

```
6
7
8
9
```

En pratique, on n'utilise pas la variable intermédiaire `r` ci-dessus et on écrit plutôt :

```
for i in range(6, 10):
    print(i)
```

Bien observer que le deuxième argument de la fonction `range` (dans l'exemple, c'est 10) est exclu des nombres générés. Toutefois, le nombre d'entiers générés par `range(a,b)` est `b` $-$ `a` (en supposant que `a` $\leq$ `b`).

Si `n` ≥ 0 est un entier, l'expression `range(n)` est un raccourci syntaxique pour `range(0,n)` :

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

```
0
1
2
```

Calcul d'une somme

La méthode pour calculer une somme de nombres doit être très bien connue tant ce type de problème est fréquent. Illustrons par un exemple.

On se donne un entier `n` ≥ 1 et on demande de calculer la somme de tous les cubes des entiers entre 1 et `n`. Par exemple, si `n` $= 9$ alors la somme vaut

$$1^3 + 2^3 + 3^3 + \cdots + 8^3 + 9^3 = 2025$$

L'algorithme, consiste à calculer successivement les sommes partielles suivantes

$$1^3,\ 1^3+2^3,\ 1^3+2^3+3^3,\ \ldots$$

jusqu'à la somme voulue, chaque somme s'obtenant à partir de la précédente en additionnant un nombre (un cube pour être précis).

En Python, on procède comme suit :

— on définit une variable « accumulatrice » `s`, initialisée à 0 et qui, en fin de programme, vaudra la somme cherchée ;

— on parcourt avec une boucle `for` tous les entiers entre 1 et `n` inclus ;

— lorsque, dans la boucle, on traite l'entier `k` entre 1 et `n`, on augmente la somme partielle `s` de la valeur k^3. Pour cela, on écrit l'instruction `s = s + k**3`

D'où le code suivant :

```
n = 9

s = 0

for k in range(1, n+1):
    s = s + k**3

print(s)
```

```
2025
```

Exercice type – De 42 en 42

Afficher les entiers obtenus en comptant 10 fois de 42 en 42 à partir de 421.

Solution

Il s'agit de répéter une action 10 fois, donc il est assez naturel de chercher un code ayant la forme suivante :

```
1 for i in range(10):
2     # Code
3     # à compléter
```

Les nombres à afficher vont évoluer dans une variable x initialisée à 421. A chaque tour de boucle, x est incrémenté de 42. D'où le code :

```
1 x=421
2 for i in range(10):
3     print(x)
4     x = x + 42
```

```
5 421
6 463
7 505
8 547
9 589
10 631
11 673
```

```
715
757
799
```

Variante :

```
for i in range(10):
    print(421 + i * 42)
```

Exercice type – Comptage de 5 en 5

On compte de 5 en 5 tous les entiers entre deux entiers a et b (bornes incluses) avec $a \leq b$, par exemple $a = 1914$ et $b = 2020$. À l'aide d'une boucle for, déterminer combien d'entiers on compte ainsi (pour l'exemple, on en trouvera 22).

Solution

Pour compter de 5 en 5 dans une suite d'entiers consécutifs, il suffit de les indexer et de ne compter que les multiples de 5 dans l'énumération.

D'où le code :

```
a=1914
b=2020
cpt = 0
j=0

for i in range(a, b+1):
    if j%5==0:
        cpt = cpt + 1
    j = j + 1

print(cpt)
```

```
22
```

— Ligne 6 : la variable `i` est factice, elle permet juste à la boucle `for` de s'exécuter.
— Lignes 4 et 9 : la variable `j` sert à énumérer les entiers parcourus.
— Ligne 8 : on compte de 5 en 5.

Exercice type – Somme alternée plus/moins

On se donne un entier positif `n`, par exemple, `n = 7`. On cherche à calculer la « somme »

$$1 - 2 + 3 - 4 + 5 - \cdots \pm \texttt{n}$$

où on alterne tantôt une addition, tantôt une soustraction. Pour `n = 7`, la somme vaut

$$1 - 2 + 3 - 4 + 5 - 6 + 7 = 4$$

Ecrire un code qui calcule `s` à partir de `n`. On pourra vérifier que pour `n` valant `2030` la somme vaut `-1015`.

Solution

En examinant l'exemple donné, on comprend qu'on effectue une addition sur les entiers impairs et une soustraction sur les entiers pairs. Il suffit donc de parcourir tous les entiers `i` entre 1 et `n` et d'ajouter ou retrancher `i` à une variable accumulatrice `s` initialisée à 0 selon que l'entier courant `i` est impair ou pair :

```
n=2030
s=0

for i in range(1, n+1):
    if i % 2 == 1:
        s = s + i
    else:
        s = s - i

print(s)
```

```
-1015
```

On peut aussi donner une solution plus matheuse en remarquant que la somme cherchée n'est autre que

$$\sum_{i=1}^{n}(-1)^{i+1}i$$

ce qui donne le code suivant :

```
n=2030
s=0

for i in range(1, n+1):
    s = s - i*(-1)**i

print(s)
```

```
-1015
```

Exercice type – Afficher des nombres par paires

On se donne deux entiers a et b, avec $a \leq b$. Soit S la succession de tous les entiers depuis l'entier a inclus jusqu'à l'entier b inclus. On demande d'afficher tous les entiers de S par lignes de deux entiers consécutifs, sauf, **éventuellement**, la dernière ligne qui ne contiendra que le dernier élément de S. Le tableau ci-dessous donne quatre exemples :

a	b	Affichage
2	9	1 2 3 2 4 5 3 6 7 4 8 9
2	8	1 2 3 2 4 5 3 6 7 4 8
2	2	1 2
2	3	1 2 3

Bien tester tous ces cas.

Solution

On compte entre a et b un nombre de $N = b - a + 1$ entiers. Puisque les entiers doivent groupés par deux il y a aura $q = N//2$ lignes de deux entiers et éventuellement une ligne de plus. Tout dépend de la parité de N. L'idée est donc d'afficher les q premières lignes et selon les cas de rajouter une ligne. D'où le code suivant :

```
1  a=2
2  b=9
3  N=b+1-a
4  q=N//2
5  x=a
6
7  for i in range(q):
8      print(x, x+1)
9      x=x+2
10 if N%2==1:
11     print(b)
```

```
12 2 3
13 4 5
14 6 7
15 8 9
```

— Lignes 5 et 9 : x est le nombre qui est affiché en chaque début de ligne (sauf peut-être la dernière).

— Ligne 9 : en effet, les nombres vont deux par deux.

— Lignes 11-12 : s'il y a un nombre impair de nombres à afficher, c'est que le dernier est seul sur sa ligne, et le dernier entier à afficher est `b`.

Le code ci-dessus ne montre qu'un seul exemple, pour en voir d'autres, changer les valeurs de `a` ou `b` et exécuter à nouveau le code.

On peut simplifier le code de la manière suivante :

```
1 a=2
2 b=6
3
4 for i in range((b+1-a)//2):
5     print(a, a+1)
6     a=a+2
7 if a-1!=b:
8     print(b)
```

```
9 2 3
10 4 5
11 6
```

— La variable `x` du précédent code n'est pas indispensable, il suffit d'utiliser `a` mais la valeur initiale de `a` est alors perdue à la fin du programme.

— Une fois que les lignes contenant deux entiers sont affichés (lignes 4-6) on regarde si le dernier entier affiché est `b`. Si c'est la cas, tous les entiers ont été affiché et il n'y a rien de plus à afficher. Si ce n'est pas le cas, il faut afficher `b` (ligne 8). Quel est le dernier nombre affiché ? réponse : c'est `a - 2 + 1` et donc `a - 1`, d'où la ligne 7.

Il était également possible de ne pas diviser par deux le nombre d'entiers à afficher :

```
1 a=2
2 b=8
3 cpt=0
4
5 for i in range(a,b+1):
6     cpt+=1
7     if cpt ==2:
8         print(i-1, i)
9         cpt=0
10
11 if cpt !=0:
12     print(b)
```

```
13 2 3
14 4 5
15 6 7
16 8
```

Le principe de ce code est de compter les nombres par groupes de 2 à l'aide du compteur `cpt` et lorsque le compteur atteint 2, les nombres sont affichés (ligne 8) et le compteur est remis à zéro (ligne 9).

EXERCICES

Répéter 2030

Afficher les uns en dessous des autres 10 fois de suite l'entier 2030.

Afficher des entiers consécutifs

Afficher les uns en dessous des autres tous les entiers consécutifs entre 2020 et 2038.

Afficher deux listes l'une après l'autre

Écrire un code, utilisant deux boucles `for` *successives* qui produise l'affichage exact suivant (noter qu'il y a deux lignes dans le même affichage) :

```
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
```

Itérer 2x+1

On considère la suite de nombres entiers dont les premiers termes sont :

```
10, 21, 43, 87, 175, 351, etc
```

Cette suite est construite de la manière suivante :

— son premier terme est 10

— si x est un terme de la suite, le terme suivant vaut $2x + 1$.

Par exemple, si $x = 87$ alors le terme suivant dans la suite est $2x + 1 = 2 \times 87 + 1 = 175$.

Ecrire un code qui à partir d'un entier $n > 0$ affiche le n^{e} terme de la suite. Exemple de comportements selon la valeur de `n` :

```
3 -> 43
1 -> 10
7 -> 703
```

Somme des n premiers entiers

On se donne un entier $n \geq 0$. Calculer la somme $1 + 2 + \ldots + n$. Vérifier sur quelques exemples, que le résultat obtenu est bien $\dfrac{n(n+1)}{2}$.

Somme de n entiers consécutifs

On se donne un entier `n` ≥ 0 et on demande de calculer la somme de tous les entiers entre `n` et `2n`. Par exemple, si `n` $= 5$ alors la somme vaut

$$5 + 6 + 7 + 8 + 9 + 10 = 45$$

Somme d'entiers impairs consécutifs

On donne un entier $N \geq 0$ et on demande de calculer la somme des N premiers entiers impairs, c'est-à-dire

$$1, 3, 5, 7, 9, 11, \text{etc}$$

Par exemple, si $N = 6$, on trouvera $1 + 3 + 5 + 7 + 9 + 11 = 36$.
Vérifier dans vos tests que cette somme vaut toujours N^2.

Sommes de puissances consécutives

On se donne un entier $n \geq 0$. Calculer la somme $1 + 10 + ... + 10^{n-1}$. Par exemple, si $n = 5$, on obtiendra 11111 pour somme. Vérifier que vous obtenez bien $\frac{10^n-1}{9}$.

Sommes de puissances de 2 ou de 3

On donne un entier `n` > 0 et on demande de calculer la somme suivante :

$$S_n = 3^1 + 2^2 + 3^3 + 2^4 + \cdots + ?^n$$

Si l'exposant courant k est pair, on ajoute 2^k sinon on ajoute 3^k. La somme s'arrête lorsqu'elle contient n termes. Par exemple, $S_4 = 3^1 + 2^2 + 3^3 + 2^4 = 3 + 4 + 27 + 16 = 50$ et $S_5 = 3^1 + 2^2 + 3^3 + 2^4 + 3^5 = 50 + 243 = 293$

Somme alternée plus/moins en décroissant

On cherche à écrire un code capable de calculer des expressions du genre

$$7 - 6 + 5 - 4 + 3 - 2 + 1.$$

Plus précisément, on se donne un entier $n > 0$ (ci-dessus, c'était $n = 7$) et on demande d'écrire un code qui renvoie la valeur de l'expression :

$$n - (n - 1) + (n - 2) - (n - 3) + \ldots$$

expression qui se poursuit jusqu'à ce que le terme courant vaille 1. Noter que l'expression commence par n et alterne soustraction et addition. Si $n = 2037$ ou $n = 2038$ on trouvera que la somme vaut 1019.

Calcul d'une somme (sigma)

Calculer pour différentes valeurs de N la somme $\displaystyle\sum_{n=1}^{N} \left(\frac{1}{2n} - \frac{1}{n+1} + \frac{1}{2(n+2)} \right)$. Vers quelle valeur semble se rapprocher la somme lorsque N augmente ?

Plus grand diviseur impair

Soit un entier $n > 0$. On cherche le plus grand diviseur impair d de n. Par exemple :

- si $n = 42$ alors $d = 21$,
- si $n = 45$ alors $d = 45$,
- si $n = 64$ alors $d = 1$,
- si $n = 1000$ alors $d = 125$.

Déterminer d est parcourant tous les entiers entre 1 et n. En particulier, déterminer le plus grand diviseur impair de $n = 416241604$.

Restes de deux entiers

Si on divise 1234567 par un certain nombre et on obtient 124 pour reste et si on divise 87654321 par le même nombre on trouve 760 pour reste. Trouver bestialement le nombre par lequel on a divisé.

Compter de 0,5 en 0,5

Cet exercice est corrigé en vidéo.

On donne deux entiers a et b avec $a \leq b$ et on demande d'afficher tous les nombres entre a et b si on compte de 0,5 en 0,5. Par exemple si $a = 42$ et $b = 49$, il faut afficher les nombres suivants, de préférence sur une même ligne :

```
42 42.5 43 43.5 44 44.5 45 45.5 46 46.5 47 47.5 48 48.5 49
```

Il est attendu qu'un entier soit affiché sans point décimal, par exemple `42` et pas `42.0`

Compter par tiers

On donne deux entiers `a` et `b` avec `a` $\leq$ `b`. On demande d'afficher le décompte entre `a` et `b` en progressant par tiers

Restes de deux entiers

Si on divise 1234567 par un certain nombre et on obtient 124 pour reste et si on divise 87654321 par le même nombre on trouve 760 pour reste. Trouver bestialement le nombre par lequel on a divisé.

Entier pyramidal

Un entier $N > 0$ est dit *pyramidal* si c'est le nombre de boules nécessaires pour bâtir une pyramide de hauteur n et de base un carré de côté ayant n boules, cf. l'image ci-dessous correspondant à $n = 5$:

Pour la suite, on a juste besoin de savoir qu'un nombre pyramidal est de la forme

$$N = \frac{n(n+1)(2n+1)}{6}$$

Ainsi l'entier $N = 55$ est un nombre pyramidal car si $n = 5$ alors $N = \frac{n(n+1)(2n+1)}{6} = 55$. Les premiers nombres pyramidaux sont :

$$1, 5, 14, 30, 55, 91, 140.$$

Déterminer si les entiers suivants sont pyramidaux :

- $N = 1785$,
- $N = 2020$,
- $N = 4900$,
- $N = 317104004$,
- $N = 58963256321402313634$,
- $N = 30979120769004056775$

On pourra utiliser que si $N = \frac{n(n+1)(2n+1)}{6}$ alors $n \leq 2 \times N^{\frac{1}{3}}$

Entiers oblongs

Un entier $p > 0$ est dit *oblong* (on dit aussi *pronique*) s'il peut s'écrire comme le produit de deux entiers consécutifs n et $n+1$, autrement dit si $p = n(n+1)$. Les nombres $20 = 4 \times 5$ ou $42 = 6 \times 7$ sont donc oblongs mais ce n'est pas le cas, par exemple, de

$$18 = 1 \times 18 = 2 \times 9 = 3 \times 6$$

On demande de dire si les entiers suivants sont oblongs :

- $p = 4066272$,
- $p = 4088459$,
- $p = 738912181599504601846432053 0$,
- $p = 582101256399963012457850014 5$.

Il pourra être utile, dans les deux derniers cas, d'observer que si $p = n(n+1)$ alors forcément $n \leq \sqrt{p}$.

Produit d'entiers

Calculer le produit P des entiers entre 42 et 421, autrement dit $P = 42 \times 43 \times \cdots \times 420 \times 421$.

On trouvera un très grand entier qui commence par 1484.

Allers et retours

On se donne deux entiers a et b avec $a < b$ ainsi qu'un entier `N` > 0. Un puce, placée initialement à l'entier a, se déplace pendant `N` secondes. A l'issue des `N` secondes elle s'arrête à la position où elle se trouve. La puce se déplace entre l'entier a et l'entier $b \geq a$ à raison d'une unité par seconde. Quand elle atteint b et que son parcours n'est pas fini, elle repart vers a et, de même, quand elle atteint a elle repart vers b. On demande d'afficher seconde par seconde, toutes les étapes de son parcours lorsque son déplacement dure N secondes.

Par exemple, si $a = 5$, $b = 8$ et $N = 12$ alors l'affichage doit être :

```
0 seconde -> 5
1 secondes -> 6
2 secondes -> 7
3 secondes -> 8
4 secondes -> 7
5 secondes -> 6
6 secondes -> 5
7 secondes -> 6
8 secondes -> 7
9 secondes -> 8
10 secondes -> 7
11 secondes -> 6
12 secondes -> 5
```

On utilisera une variable `x` référençant la position courante de la puce et un drapeau appelé `sens` qui vaudra `1` si le déplacement courant se fait vers la droite et qui vaudra `-1` si le déplacement courant se fait vers la gauche. Voici un template de code :

```
sens=1
N=12
a=5
b=8
x=a

print(0, "seconde ->", x)

for i in range(N):
    # Votre code
```

Pomodoro

Soit à afficher 10 fois l'heure à partir de 20h42 avec des intervalles d'attente de 25 minutes. On obtient alors l'affichage suivant :

```
20 h 42
21 h 7
21 h 32
21 h 57
22 h 22
22 h 47
23 h 12
23 h 37
0 h 2
0 h 27
```

Plus généralement, on donne un moment de la journée par

— son nombre d'heures `h`,

— son nombre de minutes `m`,

par exemple, `h=20` et `m =42`;
on donne aussi

— une période de temps, en minutes, par exemple `periode=25` en supposant que `periode` < 60

— un nombre entier, disons `n` ≥ 1

et on demande d'afficher l'heure qu'il est aux `n` moments à partir de l'heure donnée et à intervalles réguliers de `periode` minutes. Le nombre d'heures affiché doit toujours être strictement inférieur à 24 et le nombre de minutes strictement inférieur à 60.

Afficher par paires verticales

On donne deux entiers a et b et on demande d'afficher, par ordre croissant tous les entiers de a à b sur deux lignes en plaçant alternativement les entiers sur la ligne du haut et sur la ligne du bas. Par exemple, si $a = 2018$ et $b = 2025$ le code doit afficher

```
2018 2020 2022 2024
2019 2021 2023 2025
```

et si $a = 2020$ et $b = 2038$ le code doit afficher

```
2020 2022 2024 2026 2028 2030 2032 2034 2036 2038
2021 2023 2025 2027 2029 2031 2033 2035 2037
```

Afficher par triplets verticaux

On donne deux entiers a et b et on demande d'afficher, par ordre croissant tous les entiers de a à b sur trois lignes en plaçant successivement les entiers sur la ligne du haut, la ligne centrale et sur la ligne du bas. Par exemple, si $a = 42$ et $b = 53$ le code doit afficher

```
42 45 48 51
43 46 49 52
44 47 50 53
```

et si $a = 2019$ et $b = 2038$ le code doit afficher

```
2019 2022 2025 2028 2031 2034 2037
2020 2023 2026 2029 2032 2035 2038
2021 2024 2027 2030 2033 2036
```

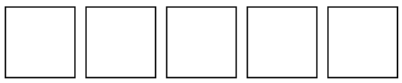
Suite de disques

On se donne des entiers $n, R, s > 0$, par exemple $n = 8$, $R = 2.5$ et $s = 1$. Dessiner un alignement horizontal de n disques noirs de rayon R, deux disques successifs étant séparés par une distance de s.

Carrés côte-à-côte

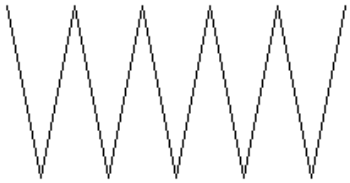
On donne un entier `n` ≥ 0, un entier `c` > 0 et un entier `sep` > 0. Construire une suite de `n` carrés, alignés horizontalement, de côté `c` et séparés d'une longueur `sep`.

Par exemple, si `n` $= 5$, `sep` $= 8$ et `c` $= 50$ alors le code affichera :



Dents de scie

Dessiner une ligne polygonale formée de `n` dents de scie. La hauteur de chaque dent sera enregistrée dans une variable `h` et la largeur dans une variable `L`. Le dessin ci-dessous correspond à `n` = 5, `h` = 100 et `L` = 40.



Cible

On se donne un entier $n > 0$ par exemple $n = 10$. Dessiner une cible bicolore noire et jaune constituée de n disques :

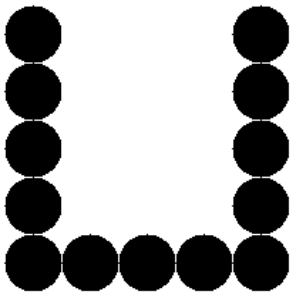
le disque central de la cible sera indifféremment jaune ou noir et les couronnes seront de largeur constante. Aucune contrainte n'est imposée sur la valeur du rayon du disque central. La couronne extérieure sera noire.

Indications : on dessinera des disques en commençant par celui de plus grand rayon. Pour gérer l'alternance des couleurs, on prendra en compte la parité du compteur de boucle.

[Exercice fourni par mon collègue P. Piccinini.]

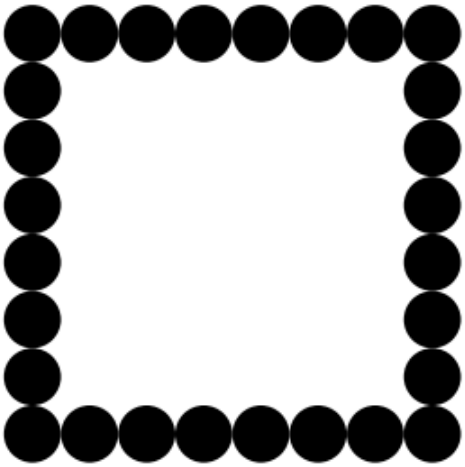
Dessiner un U

Dessiner le motif ci-dessous. Chaque disque est tangent à son ou ses voisins. Chaque côté contient le même nombre n de disques, dans l'exemple $n = 5$. Le rayon de chaque disque sera de 20.

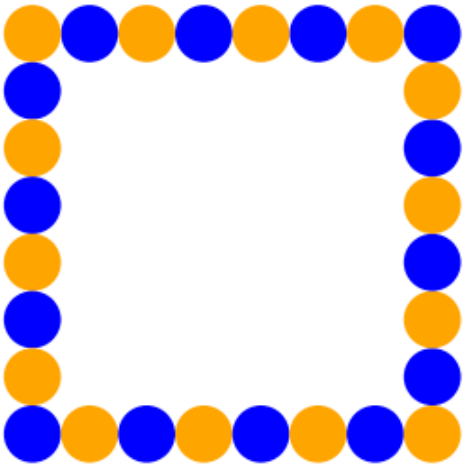


Disques en carrés

1. Dessiner une suite de disques de diamètre `d` donné qui forment un carré ayant n disques par côté où n est donné.

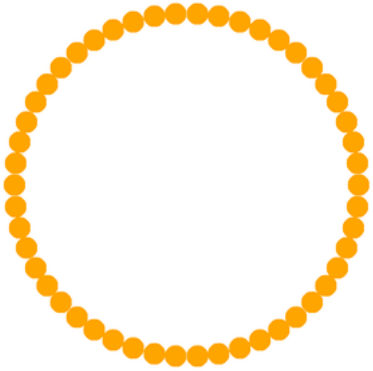


2. Refaire le même dessin mais en alternant les couleurs orange et bleue.

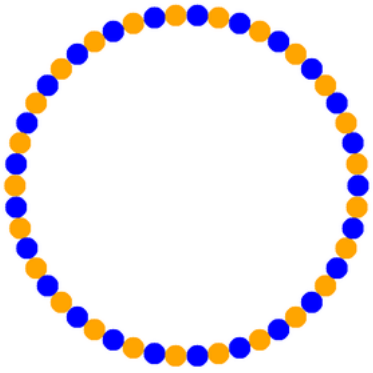


Collier de perles

Dessiner un collier monocolore :



puis un collier bicolore :

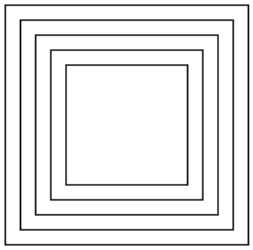


Le collier aura la forme d'un cercle de rayon R. Si le collier a n perles, chaque perle sera un disque de centre $(R\cos(2k\pi/n), R\sin(2k\pi/n))$ où k prendra n valeurs entières à partir de 0.

L'observation attentive du dessin vous fera comprendre comment déterminer la dimension de chaque perle. Le dessin ci-dessus a été effectué avec 50 perles de couleur orange ou bleu (si le collier est bicolore, n est pair).

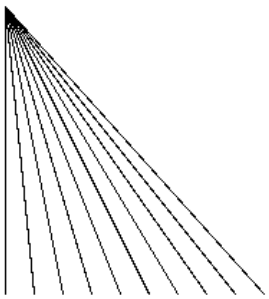
Carrés concentriques

Écrire un programme qui dessine n carrés concentriques, centrés en l'origine, avec un écart de e entre deux carrés successifs. Utiliser une variable pour paramétrer la longueur du côté du carré initial.



Dessiner un faisceau de demi-droites

Dessiner le motif ci-dessous.



Plus précisément, on demande de construire n segments dont une extrémité est le point $(0, 0)$ et les autres extrémités étant placées

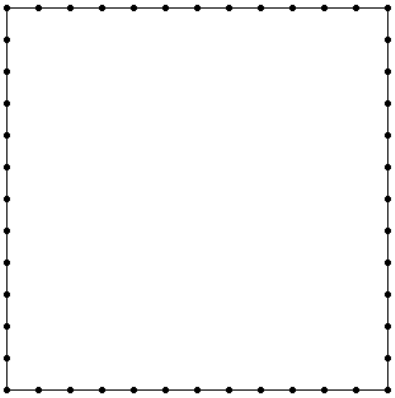
— sur une base horizontale située à une distance `d` de $(0, 0)$ et située en-dessous de $(0, 0)$

— régulièrement espacées (espacement régulier valant `delta`)

Le segment le plus à gauche sera vertical. Sur la figure, `n`=10, `d`=200 et `delta`=20.

Clôturer un champ carré

On veut graphiquement clôturer un champ carré de côté a :

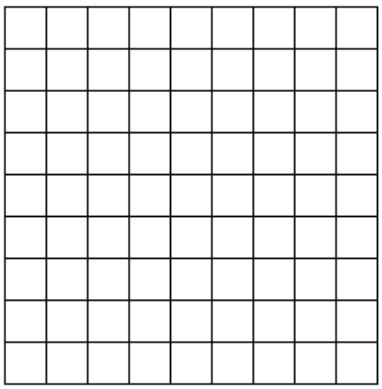


La distance entre deux poteaux est constante et vaut 25. Chaque coin du champ comporte un poteau.

Dessiner le champ ainsi que tous les poteaux de la clôture. Le côté a du champ pourra être un quelconque multiple de 25 ; le dessin a été réalisé pour $a = 300$.

Grille carrée

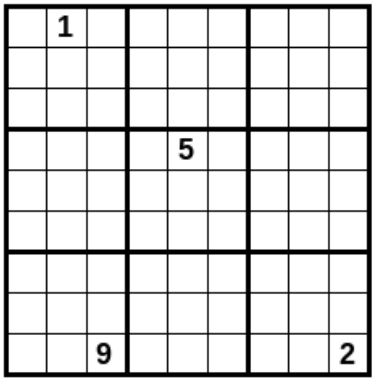
Dessiner une grille carrée comme ci-dessous :



La largeur de chaque case sera enregistrée dans une variable `c` et le nombre de cases par côté dans une variable `n`. On dessinera la grille non pas case par case mais comme on le ferait à la main : on trace d'abord `n + 1` lignes verticales que l'on croise ensuite avec `n + 1` lignes horizontales.

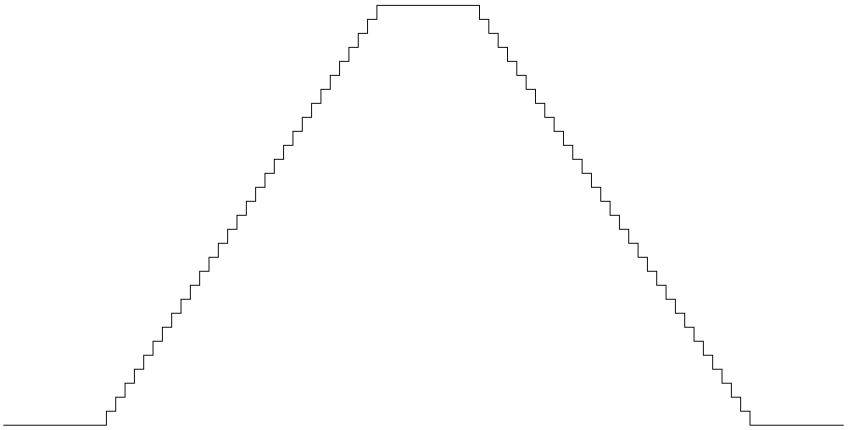
Grille de Sudoku

Dessiner une grille de sudoku comme ci-dessous :



Pyramide maya

On demande de dessiner une pyramide telle que celle ci-dessous :



On utilisera des variables :

— `n` : le nombre de marches
— `h` : la hauteur de chaque marche
— `p` : la profondeur de chaque marche
— `L` : la longueur commune des autres traits.

2 Listes et matrices

Notion de liste

En Python, une liste est une structure de données modifiable représentant une « succession » d'objets :

```
1 L = [2020, 42, 2038]
2 print(L)
```

```
3 [2020, 42, 2038]
```

— Ligne 1 : L désigne une liste, formée de trois éléments. Une paire de crochets entoure les éléments de la liste et ces éléments sont séparés par des virgules.

— Ligne 2 : La fonction `print` est le moyen le plus simple d'obtenir un affichage sommaire d'une liste L.

La liste est la structure couramment utilisée pour représenter des « listes » au sens usuel du terme, comme une liste de mots :

```
1 L = ["Printemps", "Été", "Automne", "Hiver"]
2 print(L)
```

```
3 ['Printemps', 'Été', 'Automne', 'Hiver']
```

Les éléments d'une liste peuvent aussi être des expressions utilisant des variables :

```
1 x=4
2 L = [2020, 10 * x + 2, 2038]
3 print(L)
```

```
4 [2020, 42, 2038]
```

Les listes ci-dessus sont des listes dites *littérales* : elles sont définies en explicitant ses éléments dans le code-source, placés entre crochets, un par un et en les séparant par des virgules.

Indexation des éléments d'une liste

Une propriété fondamentale des listes est qu'on accède directement à chaque élément d'une liste par un indice entier. Les différents éléments de la liste sont indexés par les entiers 0, 1, 2, etc. jusqu'à la fin de la liste.

```
1 L = [2020, 42, 2038]
2 print(L[0])
3 print(L[1])
4 print(L[2])
```

```
5 2020
6 42
7 2038
```

— Ligne 2 : les éléments de la liste sont indexés, **en commençant par 0**.

— Lignes 2-4 : on peut accéder à chaque élément de la liste L par son indice entouré d'une paire de crochets.

Attention ! l'indexation commence à 0 et non à 1.

Si une liste contient n éléments, ses indices sont tous les entiers entre 0 et $n-1$, bornes 0 et $n-1$ incluses, par exemple, une liste de 4 éléments est indexée par les entiers 0, 1, 2 et 3. L'élément d'indice i de la liste L est l'objet `L[i]`. L'indice 0 correspond toujours au premier élément de la liste et l'indice $n-1$ d'une liste L de longueur n correspond toujours au dernier élément de L.

Tenter d'accéder à un indice se référant à un élément hors de la liste déclenche une erreur de type `IndexError` :

```
1 L = [2020, 42, 2038]
2 print(L[3])
```

```
3 Traceback (most recent call last):
4   File "liste_indice_impossible.py", line 2, in <module>
5     print(L[3])
6 IndexError: list index out of range
```

— Ligne 1 : les indices de la liste `L` sont : 0, 1 et 2

— Ligne 2 : on essaye d'accéder à un indice de `L` qui n'existe pas

La tentative d'accès à un élément d'une liste par un indice entier impossible est appelé un *débordement d'indice* ou aussi *dépassement d'indice*.

Nombre d'éléments d'une liste

La fonction built-in `len` appliquée à une liste renvoie le nombre d'éléments de la liste :

```
1 t = [81, 12, 65, 31]
2 n = len(t)
3 print(n)
```

```
4 4
```

— Ligne 2 : `n` est le nombre d'éléments de la liste `t`.

Une liste peut contenir un nombre quelconque d'éléments ; en particulier une liste peut contenir un seul élément et même aucun élément. La liste littérale `[]` ne contient aucun élément :

```
1 L= [42]
2 print(len(L))
3 L= []
4 print(len(L))
```

```
5 1
6 0
```

— Ligne 1 : liste ayant un seul élément.

— Ligne 3 : la liste vide de longueur nulle (ligne 6).

Opérations sur les éléments d'une liste

Les éléments d'une liste `L` sont de la forme `L[i]`. Cette notation peut être utilisée pour des opérations comme on le ferait avec des variables :

```
1 t = [5, 3, 25, 3]
2 x = t[0] * t[1] + t[2] *t[3]
3 print(x)
```

```
90
```

Dans une certaine mesure, on peut considérer une liste comme une suite ordonnée de variables.

Appartenance à une liste

L'opérateur `in` permet de tester l'appartenance d'un élément à une liste :

```
L = [65, 31, 9, 32, 81, 82, 46, 12]
print(75 in L)
print(81 in L)
```

```
False
True
```

L'opérateur `in` renvoie un booléen. Le mot `in` est un mot-clé du langage Python.

Pour tester l'appartenance de `a` à une liste `L`, Python parcourt toute la liste depuis le début (l'indice 0) et jusqu'à ce que l'élément `a` soit trouvé ou que la fin de `L` soit rencontrée. Cela signifie que le test d'appartenance, bien que tenant sur une seule instruction, peut être très coûteux si `L` est longue.

Non-appartenance à une liste

L'opérateur `not in` permet de tester la non-appartenance d'un élément à une liste :

```
L = [65, 31, 9, 32, 81, 82, 46, 12]
print(75 not in L)
print(81 not in L)
```

```
True
False
```

L'opérateur `not in` renvoie un booléen. L'expression `a not in L` a même valeur de vérité que `not a in L` :

```
L = [65, 31, 9, 32, 81, 82, 46, 12]
print(75 not in L)
print(not 75 in L)
```

```
True
True
```

Le test de non-appartenance est basé sur le même principe que le test d'appartenance : un parcours de la liste depuis son début et arrêt du parcours lorsque la réponse est connue.

Modifier les éléments d'une liste

C'est une propriété fondamentale des listes : il est possible d'en modifier les éléments. Pour cela :

— on accède à l'élément par son indice

— on modifie l'élément par affectation.

Voici un exemple

```
1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2 print(L)
3
4 L[4] = 75
5 print(L)
```

```
6 [65, 31, 9, 32, 81, 82, 46, 12]
7 [65, 31, 9, 32, 75, 82, 46, 12]
```

— Ligne 4 : modification d'un élément de la liste par affectation ;

— Ligne 7 : comparer, la liste L a été modifiée.

Le fait de pouvoir modifier le contenu d'une liste se traduit en disant qu'une liste est un objet *mutable*. Si L est une liste, on peut non seulement changer ses éléments, mais lui en adjoindre, ou lui en supprimer et donc modifier le nombre d'éléments de la liste.

Comme pour une variable, on peut affecter ou réaffecter un élément d'une liste :

```
1 t = [5, 3, 25, 3]
2 t[0] = 10 * t[0]
3 print(t)
```

```
4 [50, 3, 25, 3]
```

— Ligne 2 : réaffectation de l'élément `t[0]`.

Alias de liste

Le code suivant :

```
1 L = [2020, 42, 2038]
2 M = L
```

peut donner l'illusion qu'on dispose de **deux** listes et qu'on aurait cloné L en M. Ce n'est pas le cas, M n'est qu'un *alias* de L, un renommage. C'est un code complètement différent de

```
1 L = [2020, 42, 2038]
2 M = [2020, 42, 2038]
```

Dans le premier code, toute modification de la liste par l'un des identificateurs se verra si on accède à la liste par l'autre identificateur, par exemple :

```
1 L = [2020, 42, 2038]
2 M = L
3
4 print("L =", L)
5 print("M =", M)
6
7 M[0]="XXXXX"
8 print("L =", L)
```

```
print("M =", M)
```

```
L = [2020, 42, 2038]
M = [2020, 42, 2038]
L = ['XXXXX', 42, 2038]
M = ['XXXXX', 42, 2038]
```

Tandis qu'avec le deuxième code, **seule une** liste est modifiée :

```
L = [2020, 42, 2038]
M = [2020, 42, 2038]

print("L =", L)
print("M =", M)

M[0]="XXXXX"
print("L =", L)
print("M =", M)
```

```
L = [2020, 42, 2038]
M = [2020, 42, 2038]
L = [2020, 42, 2038]
M = ['XXXXX', 42, 2038
```

Liste de listes

Les éléments d'une liste peuvent être d'autres listes :

```
1 t = [[65, 31], [], [9, 32, 81], [82], [46, 12]]
2 L = t[2]
3 print(L)
4 print(L[1])
5 print(t[2][1])
```

```
6 [9, 32, 81]
7 32
8 32
```

— Ligne 1 : `t` est une liste dont les 5 éléments sont des listes

— Ligne 2 : comme pour n'importe quelle liste, on accède aux éléments de `t` par indexation

— Ligne 4 : `L` étant elle-même une liste, on accède à l'un de ses éléments par indexation

— Ligne 5 : il n'est pas nécessaire de passer par la variable intermédiaire `L` pour accéder à un élément d'une liste de `t`, on peut utiliser une double indexation. La notation `t[2][1]` est équivalente à `(t[2])[1]`.

Pour représenter des relevés de températures, des notes par trimestre et par matières, on utilise des liste de listes, des listes de listes de listes, etc.

Notion de tableau 2D

Soit à écrire un code qui puisse implémenter la structure de données assimilée à un tableau formé de lignes et de colonnes, par exemple un tableau tel que celui-ci :

```
2 6 9
3 3 2
4 7 5
0 0 7
```

Voici un code résolvant le problème :

```
1 t=[[2, 6, 9], [3, 3, 2], [4, 7, 5], [0, 0, 7]]
2 print(t[2])
3 print(t[2][1])
```

```
4 [4, 7, 5]
5 7
```

— Ligne 1 : on implémente un tableau 2D comme une liste de listes, chaque liste représentant une ligne du tableau. Ici, `t` est une liste composée de 4 listes, et chacune des 4 listes possède 3 éléments entiers.

— Lignes 2 et 4 : chaque élément de t est une liste. Donc `t[i]` représente la ligne i du tableau, chaque ligne étant indexée à partir de 0, donc `t[2]` est la 3e ligne

— Lignes 3 et 5 : l'élément du tableau `t` situé à la ligne i et à la colonne j vaut `t[i][j]`.

Accès aux dimensions d'un tableau codé en Python

A partir de la structure de données précédente, on cherche à accéder aux nombres de lignes et de colonnes du tableau 2D suivant :

```
2 6 9
3 3 2
4 7 5
0 0 7
```

Il y a $n = 4$ lignes et $p = 3$ colonnes.

Voici le code Python :

```
1 t=[[2, 6, 9], [3, 3, 2], [4, 7, 5], [0, 0, 7]]
2 n=len(t)
3 p=len(t[0])
4 print(n,p)
```

```
5 4 3
```

— Ligne 2 : le nombre de lignes est le nombre d'éléments de `t`

— Ligne 3 : le nombre de colonnes est le nombre d'éléments de n'importe quelle ligne (puisque toutes les lignes ont même nombre d'éléments), et donc comme il y a au moins une ligne dans un tableau 2D, on cherche le nombre d'éléments de la première ligne (indexée par 0).

— Ligne 4 : affichage des dimensions.

Accès à une colonne complète

On accède facilement à une ligne d'un tableau. Par exemple, la 2e ligne du tableau `t` est `t[1]`.
On ne peut pas accéder à une colonne aussi facilement. On indexe les colonnes à partir de zéro et on reprend le tableau t ci-dessus :

```
2 6 9
3 3 2
4 7 5
0 0 7
```

Cherchons à accéder à la colonne d'indice 2, constituée de 9, 2, 5, 7. Les éléments de cette colonne sont les éléments du tableau de la forme `t[i][2]` où `i` est un indice de ligne et qui varie donc de 0 à 3, d'où le code suivant :

```
t=[[2, 6, 9], [3, 3, 2], [4, 7, 5], [0, 0, 7]]
col = 2
print(t[0][col], t[1][col], t[2][col], t[3][col])
```

```
9 2 5 7
```

Listes en compréhension

Étant donné une liste d'entiers telle que `t=[5, 2, 0, 3, 7, 10]`, on cherche à construire la liste `L` dont les éléments sont 10 fois les éléments de `t`, c'est-à-dire `[50, 20, 0, 30, 70, 100]`.
Voici un code Python répondant au problème :

```
1 t = [5, 2, 0, 3, 7, 10]
2 print(t)
3 L = [10 * x for x in t]
4 print(L)
```

```
5 [5, 2, 0, 3, 7, 10]
6 [50, 20, 0, 30, 70, 100]
```

A la ligne 3, est définie une liste `L` dite *liste en compréhension.*
Une liste en compréhension `L` a la syntaxe minimale suivante

```
[expr for x in t]
```

où

- la paire de crochets, les mots-clefs `for` et `in` sont obligatoires
- `t` est, par exemple, une liste (voir plus bas pour d'autres possibilités)
- `x` est l'élément courant qui parcourt la liste `t` ; `x` est appelé *variable de contrôle* de la liste en compréhension
- `expr` est une expression qui dépend en général de `x` et dont la valeur est placée dans `L`

Si `t` est un conteneur (une liste, une chaîne, etc), la liste en compréhension `L` avec la syntaxe ci-dessus a toujours même nombre d'éléments que le conteneur `t`.

Intérêt d'une liste en compréhension : générer une liste en une seule *expression* et non en une ou plusieurs *instructions*. L'intérêt des listes en compréhension est avant tout leur compacité d'édition dans le code et leur bonne lisibilité.

Vocabulaire : la documentation officielle en français a utilisé jusqu'à la version 3.9 de Python le terme assez incompréhensible de « compréhension de liste » pour finalement adopter le terme de « liste en compréhension ». On rencontre aussi parfois le terme de *liste en intension* censé s'opposer à l'expression *liste en extension* ce dernier signifiant, grosso modo, liste littérale, voir cette discussion.

Liste en compréhension et boucle `for`

Une liste en compréhension admet un équivalent créé avec une boucle `for` mais nécessitant un code plus long.

Soit la liste en compréhension `L` suivante :

```
t = [5, 2, 0, 3, 7, 10]
L = [10 * x for x in t]
print(L)
```

```
[50, 20, 0, 30, 70, 100]
```

Elle pourrait être obtenue avec une boucle `for` :

```
t = [5, 2, 0, 3, 7, 10]
L = []
for x in t:
    L.append(10*x)
print(L)
```

```
[50, 20, 0, 30, 70, 100]
```

— La liste `L` est obtenue en trois lignes de code au lieu d'une.

Création d'un tableau 2D initialisé

Une liste en compréhension est un moyen **très simple** de créer un conteneur pour un tableau ayant n lignes et p colonnes. Le code ci-dessous crée un tableau à 2 lignes et 3 colonnes initialisé avec des 0 :

```
L = [[0 for j in range(3)] for i in range(2)]
print(L)
L[1][2]=42
print(L)
```

```
[[0, 0, 0], [0, 0, 0]]
[[0, 0, 0], [0, 0, 42]]
```

— On crée une liste de 2 lignes, chaque ligne étant une liste de 3 éléments. L'indice `i` parcourt les lignes et l'indice `j` parcourt chaque colonne de chaque ligne.

Au passage, comme l'élément le plus interne dans les listes est immuable (il s'agit de l'entier 0), on peut même simplifier la syntaxe :

```
L = [[0]*3 for i in range(2)]
print(L)
L[1][2]=42
print(L)
```

```
[[0, 0, 0], [0, 0, 0]]
[[0, 0, 0], [0, 0, 42]]
```

Confusion possible

Attention toutefois que le code suivant lui ne donne pas le résultat attendu :

```
L = [[0]*3]*2
print(L)
L[1][2]=42
print(L)
```

```
[[0, 0, 0], [0, 0, 0]]
[[0, 0, 42], [0, 0, 42]]
```

Code équivalent

Voici l'équivalent du code correct donné en début de section **sans utiliser** de liste en compréhension et en utilisant deux boucles for imbriquées. On notera que le code est plus long :

```
L=[]
for i in range(2):
    lig = []
    for j in range(3):
        lig.append(0)
    L.append(lig)
print(L)
```

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

Listes en compréhension et la clause `if`

Il existe une variante dans la syntaxe des listes de compréhension utilisant une clause `if`.

Par exemple, étant donné une liste `L` d'entiers, soit à construire la liste `t` obtenue en ne gardant que les entiers x de `L` tel que $x > 42$:

```
t = [65, 31, 9, 32, 81, 82, 46, 12]
L = [x for x in t if x >= 42]
print(L)
```

```
[65, 81, 82, 46]
```

— `x` varie dans `t` et est inséré dans la liste `L` en construction si $x \geq 42$.

On pourrait même écrire tout le code en une seule ligne.

L'équivalent avec une boucle `for` serait le suivant :

```
t = [65, 31, 9, 32, 81, 82, 46, 12]
L= []
for x in t:
    if x >= 42:
        L.append(x)
print(L)
```

Listes en compréhension imbriquées

Des listes en compréhension peuvent être imbriquées. Néanmoins la liste n'est pas forcément créée dans l'ordre où on s'y attendrait.

Soit à créer une liste formée de tous « mots » commençant par une lettre A ou B et suivie d'un chiffre parmi 1, 2 ou 3. On peut utiliser une liste en compréhension :

```
1 L = [x+y for x in "AB" for y in "123"]
2 print(L)
```

```
3 ['A1', 'A2', 'A3', 'B1', 'B2', 'B3']
```

Le résultat ligne 3 montre que l'on fixe d'abord la lettre, c'est-à-dire les éléments du `for` lexicalement le plus interne (ici `for x`) puis que y varie. Ce n'est pas forcément très intuitif.

Le résultat est plus facilement compréhensible si la liste en compréhension est interprétée par deux boucles `for` imbriquées **exactement** dans l'ordre où on lit les apparitions des `for` dans la liste en compréhension, ce qui donne ici :

```
L = []
for x in "AB":
    for y in "123":
        L.append(x+y)
print(L)
```

```
['A1', 'A2', 'A3', 'B1', 'B2', 'B3']
```

— la boucle `for x` apparaît avant la boucle `for y`, dans le même ordre que la liste en compréhension `[x+y for x in "AB" for y in "123"]`.

Notion de matrice

Une matrice est un objet mathématique s'assimilant à un tableau à double entrée contenant des objets (en général des nombres) en sorte que chaque objet de la matrice soit accessible à l'aide d'un couple d'indices entiers. En pratique, on se représente une matrice par un tableau 2D.

Ainsi, on peut représenter une matrice M à coefficients entiers ayant 2 lignes et 3 colonnes par le tableau :

$$M = \begin{pmatrix} 13 & 12 & 31 \\ 81 & 77 & 42 \end{pmatrix}$$

En mathématiques, l'usage est de placer des parenthèses autour du tableau. En mathématiques encore, l'élément de la matrice M placé à la i-ème ligne et à la j-ème colonne est noté $M_{i,j}$ avec un double indice. Par exemple, avec la matrice ci-dessus, on a $M_{2,3} = 42$ qui est l'élément à la deuxième ligne et troisième colonne.

Dans un contexte mathématique, l'indexation des lignes commence à 1 et non à 0. Dans un contexte de programmation Python (mais aussi Java, C/C++, Javascript, etc), l'indexation des lignes et des colonnes **commence à zéro**.

Conventionnellement, étant donné une matrice ou encore un terme d'une matrice, on donne d'abord l'indice de ligne PUIS l'indice de colonne. Sauf contexte particulier, je désignerai toujours le nombre de lignes d'une matrice M par n et le nombre de colonnes par p où n, p sont des entiers strictement positifs et on dit alors que M est de taille $n \times p$. Ci-dessus, la matrice donnée en exemple est de taille 2×3.

Ce qui distingue conceptuellement les matrices de simples tableaux 2D est qu'on associe à une matrice un certain nombre d'opérations mathématiques comme l'addition ou le produit.

Les matrices sont des outils très utilisés dans certains domaines de l'informatique qui nécessitent de la théorie des graphes ou de la recherche opérationnelle.

Matrices égales

Deux matrices M et N sont considérées comme égales lorsque :

- elles ont le même nombre de lignes n
- elles ont le même nombre de colonnes p
- pour chaque indice $i \in \{1, \ldots, n\}$ et $j \in \{1, \ldots, p\}$ on a $M_{i,j} = N_{i,j}$.

Matrice carrée, diagonale principale

Une *matrice carrée* est une matrice de taille $n \times n$ (autant de lignes que de colonnes) et on parle dans ce cas de matrice carrée d'ordre n.

On appelle *diagonale* d'une matrice carrée M la liste des coefficients situés sur la diagonale commençant en haut à gauche, à l'indice $(0, 0)$, et se terminant en bas à droite, à l'indice $(n - 1, n - 1)$. Les termes de la diagonale de M sont les $M_{i,i}$. Cette diagonale est dite *principale*. L'autre diagonale est dite *secondaire*.

Les matrices sous le module Numpy

Numpy est un module Python utilisé pour faire efficacement du calcul sur des tableaux multidimensionnels. Son usage est très répandu en calcul numérique et en apprentissage automatique, souvent couplé à d'autres bibliothèques Python.

Nous ne l'utiliserons que pour créer, manipuler et afficher commodément des matrices (des tableaux à deux dimensions). Nous n'utiliserons pas les opérations que met Numpy à disposition pour faire du calcul sur des matrices. La plupart du temps les matrices seront à coefficients entiers.

Accès à Numpy

Numpy n'est pas un module standard Python, donc pour l'utiliser, il faut que la bibliothèque ait été installée dans l'environnement de travail (et une fois installée, il faudra l'importer dans son fichier de travail).

Si vous utilisez Python en ligne avec Google Colab ou Cocalc, Numpy est déjà installé et directement utilisable.

Si vous utilisez l'environnement de bureau Anaconda, Numpy aura été installé.

Sinon, vous pouvez installer Numpy via pip en ligne de commande sous les 3 OS de bureau majoritaires. La commande est tout simplement

```
pip install numpy
```

La documentation officielle pour l'installation est ICI.

Importation de Numpy

Pour travailler avec Numpy, la méthode la plus usuelle consiste à l'importer en plaçant la ligne suivante dans votre code :

```
import numpy as np
```

(noter que l'initiale de Numpy s'écrit ici en minuscule).

Ensuite, pour utiliser une fonctionnalité de Numpy, il suffira de la préfixer par np, par exemple

```
import numpy as np

u=np.ones((3, 2), dtype='int')
print(u)
```

```
[[1 1]
 [1 1]
 [1 1]]
```

où la fonction ones est appelée préfixée par np.

Création de matrices sous Numpy

Pour définir une matrice sous Numpy, on définit en fait un tableau (*array*) à deux dimensions. Il existe de nombreuses façons de le faire et on n'en donnera ici que deux.

Si on connaît les coefficients de la matrice, sous forme d'une liste de listes de nombres, on les transmet en argument au contructeur np.array et on obtient une matrice :

```
1 import numpy as np
2
3 L=[[5, 81], [12,2030], [21, 12]]
4 M=np.array(L)
5
6 print(M)
```

```
7 [[   5   81]
8  [  12 2030]
9  [  21   12]]
```

La liste donnée doit comporter des listes toutes de la même dimension.

L'autre procédé consiste à créer une matrice ayant la dimension souhaitée mais remplie de zéros :

```
import numpy as np

M=np.zeros((3, 4),  dtype='int')
N=np.zeros((2, 3))
print(M)
print()
print(N)
```

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

[[0. 0. 0.]
 [0. 0. 0.]]
```

Cette matrice est dite *matrice nulle*.

Si on précise l'option `dtype=int`, la matrice sera considérée comme matrice à coefficients entiers. Sinon, c'est le comportement par défaut, la matrice est à coefficients flottants.

La taille est donnée par une liste ou un tuple `(n, p)` de deux entiers désignant d'abord le nombre `n` de lignes puis le nombre `p` de colonnes.

Ultérieurement, on peut « remplir » la matrice avec des coefficients que l'on choisira en fonction des besoins.

Dimensions d'une matrice sous Numpy

Les dimensions d'une matrice s'obtiennent avec l'attribut `shape` de la matrice :

```
import numpy as np

L=[[5, 81], [12,2030], [21, 12]]
M=np.array(L)

dim=M.shape
print(dim)
print(dim[0])
print(dim[1])
```

```
(3, 2)
```

qui renvoie un tuple composé du nombre de lignes et du nombre de colonnes (noter que `shape` n'est pas suivi d'une paire de parenthèses, ce n'est pas une méthode).

Une matrice *carrée* d'ordre `n` est simplement une matrice qui a même nombre `n` de lignes et de colonnes.

Accès aux éléments d'une matrice

L'indexation des lignes et des colonnes, comme pour une liste, commence par 0. Pour accéder à l'élément de la matrice `M` situé en ligne d'indice `i` et en colonne d'indice `j`, on utilise la notation `M[i,j]` :

```
import numpy as np

L=[[4, 9, 13, 7], [14, 20, 4, 16], [7, 3, 15, 1]]
M=np.array(L)
print(M)

print(M[1,2])
```

```
[[ 4  9 13  7]
 [14 20  4 16]
 [ 7  3 15  1]]
4
```

On peut aussi modifier une matrice à une position ligne x colonne donnée, en écrasant la valeur présente par une valeur donnée. Il suffit d'affecter l'élément de la matrice concerné :

```
import numpy as np

L=[[4, 9, 13, 7], [14, 20, 4, 16], [7, 3, 15, 1]]
M=np.array(L)
print(M)

print()
print(M[1,2])
print()

M[1,2]=1000
print(M)
```

```
[[ 4  9 13  7]
 [14 20  4 16]
 [ 7  3 15  1]]
4
```

Accès à une ligne donnée

En fait, une matrice Numpy est un tableau de tableaux 1D. Donc, si M est une matrice Numpy, `M[i]` donne accès à la ligne d'indice `i` :

```
import numpy as np

L=[[4, 9, 13, 7], [14, 20, 4, 16], [7, 3, 15, 1]]
M=np.array(L)
print(M)

print()
print(M[2])
```

```
[[ 4  9 13  7]
 [14 20  4 16]
 [ 7  3 15  1]]
```

```
[ 7  3 15  1]
```

Il est possible d'accéder à une colonne donnée mais ce ne sera pas considéré dans ce cours.

Matrice aléatoire

Pour faire des tests ou obtenir des exemples, il peut être utile de savoir générer une matrice aléatoire de taille donnée. Voici un exemple :

```
import numpy as np

M=np.random.randint(10, 21, size=(2, 3))
print(M)
```

```
[[20 20 12]
 [15 14 14]]
```

Ici, on a généré une matrice à coefficients entiers aléatoires entre 10 (inclus) et 21 (exclu) et de dimension 2 x 3.

Parcours de tous les coefficients d'une matrice

Pour parcourir une matrice coefficient par coefficient, on utilise deux boucles `for` imbriquées. On peut parcourir la matrice suivant ses lignes ou suivant ses colonnes :

```
import numpy as np

n,p =2,3
M=np.random.randint(10, 21, size=(n, p))
print(M)
print()

# Parcours par lignes
for i in range(n):
    for j in range(p):
        print(M[i,j])
    print(".....")
print("---------------")
print(M)
print()
# Parcours par colonnes
for j in range(p):
    for i in range(n):
        print(M[i,j])
    print(".....")
```

```
[[10 14 13]
 [13 13 13]]

10
14
13
```

```
..........
13
13
13
..........
---------------
[[10 14 13]
 [13 13 13]]

10
13
..........
14
13
..........
13
13
..........
```

Parcours sans indice

On peut aussi parcourir une matrice par ligne et sans utiliser d'indices :

```
import numpy as np

n,p =2,3
M=np.random.randint(10, 21, size=(n, p))
print(M)
print()

# Parcours par lignes
# sans indice

for L in M:
    for z in L:
        print(z)
    print("..........")
```

```
[[20 15 18]
 [19 16 13]]

20
15
18
..........
19
16
13
..........
```

Comparer deux matrices

Pour comparer deux matrices M et N, on n'utilise pas l'opération `M == N` mais la fonction `array_equal` :

```
import numpy as np

u=np.ones((3, 2), dtype='int')
v=np.ones((3, 2))

print(u)
print()
print(v)
print()

print(np.array_equal(u, v))
```

```
[[1 1]
 [1 1]
 [1 1]]

[[1. 1.]
 [1. 1.]
 [1. 1.]]

True
```

Exercice type – Éléments central ou centraux

Cet exercice est corrigé en vidéo ICI
On se donne une liste non vide d'entiers L. On appelle *éléments centraux* d'une liste :

— l'élément situé au milieu de la liste si la liste contient un nombre impair d'éléments

— les deux éléments centraux de la liste si la liste contient un nombre pair d'éléments.

Par exemple :

Liste	Éléments centraux
[31, 12, 81, 9, 65]	81
[31, 12, 81, 9, 65, 32]	81 9
[31, 12]	31 12
[81]	81

On demande d'écrire un code qui partant d'une liste L affiche l'élément central ou les éléments centraux.

Solution

Si la liste contient un nombre impair n d'entiers, par exemple la liste suivante pour $n = 15$:

```
31 12 81 99 65 44 58 63 42 21 33 16 14 81 50
```

alors il y a un unique élément central. Quel est son indice `i` ? Dans notre exemple, c'est l'élément qui vaut 63 et son indice est 7. Dans le cas général, comme

$$\texttt{n} = \texttt{k} + 1 + \texttt{k}$$

on voit bien que l'élément central est le $\texttt{k}+1$-ème élément de la liste et donc son indice `i` est 1 de moins que son rang c'est-à-dire $\texttt{i} = \texttt{k}$.

Comme $\texttt{n} = 2\texttt{k} + 1$, on a $\texttt{n}//2 = \texttt{k}$ donc l'élément central de `L` est $\texttt{L}[\texttt{n}//2]$.

Si la liste contient un nombre pair `n` d'entiers, par exemple la liste suivante pour $\texttt{n} = 16$:

```
31 12 81 99 65 44 58 63 42 21 33 16 14 81 50 96
```

alors il y a deux éléments centraux. Quel sont leurs indices ? Dans notre exemple, ce sont les éléments qui valent 63 et 42 donc d'indice 7 et 8. Dans le cas général, comme

$\texttt{n} = \texttt{k} + \texttt{k}$

on voit bien que les éléments centraux sont aux rangs `k` et $\texttt{k}+1$ et donc d'indices $\texttt{k}-1$ et `k`. Or `k` représente $\texttt{n}//2$ et donc les éléments centraux sont

$$\texttt{L}[\texttt{n}//2 - 1] \text{ et } \texttt{L}[\texttt{n}//2].$$

D'où le code :

```
L = [31, 12]
L=[81]
L = [31, 12, 81, 9, 65]
L = [31, 12, 81, 9, 65, 32]

n=len(L)
if n%2==1:
    print(L[n//2])
else :
    print(L[n//2-1], L[n//2])
```

```
81 9
```

Exercice type – Générer la matrice identité

On appelle matrice-identité d'ordre n la matrice carrée d'ordre n dont tous les coefficients sont nuls sauf ceux de la diagonale qui valent 1.
Écrire un code qui partant d'un entier $\texttt{n} > 0$ génère la matrice-identité `In`. Pour cela, on créera à l'aide de Numpy une matrice remplie de zéros et ayant la taille qu'il faut et on placera des 1 sur la diagonale principale de cette matrice
Une fois le code demandé écrit, on prendra en compte qu'en Numpy, la matrice identité s'obtient avec la fonction eye que l'on consultera pour voir comment elle fonctionne. **Vérifier** sur des exemples avec cette dernière fonction que votre code renvoie la bonne réponse.

Solution

Voici le code commenté :

```
1 import numpy as np
2
3 n=5
4 In=np.zeros((n, n), dtype='int')
5
6 for i in range(n):
7     In[i,i]=1
8
9 print(In)
10 Jn=np.eye(5, dtype='int')
11
12 print(np.array_equal(In, Jn))
```

```
13 [[1 0 0 0 0]
14  [0 1 0 0 0]
15  [0 0 1 0 0]
16  [0 0 0 1 0]
17  [0 0 0 0 1]]
18 True
```

— Ligne 1 : notre matrice sera une matrice Numpy, donc on importe ce module.

— Ligne 3 : la taille de la matrice

— Ligne 4 : on crée une matrice Numpy `In`, carrée de taille `n x n` d'où l'argument `(n, n)`. On utilise la fonction `np.zeros` qui crée une matrice formée uniquement de coefficients 0. L'argument `dtype="int"` aura pour conséquence que ses coefficients seront vus comme des entiers ce qui produira un affichage plus sobre. Noter les guillemets autour de `int` mais en fait ne sont pas indispensables ici.

— Lignes 6-7 : Il s'agit d'écrire les 1 sur la diagonale. Pour cela, il est **inutile** de parcourir *toute* la matrice `In`, il suffit de se placer aux bons endroits (la diagonale principale) et de placer le 1 voulu. Le bon endroit est chaque position `(i, i)` de la matrice.

— Ligne 7 : noter qu'on accède à la position avec un double indice `i, i` qui pourrait encore s'écrire `(i, i)`

— Lignes 9 et 13-17 : on affiche le résultat et on voit qu'il correspond à ce qui était attendu.

— Ligne 10 : on demande à Numpy de créer lui-même la même matrice avec la fonction `eye`

— Ligne 12 : Python nous confirme que notre matrice `In` et la matrice produite par Numpy sont bien égales (noter l'usage de la fonction `np.array_equal`).

EXERCICES

Liste de trois éléments

On donne un entier x et on compte de 10 en 10 trois fois à partir de x. Construire la liste L qui contient les trois nombres obtenus. Par exemple, si x=42, la liste est formée des trois entiers 42, 52 et 62.

Liste : premier et dernier

On se donne une liste d'entiers `L`. Afficher sur une même ligne et séparés par une espace le premier élément de la liste et le dernier élément de la liste. Par exemple,

— si `L = [31, 12, 81, 9, 65]` alors l'affichage est : `31 65`

— si `L = [42]` alors l'affichage est : `42 42`

Liste : k-ème élément

On se donne une liste d'entiers `L`. On suppose que la liste `L` contient au moins deux éléments. On donne un entier $k \geq 1$. Afficher le k-ème élément de la liste si cet élément existe sinon, on affiche le message *rang invalide*.

Exemples d'exécution :

— `L = [31, 12, 81, 9, 65], k = 3 -> 81`

— `L = [31, 12, 81, 9, 65], k = 42 -> rang invalide`

Liste des solutions de l'équation du second degré

Soit l'équation du second degré $ax^2 + bx + c = 0$. On supposera que a, b, c sont des entiers, que a est non nul et que l'inconnue x est un nombre réel.

On rappelle le code Python de la résolution d'une équation du second degré :

```
a = 2
b = -3
c = 1
delta = b*b -4*a*c

if delta >0:
    print("2 solutions distinctes :")
    print("x1 =", (-b - delta**0.5)/(2*a))
    print("x2 =", (-b + delta**0.5)/(2*a))
else:
    if delta == 0:
        print("une seule solution")
        print("x =", -b/(2*a))
    else:
        print("Aucune solution")
```

ou mieux, avec une instruction `elif` :

```
a = 2
b = -3
c = 1
delta = b*b -4*a*c

if delta >0:
    print("2 solutions distinctes :")
    print("x1 =", (-b - delta**0.5)/(2*a))
    print("x2 =", (-b + delta**0.5)/(2*a))
elif delta == 0:
    print("une seule solution")
    print("x =", -b/(2*a))
else:
    print("Aucune solution")
```

Adapter le code précédent pour construire la liste S contenant toutes les solutions de l'équation. Par exemple,

- si a = 2, b=-3 et c = 1 alors `S = [1, 0.5]`
- si a = 4, b=4 et c = 1 alors `S = [-0.5]`
- si a = 4, b=4 et c = 4 alors S est la liste vide

Changer certains éléments d'une liste

On donne une liste L d'entiers ayant au moins 3 valeurs. Modifier cette liste pour que les éléments x de la liste qui sont

- en première position
- en deuxième position
- en dernière position

soient remplacés par $10x$. Par exemple, si `L = [31, 12, 81, 9, 65]` alors, après modification, on aura `L = [310, 120, 81, 9, 650]`.

Echanger les extrémités d'une liste

Cette question nécessite d'adapter aux éléments d'une liste le code d'échange des valeurs de deux variables.

Soit une liste L. Modifier L pour que son premier élément et son dernier élément soient échangés. Par exemple, si `L = [31, 12, 9, 65, 81]` alors après échange, on aura `L = [81, 12, 9, 65, 31]`.

Kubernetes -> K8s

On donne une liste, par exemple `L=['P', 'y', 't', 'h', 'o', 'n']` et on demande de construire une nouvelle liste LL, dans notre exemple, ce sera `LL=['P', 4, 'n']`. La règle de construction de la liste LL est précisée ci-dessous :

- si L a moins de trois éléments alors LL et L ont même contenu (mais LL doit être une liste *différente* de L, autrement dit, avec le vocabulaire vu dans le cours, LL n'est pas un alias de L);

— sinon, LL a au moins trois éléments :

  — le premier et le dernier élément de LL sont identiques à ceux de L
  — l'élément central de LL est le nombre d'éléments de L **strictement** compris entre les extrémités de L

Voici quelques exemples de comportements :

```
[42, 1, 9, 4, 7, 5, 3, 6, 0, 2020] -> [42, 8, 2020]
[31, 81, 12] -> [31, 1, 12]
['X', 'Y'] -> ['X', 'Y']
['X'] -> ['X']
[] -> []
```

Tester les mois de 31 jours avec une liste

Construire une liste littérale `jours` telle que pour chaque numéro `i` entre 1 et 12 d'un mois d'une année non bissextile, `jours[i]` désigne le nombre de jours du mois de numéro `i`. Par exemple, `jours[10] = 31`.

En déduire une variable booléenne `mois31` qui, étant donné un numéro de mois `i`, vaut `True` si le mois de numéro `i` possède 31 jours et `False` sinon.

Test de présence

On donne

— une liste d'entiers `L`

— un entier `x`

On rappelle que `x in L` est un booléen qui teste la présence de `x` dans la liste `L`.

Construire un booléen `ok` qui vaut `True` si l'un des trois entiers `x`, `x - 1` et `x + 1` est dans la liste et `False` sinon. Exemple avec `x = 42` :

— si `L =[31, 12, 42, 9, 65]` alors `ok = True`

— si `L =[31, 12, 81, 43]` alors `ok = True`

— si `L =[41]` alors `ok = True`

— si `L =[31, 12, 81, 9, 65]` alors `ok = False`

Mois de 31 jours et liste

Cet exercice est corrigé en vidéo

On donne un numéro de mois entre 1 et 12. Créer une variable booléenne `est_mois_31` (prenant comme valeur True ou False) qui teste si `m` est le numéro d'un mois ayant 31 jours[2] comme janvier (numéro 1) ou juillet (numéro 7) mais pas février (numéro 2). On utilisera impérativement l'appartenance à une liste

Afficher toutes les dates de l'année

On vous demande d'afficher sur 365 lignes successives les dates de tous les jours de l'année. Pour simplifier on suppose que février est formé de 28 jours. Par exemple, si le jour de l'An est un vendredi, le code devra afficher :

2. Les mois ayant 31 jours sont : janvier (1), mars (3), mai (5), juillet (7), août (8), octobre (10), décembre (12).

```
vendredi 1 janvier
samedi 2 janvier
dimanche 3 janvier
lundi 4 janvier
mardi 5 janvier
mercredi 6 janvier
....
....
vendredi 24 décembre
samedi 25 décembre
dimanche 26 décembre
lundi 27 décembre
mardi 28 décembre
mercredi 29 décembre
jeudi 30 décembre
vendredi 31 décembre
```

On utilisera une liste pour les noms des jours de la semaine et une liste pour le nom des mois.

Lister les non multiples

Utiliser une liste en compréhension pour construire la liste des entiers entre 1 et 20000 qui ne sont ni pairs ni multiples de 5. Par exemple, 421 ou 2023 feront partie de la liste des entiers concernés mais pas 42 ni 2025. La longueur de la liste à trouver est de 8000.

Liste d'entiers consécutifs décroissants et alternant de signe

On donne deux entiers a et b avec $a \geq b \geq 0$, par exemple a=2035 et b=2025. En utilisant une liste en compréhension, on demande de créer la liste des entiers consécutifs depuis a jusqu'à b en ordre décroissant et en alternant les signes, a devant apparaître positivement. Par exemple, si a=2035 et b=2025, la liste à contruire est

```
[2035, -2034, 2033, -2032, 2031, -2030, 2029, -2028, 2027, -2026, 2025]
```

Liste des entiers impairs suivi des pairs

On pourra utiliser que si L et M sont des listes alors L+M est une nouvelle liste qui donne accès au contenu de L suivi du contenu de M.

On donne un entier n ≥ 1 on demande d'écrire une liste L formée d'abord de tous les entiers impairs entre 1 et n puis de tous les entiers pairs. Par exemple, si n $= 6$, on obtiendra la liste L $= [1, 3, 5, 2, 4, 6]$ et si n $= 5$, on obtiendra la liste L $= [1, 3, 5, 2, 4]$.

Le code de construction de la liste doit tenir en une seule ligne et utilisera des listes en compréhension.

Liste des carrés entre deux nombres

On donne deux entiers positifs a et b avec a $\leq$ b. Créer, en utilisant une liste en compréhension, la liste croissante des carrés parfaits entre a et b.

Si a=2000 et b=3000, on trouvera la liste suivante

```
[2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809]
```

Si a=2025 et b=2500, on trouvera la liste suivante

```
[2025, 2116, 2209, 2304, 2401, 2500]
```

La liste obtenue peut très bien être vide.

Séparer en indices pairs et indices impairs

On pourra utiliser que si L1 et L2 sont des listes alors L1 + L2 est une nouvelle liste qui donne accès au contenu de L1 suivi du contenu de L2.

Soit une liste L. En utilisant une liste en compréhension, séparer cette liste en deux listes :

— la liste IP des éléments de L d'indices pairs

— la liste II des éléments de L d'indices impairs

Par exemple, si L = [31, 12, 9, 65, 81, 42] alors IP = [31, 9, 81] et II = [12, 65, 42].

Tables de multiplication via des listes en compréhension

On représente la table de multiplication d'un entier m par une liste de ces 10 premiers multiples m, $2m$, $3m$, etc ; par exemple la table de 7 est la liste

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Des tables de multiplication sont représentées par une liste de tables de multiplication et donc par une liste de listes.

Ecrire un code qui à partir d'une liste L d'entiers strictement positifs renvoie les tables de multiplication des éléments de L. Par exemple, si L=[5, 0, 7, 42] alors la table sera la liste suivante :

```
[[5, 10, 15, 20, 25, 30, 35, 40, 45, 50],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70],
[42, 84, 126, 168, 210, 252, 294, 336, 378, 420]]
```

Décompression de liste via des listes en compréhension

On donne une liste L de listes de deux éléments, par exemple

```
L=[["Pomme", 2], ["Kiwi", 3], ["Orange", 0], ["Poire", 2]]
```

Chaque liste de deux éléments dans L est formée d'une chaîne de caractères s et d'un entier positif ou nul n. On demande de contruire une liste M de chaînes contenant pour chaque liste [s, n] figurant dans L, la chaîne s mais répétée n fois.

Avec l'exemple ci-dessus, la liste M attendue est :

```
['Pomme', 'Pomme', 'Kiwi', 'Kiwi', 'Kiwi', 'Poire', 'Poire']
```

On utilisera une liste en compréhension ainsi que le décompactage de la variable de contrôle d'une boucle for.

Liste des diviseurs via des listes en compréhension

On donne une liste L d'entiers strictement positifs et on demande d'écrire un code qui génère une liste D des listes de diviseurs de chaque entier de L. On rappelle que d est un diviseur de N signifie juste que N est un multiple de d autrement dit que $N = d \times q$ pour un certain entier q. Par exemple, si L=[2049,2027, 43, 25] alors D=[[1, 3, 683, 2049], [1, 2027], [1, 43], [1, 5, 25]

Les diviseurs apparaîtront dans l'ordre croissant.

Voisins d'un coefficient d'une matrice

Etant donné une matrice `A`, on appelle *voisin* d'un élément `a` de cette matrice, tout élément de la matrice situé soit juste à droite, soit juste à gauche soit juste au-dessus soit juste en-dessous de `a`. Par exemple si

$$\texttt{A} = \begin{pmatrix} 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{pmatrix}$$

alors :

— 9 a deux voisins (2 et 3)

— 22 a trois voisins (20, 16 et 3)

— 20 a quatre voisins (14, 13, 21 et 22)

On rappelle que les indices de lignes ou de colonnes sont comptés à partir de 0.

1. Ecrire un code qui, partant de la matrice `A` et de deux indices valides `i` et `j` de la matrice, construit une liste de tous les voisins de l'élément `A[i][j]` de la matrice `A`. On pourra supposer que `i` et `j` représentent des indices valides de `A`. L'ordre des éléments dans la liste renvoyée sera sans importance.
   Avec la matrice `A` ci-dessus, on obtiendra par exemple que
   — pour `(i, j) = (3,4)` la liste cherchée est `[2, 3]`
   — pour `(i, j) = (1,3)` la liste cherchée est `[14, 13, 21, 22]`
   De même, si `A` est la matrice représentée ci-dessous :

$$\texttt{A} = \begin{pmatrix} 23 \\ 4 \\ 10 \\ 11 \end{pmatrix}$$

   alors
   — pour `(i, j) = (1,0)` la liste cherchée est `[23, 10]`
   — pour `(i, j) = (0,0)` la liste cherchée est `[4]`
   Votre code doit être capable de traiter n'importe quelle matrice, y compris une matrice de taille 1×1.
2. Ecrire un code qui, à partir d'une matrice `A` d'entiers, renvoie `V`, une liste de listes de listes et telle que si `i` et `j` sont des indices valides de `A` alors `V[i][j]` est la liste des voisins de `A[i][j]` dans la matrice `A`.

Extraction d'une colonne d'une matrice

Soit une matrice `A`. Écrire un code qui à partir d'un entier positif `j` renvoie, sous forme de liste, la colonne d'indice `j` de la matrice `A`. On suppose que l'indice de colonne est valide.

Par exemple si

$$A = \begin{pmatrix} 71 & 21 & 96 & 90 & 42 \\ 10 & 18 & 51 & 98 & 15 \\ 93 & 93 & 84 & 13 & 95 \\ 39 & 50 & 47 & 65 & 90 \end{pmatrix}$$

et si `j` $= 3$ alors la colonne demandée est

```
[90, 98, 13, 65]
```

Autant que possible, on utilisera une liste en compréhension. On testera avec une matrice aléatoire.

Trace d'une matrice

Si une matrice M est carrée, on appelle trace de M la somme des coefficients de la diagonale descendante de M issue du coefficient en position (0,0).

Ecrire un code qui partant d'une matrice carrée calcule sa trace. Par exemple, si

$$M = \begin{pmatrix} 17 & 14 & 11 \\ 10 & 10 & 79 \\ 88 & 52 & 49 \end{pmatrix}$$

alors sa trace vaut 76.

Vérifier en appelant la méthode `M.trace`.

Transposée d'une matrice

On donne une matrice A ayant `n` lignes et `p` colonnes. On demande de construire la matrice `B` dite *transposée* de `A` et qui s'obtient ainsi : chaque ligne de `B` est la colonne de `A` ayant même indice que la ligne. Par exemple, si $A = \begin{pmatrix} 4 & 4 & 8 & 2 \\ 4 & 0 & 0 & 4 \\ 9 & 3 & 2 & 0 \end{pmatrix}$

alors $B = \begin{pmatrix} 4 & 4 & 9 \\ 4 & 0 & 3 \\ 8 & 0 & 2 \\ 2 & 4 & 0 \end{pmatrix}$

Etre une matrice diagonale

Une matrice `D` est dite diagonale si elle est carrée et si les éléments hors de la diagonale dont nuls. Ecrire une code qui teste sir une matrice à coefficients entiers donnée est diagonale ou non.

Etre une matrice triangulaire supérieure

Soit `M` une matrice carrée d'ordre `n`. On dit que `M` est une matrice *triangulaire supérieure* si tous les coefficients de `M` situés strictement sous la diagonale sont nuls.

1. Ecrire un code qui définisse un booléen `estTriangulaireSup` qui vaut `True` si `M` est une matrice triangulaire supérieure et `False` sinon.
2. Ecrire un code qui génère une matrice triangulaire supérieure d'ordre `n` donné.

Etre une matrice scalaire

Une matrice `M` est dite *scalaire* si les trois conditions suivantes sont satisfaites :

- elle est carrée
- les éléments hors de la diagonale (principale) sont nuls
- les éléments de la diagonale ont tous la même valeur.

Une matrice `M` étant donnée, écrire un booléen `estScalaire` qui vaut `True` si `M` est une matrice scalaire et `False` sinon.

Les trois matrices suivantes ne sont pas des matrices scalaires :

$$\begin{pmatrix} 5 & 0 & 2 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{pmatrix}, \quad \begin{pmatrix} 5 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 5 \end{pmatrix}, \quad \begin{pmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \end{pmatrix}.$$

La matrice suivante est scalaire :

$$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{pmatrix}.$$

Générer une matrice diagonale

On donne une liste `L` de `n` entiers. Ecrire la matrice `D` carrée d'ordre `n` telle que

- la diagonale (principale) de `D` est identique à la liste `L`
- les termes non diagonaux de `D` sont nuls.

Anti-transposée d'une matrice

Soit une matrice carrée `M`. On appelle *diagonale secondaire* de `M` la suite de coefficients de `M` allant du coin inférieur gauche de `M` vers le coin supérieur droit de `M`. Par exemple, si `M` est la matrice

$$\begin{pmatrix} 5 & 3 & 2 \\ 6 & 4 & 0 \\ 7 & 1 & 9 \end{pmatrix}$$

la diagonale secondaire de `M` est `7, 4, 2`.

On appelle *anti-transposée* d'une matrice carrée `M` la matrice `N` dont les coefficients sont les symétriques des coefficients de `M` par rapport à la diagonale secondaire. Par exemple, si `M` est la matrice
$\begin{pmatrix} 5 & 3 & 2 \\ 6 & 4 & 0 \\ 7 & 1 & 9 \end{pmatrix}$ alors son anti-transposée est $\begin{pmatrix} 9 & 0 & 2 \\ 1 & 4 & 3 \\ 7 & 6 & 5 \end{pmatrix}$

Ecrire un code qui à partir d'une matrice `M` construise l'anti-transposée `N` de la matrice carrée `M`. Pour cela, on procédera comme suit :

- on construira une matrice nulle `N` de même taille que `M`;
- soit une position (ligne, colonne) dans `M` d'indices `(a, b)` et soient `(c, d)` les indices de la position symétrique de `(a, b)` par rapport à la diagonale secondaire de `M`. Que valent alors `a + d` et `b + c` en fonction de la taille `n` de la matrice `M`?

Bords d'une matrice

Étant donné une matrice `M`, on appelle 1^{er} *bord* de `M` les coefficients de cette matrice qui sont soit sur la première ligne, soit sur la dernière ligne, soit sur la première colonne, soit sur la dernière colonne.

De même, on appelle 2^e bord de `M` les coefficients de `M` qui ne sont pas sur le 1^{er} bord et qui sont soit sur la 2^e ligne, soit sur la 2^e colonne soit sur l'avant-dernière ligne soit sur l'avant dernière colonne.

De manière analogue, on définit le 3^e, 4^e, etc bord de `M`.

Ecrire un code qui construit la matrice de taille `n` $\times$ `n` dont

— le 1^{er} bord est formé uniquement de coefficients valant 1,

— le 2^e bord uniquement de coefficients valant 2

et ainsi de suite jusqu'à ce que tous les coefficients de la matrice soient énumérés.

Par exemple, si `n` $=$ `5` le code doit produire la matrice suivante :

$$\texttt{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 3 & 2 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

ou encore si `n` $=$ `15` la matrice recherchée doit être la matrice suivante :

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 6 & 6 & 6 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 7 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 7 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 6 & 6 & 6 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 2 & 1 \\ 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Cet exercice est très formateur et vaut la peine de **chercher**. *Y arriver est surtout une affaire d'organisation et de méticulosité. Utilisez une feuille quadrillée. Cet exercice peut nécessiter pas loin d'une heure de recherche (mais il peut être fait en 10 minutes si vous êtes algorithmicien).*

Quelques indications :

— On commencera par créer une matrice d'ordre `n` remplie de `0`

— Regardez quelques exemples et vérifiez que si `n` est l'ordre de `M` alors le nombre de bords est `m = (n+1)//2`.

— Cela veut dire que si vous bouclez `m` fois avec une boucle `for`, vous pouvez remplir la matrice, bord après bord

— Pour remplir chaque bord avec la valeur disons `k`, utiliser la position ligne/colonne `(i, j)` du coin supérieur gauche du bord ; vous remarquerez que `i = j`. Vous noterez qu'il y a aussi un lien entre `i` et `k`.

— En connaissant la longueur du bord, vous pouvez calculer la position des quatre coins. Ensuite, avec 4 boucles for successives, vous placerez la valeur `k` dans les quatre côtés du bord (en fait, une seule boucle remplissant les 4 côtés du bord suffit).

Implémenter le produit de deux matrices

On va définir le produit de deux matrices M et N. Attention, cette opération n'est pas complètement intuitive.

Avant d'expliquer comment se calcule le produit $M \times N$ de deux matrices, il faut préciser que les entiers suivants doivent être égaux :

— nombre de colonnes de la première matrice M

— nombre de lignes de la deuxième matrice N.

Ainsi, le produit d'une matrice de taille 4×2 et d'une matrice de taille 2×3 aura bien un sens mais pas le produit d'une matrice 2×3 et d'une matrice 2×3.

Soient donc A une matrice de taille $n \times m$ et B une matrice de taille $m \times p$. On définit le produit $C := A \times B$ comme étant la matrice C de taille $n \times p$ telle que

$$c_{i,j} = a_{i,0} \times b_{0,j} + a_{i,1} \times b_{1,j} + \cdots + a_{i,m-1} \times b_{m-1,j}$$

Ainsi

$$\begin{pmatrix} 10 & 40 & 20 & 30 \\ 20 & 70 & 50 & 90 \end{pmatrix} \begin{pmatrix} 3 & 1 & 5 \\ 2 & 4 & 7 \\ 8 & 0 & 6 \\ 1 & 5 & 4 \end{pmatrix} = \begin{pmatrix} 300 & 320 & 570 \\ 690 & 750 & 1250 \end{pmatrix}$$

Comment est obtenu le coefficient 570 du produit AB à la ligne d'indice 0 et la colonne d'indice 2 ? On examine la ligne $\begin{pmatrix} 10 & 40 & 20 & 30 \end{pmatrix}$ d'indice 0 de A et la colonne $\begin{pmatrix} 5 \\ 7 \\ 6 \\ 4 \end{pmatrix}$ d'indice 2 de B et on fait le calcul suivant :

$$\begin{pmatrix} 10 & 40 & 20 & 30 \end{pmatrix} \times \begin{pmatrix} 5 \\ 7 \\ 6 \\ 4 \end{pmatrix} = 10 \times 5 + 40 \times 7 + 20 \times 6 + 30 \times 4 = 570$$

Voilà pourquoi on dit que le produit matriciel s'effectue *ligne par colonne*.

Ecrire le code d'une fonction `produit(A,B)` qui renvoie la matrice produit `AB`. On supposera que la fonction `produit(A,B)` ne reçoit que des matrices `A` et `B` de tailles permettant d'effectuer leur produit.

On testera les deux matrices montrées ci-dessus dont les coefficients sont extractibles ci-dessous :

```
[[10, 40, 20, 30 ],[20, 70, 50, 90 ]]
[[3, 1, 5 ],[2, 4, 7 ],[8, 0, 6 ],[1, 5, 4 ]]
```

On vérifiera la justesse du calcul réalisé en comparant avec le résultat donné par la fonction `np.matmul`.

Générer un motif de matrice : équerres

Etant donné un entier `n` > 0, on demande de construire la matrice `M` suivante :

$$\begin{pmatrix} 0 & 1 & 2 & \dots & n-2 & n-1 \\ 1 & 1 & 2 & \dots & n-2 & n-1 \\ 2 & 2 & 2 & \dots & n-2 & n-1 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ n-2 & n-2 & n-2 & \dots & n-2 & n-1 \\ n-1 & n-1 & n-1 & \dots & n-1 & n-1 \end{pmatrix}$$

Explication : pour tout tout indice `k` $\in \{0, \dots, n-1\}$, les `k` $+ 1$ premiers éléments de la ligne d'indice `k` de `M` ainsi que les `k` $+ 1$ premiers éléments de la colonne d'indice `k` de `M` valent tous `k`.

Générer une matrice tridiagonale

On donne trois entiers `a`, `b` et `c`. On se donne aussi un entier `n` > 0. Construire une matrice carrée `M` d'ordre `n` (dite « tridiagonale ») dont

- la diagonale juste au-dessus de la diagonale principale est formée uniquement du coefficient `a`,
- la diagonale principale est formée uniquement du coefficient `b`,
- la diagonale juste en-dessous de la diagonale principale est formée uniquement du coefficient `c`,
- les autres coefficients sont nuls

Par exemple, si `a` $= 4$, `b` $= 2$, `c` $= 1$ et `n` $= 4$ alors `M` est la matrice suivante

$$\begin{pmatrix} 2 & 4 & 0 & 0 \\ 1 & 2 & 4 & 0 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

On commencera par créer une matrice nulle `M` de taille `n` $\times$ `n` et on y placera ensuite les coefficients de la matrice à construire.

Insérer une ligne dans une matrice

On donne une matrice `M` de taille `n` lignes et `p` colonnes. On donne une liste `L` de `p` nombres et on donne aussi un indice `k` $\in \{0, \dots, n\}$. On demande de construire la matrice `N` obtenue en insérant, dans la matrice `M` une ligne dont les `p` coefficients sont formés des éléments de la liste `L` en sorte que l'indice de la nouvelle ligne dans `N` soit `k`.

Exemple : pour `k` $= 2$, on lit successivement `M` et `L` et la nouvelle matrice `N` :

```
[[70 88 65]
 [22 67 58]
 [55 44 36]
 [12 60 53]]

[[10 80 26]]

[[70 88 65]
```

```
[22 67 58]
[10 80 26]
[55 44 36]
[12 60 53]]
```

On procédera comme suit :

— construire une matrice nulle `N` ayant la bonne taille

— recopier là où il faut dans `N`, ligne par ligne, les coefficients demandés.

Insérer une colonne dans une matrice

On donne une matrice `M` de taille `n` lignes et `p` colonnes. On donne une liste `L` de `n` nombres et on donne aussi un indice de colonne `j` $\in \{1, \ldots, p\}$. On demande de construire la matrice `N` obtenue en insérant, dans la matrice `M`, entre la colonne d'indice `j` $-$ `1` et la colonne d'indice `j` de `M`, une colonne dont les `n` coefficients sont formés de éléments de la liste `L`. Il faudra créer une nouvelle matrice Par exemple, si `M` est la matrice

$$\begin{pmatrix} 4 & 6 & 7 & 3 \\ 5 & 0 & 1 & 2 \\ 4 & 4 & 9 & 3 \end{pmatrix}$$

et si `j` $=$ `2` et si `L` `=[2030, 2031, 2032]` alors `N` est la matrice

$$\begin{pmatrix} 4 & 6 & 2030 & 7 & 3 \\ 5 & 0 & 2031 & 1 & 2 \\ 4 & 4 & 2032 & 9 & 3 \end{pmatrix}$$

Inverser l'ordre des colonnes d'une matrice

On donne une matrice `M` et on demande de construire une nouvelle matrice `N` obtenue à partir de `M` en renversant l'ordre des colonnes. Par exemple, si `M` est la matrice

```
4 2 7 3
8 5 1 2
9 6 4 2
```

alors `N` est la matrice

```
3 7 2 4
2 1 5 8
2 4 6 9
```

Transformation de Sylvester

Etant donné une matrice carrée M de taille $n \times n$, on appelle transformée de Sylvester de M la matrice S suivante :

$$S = \left[\begin{array}{c|c} M & M \\ \hline M & -M \end{array}\right]$$

La matrice S est de de taille $2n \times 2n$.

Par exemple, si $M = \begin{pmatrix} 42 & 81 \\ 31 & 75 \end{pmatrix}$ alors sa transformée de Sylvester est

$$\left(\begin{array}{cc|cc} 42 & 81 & 42 & 81 \\ 31 & 75 & 31 & 75 \\ \hline 42 & 81 & -42 & -81 \\ 31 & 75 & -31 & -75 \end{array}\right)$$

Étant donné une matrice `M`, construire la transformée de Sylvester de `M`.

Remplissage d'une grille en hélice

On se propose de remplir une grille de dimension $n \times p$ par tous les entiers consécutifs entre 1 et np formant un motif en hélice. Voici le remplissage si $n = 6$ et $p = 7$:

$$\begin{pmatrix} 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 5 & 26 & 27 & 28 & 29 & 30 & 13 \\ 4 & 25 & 38 & 39 & 40 & 31 & 14 \\ 3 & 24 & 37 & 42 & 41 & 32 & 15 \\ 2 & 23 & 36 & 35 & 34 & 33 & 16 \\ 1 & 22 & 21 & 20 & 19 & 18 & 17 \end{pmatrix}$$

Le remplissage de la grille commence en bas à gauche et s'effectue vers le haut. On garde la direction courante sauf si la case suivante est hors de la grille ou si elle a déjà été remplie auquel cas on prend la direction à droite.

3 Parcourir, répéter

`append` : adjoindre un élément à une liste

La méthode `append` modifie une liste en lui ajoutant un élément à sa fin. Exemple :

```
1 L = [65, 31, 9]
2 print(L)
3 print("taille de L :", len(L))
4 print()
5 L.append(81)
6 print(L)
7 print("taille de L :", len(L))
```

```
8 [65, 31, 9]
9 taille de L : 3
10
11 [65, 31, 9, 81]
12 taille de L : 4
```

— Lignes 5, 11 : l'élément 81 est ajouté à la fin de la liste.

— Lignes 9, 12 : on voit aussi que la taille de la liste a changé.

— Lignes 5 et 6 : bien qu'ayant changée, la liste est toujours référencée par `L`.

On notera la syntaxe particulière de `append` utilisant un point (« la notation suffixe ») : chaque liste `L` admet un « attribut » `append` que l'on obtient par `L.append` et qui s'utilise comme une fonction.

La méthode append ne renvoie rien :

```
1 L = [65, 31, 9]
2 print(L)
3 x = L.append(81)
4 print(L)
5 print(x)
```

```
6 [65, 31, 9]
7 [65, 31, 9, 81]
8 None
```

— Ligne 3 : le membre de droite est un *appel* de la méthode append. Le membre de gauche référence le retour de cet appel.

— Ligne 8 : on observe qu'une fois l'appel à append effectué, x vaut `None` autrement dit *rien*.

Ainsi, contrairement à ce que l'on pourrait imaginer, `L.append(truc)` **ne** représente **pas** la liste L complétée par `truc`.

Adjoindre plusieurs éléments

Un unique appel à la méthode append ne permet pas d'ajouter plusieurs éléments. Pour ajouter plusieurs éléments, il faut appeler autant de fois la méthode append :

```
1 L = [65, 31, 9]
2 L.append(81)
3 L.append(100)
4 print(L)
```

```
5 [65, 31, 9, 81, 100]
```

Parcours complet d'une liste sans indice

Soit une liste de nombres, comme `L = [65, 31, 9, 32, 81, 82, 46, 12]`. On cherche à afficher le produit $10 \times z$ pour chaque terme z de la liste. La façon la plus simple d'y parvenir en Python est :

```
1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2
3 for z in L:
4     print(10 * z)
```

```
5 650
6 310
7 90
8 320
9 810
10 820
11 460
12 120
```

On observera qu'on accède aux éléments de la liste sans utiliser d'indice et juste en nommant l'élément courant. Cette syntaxe s'applique chaque fois qu'on cherche à **lire** successivement la **totalité** des éléments d'une liste. C'est un idiome de la programmation Python.

Cette syntaxe ne sera pas adaptée si :

— on veut **modifier** la liste

— si on ne veut lire qu'à certaines positions définies par un **indice** de la liste.

Beaucoup de débutants confondent le parcours d'une liste sans indice ou avec indice. Ce dernier parcours est considéré, dans certains cas, comme non pythonnique mais il a l'avantage de s'appliquer à de nombreuses situations.

Parcours complet d'une liste par indices

Comme on peut accéder à chaque élément d'une liste de n éléments par son indice i avec $0 \leq i < n$, un usage très fréquent [3] de la boucle `for` est le parcours d'une liste par ses indices. C'est un idiome de la programmation impérative.

Par exemple, soit une liste de nombres, comme `t = [65, 31, 9, 32, 81, 82, 46, 12]`. On cherche à afficher le produit `10 × x` pour chaque terme `x` de la liste :

```
1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2
3 for i in range(8):
4     print(10 * t[i])
```

```
5 650
6 310
7 90
8 320
9 810
10 820
11 460
12 120
```

— Ligne 1 : `t` contient 8 éléments donc pour parcourir `t` avec un indice `i`, on utilise l'en-tête `for i in range(8)`

— Ligne 4 : `t[i]` permet d'accéder à l'élément courant de `t`.

Code amélioré

Pour une meilleure lisibilité et une meilleure maintenance, un des deux codes ci-dessous est nettement préférable au code précédent :

```
1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2 n = len(t)
3
4 for i in range(n):
5     print(10 * t[i])
```

3. Mais parfois considéré comme non pythonique, cf. foreach loop et unpythonic loop.

```
6 t = [65, 31, 9, 32, 81, 82, 46, 12]
7
8 for i in range(len(t)):
9     print(10 * t[i])
```

— Ligne 2 ou ligne 8 : la longueur de `t` n'est pas écrite littéralement : si dans le code ligne 1, on modifie `t` en rajoutant un élément, le code en-dessous n'a pas besoin d'être modifié.

Boucle for sans indice et calcul de somme

La technique de la variable accumulatrice permet de calculer la somme des termes d'une liste sans utiliser d'indice des éléments de la liste :

```
1 t = [10, 3, 12, 5]
2 s=0
3
4 for x in t:
5     s = s + x
6
7 print(s)
```

```
8 30
```

— variable accumulatrice initialisée qui va accumuler les termes de la liste `t` et fournir la somme cherchée.

— La liste est parcourue et à chaque itération, le terme courant de la liste est ajouté à `s` et qui est mis-à-jour.

Boucle for : décompression de la variable de contrôle

Le problème suivant illustre un usage de la syntaxe de décompression d'une variable de contrôle d'une boucle `for`.

On dispose d'une suite de couples d'entiers, par exemple :

```
1 [81, 65], [32, 31], [31, 9], [82, 32], [12, 81]
```

Pour chaque couple, on veut calculer la somme des deux entiers. On peut procéder ainsi :

```
1 t = [[81, 65], [32, 31], [31, 9], [82, 32], [12, 81]]
2
3 for a,b in t:
4     print(a+b)
```

```
5 146
6 63
7 40
8 114
9 93
```

— Ligne 3 : au lieu de parcourir la liste `t` avec une variable unique `x` et qui serait une liste à deux éléments (comme les éléments de `t`), on extrait les éléments de cette directement à l'aide de

variables `a` et `b` les deux élements de l'élément courant de la liste.

Le fait de placer directement dans des variables les éléments d'une liste s'appelle la *décompression* ou le *décompactage* (*unpacking* en anglais) de la liste.

Exercice type – Somme des positifs

On donne une liste `L` d'entiers et on demande de calculer la somme `s` des éléments positifs de cette liste. Si aucun élément de la liste n'est positif, on conviendra que la somme `s` vaut 0. Exemples

```
L = [-2, 6, 0, -1, 4, 5, -3] -> s = 15
L = [-2, -3] -> s = 0
L = [42] -> s = 42
```

Solution

Le principe de calcul est le suivant, il s'agit d'une légère variante du calcul d'une somme : on parcourt la liste et on ajoute à une variable accumulatrice `s` initialisée à 0 chaque terme de la liste qui est strictement positif :

```
1 L=[-2, 6, 0, -1, 4, 5, -3]
2 s=0
3
4 for v in L:
5     if v > 0:
6         s = s + v
7
8 print(L, "->",s)
```

```
9 [-2, 6, 0, -1, 4, 5, -3] -> 15
```

Exercice type – Somme sauf des extrémités

On donne une liste `L` d'entiers. Calculer la somme `s` des éléments de `L` en excluant du calcul de la somme le premier et le dernier élément de `L`. Par exemple, si `L = [310, 12, 8100, 90, 31]` alors `s = 8202`.

Solution

Il va falloir parcourir la liste mais comme le résultat demandé dépend de la position particulière de l'élément dans la liste, on utilise un parcours de la liste **avec indice**.

La solution qui peut venir spontanément à l'esprit est de parcourir la liste avec l'idiome habituel et de filtrer avec une instruction `if` pour savoir si le terme courant est au début ainsi qu'à la fin et ainsi exclure ces termes de la somme. Ce qui donnerait le code suivant :

```
1 L =  [310, 12, 8100, 90, 31]
2 n=len(L)
3
4 s=0
```

```
for i in range(n):
    if i != 0 and i != n -1:
     s=s+L[i]
print(s)
```

```
8202
```

C'est bien le résultat attendu mais le code est particulièrement maladroit. En effet, l'instruction `if` ne sert que deux fois sur toute la liste et à des endroits prévisibles. Or, le début de la liste est à l'indice 0 et la fin à l'indice `n-1` où `n = len(L)`. Par suite, il suffit juste d'itérer entre les indices 1 et `n - 2`. D'où le code :

```
L =  [310, 12, 8100, 90, 31]
s=0
for i in range(1,len(L)-1):
     s=s+L[i]
print(s)
```

```
8202
```

Des mesures de vitesse d'exécution montreraient que le premier code à une pénalité de 10% par rapport au second.

Il serait possible de ne pas utiliser d'indice en ayant des connaissances supplémentaires ou en simulant un indice. Ou encore, il était possible de faire la somme de tous les éléments et de retirer le premier et le dernier du résultat :

```
L =  [310, 12, 8100, 90, 31]

s=0
for z in L:
    s = s + z

s=s-L[0]-L[len(L)-1]
print(s)
```

```
8202
```

Boucle for, filtrage et comptage

Étant donné une liste L d'entiers, soit à afficher les entiers pairs de L ; dans le jargon, on dit qu'on effectue un *filtrage* des éléments de L, le filtre étant ici le caractère pair de l'élément de la liste. Voici un code répondant au problème :

```
L = [65, 31, 9, 32, 81, 82, 46, 12]

for z in L:
    if z % 2 == 0:
        print(z)
```

```
32
82
46
12
```

— Lignes 3-5 : parcours de la liste

— Ligne 4 : filtrage avec une instruction `if`

Comptage

Pour un programme de comptage, on crée à l'aide d'une variable un *compteur*. La variable porte souvent le nom de `cpt` ou encore (en anglais) `cnt`. Le plus souvent la variable est initialisée à 0. Le comptage est souvent associé à un filtre qui indique ce que l'on compte. Par exemple, soit à compter les entiers pairs d'une liste donnée :

```
1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2
3 cpt=0
4
5 for z in L:
6     if z % 2 == 0:
7         cpt = cpt + 1
8
9 print(cpt)
```

```
10 4
```

— Ligne 3 : initialisation d'un compteur.

— Lignes 5-7 : parcours de la liste.

— Ligne 6 : filtrage.

— Ligne 7 : incrémentation du compteur lorsque l'élément est filtré.

Boucle for et calcul de maximum

La technique de la « variable accumulatrice » peut s'adapter au calcul du plus grand élément des termes d'une liste :

```
1 L = [10, 3, 12, 5]
2 maxi = L[0]
3 n = len(L)
4
5 for i in range(1, n):
6     if L[i] > maxi:
7         maxi = L[i]
8
9 print(maxi)
```

```
10 12
```

— Ligne 1 : on observe que le plus grand élément de L est 12.

— Ligne 2 : variable initialisée au premier terme de la liste et qui, en fin de programme, contiendra le maximum la liste `L`.
— Lignes 5-7 : la liste est parcourue et à la fin de chaque itération, `maxi` est le plus grand de tous les termes examinés.
— Ligne 5 : le `range` commence à 1 car comme `maxi` est initialisé à `L[0]`, il est inutile d'effectuer la comparaison pour `i = 0` puisque `L[0] > maxi` (cf. ligne 6) autrement dit `L[0] > L[0]` vaut `False`.

Même si ce n'est pas formellement interdit, on évitera d'appeler `max` la variable accumulatrice du maximum car cela empêcherait d'utiliser la fonction Python native nommé `max`. De même on évitera le nom `min` pour un minimum.

Il aurait été possible de ne pas utiliser d'indice pour parcourir la liste :

```
1 L = [10, 3, 12, 5]
2 maxi = L[0]
3 n = len(L)
4
5 for z in L:
6     if z > maxi:
7         maxi = z
8
9 print(maxi)
```

Le seul inconvénient est que le premier passage dans l'instruction `if` compare `maxi` avec lui-même.

Parcours de liste : indices voisins de l'indice courant

L'avantage de parcourir une liste via ses indices est que l'on peut accéder à l'indice suivant ou l'indice précédent. Par exemple, soit à afficher toutes les sommes de deux termes successifs de la liste `t = [65, 31, 9, 32, 81, 82, 46, 12]`, à savoir les sommes suivantes :

96, 40, 41, etc.

L'algorithme de base consiste à parcourir `t` jusqu'à son **avant-dernier** terme, et à afficher $x+y$ où x est le terme courant de `t` et y le terme suivant dans `t` :

```
1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2 n= len(t)
3
4 for i in range(n-1):
5     print(t[i] + t[i+1])
```

```
6 96
7 40
8 41
9 113
10 163
11 128
12 58
```

— Ligne 4 : Le choix de `(n-1)` se justifie par le fait que le parcours de `t` s'arrête à l'avant-dernier terme de `t` puisqu'il faut à chaque étape du parcours considérer le terme *suivant*.

— Ligne 5 : `t[i+1]` est le terme suivant du terme courant, ce qui suppose que $i+1 < n$ et donc que $i < n-1$

— Ligne 4 : écrire `range(n)` au lieu de `range(n-1)` entraînerait une erreur (un débordement d'indice) lorsque $i = n-1$ et que l'interpréteur calcule `t[i+1]`.

La technique du drapeau dans une boucle for

Soit à définir un code Python qui partant d'une liste L d'entiers détermine si L ne contient que des entiers positifs. Plus précisément, le code doit définir un booléen nommé `tousPositifs` qui vaut True si L ne contient que des entiers positifs et False sinon.

Le code suivant réalise cette tâche sur deux exemples de liste :

```
1  # 1er exemple
2  L= [31, 82, -42, 32, -81]
3
4  tousPositifs = True
5  for z in L:
6      if z < 0:
7          tousPositifs = False
8  print(L, '->', tousPositifs)
9
10 print("---------------------")
11
12 # 2e exemple
13 L= [31, 82, 421, 32, 81]
14
15 tousPositifs = True
16 for z in L:
17     if z < 0:
18         tousPositifs = False
19 print(L, '->', tousPositifs)
```

```
20 [31, 82, -42, 32, -81] -> False
21 ---------------------
22 [31, 82, 421, 32, 81] -> True
```

— Le même code (lignes 4-8 et lignes 15-19) est testé avec deux listes différentes (lignes 2 et 13). Ci-dessous, les commentaires ne portent que sur le premier groupe de code.

— Dans chaque cas, on affiche la liste L (lignes 8) et le booléen `tousPositifs`.

— Lignes 4 : on définit au début du code un « drapeau », ici le booléen `tousPositifs` ; ce drapeau témoigne d'un état, l'état étant que la liste L contienne uniquement des nombres positifs (`True`) ou non (`False`). Au départ, le drapeau `tousPositifs` est placé sur `True`.

— lignes 6-7 : La liste L est parcourue et si au cours du parcours de L, la présence dans la liste d'un entier strictement négatif est détectée (ligne 6), le drapeau change de couleur, ici il passe à False (ligne 7). Le drapeau ne change **que si** un entier négatif est détecté, autrement dit le `if` n'est pas couplé à un `else`.

— Ligne 5-8 : si le drapeau change par rapport à son état initial, c'est qu'un entier négatif a été détecté dans la liste L et donc il est attendu que le drapeau tousPositifs vaille False, ce qui est bien le cas, cf. ligne 20.

— Lignes 15-18 : si au contraire, la liste L est parcourue sans que le drapeau ait changé, c'est qu'aucun entier négatif n'est jamais rencontré et donc il est attendu que le drapeau tousPositifs vaille True, ce qui est bien le cas, cf. ligne 22.

Ainsi le code ci-dessus réalise bien la tâche attendue.

Initialisation du drapeau

Comment est déterminée la valeur initiale du drapeau ? Réponse : en fonction de ce que représente le booléen une fois toute la liste parcourue. Si le booléen True doit correspondre au fait que tous les entiers sont positifs, c'est que le booléen doit changer en False dans la boucle quand un entier négatif est détecté et donc, le booléen doit être initialisé à True. Il serait possible de réaliser le même objectif en utilisant un autre drapeau initialisé à False :

```
1 L= [31, 82, -42, 32, -81]
2
3 contientNegatif = False
4 for z in L:
5     if z < 0:
6         contientNegatif = True
7 print(L, '->', contientNegatif)
8
9 print("=========================")
10
11 L= [31, 82, 421, 32, 81]
12
13 contientNegatif = False
14 for z in L:
15     if z < 0:
16         contientNegatif = True
17 print(L, '->', contientNegatif)
```

```
18 [31, 82, -42, 32, -81] -> True
19 =========================
20 [31, 82, 421, 32, 81] -> False
```

— Ligne 3 : on définit un drapeau `contientNegatif` qui vaudra True si L contient au moins un entier strictement négatif et False sinon.

— Lignes 5-6 : lorsqu'on parcourt la liste, le drapeau doit passer à True si un entier strictement négatif apparaît, donc le drapeau doit être initialisé à False.

Exercice type – Alternance de parité

On donne une liste L d'entiers et on demande de créer un booléen **alterneParite** valant `True` si les éléments de L se suivent en alternant de parité et `False` sinon. Voici des exemples de comportements attendus :

L	Alternance de parité
`[81, 32, 9, 12]`	`True`
`[32, 9, 32, 65]`	`True`
`[32, 9, 31, 82]`	`False`
`[81]`	`True`

Solution

Il s'agit d'appliquer la technique du drapeau. Il faut parcourir la liste de `L`. Mais comme il faut à chaque étape comparer les parités de deux éléments consécutifs, la boucle de parcours a la forme avec indice suivante :

```
n=len(L)

for i in range(n-1):
    # Code à compléter
```

La parité d'un entier `a` est donnée par le reste `a % 2`. Donc, deux éléments consécutifs de `L` ont des parités différentes si `L[i] % 2 != L[i+1] % 2`. Dans le cas contraire, le drapeau de surveillance de parité doit changer. D'où le code suivant :

```
L = [81, 32, 9, 12]
alterneParite = True

for i in range(0,len(L)-1):
    if L[i] % 2 == L[i+1] % 2:
        alterneParite = False

print(alterneParite)
```

```
True
```

Testons plusieurs listes placées dans une liste `tests` :

```
tests = [[81, 32, 9, 12],[32, 9, 32, 65], [32, 9, 31, 82],[81]]

for L in tests:

    alterneParite = True

    for i in range(0,len(L)-1):
        if L[i] % 2 == L[i+1] % 2:
            alterneParite = False

```

```
11     print(L, "->", alterneParite)
```

```
12 [81, 32, 9, 12] -> True
13 [32, 9, 32, 65] -> True
14 [32, 9, 31, 82] -> False
15 [81] -> True
```

On pourra remarquer que le code

```
1 L = [81, 32, 9, 12]
2 alterneParite = True
3
4 for i in range(0,len(L)-1):
5     if L[i] % 2 == L[i+1] % 2:
6         alterneParite = False
7
8 print(alterneParite)
```

n'est pas parfaitement optimal, déjà parce que la technique du drapeau n'est pas optimale et aussi parce que la parité de chaque élément de la liste ayant un voisin est évaluée **deux** fois (cf. ligne 5).

Liste construite depuis la liste vide

Pour créer une liste d'éléments vérifiant une certaine propriété P, on procède souvent ainsi :

— on crée une liste L, initialement vide,

— on garnit successivement L d'éléments vérifiant P en utilisant la méthode `append`.

Exemple

Étant donné une liste `t` d'entiers, on veut extraire de `t` la liste `L` des entiers x tels que $x \geq 42$.

Solution :

```
1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2 L= []
3 for z in t:
4     if z >= 42:
5         L.append(z)
6 print(L)
```

```
7 [65, 81, 82, 46]
```

— Ligne 2 : on crée une liste vide à laquelle on va adjoindre les éléments de L qui conviennent.

— Lignes 3-5 : on parcourt la liste `t`.

— Ligne 4 : on teste si l'élément courant vérifie la condition P.

— Lignes 4-5 : si un élément la vérifie, l'élément est placé à la fin de la liste L.

Construction de petites listes

Pour construire de petites listes, ayant jusqu'à 3 ou 4 éléments, il est souvent plus simple de l'écrire directement avec ses éléments entre crochets plutôt que de la construire depuis la liste vide et après applications répétées de la méthode append.

Par exemple, on dispose de deux variables entières u et v et on veut regrouper dans une liste L ces éléments en sorte que l'élément ayant la plus petite valeur soit au début de la liste, ou ne mettre qu'un seul élément dans la liste si les deux éléments sont identiques. Par exemple,

— si u = 42 et v = 42 alors L = [42] ;

— si u = 81 et v = 42 alors L = [42, 81].

Le problème peut se résoudre ainsi :

```
u = 81
v = 42

if u == v:
    L = [u]
else:
    if u < v:
        L = [u, v]
    else:
        L = [v, u]
```

Il est maladroit d'utiliser append comme dans le code ci-dessous :

```
u = 81
v = 42

L = []

if u == v:
    L.append(u)
else:
    if u < v:
        L.append(u)
        L.append(v)
    else:
        L.append(v)
        L.append(u)

print(L)
```

Exercice type – Liste des N premiers multiples de d

Soient N et d deux entiers positifs. Créer la liste L contenant les N premiers multiples de d en commençant par d. Par exemple, si N = 7 et d = 10 alors L = [10, 20, 30, 40, 50, 60, 70].

Solution

```
N=7
d=10
L=[]
for i in range(1,N+1):
    L.append(i*d)
print(L)
```

EXERCICES

Parcours *indice -> élément*

Etant donné une liste, afficher ligne par ligne chaque indice de L, suivi d'une flèche `->` suivie de la valeur de `L` correspondant à l'indice. Par exemple, si `L = [31, 12, 81, 9, 31]` alors l'affichage est

```
0 -> 31
1 -> 12
2 -> 81
3 -> 9
4 -> 31
```

Valeur et indice de même parité

On donne une liste `L` d'entiers et on demande de construire un booléen `memeParite` qui, à partir d'un indice valide de `L` renvoie True si l'indice et la valeur de l'élément de `L` ayant cet indice sont de même parité et False sinon. Par exemple, si `L`

Tester avec une boucle `for`

Dans chaque question, les tests sont à effectuer au sein d'une boucle `for`.

1. Le code suivant montre comment, à partir d'un entier `n`, générer la liste formée de `n-1`, `n` et `n+1` :

```
n=42
L=[n-1, n, n+1]
```

Tester le code précédent pour les entiers `n` valant 42, 100, 0 et 2020. On devra obtenir l'affichage suivant :

```
42 -> [41, 42, 43]
100 -> [99, 100, 101]
0 -> [-1, 0, 1]
2020 -> [2019, 2020, 2021]
```

2. Le code suivant montre (et on l'admettra) comment coder un booléen qui traduit que le minimum d'une liste `L` d'entiers est unique :

```
L=[81, 42, 87, 63, 50, 42, 42]
miniUnique = L.count(min(L))==1
print(miniUnique)
```

```
False
```

Appliquer le code précédent aux 5 listes suivantes :

```
[42], [42, 42, 42], [81, 42, 87, 63, 50, 42, 42], [80, 42, 53, 64]
```

Le code devra afficher

```
[42] -> True
[42, 42, 42] -> False
[81, 42, 87, 63, 50, 42, 42] -> False
[80, 42, 53, 64] -> True
```

Tests aléatoires

Le code suivant montre comment coder un booléen qui traduit que le minimum d'une liste `L` d'entiers est unique :

```
L=[81, 42, 87, 63, 50, 42, 42]
miniUnique = L.count(min(L))==1
print(miniUnique)
```

```
False
```

Générer aléatoirement 10 listes d'entiers entre 0 et 9, de longueur variable et s'assurer que le code précédent fonctionne en testant les 10 listes avec une boucle `for`. La sortie pourra être de la forme suivante :

```
[7, 9, 3, 0, 6] -> True
[2, 4, 4, 7, 3, 6, 2] -> False
[1, 7, 0] -> True
[1, 1, 3, 2, 5, 1] -> False
[6, 9, 2] -> True
[1, 9, 8, 7, 9, 2] -> True
[4, 7, 0, 7, 8, 0] -> False
[4, 9, 6] -> True
[8, 8, 1, 7] -> True
[4, 9, 5, 0, 6, 4, 8, 2] -> True
```

Non multiples

Combien y-a-t-il d'entiers entre 1 et 20000 qui ne sont ni pairs ni multiples de 5 ? Par exemple, 421 ou 2023 font partie des entiers concernés mais pas 42 ni 2025. Le nombre à trouver est 8000.

Liste cardinale

On dit qu'une liste d'entiers est *cardinale* si elle contient :

— un entier pair

— un entier multiple de 5

— un entier entre 0 et 9

— un entier strictement négatif

Par exemple, la liste `[7, -1, 15, 12]` est cardinale. On donne une liste d'entiers et vous devez dire si elle est cardinale ou pas.

Compter les multiples de 10

Calculer le nombre de multiples m de 10 tels que `a` $\leq$ `m` $<$ `b` où `a` et `b` sont des entiers donnés. Par exemple,

```
a = 24 et b =  42 → 2
a = 40 et b =  42 → 1
a = 42 et b =  55 → 1
a = 40 et b =  40 → 0
a = -5 et b =  15 → 2
```

Décompte pair/impair

On donne une liste `L` d'entiers et on demande de calculer

— le nombre `n_pair` d'indices `i` tel que l'élément `L[i]` est pair

— le nombre `n_impair` d'indices `i` tel que l'élément `L[i]` est impair

Par exemple, si `L = [31, 12, 9, 65, 81, 42]` alors `n_pair = 2` et `n_impair = 4`.

De deux en deux dans la deuxième moitié

On donne une liste `L` d'entiers, de longueur `n`. Si `n` est pair, on comprend ce qu'est la « deuxième » moitié de la liste. Si `n` est impair, on comprendra que la « deuxième » moitié de la liste est la plus grande de deux moitiés. Par exemple, si `L = [42, 81, 12, 31, 9]`, la deuxième moitié de `L` est la liste `[12, 31, 9]`.

On demande d'afficher un élément sur deux de la deuxième moitié de `L`, en commençant par son premier élément. Ci-dessous, deux exemples :

```
[42, 81, 12, 31, 9] -> 12 9
[42, 81, 12, 31, 9, 65] -> 31 65
[42, 81, 12, 31, 9, 65, 32, 82, 46] -> 9 32 46
```

Cet exercice est inspiré de Half of the half.

Impairs puis pairs

On donne un entier $n \geq 1$ on demande d'écrire une liste `L` formée d'abord de tous les entiers impairs entre 1 et `n` puis de tous les entiers pairs. Par exemple, si $n = 6$, on obtiendra la liste $L = [1, 3, 5, 2, 4, 6]$ et si $n = 5$, on obtiendra la liste $L = [1, 3, 5, 2, 4]$. Il est envisageable de parcourir la liste `L` plus qu'une fois.

Echanger côte à côte

Soit une liste `L`. Modifier `L` pour que

— le 1er élément et le 2e élément soient échangés de position

— le 3e élément et le 4e élément soient échangés de position

— ...

— et ainsi de suite jusqu'à ce que plus aucun échange ne soit possible.

Par exemple,

— si `L = [31, 12, 9, 65, 81, 42]` alors, après échanges, `L = [12, 31, 65, 9,42, 81]`

— si `L = [31, 12, 9, 65, 81]` alors, après échanges, `L = [12, 31, 65, 9, 81]`

— si `L = [31]` alors, après échange, `L = [31]`

Modifier une liste en multipliant par 10

On donne une liste d'entiers. Modifier cette liste pour que chaque élément x de la liste soit remplacé par $10x$. Par exemple, si `L = [31, 12, 81, 9, 65]` alors, après modification, on aura `L = [310, 120, 810, 90, 650]`.

Inverser une liste

Soit une liste `L`. Modifier `L` pour que

— le 1er élément et le dernier élément soient échangés de position
— le 2e élément et l'avant-dernier élément soient échangés de position
— ...
— et ainsi de suite jusqu'à ce que tous les échanges possibles soient effectués.

Par exemple,

— si `L = [31, 12, 9, 65]` alors, après échanges, `L = [65, 9, 12, 31]`
— si `L = [31, 12, 81, 9, 65]` alors, après échanges, `L = [65, 9, 81, 12, 31]`
— si `L = [31]` alors, après échange, `L = [31]`

On ne demande pas de créer une *nouvelle* liste, c'est simplement le contenu de `L` qui doit changer.

Décalages de liste

Les questions sont indépendantes et de difficulté croissante.
Dans ce qui suit on se donne une liste `L` d'entiers de longueur `n`.

1. **Décalage à gauche**. Construire une *nouvelle* liste `M` qui soit obtenue à partir de `L` en décalant chaque élément de `L` à sa gauche, sauf le premier qui est placé à l'extrémité droite de `M`. La liste `L`, elle, doit rester préservée suite à cette opération. Par exemple, si `L = [9, 0, 8, 3, 0]` alors alors `M = [0, 8, 3, 0, 9]`.
2. **Décalage à droite**. Construire une *nouvelle* liste `M` qui soit obtenue à partir de `L` en décalant chaque élément à sa droite, sauf le dernier, qui est placé tout à la gauche de `M`. La liste `L`, elle, doit rester préservée suite à cette opération. Par exemple, si `L = [9, 0, 8, 3, 0]` alors alors `M = [0, 9, 0, 8, 3]`.
3. **Décalage à gauche en place.** *Modifier* la liste `L` en décalant chaque élément à sa gauche, sauf le premier élément de `L` qui est placé tout à la droite de `L` (il n'y a pas création de nouvelle liste). Par exemple, si `L = [9, 0, 8, 3, 0]` alors alors `L` devient `L = [0, 8, 3, 0, 9]`.
4. **Décalage à droite en place.** *Modifier* la liste `L` en décalant chaque élément à sa droite, sauf le dernier élément de `L` qui sera placé tout au début (il n'y a pas création de nouvelle liste). Par exemple, si `L = [9, 0, 8, 3, 0]` alors alors `L` devient `L = [0, 9, 0, 8, 3]`.

Rotation de liste

Voici, sur un exemple, ce que l'on veut coder dans le cas général. On se donne une liste, par exemple

```
L = [17, 39, 61, 48, 42, 27, 75, 81, 64, 10]
```

et un indice `k` de cette liste, ici ce sera `k=4` qui pointe vers l'élément 42. On souhaite effectuer une rotation des éléments dans la liste en sorte que l'élément à l'indice `k` soit désormais en début de liste. Donc ici, le contenu de `L` devra avoir changé et valoir :

```
L = [42, 27, 75, 81, 64, 10, 17, 39, 61, 48]
```

Plus généralement, les éléments à partir de l'indice `k` se retrouvent en début de liste et le bloc des éléments d'indices initialement inférieurs à `k` se retrouve, dans le même ordre, en un bloc en fin de liste. Bien noter qu'on ne demande pas de créer une *nouvelle* liste mais de *modifier* la liste initiale.

Cette manipulation de liste est l'équivalent de la fonction template standard `rotate` du C++.

Liste : négatif -> opposé

On donne une liste `L` d'entiers et on demande d'écrire un code qui modifie `L` en sorte que chaque élément x négatif de `L` est changé en son opposé $-x$. Par exemple, si `L` est la liste `[42, -9, 81, -12, -65, 0, 46]` alors `L` doit être modifiée en la liste `[42, 9, 81, 12, 65, 0, 46]`

Inverser indices et valeurs dans une liste

On donne un entier `n` $\geq$ `0` et on donne une liste `L` de `n` entiers qui contient une fois et une seule chaque entier entre `0` et `n-1`. Par exemple, pour `n` $=$ `5`, on pourrait avoir :

```
L = [1, 4, 0, 3, 2]
```

On demande de construire une *nouvelle* liste `M` de `n` entiers et dont l'élément à chaque indice `i` est l'indice de l'entier `i` dans la liste `L`. Avec l'exemple précédent :

```
M = [2, 0, 4, 3, 1]
```

Explication : par exemple,

— `M[2] = 4` car `L[4] = 2`.

— `M[1] = 0` car `L[0] = 1`.

La construction de `M` ne doit pas modifier la liste `L`. On commencera par créer, avec la méthode `append` une liste `M` de `n` entiers valant tous -1 puis on remplira convenablement la liste `M`.

Cet exercice est directement inspiré de l'exercice Choix des emplacements sur le site france-ioi.

Un, deux ou cinq

On considère la suite d'entiers suivante :

```
1, 2, 5, 10, 20, 50, 100, 200, 500, ...
```

où les nombres se suivent par paquet de trois en rajoutant à chaque paquet un zéro terminal au paquet précédent.

On donne un entier `n` $\geq$ `0` et on demande de construire la liste des `n` premiers éléments de la suite. Par exemple, si `n` $=$ 7, le code doit produire la liste :

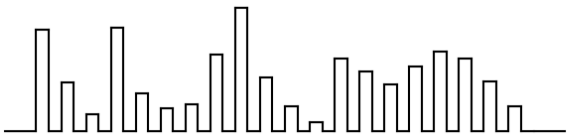
```
[1, 2, 5, 10, 20, 50, 100]
```

Créneaux

On donne une liste `L` de n entiers aléatoires entre 1 et 50 (vous n'avez pas à la créer). On demande de dessiner une suite de « créneaux » ayant pour hauteur les entiers de la liste `L` et une largeur fixe de votre choix. Les créneaux seront précédés et suivis d'une ligne horizontale. Exemple de sortie pour la liste

```
L = [162, 77, 26, 165, 60, 37, 43, 122, 197, 86,
     40, 14, 116, 96, 74, 103, 128, 116, 79, 40]
```

de longueur $n = 20$.



Disques tangents de couleurs alternées

On donne une liste `D` de n diamètres de disque et on demande de tracer ces disques, de centres alignés, chaque disque étant tangent au suivant et les couleurs de remplissage orange et bleu étant alternées. Par exemple, pour la liste

```
D=[47, 21, 23, 50, 37, 22, 50, 37, 55, 27,
23, 30, 41, 20, 56, 34, 5, 48, 51, 47, 27, 5, 53]
```

de $n = 23$ diamètres, on obtient la figure suivante :

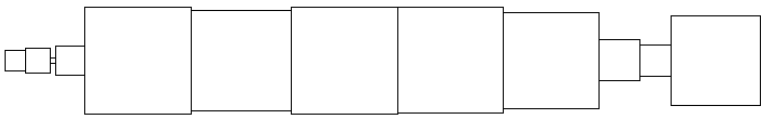
Carrés bords à bords

On donne une liste `L` d'entiers positifs aléatoires et on demande de dessiner des carrés successifs tels que :

— les bords verticaux se touchent,

— `L` contient les valeurs des côtés des carrés à dessiner
— la figure dans son ensemble est symétrique par rapport à un axe horizontal.

Par exemple, si `L=[19, 23, 5, 27, 99, 93, 99, 98, 89, 38, 29, 83]` alors on obtient la figure suivante :



Si vous n'arrivez pas produire le motif avec ces trois contraintes, vous essayerez de produire un motif qui prend en compte seulement deux voire une seule contrainte.

Somme des multiples de 10

On donne une liste `L` d'entiers. Calculer la somme `s` des éléments de `L` qui sont des multiples de 10. Par exemple, si `L = [310, 12, 8100, 90, 31]` alors `s = 8500`.

Somme suivant les indices impairs

On donne une liste `L` d'entiers. Calculer la somme `s` des éléments de `L` dont l'indice est impair. Par exemple, si `L = [31, 12, 81, 9, 31]` alors `s = 21`.

Somme de n entiers consécutifs à partir de d

On vous donne deux entiers positifs `d` et `n`. On vous demande de calculer la somme des `n` entiers consécutifs à partir de `d`. Par exemple, si $d = 10$ et $n = 4$, vous devez calculer $10+11+12+13 = 46$. Voici d'autres exemples :

```
(d, n) = (10, 1) -> 10
(d, n) = (10, 2) -> 21
(d, n) = (10, 100) -> 5950
```

Calcul depuis la base b

On donne la liste `L` des chiffres d'un nombre entier N écrit en base b (en commençant pas le chiffre des unités) et on demande d'afficher en base 10 la valeur de ce nombre. Par exemple, `L = [4, 2, 7, 0, 5, 3]` représente l'entier écrit en base 8 sous la forme `350724` et dont la valeur en base 10 est 119252.

Variations des éléments d'une liste de nombres

On donne une liste `L` d'entiers et on demande d'afficher les variations de chaque terme par rapport au précédent : si le terme augmente, le code affiche `augmente` sinon le code affiche `diminue`. Par exemple, si `L=[42, 81, 12, 9, 6, 34, 30, 35, 48, 82, 32]`, le code affichera :

```
augmente
diminue
diminue
diminue
augmente
diminue
augmente
augmente
augmente
diminue
```

Calcul du minimum

On donne une liste L d'entiers. Calculer le plus petit élément `mini` de la liste. Par exemple, si `L = [31, 9, 81, 9, 31]` alors `mini = 9`.

Indice du minimum

On donne une liste d'entiers. Déterminer un indice `i_mini` correspondant au plus petit élément `mini` de la liste. Par exemple

— si `L = [31, 12, 81, 9, 31]` alors `mini = 9` et `i_mini = 3`

— si `L = [31, 9, 81, 9, 31]` alors `mini = 9` et `i_mini = 1` (ou encore `i_mini = 3`)

En Numpy, cet indice est noté `argmin`.

Plus petit entier non nul (hypothèse simplificatrice)

On donne une liste d'entiers positifs ou nuls et telle que *le premier de la liste soit non nul.* On demande de déterminer le plus petit entier de la liste qui soit non nul. Exemples de fonctionnement du code :

```
[5, 8, 0, 9, 1, 0, 6, 4] -> 1
[5, 8, 1, 6, 4] -> 1
[5, 8, 1, -6, 4] -> -6
[-5, -8] -> -8
[-3, 0, -5, -8] -> -8
[42] -> 42
[-42] -> -42
```

Minimum des entiers d'indices pairs

On donne une liste `L` d'entiers. Créer et calculer une variable `mini_pairs` qui soit le plus petit des éléments d'indices pairs de la liste. Par exemple, si `L = [81, 32, 12, 9, 10, 65, 46]` alors `mini_pairs = 10`.

Élément de carré minimal

On donne une liste `L` d'entiers et on demande de déterminer un élément `m` de cette liste tel que m^2 soit minimal parmi les carrés de tous les éléments de `L`. Par exemple, si `L` est la liste suivante

```
L = [-4, 3, -4, 2, 5, -8, 6]
```

l'élément $m = 2$ convient.

Ecart maximal de températures

On donne une liste `L` de n températures relevées au cours de $n \geq 2$ jours consécutifs. Afficher deux températures représentant la température du jour et la température du lendemain en sorte que l'écart entre ces températures soit maximal. On pourra utiliser la fonction standard `abs` pour mesurer l'écart entre deux températures c'est-à-dire que `abs(u-v)` représente l'écart de température entre la température u et la température v. Par exemple, si `L = [25, 25, 6, 14, -1, 10, 6, 10]` alors le programme affichera :

```
25 6 -> écart maxi = 19
```

Alternativement début et fin

On donne une liste d'entiers `L` et on demande de réarranger les éléments de `L` dans une liste `M` en sorte qu'ils soient rangés dans l'ordre suivant :

- le premier de `L`
- le dernier de `L`
- le second de `L`
- l'avant dernier de `L`
- etc

Par exemple, la liste `[5,7,9,0,2,1,3]` est réarrangée en la liste `[5, 3, 7, 1, 9, 2, 0]`.

Affichage en tas binaire (une seule boucle `for`)

Observez l'exemple suivant que l'on demande de coder dans toute sa généralité : on donne une liste `L` d'entiers telle que

```
[99, 28, 51, 76, 85, 52, 97, 97, 20, 78, 74, 34, 72, 75]
```

et on demande de l'afficher comme suit :

```
99
28 51
76 85 52 97
97 20 78 74 34 72 75
```

La première ligne contient le premier élément, la 2e les deux éléments suivants, la 3e contient les 4 éléments suivants et ainsi de suite, chaque ligne (sauf éventuellement la dernière) contenant deux fois plus d'éléments que la précédente.

Il est attendu de n'utiliser qu'une seule boucle `for` parcourant les éléments de la liste `L`.

Indications :

- Maintenir un compteur pour chaque ligne mémorisant le nombre d'éléments de la ligne courante qui ont été affichés.
- Maintenir une variable indiquant la longueur finale de la ligne que l'on est en train d'afficher
- Lorsqu'on arrive en fin de ligne, afficher un saut de ligne et réinitialiser le compteur

Somme des valeurs ayant même parité que l'indice

On donne une liste d'entiers, par exemple

```
L = [81, 32, 42, 82, 41, 31, 16]
```

On demande de calculer la somme des éléments de L qui ont même parité que l'indice où se trouve l'élément. Par exemple, ci-dessus, 81 n'intervient pas dans la somme puisque 81 qui est impair est à un indice pair, à savoir 0. De même pour 32. Mais 42 est à ajouter puisque 42 est à l'indice 2 qui est un indice pair et 42 est lui-même pair. De même pour 31 et 16. Ainsi, dans l'exemple, la somme attendue est $s = 42 + 31 + 16 = 89$.

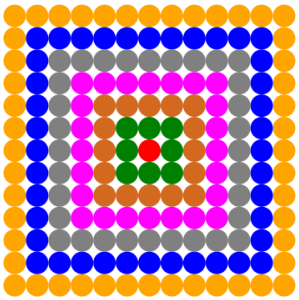
Le double du précédent

On donne une liste L d'entiers. Définir un booléen ok qui vaut True si chaque élément de la liste est le double du précédent et qui vaut False sinon. Exemple de comportements :

```
[10, 20, 40, 80] -> True
[12, 24, 48] -> True
[42] -> True
[0, 0, 0, 0, 0] -> True
[10, 40, 80] -> False
```

Nombre de carrés dans une figure (boucle)

On considère une figure carrée composée de disques alignés, comme ci-dessous :



Le côté du carré est formé de n disques. La figure est formée d'une succession de p carrés concentriques, tracés ici avec des couleurs différentes pour qu'ils soient bien identifiables. Dans le modèle ci-dessus, on a $n = 13$ et $p = 7$.

Notons p_n le nombre de carrés concentriques en fonction du nombre n de disques formant le côté du carré. Le dessin ci-dessus montre ainsi que $p_{13} = 7$. Vérifier que $p_{n+2} = p_n + 1$.

A partir de ce résultat, le côté n étant donné, écrire une boucle `for` permettant de calculer la valeur p_n. Par exemple, si $n = 49$ alors on trouvera $p_n = 25$.

Entiers consécutifs en décroissant

On donne deux entiers a et b avec $a \geq b$, par exemple $a = 2020$ et $b = 2000$. Afficher les entiers consécutifs depuis a jusqu'à b en ordre décroissant.

On observera qu'il y a $a - b + 1$ nombres à afficher et on pourra, par exemple, introduire dans une boucle for une variable auxiliaire j initialisée à a. Mais d'autres codes sont envisageables.

Dix de plus ou de moins

On donne une liste L d'entiers et on demande de construire une variable booléenne OK valant True si chaque nombre de la liste vaut 10 de plus ou 10 de moins que l'élément suivant dans la liste. Sinon la variable OK vaudra `False`. Voici quelques exemples d'exécution attendue :

```
[42, 32, 22, 32, 42, 52, 62] -> True
[42, 52] -> True
[42] -> True
[42, 52, 32, 22, 32, 42, 52, 72] -> False
```

Liste constante

On donne une liste L et on demande de créer un booléen **tousEgaux** valant **True** si tous les éléments de la liste L sont égaux et **False** sinon. Par exemple si L = [42, 42, 42] alors **tousEgaux** vaudra **True** et si L = [42, 421, 42, 42] alors **tousEgaux** vaudra **False**.

Listes « opposées »

Ecrire un code qui partant deux listes d'entiers L et M crée un booléen **sontOpposees** valant `True` si les deux listes sont « opposées » et `False` sinon. Deux listes sont considérées comme « opposées » si elles ont le même nombre d'éléments et si, à des indices identiques, elles possèdent des éléments opposés (comme -81 et 81). Voici quelques exemples de comportements attendus :

```
[81, -12, 0, -81, -31] [-81, 12, 0, 81, 31] -> True
                                  [-81] [81] -> True
                               [0, 0] [0, 0] -> True
                                    [ ] [ ] -> True
                        [81, -12] [-81, -12] -> False
                      [-81, 12, 0] [81, -12] -> False
```

Suite croissante d'entiers consécutifs

Écrire un code qui à partir d'une liste L d'entiers définit une variable booléenne nommée **consecutifs** qui vaut **True** si la liste est constituée d'entiers CONSÉCUTIFS croissants et **False** sinon. Ci-dessous, voici quelques exemples de comportements attendus :

```
[81, 82, 83] -> True
[82, 81, 83, 84] -> False
[2030, 2038, 3000] -> False
[81] -> True
```

Suite en zig-zag

Une liste d'entiers, par exemple

42 81 12 98 39 48

est dite « en zig-zag » si pour chaque élément z de la liste qui n'est pas aux extrémités, son prédécesseur p dans la liste et son successeur s dans la liste vérifient

— ou bien $z \leq p$ et $z \leq s$

— ou bien z ≥ p et z ≥ s

Étant donné une liste L d'entiers, écrire un booléen zigzag qui vaut True si la liste L est en zigzag et False sinon. Ci-dessous, quelques exemples de comportements

```
[42, 81, 12, 98, 39, 48] → True
[81, 12, 98, 39, 48, 11] → True
[42] → True
[42, 81] → True
[42, 81, 82, 12] → False
```

Une liste de un ou deux éléments est considérée comme en zigzag.

Liste creuse

On dit qu'une liste formée de chiffres entre 0 et 9 est *creuse* s'il n'existe jamais deux chiffres voisins non nuls. Par exemple, les listes suivantes sont creuses

```
[4, 0, 2, 0, 5, 0, 0, 0, 6, 0, 4]
[0, 0]
[0]
[5]
[]
```

tandis que la liste

```
[4, 0, 2, 5, 0, 0, 0, 6, 0, 4]
```

n'est pas creuse puisque les chiffres non nuls 2 et 5 sont voisins.

Ecrire un code qui partant d'une liste de chiffres entre 0 et 9 définit une variable booléenne estCreuse indiquant si la liste L est creuse ou pas.

Liste qui patine

On dit qu'une liste d'entiers L *patine* s'il existe une position dans la liste telle que l'élément qui suit cette position et l'élément qui précède cette position ont même valeur. Par exemple, [5, 3, 6, 3, 8] est une liste qui patine puisque, autour de l'élément 6, il y a deux valeurs identiques ; en revanche, la liste [5, 3, 6, 4, 8] n'est pas une liste qui patine.

On demande d'écrire un booléen patine qui dit si oui ou non, une liste L patine.

Voici quelques exemples de comportements :

```
[5, 3, 6, 3, 8] → True
[5, 3, 6, 5, 8] → False
[5, 3, 5] → True
[5, 5] → False
[5] → False
```

Regroupement par signe

On donne une liste L d'entiers et on demande de construire un booléen signe qui vaut True si les deux conditions suivantes sont satisfaites :

— tous les éléments positifs ou nuls sont côte à côte

— tous les éléments négatifs ou nuls sont côte à côte

et qui vaut `False` sinon.

Voici quelques exemples de comportements :

```
[-42, -81, 31, 12, 32] → True
[-42, -81, 31, -12, 32] → False
[31, 12, 32] → True
[-42, -81] → True
[-42, -81, 31, -12, -32] → False
```

On pourra introduire un booléen `change`, initalisé à `False` et qui surveillera le premier moment dans le parcours de `L` où deux termes successifs ont des signes différents.

Égaux et côte-à-côte

Étant donné une liste d'entiers, par exemple

`L=[81, 50, 32, 50, 42, 42, 24, 10, 10, 10, 25]`

on demande de construire un entier `j` valant le premier indice `i` tel que les élements de `L` aux indices `i` et `i + 1` soient égaux et si cet indice n'existe pas, `j` vaudra la longueur de la liste `L`.

Pour la liste `L` ci-dessus, la réponse est `j=4`. Pour la liste `L=[31, 42, 31]`, la réponse est `j=3` et pour la liste `L=[42]`, la réponse est `j=1`.

Pour information, signalons qu'une fonction `adjacent_find` de la bibliothèque standard du C++ résout ce problème.

Cumul de dénivelé

Un randonneur effectue un parcours en montagne. On relève différentes altitudes auxquelles il passe. On demande de calculer le dénivelé total qu'il aura effectué. Par exemple, si la liste des altitudes auxquelles il est passé est

`L=[1000, 1200, 1300, 800, 1000, 900]`

alors le cumul de dénivelé est de

`d= 200+100+500+200+100=1100`.

Home Sweet Home

Lorsque les parents du Petit Poucet vont faire disparaître leurs enfants dans la forêt, le Petit Poucet marque comme indiqué ci-dessous tous les déplacements :

— 1 pour aller à droite
— 2 pour aller à gauche
— 3 pour aller tout droit
— 4 pour monter
— 5 pour descendre.

On vous donne la liste de ces déplacements, par exemple

```
[3, 2, 3, 4, 5, 1]
```

et on demande d'afficher la suite des déplacements que le Petit Poucet doit effectuer pour ramener sa fratrie à la maison. Avec l'exemple ci-dessus, votre code doit afficher :

```
2
4
5
3
1
3
```

Explication de la réponse : pour revenir à la maison, le Petit Poucet doit **inverser** tous les déplacements effectués en commençant par les **derniers** déplacements. Ainsi :

— le premier déplacement du Petit Poucet pour rentrer sera d'aller à gauche (code 2) car le dernier déplacement dans la forêt était d'aller à droite (code 1) ;
— le déplacement suivant du Petit Poucet pour rentrer sera de monter (code 4) car l'avant-dernier déplacement dans la forêt était de descendre (code 5) ;
— et ainsi de suite pour les 4 autres déplacements du retour.

Cet exercice est une adaptation du problème de france-ioi intitulé Visite de la mine.

Concaténer des entiers consécutifs

Si on écrit côte-à-côte tous les entiers entre 1 et `n = 42`, on obtient le très grand entier `N` suivant :

```
12345678910111213141516171819202122232425262728293031323334353637383940414 2
```

On peut vérifier que `N` est un entier composé de $p = 75$ chiffres. Ce qui peut se calculer de la manière suivante :

— pour les entiers entre 1 et 9 : 9 chiffres
— pour les entiers entre 10 et 42 : $(42 - 10 + 1) \times 2 = 66$ chiffres

d'où le total de $9 + 66 = 75$ chiffres.

Plus généralement, étant donné un entier $n \geq 1$, on place côte-à-côte tous les entiers entre 1 et `n` ce qui construit un très grand entier `N`. On demande de calculer le nombre `p` de chiffres de `N`.

On n'utilisera pas de chaînes pour résoudre ce problème. On procédera plutôt de la manière suivante :

— on parcourt dans une boucle tous les entiers `i` entre 1 et `n` ;
— dans une variable `k` on conserve le nombre de chiffres de l'entier courant `i` ;
— lorsque `i` franchit une puissance de 10, on met à jour `k` en l'augmentant de 1 puisque quand on passe une puissance de 10, les nombres ont un chiffre de plus : par exemple avant 10^3 les nombres ont trois chiffres et à partir de 10^3, ils en ont quatre.

Le code devra pouvoir répondre en quelques secondes pour des entiers très grands, de l'ordre de plusieurs dizaines de millions. Pour $n = 2020$, on trouvera $p = 6973$.

Cette famille de nombres est enregistrée dans la base de suites d'entiers OEIS.

Médiane d'une liste de nombres

Etant donné une liste `L` de nombres, on dira qu'un nombre `m` est une médiane de `L` si le nombre d'éléments `x` de la liste `L` qui vérifient $x \leq m$ est égal au nombre d'éléments `y` de la liste `L` qui vérifient $y \geq m$. Par exemple, si `L` est la liste de 8 éléments suivante

```
L = [17, 24, 17, 22, 22, 24, 22]
```

alors

— le nombre `m` = 22 est une médiane de `L` car il existe 5 éléments `x` dans la liste qui vérifient $x \leq 22$ et il y en a autant qui vérifient $x \geq 22$; dans le premier cas, on trouve 17, 17, 22, 22 et 22 et dans le second, on trouve 24, 22, 22, 24 et 22, ce qui fait 5 éléments dans les deux cas ;
— le nombre `m` = 20 n'est pas une médiane de la liste puisqu'il existe 2 éléments `x` dans la liste qui vérifient $x \leq 20$ mais il en existe 5 qui vérifient $x \geq 20$.

On donne une liste de nombres `L` et un nombre `m` est on demande de calculer un booléen `estMediane` qui vaut `True` si `m` est une médiane de `L` et `False` sinon.

Compter les hausses et les baisses

On donne une liste `L` d'entiers, par exemple

```
L = [12, 15, 14, 14, 14, 10, 16, 15]
```

Soit `i` un indice de cette liste, autre que le dernier. On dit que `L` présente :

— une *hausse* à l'indice `i` si $h = L[i+1] - L[i] \geq 0$ et dans ce cas on dit que la valeur de la hausse est `h`
— une *baisse* à l'indice `i` si $b = L[i+1] - L[i] \leq 0$ et dans ce cas on dit que la valeur de la baisse est $-b$.

On demande d'écrire un programme qui détermine la liste `[H, B]` où `H` est la somme des hausses et `B` est la somme des baisses de la liste `L`. Dans le cas de l'exemple ci-dessus, la réponse attendue est `[9, 6]`.

Cet exercice est inspiré de l'exercice *Alpiniste* de la demi-finale du concours Algoréa 2018

Valeurs paires d'une liste

Soit une liste `L` d'entiers. Séparer cette liste en deux listes :

— la liste `P` des entiers pairs de `L`
— la liste `I` des entiers impairs de `L`

Par exemple, si `L = [31, 12, 9, 65, 81, 42]` alors `P = [12, 42]` et `I = [31, 9, 65, 81]`

Sous-liste strictement croissante

On donne une liste non vide d'entiers. On veut en extraire la sous-liste obtenue de la manière suivante : on part du premier élément de la liste, on la parcourt du début à la fin et on ne garde un élément que s'il est strictement supérieur au dernier choisi.

Par exemple, si la liste est

```
[2, 1, 2, 5, 5, 4, 6, 8, 4, 9]
```

la liste à construire est

```
[2, 5, 6, 8, 9]
```

Cet exercice est inspiré de l'exercice Lire ou ne pas lire, telle est la question sur France-ioi.

Trier une liste par signe

On donne une liste d'entiers, par exemple `L=[42, -31, -42, 0, 33, 75, -12, -13, 0]` et on demande de construire une liste `M` dont les éléments soient ceux de `L` mais triés suivant leurs signes : les négatifs d'abord, les éléments nuls puis les positifs. Avec la liste précédente, `M` sera la liste suivante :

```
[-31, -42, -12, -13, 0, 0, 42, 33, 75]
```

Il est envisageable de parcourir la liste L plusieurs fois.

Indices pairs et indices impairs

Soit une liste L. Séparer cette liste en deux listes :

— la liste IP des éléments de L d'indices pairs
— la liste II des éléments de L d'indices impairs

Par exemple, si L = [31, 12, 9, 65, 81, 42] alors IP = [31, 9, 81] et II = [12, 65, 42].

Subdivision régulière

On donne deux nombres a et b avec a $\leq$ b et un entier n $\geq$ 1. On demande de construire la subdivision régulière S d'origine a et d'extrémité b et de pas $\frac{b-a}{n}$ c'est à dire la liste des n + 1 éléments commençant à a, se terminant à b et régulièrement espacés de $\frac{b-a}{n}$. Par exemple, la subdivision d'origine a = 4.2, d'extrémité b = 6 et pour n = 20 est la liste suivante :

```
[4.2, 4.29, 4.38, 4.47, 4.56, 4.65,  4.74,
4.83, 4.92, 5.01, 5.1, 5.19, 5.28, 5.37,
5.46, 5.55, 5.64, 5.73, 5.82, 5.91, 6.0]
```

On demande en fait de réimplémenter un comportement très proche de la fonction Numpy arange.

Pour limiter l'amplitude d'affichage de la partie décimale, on pourra utiliser la fonction round suivant l'exemple ci-dessous :

```
x=4.2+(6-4.2)/20*3
print(x)
print(round(x, 2))
```

```
4.470000000000001
4.47
```

Fonction linspace

On donne trois entiers a, b et n où n $\geq$ 0 et on demande de créer une liste L de n nombres, tels que

— le premier élément soit a,
— le dernier soit b,
— les éléments de L soient **régulièrement espacés**.

Par exemple, si a=42, b=72 et n=4 alors le programme doit créer la liste

```
L = [42.0, 52.0, 62.0, 72.0]
```

(il y a un écart de 10 entre chaque nombre).

On pourra remarquer que si n $\neq$ 1 alors l'écart entre deux éléments consécutifs de L est $\frac{b-a}{n-1}$ (cet écart peut être négatif). Si n=1, on conviendra que L=[a].

Voici quelques exemples du comportement du programme :

```
a=42, b=72, n=4
→ L = [42.0, 52.0, 62.0, 72.0]

a=2020, b=2042, n=6
→ L = [2020.0, 2024.4, 2028.8, 2033.2, 2037.6, 2042.0]

a=2042, b=2020, n=6
→ L = [2042.0, 2037.6, 2033.2, 2028.8, 2024.4, 2020.0]

a=2020, b=2042, n=0
→ L = []

a=2020, b=2042, n=1
→ L = [2020]

a=2042, b=2042, n=4
→ L = [2042.0, 2042.0, 2042.0, 2042.0]
```

Le comportement du programme est inspiré d'une fonction du nom de linspace disponible dans le module Numpy.

Concaténer deux listes

On donne deux listes `L` et `M`. Créer en utilisant la méthode `append` une nouvelle liste `C` formée des éléments de `L` suivis des éléments de `M`. Les listes `L` et `M` doivent être préservées. Par exemple, si `L = [31, 12, 9]` et si `M = [65, 81]` alors `C = [31, 12, 9, 65, 81]`.

Somme de deux listes

Soient les deux listes `L` et `M` suivantes :

```
L = [16, 19, 11, 18]
M = [13, 16, 16, 15, 11, 10, 10]
```

On définit la somme `S` de ces deux listes comme étant la liste suivante :

```
S = [29, 35, 27, 33, 11, 10, 10]
```

Plus généralement, on donne deux listes `L` et `M` d'entiers, pas forcément de même taille, et on demande de construire une nouvelle liste `S` dont les termes aux indices communs soient la somme des termes de `L` et `M` ou, s'il n'y a pas d'indice commun, en gardant le terme. Suite à la construction de `S`, les listes `L` et `M` ne doivent pas êtres modifiées.

Intersection de deux segments

On donne 4 entiers `a`, `b`, `c` et `d` avec `a` $<$ `b` et `c` $<$ `d`. Déterminer la liste `L` des entiers communs à `range(a, b)` et `range(c,d)`. Cette liste sera écrite dans l'ordre croissant et la liste pourra, bien sûr, être vide.

Voici quelques exemples :

```
a = 16 b = 28
c = 11 d = 20
L = [16, 17, 18, 19]
------------------------
```

```
a = 26 b = 27
c = 21 d = 27
L = [26]
------------------------
a = 22 b = 27
c = 10 d = 18
L = []
------------------------
a = 16 b = 22
c = 17 d = 27
L = [17, 18, 19, 20, 21]
------------------------
a = 14 b = 18
c = 13 d = 21
L = [14, 15, 16, 17]
```

Répartition dans l'ordre

On veut répartir N pièces de monnaie parmi p individus i1, i2, ..., ip. Pour cela, on procède de la manière suivante : on donne 1 pièce à i1, puis une pièce à i2, et ainsi de suite jusqu'à ip et on recommence avec i1, puis i2, etc jusqu'à épuisement des pièces.

On demande de produire une liste de p entiers contenant le nombre de pièces que va recevoir i1, i2, etc.

Par exemple, si N = 42 et p = 5, la distribution sera la liste suivante :

```
[9, 9, 8, 8, 8]
```

Si N = 40 et p = 5, la distribution sera la liste suivante :

```
[8, 8, 8, 8, 8]
```

Si N = 4 et p = 5, la distribution sera la liste suivante :

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

Minimum des entiers pairs (boucle for)

On donne une liste L d'entiers contenant au moins un entier pair. Ecrire un code qui calcule le plus petit des éléments pairs de la liste. Exemples de comportements :

```
[81, 32, 12, 9, 12, 65, 46] -> 12
[81, 65, 46] -> 46
```

On parcourra la liste et on pourra utiliser un drapeau qui indique si un entier pair a été rencontré ou pas.

Minimum présent une seule fois

On donne une liste L d'entiers. Soit mini le plus petit élément de la liste. Construire une variable miniUnique qui vaut True ou False selon que mini apparaît une seule fois dans la liste ou au contraire, plusieurs fois. Par exemple

— si L = [31, 12, 81, 9, 31] alors mini = 9 et miniUnique=True ;

— si `L = [31, 9, 81, 9, 31]` alors `mini = 9` et `miniUnique=False`.

Entiers consécutifs mélangés

On vous donne une liste `L` de `n` entiers et vous devez construire un booléen `ok` valant True si les éléments de `L` forment tous les différents entiers entre 1 et `n` mais placés dans le désordre. Par exemple, si `L=[5, 2, 1, 4, 3]` alors `ok` vaudra `True`. En revanche si `L=[12, 1, 3]` ou encore `L=[1, 1, 3]` alors `ok` vaudra `False`. Votre code doit fonctionner pour `n` valant plusieurs millions. On pourra construire une liste auxiliaire formée de `n+1` éléments et telle qu'à l'indice `i` > 0 on lise le nombre d'occurrences de `i` dans la liste `L`.

Sommer des éléments successifs d'une liste

Illustrons sur un exemple le problème à résoudre. On se donne un entier $m = 20$ et une liste d'entiers, par exemple

```
L=[5, 10, 2, 9, 3, 8, 7, 2, 4, 5, 9]
```

On parcourt la liste `L` et on fait la somme des éléments parcourus tant que la somme s vérifie $s \leq m$. Ici, on obtient les éléments 5, 10, 2 et une somme $s = 17$. On place alors la somme s dans une liste `M`. Et on recommence avec la suite des éléments de `L` : on fait la somme des éléments parcourus tant que la somme s vérifie $s \leq m$. Ici, on obtient les éléments 9, 3, 8, de somme 20, et on rajoute le résultat à la liste `M` déjà obtenue, ce qui donne `M = [17, 20]`. Et on poursuit jusqu'à épuisement de la liste `L`. Ici, on obtient `M = [17, 20, 18, 9]`.

Plus généralement, on se donne un entier m, par exemple $m = 20$ et on se donne une liste `L` d'entiers compris entre 1 et m. On demande de créer une nouvelle liste `M` formée des sommes des éléments successifs de `L` et de valeur au plus m.

Voici quelques de comportements pour $m = 20$:

```
[7, 4, 4, 9, 8] -> [15, 17]
[7, 4, 4] -> [15]
[7, 4, 9] -> [20]
[7, 4, 10] -> [11, 10]
[7, 4, 4, 10, 10] -> [15, 20]
[17] -> [17]
[20] -> [20]

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1] -> [20]

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] -> [20, 1]

[10, 20] -> [10, 20]

[7, 4, 4, 9, 8, 9, 4, 6, 9, 9,
4, 8, 4, 7, 8, 5, 4, 4, 6, 6] -> [15, 17, 19, 18, 16, 20, 20]
```

Élément de rang n de la suite des entiers k répétés k fois

On considère la suite infinie des entiers écrits dans l'ordre croissant telle chaque entier $k \geq 1$ est répété k fois :

```
1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 7 etc
```

En examinant la suite ci-dessus, on voit, par exemple, que

— le 10e élément vaut 4

— le 16e élément vaut 6.

À partir d'un entier $n \geq 1$, on demande de déterminer l'entier N qui se trouve en n-ème position de la suite. Ainsi, si $n = 10$ alors `N=4` et si $n = 16$ alors `N=6`.

Formule pour la suite des entiers k répétés k fois

On considère la suite infinie des entiers écrits dans l'ordre croissant telle chaque entier $k \geq 1$ est répété k fois :

```
1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 7 etc
```

Vérifier pour la suite ci-dessus formée des entiers concécutifs jusqu'à 10000 que l'élément de cette suite qui arrive en `n`e position est le plus proche entier de $\sqrt{2n}$. Le proche entier d'un nombre flottant `z` s'obtient par `round(z)`. Ce résultat est annoncé sur math-stackexchange.

Plateau le plus long

On donne une liste d'entiers et on recherche un entier qui est répété consécutivement dans la liste le plus grand nombre de fois (ce qui forme un « plateau »). Par exemple, la liste

```
[4, 8, 8, 8, 8, 7, 7, 7, 7, 7, 9, 4, 4, 4, 4, 4, 5, 5, 5]
```

montre un plateau de quatre 8, de cinq 7, de cinq 4 et trois 5. Donc, une réponse au problème est 7 ou 4. Attention, bien que 4 apparaisse 6 fois, il n'apparait consécutivement que suivant un plateau de longueur 5.

Si tous les nombres sont différents, il n'y a que des plateaux de longueur 1 et le programme peut renvoyer n'importe quel élément de la liste.

On pourra procéder comme suit :

— parcourir la liste avec une boucle `for`

— chaque fois qu'on entre dans un « plateau », on définit un compteur pour ce plateau que l'on incrémente tant qu'on reste sur le plateau

— on surveille si le compteur dépasse la taille maximale déjà atteinte, et si nécessaire, on met à jour et on mémorise le nombre dont la répétition forme le plateau.

Cet exercice provient des demi-finales 2017 du concours Algoréa.

Dupliquer une liste sur une certaine longueur

On donne une liste `t` d'entiers, par exemple

```
t = [12, 81, 31]
```

et un entier `n` $\geq$ `0`, par exemple `n` $=$ `8` et on demande de créer une liste `L`, de longueur `n` et de contenu celui de `t` répété jusqu'à ce que la liste `L` soit complètement remplie. Avec l'exemple ci-dessus, `L` vaudra :

```
L = [12, 81, 31, 12, 81, 31, 12, 81]
```

Cet exercice trouve son origine dans une discussion sur le forum Python d'OpenClassrooms.

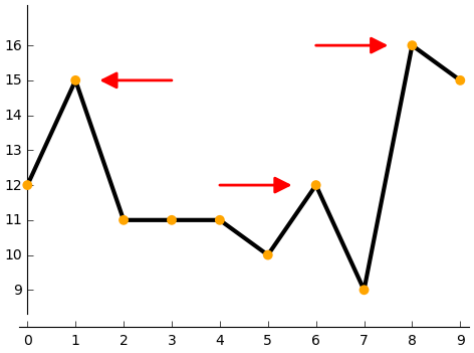
Pics d'une liste

On donne une liste L d'entiers, par exemple

```
L = [12, 15, 11, 11, 11, 10, 12, 9, 16, 15]
```

Soit i un indice de cette liste, autre que le premier ou le dernier. On dit que L présente un *pic* en l'indice i si la valeur de L en l'indice i est strictement supérieure aux valeurs de L en les deux indices voisins. Dans le cas de l'exemple ci-dessus, L présente un pic à l'indice 6.

On demande d'écrire un programme qui calcule le nombre total de pics d'une liste L. Dans le cas de l'exemple ci-dessus,



la réponse attendue est 3.

Cet exercice est inspiré de l'exercice intitulé *Sommets* de la demi-finale du concours Algoréa 2018

Compression de liste

On donne une liste L d'entiers à partir de laquelle on devra générer une nouvelle liste M. Ainsi, si

$$\texttt{L} = [81, 81, 81, 33, 42, 42, 42, 42]$$

alors la liste M sera

$$\texttt{M} = [3, 81, 1, 33, 4, 42]$$

Pour cela, on parcourt la liste L et à chaque étape, on compte les éléments successifs identiques de L. Par exemple, avec la liste L ci-dessus, on compte

— 3 fois de suite la valeur 81

— 1 fois la valeur 33

— 4 fois de suite la valeur 42.

La liste M est alors constituée de la suite formée

— du nombre de valeurs répétées consécutives
— de la valeur répétée.

On demande d'écrire un code Python qui à partir d'une liste `L` construit la liste `M`.

Nombre de McNugget

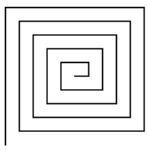
On dit qu'un entier positif est un nombre de McNugget si on peut l'écrire comme une somme de multiples positifs ou nuls de 6, 9 ou 20 (cela vient du fait que les boîtes de Chicken McNuggets sont composées de 6, 9 ou 20 pièces). Par exemple, 2077 est un nombre de McNugget car $2077 = 2 \times 6 + 5 \times 9 + 101 \times 20$. D'un autre côté, il est clair que 25 n'est pas un nombre de NcNugget car $25 - 20 = 5$ est trop petit et 25 n'est pas une somme de multiples de 6 ou 9.

Vérifier qu'entre 0 et 1 million, le plus grand nombre qui n'est pas un nombre de McNugget est 43. Pour cela, on créera une liste `L` de booléens indexée jusqu'à 1 million et telle que `L[k]` vaille `True` exactement si `k` est déjà identifié comme un nombre de McNugget. Pour cela, la liste sera remplie initialement que de valeurs `False`; on posera `L[0]=True` et on parcourra la liste `L` et à l'indice `k`, on placera la valeur `True` si à un des indices précédents et supposés valides `k-6`, `k-9` ou `k-20`, la valeur dans la liste est `True` (cette technique algorithmique s'appelle de la *programmation dynamique*).

Dessiner une spirale

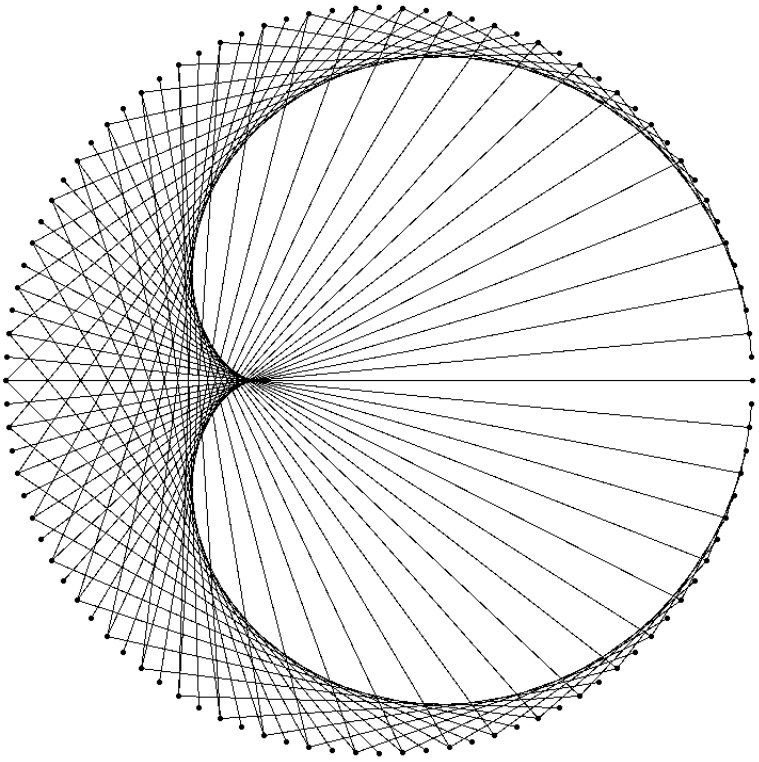
Dessiner une ligne en forme de spirale et effectuant n tours. Le départ se fera à l'origine $(0, 0)$. Pour chaque tour, on se déplacera de la même distance deux fois de suite (suivant un demi-carré) puis les deux fois suivantes la distance sera plus grande d'une distance constante `esp` connue au départ (et la trajectoire sera encore un demi carré). Chaque cycle consiste à se déplacer suivant le schéma droite - haut - gauche - bas. Au départ, on choisira la première longueur valant `esp` défini plus haut.

Le dessin ci-dessous a été construit pour `n=5` et `esp=10` (et le centre de la spirale est le point `(0,0)`) :



Tables de Plouffe

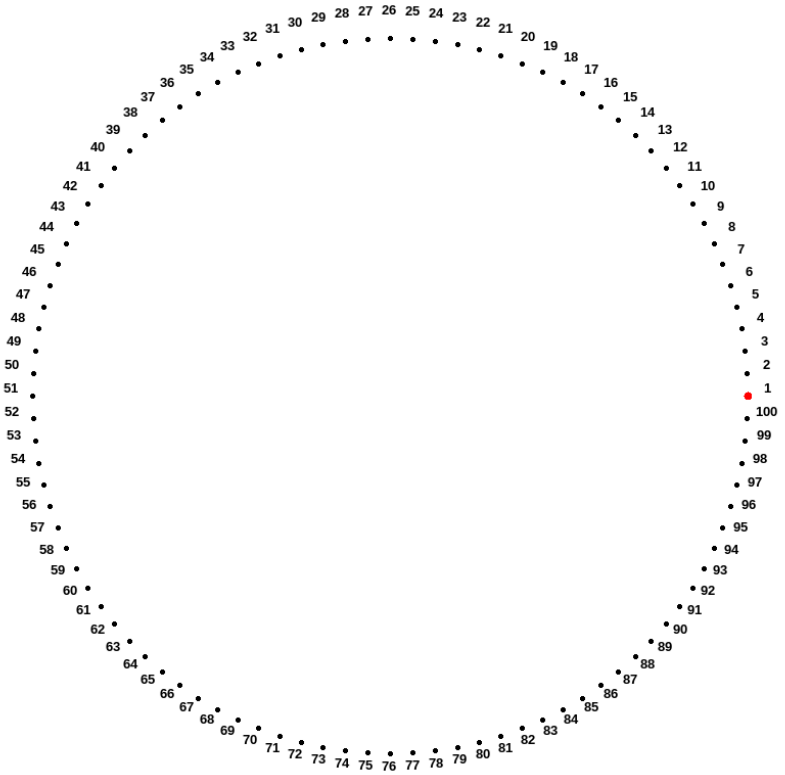
On va représenter une table de multiplication par des cordes sur un cercle ce qui va produire un motif surprenant, par exemple :



Plus précisément, on veut repésenter la table de multiplication de n avec p points placés sur un cercle. Pour fixer les idées, imaginons la table de $n = 2$ avec $p = 100$ points. Cela veut dire qu'on va représenter les 100 produits suivants :

$$1 \times n2, 2 \times 2, 3 \times 2, ..., 100 \times 2$$

On commence par placer p points régulièrement espacés sur un cercle. Le premier point placé représente l'entier 1, le 2e point représente l'entier 2 et ainsi de suite jusqu'au p-ème point :



Pour représenter des entiers plus grands que p, on continue à compter à partir du premier point, par exemple pour représenter 242, on utilise le point 42 puisque $242 = 100 + 42$; de même, 600 est représenté par 100 ; plus généralement, un entier multiple de p est représenté par p et un entier non multiple de p est repésenté par son reste modulo p.

Ensuite, chacun des p produits de la table de multiplication de n :

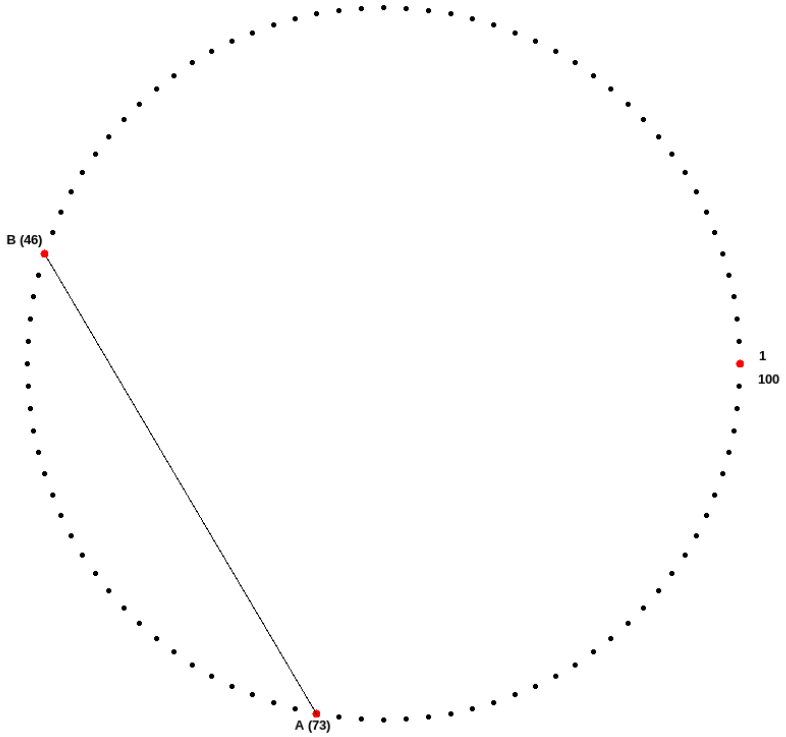
$$1 \times n, 2 \times n, ..., p \times n$$

sera représenté par un segment. Plus précisément, le produit $k \times n$ est représenté par la corde AB où

— le point A représentant k sur le cercle,

— le point B représentant sur le cercle le résultat du produit $k \times n$.

Voici un exemple, dans le cas $n = 2$ et $p = 100$, du produit $73 \times n$:



Comme on a $73 \times 2 = 146$ qui est représenté par 46, on relie le point A représentant 73 et le point B représentant 46.

Lorsque les p « produits » sont dessinés, on obtient un motif, par exemple, la figure en début d'énoncé correspond à $n = 2$ et $p = 100$.

On demande d'écrire un programme de dessin d'une trentaine de lignes qui représente la figure pour n et p donnés.

On utilisera un cercle de centre $(0, 0)$ et de rayon $R = 400$ mais qu'il sera inutile de tracer, cf. la figure en début d'énoncé. On rappelle que le point du cercle d'angle polaire t a pour coordonnées $(R\cos(t), R\sin(t))$. Pour faciliter le codage, les entiers représentés sur le cercle seront numérotés à partir de 0 (au lieu de 1 ci-dessus). On construira une liste `sommets` contenant les coordonnées des différents sommets du cercle en sorte que l'entier $k = 0, \ldots, p-1$ soit représenté par `sommets[k]`.

Une vidéo de Micmaths a popularisé cette représentation des tables de multiplication. Burkard Polster, dans la présentation d'une de ses vidéos, attribue la découverte de cette représentation à Simon Plouffe, cf. son article curves obtained with bn mod p.

4 Boucles imbriquées

Boucles for imbriquées

Soit à écrire un code Python qui affiche un texte comme le suivant :

```
1
1 2
```

```
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12 13
1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Plus précisément, chaque ligne `L` reprend la ligne précédente `K` mais `L` est complétée par l'entier qui suit le dernier entier de la liste affichée dans `K`.

Noter qu'il y a deux types de répétition d'actions :

— répétition d'affichage de ligne

— répétition d'affichage de nombres dans chaque ligne

Un code répondant au problème pourrait être le suivant :

```
1 for lig in range(1,15):
2     for col in range(1,lig + 1):
3         print(col, end = " ")
4     print()
```

On constate qu'une boucle for (lignes 2-3) est imbriquée dans une boucle for (lignes 1-4). On dit que :

— la boucle ligne 1 est la *boucle externe* ;

— la boucle ligne 2 est la *boucle interne* (ou encore la boucle la plus *profonde*).

La boucle `for` (ligne 1) contrôle la progression par ligne. Un numéro `lig` de ligne étant fixée, l'autre boucle for (ligne 2) contrôle la progression de gauche à droite dans la ligne fixée.

Noter à la ligne 3 la présence de l'argument nommé de la fonction print :

```
end = " "
```

qui permet

— d'empêcher le passage à la ligne pendant qu'on affiche les éléments de la i^{e} ligne,

— de séparer deux nombres par un espace.

Noter aussi ligne 4, l'appel à la fonction `print` pour séparer deux lignes consécutives de l'affichage.

D'une façon générale, le corps de toute boucle `for` peut contenir n'importe quel type d'instructions, et en particulier une autre boucle `for`.

Exercice type – Liste d'entiers sans éléments consécutifs

On donne une liste `L` d'entiers et on cherche à construire un booléen `sansConsecutifs` et qui vaut `True` si `L` ne contient pas deux entiers consécutifs. Voici en fonction de `L` différentes valeurs pour le booléen `sansConsecutifs`

```
[9, 2, 0, 4, 4, 0, 7] -> True
[9, 5, 0, 4, 4, 0, 7] -> False
[7, 2, 0, 4, 4, 3, 7] -> False
[7] -> True
```

Explication :

— dans le 1er cas, on observe bien qu'il n'y a aucune valeur k de la liste pour laquelle $k-1$ ou $k+1$ soit présente dans la liste ;

— dans le 2e cas, la liste contient un 5 et un 4 (qui sont consécutifs)

— dans le 3e cas, la liste contient un 2 et un 3 (qui sont consécutifs) ; on pouvait aussi remarquer qu'il y a un 4 et un 3.

On cherchera à examiner toutes les paires possibles d'éléments de la liste.

Solution

On rappelle que *consécutif* signifie qu'il y a 1 d'écart, comme entre 2020 et 2021 ou entre 42 et 41.

L'idée est d'essayer tous les cas : on compare le premier élément de `L` à chaque élément de `L` et on regarde s'il y aurait 1 d'écart entre les éléments et si oui, on met à jour un drapeau `sansConsecutifs` en le faisant passer à `True` (il est placé à `False` au départ). Puis on recommence avec le deuxième élément de `L`.

Dans le cas du 2e exemple ci-dessus, il y aura 1 d'écart avec le 4e élément de L et donc le drapeau va changer. Et ainsi de suite jusqu'à épuisement de la liste.

Si après toutes ces comparaisons, le drapeau est resté inchangé (à `False` donc), c'est qu'il n'y a jamais deux éléments consécutifs dans la liste et si au contraire, il a un moment basculé à `True` , c'est que la liste contient deux éléments consécutifs.

D'où le code suivant :

```
tests=[
    [9, 2, 0, 4, 4, 0, 7],
    [9, 5, 0, 4, 4, 0, 7],
    [7, 2, 0, 4, 4, 3, 7],
    [7]]


for L in tests:
    sansConsecutifs=False
    for a in L:
        for b in L:
            if a-b==1:
                sansConsecutifs=True
    print(L, '->', not sansConsecutifs)
```

Signalons juste que ce code présente plusieurs défauts :

— il compare systématiquement l'écart entre `a` et `b` puis entre `b` et `a`, ce qui est inutile ;
— les boucles continuent à tourner si le drapeau est passé à `True` ce qui évidemment n'est pas optimal.

Exercice type – Grille : remplissage par lignes

Vous allez devoir écrire un code qui affiche un motif dont un exemple est visible ci-dessous :

```
42 43 44 45
46 47 48 49
50 51 52 53
54 55 56 57
58 59 60 61
62 63 64 65
66 67 68 69
70 71 72 73
74 75 76 77
78 79 80 81
```

Ce motif consiste en une grille rectangulaire de dimensions n x p données (n : nombre de lignes, p : nombre de colonnes) de tous les entiers consécutifs à partir d'un entier donné d, le remplissage devant se faire *ligne par ligne*.
Ci-dessus, l'affichage est obtenu pour $n = 10$, $p = 4$ et $d = 42$. Noter que deux colonnes sont séparées par un espace.
On répétera n fois l'écriture d'une ligne et chaque ligne sera formée de la répétition des p éléments de la ligne. Pour savoir quel nombre afficher à une position donnée on maintiendra à jour une variable `N` initialisée à `d`.

Solution

Un motif 2D nécessite typiquement deux boucles imbriquées. Ici, on doit répéter n fois l'écriture d'une ligne et chaque ligne est formée de la répétition des p éléments de la ligne. On a donc un schéma du type :

```
1 for i in range(n):
2     for j in range(p):
3         # A COMPLÉTER
```

D'où, par exemple, le code suivant :

```
1 n=10
2 p=4
3 d=42
4 k=0
5 for i in range(n):
6     for j in range(p):
7         print(d+k, end=' ')
8         k+=1
9     print()
```

Effectuer plusieurs tests à l'aide d'une boucle for

Pour conforter la validité d'un programme, il est d'usage d'effectuer des tests. Pour tester un morceau de code sur un grand nombre d'entrées, on peut utiliser une boucle for qui parcourt une liste de données à tester.

Soit un code qui doit tester si un entier $n \geq 0$ est strictement plus grand que 1000 ou, sinon, multiple de 10 mais pas multiple de 100. Par exemple, 2021 vérifie le test car $2021 > 1000$, 120 aussi (c'est un multiple de 10 mais pas de 100) mais pas 700 car $700 < 1000$ et 700 est multiple de 100.

D'une façon générale , pour vérifier le caractère correct d'un programme, on ne se limite pas à un ou deux tests de ce programme. On accumule les tests en ayant la volonté de mettre en défaut le programme :

```
1 tests = [700, 20, 42, 2020, 2021, 3000, 1000, 0]
2 for i in range(len(tests)):
3     x=tests[i]
4     if x > 1000 or (x % 10 == 0 and x % 100 != 0):
5         print(x, True)
6     else:
7         print(x, False)
```

```
8 700 False
9 20 True
10 42 False
11 2020 True
12 2021 True
13 3000 True
14 1000 False
15 0 False
```

— Ligne 1 : les données à tester sont placées dans une liste

— La totalité du code à tester (lignes 3-7) est placée dans le corps de la boucle for parcourant la liste (ligne 2).

— Lignes 5 et 7 : la sortie (lignes 8-15) doit être lisible pour qu'on puisse vérifier rapidement l'exactitude de la réponse donnée par le programme (c'est pour cela que x est affiché)

— Ligne 1 : on essaye de mettre en défaut le programme en examinant tous les cas typiques possibles et en étudiant les valeurs particulières situées aux bords comme ici 0, 1000.

EXERCICES

Intersection de listes

On donne deux listes L et M chacune formée d'entiers distincts, par exemple `L = [8, 9, 3, 5]` et `M = [1, 2, 3, 4, 6, 8, 9]`. On demande d'écrire une liste `inter` valant la liste des éléments communs aux deux listes. Voici quelques exemples de comportements

```
L = [8, 9, 3, 5] M = [1, 2, 3, 4, 6, 8, 9] -> [8, 9, 3]
L = [8, 9, 2, 4] M = [8, 4, 7] -> [8, 4]
L = [6] M = [1, 2, 5] -> []
```

Répéter l'affichage du contenu d'une liste

Soit une liste L d'entiers (entre 0 et 9, pour des raisons d'esthétique de l'affichage). On demande d'afficher n fois cette liste où $n > 0$ est un entier donné. Par exemple, si L est la liste de contenu :

```
4 8 0 9 6 3 2 8
```

et si $n = 6$, l'affichage obtenu doit être :

```
4 8 0 9 6 3 2 8
4 8 0 9 6 3 2 8
4 8 0 9 6 3 2 8
4 8 0 9 6 3 2 8
4 8 0 9 6 3 2 8
4 8 0 9 6 3 2 8
```

Deux chiffres successifs seront séparées d'un espace.

Remplir une liste avec des 0, des 1 et des 2

Afficher toutes les listes de 4 éléments formées d'éléments parmi 0, 1 ou 2.

Par exemple, les listes suivantes conviennent

```
[0, 2, 0, 1]
[0, 2, 0, 2]
[1, 1, 1, 1]
```

Somme qui vaut 42

On donne deux listes d'entiers L et M. Créer un booléen **somme42** qui vaut True si la somme de deux nombres, l'un dans L et l'autre dans M, vaut 42. Sinon, le booléen vaudra False. Voici quelques exemples de comportements attendus :

```
[17, 22, 5, 5, 33, 8] [34, 8, 20] -> True
[6, 22, 5, 5, 33, 8] [35, 8, 25]  -> False
```

Explication pour le premier cas : remarquer que $8 + 34 = 42$ ce qui explique la réponse.

Entiers opposés dans une liste

On donne une liste L d'entiers non nuls et on demande de dire si cette liste contient deux entiers

opposés (comme -42 et 42). Par exemple, si la liste L est

```
[-2, 1, -3, 5, -7, -4, 5, -5, 3, -8]
```

alors, L contient deux entiers opposés (-3 et 3) aux indices 2 et 8.
et si la liste L est

```
L = [-5, 4, -1, 4, 8, 6, -9, 8, -3, 8]
```

alors cette liste ne contient jamais deux éléments qui sont opposés.
Ecrire un code qui définit une variable opp qui vaut

— `None` si deux entiers quelconques de la liste ne sont jamais opposés ;
— une liste `[i, j]` si aux indices `i` et `j` de L, les éléments sont opposés.

Dans le cas de la première liste donnée en exemple, une réponse possible attendue est

```
opp = [2, 8]
```

et dans le cas de la seconde, la réponse attendue est :

```
opp = None
```

Somme nulle

On donne une liste L d'entiers et on demande de construire un booléen `somme0` valant `True` s'il existe des entiers successifs dans L et dont la somme vaut 0.
Par exemple,

— si `L=[-4, 2, -4, 1, 9, -6, -4]` alors `somme0 = True` puisque `(-4) + 1 + 9 + (-6) = 0`,
— si `L=[-4, 0, 5, 1]` alors `somme0 = True` puisque la liste contient un terme nul,
— si `L=[-3, 2, 4]` alors `somme0 = False`.

On utilisera trois ou même deux boucles `for` imbriquées. L'idée est de parcourir toutes les sous-listes de L dont la somme est à calculer. Chacune de ces sous-listes commence à l'indice `d` et termine à l'indice `f` où `0` $\geq$ `d` $\leq$ `f` $<$ `n`. On génère `d` dans une première boucle puis `f` dans une boucle imbriquée. Ensuite, on calcule la somme correspondante ce qui nécessite une troisième boucle `for` imbriquée. En fait, il est possible de calculer la somme avec la 2^e boucle `for`.

Etude d'une suite définie par une somme

On pose, pour $n \geq 1$ entier, $S_n = \sum_{k=1}^{n} \left(\frac{k}{n}\right)^k$. Par exemple,

$$S_5 = \frac{1}{5} + \frac{4}{25} + \frac{27}{125} + \frac{256}{625} + \frac{3125}{3125} = 1241/625 = 1.9856$$

En calculant, les 20 premiers termes de la suite, examiner si la suite S_n est monotone (autrement dit, elle croît toujours ou elle décroît toujours).
La suite semble-t-elle tendre vers une certaine valeur ?

Afficher les effectifs d'une liste d'entiers

On donne une liste L d'entiers, par exemple

```
L = [81, 31, 81, 12, 81, 9, 12, 65]
```

On demande d'afficher le nombre de fois que les différents nombres de la liste `L` apparaissent dans `L`. L'ordre d'affichage devra respecter l'ordre d'apparition dans la liste `L`. Avec l'exemple ci-dessus, le programme devra afficher

```
81 : 3
31 : 1
12 : 2
9 : 1
65 : 1
```

La ligne

```
12 : 2
```

de l'affichage signifie juste que `12` apparaît 2 fois dans la liste `L`. On remarquera que dans la colonne de gauche de l'affichage , aucun élément de la liste n'est répété.

On pourra appliquer la méthode suivante. On parcourra la liste `L` dans une boucle `for`. Auparavant, on aura créé une liste vide `dejaVus` destinés à enregistrer les éléments déjà rencontrés à l'indice `i` de la boucle de parcours. À chaque indice `i` de la liste `L`, on regardera si l'élément `L[i]` est dans la liste `dejaVus` et si non, on comptera, avec une boucle `for` imbriquée, le nombre d'éléments de `L` d'indices `j` $>$ `i` et valant `L[i]`.

Pièces de monnaie

Dans un pays imaginaire, les pièces de monnaie sont de `a` unités ou de `b` unités. On demande d'écrire un code qui détermine si, oui ou non, il est possible de régler un certain montant donné de `m` unités. Par exemple si `a = 49` et `b = 10` alors on ne peut pas régler `m = 42` unités ni `m = 105` mais on peut régler `m = 128` puisque `128 = 3*10 + 2*49`.

Vous ne devez pas utiliser de boucles imbriquées.

Montant irréalisable

Un objet de type A coûte n euros, un objet de type B coûte $n+1$ euros. On dit qu'un montant m est réalisable s'il est possible de choisir des objets de type A ou B et dont l'achat corresponde à ce montant m.

Par exemple, si $n = 10$ alors le montant 119 est réalisable car il correspond à l'achat de 2 objets de type A et de 9 objets de type B : $119 = 2 \times 10 + 9 \times 11$. Mais certains montants sont irréalisables, par exemple $m = 45$ ou $m = 78$.

Par ailleurs, on peut montrer que tout montant assez grand est forcément réalisable. On demande de conjecturer, en expérimentant autant de valeurs de n qu'il vous paraît nécessaire, le plus grand montant non réalisable en fonction de n. Pour $n = 10$, ce montant est de 89 euros.

On pourra utiliser la fonction `sorted(L)` qui renvoie une liste de même contenu que `L` mais triée.

Multiple de 42 qui soit somme de deux carrés

Un carré [4] est un nombre qui est de la forme $n \times n = n^2$ où n est un entier **strictement positif**, par exemple 36 ou 49 sont des carrés mais pas 42.

Noter que, pour la suite, 0 n'est pas considéré comme un carré.

4. Sous-entendu, « carré parfait ».

En examinant toutes les sommes de carrés possibles, on peut, par exemple, affirmer que

— 42 n'est pas une somme de deux carrés
— 41 est une somme de deux carrés : $41 = 4^2 + 5^2$
— 25 est une somme de deux carrés non nuls : $25 = 3^2 + 4^2$
— 9 n'est pas une somme de deux carrés non nuls.

Il se trouve qu'il existe 5 entiers entre 1 et 10000 qui sont multiples de 42 et qui peuvent s'écrire comme somme de deux carrés. On vous demande de trouver ces 5 nombres.

On recherchera toutes les façons d'obtenir une somme de deux carrés et on observera aussi que $100 \times 100 = 10000$ en sorte que si $n = a^2 + b^2$ alors on peut supposer que $0 \leq a, b \leq 100$.

Nombre de rectangles d'aire plafonnée

Les rectangles différents à côtés entiers et d'aire valant au plus 6 sont de tailles :

```
1x1, 1x2, ..., 1x6, 2x2, 2x3
```

il y en a donc 8.

Plus généralement, on donne un entier strictemement positif `N` et on demande de dire combien de rectangles de dimensions entières `a`, `b` avec `a` $\leq$ `b` ont une aire valant au plus `N`.

Par exemple, pour `N` $=$ `1821`, on en trouvera 7000.

Pour une recherche compétitive, voir sur SPOJ le problème Rectangles. Voir aussi cette suite.

Problème de programmation linéaire

On cherche à maximiser la quantité $z = x + 6y$ lorsque x et y sont des **entiers** satisfaisant les contraintes suivantes :

$$\begin{cases} x, y \geq 0 \\ x \leq 200 \\ y \leq 300 \\ x + y \leq 400 \end{cases}$$

Pour cela on testera tous les cas possibles en utilisant deux boucles `for` imbriquées. On trouvera une valeur maximum de $z = 1900$.

Problème de dénombrement

Ce problème a été posé dans un questionnaire de Mathraining. On donne un entier positif `n`, typiquement `n` $=$ `60`. Déterminer le nombre de triplets (`x`, `y`, `z`) d'entiers strictement positifs vérifiant `x` $+$ `y` $+$ `z` $=$ `n`. Pour `n` $=$ `60`, on trouvera 1711 triplets.

Nombres consécutifs dont la somme vaut 54

Trouver toutes les listes d'entiers positifs consécutifs dont la somme vaille 54. Parmi toutes les solutions, on trouvera la liste d'entiers 17, 18 et 19 puisque $17 + 18 + 19 = 54$.

Cet exercice a pour origine cette question.

Nombres de Harshad

On dit qu'un entier `n>0` est un nombre de Harshad s'il est divisible par la somme de ses chiffres. Par exemple, 42 est un nombre de Harshad car la somme de ses chiffres est 6 et 42 est un multiple de 6. En revanche, 32 n'est pas un nombre de Harshad puisque 32 n'est pas un multiple de 5.

Déterminer la liste des nombres de Harshad ayant 1, 2 ou 3 chiffres (on trouvera une liste de 212 entiers). On utilisera 3 boucles `for` imbriquées.

Somme de sommes

Soit la suite (U) des entiers positifs qui ne s'écrivent, en base 10, qu'avec le chiffre 1 :

$$1, 11, 111, 1111, 11111, \ldots$$

On remarquera qu'un tel nombre est une somme de puissances de 10, par exemple pour *mille cent-onze* :

$$1111 = 10^0 + 10^1 + 10^2 + 10^3 = 1 + 10 + 100 + 1000.$$

(1) Soit $n \geq 1$ un entier donné. Calculer, avec une boucle `for`, le n-ième nombre de la suite (U) (on ne se contentera pas d'afficher le nombre mais on le placera dans une variable). Par exemple, si $n = 4$, le programme doit calculer le nombre 1111.

(2) Calculer la somme des N premiers nombres de la suite (U). Par exemple, si $N = 20$:

$$1 + 11 + 111 + 1111 + 11111 + \ldots + 11111111111111111$$

Par exemple, pour $N = 10$ on trouvera 1234567900.
Pour cela, on utilisera une boucle `for` suivant deux méthodes :

- placer une partie du code de la question précédente dans le corps de la boucle (donc deux boucles imbriquées) ;
- générer simultanément les termes de la somme et les sommes partielles (simple boucle).

Tri-casier

On donne une liste `L` d'entiers positifs, par exemple

```
[47, 5, 5, 49, 23, 31, 13, 46, 48, 20,
9, 19, 19, 40, 42, 31, 38, 9, 45, 26]
```

On demande de construire une nouvelle liste `M` dont les éléments sont ceux de `L` mais triés suivant leur chiffre des unités.

Par exemple, dans le cas de la liste ci-dessus, `M` est la liste suivante :

```
[20, 40, 31, 31, 42, 23, 13, 5, 5, 45,
46, 26, 47, 48, 38, 49, 9, 19, 19, 9]
```

Explication : on lit d'abord les éléments de `L` qui se terminent par 0 (20, 40) puis ceux qui se terminent par 1 (31, 31), et ainsi de suite jusqu'aux éléments de `L` qui se terminent par 9.

Inversions dans une liste

On donne une liste `L` d'entiers, par exemple

```
[5, 5, 4, 11, 9, 1, 5]
```

On appelle *inversion* de `L` tout couple `(a, b)` de valeurs dans `L` telles que :

— a apparaisse à gauche de b
— a > b.

Par exemple, (4, 1) est un inversion de la liste précédente.

Ecrire un code qui affiche toutes les inversions d'une liste L. Dans le cas de la liste ci-dessus, le code doit afficher :

```
5 4
5 1
5 4
5 1
4 1
11 9
11 1
11 5
9 1
9 5
```

mais pas forcément dans cet ordre.

Ecart minimal d'indices (boucles imbriquées)

On donne une liste L d'entiers, par exemple

```
L = [16, 14, 17, 13, 15, 14, 16, 15, 13, 14, 16, 15, 13, 16]
```

On demande de déterminer l'écart minimal d'indices où se trouveraient des valeurs identiques. Cette valeur est 3 dans la liste ci-dessus ; en effet, 15 apparaît aux indices 4 et 7 et tout autre répétition de valeur, comme 13, apparaît dans la liste avec un écart d'indices valant au moins 3. On notera que l'écart minimal est aussi atteint pour la valeur 16 aux indices 10 et 13.

Si la liste ne contient que des valeurs différentes, le nombre minimal recherché sera 0.

L'algorithme à appliquer est le suivant : on parcourra la liste en utilisant deux boucles `for` imbriquées. Ce n'est pas une méthode optimale et elle ne conviendrait pas pour une liste ayant plusieurs millions d'éléments.

Cet exercice est inspiré de l'exercice *Distance minimale* de la demi-finale du concours Algoréa 2017.

Différences successives

On donne une liste L d'entiers, par exemple :

```
L = [9, 5, 6, 3, 6, 2, 6]
```

et on va réduire L à un unique élément par le procédé suivant : à chaque étape, on constitue une nouvelle liste dont les éléments sont la différence de deux éléments successifs de L. Avec l'exemple, la nouvelle liste M est :

```
M = [4, -1, 3, -3, 4, -4]
```

(par exemple, le coefficient -1 en 2e position provient de la différence 5 − 6 entre le 2e et le 3e coefficient de L)

On recommence ainsi avec la nouvelle liste et ainsi de suite jusqu'à ce que la liste ne contienne plus qu'un élément. A chaque étape, la nouvelle liste a un élément de moins que la précédente.

La question posée est de calculer l'unique élément de la **dernière liste**.

Dans le cas de l'exemple ci-dessus, la valeur à trouver est 42.

Damier de nombres

Réaliser un programme qui affiche un damier de forme carrée, de côté de longueur $n > 0$ et rempli alternativement du nombre 4 et du nombre 2. La case en haut à gauche sera toujours 4. Par exemple, pour $n = 7$, le damier aura l'allure suivante :

```
4 2 4 2 4 2 4
2 4 2 4 2 4 2
4 2 4 2 4 2 4
2 4 2 4 2 4 2
4 2 4 2 4 2 4
2 4 2 4 2 4 2
4 2 4 2 4 2 4
```

Il est possible de coder l'exercice de plusieurs façons. On pourra imaginer que la grille est numérotée par ligne et par colonne et on remarquera que tous les 4 de la grille sont à des indices de ligne et de colonne qui ont toujours la même parité (soit les deux indices sont pairs, soit les deux indices sont impairs).

Pyramide croissante de nombres

Observez le motif suivant obtenu pour $n = 6$:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
```

On vous donne un entier $n > 0$ et on vous demande de construire le motif formé de n lignes de la manière suivante :

— la 1re ligne contient le nombre 1 écrit une seule fois,
— la 2e ligne contient le nombre 2 écrit 2 fois,
— la 3e ligne contient le nombre 3 écrit 3 fois,
— et ainsi de suite jusqu'à avoir écrit n lignes.

Sur une même ligne, deux nombres seront séparés par un espace.

Motif carré formé des chiffres 4 ou 2

Écrire un programme qui à partir d'un entier $n > 0$ affiche un carré de côté n, dont la bordure est faite avec le nombre 4 et l'intérieur rempli par le nombre 2. Deux chiffres successifs seront séparés par une espace.

Voici un exemple avec un carré de côté $n = 8$:

```
4 4 4 4 4 4 4 4
4 2 2 2 2 2 2 4
4 2 2 2 2 2 2 4
4 2 2 2 2 2 2 4
4 2 2 2 2 2 2 4
```

```
4 2 2 2 2 2 2 4
4 2 2 2 2 2 2 4
4 4 4 4 4 4 4 4
```

Méthode. On affichera ligne par ligne, en commençant par le haut. Chaque ligne sera affichée par une boucle `for`. La première et la dernière ligne seront traitées à part. Les autres lignes seront traitées par une boucle `for` répétant $n-2$ fois le même motif. On peut aussi imaginer chaque ligne et chaque colonne indexée par un entier `i` et un entier `j` et afficher le chiffre qu'il faut en fonction des valeurs de `i` et de `j`.

Motif carré et sa diagonale

Étant donné un entier naturel $n \geq 1$, on demande de construire un carré de côté n et rempli de chiffres et respectant les deux contraintes suivantes :

— la diagonale descendante est constituée du chiffre 0,

— les autres cases du carré sont constituées du chiffre 1.

Par exemple, si $n = 10$, le carré a l'allure suivante :

```
0 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1 1
1 1 0 1 1 1 1 1 1 1
1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1
1 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 0 1 1
1 1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 1 1 0
```

Deux chiffres côte à côte sur une même ligne sont séparés par exactement un espace.

Triangle décroissant de nombres

Observez le motif suivant obtenu pour $n = 6$:

```
6 6 6 6 6 6
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

On vous donne un entier $n > 0$ et on vous demande de construire le motif formé de n lignes de la manière suivante :

— la 1re ligne contenant le nombre n écrit n fois,

— la 2e ligne contenant le nombre $n-1$ écrit $n-1$ fois,

— la 3e ligne contenant le nombre $n-2$ écrit $n-2$ fois

— et ainsi de suite jusqu'à avoir écrit n lignes.

Sur une même ligne, deux nombres seront séparés par un espace.

Triangles de 0 et de 1

Dans cet exercice, vous allez devoir afficher un motif dépendant d'un entier n. Par exemple, pour $n = 6$ le motif est le triangle suivant :

```
0
1 1
0 0 0
1 1 1 1
0 0 0 0 0
1 1 1 1 1 1
```

Explications :

— il y a n lignes
— les lignes de 0 et les lignes de 1 alternent
— chaque ligne contient un chiffre de plus que la précédente
— deux chiffres voisins sont séparés par un espace.

Ecrire une code qui partant de l'entier `n` génère le motif ci-dessus.

Triangle à lignes alternés

Dans cet exercice, vous allez devoir afficher un motif dépendant d'un entier n et de deux entiers a et b. Par exemple, pour $n = 6$, $a = 42$ et $b = 81$, le motif est le triangle suivant :

```
42
81 81
42 42 42
81 81 81 81
42 42 42 42 42
81 81 81 81 81 81
```

Explications :

— il y a n lignes
— la première ligne ne contient que le nombre `a`
— les lignes de `a` et les lignes de `b` alternent
— chaque ligne contient un nombre de plus que la précédente
— deux nombres voisins sont séparés par un espace.

Ecrire une code qui partant de l'entier `n` génère le motif ci-dessus.

Triangle : lignes consécutives décroissantes

Vous devez écrire un code qui dépend d'un entier n et qui affiche un motif dont un exemple est visible ci-dessous pour $n = 6$:

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

```
6 5 4 3 2 1
```

Explication

— il y a n lignes

— chaque ligne contient un nombre de plus que la précédente

— la k-ème ligne affiche tous les entiers en décroissant de k à 1.

Ecrire un code qui partant d'un entier n génère le motif ci-dessus.

Grille : remplissage par colonnes

Vous devez écrire un code qui affiche un motif dont un exemple est visible ci-dessous :

```
42 52 62 72
43 53 63 73
44 54 64 74
45 55 65 75
46 56 66 76
47 57 67 77
48 58 68 78
49 59 69 79
50 60 70 80
51 61 71 81
```

Ce motif consiste en une grille rectangulaire de dimensions n x p données (n : nombre de lignes, p : nombre de colonnes) de tous les entiers consécutifs à partir d'un entier donné d, le remplissage devant se faire *colonne par colonne*.

Ci-dessus, l'affichage obtenu pour n=10, p=4 et d=42. Noter que deux colonnes sont séparées par un espace.

Vous pourrez remarquer que dans chaque ligne, les valeurs des nombres se suivent avec un écart valant le nombre p de colonnes, par exemple 42, 52, 62 et 72 dans la première ligne.

Sapin

Dessiner, en mode texte, un sapin suivant le modèle ci-dessous. Le dessin dépendra de l'entier $n > 0$ qui représente la hauteur du feuillage (dans l'exemple, $n = 10$).

Avant de faire apparaître le bord gauche du feuillage, il faut placer des espaces (caractère `" "`) en début de ligne.

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
*******************
          #
          #
```

Triangle et son bord

On donne un entier `n` ≥ 1 et on demande de dessiner en mode texte un triangle de côté `n`. Ci-dessous, ce que le programme doit afficher si `n` $= 9$:

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

L'intérieur du triangle est obtenu avec le caractère espace, tapé avec la barre d'espace. De haut en bas, le nombre d'espaces va en augmentant de 1. Chaque côté est formé de `n` astérisques. Bien vérifier le résultat pour `n` valant 1, 2 ou 3.

Carrés concentriques alternés

On demande de construire un motif ayant la forme d'un carré de côté `n`. Par exemple, si `n=9` le motif est :

```
+ + + + + + + + +
+ . . . . . . . +
+ . + + + + + . +
+ . + . . . + . +
+ . + . + . + . +
+ . + . . . + . +
+ . + + + + + . +
+ . . . . . . . +
+ + + + + + + + +
```

ou si `n=10` le motif est :

```
+ + + + + + + + + +
+ . . . . . . . . +
+ . + + + + + + . +
+ . + . . . . + . +
+ . + . + + . + . +
+ . + . + + . + . +
+ . + . . . . + . +
+ . + + + + + + . +
+ . . . . . . . . +
+ + + + + + + + + +
```

Il s'agit de dessiner des carrés concentriques formés alternativement du symbole plus et du symbole point, le carré extérieur étant formé de signe plus et de côté `n`.

En indexant chaque symbole du carré par deux indices `i` ≥ 0 et `j` ≥ 0, on pourra considérer le plus petit des 4 entiers `i`, `j`, `n` $- 1 -$ `i`, `n` $- 1 -$ `j` ce qui mesure la distance du symbole considéré avec les bords du carré (on pourra utiliser la fonction standard `min`).

Liste des entiers k répétés k fois

On considère la suite infinie des entiers écrits dans l'ordre croissant telle chaque entier $k \geq 1$ est répété k fois :

```
1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 7 etc
```

Etant donné p un entier strictement positif, construire une liste L dont les éléments sont, dans l'ordre, les suivants :

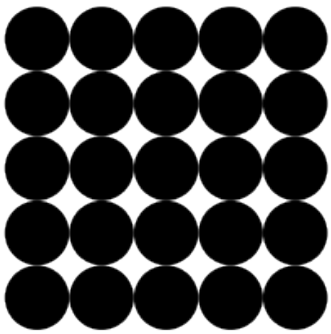
- 1 une seule fois
- 2 deux fois de suite,
- 3 trois fois de suite
- etc
- jusqu'à p qui apparaîtra p fois de suite.

Par exemple, si p=5 alors L est la liste suivante :

```
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]
```

Damier de disques

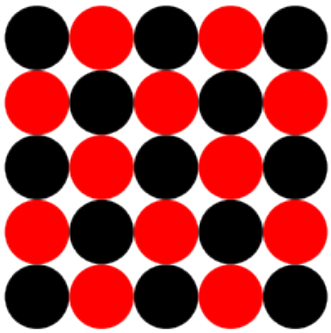
(1) Écrire un code qui, à partir d'un entier $n > 0$, dessine n piles de n disques ($n = 5$ ci-dessous) :



Les disques seront bord à bord, de couleur noire et de diamètre d de votre choix.
Le motif est obtenu en réalisant deux boucles `for` imbriquées : la boucle externe va dessiner chaque ligne des n lignes, la boucle interne va dessiner chaque disque des n disques d'une ligne.
Il y a deux méthodes pour obtenir les centres des différents disques à dessiner.

- 1re méthode : on calcule les coordonnées de chaque centre en imaginant que le disque en bas à droite a pour coordonnées `(0, 0)`. Pour cela, on imaginera que chaque colonne de disques est indexée par un entier `i` de 0 à `n` − 1 et de même pour les lignes ;
- 2e méthode : on utilise deux variables x et y pour systématiquement désigner les coordonnées du centre du disque à dessiner. On dessine alors le disque puis on met à jour x et y pour le centre du disque suivant à dessiner.

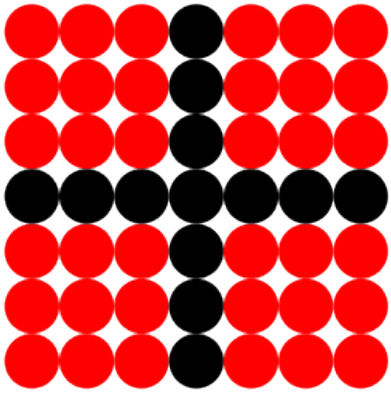
(2) Modifier le code précédent pour obtenir le dessin d'un damier de $n \times n$ disques alternant les couleurs noir/rouge. Le disque en bas à gauche sera noir :



Pour cela, on pourra par exemple imaginer que chaque disque est indexé par un couple `(i, j)` de deux entiers entre `0` et `n-1` où `i` représente la colonne contenant le disque et de même `j` pour la ligne. On observera alors que la couleur noire ou rouge dépend de la parité de `i` et `j`.

Dessin d'un motif en forme de croix

On vous donne un entier impair $n > 1$ et un entier $c > 0$ et vous devez générer un motif de forme carrée de côté c et rempli de n^2 disques colorés. Par exemple, si $n = 7$, le motif a la forme suivante :

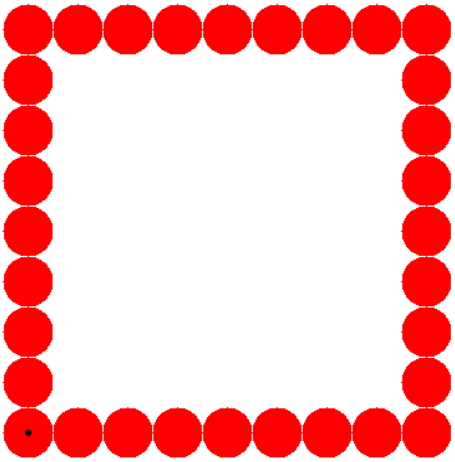


Le motif est un carré rempli de $n \times n$ disques noirs ou rouges. Deux disques voisins seront tangents. Les axes vertical et horizontal de symétrie du carré sont formés de disques noirs et les autres disques sont de couleur rouge.

Dessiner des carrés concentriques alternés

On donne un entier `n` ≥ 1 et un diamètre entier `d` > 0, par exemple `d` $= 40$.

(1) Dessiner un carré tel que chaque côté soit composé de `n` disques rouges de diamètre `d` :



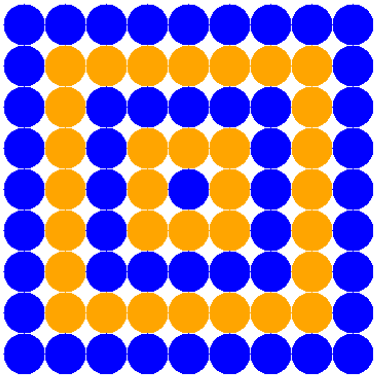
Voici quelques indications :

— le code est composée de 4 boucles `for`

— (optionnel) dans chaque boucle, on n'a besoin de ne dessiner que `n - 1` disques

— faire en sorte que `(x,y)` désigne la position du centre du disque courant.

(2) On demande de dessiner un motif tel que le suivant :



Le motif est une succession de carrés concentriques formés de disques de couleur alternativement orange et bleue. Le carré extérieur est de côté `n` et formé de disques bleus (sur le dessin, $n = 9$).
Voici quelques indications :

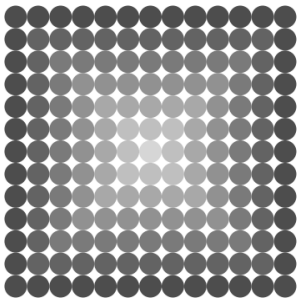
— calculer à l'avance le nombre `N` de carrés à dessiner et boucler sur ce nombre

— à la fin de chaque étape de la boucle précédente, on ajuste la couleur et le côté du prochain carré à dessiner (le nombre diminue de 2 à chaque étape)

— dans de corps de la boucle, placer le code de dessin de la question précédente mais adapté au carré à construire.

Carrés concentriques en dégradé de gris

Dessiner une figure suivant le modèle ci-dessous :



Le côté du carré contient n disques (13 sur le modèle ci-dessus). La figure est formée d'une succession de p carrés concentriques (sur le modèle $p = 7$) constitués de disques. Les couleurs des disques des carrés successifs sont plus clairs lorsqu'on se rapproche du centre. Une couleur grise sera codée en RGB décimal entre 0 et 1, par exemple dans la figure ci-dessus, la couleur du carré extérieur est `c = 0.30` et la couleur suivante s'obtient en ajoutant `0.09`. On poura utiliser que $\texttt{p} = (\texttt{n}+1)//2$. Comme l'option de couleur doit être donnée sous forme de chaîne, on devra utiliser le constructeur str pour faire évoluer la valeur de la couleur.

Triangle de Floyd

(1) Le triangle suivant est un triangle de Floyd à 5 lignes :

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

Le triangle de Floyd à n lignes est obtenu en plaçant sur n lignes successives, dans l'ordre croissant, les entiers consécutifs à partir de 1, la ligne numéro k comportant exactement k entiers. Ecrire un code qui, à partir d'un nombre de lignes $n > 0$, affiche le triangle de Floyd à n lignes. Par exemple, le code exécuté pour $\texttt{n} = 5$ affichera le triangle ci-dessus. Cette question est corrigée dans cette vidéo et si vous êtes en TP, merci de la regarder plus tard !.

(2) Ecrire un code qui à partir d'un entier `N`, construit une liste à deux éléments qui sont le numéro de ligne et le numéro de colonne dans le triangle de Floyd de la position où se trouve l'entier `N`. Par exemple, si $\texttt{N} = 13$, la liste cherchée est `[5, 3]` car 13 se trouve à la 5^{e} ligne et en 3^{e} position dans sa ligne. De même, si $\texttt{N} = 4$, la liste cherchée est `[3, 1]`.

Affichage en tas binaire (boucles `for` imbriquées)

Observez l'exemple suivant que l'on demande de coder dans toute sa généralité : on donne une liste `L` d'entiers telle que

```
[99, 28, 51, 76, 85, 52, 97, 97, 20, 78, 74, 34, 72, 75]
```

et on demande de l'afficher comme suit :

```
99
28 51
76 85 52 97
97 20 78 74 34 72 75
```

La première ligne contient le premier élément, la 2e les deux éléments suivants, la 3e contient les 4 éléments suivants et ainsi de suite, chaque ligne (sauf éventuellement la dernière) contenant deux fois plus d'éléments que la précédente.

Indications

- Le nombre `N` de lignes est la partie entière supérieure du logarithme en base 2 de `n + 1` où `n` est la longueur de `L`
  - pour la partie entière, importer la fonction `ceil` du module `math`
  - pour le logarithme en base 2, importer la fonction `log2` du module `math`
- Boucler sur `N - 1` et imbriquer une boucle pour afficher les éléments de la ligne courante
- En fin de corps de la boucle principale, mettre à jour la longueur de la ligne courante
- Il restera à afficher la dernière ligne (dont la longueur n'est pas une puissance de 2)

Générer toutes les permutations

On se donne un entier `n` ≥ 1 et on veut générer itérativement toutes les permutations des entiers de 1 à `n`, autrement dit tous les alignements possibles de ces entiers. Par exemple, si `n` $= 4$, la liste `[3, 1, 4, 2]` est une permutation, parmi 24 possibles. On demande d'écrire un code qui crée une liste contenant, sous forme de listes, toutes les permutations des entiers de 1 à `n`. Par exemple, si `n` $= 3$, on devra trouver une liste de 6 listes comme celle-ci :

```
[[3, 2, 1], [2, 3, 1], [2, 1, 3], [3, 1, 2], [1, 3, 2], [1, 2, 3]]
```

On appliquera l'algorithme décrit ci-après. Pour construire la liste `L` des permutations des entiers de 1 à `n`, on suppose qu'on aura déjà construit la liste `M` des permutations de 1 à `n - 1`. On obtiendra alors toutes les permutations cherchées en insérant l'objet `n` à l'une des `n` positions possibles d'une quelconque permutation venant de la liste `M`. Par exemple, avec `n` $= 4$, la permutation `[2,1,3]` donnera les `n` permutations suivantes :

```
[4, 2, 1, 3], [2, 4, 1, 3], [2, 1, 4, 3], [2, 1, 3, 4]
```

et si on répète cette opération avec les 6 permutations de 1, 2 et 3, on obtiendra *toutes* les permutations sur 1, 2, 3 et 4.

Il s'agit de créer les permutations itérativement, sans utiliser une fonction récursive.

Tester avec une boucle `for`

Dans chaque question, les tests sont à effectuer au sein d'une boucle `for`.

(1) Le code suivant montre comment, à partir d'un entier `n`, générer la liste formée de `n-1`, `n` et `n+1` :

```
n=42
L=[n-1, n, n+1]
```

Tester le code précédent pour les entiers n valant 42, 100, 0 et 2020. On devra obtenir l'affichage suivant :

```
42 -> [41, 42, 43]
100 -> [99, 100, 101]
0 -> [-1, 0, 1]
2020 -> [2019, 2020, 2021]
```

(2) Le code suivant montre (et on l'admettra) comment coder un booléen qui traduit que le minimum d'une liste L d'entiers est unique :

```
L=[81, 42, 87, 63, 50, 42, 42]
miniUnique = L.count(min(L))==1
print(miniUnique)
```

```
False
```

Appliquer le code précédent aux 5 listes suivantes :

```
[42], [42, 42, 42], [81, 42, 87, 63, 50, 42, 42], [80, 42, 53, 64]
```

Le code devra afficher

```
[42] -> True
[42, 42, 42] -> False
[81, 42, 87, 63, 50, 42, 42] -> False
[80, 42, 53, 64] -> True
```

Tests aléatoires

Le code suivant montre comment coder un booléen qui traduit que le minimum d'une liste L d'entiers est unique :

```
L=[81, 42, 87, 63, 50, 42, 42]
miniUnique = L.count(min(L))==1
print(miniUnique)
```

```
False
```

Générer aléatoirement 10 listes d'entiers entre 0 et 9, de longueur variable et s'assurer que le code précédent fonctionne en testant les 10 listes avec une boucle `for`. La sortie pourra être de la forme suivante :

```
[7, 9, 3, 0, 6] -> True
[2, 4, 4, 7, 3, 6, 2] -> False
[1, 7, 0] -> True
[1, 1, 3, 2, 5, 1] -> False
[6, 9, 2] -> True
[1, 9, 8, 7, 9, 2] -> True
[4, 7, 0, 7, 8, 0] -> False
[4, 9, 6] -> True
[8, 8, 1, 7] -> True
[4, 9, 5, 0, 6, 4, 8, 2] -> True
```

5 Boucle while

Boucle while

Un problème typique

Étant donné un entier n, on cherche un entier noté m qui soit le premier multiple de 10 supérieur ou égal à n. Si par exemple $n = 42$ alors $m = 50$: en effet, 50 est bien un multiple de 10 et il n'y en pas d'autre entre 42 et 50. Voici d'autres exemples :

n	2038	420	0
m	2040	420	0

L'idée à implémenter

Les multiples de 10 sont les entiers : $0, 10, 20, 30$ etc. Ils s'écrivent sous la forme $10k$ où k varie à partir de 0.

Pour trouver m, on parcourt les multiples de 10 depuis 0 et on s'arrête une fois qu'on a atteint ou dépassé l'entier n donné. Autrement dit, on énumère les multiples m de 10 **tant que** $m < n$. Le parcours est montré dans le tableau ci-dessous où on suppose que $n = 42$.

k	0	1	2	3	4	5
$10k < 42$?	oui	oui	oui	oui	oui	non

Implémentation en Python

L'idée de « tant que », « jusqu'à ce que », « aussi longtemps que » est traduite dans le code Python par l'instruction `while` (qui veut dire « tant que » en anglais).

Le code suivant répond au problème posé :

`while_typique.py`

```
1 n = 42
2 k = 0
3
4 while 10 * k < n:
5     k = k + 1
6
7 print(10 * k)
```

```
8 50
```

On voit (cf. lignes 7 et 8) que la solution au problème est 50.

Analyse de code

On passe en revue le code de `while_typique.py`.

Vocabulaire de la boucle `while`

— Lignes 4-5 : ces deux lignes forment une seule et même instruction, dite *instruction* `while`.

— Ligne 4 : l'*en-tête* de la boucle `while` est la partie qui commence avec `while` et se termine avant le séparateur **:** (deux-points).
— Ligne 5 : le *corps* de la boucle `while` qui est le bloc indenté sous les deux-points.

Répétition

Une boucle `while` répète une action (ligne 5) tant qu'une *condition*, ici (cf. ligne 4) :

```
10 * k < n
```

reste vraie. Plus précisément,

— si la condition est vraie, l'exécution entre dans le corps de la boucle et à la fin du corps de la boucle, la condition est à nouveau testée (d'où le terme de *boucle*)
— si la condition est fausse, l'exécution du programme va au-delà du corps de la boucle, ici à la ligne 6.

Variable de contrôle

Une boucle `while` fait souvent appel à une *variable de contrôle*, ici `k`, qui évolue pendant la boucle. Typiquement,

— cette variable est **initialisée** avant la boucle, ici ligne 2 ;
— cette variable est **réaffectée**, dans le corps de la boucle `while`, ici par l'instruction ligne 5 ;
— la condition à tester (ici ligne 4) est différente d'un tour de boucle à l'autre car elle dépend de `k` qui, entre-temps, a varié.

Remarques

— Ligne 4 : dans l'immense majorité des cas, le corps d'une boucle `while` est indenté.
— Ligne 1 : il est possible de faire d'autres essais de code en changeant juste la valeur de n. Par exemple, si on change n en 2038, le programme affichera 2040.
— Il se peut que l'exécution du code n'entre même pas dans le corps de la boucle `while`. Par exemple, si ligne 1, on choisit `n=0` au lieu de `n=42`, le test `10 * k < n` (ligne 4) échoue **immédiatement** et donc le programme continue à la ligne 6 sans même passer par la ligne 5.
— À la différence de la syntaxe des boucles while du C et de Java, le booléen évalué avant chaque tour de boucle n'a pas besoin d'être entouré de parenthèses.

Comprendre comment s'exécute une boucle `while`

Modifions le code précédent `while_typique.py` pour mieux comprendre l'exécution de la boucle `while` :

`while_typique_affichage.py`

```
1 n = 42
2 k = 0
3 print("avant while", "-> k=", k)
4 print()
5
```

```
6 while 10 * k < n:
7   print("debut while", "k=", k, "-> 10*k=", 10*k)
8   k = k + 1
9   print("fin while", "-> k=", k)
10  print()
11 print("apres while")
12 print(10 * k)
```

```
13 avant while -> k= 0
14
15 debut while k= 0 -> 10*k= 0
16 fin while -> k= 1
17
18 debut while k= 1 -> 10*k= 10
19 fin while -> k= 2
20
21 debut while k= 2 -> 10*k= 20
22 fin while -> k= 3
23
24 debut while k= 3 -> 10*k= 30
25 fin while -> k= 4
26
27 debut while k= 4 -> 10*k= 40
28 fin while -> k= 5
29
30 apres while
31 50
```

En plus du code initial, `while_typique_affichage.py` contient des instructions d'affichage (lignes 3, 7, 9 et 11) et des instructions de sauts de ligne (lignes 4 et 10) dans la sortie pour observer l'évolution de `k` et de `10 * k` avant, pendant et après la boucle `while`.

L'exécution du programme est la suivante :

— Lignes 2 et 7 : la valeur de `k` avant le commencement de la boucle `while`

— Ligne 6 : `k` vaut 0. Le test de la boucle `while` est effectué : a-t-on `0 < 42` ? La réponse est *oui* donc, l'exécution du code continue dans le corps de la boucle `while`

— Les affichages sont effectués : `k` est changé de 0 à 1 (lignes 8 et 16).

— Ligne 6 : le test de la boucle `while` est à nouveau effectué : a-t-on $10 < 42$? La réponse est *oui* et l'exécution entre à nouveau dans le corps de la boucle. Cette opération se répète jusqu'à ce que `k = 5` (ligne 8 et ligne 28).

— Ligne 6 : le test de la boucle `while` est effectué : a-t-on $50 < 42$? Cette fois, la réponse est *non* donc l'exécution quitte la boucle et continue lignes 11 et 12, cf. lignes 30 et 31.

— Ligne 12 : le résultat affiché (cf. ligne 31) est bien le résultat demandé. Comme le test `10 * k < n` a échoué pour la première fois, c'est qu'on a `10 * k >= n` avec `k` minimal et c'est bien ce que l'on cherchait.

Boucle for vs boucle while

Python dispose de deux types de boucle : la boucle `for` et la boucle `while`. Elles ont chacune des usages spécifiques.

Usage canonique de la boucle for

La boucle for s'emploie si :

— une action doit être effectuée un nombre prédéterminé de fois, et, surtout, connu avant l'exécution de la boucle. Par exemple, s'il faut afficher 42 nombres vérifiant une certaine propriété, on utilise une boucle du type `for i in range(42)`.

— on parcourt une liste entre deux indices connus à l'avance, ce qui peut être la totalité de la liste ou, par exemple, de la 5e position à l'avant-avant dernière.

Par exemple, étant donné une liste L de nombres, si on cherche à déterminer combien L contient d'entiers positifs, on utilisera une boucle d'en-tête du type `for x in L` ou encore `for i in range(len(L))`.

Usage canonique de la boucle while

La boucle `while` s'emploie si une action doit être répétée tant qu'une certaine condition reste vraie et sans qu'on sache à l'avance combien de fois la boucle va être répétée. Par exemple,

— on a une liste d'entiers et on cherche le **premier** entier de la liste qui soit pair ;

— on cherche la plus petite puissance de 10 qui dépasse un entier donné.

De même, le codage d'une situation utilisant les termes « dès lors que », « à partir du moment où » nécessitent souvent d'employer une boucle `while`. Par exemple, si on donne une liste `L` d'entiers et qu'on demande de faire la somme dès éléments `x` de `L` dès lors que `x>0`, il va s'agir de parcourir la liste `L` jusqu'à ce que ses termes soient négatif ou nuls et ne commencer la somme qu'une fois la boucle `while` interrompue.

Usage inapproprié d'une boucle while

Le programme ci-dessous utilise une boucle `while` pour afficher les 5 premiers multiples de 10 :

```
1 i = 1
2 while i <= 5:
3   print(10 * i)
4   i = i + 1
5
6 print("Fin")
```

```
7 10
8 20
9 30
10 40
11 50
12 Fin
```

Le code `while_inapproprie.py` est un usage non recommandé de la boucle while. En effet, le code contient des instructions inutiles par rapport à une boucle for :

— Ligne 1 : initialisation inutile, c'est automatique avec une boucle for.

— Ligne 2 : test de condition, automatique avec la boucle for.

— Ligne 4 : incrémentation inutile, c'est automatique avec une boucle for.

Comparer avec un code équivalent écrit avec une boucle for :

```
1 for i in range(5):
2     print(10 * i)
3
4 print("Fin")
```

Boucle while et parcours de liste

Usuellement, le parcours d'une liste est associé à une boucle `for`. Toutefois, dans certaines circonstances, il est plus approprié de parcourir une liste avec une boucle `while`.

Soit à définir un code Python qui, partant d'une liste `L` d'entiers détermine si `L` ne contient que des entiers positifs. Plus précisément, le code doit définir un booléen nommé `tousPositifs` qui vaut True si L ne contient que des entiers positifs et False sinon.

L'idée est de parcourir la liste L avec une boucle while à l'aide d'un indice `i` **valide** et tant que le terme courant L[i] est positif ou nul :

```
1 L= [31, 82, -42, 32, -81]
2
3 i=0
4 n = len(L)
5
6 while i<n and L[i]>=0:
7     i = i + 1
8
9 tousPositifs = (i==n)
10
11 print(L, tousPositifs)
```

```
12 [31, 82, -42, 32, -81] False
```

— Ligne 6 : la condition doit D'ABORD gérer la sortie de liste d'où la condition `i < n` qui évite le **débordement d'indice**.

— Ligne 6 : la condition doit aussi filtrer suivant le critère, ici le fait que chaque terme soit positif.

— Ligne 9 : Il s'agit de définir le booléen.

  — Si un élément négatif est apparu dans la liste, soit `i` son indice. Alors comme le booléen `i<n and L[i]>=0` vaut `False` (puisque `L[i]<0`), la boucle `while` s'arrête et l'indice `i` vérifie `i<n`.

  — En revanche, si aucun élément de la liste n'est strictement négatif, le corps de la boucle `while` sera exécuté tant que la condition `i<n` sera vraie. Lorsque `i = n-1`, le corps de la boucle est exécuté et `i` devient `i+1` donc `i=n` et la boucle se termine.

Donc, ce qui différencie les deux cas, c'est la valeur de `i` en sortie de boucle : dans le second cas, `i` vaut `n` et pas dans le premier cas, ce qui explique la définition ligne 9 de `tousPositifs`.

Même si c'est un détail, noter que le code suivant devrait être évité :

```
1 L= [31, 82, -42, 32, -81]
2
3 i=0
4
5 while i< len(L) and L[i]>=0:
6     i = i + 1
7
8 tousPositifs = (i==len(L))
9
10 print(L, tousPositifs)
```

— Ligne 5 : à chaque tour de boucle, l'appel `len(L)` est recalculé alors que `len(L)` est un nombre qui ne varie pas.

Boucle `for` vs boucle `while` : tableau récapitulatif

Utiliser une boucle while par rapport à une boucle for nécessite plus de soin comme le résume le tableau ci-dessous :

	Boucle `while`	Boucle `for`
Initialisation indice `i`	manuelle	automatique (en-tête de la boucle)
Incrémentation de l'indice `i`	manuelle	automatique
Sortie de boucle	manuelle (avec drapeau dans l'en-tête)	nécessite des instructions de saut (`break`, `return`)
Sortie de liste de longueur `n`	manuelle (`i<n`)	automatique

Exercice type – La petite exponentielle contre le grand produit

Trouver le plus grand entier $n > 0$ tel que $1.0001^n < 10000n$ (on trouvera $n = 214893$).

Solution

Posons `a = 1.0001`. Comme `a` est très proche de 1, les puissances de `a` augmentent mais très lentement. Par exemple :

```
1 a=1.0001
2 print(a**10000)
```

```
3 2.7181459268249255
```

Toutefois, comme `a` est strictement plus grand que 1, le nombre a^n peut être rendu aussi grand que l'on veut en choisissant `n` assez grand.

Pour le code lui-même, l'idée est très simple : on compare a^n et $10000 \times n$ en faisant varier `n` suivant les entiers consécutifs jusqu'à ce que le premier dépasse le second. D'où le code :

```
1 k=1
2
3 while 1.0001**k < 10000*k:
4     k=k+1
5
6 print("n =", k-1)
```

```
7 n = 214893
```

— Ligne 1 : l'énoncé demande de trouver un entier $k > 0$.

— Ligne 6 : en sortie de boucle `while` l'entier `k` trouvé est le premier qui vérifie $1.0001^k \geq 10000 \times k$. C'est l'entier `k` précédent qui est tel que l'en-tête de la boucle `while` est vrai. C'est donc lui la solution du problème.

On peut bien sûr vérifier que le résultat `n = 214893` est conforme :

```
1 k = 214893
2 print(k, "->", 1.0001**k < 10000*k)
3
4 k = 214894
5 print(k, "->", 1.0001**k < 10000*k)
```

```
6 214893 -> True
7 214894 -> False
```

Exercice type – Premier entier pair

Ecrire un code qui à partir d'une liste `L` d'entiers :

— affiche le premier entier pair rencontré dans la liste `L` si `L` contient au moins un entier pair,

— affiche le nombre -1 s'il n'existe aucun entier pair dans la liste.

Voici quelques exemples de comportement du programme :

```
[1, 3, 42, 51, 32, 9] -> 42
[1, 3, 42, 51, 33] -> 42
[1, 3, 51, 33] -> -1
[1, 3, 51, 33, 42] -> 42
[42, 82] -> 42
[42] -> 42
[81] -> -1
[] -> -1
```

Solution

Version `for`

```
L=[1, 3, 41, 51, 31, 9]
ok=False
i=0
n=len(L)

for i in range(n):
    if L[i]%2==0 and not ok:
        print(L[i])
        ok=True
if not ok:
    print(-1)
```

Il suffit d'appliquer la technique vue dans le paragraphe *Boucle while et parcours de liste.*

```
L=[1, 3, 42, 51, 32, 9]

i=0
n=len(L)

while i<n and L[i]%2 == 1:
    i = i + 1

if (i==n):
    print(L, "->", -1)
else:
    print(L, "->", L[i])
```

```
[1, 3, 42, 51, 32, 9] -> 42
```

— Lignes 6-7 : la boucle `while` parcourt la liste `L` jusqu'à ce que la liste soit épuisée ou bien qu'un élément pair soit rencontrée.

— Lignes 9-10 : en sortie de boucle, ou bien l'indice `i` est hors de la liste et dans ce cas la liste ne contient aucun élément pair (si un élément pair est présent uniquement en dernière position alors, en sortie de boucle, `i` vaut `n-1`.

Pour traiter le cas de plusieurs listes, au lieu de recopier du code, on place les liste dans une liste `tests` et on parcourt la liste `tests` pour tester chacune des lignes :

```
tests = [
[1, 3, 42, 51, 32, 9],
[1, 3, 42, 51, 33],
[1, 3, 51, 33],
[1, 3, 51, 33, 42],
[42, 82],
[42],
[81],
[]]

```

```
11 for L in tests:
12     i=0
13     n=len(L)
14 
15     while i<n and L[i]%2 == 1:
16         i = i + 1
17 
18     if (i==n):
19         print(L, "->", -1)
20     else:
21         print(L, "->", L[i])
```

```
22 [1, 3, 42, 51, 32, 9] -> 42
23 [1, 3, 42, 51, 33] -> 42
24 [1, 3, 51, 33] -> -1
25 [1, 3, 51, 33, 42] -> 42
26 [42, 82] -> 42
27 [42] -> 42
28 [81] -> -1
29 [] -> -1
```

EXERCICES

Somme harmonique dépassant 18

Trouver le plus petit entier `n` tel que $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} > 18$.

ppcm

Construire une variable `ppcm` qui référence le plus petit commun multiple de deux entiers donnés a et b. Par exemple, si les entiers sont $a = 16$ et $b = 60$ alors `ppcm = 240` puisque si on liste les multiples de a, on obtient $16, 32, 48, 64, 80, \ldots$ et les multiples de b d'autre part, à savoir $60, 120, 180, \ldots$ et la plus petite valeur commune entre ces listes est 240.

Il est attendu d'écrire une version « naïve » (et non pas d'utiliser l'algorithme d'Euclide pour obtenir d'abord le pgcd). La version naïve consistera par exemple à examiner les multiples non nuls de `a` et à tester jusqu'à ce qu'un de ces multiples soit multiple de `b`.

Même avec un code « naïf », vous devez pouvoir traiter des entiers `a` et `b` de l'ordre de un million.

Restes chinois

Soit `n` le plus petit entier vérifiant les deux propriétés suivantes :

— si on divise `n` par 2042, on obtient 2040 pour reste

— si on divise `n` par 2043, on trouve 2041 pour reste.

Trouver bestialement `n` avec une boucle `while`.

Diviser par deux ou retirer 1

Observez le motif ci-dessous formé de 10 entiers. On part d'un entier $a > 0$, ici $a = 81$, et on construit une suite d'entiers ainsi :

```
81 80 40 20 10 5 4 2 1 0.
```

Le procédé de construction est défini ainsi :

— si l'entier x est pair, le successeur de x de la suite est la moitié de x

— sinon, ce successeur est $x - 1$;

— la suite se termine dès qu'un élément de la suite vaut 0.

Par exemple, dans la suite ci-dessus, le successeur de 80 est 40 car 80 est pair et que sa moitié est 40, et de même le successeur de 5 est 4 car 5 est impair et que $4 = 5 - 1$.

(1) On vous donne un entier $a \geq 0$ et on vous demande d'afficher la suite générée à partir de a et de calculer la longueur de la suite ainsi générée. Par exemple, si $a = 79$ vous devez afficher la suite suivante :

```
79 78 39 38 19 18 9 8 4 2 1 0
```

et la longueur à calculer est de 12.

(2) Parmi toutes les listes commençant par un entier entre 0 et 1000, quelle est la longueur maximale d'une suite ?

Soustraire jusqu'à équilibre

On donne deux entiers strictement positifs `a` et `b`. On répète l'action consistant à soustraire le plus petit des deux nombres au plus grand des deux jusqu'à ce que les deux nombres soient égaux. Ci-dessous, le déroulement de la répétition si au départ $a = 5$ et $b = 7$:

```
 a b
------
 5 7
 5 2
 3 2
 1 2
 1 1
```

Ecrire un code qui exécute le programme décrit ci-dessus et détermine le nombre total de soustractions. Dans l'exemple, ce nombre est de 4 soustractions : $7 - 5$ puis $5 - 2$ puis $3 - 2$ et enfin $2 - 1$.

Cet exercice a été proposé sur le forum Python d'OpenClassrooms.

Divisions successives par deux

On part d'un entier `n>1` et :

— s'il est pair, on le divise par 2

— s'il est impair, on lui ajoute 1 et on divise par 2.

Si par exemple `n=42` on obtient 21 et si `n=99` on obtient 50.

Maintenant, on répète ce procédé jusqu'à ce que le nombre obtenu vaille 1. On demande d'écrire un code qui détermine le nombre de divisions effectuées.

Par exemple, si `n=25`, les différentes étapes seront :

```
13
7
4
2
1
```

et le nombre de divisions effectuées est 5.

Tirage du loto

Écrire un code qui crée un tirage aléatoire des 5 numéros d'un loto. On rappelle qu'un tirage de loto est formée de 5 numéros **distincts** entre 1 et 49. On ne tirera pas de « numéro Chance ».

Ainsi, le code pourra générer une liste telle que `[42, 32, 48, 47, 20]` mais pas telle que `[42, 32, 48, 42, 20]` puisque dans cette dernière liste, le numéro 42 apparaît deux fois.

On utilisera une boucle `while` pour placer au fur et à mesure les numéros tirés dans une liste `L` initialement vide. On rappelle que si `L` est une liste d'entiers et `x` un entier alors l'expression `x in L` vaut `True` si l'entier `x` est dans la liste `L` et `False` sinon.

Calculer le nombre de chiffres d'un entier

On donne un entier $n \geq 0$ et on cherche le nombre de chiffres de `n` (ce nombre de chiffres sera appelé `nchiffres`). Par exemple, si $n = 2020$ alors `nchiffres = 4` ou encore si $n = 42$ alors `nchiffres = 2`.

L'idée pour calculer `nchiffres` est de compter le nombre de divisions successives de `n` par 10 jusqu'à ce que le quotient (entier) soit nul. Par exemple

— le quotient entier de 2020 par 10 est 202,
— le quotient entier de 202 par 10 est 20,
— le quotient entier de 20 par 10 est 2,
— le quotient entier de 2 par 10 est 0.

C'est parce 2020 a justement 4 chiffres qu'on a effectué 4 divisions successives par 10 avant d'obtenir un quotient nul.

Ecrire un code Python utilisant une boucle `while` et qui détermine `nchiffres` connaissant `n`.

Voici quelques exemples de comportements

```
42 -> 2
2020 -> 4
10 -> 2
7 -> 1
0 -> 1
74152003636525362514521174152363685419854l -> 42
```

Nombre de chiffres d'un entier

Soit un entier $n \geq 0$ et soit N le nombre de chiffres de n. Par exemple, si $n = 25420$ alors $N = 5$.

On veut écrire un code qui détermine N avec la méthode suivante et qu'on va illustrer sur l'exemple de $n = 25420$. On calcule les puissances 10^k donc $1, 10, 100$, etc jusqu'à ce que $10^k > 25420$, ce qui s'obtient pour pour $k = 5$ puisque $10^5 = 100000 > n$ et le nombre de chiffres de n est alors $k = 5$

En appliquant cette méthode à l'aide d'une boucle `while`, écrire un programme qui calcule le nombre de chiffres de n.

Nombre de zéros qui terminent un entier

On donne un entier $n \geq 0$, par exemple $n = 4205000$ et on demande de déterminer le nombre de zéros qui terminent l'écriture décimale du nombre ; dans l'exemple précédent, ce nombre est 3. On pourra remarquer que c'est l'exposant de la plus grande puissance de 10 dont n soit multiple. On utilisera obligatoirement une boucle `while`. Ne pas oublier le cas $n = 0$.

Inverser l'écriture d'un nombre

On donne un entier, par exemple `n = 25048` et on demande de construire l'entier `m`, dans l'exemple $m = 84052$, dont l'écriture décimale est celle de n mais inversée. Le nombre n peut se terminer par des zéros, par exemple si $n = 25000$ alors $m = 52$.

L'algorithme à utiliser est illustré sur le cas de `n = 25048`. Au départ, on dispose d'une variable `s=0` et qui va évoluer en prenant les valeurs des différents nombres inversés, dans notre cas, il s'agira de :

$$8, \quad 84, \quad 840, \quad 8405, \quad 84052.$$

On pourra remaquer que chaque valeur de `s` s'obtient en multipliant par 10 la précédente et en ajoutant un chiffre de n ; par exemple, 8405 s'obtient par $840 \times 10 + 5$.

On fera des divisions successives de n par 10, en utilisant à chaque fois le quotient et le reste. Le code utilisera une boucle `while` et qui se terminera lorsque le quotient courant par 10 vaudra 0.

Cet exercice est repris d'un exercice du site d'apprentissage AlgoPython.

Etre un entier léger

Un entier strictement positif sera dit « léger » si tous ses chiffres, sauf le premier, sont nuls. Par exemple, les entiers 7000, 40 ou 5 sont légers tandis que 2200, 50050 ou 42 ne le ont pas. Etant donné un entier strictement positif `n`, écrire un code utilisant une boucle `while` et créant un booléen `estLeger` qui vaut `True` si `n` est un entier léger et `False` sinon. On pourra diviser `n` par 10 tant que le quotient est encore multiple de 10. Tester sur tous les entiers donnés en exemple ci-dessus.

Placement dans une suite croissante

On donne une liste `L` croissante d'entiers ainsi qu'un entier `x`.

On demande de construire le nombre `k` d'éléments `y` de `L` tels que $y \leq x$. On utilisera une boucle `while`.

Ci-dessous, quelques exemples de comportements attendus :

```
L = [33, 36, 56, 76, 88], x = 62
k = 3
--------------
L = [33, 36, 62, 62, 76, 88], x = 62
k = 4
--------------
L = [33, 36, 62, 62], x = 62
k = 4
--------------
L = [33, 36, 62, 62], x = 21
k = 0
--------------
L = [42], x = 81
k = 1
--------------
L = [81], x = 42
k = 0
--------------
L = [42], x = 42
k = 1
--------------
```

Insertion dans une suite croissante

On donne une liste `L` croissante de `n` entiers ainsi qu'un entier `x`.

On demande de construire une **nouvelle** liste `M` **croissante**, formée des `n` entiers de `L` ainsi que de `x` mais placé convenablement pour que la liste obtenue soit toujours croissante. On utilisera une boucle `while`.

Ci-dessous, quelques exemples de comportements attendus :

```
L = [1, 3, 7, 7, 8], x = 5
M = [1, 3, 5, 7, 7, 8]
--------------
L = [1, 2, 3], x = 2
M = [1, 2, 2, 3]
```

```
--------------
L = [1, 2, 8, 8], x = 0
M = [0, 1, 2, 8, 8]
--------------
L = [0], x = 5
M = [0, 5]
--------------
L = [5], x = 0
M = [0, 5]
--------------
```

Pas d'impair

Soit une liste L formée d'entiers. Écrire, à l'aide d'une boucle `while`, un booléen `queDesPairs` traduisant que L ne contient que des entiers pairs.

Comportement attendu :

```
[82, 31, 82] -> False
[82, 12, 46] -> True
[82] -> True
[81] -> False
```

Liste d'entiers en miroir

On donne une liste L d'entiers, par exemple `L = [4, 2, 2, 4]` et on demande d'écrire une variable booléenne `estEnMiroir` qui vaut `True` si cette liste est en miroir, autrement dit si

— le premier et le dernier élément de L sont égaux,

— le deuxième et l'avant-dernier élément de L sont égaux,

— et ainsi de suite jusqu'à épuisement de la liste.

Dans le cas de la liste `L = [4, 2, 2, 4]` ou encore `L = [4, 2, 4]`, la liste est en miroir. Dans le cas de la liste `L = [4, 2, 1]`, la liste n'est pas en miroir.

Votre code doit impérativement utiliser une boucle `while` pertinente.

Listes « opposées » (boucle `while`)

Ecrire un code qui partant deux listes d'entiers L et M, de même longueur, crée un booléen **sontOpposees** valant `True` si les deux listes sont « opposées » et `False` sinon. Deux listes sont considérées comme « opposées » si, à des indices identiques, elles possèdent des éléments opposés (comme -81 et 81). Voici quelques exemples de comportements attendus :

```
[81, -12, 0, -81, -31] [-81, 12, 0, 81, 31] -> True
                                  [-81] [81] -> True
                               [0, 0] [0, 0] -> True
                                    [ ] [ ] -> True
                         [81, -12] [-81, -12] -> False
                       [-81, 12, 0] [81, -12] -> False
```

Vous ne devez pas utiliser de boucle `for` mais une boucle `while`

Liste en miroir opposé

La liste `[-42, 2016, 2020, -2020, -2016, 42]` sera dite une *liste en miroir opposé* : si on énumère la liste de la gauche vers la droite, c'est comme si on énumérait **de la droite vers la gauche** la liste des OPPOSÉS.

On vous donne une liste non vide d'entiers est vous devez construire un booléen `miroirOpp` qui vaut True si L est une liste d'entiers en miroir opposé et False sinon. Voici quelques exemples de comportements attendus :

```
[1, -2, -5, 3, -3, 5, 2, -1] -> True
[1, -2, -5, 3, 0, 0, 0, -3, 5, 2, -1] -> True
[42, 0, -42] -> True
[42, 81, -81] -> False
[5] -> False
[0] -> True
[-42, 42] -> True
```

Alternance de parité, version `while`

On donne une liste L d'entiers et on demande de créer un booléen **alterneParite** valant `True` si les éléments de L se suivent en alternant de parité et `False` sinon. On utilisera une boucle `while`. Voici des exemples de comportements attendus :

L	Alternance de parité
`[81, 32, 9, 12]`	`True`
`[32, 9, 32, 65]`	`True`
`[32, 9, 31, 82]`	`False`
`[81]`	`True`

Suite croissante d'entiers consécutifs (avec `while`)

L'exercice doit être codé en utilisant une boucle `while`.

Écrire un code qui à partir d'une liste L d'entiers définit une variable booléenne nommée **consecutifs** qui vaut **True** si la liste est constituée d'entiers CONSÉCUTIFS croissants et **False** sinon. Ci-dessous, voici quelques exemples de comportements attendus

```
[81, 82, 83]       -> True
[82, 81, 83]       -> False
[2020, 2038, 3000] -> False
[81]               -> True
```

Que des entiers consécutifs

On donne une liste non vide d'entiers `L` et on demande de déterminer le plus petit indice `i` de `L` tels que tous les éléments de `L` à partir de l'indice `i` soient des entiers consécutifs. Par exemple, si `L` est la liste :

```
L=[34, 35, 10, 25, 40, 42, 43, 44]
```

alors l'indice attendu est `i = 5` car `L[i] = 42` et les éléments suivants jusqu'à la fin de la liste sont consécutifs (42, 43, 44) mais cela ne se produit pas avant puisque 40 et 42 ne sont pas consécutifs.

Que des 81 puis que des 12

Soit une liste `L` non vide formée d'abord d'un certain nombre de fois de l'entier 81 puis d'un certain nombre de fois de l'entier 12. Par exemple, `L=[81, 81, 81, 12, 12, 12, 12]` ou encore `L=[81, 12]` ou même `L=[12, 12]` ainsi que `L=[81, 81, 81]`.

Écrire, à l'aide d'une boucle `while`, une variable `indicePremier_12` qui définit le premier indice du terme de `L` qui vaut 12. Par exemple, si `L = [81, 81, 81, 12, 12, 12, 12]` alors `indicePremier_12 = 3`. Si 12 n'apparaît pas dans `L`, la variable `indicePremier_12` vaudra `n` où `n` est la longueur de la liste `L`.

Exemples de comportement :

```
[81, 81, 81, 12, 12, 12, 12] -> 3
[81, 12] -> 1
[81, 81] -> 2
[12, 12] -> 0
```

Déclin (boucle `while`)

On dit qu'une suite d'entiers est une *suite en déclin* si

— la suite comporte au moins deux éléments

— seul le dernier terme de la suite est strictement inférieur au précédent.

Voici trois exemples de *suites en déclin* :

```
2 4 5 5 2
4 1
5 5 5 6 6 6 5
```

En revanche, la suite suivante n'est pas une *suite en déclin* :

```
4 4 6 3 5 1
```

car le terme 3 de la suite est strictement inférieur à son précédent (qui vaut 6) alors que 3 n'est pas le dernier terme de la suite.

De même, la suite suivante n'est pas une *suite en déclin* :

```
4 4 6 6
```

car le dernier terme de la suite n'est pas strictement inférieur à son précédent.

On se donne une liste `L`. Construire un booléen `estEnDeclin` qui vaut `True` si la suite représentée par `L` est en déclin et `False` sinon. La liste devra être parcourue avec une boucle `while`.

Négatif, nul, positif

On donne une liste `L` d'entiers et on demande d'écrire un booléen `neg0pos` qui teste si la liste `L` respecte le motif vérifiant les propriétés suivantes :

— un entier strictement négatif ne peut être suivi que de 0

— un entier nul dans la liste ne peut être suivi que d'un entier strictement positif

— un entier strictement positif dans la liste ne peut être suivi que d'un entier strictement négatif.

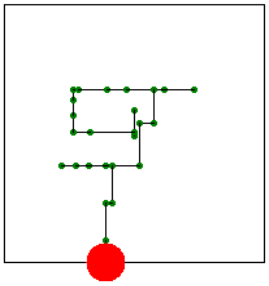
Le code devra obligatoirement utiliser une boucle `while` pertinente.

Exemples de comportement :

```
[0, 2, -1, 0, 3, -2, 0, 5] -> True
[0, 2, -1, 0, 3, -2, 0] -> True
[0, 2, -1, 0, 3, -2] -> True
[-2, 0, 2, -1, 0, 3, -2, 0, 5] -> True
[1, -1] -> True
[0] -> True
[2, -1, 0, 3, 0, -2, 0, 5] -> False
[0, 2, -1, -4, 0, 3, -2, 0, 5] -> False
```

Grenouille dans un enclos

Cet exercice est un exercice de dessin. Une grenouille se déplace dans un enclos carré limité par les 4 points $(200, 200)$, $(-200, 200)$, $(-200, -200)$ et $(200, -200)$. Sur le dessin, la grenouille se déplace aléatoirement horizontalement ou verticalement par bond d'une distance aléatoire d'au plus 50 :



Tracez l'enclos, placez la grenouille au centre et représentez la trajectoire de la grenouille jusqu'à ce qu'elle touche le grillage. Marquez d'un point vert chaque nouvelle position de la grenouille et d'un gros point rouge l'endroit où la grenouille entre en contact avec la clôture.

Plateau le plus long

On donne une liste d'entiers et on recherche un entier qui est répété consécutivement dans la liste le plus grand nombre de fois (ce qui forme un « plateau »). Par exemple, la liste

```
[4, 8, 8, 8, 8, 7, 7, 7, 7, 7, 9, 4, 4, 4, 4, 4, 5, 5, 5]
```

montre un plateau de quatre 8, de cinq 7, de cinq 4 et trois 5. Donc, une réponse au problème est 7 ou 4. Attention, bien que 4 apparaisse 6 fois, il n'apparait consécutivement que suivant un plateau de longueur 5.

Si tous les nombres sont différents, il n'y a que des plateaux de longueur 1 et le programme peut renvoyer n'importe quel élément de la liste.

On pourra procéder comme suit :

— parcourir la liste avec une boucle `for`

— chaque fois qu'on entre dans un « plateau », on définit un compteur pour ce plateau que l'on incrémente tant qu'on reste sur le plateau

— on surveille si le compteur dépasse la taille maximale déjà atteinte, et si nécessaire, on met à jour et on mémorise le nombre dont la répétition forme le plateau.

Cet exercice provient des demi-finales 2017 du concours Algoréa.

Zéros en fin de liste

On donne une liste d'entiers et on demande d'écrire un code utilisant une boucle `while` pertinente et qui détermine le nombre de zéros situés à la fin de la liste. On considérera un indice `i` initialisé **en fin de liste** et que l'on fera évoluer. Voici un exemple de comportements :

```
[3, 5, 0, 0, 0, 0] → 4
[0, 0, 0] → 3
[10, 10] → 0
[42] → 0
```

Plus long préfixe commun

On donne deux listes `L` et `M` d'entiers et on demande de déterminer la sous-liste `P` qui soit à la fois :

— commune à `L` et `M`,

— commençant au début de chaque liste,

— la plus longue possible.

Il est attendu d'utiliser une boucle `while`. Voici quelques exemples de comportements :

```
L = [0, 5, 8, 6, 4, 7, 2]
M = [0, 5, 8, 9, 4, 7, 2, 4]
P = [0, 5, 8]
----------------
L = [0, 5, 8, 6, 4, 7, 2]
M = [0, 5, 8, 6, 4, 7, 2, 4]
P = [0, 5, 8, 6, 4, 7, 2]
----------------
L = [0, 5, 8, 6, 4, 7, 2]
M = [1, 5, 8, 6, 4, 7, 2, 4]
P = []
----------------
```

Eléments absents d'une liste croissante d'entiers

On donne une liste croissante d'entiers `L`, par exemple

```
L = [12, 13, 13, 18, 19, 19, 25, 26, 30]
```

et on demande d'afficher dans l'ordre croissant les entiers **absents** de la liste `L` et en indiquant leur rang dans leur ordre d'absence. Avec l'exemple ci-dessus, le programme doit afficher :

```
14 absent : rang = 1
15 absent : rang = 2
16 absent : rang = 3
17 absent : rang = 4
```

```
20 absent : rang = 5
21 absent : rang = 6
22 absent : rang = 7
23 absent : rang = 8
24 absent : rang = 9
27 absent : rang = 10
28 absent : rang = 11
29 absent : rang = 12
```

Si la liste contient tous les éléments entre son premier et son dernier élément, il ne devra y avoir aucun affichage.

Cette question est inspirée de l'exercice Leetcode : Missing Element in Sorted Array.

Chapitre V

Fonctions

1 Syntaxe, vocabulaire

Les fonctions en Python

La notion de fonction

Considérons la transformation T suivante : étant donné un nombre x, la transformation T envoie x sur $100x$. On écrirait en mathématiques :

$$T(x) = 100x.$$

Une fonction au sens de Python est souvent assez proche de la notion mathématique de fonction :

— elle reçoit des données,

— elle traite les données,

— elle renvoie une valeur.

En résumé, retenir cette équation :

fonction = données + exécution + retour

Les fonctions en Python apparaissent sous deux formes, décrites ci-dessous. Le première forme a déjà été abordée, pas la deuxième.

Les fonctions built-in

Les fonctions par défaut [1] peuvent être utilisées telles que Python les propose, par exemple la fonction `max` qui donne le maximum d'une séquence de nombres :

```
1 y = max(81, 12)
2 print(y)
```

```
3 81
```

— Ligne 1 : la fonction `max` a juste été appelée sur les données 81 et 12 et elle a renvoyé une valeur (le plus grand des deux nombres, placé dans la variable y)

1. Ou encore les fonctions *natives* ou *intégrées*, en anglais, *built-in functions*.

Les fonctions « custom »

Les fonctions personnalisées (*custom*), construites par le programmeur lui-même, pour les besoins spécifiques de son programme :

```
1 def f(x):
2     return 100*x
3
4 y = f(42)
5 print(y)
```

```
6 4200
```

— Lignes 1-2 : ici, `f` désigne une fonction. Le programmeur a *défini* une fonction `f`, très très simple, en écrivant (ligne 2) le code de ce que `f` va faire (`f` multiplie par 100 la valeur `x` qu'elle reçoit et « renvoie » le résultat). C'est le programmeur qui a écrit le code exécutable de la fonction.

— Ligne 4 : le programmeur *appelle* (ie exécute) la fonction f qu'il a créée avec la donnée 42. La fonction `f` renvoie le résultat qu'elle a calculé.

Différences

La différence essentielle entre les deux types de fonctions :

— une fonction built-in est immédiatement disponible sans que le programmeur ait eu à la définir ou la programmer,

— une fonction custom doit être imaginée, nommée par le programmeur et le code exécutable de la fonction est écrit par le programmeur.

Une fonction typique

Examinons un exemple typique de création de fonction, que l'on va appeler `f` et utilisons cet exemple pour illustrer les mécanismes de *définition* et d'*appel* de fonction qui sont les deux notions-clé à bien comprendre.

On donne deux notes sur 20, disons `a` et `b`, à la fonction `f` et `f` nous renvoie True si la moyenne de ces deux notes est supérieure à 10 et False sinon.

Pour cela, `f` doit calculer la moyenne

$$\mathtt{m} = \frac{\mathtt{a} + \mathtt{b}}{2}$$

et d'examiner si, oui ou non, l'assertion $\mathtt{m} \geq 10$ est vraie ou pas.

Voici un code possible en Python d'une telle fonction :

`fonction_typique.py`

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
```

```
7 print(resultat)
8
9 resultat = f(11, 8)
10 print(resultat)
```

```
11 True
12 False
```

— Lignes 1-4 : le code de la fonction `f`. Ce code est ce qu'on appelle en fait la *définition* de la fonction `f`.

— Ligne 6 : le programme demande à la fonction `f` de calculer s'il est vrai ou pas que le notes 14 et 8 donnent une moyenne supérieure à 10 ou pas. Ce que renvoie la fonction `f` (soit `True`, soit `False` ici) est placé dans la variable `resultat`.

— Ligne 7 : on affiche le booléen `resultat`.

— Ligne 4 : la fonction f renvoie un résultat, ici un booléen, à l'aide du mot-clé `return` qui signifie « retourner » ou, mieux, « renvoyer ».

— Ligne 9-10 et 12 : on réutilise la fonction `f` mais pour des entrées différentes (les notes sont cette fois 11 et 8).

Sémantique d'un appel de fonction

On reprend l'essentiel du code de `fonction_typique.py` ci-dessus :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)
```

```
8 True
```

Que se passe-t-il lors de l'appel `f(14, 8)` à la ligne 6 ? Réponse : les **paramètres** a et b (ligne 1) sont remplacés, dans cet ordre, par les **arguments** 14 et 8 dans la définition de la fonction `f` et le code Python de la définition de `f` (lignes 2 à 4) est exécuté avec les valeurs remplacées.

Autrement dit, une fois l'appel ligne 6 lancé, c'est **comme si** le code suivant était exécuté :

`equiv_fonction_typique.py`

```
1 a = 14
2 b = 8
3
4 m = (a + b)/2
5 ok = (m >= 10)
6
7 resultat = ok
8
9
10 print(resultat)
```

```
11 True
```

— Lignes 1-7 : l'exécution de ce code est équivalente à l'exécution de la fonction `f` lors de l'appel `f(14, 8)`.

Vocabulaire associé à la notion de fonction

On reprend le code de `fonction_typique.py` ci-dessus :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)
```

— Ligne 1 : la première ligne de la définition de la fonction est ce qu'on appelle l'*en-tête* de la fonction. Cet en-tête commence toujours par le mot-clé `def`.

— Ligne 1 : l'en-tête se termine par le symbole « deux-points ».

— Lignes 2-4 : la partie qui suit l'en-tête est ce qu'on appelle le *corps* de la fonction `f`. Ce corps est un bloc indenté.

— Ligne 6 : l'expression `f(14, 8)` est ce qu'on appelle un *appel* de la fonction `f`. Cette expression est placée hors de la définition de fonction et est dite *expression appelante*.

Les valeurs que l'on passe à `f` lors de cet appel (ici 14 et 8) s'appellent les *arguments* de l'appel.

— Ligne 1 : dans la définition de la fonction, ici les variables `a` et `b`, s'appellent les *paramètres* de la fonction `f`.

— Ligne 4 : noter que l'objet qui suit le mot-clé `return` n'est pas entouré de parenthèses. Il est possible d'en placer mais ce n'est pas utile.

Les *paramètres* n'existent que dans la définition de la fonction. Les paramètres sont juste des noms qui servent à désigner les valeurs que la fonction va recevoir **plus tard**, lorsque la fonction sera appelée. Les paramètres ont donc un rôle purement *formel*.

Les arguments ne sont pas des variables, ce sont des valeurs, éventuellement des valeurs de variables. On dit parfois que les arguments sont des paramètres *réels* (autrement dit, non formels).

Le mécanisme d'appel est ce qu'on désigne parfois sous le terme de *passage des arguments* ou encore *transmission des arguments*.

Ordre d'exécution du code

On reprend le code de `fonction_typique.py` ci-dessus :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
```

```
7 print(resultat)
```

— Ligne 1 : l'interpréteur Python commence à cette ligne la lecture du code. En voyant l'en-tête de la fonction, il prend en compte la déclaration de la fonction `f`. Autrement dit à partir de ce moment-là, le nom `f` est reconnu comme celui de la fonction qui est définie à partir de la ligne 1. L'exécution du code continue ligne 6 en ignorant le corps de la fonction.

— Lignes 2-4 : ces lignes ne sont pas exécutées car il n'y a pas eu, pour l'instant, d'appel à la fonction `f`.

— Ligne 6 : l'ordre par défaut d'exécution est modifié puisque l'exécution retourne en arrière dans le code source pour exécuter la fonction `f` des lignes 1 à 4.

— Ligne 7 : une fois l'exécution terminée, l'exécution du code passe de la ligne 4 à la ligne 7.

return et fin d'exécution

Un point essentiel à comprendre concernant le `return` d'une fonction peut être résumé ainsi :

`return = GAME OVER`

autrement dit, une fois l'instruction `return` exécutée, l'appel de la fonction est définitivement interrompu, sans aucune exception.

Illustrons avec le code suivant :

```
1 def f(x,y,z):
2     v= x**2 + y**2 - z**2
3     return v
4
5 t=f(4,3,5)
6 print(t)
```

```
7 0
```

Le code de f s'exécute ligne par ligne. Lorsque l'exécution arrive
à la ligne 3 :

— l'exécution de la fonction f s'interrompt,

— la valeur v obtenue est « renvoyée » et affectée à t.

Tout code figurant dans la définition de f après l'instruction return (ligne 3) serait ignoré lors de l'exécution, par exemple

```
1 def f(x,y,z):
2     v= x**2 + y**2 - z**2
3     return v
4     v= 42
5
6 t=f(4,3,5)
7 print(t)
```

```
8 0
```

— Ligne 4 : cette ligne est comme ignorée et n'est jamais exécutée par `f`.

Exercice type – Fonction aire de disque

Écrire une fonction `aire_disque` qui prend en paramètre un entier `r` et renvoie la surface du disque de rayon r, définie par $S = \pi r^2$. On prendra `3.14` comme valeur approchée de π.

Solution

Le code ne présente pas de difficulté, il s'agit d'implémenter une simple formule. Seule la manière de présenter π peut poser question.

On peut toujours écrire :

```
1 def aire_disque(r):
2     return 3.14 * r**2
3
4 print(aire_disque(10))
```

```
5 314.0
```

Il est plus lisible d'utiliser un nom de variable. Voici deux variantes :

```
1 PI = 3.14
2
3 def aire_disque(r):
4     return PI * r**2
5
6 print(aire_disque(10))
```

```
7 def aire_disque(r):
8     PI = 3.14
9     return PI * r**2
10
11 print(aire_disque(10))
```

Il est d'usage d'écrire des variables qui représentent des constantes en lettres capitales.

Il est aussi possible pour obtenir des calculs plus précis de demander au module `math` de Python de fournir une valeur approchée de π :

```
1 from math import pi
2
3 def aire_disque(r):
4     return pi * r**2
5
6 print(aire_disque(10))
```

```
7 314.1592653589793
```

Exercice type – Implémenter la fonction *signe*

Écrire une fonction `signe` qui évalue le signe d'un paramètre `x` :

$$\texttt{signe}(\texttt{x}) = \begin{cases} 1 & \text{si } \texttt{x} > 0 \\ -1 & \text{si } \texttt{x} < 0 \\ 0 & \text{si } \texttt{x} = 0 \end{cases}$$

Solution

Il faut utiliser des instructions `if/else` :

```
1 def signe(x):
2     if x>0:
3         return 1
4     else:
5         if x<0:
6             return -1
7         else:
8             return 0
9 print(signe(0), signe(4), signe(-7))
```

```
10 0 1 -1
```

Il est possible d'alléger un peu le code en se rappelant qu'une instruction `return` interrompt définitivement l'exécution d'une fonction :

```
1 # Alternative
2
3 def signe(x):
4     if x>0:
5         return 1
6     if x<0:
7         return -1
8     return 0
9
10
11 print(signe(0), signe(4), signe(-7))
```

```
12 0 1 -1
```

— Ligne 6 : un `else` n'est pas utile car l'instruction `return` au-dessus entraîne que si le code accède à la ligne 6, c'est que forcément `x>0` est faux, ce qui correspond à la situation d'un `else`.

— Ligne 8 : pour la même raison, si l'exécution du code arrive à cette ligne, c'est forcément que `x=0`.

Alternative utilisant `if/elif/else`

Si la clause `elif` est connue, il est préférable d'écrire le code ainsi :

```
1 def signe(x):
2     if x>0:
3         return 1
4     elif x<0:
5         return -1
6     else:
7         return 0
8
9 print(signe(0), signe(4), signe(-7))
```

Fonction renvoyant plusieurs valeurs

A proprement parler, une fonction ne peut renvoyer qu'une seule valeur. Cependant, comme une fonction peut renvoyer un conteneur tel qu'une liste, on peut donner l'illusion de renvoyer plusieurs valeurs. Par exemple :

```
1 def f(x, y):
2     return [x + y , x - y]
3
4 L = f(5, 2)
5 print(L)
6 print(L[0]+L[1])
```

```
7 [7, 3]
8 10
```

— Ligne 2 : la fonction f renvoie un seul objet, à savoir une liste

— Lignes 4 et 6 : une liste ayant plusieurs éléments, on peut avoir l'illusion que la fonction `f` a renvoyé deux valeurs.

Fonction sans paramètre

Il est possible définir des fonctions n'admettant pas de paramètre, comme la fonction `f` ci-dessous (ligne 1) :

```
1 def f():
2     return 42
3
4 print(f())
```

```
5 42
```

Pour appeler la fonction `f`, on écrit (ligne 4) juste `f()` autrement dit, on place juste une paire de parenthèses vides.

En pratique, les fonctions sans paramètre sont beaucoup moins fréquentes que leurs analogues avec paramètres. Mais les fonctions sans paramètre servent parfois à regrouper des instructions dans une fonction pour les appeler en une seule fois, ce qui permet d'avoir un code plus structuré.

Un cas typique serait une fonction utilisée pour tester d'autres fonctions :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6
7 def tests():
8     resultat = f(14, 8)
9     print(resultat)
10    resultat = f(0, 20)
11    print(resultat)
12    resultat = f(5, 6)
13    print(resultat)
14
15 tests()
```

```
16 True
17 True
18 False
```

— La fonction `tests` n'utilise aucune donnée et donc n'admet aucun paramètre.

— La fonction `tests` n'admettant aucun paramètre, elle n'admet non plus aucun argument quand elle est appelée.

L'intérêt est qu'il suffit de retirer l'appel à `tests` pour désactiver le code qui est dans la fonction `tests`, ce qui n'était pas faisable facilement si le code était placé hors d'une fonction.

Autre exemple

Soit à coder une fonction `lancer_un_de` simulant le jet d'un dé à 6 faces : la fonction renvoie un entier au hasard entre 1 et 6. Une telle fonction ne dépend de rien et donc sera définie sans paramètre, plus précisément :

```
1 from random import randrange
2
3 def lancer_un_de():
4    return randrange(1,7)
5
6 jet=lancer_un_de()
7 print(jet)
```

```
8 5
```

— Lignes 3-4 : fonction sans paramètre.

— Ligne 6 : appel sans argument de la fonction.

Absence de `return`

Une fonction agit sur des données et peut retourner une valeur explicitement :

avec_return.py

```
1 def f(x):
2   return 100*x
3
4 y=f(42)
5 print(y)
```

```
6 4200
```

Cependant, une fonction ne comporte pas nécessairement une instruction `return`. Voici un exemple :

```
1 def g(x):
2   print(100 * x)
3
4 g(42)
```

```
5 4200
```

— Lignes 1-2 : la fonction g ne contient pas d'instruction `return`. Par contre, elle contient un appel à la fonction `print` qui exprime que l'action de g consiste à afficher quelque chose.

— Ligne 4 : la fonction g est appelée mais `g(42)` ne renvoie rien. Noter qu'il n'y a pas d'appel à la fonction `print` à cette ligne.

— Ligne 5 : cet affichage n'est pas la valeur de g(42) mais le résultat de l'action de l'appel g(42) : un affichage de la valeur de `100 * 42`.

Dans une terminologie aujourd'hui un peu désuète, une fonction qui ne retourne pas de valeur est appelée une *procédure*.

Retour d'une fonction sans instruction `return`

La fonction suivante ne possède aucune instruction `return` explicite :

```
1 def g(x):
2   print(100 * x)
3
4 y = g(42)
5 print(y)
```

```
6 4200
7 None
```

Toutefois, une telle fonction renvoie quelque chose : l'objet `None`. Le code ci-dessus le confirme : la variable y référence la valeur de retour de l'appel à fonction g et comme l'affichage le montre, cette valeur vaut `None`.

`None` est un mot-clé du langage Python et dont la signification dépend de son utilisation. Il est utilisé pour signifier l'absence de résultat, ou le fait que le résultat doit être considéré comme indéterminé.

L'exemple précédent n'a d'intérêt que pour comprendre qu'une fonction sans `return` renvoie néanmoins quelque chose. Toutefois, la plupart du temps, il n'y a aucun sens à afficher le retour

d'une fonction qui ne renvoie rien. Dans le cas de la fonction ci-dessus, une version correcte serait :

```
1 def g(x):
2   print(100 * x)
3
4 g(42)
```

— Ligne 4 : on appelle g mais on n'affiche pas la valeur de retour de g.

Procédure ou pas ?

Si une fonction a pour tâche exclusive d'afficher, on utilisera une procédure autrement dit `f` ne renverra rien et se contentera d'afficher avec la fonction `print`.

Par exemple, si un programme doit déterminer si deux notes sur 20 données conduisent ou pas à une moyenne supérieure à 10, il serait possible d'écrire une fonction comme ceci :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     if ok:
5         print("Moyenne >= 10")
6     else:
7         print("Moyenne < 10")
8
9
10 f(14, 8)
```

```
11 Moyenne >= 10
```

Toutefois, ce type d'usage doit rester rare.

L'inconvénient de la fonction `f` est qu'elle ne renvoie rien, elle se contente d'afficher, donc c'est utile uniquement pour un humain qui lit la zone d'affichage. Le message affiché ne peut être exploité par un autre programme Python alors que, pourtant, la fonction `f` a fait l'essentiel de la tâche, à savoir :

— le calcul de la moyenne (ligne 2)

— sa comparaison avec la note de 10 (ligne 3).

Pourtant, ces calculs demeurent inaccessibles à l'*extérieur* de la fonction, car la fonction ne renvoie rien.

Il est possible de remédier à cette difficulté en créant *deux* fonctions :

— une fonction `f` qui regarde si, oui ou non, la moyenne est supérieure à 10 et *renvoie* cette comparaison ;

— une procédure qui se contente d'afficher de façon interprétable pour un humain, le résultat renvoyé par `f`.

D'où le programme préférable :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 def afficher(plus_que_10):
7     if plus_que_10:
8         print("Moyenne >= 10")
9     else:
10        print("Moyenne < 10")
11
12 resultat = f(14, 8)
13 afficher(resultat)
```

```
14 Moyenne >= 10
```

— Ligne 1 : fonction qui renvoie le résultat de la comparaison avec 10 de la moyenne des notes a et b.

— Ligne 6 : fonction qui affiche un message concernant un résultat (placé dans la variable plus_que_10).

Notion d'effet de bord

Soit la fonction suivante :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)
```

```
8 True
```

Cette fonction agit de la manière suivante : elle reçoit des arguments et elle renvoie une valeur. Son action est uniquement déterminée par ce qu'elle renvoie. Toutefois, certaines fonctions ont une action qui ne se limite pas à renvoyer une valeur. L'exemple le plus simple est le cas d'un affichage :

```
1 def f(x):
2     y=10*x
3     print(y+1)
4     return y
5
6 z=f(42)
7 print(z)
```

```
8 421
9 420
```

La fonction ci-dessus renvoie un objet mais accessoirement a une autre action : l'affichage d'une certaine valeur. Cet action qui n'est pas l'action de renvoyer un objet est appelé un *effet de bord*. En pratique, un effet de bord peut prendre des formes très variées comme

— un affichage
— la production d'un dessin avec matplotlib ou tout autre bibliothèque de dessin,
— la création d'un fichier
— la modification d'objets « extérieurs » à la fonction.

Instructions `return` multiples

La définition d'une fonction peut contenir des instructions `return` en plusieurs endroits.

Par exemple, soit la fonction `f` qui prend un entier `x` en paramètre et renvoie $x/2$ (la moitié de `x`) si `x` est pair et renvoie $3x + 1$ si `x` est impair. Par exemple $f(5) = 16$ ou encore $f(20) = 10$.

On peut coder cette fonction de la manière suivante :

```
1 def f(x):
2     if x%2 ==0:
3         return x//2
4     else:
5         return 3*x+1
6
7 print(f(5))
8 print(f(20))
```

```
9 16
10 10
```

La définition de cette fonction utilise deux instructions `return` : la fin de l'exécution de la fonction se produira, selon les cas, suivant la première instruction `return` (ligne 3) ou suivant la seconde instruction `return` (ligne 5). Bien sûr, un appel de fonction n'exécute jamais *deux* instructions `return` l'une à la suite de l'autre puisque toute instruction `return` achève définitivement l'exécution d'une fonction.

La programmation « structurée »

Une certaine conception de la programmation, dite « programmation structurée » recommande de ne pas placer plusieurs instructions `return` dans une fonction parce que cela rendrait plus difficile la compréhension de l'exécution de la fonction puisqu'elle admet plusieurs « points de sortie » [2]. Le code conforme aux principes de la programmation structurée serait plutôt le suivant :

```
1 def f(x):
2     if x%2 ==0:
3         y = x//2
4     else:
5         y = 3*x+1
6     return y
7
```

2. La validité de ce principe est controversée.

```
8 print(f(5))
9 print(f(20))
```

— La définition de `f` ne comprend qu'un seul `return` : la fonction ne peut terminer son exécution avant d'atteindre la ligne 6.

EXERCICES

Fonction de comparaison de signe

Ecrire une fonction `meme_signe(a, b)` qui renvoie `True` si les nombres `a` et `b` sont de même signe, et `False` dans le cas inverse. On considérera que `0` est aussi bien positif que négatif. Exemples de comportements :

```
42 17 -> True
42 0 -> True
-42 17 -> False
42 -17 -> False
-42 -17 -> True
-42 0 -> True
0 -42 -> True
```

Fonctions Celsius <-> Fahrenheit

1. La formule de conversion de degrés Fahrenheit vers des degrés Celsius est $c = \frac{5}{9}(f - 32)$.
   a. Écrire une fonction `f2c(f)` qui prend en paramètre une température f (en degrés Fahrenheit) et qui renvoie sa conversion c (en degrés Celsius).
   b. Combien font $451°F$ et $0°F$ en $°C$?
2. La formule de conversion de degrés Celsius vers des degrés Fahrenheit est $f = \frac{9}{5}c + 32$.
   a. Écrire une fonction `c2f(c)` qui prend en paramètre une température c (en degrés Celsius) et qui renvoie sa conversion f (en degrés Fahrenheit)
   b. Combien font $42°C$ et $-273,15°C$ en $°F$?
3. Pour trois valeurs `x` que vous choisirez librement, vérifiez que `c2f(f2c(x))` et `f2c(c2f(x))` sont proches de `x`.

Conversions de devises

1. Au taux de change, 1 dollar vaut 0,83 euros. Ecrire une fonction `dollarVersEuro` de conversion de dollar en euro.
2. Au taux de change, 1 euro vaut 91 roubles. Ecrire une fonction `euroVersRouble` de conversion d'euro en rouble.
3. Ecrire une fonction `dollarVersrouble` de conversion de dollar en rouble utilisant les deux fonctions précédentes.

Nombre d'entiers positifs

Dans cet exercice, on dira qu'un entier N est *positif* si $N \geq 0$.

Ecrire un code qui partant de deux entiers `a` et `b` avec `a` $\leq$ `b` détermine le nombre d'entiers positifs entre `a` et `b` (les bornes `a` et `b` étant incluses). Voici quelques exemples de valeurs de selon les valeurs de `a` et `b` :

```
3 10 -> 8
3 3 -> 1
0 5 -> 6
0 0 -> 1
-5 -2 -> 0
-5 0 -> 1
-5 3 -> 4
-10000000000 10000000000 -> 10000000001
```

Explication pour `a=-5` et `b=3` : les entiers positifs entre `-5` et `3` sont : `0`, `1`, `2` et `3`, il y en a donc 4.

On écrira une fonction `nbPositifs(a, b)`.

Votre code doit être capable de traiter instantanément des entiers de l'ordre de plusieurs milliards.

Fonction pour tester une année bissextile

Écrire une fonction `estBissextile` qui prend en paramètre un entier `a` et retourne `True` si l'année `a` est bissextile (`False` sinon). Pour rappel, une année a est bissextile si le booléen suivant

```
(a%4==0 and a%100!=0) or a%400==0
```

vaut `True`.

Date valide

On donne une liste `[jour, mois]` de deux entiers strictement positifs et on demande d'écrire une fonction `estDateValide(j, m)` qui dise si la date `jour/mois` définit une date valide du calendrier. La fonction doit renvoyer un booléen, par exemple :

Liste	Valide	Explication
`[18, 3]`	True	18 mars
`[31, 12]`	True	31 décembre
`[35, 3]`	False	un mois n'a pas 35 jours
`[31, 11]`	False	pas de 31 novembre
`[31, 13]`	False	pas de 13^{e} mois

On supposera que le mois de février a 28 jours.

On pourra créer une liste `JOURS_MOIS` telle que `JOURS_MOIS[m]` est le nombre de jours du `m`-ème mois (supposé entre 1 et 12).

Autre possibilité : utiliser une condition testant si le mois est en 31, 30 ou 28 jours.

Calculer son âge

Ecrire une fonction `ecart(naissance, aujourdhui)` qui partant de la date de naissance de Mady et de la date d'aujourd'hui détermine l'âge de Mady. Une date est donnée par une liste de la forme `[j, m, a]` où `j` désigne le jour, `m` le mois et `a` l'année. La liste donnant la date de naissance sera appelée `naissance` et la date du jour sera nommée `aujourdhui`.

Par exemple, supposons que `naissance = [14, 11, 2005]`. Si `aujourdhui = [5, 1, 2025]` alors `age = 19`.

Implémenter la fonction `max`

Écrire une fonction `maxi` qui renvoie le maximum de deux nombres `a` et `b` passés en paramètre.

Implémenter la fonction *valeur absolue*

Écrire une fonction `val_abs` qui retourne la valeur absolue d'un nombre `x` passé en paramètre.

Fonction copysign

Ecrire le code d'une fonction `copy_sign(x, y)` qui prend en paramètres deux nombres réels et renvoie un nombre qui est positif si $y \geq 0$ et dont la valeur absolue est celle de `x` et l'opposé de la valeur absolue si $y < 0$. Cette fonction est presque identique à la fonction `copysign` du module `math` de Python. Voici, sur quelques exemples, le comportement attendu de `copy_sign` :

```
(10, 42) → 10
(10, -42) → -10.0
(-10, 42) → 10.0
(-10, -42) → -10.0
(10, 0) → 10.0
(-10, 0) → 10.0
(0, 42) → 0.0
(0, -42) → 0.0
```

Distance entre deux nombres

Ecrire une fonction `distance(a, b)` qui calcule la distance entre deux nombres `a` et `b`. Par exemple, `distance(3, 5)` vaudra `distance(5, 3)` qui vaudra 2.

Fonction ensureRange

On donne trois nombres `a`, `b` et `x`. On demande d'écrire une fonction `ensureRange(a, b, x)` valant l'élément le plus proche de `x` situé dans l'intervalle d'extrémités `a` et `b`. Voici quelques exemples de comportements :

```
0, 5, 2.42 -> 2.42
5, 0, 2.42 -> 2.42
0, 5, 5 -> 5
5, 0, 5 -> 5
0, 5, 6.5 -> 5
5, 0, 6.5 -> 5
0, 5, -1 -> 0
5, 0, -1 -> 0
0, 0, 5 -> 0
0, 0, -1 -> 0
0, 0, 0 -> 0
```

Cette fonction est implémentée sous différents langages par une fonction nommée `constrain` (p5js) ou `ensureRange` (Pascal, Delphi) ou parfois `valBetween` en Javascript.

Le plus proche et entre -1 et 1

Ecrire une fonction `plusProche(x)` qui à partir d'un nombre `x` renvoie `x` si $-1 \leq x \leq 1$, renvoie -1 si $x < -1$ et renvoie 1 si $x > 1$. Autrement dit, le nombre renvoyé est le nombre de

l'intervalle $[-1, 1]$ le plus proche de `x`.

Le plus proche parmi deux nombres

Ecrire une fonction `plus_proche(a, b, x)` qui renvoie parmi les nombres `a` ou `b` celui dont `x` est le plus proche. Si les deux nombres conviennent, la fonction renverra le plus petit des deux.

Exemples d'exécution de `plus_proche(a, b, x)` :

```
plusproche(40, 45, 42) = 40
plusproche(40, 43, 42) = 43
plusproche(40, 44, 42) = 40
plusproche(42, 42, 42) = 42
plusproche(38, 40, 42) = 40
plusproche(45, 49, 42) = 45
```

Inclusion d'intervalles

Dans cet exercice, l'intervalle des entiers `x` compris, au sens large, entre deux entiers `a` et `b` sera représenté par la liste `[a, b]`. Par exemple, la liste `L=[14, 18]` représente les entiers 14, 15, 16, 17 et 18.

Ecrire une fonction `inclus` qui accepte comme paramètres deux intervalles `A` et `B` représentés chacun par une liste de 2 entiers et qui retourne `True` si l'intervalle `A` est inclus dans l'intervalle `B`, et `False` sinon. Exemples de comportement de la fonction :

```
[2, 3] inclus dans [1, 5] ? True
[0, 3] inclus dans [1, 5] ? False
[0, 6] inclus dans [1, 5] ? False
```

Segment découpé en parties de même longueur

Cet exercice n'est pas un exercice de dessin.

Ecrire une fonction `decouper(a, b, n)` qui affiche les valeurs où s'effectuent les découpages en n segments de même longueur d'un segment donné d'extrémités deux entiers a et b avec $a \leq b$. Par exemple, `decouper(10,12,8)` doit afficher :

```
10
10.25
10.5
10.75
11.0
11.25
11.5
11.75
12.0
```

On observera qu'il y a $n + 1$ extrémités à donner et on pourra introduire une variable h pour la longueur de chaque segment.

Équation du second degré (version fonction)

On rappelle ci-dessous un code de résolution et d'affichage des solutions d'une équation du second degré $ax^2 + bx + c = 0$ mais n'utilisant ni fonction ni liste :

```
a = 6
b = -5
c = 1
delta = b*b -4*a*c
if delta >0:
    print("2 solutions distinctes :")
    print("x1 =", (-b - delta**0.5)/(2.*a))
    print("x2 =", (-b + delta**0.5)/(2.*a))
elif delta == 0:
    print("une seule solution")
    print("x =", (-b)/(2.*a))
else:
    print("Aucune solution")
```

Vous devez adapter ce code pour les questions ci-dessous.

1. Ecrire une fonction `resoudre_2nd_degre(a, b, c)` qui résout dans l'ensemble des nombres réels l'équation du second degré. La fonction prend en paramètre `a`, `b`, `c` les coefficients supposés entiers de l'équation $ax^2 + bx + c = 0$ et renvoie une **liste** de solutions, éventuellement vide.
2. Ecrire une fonction d'affichage `afficher_solutions(a, b, c)` des solutions. La fonction affichage doit utiliser la fonction précédente.
   Voici un exemple de comportements du code à écrire :

```
a = 6 b = -5 c = 1 :

2 solutions distinctes :
x1 = 0.3333333333333333
x2 = 0.5
-----------------------------------------------
a = 4 b = -12 c = 9 :

une seule solution
x = 1.5
-----------------------------------------------
a = 6 b = 7 c = 7 :

Aucune solution
-----------------------------------------------
```

Fonction de calcul du volume du ballon de rugby

Consulter l'article de wikipedia donnant le volume d'un ellipsoïde de révolution. Ne pas hésiter à faire un petit croquis en marquant le nom des variables qui interviennent dans la formule.

Ecrire une fonction `volumeRugby(H, L)` qui détermine le volume d'un ballon de rugby de hauteur `H` et de largeur `L`. On écrira explicitement la formule utilisée par Wikipedia, avec ses variables `p` et `q` et après avoir fait le lien avec les variables `H` et `L`.

On importera `pi` avec le module math.

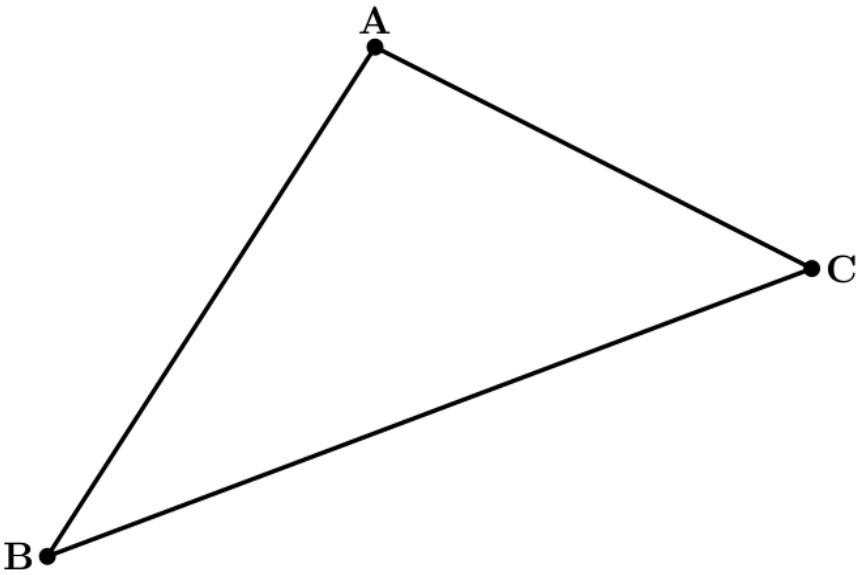
Appliquer à un ballon standard de tailles $H = 29$cm et $L = 19$cm, de volume environ 5,5 litres (rappel : 1 litre = 1dm^3 donc convertir les mesures en décimètres avant appel).

Volume d'un tétraèdre (version fonctions)

1. Ecrire une fonction `volume(AB, BC, CA, DA, DB, DC)` qui renvoie le volume du tétraèdre ABCD en fonction des longueurs de ses six côtés. On utilisera pour cela une des formules données par Wikipedia, celle dont le dénominateur est 12.
   Appliquer au tétraèdre ABCD de dimensions : AB=8, BC=5, CA=6, DA=4, DB=7 et DC=3. On trouvera un volume de `5.59` à 1/1000 près.
2. La formule de Héron donne l'aire d'un triangle lorsqu'on connaît la longueur de ses 3 côtés. Ecrire une fonction qui donne l'aire du triangle en fonction des ses côtés. On utilisera la variable auxiliaire accompagnant la formule.
3. Le rayon de la sphère inscrite à un tétraèdre est donné par la formule $r = 3V/\mathcal{A}$ où V est le volume du tétraèdre et $\mathcal{A}$ la somme des aires des 4 faces. Appliquer au tétraèdre ABCD de dimensions : AB=8, BC=5, CA=6, DA=4, DB=7 et DC=3. On trouvera `r` $=$ `0.411` à 1/1000 près. On ne demande pas d'écrire une fonction.
   Cette question est une réponse au problème SPOJ : Sphere in a tetrahedron

Triangle direct

On dit qu'un triangle `ABC` est *direct* ou encore *orienté dans le sens direct* si lorsqu'on parcourt les sommets dans l'ordre `A`, puis `B` suivi de `C` et enfin `A`, on a tourné dans le sens **inverse** des aiguilles d'une montre. Par exemple, le triangle `ABC` ci-dessous est un triangle direct :



On rappele que si `M` est le point de coordonnées $(\mathtt{x_M}, \mathtt{y_M})$ et si `N` est le point de coordonnées $(\mathtt{x_N}, \mathtt{y_N})$ alors les coordonnées du vecteur $\overrightarrow{\mathtt{MN}}$ sont :

$$(\mathtt{x_N} - \mathtt{x_M}, \mathtt{y_N} - \mathtt{y_M})$$

On donne trois points `A`, `B` et `C` et on demande d'écrire une fonction `estDirect(A, B, C)` qui dise si oui ou non, le triangle `ABC` est un triangle direct. Pour cela,

— on calcule les coordonnées $(\mathtt{x_1}, \mathtt{y_1})$ du vecteur $\overrightarrow{CA}$

— on calcule les coordonnées (x_2, y_2) du vecteur $\overrightarrow{CB}$
— on détermine le signe de $d = x_1y_2 - x_2y_1$:
  — si $d > 0$ le triangle ABC est direct ;
  — si $d < 0$ le triangle ABC n'est pas direct.

Plus précisément, on fournit les coordonnées des trois sommets A, B et C pour chacun sous la forme d'une liste de deux nombres. L'appel `estDirect(A, B, C)` vaut `True` si le triangle ABC est direct et `False` sinon.
Appliquer au triangle ABC où

— `A = [60, 70]`
— `B = [30, 24]`
— `C = [100, 50]`

Écart-type d'une liste

Si on dispose d'une liste de n nombres x_1, x_2, etc, x_n, on définit l' *écart type* de la liste comme le nombre suivant :

$$\sqrt{\frac{(x_1 - m)^2 + \cdots + (x_n - m)^2}{n}}$$

où $m = \dfrac{x_1 + \cdots + x_n}{n}$ est la moyenne de la liste.
Pour tester les questions, on pourra utiliser la liste L ci-dessous

```
L = [17, 11, 19, 13, 7, 7, 14, 0, 9, 16]
```

1. Ecrire une fonction `moy(L)` qui calcule la moyenne des nombres de la liste L. Dans le cas de la liste donnée en exemple, on trouvera une moyenne de 11,3
2. Ecrire une fonction `decal(L, v)` qui renvoie la liste dont les éléments sont les $(z - v)^2$ où z parcourt les éléments de L.
3. En utilisant les questions précédentes, écrire une fonction `sigma(L)` qui renvoie l'écart-type de L. Dans le cas de la liste donnée en exemple, on trouvera un écart-type valant environ 5,42

Somme des inverses des entiers

1. Ecrire une fonction `harmonique(n)` qui calcule la somme de tous les inverses des entiers entre 1 et n, autrement dit, la somme $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$.
   Voici quelques exemples du comportement de la fonction :

```
1 -> 1.0
2 -> 1.5
3 -> 1.8333333333333333
10 -> 2.9289682539682538
2015 -> 8.18583825771582
```

2. Ecrire une fonction `depasse(n, v)` qui étant donné un entier n et un nombre entier v renvoie `True` si

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq v$$

et `False` sinon. La fonction `depasse` devra utiliser la fonction `harmonique`. Voici quelques exemples du comportement de la fonction :

```
10, 2 -> True
10, 5 -> False
100, 5 -> True
1000000, 13 -> True
```

Nombre d'arrangements

Soient n et p deux entiers tels que $0 \leq p \leq n$. On définit l'entier noté $A(n, p)$ représentant le nombre d'arrangements de p objets pris parmi n : c'est le produit des p entiers consécutifs en décroissant à partir de n. Par exemple,

$$A(7, 3) = 7 \times 6 \times 5 = 210$$

Ecrire une fonction `arrang(n, p)` qui renvoie le nombre d'arrangements de p objets pris parmi n.

Nombre de chiffres

(1) Écrire une fonction `nombreChiffres(n)` qui étant donné un entier n entre 0 et 999 renvoie le nombre de chiffres de n. Voici quelques exemples de comportement de la fonction :

```
0 -> 1
42 -> 2
100 -> 3
421 -> 3
```

(2) Ecrire une fonction `afficherNombreChiffres(n)` qui affiche le nombre de chiffres d'un entier n supposé être entre 0 et 999. La fonction devra utiliser la fonction `nombreChiffres`. Voici quelques exemples de comportement de la fonction :

n	afficherNombreChiffres(n) affiche
0	Le nombre de chiffres de 0 est 1
42	Le nombre de chiffres de 42 est 2
100	Le nombre de chiffres de 100 est 3
421	Le nombre de chiffres de 421 est 3

L'affichage devra être strictement conforme à l'affichage indiqué dans le tableau ci-dessus.

Nombre de chiffres de $n!$

(*Exercice pour matheux*).

Le module standard math possède

- une fonction factorielle (`factorial`)
- une fonction partie entière (`floor`)

— une fonction logarithme en base 10 (`log10`).

Voici des exemple d'utilisation (exécutez-le) :

```
from math import *

print(factorial(42), end ='\n\n')
print(floor(3.14), end ='\n\n')
print(log10(10000), end ='\n\n')
print(log10(2016), end ='\n\n')
print(10**(log10(2016)))
```

On rappelle que si $N > 0$ est un entier, le nombre de chiffres de N vaut $\lfloor \log_{10} N + 1 \rfloor$ où $\lfloor x \rfloor$ désigne la partie entière de x et où $\log_{10}$ désigne le logarithme en base 10. Par exemple,

$$42! = 1405006117752879898543142606244511569936384000000000$$

et qui a 52 chiffres comme le code Python suivant nous le confirme (exécutez ce code) :

```
from math import *

x=factorial(42)
print(floor(log10(x)+1))
```

Ecrire une fonction `nb_chiffres_factoriel` qui calcule le nombre de chiffres de $n!$. Votre code devra savoir répondre pour de grandes valeurs de n comme 1 million ou 1 milliard voire encore bien plus grand.

Moyenne harmonique

Etant donné n nombres non nuls $a_1, a_2, \ldots, a_n$, on définit la moyenne **harmonique** h de ces n nombres comme le nombre suivant :

$$h = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_n}}$$

Par exemple, si les entiers a_i sont ceux de la liste `[6, 2, 7, 3]` alors leur moyenne harmonique est `3,5`.

La moyenne **arithmétique** de $a_1, a_2, \ldots, a_n$ est la moyenne habituelle, c'est-à-dire :

$$\frac{a_1 + a_2 + \cdots + a_n}{n}$$

Par exemple, si les entiers a_i sont ceux de la liste `[6, 2, 7, 3]` alors leur moyenne arithmétique est `4,5`.

[Ainsi, la moyenne **harmonique** de n nombres est l'inverse de la moyenne arithmétique des inverses de ces nombres.]

1. Ecrire une fonction `moyenne(L)` qui renvoie la moyenne arithmétique d'une liste de nombres.
2. Ecrire une fonction `harmonique(L)` qui renvoie la moyenne harmonique d'une liste `L` de nombres. Selon votre préférence, vous pouvez ou non, utiliser la fonction `moyenne` de la question précédente.

Somme supérieure à ...

1. Ecrire une fonction `s(n)` qui renvoie la somme $S = 1^2 + 2^2 + \cdots + n^2$. Ainsi, on trouvera que $s(17) = 1785$ ou encore que $s(18) = 2109$
2. En utilisant la fonction `s`, construire une fonction `f(N)` où $N > 0$ est un entier et qui renvoie le plus petit entier n tel que la somme $S = 1^2 + 2^2 + \cdots + n^2$ vérifie $S \geq N$. Voici deux exemples de comportement de la fonction `f` :

```
385 -> 10
2015 -> 18
```

3. Le code précédent appliqué à rechercher le plus petit entier n tel que la somme $S = 1^2 + 2^2 + \cdots + n^2$ vérifie $S \geq 123456789101112$ sera beaucoup trop lent à donner une réponse. Construire une fonction `g(N)` où $N > 0$ est un entier, qui n'utilise pas la fonction `s` et qui renvoie le plus petit entier n tel que la somme $S = 1^2 + 2^2 + \cdots + n^2$ vérifie $S \geq N$. Pour `N = 123456789101112`, on trouvera que n vaut 71814.

Somme multiple de ...

1. Ecrire une fonction `s(n)` qui renvoie la somme $S = 1^2 + 2^2 + \cdots + n^2$. Ainsi, on trouvera que $s(24) = 4900$ ou encore que $s(624) = 81185000$
2. En utilisant la fonction `s`, construire une fonction `f(d)` où $d > 0$ est un entier et qui renvoie le plus petit entier $n > 0$ tel que la somme $S = 1^2 + 2^2 + \cdots + n^2$ soit multiple de d. Voici deux exemples de comportement de la fonction `f` :

```
100 -> 24
1000 -> 624
```

3. Le code précédent appliqué à rechercher le plus petit entier n tel que la somme $S = 1^2 + 2^2 + \cdots + n^2$ soit multiple de 10^{10} sera beaucoup trop lent à donner une réponse. Construire une fonction `g(d)` où $d > 0$ est un entier, qui n'utilise pas la fonction `s` et qui renvoie le plus petit entier $n > 0$ tel que la somme $S = 1^2 + 2^2 + \cdots + n^2$ soit multiple de d. Pour $d = 10^8$, on trouvera que n vaut 12890624.

Compter les entiers négatifs

Ecrire une fonction `compter_neg` qui prend en paramètre une liste d'entiers `L` et retourne le nombre de valeurs strictement négatives dans la liste L. On utilisera une variable nommée `cpt`.

Si vous connaissez les listes en compréhension, essayez de les utilisez. Votre code devrait tenir en une ligne et être aussi rapide (voire largement plus) sur de grosses listes que le code standard.

Si vous connaissez les listes en compréhension, essayez de les utilisez. Votre code devrait tenir en une ligne et être aussi rapide (voire largement plus) sur de grosses listes que le code standard.

Aucun multiple de 10

Écrire une fonction `aucunMultiple10` qui prend en paramètre une liste d'entiers et qui renvoie True si la liste ne contient aucun multiple de 10. Voici deux exemples de comportement de la fonction :

```
[42, 81, 33, 81] -> True
[42, 91, 75, 90, 33] -> False
```

Valeurs non nulles d'une liste

Ecrire une fonction `f` qui prend en paramètre une liste `L` d'entiers et renvoie une nouvelle liste dont le contenu est formé des valeurs de `L` non nulles. Par exemple :

— si `L` est `[5, 0, -3, 5, 0]` alors `f(L)` est la liste `[5, -3, 5]`

— si `L` est `[5, 2, 1]` alors `f(L)` est la liste `[5, 2, 1]`

— si `L` est `[0, 0, 0]` alors `f(L)` est la liste vide.

Positivons ...

Ecrire une fonction `suppr_neg` qui prend en paramètre une liste d'entiers `L` et retourne une (nouvelle) liste `M` déterminée à partir de `L` en en ignorant tous les éléments strictement négatifs. Par exemple :

```
[-6, 42, 5, -3, 0, 2, 0, -1] -> [42, 5, 0, 2, 0]
```

Si vous connaissez les listes en compréhension, le code devra les utiliser.

Remplacer les nombres négatifs

Pour cet exercice, on pourra consulter la partie du cours qui explique comment une fonction peut modifier un des arguments qu'il reçoit.

Ecrire une fonction `rempl_neg` qui prend en paramètre une liste d'entiers L, ne retourne rien mais remplace dans la liste L chaque entier négatif en son opposé, par exemple :

```
[-6, 2, 5, -3, 0, 2, 0, -1] -> [6, 2, 5, 3, 0, 2, 0, 1]
```

Ainsi, la fonction `rempl_neg` **modifie** la liste qu'elle reçoit en argument.

Liste des multiples de 2, 3 ou 5

Écrire une fonction `f` qui prend en paramètre un entier `n` ≥ 0 et qui renvoie la liste de tous les entiers `k` tels que $0 \leq$ `k` $\leq$ `n` et `k` est multiple de 2, 3 ou 5. Par exemple,

`f(15)` est la liste $[0, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 15]$.

Nombre d'éléments entre deux valeurs

Soit une liste `L` d'entiers et soit z un entier. Ecrire une fonction `nbOccur(L, z)` qui renvoie le nombre d'éléments situées **strictement** entre le premier élément de `L` valant z et le dernier élément de `L` valant z. Si z n'est pas présent ou est présent seulement une fois dans la liste `L`, la fonction renverra `-1`.

Ci-dessous, pour $z = 42$, quelques exemples du comportement attendu :

```
[42, 12, 81, 12, 42, 37, 42, 56] 42 -> 5
[42, 5, 42] 42 -> 1
[5, 42, 42, 5] 42 -> 0
[42, 42] 42 -> 0
[42, 42, 42, 42] 42 -> 2
[12, 81, 12, 42, 37] 42 -> -1
[12, 81, 12, 37] 42 -> -1
```

Liste d'entiers aléatoires

Ecrire une fonction `listeAleatoire(N, a, b)` qui renvoie une liste de `N` entiers aléatoires entre `a` et `b` au sens large. Voici quelques exemples de comportements

N	a	b	Liste aléatoire
5	10	50	`[45, 27, 43, 36, 43]`
5	1	9	`[9, 9, 6, 7, 4]`
6	0	1	`[0, 1, 0, 0, 0, 1]`
3	42	42	`[42, 42, 42]`
5	100	42	`[]`

Sur certaines listes de trois entiers

On s'intéresse aux listes `L` de 3 entiers vérifiant les deux propriétés suivantes :

— la liste `L` contient exactement une fois l'entier `0`,

— les deux entiers non nuls de `L` ont même valeur.

Le tableau ci-dessous donne différents exemples de listes qui, selon les cas, vérifient ou, au contraire, ne vérifient pas, les propriétés ci-dessus :

Liste	Vérifie les propriétés	Commentaire
$[0, 4, 4]$	oui	
$[5, 0, 5]$	oui	
$[0, 0, 4]$	non	i) n'est pas vérifiée
$[8, 9, 4]$	non	i) n'est pas vérifiée
$[4, 2, 0]$	non	ii) n'est pas vérifiée

Vous devez écrire une fonction `f` qui prend en paramètre une liste `L` que l'on supposera contenir exactement trois entiers et qui

— renvoie `True` si `L` vérifie les propriétés i) et ii),

— renvoie `False` sinon.

Extraction de valeurs aux indices impairs

On donne une liste `L` d'entiers et on demande de construire la liste des éléments de `L` qui sont non nuls et placés à des indices impairs. Par exemple, si la liste `L` est

```
L = [16, 10, 15, 0, 0, 12, 0, 19, 18, 0]
```

alors la liste demandée est `[10, 12, 19]`.

On écrira une fonction `indices(L)`.

Derniers éléments d'une liste

On donne une liste L d'entiers et un entier N et on demande d'écrire une fonction `derniersElement(L, N)` qui affiche les N derniers éléments de la liste L. Si N excède la longueur de la liste, on affiche tous les éléments de la liste. Si N=0, aucun élément n'est affiché. L'affichage ne doit faire apparaître ni crochets ni virgules. Voici quelques exemples de comportements avec la liste L=[42, 81, 99, 31, 12] et les entiers N de 0 à 6 :

```
0->
1-> 12
2-> 31 12
3-> 99 31 12
4-> 81 99 31 12
5-> 42 81 99 31 12
6-> 42 81 99 31 12
```

Derniers termes d'une liste

On donne un entier $p \geq 0$ et une liste L d'entiers et on appelle M la nouvelle liste formée des p derniers éléments de L. Ecrire une fonction `derniersElements(p, L)` qui renvoie M. On supposera que $0 \leq p \leq n$ où n est le nombre d'éléments de L.

Voici quelques exemples de comportement de la fonction si L = [12, 81, 31, 65, 9, 32, 82] :

```
p = 4 -> [65, 9, 32, 82]
p = 1 -> [82]
p = 7 -> [12, 81, 31, 65, 9, 32, 82]
p = 0 -> []
```

Le code devra parcourir uniquement les éléments utiles de la liste L. Si vous connaissez les listes en compréhension, vous devez les utiliser.

Indice de la dernière occurrence

Ecrire une fonction `dernier(L, v)` qui, partant d'une liste L d'entiers et d'un entier v, renvoie :

— l'indice du dernier élément de L qui vaut v si un tel élément existe

— la longueur de la liste L sinon.

Exemples de comportement :

```
L = [20, 30, 50, 20, 50, 30, 40], v = 50 : 4
L = [40, 40, 40, 20, 30], v = 50 : 5
L = [10, 20, 30, 30], v = 10 : 0
L = [50, 40], v = 50 : 0
L = [20], v = 30 : 1
```

On pourra envisager deux méthodes, l'une qui parcourt la liste dans le sens des indices croissants, une autre qui le fait en commençant par la fin.

Cet exercice a été proposé sur le forum Python d'OpenClassrooms.

Valeur absolue des éléments d'une liste

Ecrire une fonction `abs_liste` qui prend en paramètre une liste d'entiers L et retourne une nouvelle liste formée des valeurs absolues des éléments de L. Exemples :

```
[-5, 4, 2, 0, -1, -2, 3] -> [5, 4, 2, 0, 1, 2, 3]
[42] -> [42]
[0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0]
[-42, -42, -42, -42, -42] -> [42, 42, 42, 42, 42]
```

Carrés d'une liste

Ecrire une fonction `carre_liste` qui prend en paramètre une liste d'entiers `L` et retourne une nouvelle liste formée des carrés des éléments de `L`. La liste initiale `L` ne doit pas avoir été modifiée après un appel `carre_liste(L)` .

Exemples de comportement de la fonction :

```
[-5, 4, 2, 0, -1, -2, 3] -> [25, 16, 4, 0, 1, 4, 9]
[10] -> [100]
```

Minimum et maximum simultanés

Ecrire une fonction `min_max` qui prend en paramètre une liste `L` et retourne, sous forme de liste de deux éléments, la plus petite et la plus grande valeur de `L`.

Exemples :

```
[12, 81, 65, 9, 32] -> [9, 81]
[42] -> [42, 42]
[42, 42, 42, 42, 42] -> [42, 42]
```

Il est préférable que votre algorithme parcoure la liste une fois et une seule.

Extraire un point dans une liste

On donne une liste d'entiers `L` de longueur paire `2n`. Par exemple, si `n` $= 5$, on pourrait disposer de la liste suivante :

$$\texttt{L} = [0, 0, 4, 0, 4, 7, 0, 4, -3, 4]$$

La liste `L` représente `n` points du plan :

- le premier point a pour abscisse la première valeur de `L`
- le premier point a pour ordonnée la deuxième valeur de `L`
- le deuxième point a pour abscisse la troisième valeur de `L`
- le deuxième point a pour ordonnée la quatrième valeur de `L`
- et ainsi de suite.

Dans ce qui suit, un point `M` d'abscisse `x` et d'ordonnée `y` sera notée à l'aide de la liste correspondante :

```
M = [x, y]
```

La liste `L` donnée en exemple ci-dessus représente donc les 5 points de coordonnées suivants :

$$[0, 0], [4, 0], [4, 7], [0, 4] \text{ et } [-3, 4].$$

Ecrire une fonction `nroVersPoint(L, r)` qui renvoie le point de la liste `L` qui est à la `r`-ème place. Avec la liste `L` donnée en exemple ci-dessus, `nroVersPoint(L, 3)` renvoie $[4, 7]$.

Réfléchir une liste

Écrire une fonction `reflechir` qui prend en paramètre une liste `L` et qui construit une nouvelle liste constituée, dans l'ordre, des éléments de `L` puis suivis des éléments de `L`, mais placés en commençant par la fin. Voici un exemple de comportement de la fonction :

```
[1000, 81, 12] -> [1000, 81, 12, 12, 81, 1000]
```

Attention, la liste initiale doit rester inchangée.

Nombres entre 1 et 99

(1) Ecrire une fonction `chiffres` qui reçoit en paramètre un entier n tel que $0 \leq n \leq 99$ et qui renvoie la liste de ses chiffres, au sens usuel (le premier chiffre n'est pas 0 sauf pour 0). On pourra utiliser une division par 10. L'usage de la méthode `append` sera pénalisé. Voici quelques exemples de comportement de la fonction `chiffres` :

```
42 -> [4, 2]
7 -> [7]
99 -> [9, 9]
0 -> [0]
```

(2) *(Cette question est indépendante de la précédente)* On donne un entier n et une liste L formée d'entiers ayant un ou deux chiffres entre 0 et 9 et on demande d'écrire une fonction estChiffres(L, n) qui renvoie True si L est la liste des chiffres de n lus de la gauche vers la droite et False sinon. Voici quelques exemples de comportement de la fonction estChiffres :

```
[7] , 7 -> True
[4, 2] , 42 -> True
[0, 7] , 7 -> True
[0] , 0 -> True
[1, 8] , 81 -> False
[7] , 42 -> False
[8, 1] , 2016 -> False
```

Moitiés

On peut toujours décomposer un entier comme somme deux entiers, l'un étant sa « petite » moitié et l'autre sa « grande » moitié. Par exemple, comme 81=40+41, la petite moitié de 81 est 40 et sa grande moitié est 41. De même, comme 100=50+50, la petite et la grande moitié de 100 sont égales chacune à 50. Dit plus rigoureusement, si $n = p + g$ avec p et g des entiers tels que $p \leq n/2 \leq g$ et $g - p$ minimal alors p est la petite moitié de n et g sa grande moitié.

Ecrire une fonction `moities(n)` qui à partir d'un entier $n \geq 0$ retourne la liste formée de sa petite moitié et de sa grande moitié. L'usage de la méthode `append` sera pénalisé. Voici quelques exemples de comportements de la fonction `moities` :

```
81 -> [40, 41]
100 -> [50, 50]
2021 -> [1010, 1011]
2038 -> [1019, 1019]
```

```
0 -> [0, 0]
```

Eléments distincts

Dans cet exercice, l'usage de la méthode *append* sera pénalisé.

Ecrire une fonction `f` qui prend deux paramètres entiers a et b et renvoie une liste formée des éléments distincts parmi les paramètres a et b. Voici deux exemples de comportement de `f` :

```
42, 81 -> [42, 81]
42, 42 -> [42]
```

Liste ordonnée à deux éléments

Ecrire une fonction `trier(a, b)` qui prend en paramètre deux entiers `a` et `b` et qui renvoie la liste contenant ces deux entiers et ordonnée dans le sens croissant. L'usage de la méthode append sera pénalisé.

Voici quelques exemples du comportement de la fonction :

```
2016 2020 -> [2016, 2020]
2020 2016 -> [2016, 2020]
2038 2038 -> [2038, 2038]
```

Éléments non nuls

Dans cet exercice, l'usage de la méthode *append* sera pénalisé.

Ecrire une fonction `f` qui prend deux paramètres entiers a et b et renvoie une liste formée de tous les éléments non nuls parmi les paramètres a et b. Voici quatre exemples de comportement de `f` :

```
a =  42 , b =  81 -> [42, 81]
a =  0 , b =  12 -> [12]
a =  21 , b =  0 -> [21]
a =  0 , b =  0 -> []
```

Objets qui s'accordent

On dispose de deux types d'objets : les objets de type `A` et les objets de type `B`.

Un objet de type `A` est représenté par une liste de couleurs, par exemple

```
["rouge", "vert", "noir", "rouge", "bleu"]
```

Les couleurs peuvent ou non être distinctes.

Un objet de type `B` est représenté par un entier `k` et une couleur et que l'on rassemblera dans une liste, par exemple

```
[3, "gris"]
```

On dit qu'un objet de type `A` (représenté par la liste de couleurs `L`) et un objet de type `B` (représenté par l'entier `k` et la couleur `s`) **s'accordent** si les deux conditions suivantes sont satisfaites :

— le nombre de couleurs dans `L`, distinctes ou non, et l'entier `k` sont égaux

— la couleur `s` ne figure pas dans la liste `L`

Ecrire une fonction `accord(a, b)` qui prenant deux paramètres `a` et `b` représentant un objet de type A et un objet de type B renvoie `True` si A et B s'accordent et `False`. L'argument représentant `a` sera une liste de couleurs et l'argument représentant `b` sera une liste formée d'un entier et d'une couleur. Les couleurs sont données sous forme de chaîne de caractères.

Par exemple, si `a` est représenté par la liste suivante

```
a = ["rouge", "vert", "noir", "rouge", "bleu"]
```

alors on observe les comportement suivants :

```
b = [5, 'rose'] : accord
b = [3, 'rose'] : désaccord
b = [4, 'rose'] : désaccord
b = [5, 'noir'] : désaccord
```

Cet exercice est directement inspiré du 2e exercice qualificatif du concours Prologin 2022.

Nombre de bus et division

Chaque bus d'une compagnie de transport peut contenir exactement `p` passagers.

1. Ecrire une fonction `nb_bus(n, p)` qui renvoie le nombre `k` d'une flotte de bus pour le transport de `n` passagers. Par exemple, si `p=100` alors si `n=2024` on aura `k=201` et si `n=2100` on aura `k=210`.
2. On peut démontrer que le nombre `k` de la question précédente est en fait le quotient de la division entière de `n+p-1` par `p`. Vérifier ce résultat en effectuant 1.000.000 de comparaisons de chaque résultat en choisissant au hasard `n` entre 0 et 1.000.000 et `p` entre 1 et 10.000.

Afficher un mouvement

Soit une liste `L` formée d'entiers parmi -1, 0 ou 1, par exemple `L= [-1, 0, -1, 0, 0, 1, -1, 0]`. Chaque entier représente une direction de déplacement :

- l'entier -1 représente le déplacement *à gauche*
- l'entier 0 représente le déplacement *tout droit*
- l'entier 1 représente le déplacement *à droite*

Ecrire une fonction `afficherDirection` qui prend en paramètre une liste `L` formée d'entiers parmi -1, 0 ou 1 et qui affiche une suite de directions de déplacement.

Par exemple, si `L= [-1, 0, -1, 0, 0, 1, -1, 0]` alors la fonction affichera :

```
à gauche
tout droit
à gauche
tout droit
tout droit
à droite
à gauche
tout droit
```

L'utilisation d'une liste de chaînes de caractères est encouragée mais il est toutefois possible de répondre à l'exercice sans y avoir recours

Doubler

Ecrire une fonction `f` qui prend en paramètre une liste d'entiers et renvoie True si chaque élément de la liste est le double du précédent.

Le code sera pénalisé s'il n'utilise pas une boucle `while` pour résoudre le problème.

Exemples de comportement :

```
[10, 20, 40, 80, 160] -> True
[10] -> True
[10, 20, 42, 80, 160] -> False
[] -> True
```

Moyenne olympique

La *moyenne olympique* d'une liste `L` d'au moins trois nombres est la moyenne des nombres de la liste `L` dont on a retiré deux valeurs :

— la plus grande valeur de `L`

— la plus petite valeur de `L`.

Par exemple, la moyenne olympique de la liste `L = [12, 7, 18, 10, 4, 13, 9]` est `10.2`.

$$\frac{12+7+10+13+9}{5} = 10.2$$

La moyenne olympique d'une liste d'un ou deux nombres est la moyenne habituelle.

Ecrire une fonction `moy_olymp(L)` qui renvoie la moyenne olympique d'une liste de nombres. Il est préférable que votre algorithme parcoure la liste une fois et une seule.

Un élément sur trois

Ecrire une fonction f qui prend une liste `L` d'entiers et qui renvoie la liste `M` formée d'un élément sur 3 de la liste `L` en commençant par la fin. Exemples :

```
[19, 43, 25, 54, 71, 76, 88, 53, 67, 22, 90, 79, 12, 25]
-> [25, 90, 53, 71, 43]

[19, 43, 25, 54] -> [54, 19]
[19] -> [19]
```

Sommer deux par deux

On donne une liste `L` d'entiers et on demande de produire une liste `M` d'entiers telle que chaque élément de `M` soit obtenu en faisant la somme deux éléments consécutifs de `L`. Par exemple, si `L` est la liste :

```
L=[5, 3, 4, 9, 6, 6, 0, 3]
```

alors `M` sera la liste

```
M=[8, 13, 12, 3]
```

Si `L` est formé d'un nombre impair d'éléments, le dernier élément de `L` sera placé à la fin de `M`. Par exemple, si `L` est la liste :

```
L=[5, 3, 4, 9, 6, 6, 0, 3, 5]
```

alors `M` sera la liste

```
M=[8, 13, 12, 3, 5]
```

1. Ecrire une fonction qui `somme_paires(L)` qui renvoie la liste `M`. On introduira le quotient entier `q` de `n` par 2 et on traitera le cas où `n` est impair.
2. Si on répète le procédé précédent jusqu'à ce que la liste ne contienne qu'un seul élément, alors cet élément sera la somme des éléments de la liste initiale. Par exemple, si on applique cette méthode à la liste `L=[5, 3, 7, 5, 1]`, voici les différentes étapes qu'on obtient :

   ```
   [5, 3, 7, 5, 1]
   [8, 12, 1]
   [20, 1]
   [21]
   ```

   et la somme des éléments de `L` est bien 21.
   Ecrire une fonction `somme(L)` qui applique cette méthode pour calculer la somme des éléments de `L`.

2 Construire, utiliser des fonctions

Des instructions à la construction d'une fonction

Soit à écrire le code d'une fonction qui calcule le plus grand de deux nombres a et b donnés.

On va exposer le processus de transformation permettant de passer d'un code qui répond au problème mais qui n'utilise pas de fonction à un code équivalent mais placé dans une fonction.

On commence par écrire un code n'utilisant pas de fonction mais qui réponde à la question, **quel que soit le choix** pour `a` et `b` :

```
1 a = 81
2 b = 31
3
4 if a >= b:
5     print(a)
6 else:
7     print(b)
```

```
8 81
```

Il est ESSENTIEL d'utiliser des variables car ce sont elles qui vont devenir les paramètres de la fonction à construire.

On remarque que, en l'état :

— le code ne peut être exécuté que pour deux choix particuliers de `a` et de `b`, cf. lignes 1-2 ;

— le code ne place pas le maximum dans un variable mais se contente de l'afficher.

Il est essentiel que le code calcule correctement le maximum **quelles que soient les valeurs choisies** pour `a` et `b`. Par exemple, si dans le code ci-dessus, on change 81 en 2020 et 31 en 3000,

le code doit encore afficher la bonne valeur (ici 3000).

Pour faciliter la conversion du code vers celui d'une fonction qui **renvoie** le maximum entre `a` et `b`, on peut placer le maximum dans une variable, par exemple `m` :

maximum_sans_fonction.py

```
a = 81
b = 31

if a >= b:
    m = a
else:
    m = b

print(m)
```

```
81
```

On va maintenant « convertir » ce code en celui d'une fonction. La fonction admet pour paramètres `a` et `b` et elle doit renvoyer le maximum `m`, donc le schéma de la fonction, que l'on va appeler `f`, est le suivant :

```
def f(a, b):
    # code inconnu ...
    # ...
    return m
```

Le code inconnu est obtenu en observant le code `maximum_sans_fonction.py` calculant le maximum `m`. Voici les deux codes comparés :

maximum_sans_fonction.py

```
a = 81
b = 31

if a >= b:
    m = a
else:
    m = b

print(m)
```

maximum_fonction.py

```
def f(a, b):
    if a >= b:
        m = a
    else:
        m = b
    return m

a = 81
b = 31
print(f(a, b))
```

— Lignes 1-2 : ces deux instructions ne servent pas dans le corps de `f`. Ces instructions qui définissaient `a` et `b` sont remplacées par l'en-tête de `f` ligne 10.

— Lignes 4-7 : ces lignes sont préservées à l'identique dans le code de `f`.

— Ligne 9 : comme `f` doit *renvoyer* le maximum `m`, on a remplacé l'affichage par une instruction `return` (ligne 15).

— Lignes 17-19 : on teste `f` de la même façon que le code initial avait été exécuté pour deux valeurs de `a` et `b`, cf. lignes 1-2.

Enchaîner des fonctions

Il est assez fréquent que, dans un programme Python, une fonction fasse appel à une autre fonction qui elle-même fait appel à d'autres fonctions et ainsi de suite. On dit qu'on *enchaîne* les fonctions.

Par exemple, soit le code suivant :

enchainer_fonctions.py

```
1 def f(x):
2     return g(x)*10
3
4 def g(x):
5     z=h(x) + k(x)
6     print("g : ", z)
7     return z
8
9 def h(x):
10    z=x+2
11    print("h : ", z)
12    return z
13
14 def k(x):
15    z=x+3
16    print("k : ", z)
17    return z
18
19 a = 42
20 print(a, f(a))
```

```
21 h :  44
22 k :  45
23 g :  89
24 42 890
```

— Ligne 1 : une fonction `f` est définie et fait appel (au sens d'un *appel* de fonction) à une autre fonction g.

— Ligne 4 : la fonction g est définie et fait elle-même appel à deux autres fonctions, définies plus bas, lignes 9 et 14.

— Lignes 6, 11 et 16 : on a placé des instructions d'affichage pour mieux suivre l'enchaînement des appels des différentes fonctions.

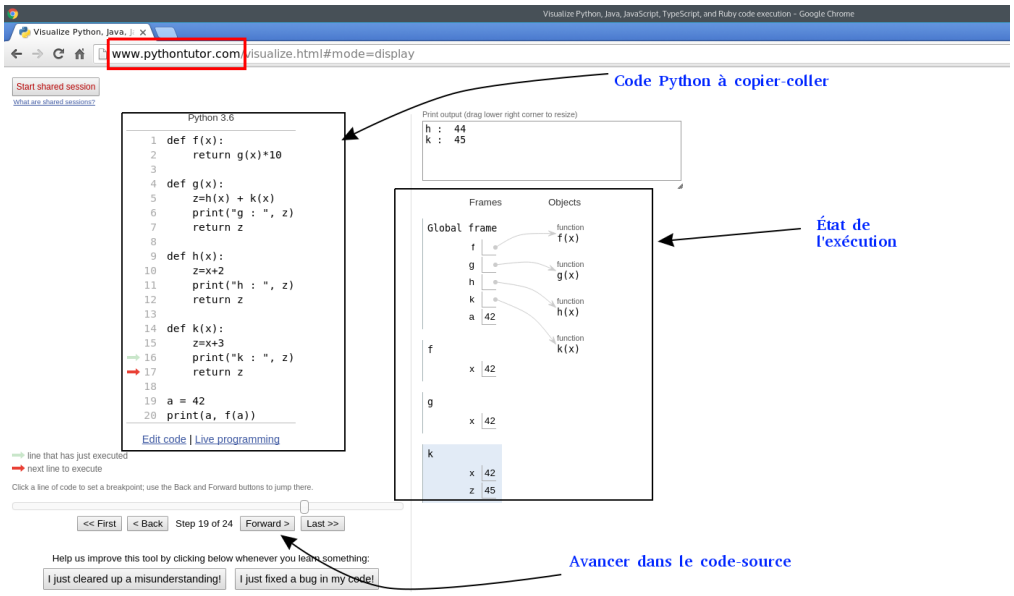
— Ligne 20 : l'appel `f(42)` est lancé.

  — Le code lignes 1-2 est exécuté mais la ligne 2 appelle la fonction g. Tant que la fonction g appelée ligne 2 n'aura pas renvoyé son résultat (la valeur de `g(x)`), la fonction `f` ne pourra rien renvoyer à l'expression appelante ligne 20 : on dit que la fonction `f` est *en attente*.

  — La fonction g appelle elle-même la fonction `h` et la fonction `k`.

  — Lorsque ces fonctions ont renvoyé leur résultat, la somme de ces résultats est placé dans la variable `z` de la définition de g. Puis g peut renvoyer son résultat à la fonction appelante, ici, la fonction f, à la ligne 2, qui elle-même peut renvoyer son résultat à l'appelant, ici la fonction `print` ligne 20.

Il peut être intéressant de visualiser la succession des appels de fonctions grâce au remarquable outil en ligne proposé par le site pythontutor. Ci-dessous une copie d'écran de l'outil appliqué au code ci-dessus :



Visualiser l'empilement des appels

Pile d'appels

On reprend le code `enchainer_fonctions.py` :

`enchainer_fonctions.py`

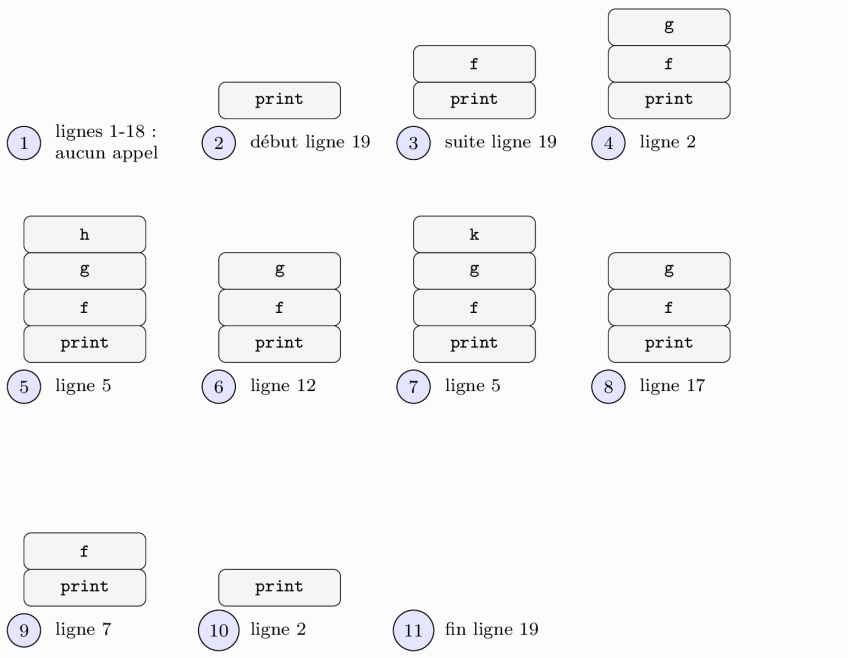
```
1 def f(x):
2     return g(x)*10
3
4 def g(x):
5     z=h(x) + k(x)
6     print("g : ", z)
7     return z
8
9 def h(x):
10    z=x+2
11    print("h : ", z)
```

```
12      return z
13
14  def k(x):
15      z=x+3
16      print("k : ", z)
17      return z
18
19  print(f(42))
```

Les appels de fonctions s'empilent pour former ce qu'on appelle la *pile* d'appels qu'on imagine comme des appels qui s'empilent de bas en haut.

— Lignes 1-18 : la pile d'appels demeure vide lors de la première exécution de toutes ces lignes.
— Ligne 19 : la fonction `print` est appelée, et donc la pile contient l'appel de `print`
— Ligne 19 : la fonction `f` est appelée, et donc la pile contient, de bas en haut, l'appel de `print` puis l'appel de `f`
— Ligne 2 : g est appelée, donc la pile contient de bas en haut, l'appel de `print`, l'appel de `f` et au-dessus l'appel de g
— Ligne 5 : la fonction `h` est appelée, la pile contient de bas en haut : les appels de `print`, de `f` puis de g puis de `h`.
— Ligne 5 : quand `h` renvoie sa valeur (lignes 12) à la fonction g qui l'a appelée, la fonction `h` quitte la pile d'appels. On dit alors que `h` *dépile*. La pile contient alors l'appel de `print`, de `f` puis l'appel de g qui sont en attente.
— Ligne 5 : la fonction `k` est appelée donc la pile est constituée de bas en haut de : l'appel de `print`, l'appel de `f`, l'appel de g, l'appel de `k`.
— `k` dépile (lignes 17) puis quand g renvoie z (lignes 7) à `f` (ligne 2) qui dépile aussi puis `print` (ligne 19) et la pile est à nouveau vide.

Ci-dessous, on peut observer l'évolution de la pile des appels.



La pile des appels

La pile des appels est bien visible à l'aide d'un débogueur.

De l'usage des fonctions

Pour un débutant qui a déjà écrit du code Python mais sans utiliser encore la notion de fonction, écrire une fonction peut être une tâche difficile car il s'agit en général de transposer en code un raisonnement.

Intérêt d'écrire des fonctions

L'utilisation de fonctions présente les trois intérêts suivants :

- le code est plus **lisible** que s'il n'est pas enveloppé dans une fonction car une fonction identifie clairement :
  - une tâche (virtuelle) à exécuter avec une interface constituée de paramètres que l'on donne à la fonction et d'un résultat (ce que renvoie l'instruction `return`),
  - d'un (ou plusieurs) appel(s) à la fonction ;
- le code est **réutilisable**. Si on n'écrit pas le code dans une fonction, il faudrait récrire tout le code correspondant chaque fois que l'on souhaiterait effectuer la même tâche ;
- le code est plus économique. Appeler une fonction, c'est une ligne de code qui en fait en appelle souvent plusieurs dizaines puisque l'appel équivaut à l'exécution de toutes les lignes de code de la fonction et des autres fonctions que la fonction appelle elle-même.

Découpage en tâches

Pour mieux organiser son code, le programmeur cherche souvent à découper la tâche à effectuer en plusieurs fonctions, chacune réalisant une certaine sous-tâche (on dit parfois « service » au lieu de « tâche » ou « sous-tâche »).
Lorsqu'on est débutant avec les fonctions, il n'est pas toujours facile d'arriver à découper son programme en fonctions pertinentes.
Lors de l'écriture d'une fonction `f`, le programmeur doit d'abord s'interroger sur l'*interface* de la fonction `f` :

- quels seront les paramètres de f? autrement dit, de quelles données la fonction `f` a-t-elle besoin ?
- qu'est ce que la fonction f doit renvoyer ?
- la fonction a-t-elle des effets de bords ?

Ensuite, il doit écrire le code de définition de la fonction `f` qui va être capable de réaliser la tâche : c'est ce qu'on appelle l'*implémentation*[3] de la fonction.

Exemple

Par exemple, soit le tableau de Pascal :

3. Terme à préférer à celui-d'*implantation* parfois utilisé mais à tort,

$n =$ \ $p =$	0	1	2	3	4	5	6	7	8	9
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1			
7	1	7	21	35	35	21	7	1		
8	1	8	28	56	70	56	28	8	1	
9	1	9	36	84	126	126	84	36	9	1

Le triangle de Pascal

Comment ça marche ? Chaque élément z d'une ligne s'obtient comme somme des deux termes au-dessus et à gauche de l'élément z. Par exemple, à la ligne 7, on a : 35 = 20 + 15. On veut écrire un programme qui affiche le tableau de Pascal jusqu'à sa n^{e} ligne .

Plus précisément, voici les règles de construction du Tableau de Pascal. Le Tableau de Pascal est une suite de lignes telles que :

— la 1re ligne est `1 1` ;
— chaque ligne commence et se termine par le nombre 1 ;
— pour chaque élément z qui ne figure pas aux extrémités d'une ligne L du tableau, on a $z = x + y$ où x est le terme sur la ligne précédente de L et situé au-dessus de z et y est le voisin de gauche de x.

Objectif : écrire une fonction `pascal(n)` qui prend en paramètre un numéro `n` de ligne et qui affiche les n premières lignes du tableau de Pascal.

Pour afficher une ligne, il suffit de connaître la précédente (disons `L`) et d'appliquer les règles de construction du Tableau de Pascal. Cette tâche sera accomplie par une fonction `ligne_suivante(L)`. Pour bien comprendre ce qui suit, il faut savoir qu'une ligne du tableau de Pascal sera représentée dans le programme par une *liste* d'entiers et donc construire le tableau de Pascal revient à construire une liste de listes.

Voici un code possible de la fonction `ligne_suivante(L)`, accompagné d'un test :

```
1  def ligne_suivante(L):
2      LL=[1]
3      n=len(L)
4      for k in range(1, n):
5          LL.append(L[k-1]+L[k])
6      LL.append(1)
7      return LL
8
9  L=[1, 3, 3, 1]
10 print(ligne_suivante(L))
```

```
11 [1, 4, 6, 4, 1]
```

— Ligne 2 : LL est la future ligne suivante et au départ une liste contenant juste le 1 initial de chaque ligne et qui va être remplie au fur et à mesure.

— Lignes 6 : l'autre extrémité de chaque ligne du tableau vaut 1.

— Lignes 4-5 : la règle de construction du terme `LL[k]` de la ligne suivante de `L` dans le Tableau de Pascal. On sait que `LL[k]` est la somme des deux termes de `L` situés en haut et à gauche de `LL[k]`.

— Ligne 7 : la fonction doit renvoyer la nouvelle ligne.

— Lignes 9-11 : un test, positif puisque la ligne suivante de la ligne `1 3 3 1` est bien `1 4 6 4 1`.

La fonction `pascal(n)` va utiliser la fonction `ligne_suivante(L)` dans une boucle pour donner une première version du tableau de Pascal que l'on va affiner ensuite :

```
1  def ligne_suivante(L):
2      LL=[]
3      LL.append(1)
4      n=len(L)
5      for k in range(1, n):
6          LL.append(L[k-1]+L[k])
7      LL.append(1)
8      return LL
9
10 def pascal(n):
11     L=[1, 1]
12     print(L)
13     for i in range(0, n-1):
14         LL=ligne_suivante(L)
15         L=LL
16         print(L)
17
18 pascal(10)
```

```
19 [1, 1]
20 [1, 2, 1]
21 [1, 3, 3, 1]
22 [1, 4, 6, 4, 1]
23 [1, 5, 10, 10, 5, 1]
24 [1, 6, 15, 20, 15, 6, 1]
25 [1, 7, 21, 35, 35, 21, 7, 1]
26 [1, 8, 28, 56, 70, 56, 28, 8, 1]
27 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
28 [1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]
```

— Ligne 10 : il y a n lignes à afficher

— Lignes 11-12 et 19 : on affiche la 1re ligne du Tableau de Pascal

— Ligne 13 : il reste `n-1` lignes à afficher d'où le `range(n-1)`.

— Ligne 14 : L est la ligne courante, LL la ligne suivante, renvoyée par la fonction `ligne_suivante`

— Ligne 15 : le point-clé du programme : on prépare le tour suivant dans la boucle en mettant à jour la ligne courante. Il aurait été possible de remplacer le code des lignes 14-15 par l'unique ligne `L=ligne_suivante(L)`.
— Ligne 16 : affichage de la ligne courante.
— Lignes 18 et 19-28 : on reconnaît bien les 10 premières lignes du Tableau de Pascal.

On se rend compte en regardant l'affichage obtenu qu'il peut être amélioré pour faire afficher juste des nombres, sans virgules ni crochets. Pour cela, il suffit d'écrire une fonction `afficher_ligne` dont la tâche est d'afficher le contenu d'une liste sur un seule ligne en séparant les éléments par une seul espace :

```
1 def afficher_ligne(L):
2     for i in range(len(L)):
3         print(L[i], end = " ")
4     print()
5
6 afficher_ligne([1, 4, 6, 4, 1])
7 afficher_ligne([1, 8, 28, 56, 70, 56, 28, 8, 1])
```

```
8 1 4 6 4 1
9 1 8 28 56 70 56 28 8 1
```

Il ne reste plus qu'à remplacer l'affichage `print(L)` de la fonction `pascal(n)` à la ligne 16 par `afficher_ligne(L)` pour obtenir le programme attendu :

```
1 def ligne_suivante(L):
2     LL=[1]
3     n=len(L)
4     for k in range(1, n):
5         LL.append(L[k-1]+L[k])
6     LL.append(1)
7     return LL
8
9 def pascal(n):
10    L=[1, 1]
11    afficher_ligne(L)
12    for i in range(0, n-1):
13        L=ligne_suivante(L)
14        afficher_ligne(L)
15
16 def afficher_ligne(L):
17    for i in range(0, len(L)):
18        print(L[i], end = " ")
19    print()
20
21 pascal(10)
```

```
22 1 1
23 1 2 1
24 1 3 3 1
25 1 4 6 4 1
```

```
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

L'affichage pourrait être encore amélioré pour faire apparaître des colonnes bien alignées.

Exercice type – Périodes de l'année

Dans cet exercice, on va considérer des périodes sous la forme

a années, m mois et j jours

où $1 \leq$ m ≤ 12 et $1 \leq$ j ≤ 30 où a, m et j sont des entiers. Pour simplifier, on considérera qu'une année comportera 12 mois et qu'un mois dure *toujours* 30 jours (et donc qu'une année comporte 360 jours et non 365).

1. Écrire une fonction `amj_to_j(a, m, j)` qui retourne le nombre de jours d'une période exprimée en années, mois et jours. Vérifier que 27 ans, 9 mois et 10 jours correspondent à 10000 jours.
2. *Cette question est indépendante de ce qui précède.* Dans cette question, l'usage de la méthode *append* est inapproprié.
   Écrire une fonction `j_to_amj(jours)` qui prend en paramètre une période exprimée en jours et la retourne exprimée en années, mois et jours sous forme de liste [a, m, j]. Bien sûr, on aura $1 \leq$ m ≤ 12 et $1 \leq$ j ≤ 30. Vérifier que 10000 jours correspondent à 27 ans, 9 mois et 10 jours.
3. *Cette question est indépendante de ce qui précède.* Écrire une procédure `afficher_periode` qui prend en paramètre une liste de 3 entiers correspondant à une période de a années, m mois et j jours et qui affiche

   ```
   a années m mois et j jours
   ```

   Par exemple, `afficher_periode([27, 9, 1])` affiche exactement :

   ```
   27 années 9 mois et 1 jours
   ```

4. *Votre réponse à cette question doit utiliser les fonctions définies dans les questions précédentes. Cette question ne demande pas de définir de nouvelle fonction.*
   Un père a un âge de 42 ans, 4 mois et 2 jours et sa fille a un âge de 14 ans 6 mois et 22 jours. Écrire un programme qui affiche la différence d'âge, exprimée en années, mois et jours, entre le père et sa fille.
   Il est attendu que votre code tienne sur une seule ligne.

Solution

1. D'après les hypothèses simplificatrices de l'énoncé, le nombre de jours dans une période de a années, m mois et j jours est donné par la formule :

   ```
   360*a+30*m+j
   ```

d'où le code :

```
1 def amj_to_j(a, m, j):
2     return 360*a+30*m+j
3
4 print(amj_to_j(10, 4, 22))
```

```
5 3742
```

— Ligne 5 : on retrouve bien le nombre de jours annoncés.

(2) Une périodes `jours` contient `jours // 360` années entières avec un reliquat de `jours % 360` jours. De la même façon, on convertit ce reliquat de jours en mois et jours. D'où le code :

```
1 def j_to_amj(jours):
2     a=jours//360
3     r=jours%360
4     m=r//30
5     j=r%30
6     return [a, m, j]
7
8 print(j_to_amj(10000))
```

```
9 [27, 9, 10]
```

— Ligne 9 : on retrouve bien 27 ans, 49 mois et 10 jours.

(3) Il s'agit d'écrire une procédure d'affichage. La fonction reçoit un **argument** sous forme de liste `[j, m, a]`. Mais le paramètre ne peut avoir cette forme, un paramètre étant toujours un nom de variable, disons ici `periode`. Dans le code de définition de la fonction, on accède aux années, mois et jours en utilisant un indice puisque `periode` représente une liste de 3 entiers. D'où le code :

```
1 def afficher_periode(periode):
2     print(periode[0], "années",  periode[1],
3           "mois", "et", periode[2], "jours")
4
5 afficher_periode([27, 9, 1])
```

```
6 27 années 9 mois et 1 jours
```

(4) Pour calculer un écart entre deux durées, il suffit de

— convertir chaque durée en jours avec la fonction `amj_to_j`
— de faire la différence des durées en jours
— de reconvertir cette durée en jours en une période en années, mois et jours avedc la fonction `j_to_amj`.

Enfin, il restera à afficher la durée avec la fonction `afficher_periode`.
D'où le code :

```
1 afficher_periode(j_to_amj(amj_to_j(42, 4, 2)-amj_to_j(9, 9, 28)))
```

```
2 32 années 6 mois et 9 jours
```

Modification par une fonction d'un de ses arguments

Une fonction peut-elle modifier un objet qu'elle reçoit en argument ? La réponse doit être nuancée en fonction du sens que l'on donne au verbe *modifier*.

En aucun cas, une fonction ne peut substituer un autre objet à un objet qu'elle reçoit en argument.

Examinons une tentative de modification par une fonction d'un objet que la fonction reçoit en argument. Soit par exemple le code ci-dessous

```
1 def f(x):
2     x = 42
3
4 a=10
5 print(a)
6 f(a)
7 print(a)
```

```
8 10
9 10
```

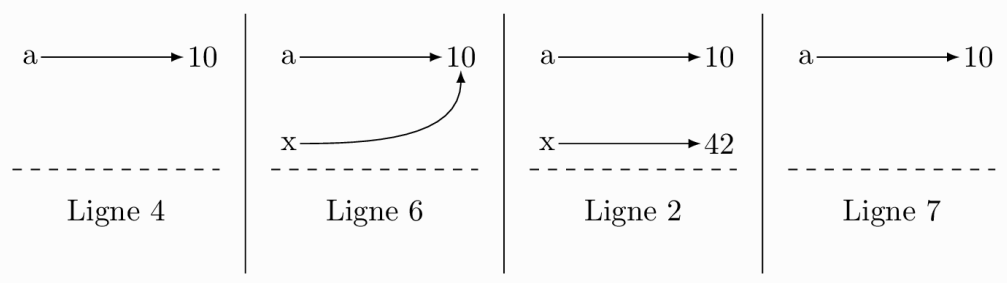
— Lignes 4 et 8 : la valeur de a avant l'appel

— Lignes 2 et 6 : l'action de f qui tente de modifier l'objet reçu en argument

— Ligne 7 : la valeur de a est inchangée et c'est le même objet 10 que celui vers lequel a se référait avant l'appel.

La fonction f « affecte » 42 au paramètre x. Si on passe en argument à f une autre valeur comme 10, cette valeur va-t-elle être remplacée par 42 ? Non, comme le montre l'affichage ci-dessus.

Le schéma ci-dessous montre les affectation effectuées tout du long du programme ci-dessus :



Passage des paramètres par affectation

Pour comprendre le comportement de f et pourquoi a n'est pas modifié, il suffit de se rappeler que l'action d'un passage des arguments en Python est une transmission par affectation et donc le code ci-dessus est équivalent à :

```
1 a=10
2 print(a)
3
4 # ----- Équivalent de l'action f(a)
5 x = a
6 x = 42
```

```
7 # -----
8
9 print(a)
```

```
10 10
11 10
```

Les affectations x=a et x=42 ne peuvent pas remplacer a par 42 ; en effet, x=42 ne fait que créer un nouvel objet 42 et lui associer le nom x, autrement dit, la variable x ne référence plus l'objet 10 qu'elle référait ligne 5. On voit que ces affectations ne peuvent modifier a qui lui ne subit aucune affectation.

À défaut de remplacer un objet par un autre, une fonction peut-elle modifier le contenu d'un objet ? La réponse dépend de la nature de l'objet.

Si une fonction reçoit en argument un objet immuable comme un entier ou une chaîne, par définition, l'objet ne pourra pas être modifié, et a fortiori par f.

Le cas d'un objet mutable

Si une fonction reçoit en argument un objet mutable comme une liste, l'objet pourra être modifié par la fonction. Voici un exemple où une liste est modifiée par une fonction :

```
1 def f(M):
2     M[0] = 42
3
4 L=[81, 12, 31, 65]
5 print(L)
6 print()
7
8 f(L)
9
10 print(L)
```

```
11 [81, 12, 31, 65]
12
13 [42, 12, 31, 65]
```

— f est appelée sur une liste d'entiers

— l'action de consiste à remplacer le premier élément de M par 42

— une fois l'appel de f terminé, on constate que l'objet L placé en paramètre a été modifié.

Une fois de plus, le mode de passage des arguments par affectation propre à Python permet de prévoir ce comportement. En effet le code ci-dessus est équivalent à

```
1 L=[81, 12, 31, 65]
2 print(L)
3 print()
4
5 # ---- Début de l'action de f
6
7 M=L
8 M[0]=42
```

```
9
10 # ---- Fin de l'action de f
11
12 print(L)
```

```
13 [81, 12, 31, 65]
14
15 [42, 12, 31, 65]
```

L'exemple ci-dessus ne foit pas faire croire que l'argument a été changé : après appel, L référence toujours le même objet. La différence, c'est que le contenu de L a changé.

Boucle for et return

Le corps d'un boucle peut contenir n'importe quelle instruction ; en particulier, si une boucle for apparaît dans la définition d'une fonction, elle peut être interrompue par une instruction return.

Par exemple, soit à définir une fonction qui prend une liste L en paramètre et renvoie True si L ne contient aucun entier strictement négatif et False sinon :

```
1 def f(L):
2     for i in range(len(L)):
3         if L[i] < 0:
4             return False
5     return True
6
7 L= [31, 82, -42, 32, -81]
8 print(L, f(L))
9
10 print("=========================")
11
12 L= [31, 82, 421, 32, 81]
13 print(L, f(L))
```

```
14 [31, 82, -42, 32, -81] False
15 =========================
16 [31, 82, 421, 32, 81] True
```

— Lignes 1-5 : la fonction f répond au problème posé.

— Ligne 4 : une instruction return interrompt l'exécution de la fonction.

— Lignes 3-4 : la liste L est parcourue et si un terme de la liste L est un nombre négatif, le parcours de la liste ainsi que l'exécution de la fonction sont interrompus et la fonction renvoie False

— Ligne 5 : si la liste L est parcourue sans que jamais le parcours ne soit interrompu par la détection d'un nombre négatif dans la liste, la fonction f se termine en renvoyant True puisque tous les termes de L sont positifs ou nuls.

Résumons le schéma ci-dessus :

— placement d'une boucle for dans une fonction ;

— parcours avec la boucle for d'une liste jusqu'à ce qu'une certaine condition soit vérifiée ;

— interruption de la boucle et de la fonction par une instruction return renvoyant un booléen.

L'intérêt de ce schéma est que le parcours de la liste est optimal : le parcours est interrompu dès que la réponse est connue, ce qui n'est pas le cas de la méthode utilisant un simple drapeau.

Exercice type – Grille en équerres

(1) Écrire une procédure `f(i,n)` où `i,n` sont des entiers tels que $1 \leq i \leq n$ qui affiche, sur une même ligne, `n` entiers séparés par une espace dont les `i` premiers entiers sont $1, 2, \ldots, i$ et les $n - i$ derniers sont tous identiques à l'entier `i`. Par exemple, `f(5,9)` doit afficher la ligne suivante :

```
1 2 3 4 5 5 5 5 5
```

(2) En déduire une procédure `g(n)` qui affiche une grille carrée « en équerres » telle que celle qui figure ci-contre (dans cet exemple $n = 9$). Deux nombres sur une même ligne seront séparés par une espace. Observez bien comment sont placés « en équerres » les nombres 1, 2, etc.

```
1 1 1 1 1 1 1 1 1
1 2 2 2 2 2 2 2 2
1 2 3 3 3 3 3 3 3
1 2 3 4 4 4 4 4 4
1 2 3 4 5 5 5 5 5
1 2 3 4 5 6 6 6 6
1 2 3 4 5 6 7 7 7
1 2 3 4 5 6 7 8 8
1 2 3 4 5 6 7 8 9
```

Solution

(1) Pour calculer `f(i,n)`, il faut effectuer deux actions :

- la génération des entiers de 1 à `i` ;
- la répétition $n - i$ fois de l'entier `i`.

Ces deux actions répètent quelque chose et donc se codent chacune avec une boucle `for`. D'où le code :

```
1  def f(i,n):
2      for j in range(0, i):
3          print(j+1, end=' ')
4
5      for k in range(0,n-i):
6          print(i, end = ' ')
7
8  f(5,9)
9  print()
10 print()
11 f(1,9)
```

```
12 1 2 3 4 5 5 5 5 5
13
```

```
14 1 1 1 1 1 1 1 1 1
```

— Lignes 3 et 6 : pour séparer chaque entier du suivant par juste un espace, on utilise l'argument nommé `end=" "` pour la fonction `print`.

(2) Pour afficher le motif 2D demandé, il suffit de remarquer que la 1re ligne du motif est `f(1, n)`, la 2e ligne est `f(2, n)` et ainsi de suite. Donc, pour afficher le motif, il suffit de répéter avec une boucle `for` l'affichage de chaque ligne. D'où le code :

```
1  def f(i,n):
2      for j in range(0, i):
3          print(j+1, end=' ')
4
5      for k in range(0,n-i):
6          print(i, end = ' ')
7
8  def g(n):
9      for k in range(n):
10         f(k+1,n)
11         print()
12
13 g(9)
```

qui affiche

```
1 1 1 1 1 1 1 1 1
1 2 2 2 2 2 2 2 2
1 2 3 3 3 3 3 3 3
1 2 3 4 4 4 4 4 4
1 2 3 4 5 5 5 5 5
1 2 3 4 5 6 6 6 6
1 2 3 4 5 6 7 7 7
1 2 3 4 5 6 7 8 8
1 2 3 4 5 6 7 8 9
```

Variables locales

La notion de variable locale

variable_locale.py

```
1 def f(x):
2     y=x+1
3     return 10*y
4
5 print(f(8))
```

Lorsqu'on définit une fonction, ici la fonction `f` lignes 1-3, les variables dites *locales* à la fonction `f` sont

— les variables données en paramètre, comme `x` (ligne 1)

— les variables **créées** dans la définition de la fonction, par exemple `y` à la ligne 2.

Les objets que représentent ces variables n'existent que pendant l'exécution de la fonction f, c'est-à-dire uniquement lors de l'appel de la fonction f. Ainsi, la variable x n'a pas d'existence tant que f n'a pas été appelée. En outre, après la fin de l'exécution de la fonction, la variable x n'a plus d'existence non seulement en mémoire mais aussi dans le code-source. Les variables locales disparaissent avec l'exécution de la fonction.

Tentative d'accès à une variable locale

Le caractère local de variables comme x ou y signifie que ces variables ne sont pas connues à l'extérieur du code de la fonction `f`. Si on tente d'y accéder, on obtient une erreur.

Voici deux codes qui illustrent des tentatives de lecture de variables locales :

```
1 def f(x):
2     y=x+1
3     return 10*y
4
5 print(f(8))
6 print(x)
```

```
7 90
8 Traceback (most recent call last):
9   File "a.py", line 6, in <module>
10    print(x)
11 NameError: name 'x' is not defined
```

— Lignes 5 et 1 : la variable x prend la valeur 8 pendant l'exécution de `f` pendant l'appel ligne 5.

— Ligne 11 : pourtant, la variable x N'est PAS reconnue en dehors du code de définition de la fonction `f`.

— Ligne 7 : le résultat de l'affichage de la ligne 5.

```
1 def f(x):
2     y=x+42
3     z=10
4     return 2*y+z
5
6 print(f(8))
7 print(z)
```

```
8 110
9 Traceback (most recent call last):
10   File "a.py", line 7, in <module>
11     print(z)
12 NameError: name 'z' is not defined
```

— Ligne 3 : définition de la variable locale `z` de la fonction `f`.

— Ligne 6 : pendant l'appel de `f`, la variable `z` est utilisée

— Lignes 7 et 12 : après l'appel de `f` , la variable `z` cesse complètement d'exister.

Remède

Une variable locale à une fonction n'a pas vocation à être accédée depuis l'extérieur de la fonction. Si on veut accéder à une variable locale (en fait à son contenu), il faut que la fonction renvoie le contenu de cette variable locale ou renvoie un objet qui permette d'accéder à la valeur de cette variable locale.

Variables globales

Soit le code suivant :

```
1 def f(x):
2     print(z)
3     return z + x
4
5 z=42
6
7 print(f(8))
```

```
8 42
9 50
```

La variable `z` déclarée à la ligne 5 en dehors de toute fonction : on dit que `z` est une variable *globale*. La fonction `f` définie à la ligne 1 a accès à cette variable (lignes 2-3). Observons que :

— la variable globale z est définie **après** la définition de `f`. En fait ce qui compte, c'est que `z` soit définie **avant l'appel** (ligne 7) à la fonction f.

— l'accès à la variable globale `z` se fait en **lecture** et non **en écriture** ie la variable z n'est pas modifiée par la fonction `f`.

Quand on débute en programmation, et qu'on commence à manipuler des fonctions, il est conseillé d'éviter d'utiliser des variables globales. L'intérêt d'une fonction est de constituer un environnement autonome d'exécution : en principe, une fonction ne doit dépendre d'aucun élément extérieur autre que des appels éventuels à d'autres fonctions [4].

Si, pour des raisons exceptionnelles, une fonction est amenée à utiliser une variable globale, il est souhaitable, pour des raisons de lisibilité, de déclarer ces variables en début de la zone d'édition du code source, avant les fonctions qui y font appel.

Constantes placées en variables globales

Limiter l'usage de variables globales est considéré comme une bonne pratique de programmation. Utiliser des constantes en variables globales est un cas toléré. Il s'agit de variables qui référencent des objets qui ne changeront pas durant toute la vie du programme.

Un programme utilisant une valeur de π pour calculer des aires ou des volumes pourra définir π comme variable globale. Voici un exemple :

```
1 PI = 3.1415926
2
```

4. Certains programmeurs vont même jusqu'à refuser d'écrire tout code exécutable autre que dans une fonction.

```
3 def disque(r):
4     return PI * r * r
5
6 def sphere(r):
7     return 4 * PI * r * r
8
9 print(disque(10))
10 print(sphere(10))
```

```
11 314.15926
12 1256.63704
```

— Ligne 1 : constante déclarée en variable globale.

— Lignes 4 et 7 : utilisation de `PI` dans des fonctions.

L'usage veut que des variables globales référençant des constantes soient écrites en début de fichier et en majuscules.

Fonction non appelée

Soit le code suivant :

```
1 z=10
2
3 def f(x):
4     print(10 * x)
5
6 print(z+1)
```

```
7 11
```

— Lignes 3-4 : une fonction `f` est définie mais cette fonction n'est appelée nulle part dans le programme.

En pratique, un code n'a pas de raison de définir une fonction sans appeler cette fonction avec des arguments. Toutefois, cela signifie que vous pouvez, sans danger, écrire la définition d'une fonction qui, par exemple, serait inachevée, ou qui ne marcherait pas encore ou qui ne serait qu'une ébauche ou un template d'un programme à exécuter ultérieurement.

Mais il existe des raisons plus sérieuses à écrire des fonctions sans les appeler dans le fichier où les fonctions sont définies : il est possible d'appeler ces fonctions depuis un autre fichier Python.

Le passage des arguments par affectation

Soit le code :

```
1 def f(d,u):
2     N = 10 * d + u
3     return N
4
5 print(f(4,2))
```

```
42
```

— Lignes 1-3 : une définition de fonction

— Ligne 5 : un appel de la fonction avec les arguments 4 et 2

Lors de l'appel f(4,2) ligne 5 dans le code ci-dessus, la transmission des arguments 4 et 2 aux paramètres de f est effectuée **exactement** comme dans le code ci-dessous :

```
d = 4
u = 2

N = 10 * d + u

print(N)
```

Le lien qui existe entre un paramètre, par exemple d, son argument 4, est une affectation (ligne 1), d'où la terminologie, propre à Python de *passage des arguments par affectation*. L'objet que le paramètre d reçoit au moment de l'exécution de la fonction f est exactement le même objet que l'argument. Non seulement, l'objet reçu a la même valeur, mais, mieux, ils sont identiques.

Pour se rendre compte qu'il s'agit du **même** objet, on utilise la fonction standard id qui permet d'identifier un objet en renvoyant son « identifiant » unique afin de le discerner d'autres objets :

```
def f(d,u):
    print("id(d) ->", id(d))
    N = 10 * d + u
    return N
x=4
print("id(x) ->", id(x))

print(f(x,2))
```

```
id(x) -> 139697383176736
id(d) -> 139697383176736
42
```

— Lignes 1-4 : on a modifié la fonction f d'un code antérieur pour qu'elle affiche l'id de l'objet qu'elle reçoit pour le paramètre d

— Ligne 2 : on affiche l'id de l'objet associé à la variable x, plus bas l'objet 4.

— Lignes 9-10 : on constate qu'il s'agit des mêmes objets.

En particulier, le passage des arguments lors de l'appel d'une fonction se fait **sans copie** de l'objet. Le fait qu'il n'y ait pas de copie est une garantie d'efficacité. Toute action que la fonction peut avoir sur un objet lors de son exécution doit pouvoir être exécutée **comme si** l'objet était accédé par une affectation.

EXERCICES

Afficher une somme

1. On donne un entier `s` et une liste L d'entiers. Ecrire une fonction `estSomme(L, s)` qui renvoie `True` si la somme des éléments de L vaut `s` et `False` sinon.
Voici quelques exemples de comportement de `estSomme` :

```
[3, 2, 1], 6 -> True
[3, 2, 1], 8 -> False
```

2. (Afficher une addition) Vous allez devoir coder une fonction d'affichage de messages comme celui-ci

```
42 + 10 + 20 est égal à 72
```

ou encore celui-là :

```
42 + 10 + 20 est différent de 81
```

Plus précisément, on vous donne

- une liste `L` de *trois* entiers (nommés ci-dessous `a`, `b` et `c` juste pour la description de la question)
- un entier s

et vous devez écrire une fonction `afficherSomme(L, s)` qui devra obligatoirement utiliser la fonction `estSomme` et qui affichera le message

```
a + b + c est égal à s
```

si on a $a + b + c = s$ et sinon qui affichera

```
a + b + c est différent de s
```

Bien sûr, dans ces messages, `a`, `b`, `c` et `s` doivent apparaître remplacés par leurs valeurs comme dans les exemples ci-dessus.

Indice de masse corporelle

1. Ecrire une fonction `imc(p, t)` qui à partir du poids `p` en kilogrammes et de la taille `t` en mètre d'un individu renvoie son indice de masse corporelle (IMC) et qui se calcule par la formule $\frac{m}{t^2}$. Par exemple, si une personne mesure `1,74 m` et pèse `65,5 kg`, son IMC vaut environ `21,63`.
2. Ecrire une fonction `info_corpulence(p, t)` qui à partir du poids `p` en kilogrammes et de la taille `t` en mètre d'un individu affiche une information sur la corpulence de la personne. La fonction calculera son IMC avec la fonction de la question précédente et indiquera les résultats suivants :

- *corpulence : anorexie* si l'IMC est inférieure à `16,5`
- *corpulence : maigreur* si l'IMC est entre `16,5` et `18,5`
- *corpulence : normale* si l'IMC est entre `18,5` et `25`

— *corpulence : surpoids* si l'IMC est entre 25 et 30
— *corpulence : obésité modérée* si l'IMC est entre 30 et 35
— *corpulence : obésité sévère* si l'IMC est entre 35 et 40
— *corpulence : obésité maladive* si l'IMC est supérieure à 40.

Les intervalles seront de la forme [a, b[. Pour une personne qui mesure 1,74 m et pèse 65,5 kg, le message à afficher sera :

```
corpulence : normale
```

Histoire de zéros

1. On donne une liste dont les éléments sont parmi 0 ou 1.
Ecrire une fonction `compterZeros(L)` qui renvoie le nombre de zéros que contient la liste L. Par exemple, si L est la liste

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

la fonction doit renvoyer 4.

2. Utiliser la fonction `compterZeros` pour écrire une fonction *estMajoritaire(L)* qui renvoie
— 0 si les zéros sont majoritaires dans L ou en nombre égal aux uns
— 1 sinon.

Avec l'exemple de la liste de la question précédente, la fonction devra renvoyer 1.

Somme des chiffres

On admet que la fonction dont le code est dans la cellule ci-dessous renvoie la somme des chiffres de l'entier `n` :

```
def sommeChiffres(n):
    return sum(map(int, str(n)))
```

1. Tester la fonction `sommeChiffres(n)` pour `n` = 578679.
2. Ecrire une fonction `afficherSomme(n)` qui se contente d'afficher le message suivant

```
La somme des chiffres du nombre n vaut s
```

et où `n` est remplacé par la valeur de `n` et où `s` est remplacé par la somme des chiffres de `n`. Par exemple, pour `n` = 578679, l'appel `sommeChiffres(n)` affiche le message suivant :

```
La somme des chiffres du nombre 578679 vaut 42
```

3. Écrire une fonction `nb_dsum(N, s)` qui renvoie le nombre d'entiers entre 0 et `N` (inclus) et dont la somme des chiffres vaut `s`. Appliquer à `N` valant 1 million et `s = 42` (on trouvera 6062 entiers).

Entiers narcissiques

L'entier 153 est dit *narcissique* car il a `p` = 3 chiffres et qu'il vaut la somme des cubes de ses chiffres (en base 10) :

$$1^p + 5^p + 3^p = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$$

Plus généralement, on dit qu'un entier `n` > 0, ayant `p` chiffres, est *narcissique* si `n` vaut la somme de ses chiffres élevés à la puissance `p`. Les chiffres sont les chiffres de l'écriture de `n` en **base 10**.

On aura besoin de la fonction `chiffres` suivante :

```
def chiffres(n):
    return [int(c) for c in str(n)]

L = chiffres(2038)
print(L)
```

et qui affiche ici

```
[2, 0, 3, 8]
```

Autrement dit, `chiffres(n)` renvoie la liste formée des **entiers** qui constituent les chiffres en base 10 de l'entier `n`.

1. Ecrire une fonction `estNarcissique(n)` qui renvoie `True` si l'entier `n` est narcissique et `False` sinon.
2. En utilisant la fonction précédente, afficher la liste de tous les entiers narcissiques avant un million.

Plus loin, plus proche

1. (Écart entre deux nombres) Ecrire une fonction ecart(a, b) qui renvoie l'écart entre les deux entiers a et b. Voici quelques exemples de comportement de la fonction ecart :

```
82 , 42 -> 40
1970 , 2038 -> 68
42 , 42 -> 0
42 , -10 -> 52
```

2. (Afficher l'écart) Ecrire une fonction afficherEcart(a, b) qui affiche le message :

```
L'écart entre a et b est de e
```

où a, b et e sont remplacées par les valeurs adaptées. Voici quelques exemples de comportement de la fonction afficherEcart :

```
82 , 42 -> L'écart entre 82 et 42 est de 40
1970 , 2038 -> L'écart entre 1970 et 2038 est de 68
42 , 42 -> L'écart entre 42 et 42 est de 0
42 , -10 -> L'écart entre 42 et -10 est de 52
```

3. (Proximité de 81) Ecrire une fonction `plusProcheDe81(a, b)` qui renvoie l'entier le plus proche de 81 parmi a ou b. La fonction devra obligatoirement utiliser la fonction `ecart`. Si les deux nombres sont à égale distance de 81, la fonction retournera indifféremment l'un ou l'autre. Voici quelques exemples de comportement de la fonction `plusProcheDe81` :

```
42 , 100 -> 100
80 , 82 -> 82
42 , 42 -> 42
```

Volume d'un cylindre

On utilisera π en l'important du module standard `math`.

1. Écrire une fonction `aire_disque` qui prend en paramètre un nombre `r` et renvoie l'aire du disque de rayon r, définie par $S = \pi r^2$. Tester la fonction pour un rayon valant 10.
2. On rappelle que le volume V d'un cylindre de hauteur h et de base un disque d'aire S vaut $V = S \times h$.
   Écrire une fonction `volume_cyl` calculant le volume d'un cylindre et qui prend en paramètres
   - un nombre `r` désignant le rayon de la base
   - un nombre `h` désignant la hauteur du cylindre

   La fonction `volume_cyl` devra impérativement utiliser la fonction `aire_disque`. Tester la fonction pour un rayon valant 10 et une hauteur valant 2.
3. *Cette question est indépendante de ce qui précède.* Ecrire une fonction `afficher_litres` qui prend en paramètre un nombre `v` et qui se contente d'afficher le volume correspondant en litres sous la forme suivante :

   ```
   Volume : v litres
   ```

   Par exemple, l'appel `afficher_litres(42.1)` doit afficher exactement ceci :

   ```
   Volume : 42.1 litres
   ```

4. *Votre réponse à cette question doit utiliser les fonctions définies dans les questions précédentes et doit tenir sur une seule ligne. Cette question ne demande pas de définir de nouvelle fonction.*
   On dispose de deux cuves cylindriques remplies de jus de pommes :
   - le premier cylindre a une hauteur de 12 décimètres et une base de rayon de 4 décimètres,
   - le second cylindre a une hauteur de 8 décimètres et une base de rayon de 2 décimètres.

   Ecrire, en utilisant la fonction `afficher_litres`, un code qui affiche le volume de jus de pommes contenu au total dans les **deux** conteneurs, exprimé en litres. Pas de panique, il n'y a aucune conversion d'unités à faire puisque 1 décimètre cube correspond a un volume de 1 litre.

Calcul d'une durée

1. Écrire une fonction `hms_to_s(h, m, s)` qui retourne en secondes une durée exprimée en heures, minutes et secondes. Vérifier que 1h30m = 5400s et 3h20m15s = 12015s.
2. Écrire une fonction `s_to_hms(sec)` qui prend en paramètre une durée exprimée en secondes et la retourne exprimée en heures, minutes et secondes sous forme de liste [h, m, s]. Vérifier que `s_to_hms(5400)` retourne la liste `[1, 30, 0]`, et que `s_to_hms(12015)` retourne la liste `[3, 20, 15]`.
3. Écrire une fonction `afficher_temps` capable d'afficher une liste [h, m, s] sous la forme `h heures m minutes et s secondes`. Par exemple, `afficher_temps([2, 42, 16])` affiche `2 heures 42 minutes et 16 secondes`.
4. Sans définir de nouvelle fonction, écrire un code, tenant de préférence sur une seule ligne, qui affiche le temps écoulé comme à la question précédente entre 13h58m et 15h31m30s.

Arrondi au quart de point

Voici des exemples d'arrondi de note au quart de point le plus proche :

```
12.66     -->     12.75
12.9     -->     13
18.11     -->     18.0
8.5     -->     8.5
9.88     -->     10
13.25     -->     13.25
```

La fonction suivante prend en paramètre un nombre qui représente une note et renvoie la note qui est son arrondi au quart de point le plus proche :

```
def arrondir(note):
    pe=int(note)
    pf=note-pe
    if 0<=pf<0.125:
        r=0.
    elif 0.125<=pf<0.375:
        r=.25
    elif 0.375<=pf<0.625:
        r=.50
    elif 0.625<=pf<.875:
        r=.75
    else:
        r=1
    return pe+r
```

Cette fonction doit être présente dans le code une fois pour toutes et ne doit pas être modifiée. Il est attendu que vous *appeliez* la fonction `arrondir` quand vous en aurez besoin. Il N'est PAS attendu que vous fassiez des copier-coller du code ci-dessus pour servir de base à de nouvelles fonctions.

1. (Tester arrondir) Tester la fonction `arrondir` sur les notes suivantes, (une par une, sans les placer dans une liste) :

   ```
   12,66    12,9    18,11    8,5    9,88    13,25
   ```

2. (Calculer une moyenne) *Cette question est indépendante de ce qui précède.* Écrire une fonction `moyenne` qui calcule la moyenne d'une liste `L` de notes donnée en paramètre. Par exemple, si L est la liste `[12.66, 12.9, 18.11, 8.5, 9.88, 13.25]` alors, la moyenne vaut 12.55

3. *Cette question est indépendante de ce qui précède.* Écrire une fonction `afficherNote` qui affiche une note `N` sur 20 sous le format suivant :

   ```
   Votre note : N / 20
   ```

   suivi du message ADMIS ou NON ADMIS suivant que la note est supérieure ou égale à 10 ou pas. La fonction ne renvoie rien.
   Tester la fonction.
   Par exemple, afficherNote(12.55) doit afficher les deux lignes suivantes :

   ```
   Votre note :  12.55 / 20
   ADMIS
   ```

4. (Arrondir une liste de notes) *Cette question utilise uniquement la fonction* `arrondir`. Ecrire une fonction `arrondirListe` qui prend une liste de notes en paramètre et renvoie une

nouvelle liste constituée des notes arrondies au quart de point le plus proche. Voici un exemple du comportement de la fonction

```
[12.66, 12.9, 18.11, 8.5, 9.88, 13.25] ->
[12.75, 13, 18.0, 8.5, 10, 13.25]
```

(5) (Synthèse de tout ce qui précède) *Cette question ne demande pas de définir de nouvelle fonction.*
Votre réponse à cette question doit utiliser les fonctions définies dans les questions précédentes.
Soit une liste de notes, par exemple [12.66, 12.9, 18.11, 8.5, 9.88, 13.25]. Écrire un code qui

- calcule la moyenne de la liste de notes arrondies au quart de point le plus proche
- arrondit cette moyenne au quart de point le plus proche
- affiche uniquement la moyenne arrondie ainsi que le message adéquat ADMIS ou NON ADMIS.

Le code doit tenir sur une seule ligne.
Par exemple, pour la liste ci-dessus, le code doit afficher :

```
Votre note :  12.5 / 20
ADMIS
```

Formule de Keith et Craver

La formule de Keith et Craver permet de déterminer le jour de la semaine (ie lundi, mardi, etc) correspondant à une date donnée (par exemple, le 14 juillet 1789 qui était un mardi).

Ci-dessous, en voici une implémentation sous forme de fonction Python. La fonction `kc` retourne sous forme d'entier (`1 = lundi, 2 = mardi, ..., 7 = dimanche`) le jour de la semaine correspondant à une date passée en paramètre comme suit : `j` (jour), `m` (mois) et `a` (année).

```
def kc(j, m, a):
    z = a - (m<3)
    return (j + 23*m//9 + 3 -2*(m>=3) + a + z//4 - z//100 + z//400)%7 +1
```

Voici un exemple d'utilisation :

```
def kc(j, m, a):
    z = a - (m<3)
    return (j + 23*m//9 + 3 -2*(m>=3) + a + z//4 - z//100 + z//400)%7 +1

# Test du jour de la semaine du 14 juillet 2018
print(kc(14, 7, 2018))
```

```
6
```

— Lignes 6 et 7 : le 14 juillet 2018 est un samedi (6^{e} jour de la semaine).

Cette fonction doit être utilisée telle quelle, sans chercher à comprendre comment elle fonctionne. Vous devez réécrire (copier-coller) une fois et une seule cette fonction pour pouvoir l'utiliser par la suite mais il est inapproprié de copier/coller le **corps** de la fonction lignes 2 et 3 dans votre propre code. Il est seulement attendu d'**utiliser** la fonction `kc`.

(1) Vérifier la validité des dates suivantes en calculant leur code entre 1 et 7 :

- dimanche 13 janvier 2019
- mardi 14 juillet 1789
- dimanche 10 mai 1981
- jeudi 16 juillet 1998
- mardi 19 janvier 2038 (le bug de l'an 2038)

(2) En utilisant un appel à la fonction `kc`, écrire une fonction `est_vendredi13(m,a)` qui renvoie True si le 13 du mois m et de l'année a est un vendredi (et False sinon). Combien y-a-t-il de vendredis 13 dans l'année 2018 ?

(3) Ecrire et tester une fonction `jour_semaine` qui accepte en argument une **liste** `date`, supposée de la forme `[jour, mois, année]` et qui **retourne** le jour de la semaine correspondant en toutes lettres (ex : *lundi*, *mardi*, .., *dimanche*).
Par exemple, `jour_semaine([14, 7, 1789])` vaut la chaîne `mardi`.
On pourra utiliser une liste `JOURS_SEMAINE` formée des noms des jours de la semaine.

Fractions

Dans cet exercice, une fraction sera représentée par une liste de 2 entiers positifs :

`[numérateur, dénominateur]`.

Par exemple, la fraction $\frac{22}{7}$ sera représentée par la liste `[22, 7]`. Plus généralement, la fraction $\frac{a}{b}$ sera représentée par la liste `[a, b]`.

(1) Écrire une fonction `aff` qui affiche une fraction `frac` donnée en paramètre. Par exemple, `aff([22,7])` affichera `22 / 7`.

(2) Écrire une fonction `add(frac1, frac2)` qui retourne la somme des fractions `frac1` et `frac2`. On appliquera la formule suivante : $\frac{a}{b}+\frac{c}{d} = \frac{ad+bc}{bd}$. Par exemple, `add([1,3],[7,10])` retournera `[31,30]`. On ne cherchera pas à obtenir une fraction simplifiée (autrement dit une fraction irréductible).

(3) Écrire une fonction `harmonique(n)` qui calcule la somme $1 + \frac{1}{2} + ... + \frac{1}{n}$. Vérifier que `harmonique(6)` affiche `1764 / 720`.

(4) Écrire une fonction `simplifier(frac)` qui simplifie la fraction `frac` reçue en paramètre. Par exemple, `simplifier([140, 100])` doit renvoyer `[7,5]`. On utilisera la fonction `gcd` importée du module standard `math` afin de simplifier la fraction.
Si vous essayez de simplifier `harmonique(12)`, vous allez constater que l'exécution va être extrêmement longue.
Quitte à récrire légèrement le code de la fonction `add`, parvenir à accélérer le calcul de `harmonique(12)`

Le moins présent

On donne une liste `L` d'entiers et on demande de déterminer l'élément **le moins présent** de la liste `L`. Par exemple, si `L` est la liste :

```
[11, 12, 12, 11, 11, 10, 11, 12, 11, 10]
```

alors l'élément le moins présent est 10 (deux fois, les autres étant présents au moins trois fois). Si deux éléments sont aussi peu présents l'un que l'autre, l'élément à donner est celui qui apparaît le premier dans la liste. Par exemple, si `L` est la liste :

```
L=[13, 12, 10, 13, 11, 13, 12, 13, 11, 10]
```

l'élément demandé est 12, présent deux fois, comme 10 mais qui lui arrive après dans la liste.
Plus précisément, on écrira deux fonctions :

— `count(L, elt)` qui renvoie le nombre d'indices de la liste `L` où l'élément `elt` est présent
— `moins_present(L)` qui renvoie l'élément demandé et qui appellera la fonction précédente.

Pour chacune des deux fonctions ci-dessus, on fera un parcours de la liste avec une boucle `for`.
L'algorithme proposé n'est pas du tout optimal.

Périmètre d'un polygone

Un point `M` du plan sera vu comme une liste $[x, y]$ de deux entiers, son abscisse `x` et son ordonnée `y`. Par exemple, on peut considérer le point $M = [4, 7]$. On considère par ailleurs une liste `L` de `n` points du plan.
Par exemple, pour $n = 5$, on pourrait disposer de la liste suivante :

$$L = [[0, 0], [4, 0], [4, 7], [0, 4], [-3, 4]]$$

La liste `L` représente un polygone du plan. Les sommets du polygone sont donnés, dans l'ordre, par les éléments de la liste.

(1) Ecrire une fonction `dist(M, N)` qui renvoie la distance entre deux points `M` et `N`. On rappelle la formule :

$$MN = \sqrt{(x_N - x_M)^2 + (y_N - y_M)^2}.$$

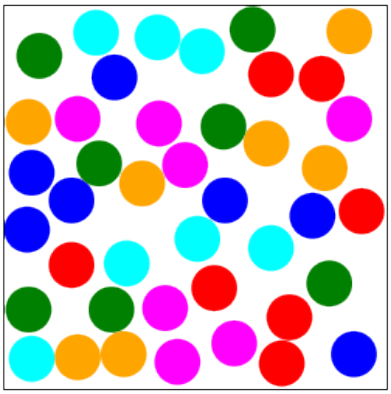
(2) Ecrire une fonction `perimetre(L)` qui renvoie le périmètre du polygone représenté par `L`. Par exemple, si `L` est une liste de 5 points `A`, `B`, `C`, `D` et `E` et représentant un polygone `ABCDE`, son périmètre est

$$AB + BC + CD + DE + EA.$$

Dans le cas de la liste `L` donnée en exemple ci-dessus, `perimetre(L)` devra renvoyer 24.
La fonction `perimetre` devra utiliser la fonction de la question 1.
Cet exercice est directement inspiré d'une question lue sur le forum Python du site Open-Classrooms.

Disques disjoints colorés

On se donne une domaine carré de taille `d` $\times$ `d` et on veut placer dans ce domaine `N` disques colorés et de même rayon `r` et qui soient deux à deux disjoints :



Les couleurs seront successivement choisies dans la liste suivante :

```
colors=["red", "green", "blue", "cyan", "orange", "magenta"]
```

Les ingrédients de l'algorithme pourront être les suivants :

— au fur et à mesure, on insère tout centre d'un nouveau disque dans une liste `L` initialement vide

— on répète `N` fois la génération d'un disque

— pour générer un disque, on choisit avec précaution un centre aléatoire, de coordonnées entières pour simplifier, dans le domaine (utiliser `randrange`)

— pour tester la validité du centre choisi, on s'assure que sa distance `d` à chacun des centres déjà construits vérifie $d > 2r$

— un centre aléatoire pouvant ne pas convenir, la recherche du centre sera placée dans une boucle `while`.

On écrira des fonctions :

— `dist(A, B)` calculant la distance entre deux points ;

— `isvalid(center, L, r)` fonction booléenne qui teste si le centre est assez éloigné des centres présents dans L ;

— `rd_disk(M, r)` qui génère un centre aléatoire

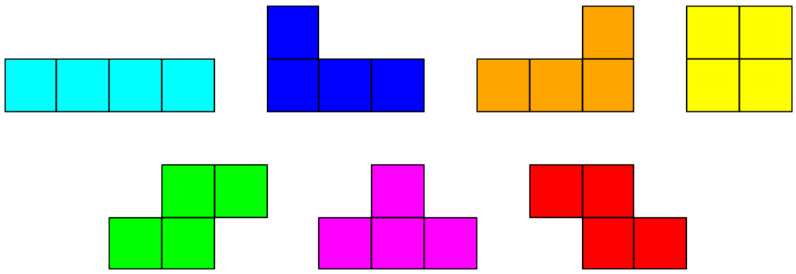
— `draw(M, r, N)` qui génère le dessin complet, y compris le rectangle englobant

Le dessin ci-dessus a été généré par l'appel `draw(250, 15, 42)`.

Cet exercice m'a été communiqué par mon collègue P. Piccinini.

Dessiner les tétraminos

On demande de dessiner la figure suivante :



Il s'agit des 7 pièces, appelées *tétraminos*, que le jeu Tétris utilise. Chaque pièce est constitué de 4 briques carrées (d'où le nom du jeu, Tétris). De haut en bas et de gauche à droite, le nom usuel des blocs et les couleurs utilisées sont données dans tableau suivant :

Nom usuel	Couleur standard
I	cyan
J	bleu
L	orange
O	jaune
S	citron vert (*lime*)
T	violet
Z	rouge

On écrira une fonction `square(tlc, side, color)` qui dessine une brique carrée connaissant

— la variable `tlc` (*top left corner*) désignant le sommet en haut à gauche du carré,

— le côté du carré

— la couleur de la pièce.

La fonction `square` fera appel à la fonction `Rectangle` de Matplotlib ; on utilisera l'option `edgecolor='black'` qui tracera en noir le contour du carré et l'option `facecolor` qui colorie l'intérieur du carré.

Toute succession rectiligne d'au moins deux carrés utilisera une boucle `for`. Le bloc `O` utilisera deux boucles `for` imbriquées.

Date du jour de Pâques

L'algorithme de Butcher-Meeus permet de calculer le jour et le mois du dimanche de Pâques d'une année donnée, à partir de l'an 1583.

Ecrire une fonction `jourPaques(y)` qui renvoie une liste `[jour, mois]` représentant la date du dimanche de Pâques de l'an `y`.

Par exemple, `jourPaques(2042)` doit renvoyer `[6, 4]` pour le 6 avril.

On appliquera pas à pas l'algorithme tel qu'expliqué sur la page Wikipedia.

Liste croissante

Avant de résourdre cet exercice, on pourra consulter la partie du cours expliquant comment on peut interrompre une boucle for avec `return`.

Ecrire une fonction `est_croissante` qui détermine si une liste d'entiers passée en paramètre est croissante (i.e. renvoie True) et False sinon.

Qu'une liste `L` soit croissante signifie que $\texttt{L[i]} \leq \texttt{L[i+1]}$ pour tout indice `i` pour lequel l'inégalité a un sens.

Exemples :

```
[5, 6, 6, 6, 10] -> True
[5, 6, 6, 6, 4] -> False
[42] -> True
```

Toujours la moyenne

Écrire une fonction `toujoursLaMoyenne` qui prend en paramètre une liste de notes (sur 20) et renvoie True si toutes les notes valent 10 ou plus que 10. Le code sera pénalisé s'il n'utilise pas une boucle `while` pour résoudre le problème.

Voici des exemples de comportements attendus :

```
[12, 19, 10, 18, 20] -> True
[10] -> True
[12, 15, 9, 19] -> False
```

Pairs d'abord, impairs ensuite

On donne une liste `L` d'entiers et on demande d'écrire une fonction `f` qui renvoie `True` si dans la liste `L` apparaissent D'ABORD les entiers pairs de la liste et ENSUITE les entiers impairs de la liste.

Voici quelques exemples de comportements de `f` :

L	Pairs puis impairs ?	Commentaire
`[12, 82, 81, 9, 31]`	`True`	D'abord 12, 82 (pairs) puis les impairs
`[81, 9, 31]`	`True`	Que des impairs
`[12, 82]`	`True`	Que des pairs
`[12, 82, 81, 9, 46, 31]`	`False`	9 (impair) est suivi d'un pair (46)

Entiers consécutifs croissants ou pas

Ecrire une fonction `f` qui prend en paramètre une liste `L` d'entiers et qui renvoie `True` si la liste est constituée d'entiers consécutifs [5] croissants et qui renvoie `False` sinon. Ci-contre, voici quelques exemples de comportements attendus de `f`.

5. Entiers *consécutifs* signifie : entiers qui se suivent avec 1 d'écart, comme 10,11 et 12.

L	entiers consécutifs croissants
`[81, 82, 83]`	`True`
`[82, 81, 83]`	`False`
`[2013, 2038, 3000]`	`False`
`[81]`	`True`

Liste monotone

Une liste d'entiers est dite *monotone* ou bien si elle est croissante ou bien si elle est décroissante.

Par exemple, les listes `[5, 5, 6, 6, 6, 10]` ou `[5, 5, 4, 1, 1]` sont monotones mais la liste `[42, 35, 35, 40, 34]` ne l'est pas.

Ecrire une fonction `est_monotone` qui détermine si une liste d'entiers est monotone (i.e. renvoie `True` et `False` sinon).

Afin de traiter uniformément les deux cas de listes croissantes ou décroissantes, on écrira une fonction booléenne `meme_signe(a, b)` qui renvoie `True` si `a` et `b` sont des entiers de même signe et `False` sinon. On considèrera que l'entier 0 a deux signes à la fois.

On fera attention à bien gérer le cas des listes dont les premiers termes sont identiques.

Exemples :

```
[5, 5, 6, 6, 6, 10] -> True
[5, 5, 6, 6, 6, 10, 9] -> False
[5, 5, 6, 6, 6, 10, 10, 9] -> False
[5, 5, 6, 6, 6, 6] -> True
[5, 5, 6, 6, 6, 6, 4, 4] -> False
[5, 5, 6, 6, 6, 5] -> False
[42, 5, 5, 6, 6, 6, 5] -> False
[42] -> True
[42, 42, 42, 42, 42] -> True
[5, 4] -> True
[4, 5] -> True
[5, 4, 3, 3, 0] -> True
[5, 4, 3, 42, 43, 44] -> False
[5, 4, 3, 3, 42] -> False
[] -> True
```

Suppression alternée d'entiers dans une liste

On écrit côte à côte tous les entiers entre 1 et 128. On supprime un élément sur deux en commençant par le premier, ce qui donne la liste suivante :

```
2, 4, 6, 8, 10, ..., 124, 126, 128
```

On répète l'opération de suppression mais en commençant cette fois par le dernier élément, ce qui donne la liste :

```
2, 6, 10, ..., 122, 126
```

et on recommence l'opération à partir du début et ainsi de suite en alternant début et fin de liste jusqu'à ce que la liste ne contiennne plus qu'un seul nombre. On demande de déterminer

ce nombre (on trouvera 86).

Pour cela, on écrira :

— une fonction `supprimer(L)` qui renvoie la liste des éléments d'une liste `L` dont on a supprimé le premier, le troisième et tous les éléments placés à un rang impair,

— une fonction `inverser(L)` qui renvoie la liste des éléments de `L` écrits dans l'ordre inverse

puis on appelera ces fonction dans une boucle `while`.

Cette question a été lue sur le forum MathStackExchange : alternate deletion of integers between 1 and 128, which is the last one?

Suite unimodale

Observez la suite suivante :

```
12 15 17 18 19 20 17 13 10
```

La suite commence par augmenter strictement pour atteindre sa plus grande valeur puis elle décroît strictement.

Une telle suite sera dite *unimodale* : c'est donc une suite d'entiers qui atteint sa plus grande valeur en une unique valeur M (dans l'exemple ci-dessus, c'est $M = 20$) et qui est croissante au sens strict avant d'atteindre M et décroissante au sens strict après avoir atteint M.

La suite ci-dessous n'est pas unimodale :

```
12 15 17 17 18 19 20 24 25 29
```

en effet, elle prend la même valeur (17) sur deux indices consécutifs.

De même, la suite ci-dessous n'est pas unimodale

```
12 15 17 14 13 20 21 19 11
```

puisqu'elle croit puis décroît puis croît encore.

Enfin la suite ci-dessous n'est pas unimodale

```
12 15 17 18 19
```

car elle ne décroît jamais.

Vous disposez d'une suite d'au moins trois entiers sous forme d'une liste `L`, par exemple

```
L = [12, 15, 17, 18, 19, 20, 17, 13, 10]
```

et vous devez construire une fonction `estUnimodale(L)` qui renvoie `True` si la suite des entiers de `L` est unimodale et `False` sinon. Dans le cas de la liste `L` donnée en exemple, `estUnimodale(L)` vaut `True`.

Suite super croissante

Soit S un série croissante de données composée de valeurs réelles. On dira que S est une *super croissante* si la suite des écarts entre termes consécutifs est strictement croissante. Par exemple, la suite ci-dessous

```
1 3 7 12
```

est super croissante puisque les écarts consécutifs valent 2, 4 et 5 et donc vont en croissant strictement. En revanche, la suite

```
1 3 4 7
```

N'est PAS super croissante puisque les écarts consécutifs valent 2, 2 et 3 et ne vont pas en croissant STRICTEMENT.

Plus précisément, en tout indice `i` de la série où cela a un sens, on examine si l'écart entre la i^e valeur et la $(i-1)^e$ valeur est bien strictement inférieur à l'écart entre la $(i+1)^e$ valeur et la i^e valeur.

Écrire une fonction `est_super_croissante` qui retourne `True` si la série S passée en paramètre, et supposée être croissante, est une super croissance, et `False` sinon.

Tester avec les trois séries suivantes :

```
[0.0, 1.0, 2.30, 3.74, 5.28, 6.90, 8.59, 10.33, 12.13, 13.97] --> True
[-278.0, -274.0, -250.14, -191.30, -83.99, 84.39, 325.79, 651.68, 1073.18, 1601.05] --
[-9.0, -2.0, 3.62, 8.81, 13.74, 18.49, 23.10, 27.60, 31.99, 36.31] --> False
```

Deuxième plus petit élément d'une liste

1. Ecrire une fonction `mini` qui prend en paramètre une liste non vide `L` d'entiers et renvoie le plus petit élément de la liste `L`.
   Voici quelques exemples de comportement de la fonction `mini` :

```
[81, 42, 65, 12, 81] -> 12
[42, 81, 65] -> 42
[81, 81, 81] -> 81
[81] -> 81
```

2. Écrire une fonction `tousSaufLui` qui prend en paramètres :
   - une liste `L` d'entiers
   - un entier `a`

   et qui renvoie une *nouvelle* liste comprenant tous les éléments de `L` sauf ceux qui valent `a`. Le contenu de la liste initiale `L` doit être préservé après un appel à la fonction `tousSaufLui`.
   Voici quelques exemples de comportement de `tousSaufLui(L, a)` pour `a = 81` et les listes `L` suivantes :

```
[65, 81, 31, 81, 9, 81, 81, 32] -> [65, 31, 9, 32]
[65, 31, 9, 32] -> [65, 31, 9, 32]
[81, 81, 81] -> []
```

3. Écrire une fonction `deuxiemePlusPetit` qui prend en paramètre une liste d'entiers `L` ayant au moins deux éléments distincts et qui renvoie la plus petite valeur de la liste `L` mais qui soit différente du minimum de la liste `L` autrement dit la fonction doit renvoyer la deuxième valeur de `L` si `L` était triée dans l'ordre croissant.
   Voici quelques exemples de comportement de la fonction `deuxiemePlusPetit` :

```
[8, 20, 8, 16] -> 16
[8, 9, 8, 7, 9, 8, 5] -> 7
[4, 4, 4, 3] -> 4
[81, 42] -> 81
```

   Votre fonction devra utiliser les fonctions précédentes.

Deux plus grands éléments d'une liste

On donne une liste d'entiers `L`, ayant au moins deux éléments et on demande d'écrire une fonction `deux_premiers(L)` qui renvoie la liste `M` formée des deux plus grands entiers de `L`. La liste `M` sera ordonnée dans le sens croissant. Il est possible que les deux entiers de `L` soient identiques (lorsque le plus grand élément de `L` est présent au moins deux fois dans `L`). Voici quelques exemples de comportements :

```
[4, 3, 6, 7, 7, 8, 8, 7, 4, 9] → [8, 9]
[4, 5] → [4, 5]
[5, 4] → [4, 5]
[5, 4, 3, 2, 5, 5] → [5, 5]
```

La méthode de codage de la fonction est libre, quitte à utiliser une fonction auxiliaire.

On pourra néanmoins suivre les indications suivantes :

- créer deux variables `a` et `b` qui initialement représentent les éléments `L[0]` et `L[1]` avec la contrainte que `a` $\leq$ `b`
- parcourir `L`
- à chaque nouvel élément de `L`, mettre à jour les variables `a` et `b` pour qu'elles représentent les deux plus grands éléments de `L` parmi ceux qui ont déjà été examinés lors du parcours.

Deux plus grands éléments distincts d'une liste

On donne une liste d'entiers `L` comprenant au moins deux entiers distincts et on demande d'écrire une fonction `deux_premiers(L)` qui renvoie la liste `M` formée des deux plus grands entiers *distincts* de `L`. La liste `M` sera ordonnée dans le sens croissant. Voici quelques exemples de comportements :

```
[4, 3, 6, 7, 7, 8, 8, 7, 4, 9] → [8, 9]
[4, 5] → [4, 5]
[5, 4] → [4, 5]
[5, 4, 3, 2, 5, 5] → [4, 5]
```

La méthode de codage de la fonction est libre, quitte à utiliser une fonction auxiliaire.

On pourra néanmoins suivre les indications suivantes :

- créer deux variables `a` et `b` qui initialement représentent les éléments `L[0]` et `L[1]` avec la contrainte que `a` $\leq$ `b`
- parcourir `L`
- à chaque nouvel élément de `L`, mettre à jour les variables `a` et `b` pour qu'elles représentent les deux plus grands éléments de `L` parmi ceux qui ont déjà été examinés lors du parcours et faire en sorte que ces éléments soient distincts si c'est possible.

Les plus petits devant

On donne une liste `L` d'entiers et on demande de **modifier** `L` pour faire en sorte que chaque occurrence du plus petit élément de `L` soit placée en début de liste. Par exemple, soit la liste `L`

```
L = [50, 20, 30, 20, 20, 10, 50, 10, 70, 10, 80, 10]
```

Son plus petit élément vaut 10 et apparaît 4 fois, aux indices 5, 7, 9 et 11. On doit donc modifier la liste L en la liste :

```
[10, 10, 10, 10, 20, 50, 50, 20, 70, 30, 80, 20]
```

Il est attendu que L soit juste modifiée (par des échanges d'éléments) et non pas qu'une nouvelle liste soit reconstruite. Toutefois, créer une nouvelle liste est préférable à un code qui ne marche pas ou à une absence de code.

On pourra parcourir L et maintenir une liste des **indices** des éléments minimaux de L, quitte à repartir d'une nouvelle liste si le minimum courant change. Il n'est pas nécessaire de parcourir plusieurs fois la liste L mais si c'est nécessaire, cette solution sera préférable à une absence de code ou à un code qui ne fonctionne pas.

Une fois la liste des indices obtenue, on procédera aux échanges demandés. On pourra utiliser la fonction d'échange suivante :

```
def echanger(L, i, j):
    # échange les termes d'indices i et j de L
    L[i],L[j]=L[j], L[i]
```

Nombre d'éléments distincts d'une suite croissante

On donne une liste croissante d'entiers, par exemple

```
L = [42, 42, 81, 82, 82, 82, 82, 89, 92, 98]
```

et on demande de calculer le nombre d'éléments distincts de la liste. Dans l'exemple précédent, ce nombre est 6. On écrira une fonction `effectif(L)` qui renverra le nombre d'éléments distincts de L. Utiliser deux boucles `for` imbriquées sera pénalisé.

Voici quelques exemples de comportement de la fonction :

```
[42, 42, 81, 82, 82, 82, 82, 89, 92, 98] -> 6
[42, 42, 42, 42, 42, 43, 43, 43, 43, 43] -> 2
[42] -> 1
[42, 42, 42, 42, 42] -> 1
[] -> 0
[10, 20, 30, 40, 50, 60, 70] -> 7
[42, 81, 81, 81, 81, 90] -> 3
```

Intersection de deux listes strictement croissantes

On donne deux listes L et M constituées d'entiers rangés dans l'ordre strictement croissant, en particulier, dans chaque liste, les entiers sont distincts. On demande de construire la liste N formée des entiers communs aux deux listes. Les éléments de N seront écrits dans l'ordre croissant. On écrira une fonction `inter(L, M)` qui renverra N.

Voici un exemple de comportement :

```
L = [12, 13, 14, 18, 21, 22, 24, 26, 28]
M = [10, 13, 15, 19, 21, 22, 26, 29]
N = [13, 21, 22, 26]
```

Votre méthode de génération de N **doit** utiliser que les listes L et M sont croissantes. Le code devra fonctionner pour des listes de tailles très grandes (de l'ordre du million ou du milliard

d'éléments). Pour tester la fonction `inter` sur des listes de taille de l'ordre du million, on utilisera le code suivant :

```
def perf(n):
    from random import sample
    from time import perf_counter
    R=range(n)
    L =sorted(sample(R, n//4))
    M=sorted(sample(R, n//3))
    begin_perf=perf_counter()
    inter(L, M)
    print("%.2f" %(perf_counter()-begin_perf))


perf(4*10**6)
```

L'affichage devra survenir dans un temps de l'ordre d'une seconde.

Liste d'entiers symétrique par rapport à zéro

On dit qu'une liste d'entiers est « symétrique par rapport à zéro » si ses éléments non nuls peuvent se partionner en paires formées d'éléments opposés ie de la forme x et $-x$. Par exemple, chacune des trois listes ci-dessous est opposée :

```
[-5, 0, 3, 5, -3]
[7, -1, -9, -7, 9, 1]
[-5, 0, 0, 4, 5, 0, -4]
```

Ecrire une fonction `estSym(L)` qui partant d'une liste `L` d'entiers renvoie un booléen qui vaut `True` si la liste `L` est symétrique par rapport à zéro et `False` sinon. Tester avec les listes ci-dessus.

Eléments majoritaires d'une liste

On donne une liste `L` dont les éléments sont des **chiffres entre 0 et 9**, par exemple

`L = [5, 4, 8, 5, 5, 2, 8, 1, 8, 2, 7, 1].`

Soit `N` le nombre d'occurrences maximal parmi les éléments de `L`. Par exemple, avec la liste ci-dessus, on a `N` = 3 puisque

— aucun élément de la liste n'apparaît plus de 3 fois dans `L`

— au moins un élément de `L` apparaît exactement 3 fois, par exemple l'élément 8.

On demande d'écrire une fonction `majoritaires(L)` qui prend en paramètre une liste `L` de chiffres et qui renvoie la liste `M` des éléments distincts de `L` dont exactement `N` occurrences sont présentes dans la liste `L` où `N` est, comme expliqué ci-dessus, le nombre d'occurrences maximal parmi les éléments de `L`. Ainsi, avec `L = [5, 4, 8, 5, 5, 2, 8, 1, 8, 2, 7, 1]`, la liste à trouver est `M = [5, 8]` car les éléments 5 et 8 de la liste `L` apparaissent chacun le nombre maximal `N` de fois.

Suggestion de résolution. On pourra créer une liste auxiliaire `nbOccur` de longueur 10 et telle que `nbOccur[i]` vaille le nombre d'occurrences de l'entier `i` dans la liste `L`.

Le tableau ci-dessous donne d'autres exemples du comportement attendu de la fonction `majoritaires` :

L	Nombre N maximal d'occurrences	Liste M des occurrences présentes N fois
[1, 0, 9, 2, 9, 9, 0, 5, 4]	3	[9]
[0, 6, 1, 0, 6, 3, 4, 4, 2, 4, 1, 9, 6]	3	[4, 6]
[6, 3, 2, 9]	1	[2, 3, 6, 9]
[7]	1	[7]

Supprimer les doublons

Ecrire une fonction `eliminer_doublons(L)` qui partant d'une liste d'entiers retourne une liste composée des valeurs de `L`, mais dans laquelle les multiples valeurs égales de `L` (s'il y en a) n'ont été copiées qu'une seule fois. L'ordre de la liste renvoyée devra repecter l'ordre d'entrée des éléments de la liste initiale. On dira ainsi de la liste retournée qu'elle ne contient pas de doublon. Exemples :

Liste	Liste sans doublons
`[42, 81, 42, 65, 12, 81, 31, 42]`	`[42, 81, 65, 12, 31]`
`[42, 42, 42, 42, 42]`	`[42]`
`[42]`	`[42]`

On rappelle que Python dispose d'un test d'appartenance dans une liste.

Ils sont uniques !

On donne une liste `L` d'entiers et on demande d'écrire une fonction `solitaires(L)` qui renvoie la liste `sansDoublons` des éléments de `L` qui ne sont présents qu'une seule fois dans `L`.

Voici quelques exemples de comportement :

```
[2, 2, 2, 2, 1, 5, 2, 3, 5, 4] → [1, 3, 4]
[42, 42, 42, 42, 42] → []
[42] → [42]
```

Ainsi, pour la liste `L=[2, 2, 2, 2, 1, 5, 2, 3, 5, 4]`, la liste `sansDoublons` ne contient pas 5 qui est présent deux fois, ni 2 présent cinq fois.

On pourra procéder comme suit :

- initialiser une liste `eff`, de même longueur que `L`, des effectifs de la liste ; au départ `eff` ne contient que des zéros et à la fin de sa construction, est telle que `eff[i]` est le nombre d'apparitions dans `L` de l'élément de `L` à l'indice `i` ;
- construire `eff`, en parcourant la liste `L` et en comptant le nombre de fois que l'élément courant est présent parmi les éléments déjà examinés ;
- construire la liste des éléments d'indice `i` dans `L` tels que `eff[i]` vaille 1.

Doublons d'une liste croissante

On donne une liste `L` d'entiers écrits **dans l'ordre croissant** et on demande de déterminer la liste `D` des éléments qui apparaissent au moins deux fois dans cette liste. Voici quelques exemples :

```
[1, 1, 2, 3, 3, 3, 3, 3] -> [1, 3]
[1, 1, 1, 1, 1, 1, 1] -> [1]
[2, 2, 4, 4] -> [2, 4]
[1, 1, 1, 2, 2, 2, 2, 2, 2] -> [1, 2]
[1, 2, 3, 4, 4] -> [4]
[2] -> []
[3, 4, 4, 5, 5] -> [4, 5]
[1, 1, 1, 4, 5, 6] -> [1]
[2, 2, 3, 3, 4, 5, 6, 6, 7] -> [2, 3, 6]
[1, 2, 3] -> []
```

On procédera de la manière suivante : parcourir la liste `L` et à certaines étapes, si l'élément courant est identique au précédent, on placera l'élément courant dans la liste `D`, vide au début.

Lancers de dé

1. Ecrire une fonction `lancer_un_de` sans paramètre et qui simule un coup de dé à 6 faces.
2. Ecrire une fonction `lancers` qui effectue n lancers de dés en utilisant la fonction `lancer_un_de` et renvoie une liste `resultats` de 7 éléments tels que si i est un entier entre 1 et 6 alors `resultats[i]` est le nombre de fois que le dé à sorti la valeur i. Le premier élément de la liste `resultats` sera arbitrairement fixé à 0.
   Tester pour n valant 1 million.

Toujours plus

On appelle *Toujours Plus* le jeu utilisant un seul dé et dont la règle est la suivante : on répète le lancer du dé jusqu'à ce que la valeur lue sur le dé diminue strictement par rapport au lancer précédent. Autrement dit, si le lancer de dé est supérieur ou égal au lancer précédent, la partie continue, sinon, la partie est terminée.

Voici trois exemples de parties complètes de *Toujours Plus* :

```
2 4 5 5 2
4 1
5 5 5 6 6 6 5
```

En particulier, une partie de *Toujours Plus* dure au moins 2 coups.

1. Écrire une fonction `jouerToujoursPlus`, sans paramètre et qui simule une partie de *Toujours Plus*. La fonction devra renvoyer la liste des lancers successifs de la partie.
2. Simuler une partie 100000 fois et vérifier qu'une partie dure, en moyenne, 3 lancers.

Balles dans des boîtes

On dispose de n balles et de n bacs (dans ce qui suit on prendra $n = 100000$). On lance au hasard N fois (par exemple $N = 100$) chacune des balles dans un des bacs. En moyenne, quel est le nombre de balles dans le bac le plus rempli ?

Ce problème est connu sous le nom de Balls into Bins et permet d'illustrer le problème des collisions dans une table de hachage.

FACE quatre fois de suite

1. Ecrire une fonction `lancer_piece` qui simule le lancer d'une pièce de monnaie équilibrée. La fonction retournera la chaîne "PILE" ou la chaîne "FACE".

(2) On dispose d'une pièce de monnaie. On cherche à savoir quel est, en moyenne, le minimum de lancers qu'il faut effectuer pour obtenir 4 fois *FACE* **consécutivement**.
Créer une fonction `attente` qui effectue une suite de lancers de la pièce et qui renvoie le nombre de lancers qu'il aura fallu faire pour obtenir pour la première fois 4 fois FACE de suite.

(3) Ecrire une fonction `test` qui itère `n` fois l'expérience de la fonction `attente` la fonction `attente`. Vérifier qu'il faut en moyenne 30 lancers pour obtenir 4 fois de suite FACE.
Tester sur $n = 10^5$ expériences.

Suite alternée pile ou face

Soit l'expérience consistant à lancer une pièce équilibrée `n` fois et à observer si oui ou non, les résultats de succèdent de manière alternée, c'est-à-dire pile, face, pile, face, etc ou bien face, pile, face, pile, etc.

Ecrire une fonction `tirage_alterne(n)` qui simule l'expérience et renvoie `True` ou `False` selon qu'il y aura eu alternance ou pas.

Suite aléatoire croissante

Dans cet exercice, on utilisera le résultat suivant : étant donné une liste `L` (disons d'entiers), la méthode `sort` appliquée à la liste `L` a pour effet de modifier la liste `L` en la triant dans l'ordre croissant. Par exemple, vous pourrez tester le code suivant :

```
L=[2, 8, 6, 7, 2, 1]
print(L)
L.sort()
print(L)
```

qui affiche

```
[2, 8, 6, 7, 2, 1]
[1, 2, 2, 6, 7, 8]
```

Noter que le code suivant est inapproprié et incorrect car `L.sort()` ne renvoie aucune liste :

```
L=[2, 8, 6, 7, 2, 1]
print(L)
L=L.sort()
print(L)
```

Écrire une fonction `aleat_croissante(a, b, n)` qui prend en paramètres trois entiers `a`, `b` et `n` avec `a` $\leq$ `b` et `n` ≥ 0 et qui renvoie une liste croissante formée de `n` entiers aléatoires compris entre `a` et `b` (au sens large).

Voici quelques exemples d'appels de la fonction :

```
aleat_croissante(17, 19, 2) -> [17, 19]
aleat_croissante(16, 27, 1) -> [22]
aleat_croissante(18, 29, 8) -> [19, 19, 24, 24, 26, 26, 26, 29]
aleat_croissante(25, 26, 0) -> []
aleat_croissante(10, 31, 3) -> [14, 22, 24]
```

On répètera `n` fois le choix d'un entier aléatoire entre `a` et `b`

Jeu de dé : premier coup gagnant

Camille et Dominique lancent un dé à tour de rôle jusqu'à ce qu'un 6 sorte et la partie est gagnée. On convient que Camille commence la partie.

1. Simuler à l'aide d'une fonction `jouer()` le jeu précédent. La fonction renverra 1 si Camille gagne et 0 sinon.
2. En répétant le jeu 1000 fois déterminer la probabilité pour Camille de gagner le jeu (on devra trouver un nombre proche de `6/11`).

Tirage du loto (version fonction)

Écrire une fonction `tirerLoto` ne prenant aucun paramètre et qui renvoie un tirage aléatoire des 5 numéros d'un loto. On rappelle qu'un tirage de loto est formée de 5 numéros **distincts** entre 1 et 49. On ne tirera pas de « numéro Chance ».

Ainsi, l'appel `tirerLoto()` pourra générer une liste telle que `[42, 32, 48, 47, 20]` mais pas telle que `[42, 32, 48, 42, 20]` puisque dans cette dernière liste, le numéro 42 apparaît deux fois.

On utilisera une boucle `while` pour placer au fur et à mesure les numéros tirés dans une liste `L` initialement vide. On rappelle que si `L` est une liste d'entiers et `x` un entier alors l'expression `x in L` vaut `True` si l'entier `x` est dans la liste `L` et `False` sinon.

Jeu du 421

Le jeu du 421 consiste à obtenir avec 3 dés une combinaison contenant une fois le chiffre 4, une fois le chiffre 2 et une fois le chiffre 1. Par exemple, si vous jetez trois dés simultanément et que vous obtenez la combinaison 2, 1 et 4, vous avez gagné ; si vous obtenez 1, 4, 1 alors votre coup n'est pas gagnant.

L'exercice va demander combien de jets de 3 dés sont nécessaires, en moyenne, pour faire un 421.

1. Ecrire une fonction `est421` qui prend en paramètre une liste de trois entiers et renvoie
   - `True` si la liste, à la fois, contient 4, contient 2 et contient 1
   - `False` sinon.

   Ainsi :
   - `est421([6, 2, 1]) = False`
   - `est421([1, 1, 1]) = False`
   - `est421([2, 4, 1]) = True`
   - `est421([1, 2, 4]) = True`
   - `est421([4, 2, 1]) = True`
2. Ecrire une fonction `tirage421` qui renvoie, sous forme de liste, un tirage aléatoire de trois dés. Par exemple, si vous tirez les nombres

   ```
   6, 2, 2
   ```

   alors la fonction doit renvoyer la liste `[6, 2, 2]`. Les nombres peuvent avoir été tirés simultanément ou l'un après l'autre, c'est indifférent.
3. Ci-dessous, on appelle *suite gagnante* toute succession de lancers de trois dés, autant de fois que nécessaire, pour « sortir » 421. Par exemple, la suite de 7 lancers suivants est une suite gagnante :

```
523 144 643 235 451 252 214
```

Écrire une fonction `nombreSuitesGagnantes` qui ne prend aucun paramètre et qui

— simule une suite gagnante,

— renvoie le nombre de lancers de la suite gagnante

Par exemple, `nombreSuitesGagnantes` renvoie 7 dans le cas de la suite gagnante ci-dessus.

(4) On fait 1000 suites gagnantes. Calculer le nombre moyen de lancers d'une suite gagnante (le résultat théorique est 36).

Paradoxe des anniversaires

On souhaite vérifier expérimentalement que si un groupe d'individus est constitué d'au moins 23 personnes, alors il y a une chance sur deux que deux personnes du groupe aient le même jour d'anniversaire.

Une année sera constituée de 365 jours et un jour de l'année sera identifié par un numéro entre 1 et 365.

(1) Ecrire une fonction `groupe_aleat(n)` qui génère un liste de `n` dates d'anniversaire aléatoires. Par exemple, si `n=8` alors `groupe_aleat(n)` pourrait renvoyer une liste telle que :

```
[221, 82, 316, 26, 93, 82, 84, 213]
```

(2) *Cette question est indépendante de la précédente.* Ecrire une fonction `contientDoublon(L)` qui renvoie `True` si la liste `L`, formée d'entiers, contient un doublon et `False` sinon *(indication : utilisez deux boucles imbriquées).*

(3) Ecrire une fonction `freq_meme_jour(n, N)` qui génère `N` listes aléatoires de n jours de l'année et renvoie la proportion de groupes parmi les N groupes qui contiennent deux jours identiques. Par exemple, si `freq_meme_jour(8, 5)` générait les 5 listes de dates suivantes :

```
[221, 82, 316, 26, 93, 82, 84, 213]
[275, 95, 102, 235, 200, 92, 244, 297]
[136, 289, 204, 284, 1, 48, 358, 177]
[179, 63, 272, 134, 160, 319, 151, 272]
[78, 331, 193, 200, 254, 193, 118, 217]
```

alors `freq_meme_jour(8, 5)` renverrait 0.6 puisqu'on constate que sur les 5 groupes, il y en a exactement trois où il y a deux jours identiques (le premier groupe avec 82, le quatrième avec 272 et le dernier avec 193) d'où la proportion de $\frac{3}{5} = 0{,}6$.

Vous aurez besoin de la fonction `contientDoublon` de la question précédente. Si vous n'avez pas réussi à la coder, vous pourrez utiliser la fonction suivante (sans chercher à comprendre comment son code fonctionne) :

```
def contientDoublon(L):
    return len(L)!=len(set(L))
```

(4) Vérifier qu'à partir de 23 personnes, il y a au moins une chance sur deux que parmi les individus du groupe, deux personnes aient le même jour d'anniversaire (répéter à chaque fois l'expérience $N = 10000$ fois).

Mélanger une liste

Ecrire une fonction `melange(n)` qui renvoie une liste `M` formée de `n` entiers qui constituent un mélange aléatoire des entiers entre 0 et `n-1` inclus. Par exemple, si `n` = 5, l'appel `melange(n)` pourrait renvoyer la liste `M = [3, 1, 4, 0, 2]`.

On appliquera la méthode suivante, qui va être illustrée sur la génération de la liste `M=melange(5)` donnée en exemple. Voici les étapes :

- on tire aléatoirement un entier entre 0 et 4, par exemple 3 et on place cette valeur dans une liste `M` au départ vide. On a donc pour l'instant `M=[3]` ;
- on tire à nouveau un entier aléatoire entre 0 et 4, par exemple 1, et on complète la liste `M` avec la valeur tirée, ce qui donne `M=[3, 1]`.
- on recommence et on tire aléatoirement 3, qui a déjà été tiré à une étape précédendente. On ne rajoute pas ce tirage à `M` et on continue ;
- on tire encore aléatoirement encore un entier entre 0 et 4, on obtient par exemple 4 et on le rajoute à la liste `M` car il n'a jamais été tiré antérieurement, ce qui donne `M=[3, 1, 4]`.
- on continue ainsi jusqu'à ce que la liste `M` contienne 5 éléments.

On utilisera une boucle `while` et on rappelle qu'on peut tester l'appartenance d'un élément `x` à une liste `M` avec l'expression `x in M`.

Tester pour n valant 10, 100, 1000 ou 10000 (on devrait observer une certaine lenteur d'exécution).

Gagner au jeu de craps

Le jeu de craps se joue avec deux dés à 6 faces. Le joueur jette une première fois les deux dés. Soit `t` le total de deux faces sorties.

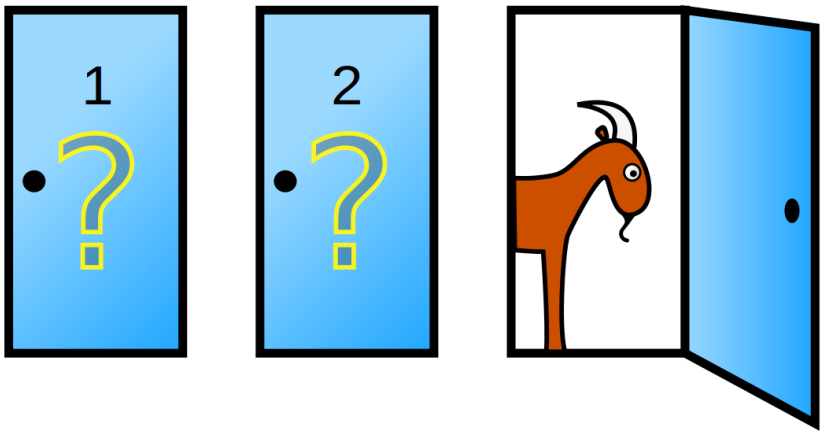
- Si `t` vaut 7 ou 11 alors le joueur a gagné.
- Si `t` vaut 2, 3 ou 12 alors le joueur a perdu.
- Sinon, le joueur rejoue les deux dés, dont le total des faces est noté `s`, **jusqu'à ce que** `s` = `t` ou que `s` = 7 :
  - lorsque `s=t` alors le joueur est gagnant,
  - lorsque `s=7` alors le joueur est perdant.

(1) Ecrire une fonction `craps` qui simule une partie de craps et renvoie `True` si le joueur a gagné et `False` sinon.
(2) En générant un million de parties de craps, déterminer la probabilité de gagner au craps.

Le dilemme de Monty Hall

Dans ce qui suit on présente le déroulement d'un ancien jeu télévisé américain, dit *jeu de Monty Hall*.

Dans un studio de télévision, Camille est en face de trois portes fermées. Derrière une seule des trois portes se trouve une voiture de luxe. Le but de Camille est de gagner cette voiture. Derrière chacune des deux autres portes se trouve une chèvre. L'animateur du jeu, Monty, est présent sur le plateau et **sait** derrière quelle porte se trouve la voiture. Monty demande à Camille de lui désigner une des trois portes, ce que fait Camille. Monty ouvre alors une **autre** porte et derrière laquelle se trouve une chèvre :



Monty demande alors à Camille d'ouvrir une des deux portes restantes avec la possibilité de gagner la voiture si elle la découvre.

Que doit faire Camille pour avoir le plus de chances de gagner la voiture ? Ouvrir la porte initiale ou ouvrir l'autre porte ? Où est-ce que le choix est indifférent ?

On définira deux fonctions `monty1()` et `monty2()` qui simuleront chacune des deux stratégies possibles. On partira d'une liste `jeu=["voiture", "chèvre", "chèvre"]` et on mélangera cette liste par `shuffle(L)` où `shuffle` est une fonction de mélange à importer du module `random`.

Puis on appellera 1000 fois chacune des deux fonctions pour savoir si une stratégie est préférable à l'autre.

Mélange de Fisher-Yates

L'algorithme de Fisher-Yates permet de mélanger une liste aléatoirement. Dans cet exercice, la liste choisie sera toujours la liste des `n` premiers entiers à partir de 1, par exemple, si `n=4` ce sera la liste `L=[1, 2, 3, 4]`. On devra écrire une fonction `fisher_yates(n)` qui génère un mélange aléatoire `M` de tous les entiers de la liste `L`. Par exemple, l'appel `fisher_yates(4)` pourrait générer la liste `M = [2, 3, 4, 1]`.

Ce qui suit décrit l'algorithme sur la base de l'exemple précédent :

— la longueur de la liste est `n=4` ;

— on dispose d'une liste, pour l'instant vide `M=[]`

— on tire un indice aléatoire entre 1 et `n=4`, par exemple `i=2` et on cherche l'élément de `L` qui est en `i`-ème position, c'est-à-dire ici 3 et on le place dans `M`, ce qui donne `M=[2]` ; puis on « raye » l'élément 2 de la liste `L` en mettant 0 à sa place ce qui donne `L=[1, 0, 3, 4]` ;

— comme il ne reste plus que 3 entiers aléatoires à trouver, on choisit un entier aléatoire `i` entre 1 et `n-1=3`, imaginons que ce soit `i=3` ; on cherche alors dans `L` le i-ème élément **parmi ceux qui n'ont pas déjà été tirés**. Ici, le 3e élément de `L=[1, 0, 3, 4]` qui n'a pas été tiré est 4 (on n'a donc pas compté l'élément de `L` où il y a 0). On rajoute cet élément à `M` ce qui donne `M=[2, 4]` ; enfin, on raye de `L` l'élément que l'on vient de tirer ce qui donne `L=[1, 0, 3, 0]` ;

— on recommence puique notre liste `M` n'a pas encore `n` éléments : on choisit un entier aléatoire `i` entre 1 et 2, par exemple `i=1` et on cherche dans `L` le i-ème élément parmi ceux qui n'ont pas déjà été tirés. Ici, le 1er élément de `L=[1, 0, 3, 0]` qui n'a pas été tiré est 1. On rajoute

cet élément à `M` ce qui donne `M=[2, 4, 1]` ; enfin, on raye de `L` l'élément que l'on vient de tirer ce qui donne `L=[0, 0, 3, 0]`.

— Et on continuerait ainsi de suite tant que la liste `L` n'est pas formée que de zéros.

Les deux premières questions sont indépendantes.

1. Ecrire une fonction `consecutifs(n)` qui renvoie la liste de tous les entiers consécutifs entre 1 et `n` (supposé entier valant au moins 1). Par exemple, `consecutif(4)` renvoie `[1, 2, 3, 4]`.
2. Ecrire une fonction `rang(L, i)` où `L` représente une liste et `i` un entier entre 1 et `n=len(L)` et qui renvoie l'indice `j` de la i-ème valeur de `L` qui soit **non nulle**. Par exemple, si

   `L=[42, 33, 0, 0, 81, 0, 82, 31]`

   et `i=4` alors `rang(L, i)=6` car 82 est la 4e valeur non nulle de `L` et que 82 est à l'indice 6 de la liste `L`. On supposera qu'il existe toujours une valeur de `L` qui convienne.
3. En déduire un code de la fonction `fisher_yates(n)`.
   Tester votre fonction pour `n` valant 10, 100, 1000 ou 10000 (et dans ce dernier cas, l'exécution ne sera pas instantanée).

L'algorithme P

L'algorithme P ou encore *mélange de Knuth* est un algorithme optimal de mélange aléatoire d'une liste `L`.

L'algorithme consiste à faire une succession d'échanges élémentaires. Précisons : si on dispose d'une liste `L`, on appelle *échange élémentaire* l'échange du dernier élément de la liste avec un élément aléatoire de la liste `L`. Par exemple, si `L` est la liste `L=[10, 11, 12, 13, 14]`, alors la liste `L=[10, 14, 12, 13, 11]` est un échange élémentaire : on a choisi une position aléatoire dans `L`, ici la 2e position et on a échangé l'élément avec le dernier.

L'algorithme P consiste à faire la succession d'échanges élémentaires :

— sur la totalité de la liste,

— puis sur tous les éléments sauf le dernier,

— puis sur tous les éléments sauf les deux derniers,

— et ainsi de suite jusqu'à n'avoir qu'un seul élément (auquel cas il n'y a plus d'échange).

Par exemple, si au départ `L=[10, 11, 12, 13, 14]` alors on peut avoir les étapes suivantes :

— on choisit au hasard disons la 2e position, ce qui donne `L=[10, 14, 12, 13, 11]`

— on choisit au hasard disons la 3e position, ce qui donne `L=[10, 14, 13, 12, 11]`

— on choisit au hasard disons la 2e position, ce qui donne `L=[10, 13, 14, 12, 11]`

— on choisit au hasard disons la 1re position, ce qui donne `L=[13, 10, 14, 12, 11]`

— et on n'a plus de choix, on est arrivé en tout début de liste.

D'où la liste aléatoire `L=[13, 10, 14, 12, 11]`.

Ecrire une fonction `algo_P(L)` qui mélange aléatoirement tous les éléments d'une liste `L`. Par exemple, si `L=[10, 11, 12, 13, 14]` alors l'appel `algo_P(L)` pourrait modifier la liste `L` en la liste `[13, 10, 14, 12, 11]`.

Pour échanger les éléments d'indice `i` et `j` d'une liste `L`, on pourra utiliser le code suivant :

```
L[i], L[j] = L[j], L[i]
```

On testera sur des listes formées d'entiers consécutifs à partir de 1 (comme [1, 2, 3, 4, 5]) de longueur 10, 100 ou 10000000.

Somme, maximum

1. Écrire le code de la fonction définie par

$$f(n) = -n^3 + 28n + 1$$

et tester ce code.

2. Ecrire une fonction `somme_f(a, b)` qui prend en paramètres deux entiers a et b que l'on supposera toujours vérifier $a \leq b$ et qui renvoie la somme suivante :

$$f(a) + f(a+1) + \cdots + f(b-1) + f(b)$$

Par exemple, `somme_f(3, 6)` vaudra $f(3) + f(4) + f(5) + f(6) = 76$. La fonction f précédente doit être réutilisée.

3. Ecrire une fonction `max_f(a, b)` qui prend en paramètres deux entiers a et b que l'on supposera toujours vérifier $a \leq b$ et qui renvoie la plus grande valeur prise par f en les points $a, a+1, ..., b$.
Par exemple, on trouvera que `max_f(-5, 10)` vaut 58.

Diviseurs, nombre premier

Dans cet exercice, tous les entiers considérés sont **strictement positifs**.

On rappelle qu'un entier d est dit un *diviseur* d'un entier n si n est un multiple de d. Par exemple, 10 est un diviseur de 2020 mais n'est pas un diviseur de 2024.

1. Ecrire une fonction `diviseurs(n)` qui renvoie la liste de tous les diviseurs de l'entier `n`. Voici quelques exemples du comportement de la fonction `diviseurs` :

```
42 -> [1, 2, 3, 6, 7, 14, 21, 42]
41 -> [1, 41]
81 -> [1, 3, 9, 27, 81]
75 -> [1, 3, 5, 15, 25, 75]
1 -> [1]
```

2. Un entier est dit **premier** s'il admet exactement DEUX diviseurs. Par exemple, 42 n'est pas premier car il admet au moins trois diviseurs, par exemple 1, 6 et 42. En revanche, 41 est premier car ses seuls diviseurs sont 1 et 41.
Ecrire une fonction `estPremier(n)` qui renvoie `True` si l'entier `n` est premier et `False` sinon. La fonction **doit** utiliser la fonction `diviseur` de la question précédente.

Plus grand diviseur impair : méthode rapide

Soit un entier $n > 0$. On cherche, de manière efficace, le plus grand diviseur impair d de n. Par exemple :

- si $n = 42$ alors $d = 21$,
- si $n = 45$ alors $d = 45$,

— si $n = 64$ alors $d = 1$,

— si $n = 1000$ alors $d = 125$.

On peut vérifier que $d = n/p$ où p est la plus grande puissance de 2 divisant n. Par exemple, si $n = 1000$ alors la plus grande puissance de 2 divisant n est $p = 2^3 = 8$ et donc le plus grand diviseur impair de n est $d = n/p = 1000/8 = 125$.

1. Ecrire une fonction `p2(n)` qui renvoie `p` en testant toutes les puissances de 2, c'est-à-dire 1, 2, 4, 8, etc.
2. Ecrire une fonction `maxDivImpair(n)` qui utilise la fonction précédente et renvoie le plus grand diviseur impair de `n`. Calculer `maxDivImpair(416241604)`.

Nombre de zéros qui terminent un entier (version fonction)

On demande d'écrire une fonction `zeros(n)` qui prenant un entier `n` $\geq$ `0` en paramètre, par exemple `n` = `4205000`, renvoie le nombre de zéros qui terminent l'écriture décimale du nombre `n` ; dans l'exemple précédent, ce nombre est 3. On pourra remarquer que c'est l'exposant de la plus grande puissance de 10 dont `n` soit multiple. On utilisera obligatoirement une boucle `for`. Ne pas oublier le cas `n` = `0`.

Décomposer en groupes de 2 ou 3

On se donne un entier positif n et on cherche les solutions (x, y) en nombres entiers positifs de l'équation

$$2x + 3y = n$$

Par exemple, si $n = 42$ alors l'équation admet la liste suivante de 8 solutions :

```
[[21, 0], [18, 2], [15, 4], [12, 6],
[9, 8], [6, 10], [3, 12], [0, 14]]
```

1. Ecrire une fonction `solve(n)` qui renvoie la liste des solutions de l'équation ci-dessus, une solution étant vue comme une liste `[x, y]` à deux éléments.
2. Si `n` est un entier positif, on appelle `q` le quotient entier de `n` par 3 et `r` le reste de `n` par 2. Ecrire une fonction `f(n)` qui renvoie le quotient par 2 de `q` + 2 − `r`.
3. Vérifier pour tous les entiers `n` entre 0 et 1000 que le nombre de solutions entières positives de l'équation 2`x` + 3`y` = `n` est `f(n)`. La suite `f(n)` est répertoriée ICI.

Entier différence de deux carrés

Tout entier qui, divisé par 4, admet un reste autre 2, peut s'écrire comme différence de deux carrés parfaits. Par exemple, prenons 2021 dont le reste de la division par 4 est 1 ou encore 2024 dont le reste de la division par 4 est 0 (et est donc différent de 2 dans chaque cas) ; alors :

$$2021 = 45^2 - 2^2, \quad 2024 = 507^2 - 505^2.$$

On demande d'écrire une fonction `diff_carres(n)` qui prend en paramètre un entier `n` et qui renvoie :

— l'objet `None` si `n` est un entier pair et non multiple de 4 (par exemple 2022),

— une liste de deux entiers `[a, b]` telle que $n = a^2 - b^2$ sinon.

On pourra utiliser les identités suivantes où `k` est un entier :

— identité A : $2k + 1 = (k + 1)^2 - k^2$

— identité B : $4k = (k + 1)^2 - (k - 1)^2$

Exemples de comportement :

```
12 : [4, 2]
13 : [7, 6]
14 : None
15 : [8, 7]
```

Calcul itératif du ppcm

On se donne deux entiers `a` et `b` strictement positifs et on cherche le ppcm `m` de ces deux entiers par application de l'algorithme suivant : on crée une suite de couples (x, y) jusqu'à ce que $x = y$. La suite est initialisée avec `(a, b)` et si (x, y) est le couple courant alors le couple suivant (X, Y) est défini par la condition suivante :

— si $x = \min(x, y)$ alors $Y = y$ et `X` est le plus petit multiple de `a` supérieur ou égal à `y`

— si $y = \min(x, y)$ alors $X = x$ et `Y` est le plus petit multiple de `b` supérieur ou égal à `x`.

Lorsque l'algorithme s'arrête alors `x` (ou `y`) est le ppcm des deux entiers `a` et `b`.

Par exemple, si $a = 16$ et $b = 60$, le tableau ci-dessous montre la suite des couples :

x	y
16	60
64	60
64	120
128	120
128	180
192	180
192	240
240	240

et on lit donc à la dernière ligne que le ppcm de `a` et `b` est 240.

Ecrire une fonction `ppcm(a, b)` qui implémente cet algorithme.

ppcm de plusieurs entiers

Si $a_1, \ldots, a_n$ sont des entiers strictement positifs, le ppcm de ces entiers est le plus petit entier qui soit multiple de chacun des entiers. Par exemple, le ppcm de 16, 42 et 60 est 1680 car

$$1680 = 16 \times 105 = 42 \times 40 = 60 \times 28$$

et aucun entier non nul et strictement inférieur à 1680 n'est multiple simultanément de 16, 42 et 60.

Pour calculer le ppcm de $n \geq 3$ entiers, il suffit de savoir calculer le ppcm de deux entiers. En effet, on peut montrer que le ppcm des entiers $a_1, \ldots, a_n$ s'obtient en prenant le ppcm de `m` et a_n

où m est le ppcm de $a_1, \ldots, a_{n-1}$. Donc pour calculer le ppcm de 16, 42 et 60, on calcule le ppcm de 16 et 42, qui vaut 336 puis on calcule le ppcm de 336 et 60 qui vaut 1680.

Ecrire une fonction `ppcm(L)` qui calcule le ppcm des entiers d'une liste donnée L. Pour cela, on donne le code d'une fonction ppcm2 qui calcule le ppcm de deux entiers et que l'on pourra utiliser librement :

```
def ppcm2(a, b):
    from math import gcd
    return a*b//gcd(a, b)
```

Somme des termes d'une récurrence double

On considère la suite d'entiers calculés successivement de la manière suivante : le nombre courant s'obtient en retirant au triple du dernier nombre calculé le double de l'avant dernier nombre calculé. Si on part de 1 et 3, la suite calculée commence par

```
1, 3, 7, 15, 31, 63, 127, 255, 511, 1023
```

Par exemple, $63 = 3 \times 31 - 2 \times 15 = 93 - 30 = 63$.

1. On demande de calculer la somme des n premiers éléments de cette suite. Par exemple, si $n = 5$ alors la somme vaut 57.
2. En comparant plusieurs valeurs successives de s à la puissance de 2 immédiatement supérieure, essayez de deviner une formule qui exprime s en fonction du rang n (on trouvera $s = 2^{n+1} - n - 2$).

Nombres de Perrin

On considère la séquence infinie de nombres dits *nombres de Perrin*. Elle est définie comme suit :

— les trois premiers sont : 3, 0 et 2

— chaque nombre suivant est la somme de l'avant-dernier et de l'avant-avant dernier calculés.

Ainsi, dans la séquence, le nombre suivant est `0 + 3 = 3`, le suivant encore est `2 + 0 = 2` qui est suivi de `3 + 2 = 5` .

Voici la liste des 10 premiers nombres de Perrin :

Rang	Nombre de Perrin
1	3
2	0
3	2
4	3
5	2
6	5
7	5
8	7
9	10
10	12

Ecrire une fonction `perrin(n)` qui renvoie le n^e nombre de Perrin. En particulier, calculer le 1000^e nombre de Perrin. On trouvera un très grand nombre commençant par `100299`.

Sommes d'entiers consécutifs

1. Ecrire une fonction `somme_consecutifs(i, j)` qui renvoie la somme de tous les entiers entre les entiers `i` et `j` où on suppose que `i` $\leq$ `j`. Par exemple, `somme_consecutifs(10, 15)` vaut $10+11+12+13+14+15=75$.
2. On donne un entier `n` ≥ 0. De combien de façons peut-on écrire comme `n` comme somme d'entiers consécutifs entre 1 et `n` ? Par exemple, pour `n` $= 15$, c'est possible de 4 façons :

$$15, \quad 7+8, \quad 4+5+6, \quad 1+2+3+4+5.$$

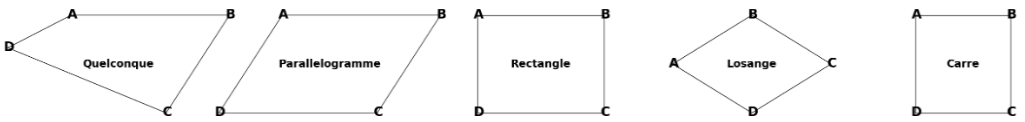
Somme des carrés

1. Ecrire une fonction `somme2(n)` qui renvoie $1^2+2^2+3^2+\cdots+(n-1)^2+n^2$.
2. Dans les deux dernières questions, il est attendu d'utiliser une boucle `while`.
   a. Montrer que 42 est le plus petit entier `n` tel que la somme $1^2+2^2+3^2+\cdots+(n-1)^2+n^2$ dépasse strictement 25000.
   b. Trouver le plus petit entier `n` tel que la somme $1^2+2^2+3^2+\cdots+(n-1)^2+n^2$ dépasse strictement 25000.
   c. Trouver le plus petit entier `n` tel que la somme $1^2+2^2+3^2+\cdots+(n-1)^2+n^2$ dépasse strictement 10^{14}.

Nature d'un quadrilatère

Rappelons quelques définitions de géométrie. Un quadrilatère `ABCD` est dit un

- *parallélogramme* si `AB` $=$ `CD` et `AD` $=$ `BC`
- *rectangle* si c'est un parallélogramme tel que l'angle en `A` soit droit
- *losange* si c'est un parallélogramme tel que `AB` $=$ `AD`
- *carré* si c'est un rectangle et aussi un losange.



Par ailleurs, on identifie un point `M` du plan à la liste `[x, y]` de ses coordonnées dans un repère orthonormé donné. On rappelle que si `A` $=$ `[xA, yA]` et `B` $=$ `[xB, yB]` alors le carré de la distance entre `A` et `B` vaut $\texttt{AB}^2 = (\texttt{xB}-\texttt{xA})^2+(\texttt{yB}-\texttt{yA})^2$.

On observera que pour comparer des distances, on peut se contenter de comparer les carrés des distances. Pour établir que l'angle en U du triangle `UVW` est un angle droit, on peut utiliser le théorème de Pythagore et établir que $\texttt{UV}^2+\texttt{UW}^2=\texttt{VW}^2$.

1. Écrire une fonction `longueurs(A, B, C, D)` qui recevant les coordonnées des sommets `A`, `B`, `C` et `D` d'un quadrilatère renvoie la liste des carrés des quatre côtés $\texttt{AB}^2$, $\texttt{BC}^2$, $\texttt{CD}^2$ et $\texttt{DA}^2$ ainsi que $\texttt{AC}^2$, le carré de la diagonale.
2. Ecrire une fonction `quadrilatere(A, B, C, D)` qui recevant les coordonnées des sommets `A`, `B`, `C` et `D` d'un quadrilatère, affiche exactement une des chaînes suivantes :

— `"Carré"` si le quadrilatère `ABCD` est un carré,

— `"Rectangle"` si le quadrilatère `ABCD` est un rectangle,

— `"Losange"` si le quadrilatère `ABCD` est un losange,

— `"Parallélogramme"` si le quadrilatère `ABCD` est un parallélogramme

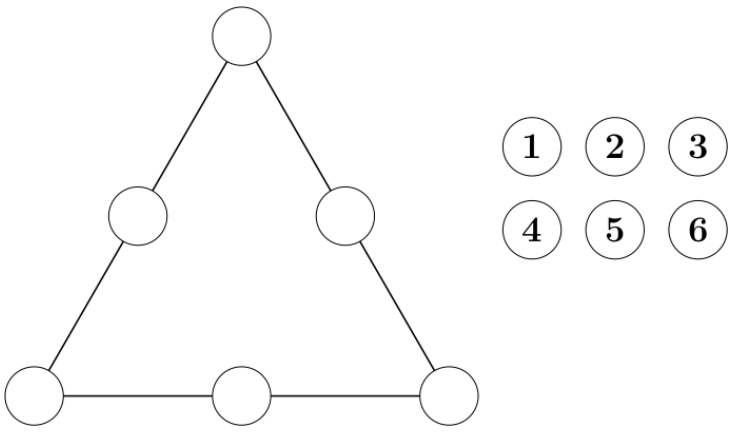
— `"Quelconque"` sinon.

On utilisera la question 1 et on supposera que les points ont des coordonnées entières. Voici 5 exemples de sortie du programme :

```
(0, 0) (5, 0) (7, 3) (2, 8) : Quelconque
(0, 0) (5, 0) (7, 3) (2, 3) : Parallélogramme
(1, 2) (7, -1) (5, -5) (-1, -2) : Rectangle
(-1, -2) (3, 1) (3, 6) (-1, 3) : Losange
(0, 0) (3, 1) (2, 4) (-1, 3) : Carré
```

Triangle magique

Dans le triangle ci-dessous



on demande de trouver toutes les façons de répartir les 6 jetons sur leurs emplacements en sorte que sur chaque côté, la somme des numéros fasse toujours 11.

Plusieurs méthodes sont possibles. On pourra tester toutes les répartitions a priori possibles en utilisant des boucles `for` imbriquées. On pourra avoir besoin d'écrire une fonction `sans_doublon(L)` qui prend en argument une liste `L` d'entiers entre 1 et 6 et renvoie `True` si cette liste contient une fois et une seule chaque entier entre 1 et 6.

Divisibilité par 7 : le critère de Chika

Pour savoir si un entier n, écrit en base 10, est multiple de 7, comme c'est le cas de $77 = 7 \times 11$, $42 = 7 \times 6$ ou $490 = 7 \times 70$, on peut utiliser le procédé suivant et redécouvert en 2019 par Chika, un collégien nigérian de 12 ans :

— on supprime le chiffre des unités u de n ce qui donne un entier nommé m

— on calcule $N = m + 5u$

Alors n est multiple de 7 si et seulement N l'est. Si nécessaire, on peut recommencer le procédé avec N et cela jusqu'à ce qu'on rencontre un entier assez petit que l'on sait être (ou ne pas être) multiple de 7.

Par exemple, 69132 est-il un multiple de 7 ? On applique la règle plusieurs fois de suite :

- $6913 + 5 \times 2 = 6923$
- $692 + 5 \times 3 = 707$

Il est clair que 707 est un multiple de 7 donc cela signifie que 69132 est un multiple de 7. Bien sûr, on aurait pu continuer comme suit :

- $70 + 5 \times 7 = 105$
- $10 + 5 \times 7 = 35$
- $3 + 5 \times 5 = 28$
- $2 + 5 \times 8 = 42$
- $4 + 5 \times 2 = 14$
- $1 + 5 \times 4 = 21$
- $2 + 5 \times 1 = 7$

jusqu'à obtenir 7.

Ecrire une fonction `est_multiple_7(n)` qui partant d'un entier `n` applique l'algorithme ci-dessus et renvoie `True` si `n` est multiple de 7 et `False` sinon. A l'aide d'une boucle `while`, on continuera le procédé jusqu'à trouver un entier à un chiffre ou bien 49.

Puissance de 2 commençant par ...

On cherche la plus petite puissance de 2 dont l'écriture en base 10 commence par un entier donné. Par exemple, la plus petite puissance de 2 commençant par 42 est $n = 2^{32} = 4294967296$. On admet qu'une telle puissance de deux existe toujours.

Ecrire une fonction `dominant(d)` qui renvoie l'exposant de la plus petite puissance de 2 qui commence par l'entier `d`. Ainsi `dominant(42)` vaut 32. On devra en particulier déterminer `dominant(2020)` et `dominant(7629)`.

On remarquera que les `k` premiers chiffres d'un entier `n` sont en fait le quotient de `n` par une puissance de 10 bien choisie. On pourra utiliser la fonction suivante qui retourne le nombre de chiffres d'un entier `n` :

```
from math import log10

def nb_chiffres10(n):
    return int(1+log10(n))

# Affiche 8
print(nb_chiffres10(56965011))
```

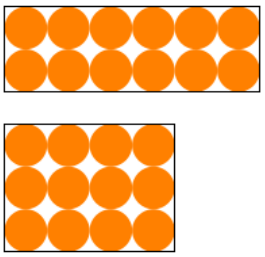
Nombres hautement composés

Un nombre entier `n` > 1 est dit *hautement composé* s'il contient strictement plus de diviseurs que tous les entiers `m` tels que $1 \leq$ `m` $<$ `n`. Par exemple, 6 est hautement composé car un entier entre 1 et 5 admet au plus 3 diviseurs tandis que 6 en admet 4.

1. Ecrire une fonction `diviseurs(n)` qui renvoie la liste des diviseurs de l'entier `n`.
2. Ecrire une fonction `hautementPremiers(N)` qui renvoie la liste de tous les entiers hautement premiers `n` tels que $1 \leq n \leq N$. Appliquer à $N = 10000$.

Billes en rectangle

Cet exercice n'est pas un exercice de dessin. On se donne N billes, ci-dessous $N = 12$



et on veut les regrouper de toutes les façons possibles en rectangles de tailles $n \times p$ (ci-dessus $2\times6, 3\times4$) puis sélectionner un rectangle de contour le plus petit possible (ci-dessus, les contours ont pour longueurs 16 et 14 donc le rectangle cherché est de dimensions 3×4). On supposera que les rectangles ne sont pas aplatis.

Ecrire une fonction `plus_petit_contour(N)` qui renvoie la liste `[n, p]` formée des dimensions d'un rectangle contenant les `N` billes et de contour minimal. Par exemple,

- `plus_petit_contour(12)=[3, 4]`
- `plus_petit_contour(2020)=[20, 101]`
- `plus_petit_contour(2024)=[44, 46]`
- `plus_petit_contour(6804000)=[2592, 2625]`

On procédera de la manière suivante : chaque dimension du rectangle étant un diviseur de N, on parcourra les diviseurs de N et pour chaque dimension possible, on recherchera l'autre dimension ce qui permettra de faire la sélection.

Elément le plus proche des éléments d'une liste

On donne une liste `L` formée de $n \geq 1$ nombres entiers, par exemple

$$L = [32, 79, 52, 87, 59, 38, 27, 71]$$

et on donne un entier `x`. L'objectif de l'exercice est de déterminer l'entier `y` de la liste `L` dont `x` est le plus proche. En cas d'ex aequo, on prendra celui qui apparait le premier dans la liste.

Pour la liste $L = [32, 79, 52, 87, 59, 38, 27, 71]$, voici des exemples de comportement :

```
x = 45 -> y = 52
x = 100 -> y = 87
x = 9 -> y = 27
```

1. Ecrire une fonction `dist(a, b)` qui renvoie la distance entre deux nombres `a` et `b`. On n'utilisera pas la fonction `abs`. Par exemple, `dist(42, 30)` tout comme `dist(30, 42)` renverront 12.
2. Ecrire une fonction `plus_proche(L, x)` qui renvoie le nombre y décrit au début de l'énoncé. La fonction devra utiliser la fonction `dist`.
   Cet exercice a été proposé sur le forum Python d'OpenClassrooms.

Elément le plus proche d'une subdivision régulière

On donne une liste `L` formée de `n` ≥ 1 nombres entiers régulièrement espacés, par exemple

$$L = [-6, 4, 14, 24, 34, 44]$$

et on donne un entier `x`. On demande de déterminer l'entier `y` de la liste `L` dont `x` est le plus proche. En cas d'égalité, on prendra le plus petit. On écrira une fonction `plus_proche(L, x)`. Il n'y a pas de limitation sur la taille `n` de la liste `L`. On pourra commencer par traiter les cas particuliers où `x` n'est pas compris entre les éléments extrêmes de `L`. Ensuite, avec une division entière, on cherchera le plus grand élément de `L` inférieur ou égal à `x`.

Pour la liste `L` $= [-6, 4, 14, 24, 34, 44]$, voici des exemples de comportement :

```
x = -9 -> y = -6
x = -6 -> y = -6
x = 12 -> y = 14
x = 19 -> y = 14
x = 20 -> y = 24
x = 44 -> y = 44
x = 46 -> y = 44
```

Cet exercice a été proposé sur le forum Python d'OpenClassrooms.

Comparer des moyennes

Chaque semestre, Arthur passe 5 matières dont les coefficients sont

```
1er semestre
2, 3, 1, 1, 3

2e semestre
4, 2, 3, 3, 4
```

Voici ses notes pour les deux semestres de cette année, dans l'ordre des coefficients ci-dessus :

```
1er semestre
20, 9, 19, 16, 19

2e semestre
 3, 12, 2, 1, 11
```

On dispose par ailleurs de deux formules pour calculer sa moyenne :

— soit on calcule la moyenne pondérée de ses notes sur **toute l'année** sans tenir compte du fait qu'il y a deux semestres ;

— soit on fait la moyenne pondérée de **chaque semestre** puis on fait la moyenne des deux semestres.

(1) Vérifier qu'avec la 1re formule, Arthur n'est pas reçu à son année tandis qu'il est reçu si on utilise la 2de formule

(2) (a) Plus généralement, si le semestre 1 comporte N1 matières, le semestre 2 de N2 matières, si `coeffs1` et `coeffs2` sont les listes des coefficients de chaque semestre, et si `notes1` et `notes2` sont les listes des notes de chaque semestre, écrire une fonction `compare(note1, notes2, coeffs1, coeffs2)` qui indique si, oui ou non, $\texttt{m1} \leq \texttt{m2}$ où `m1` est la moyenne obtenue par le premier calcul et `m2` par le 2nd.
On commencera par écrire une fonction `moyenne(notes, coeffs)` qui calcule une moyenne pondérée d'une liste de `notes` de coefficients `coeffs` et on ramènera tous les calculs de moyenne à une utilisation de cette fonction. On pouura utiliser que si `L` et `M` sont deux listes alors `L+M` est la concaténation des listes `L` et `M`. Tester avec les notes d'Arthur.

(b) Générer aléatoirement N1, N2, la liste des coefficients, une liste de notes et en répétant l'expérience N fois, examiner si une formule est toujours plus intéressante qu'une autre.

Fonction non injective

Soit la fonction mathématique $\texttt{f}(\texttt{x},\texttt{y}) = 173\texttt{xy} - 410\texttt{x}^2\texttt{y}^2 + 42\texttt{y} + 2024\texttt{x}$.

On recherche 2 points *distincts* à coordonnées entières, $(\texttt{a},\texttt{b})$ d'une part et $(\texttt{c},\texttt{d})$ d'autre part, tels que $\texttt{f}(\texttt{a},\texttt{b}) = \texttt{f}(\texttt{c},\texttt{d})$.

Pour cela, on considère un entier $A > 0$ assez grand et un « carré » de points $-\texttt{A} \leq \texttt{x},\texttt{y} \leq \texttt{A}$ et on parcourt les couples d'entiers `x`, `y` du carré en mémorisant dans une liste `V` les valeurs $\texttt{f}(\texttt{x},\texttt{y})$ en les points non déjà rencontrés ainsi que, dans une 2e liste `XY`, les couples `(x, y)` correspondants et à chaque étape, on examine si la valeur de la fonction a déjà été rencontrée (grâce à la liste `V`) et si oui, en quel couple (grâce à la liste `XY`). On pourra mémoriser dans `XY` les couples sous forme de liste `[x, y]`.

Dans le cas de la fonction ci-dessus, on trouvera par exemple que $\texttt{f}(0,81) = \texttt{f}(-1,-2)$.

Persistance multiplicative

Etant donné un entier positif `n` écrit en base 10, on peut calculer le produit `p` de ses chiffres. Par exemple, si $\texttt{n} = 39$ alors $\texttt{p} = 3 \times 9 = 27$. La *persistance multiplicative* de `n` est alors le nombre de fois qu'il faut répéter cette opération pour que le produit soit un nombre ayant un seul chiffre. Ainsi, la persistance de $\texttt{n} = 39$ est 3 car les produits successifs sont : 39 → 27 → 14 → 4. La persistance de 5 est 0 puisqu'il n'y aucune multiplication à faire.

On aura besoin de la fonction ci-dessous :

```
def digits(n):
    return list(map(int, str(n)))
```

Cette fonction renvoie la liste des chiffres en base 10 de l'entier `n`. Par exemple,

`digits(2024)=[2, 0, 2, 4]`.

(1) Ecrire une fonction `prod_liste(L)` qui étant donné une liste `L` d'entiers renvoie le produit des entiers de la liste. Par exemple, `prod([25, 3, 2]) = 150`. On pourra considérer que si `L` est la liste vide alors `prod_liste(L)=1`.

(2) Utiliser la fonction `digits(n)` pour écrire le code d'une fonction `prod(n)` qui renvoie le produit des chiffres de l'entier `n`.

(3) (a) Écrire une fonction `persistance(n)` qui renvoie la persistance multiplicative de `n`. Ainsi, `persistance(39)=3`.
(b) Le nombre actuellement connu ayant la plus grande persistance est 277777788888899. Calculer sa persistance.
(4) Trouver le premier entier entre 1 et 1000 ayant la plus grande persistance multiplicative.

Nombres de Catalan

Le nombre de Catalan d'indice n, noté C_n, est défini à partir des nombres de Catalan précédents. Plus précisément, $C_1 = 1$ et

$$C_n = C_1 \times C_{n-1} + C_2 \times C_{n-2} + \cdots + C_{n-1} \times C_1.$$

Ainsi

- $C_2 = C_1 \times C_1 = 1 \times 1 = 1$,
- $C_3 = C_1 \times C_2 + C_2 \times C_1 = 1 \times 1 + 1 \times 1 = 2$,
- $C_4 = C_1 \times C_3 + C_2 \times C_2 + C_3 \times C_1 = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5$
- $C_5 = C_1 \times C_4 + C_2 \times C_3 + C_3 \times C_2 + C_4 \times C_1 = 1 \times 5 + 1 \times 2 + 2 \times 1 + 5 \times 1 = 14$.

Les 10 premiers nombres de Catalan sont :

```
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862
```

(1) Ecrire une fonction `cat(L)` qui partant d'une liste `L` de `n` éléments renvoie la somme suivante :

```
L[0]*L[n-1] + L[1]*L[n-2] +  L[2]*L[n-3] + ... + L[n-1]*L[0]
```

Par exemple, si `L=[5, 2, 4, 7, 3]` alors `cat(L) = 74`.
(2) En déduire une fonction `catalan(n)` qui renvoie le nombre de Catalan d'indice `n`. Par exemple, `catalan(42)` doit renvoyer 10113918591637898134020.

Nombres de Bernoulli

Les nombres de Bernoulli forment une suite infinie de fractions dont voici les 11 premières valeurs :

$$1, \frac{1}{2}, -\frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, 0, -\frac{1}{30}, 0, \frac{5}{66}, \ldots$$

Ces nombres seront notés B_m où $m = 0, 1, 2 \ldots$ (noter que l'indice m commence à 0, comme pour les indices d'une liste). Chaque nombre de Bernoulli B_m (où $m > 0$) peut se calculer à l'aide des précédents $B_0, B_1, \ldots, B_{m-1}$ grâce à la formule suivante :

$$(m+1)B_m = -\sum_{k=0}^{m-1} \binom{m+1}{k} B_k.$$

Pour calculer explicitement B_m, il suffit donc de calculer le membre de droite et de diviser le résultat obtenu par $m + 1$. Le lien Wikipédia fourni ci-dessus donne le détail du calcul de B_1, B_2, B_3 et B_4.

Ecrire un fonction `bernoulli(n)` qui renvoie la liste des `n` + 1 premiers nombres de Bernoulli. On construira une liste initialement formée uniquement du premier nombre de Bernoullli et, à chaque étape d'une boucle, on calculera la nombre de Bernoulli courant et on l'ajoutera à la fin de la liste. Pour calculer le coefficient binomial, on importera la fonction comb du module `math` (on suppose que votre version de Python est au moins 3.7). Pour calculer avec des fractions et non pas des nombres flottants, on importera la classe `Fraction` du module standard fractions.

Vérifier que

$$B_{42} = \frac{1520097643918070802691}{1806}.$$

Fonction binomiale

1. Écrire une fonction `factorielle` qui renvoie la valeur factorielle de `n` où `n` étant passé en paramètre. Par exemple $1! = 1$ et $5! = 120$. On convient que $0! = 1$. On écrira une version de la fonction factorielle utilisant une boucle `for`.
2. On rappelle que si n et p sont des entiers positifs ou nuls, le coefficient binomial est défini par

$$\binom{n}{p} = \begin{cases} \dfrac{n!}{p!(n-p)!} & \text{si } 0 \leq p \leq n \\ 0 & \text{sinon} \end{cases}$$

   Ecrire une fonction `binomial(n, p)` qui renvoie le coefficient binomial ci-dessus.
3. a. Ecrire une fonction `pascal` qui vérifie l'identité de Pascal : $\binom{n}{p} + \binom{n}{p+1} = \binom{n+1}{p+1}$ valable pour tous entiers $n, p \geq 0$.
   b. Vérifier l'identité pour tous les entiers n et p entre 0 et 100. Un mauvais codage de la question précédente pourra produite une erreur pour n un peu au delà de 50.
4. Ecrire une fonction `maxBinomial(n)` qui renvoie la valeur maximale de tous les coefficients binomiaux $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \cdots, \binom{n}{n}$
5. Ecrire une fonction `verifMaxBinomial(n)` qui vérifie que la valeur maximale de tous les coefficients binomiaux $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \cdots, \binom{n}{n}$ est $\binom{n}{m}$ où m est le quotient la division entière de n par 2.

Valeur approchée de π

Pour chaque entier n, on considère la « somme » suivante :

$$S_n = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} - \cdots \pm \frac{1}{2n-1}$$

Explication : la somme alterne addition et soustraction et contient au total n fractions. Par exemple, $S_4 = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7}$ et $S_5 = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9}$. La dernière opération effectuée (plus ou moins) dépend de la valeur de n.

1. On admet que $4 \times S_n$ est une valeur approchée de $\pi = 3,1415926\ldots$ et qu'elle est d'autant meilleure que n est grand. Ecrire une fonction `pi_approx(n)` qui calcule S_n. Calculer une valeur approchée de π en prenant $n = 1000$.
2. Trouver le plus petit entier n tel que l'écart entre S_n et 3.141592 soit inférieur à 10^{-6} ?

Quotient illimité

On donne trois entiers `a`, `b`, `n` > 0 et on demande d'écrire une fonction `quo_dec(a, b, n)` qui détermine la liste `L` des chiffres du quotient de `a` par `b` avec `n` décimales. Par exemple, si `a` $= 22$ et `b` $= 7$ et `n` $= 20$ on obtiendra que `L` vaut :

```
[3, 1, 4, 2, 8, 5, 7, 1, 4, 2, 8, 5, 7, 1, 4, 2, 8, 5, 7, 1, 4]
```

Pour cela, on commencera par placer le quotient entier de `a` par `b` en première position de `L` et on fera suivre des décimales demandées. Pour obtenir ces décimales, on appliquera la méthode vue en classe de 6e, voir par exemple Pratique avec DÉCIMALES pour vous rafraîchir la mémoire.

Application

On donne les entiers suivants :

```
a=1076825233969550892368765865908909828003038267654921050616858441627304937095412
b=342764117664681504901177527623702904503014768549643256761838650532817379076979
```

et on admet que $\frac{a}{b}$ a 156 décimales communes avec π. Afficher la valeur approchée correspondante. On pourra utiliser la fonction `list2digits` ci-dessous pour obtenir un affichage lisible :

```
def list2digits(L):
    return str(L[0]) + ',' + ''.join(map(str,L[1:]))


L=[3, 1, 4, 2, 8, 5, 7, 1, 4, 2, 8, 5, 7, 1, 4, 2, 8, 5, 7, 1, 4]
print( list2digits(L))
```

```
3,14285714285714285714
```

Nombre de Lychrel

1. On donne un nombre entier `n` par la liste `L` de ses chiffres en base 10, en commençant par le chiffre des unités. Par exemple, si `n = 2038` alors `L = [8, 3, 0, 2]`. Soit `m` l'entier déduit de `n` en inversant l'ordre des chiffres de `n`. Par exemple, si `n=2030` alors `m = 302`. On demande de construire la liste des chiffres de `n + m`. On écrira une fonction `somme_miroir(n)` qui renvoie la liste des chiffres de `n + m`. On dira que `n + m` est la somme en miroir de `n`.
2. On donne un nombre entier `n` par la liste `L` de ses chiffres en base 10. Ecrire une fonction `estPalindrome(n)` qui renvoie `True` si `n` est un nombre palindromique, c'est-à-dire que `n = m` où `m` l'entier déduit de `n` en inversant l'ordre des chiffres de `n`. Par exemple, `4702074` est un nombre palindromique.
3. Etant donné un entier positif `n`, construire la liste de ses chiffres, en commençant par le chiffre des unités. On écrira une fonction `nombreChiffres(n)`.
   Par exemple, `nombreChiffres(2038)=[8, 3, 0, 2]`.
4. Etant donné la liste `L` des chiffres d'un entier positif `n` (commençant par le chiffre des unités), déterminer la valeur de `n`. On écrira une fonction `chiffresNombre(L)`.
   Par exemple, `chiffresNombre([8, 3, 0, 2])=2038`.
5. On part d'un entier `n` et on calcule les sommes en miroir successives en partant de `n` jusqu'à ce que le nombre obtenu soit une nombre palindromique (en supposant que cela se produise). Ecrire une fonction `lychrel(n)` qui renvoie le nombre palindromique trouvé ainsi que le nombre d'itérations. Pour `n=89`, on trouvera 24 itérations et pour `n` $= 10911$,

on trouvera 55 étapes.
Pour information, un nombre de Lychrel est un entier pour lequel la suite des itérés ne donne jamais un nombre palindromique (on ne connait pour l'instant aucun tel nombre, le plus petit candidat est 196).

Racine carrée d'un nombre complexe

Python prend en charge la gestion des nombres complexes : le nombre complexe `a + ib` (où `a`, `b` sont réels) est défini en Python par un appel `complex(a, b)`, ou encore par `a + jb` :

```
z1 = complex(1, -2)
z2 = 1+3j
print(z1)
print(z1 * z2)
```

```
(1-2j)
(7+1j)
```

On cherche à écrire notre propre fonction `csqrt(z)` qui renvoie une racine carrée du nombre complexe `z`. On utilisera le résultat mathématique suivant : si $z = a+ib$ est un nombre complexe (a et b étant réels) alors le nombre complexe

$$Z = \sqrt{\frac{a+\sqrt{a^2+b^2}}{2}} + i\frac{b}{\sqrt{2(a+\sqrt{a^2+b^2})}}$$

est une racine carrée de z.

On demande d'écrire une fonction `csqrt(z)` qui renvoie `Z`. On testera avec le nombre complexe $z = 7 + 24i$ et on devra trouver $Z = 4 + 3i$. On vérifiera en calculant Z^2 et aussi en calculant `z**0.5` (qui est permis en Python).

Tables de multiplication

On représente la table de multiplication d'un entier m par une liste de ces 10 premiers multiples m, $2m$, $3m$, etc ; par exemple la table de 7 est la liste

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Des tables de multiplication sont représentées par une liste de tables de multiplication et donc par une liste de listes.

Ecrire une fonction `tables_mult(L)` qui à partir d'une liste `L` d'entiers strictement positifs renvoie les tables de multiplication des éléments de `L`. Par exemple, si `L=[5, 0, 7, 42]` alors `table_mult(L)` sera la liste suivante :

```
[[5, 10, 15, 20, 25, 30, 35, 40, 45, 50],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70],
[42, 84, 126, 168, 210, 252, 294, 336, 378, 420]]
```

Regrouper par signe

On donne une liste `L` d'entiers et on demande de regrouper les éléments de `L` suivant leur signe. Le regroupement sera une liste `M` formée de 3 objets :

— la liste N formée des éléments négatifs de L ;
— un entier z indiquant le nombre d'occurrences de zéro dans la liste L ;
— la liste P formée des éléments positifs de L.

Voici quelques exemples de comportement :

```
[42, -17, 33] → [[-17], 0, [42, 33]]
[42, 17, -81, 0, 33, -53, 0, 25, 0] → [[-81, -53], 3, [42, 17, 33, 25]]
[21, 23, 42] → [[], 0, [21, 23, 42]]
```

On ecrira une fonction `signes(L)` qui renverra la liste M.

Regrouper par paires

On donne une liste L et on demande de regrouper les éléments de L par paires d'éléments successifs. Ci-dessous quelques exemples de comportements :

```
[] → []
[42] → [[42]]
[42, 17] → [[42, 17]]
[42, 17, 33] → [[42, 17], [33]]
[42, 17, 33, 25] → [[42, 17], [33, 25]]
```

On ecrira une fonction `parPaires(L)` qui renverra la liste M des paires. Une paire sera une liste formée de 2 éléments successifs de L. La dernière liste de M ne contiendra qu'un seul élément si la liste L est formé d'un nombre impair d'éléments.

Chaînage par paires

Dans ce qui suit, partant d'une liste d'entiers L telle que

```
L = [15, 12, 16, 15, 10, 11]
```

on cherche à générer la liste suivante :

```
M = [[15, 12], [12, 16], [16, 15], [15, 10], [10, 11]]
```

Plus précisément, on donne une liste L d'entiers, et on construit la liste M formée des listes `[a, b]` à deux éléments a et b qui sont placés à des positions consécutives dans la liste L et où a prend toutes les positions successives possibles dans la liste L. On écrira une fonction `chainage(L)` qui renvoie M.

Pour information, cette fonction imite la fonction `pairwise` du module itertools.

Séparer suivant la parité des valeurs

On donne une liste L d'entiers et on demande de construire une liste `P=[L0, L1]` où L0 est la liste (éventuellement vide) des entiers pairs de L, placés dans leur ordre d'apparition dans L et, de même, L1 est la liste des entiers impairs de L.

Par exemple, si `L=[42, 81, 31, 82, 81, 65, 9]` alors `P=[[42, 82], [81, 31, 81, 65, 9]]`.

On construira une fonction `separer(L)` qui renverra P.

Séparer des entiers selon la parité des indices

On donne une liste L d'entiers et on demande de placer dans une liste C deux listes :

— la première contient les entiers de L placés à un indice pair,
— la deuxième contient les entiers de L placés à un indice impair.

Par exemple si `L=[42, 81, 31, 42, 2019, 2024, 33]` alors

```
C=[[42, 31, 2019, 33], [81, 42, 2024]]
```

On écrira une fonction `separerIndices(L)` qui renverra `C`.

Pivot

1. On donne une liste d'entiers, par exemple `L=[24, 81, 12, 12, 42, 31, 82, 75, 42, 13]`. On donne un entier `p` de la liste `L`, qu'on appelera le pivot, par exemple `p=42`. On demande de construire deux listes d'entiers `A` et `B` où
   — la liste `A` est formée des entiers `x` de `L` tels que $x < p$,
   — la liste `B` est formée des entiers `x` de `L` tels que $x > p$.

   On écrira une fonction `separer(L, p)` qui renverra la liste `[A, B]`. Dans le cas de l'exemple, la liste renvoyée sera :

   ```
   [[24, 12, 12, 31, 13], [81, 82, 75]]
   ```

   Si les listes en compréhension vous sont connues, vous les utiliserez.
2. Ecrire une fonction `npivots(L, p)` utilisant la fonction `separer()` et renvoyant le nombre d'éléments de `L` dont la valeur est `p`. Dans le cas de l'exemple, `npivots(L, p)` vaudra 2.

Décompression de liste

On donne une liste `L` de listes de deux éléments, par exemple

```
L=[["Pomme", 2], ["Kiwi", 3], ["Orange", 0], ["Poire", 2]]
```

Chaque liste de deux éléments dans `L` est formée d'une chaîne de caractères `s` et d'un entier positif ou nul `n`. On demande de contruire une liste `M` de chaînes contenant pour chaque liste `[s, n]` figurant dans `L`, la chaîne `s` mais répétée `n` fois.

Avec l'exemple ci-dessus, la liste `M` attendue est :

```
['Pomme', 'Pomme', 'Kiwi', 'Kiwi', 'Kiwi', 'Poire', 'Poire']
```

Séparer les abscisses et les ordonnées

On donne une liste `L` de points du plan, chaque point étant donné par une liste de ses deux coordonnées. Ecrire une fonction `separer(L)` qui renvoie la liste formée de deux listes suivantes :

— la liste des abscisses des points de `L`, dans leur ordre d'apparition dans `L`
— la liste des ordonnées des points de `L`, dans leur ordre d'apparition dans `L`.

Par exemple, si `L` est la liste suivante

```
L = [[4, 9], [-2, -3], [0, -3], [2038, 2048]]
```

alors, `separer(L)` doit renvoyer la liste suivante :

```
[[4, -2, 0, 2038], [9, -3, -3, 2048]]
```

Si cette notion vous est connue, il est possible de donner une solution utilisant une liste en compréhension.

Rectangle à partir d'une de ses diagonales

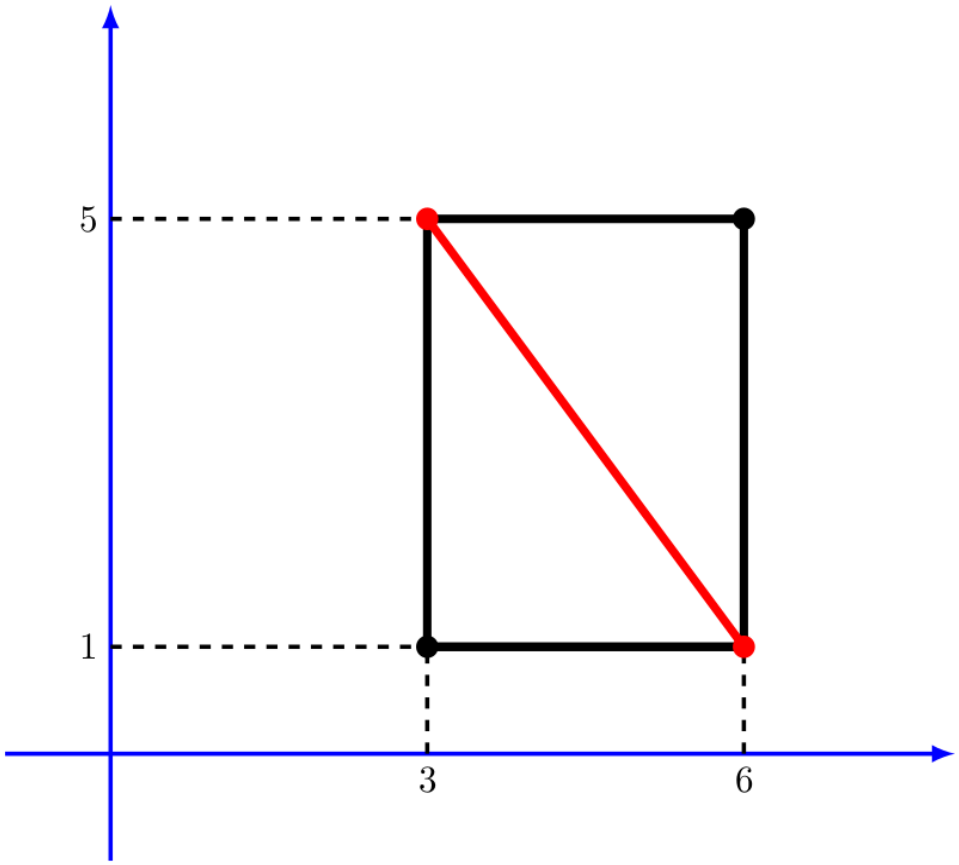
Cet exercice n'est pas un exercice de dessin

Un point du plan est représenté par une liste de deux entiers. Par exemple, le point d'abscisse 42 et d'ordonnée 81 est représenté par la liste `[42, 81]`.

On considère un rectangle dont les sommets sont des points à coordonnées entières. On donne les coordonnées de deux extrémités `D1` et `D2` d'une de ses deux diagonales. On demande d'écrire une fonction `sommets(D1, D2)` qui renvoie la liste des 4 sommets du rectangle correspondant, en commençant par le sommets `D1` et en sorte que les sommets soient listés dans le sens des aiguilles d'une montre.

Par exemple, si `D1 = [6, 1]` et `D2 = [3, 5]` alors `sommets(D1, D2)` sera la liste suivante :

```
[[6, 1], [3, 1], [3, 5], [6, 5]]
```



Reconstruire les sommets d'un rectangle

Cet exercice n'est pas un exercice de dessin.

Un point du plan est représenté par une liste de deux entiers. Par exemple, le point d'abscisse 42 et d'ordonnée 81 est représenté par la liste `[42, 81]`.

On considère un rectangle ABCD dont les sommets sont à coordonnées entières et dont les côtés AB, BC, CD et DA sont parallèles aux axes. On donne une liste `L=[x1, x2, y1, y2]` où

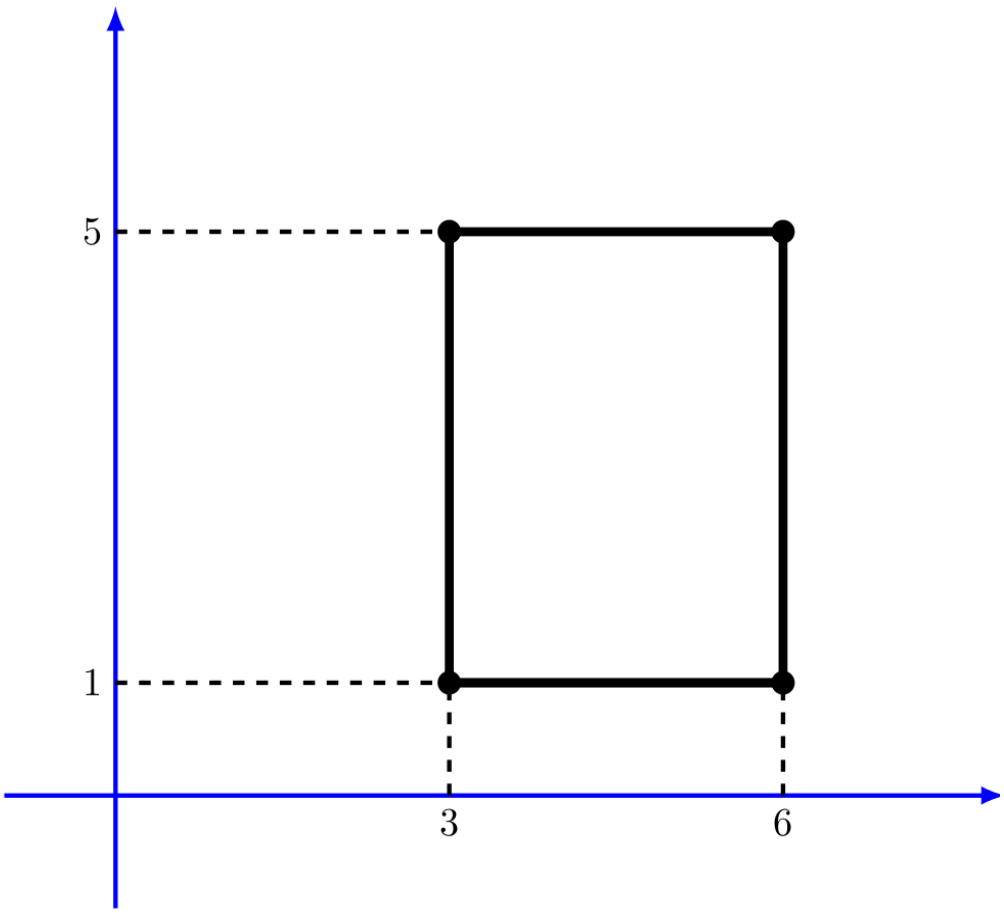
- `x1` est la plus petite des abscisses des sommets du rectangle,
- `x2` est la plus grande des abscisses des sommets du rectangle,
- `y1` est la plus petite des ordonnées des sommets du rectangle,
- `y2` est la plus grande des ordonnées des sommets du rectangle.

On demande de déterminer la liste des sommets `[A, B, C, D]` du rectangle. Le premier sommet `A` de la liste sera le sommet placé en bas à gauche. Les sommets seront listés dans le sens inverse des aiguilles d'une montre. Chaque sommet sera représenté par une liste `[x, y]` des coordonnées du sommet.

On écrira une fonction `rect(L)` où `L` sera la liste `L=[x1, x2, y1, y2]`.

Par exemple, si `L=[3, 6, 1, 5]` alors `rect(L)` sera la liste

```
[[3, 1], [6, 1], [6, 5], [3, 5]]
```



Jeu de cartes

Une carte à jouer, par exemple le Valet de Pique, sera vue comme une liste de deux chaînes, dans l'exemple ce sera

```
ma_carte = ["Valet", "Pique"]
```

Un jeu de 32 ou 52 cartes sera simplement vu comme une liste Python de toutes les cartes et donc une liste de listes de deux chaînes. Ecrire une fonction `jeu_cartes(n)` qui génère et renvoie un jeu de `n` cartes où `n` vaut 32 ou 52. On utilisera une liste de valeurs et d'enseignes :

```
VALEURS = ["As", "Deux", etc, "Dame", "Roi"]
ENSEIGNES = ["Pic", "Cœur", "Carreau", "Trèfle"]
```

Evénement périodique

Imaginons un événement se répétant régulièrement un certain nombre de fois dans la journée à partir d'une certaine heure ; on cherche à générer la liste des moments de la journée où l'événement se produit. Par exemple, si un événement se produit 7 fois à partir de 13h15 tous les 2h15, la liste cherchée sera :

```
[[13, 15], [15, 30], [17, 45], [20, 0], [22, 15], [0, 30], [2, 45]]
```

ce qui se comprend : 13h15, puis 15h30, puis 17h45, etc. jusqu'à 2h45.

On notera qu'on indique une heure de la journée par `[h, m]` où le nombre d'heures vérifie $0 \leq h < 24$ et le nombre de minutes vérifie $0 \leq m < 60$.

Plus précisément, on donne

- une période de temps `T`, en heures et minutes, sous forme de liste, par exemple `[1, 15]` pour traduire `1 h 15 min` ;
- un moment de départ, sous la forme d'une liste indiquant l'heure en heures et minutes, par exemple `[15, 30]` signifie `15 h 30` ;
- un entier désignant le nombre de fois qu'on va répéter la période, par exemple `n=6`.

On demande de produire les `n` moments de la journée où l'événement a lieu. On écrira une fonction.

Nom de qui a la meilleure note

On donne une liste `L` formée de listes de deux éléments :

- le premier est un nom
- le second est une note,

par exemple :

```
L =[
["Florian", 12], ["Ambre", 13],
["Emile", 5], ["Emma", 13],
["Aristide", 7]]
```

Ecrire une fonction `nom_meilleure_note(L)` qui renvoie le nom de la personne qui a eu la meilleure note. Si la meilleure note a été obtenue plusieurs fois, le nom donné sera le dernier de la liste. Avec la liste ci-dessus, la fonction doit renvoyer la chaîne `"Emma"`.

Cette question a été discutée sur le forum Python d'OpenClassrooms.

Moyenne pondérée

On donne une liste formée de notes pondérées, une note pondérée étant une liste formée d'une note entière entre 0 et 20 et d'un coefficient associé à la note.

On demande de calculer la moyenne de la liste de notes pondérées. On arrondira cette moyenne à 2 chiffres après la virgule, comme le montre l'exemple ci-dessous :

```
print(round(10.357142857142858,2))
```

```
10.36
```

Si on prend la liste des 5 notes pondérées ci-dessous :

```
[14, 3]
[8, 4]
[12, 4]
[7, 2]
[9, 1]
```

on obtiendra une moyenne de `10,36`.

On écrira une fonction `myn(N)` où `N` est un liste de notes pondérées autrement dit une liste de listes de deux éléments, le premier élément étant une note, le second étant le coefficient.

Séparer des entiers suivant leur nombre de chiffres

1. On donne une liste d'entiers positifs et on demande de placer dans une liste `C` quatre listes :
   - la première contient les entiers de `L` ayant 1 seul chiffre
   - la deuxième contient les entiers de `L` ayant 2 chiffres
   - la troisième contient les entiers de `L` ayant 3 chiffres
   - la dernière contient les autres nombres.

   Par exemple si `L=[42, 81, 5, 2019, 424242, 2020, 2024]` alors

   `C=[[5], [42, 81], [], [2019, 424242, 2020, 2024]].`

   On écrira une fonction `separer(L)`.
2. Ecrire une fonction `maj(L)` utilisant la fonction `separer(L)` qui renvoie `True` si les nombres ayant au moins 4 chiffres sont majoritaires dans `L` et qui renvoie `False` sinon.

Liste des diviseurs

1. On donne une liste `L` d'entiers strictement positifs et on demande d'écrire un code qui génère une liste `D` des listes de diviseurs de chaque entier de `L`. On rappelle que `d` est un diviseur de `N` signifie juste que `N` est un multiple de `d` autrement dit que $N = d \times q$ pour un certain entier `q`. Par exemple, si `L=[2019,2027, 43, 25]` alors

   `D=[[1, 3, 673, 2019], [1, 2027], [1, 43], [1, 5, 25]].`

   On écrira une fonction `div(L)`. Les diviseurs apparaîtront dans l'ordre croissant.
2. Utiliser la fonction précédente pour générer une liste `P` des nombres premiers de `L`. On rappelle qu'un nombre premier `p` (par exemple 43 mais pas 42) est un entier `p>1` et admettant exactement **deux** diviseurs strictement positifs, à savoir lui-même `p` et 1. Dans l'exemple ci-dessus, on obtiendra que `P=[2027, 43]`.

Statistiques sur des contrôles

Ci-dessous, le tableau des notes sur 20 de 12 étudiants aux 3 contrôles d'une UE :

Id	Prénom	CC n°1	CC n°2	CC n°3
1	LEO	12	5	11
2	MARC	18	17	16
3	ANTOINE	-1	9	-1
4	NATHAN	8	12	15
5	KEVIN	2	4	5
6	HUGO	10	9	11
7	CHAIMAE	7	7	5
8	JUSTINE	20	20	20
9	LEA	14	-1	-1
10	MARION	5	8	13
11	AUDREY	12	11	-1
12	LIONEL	-1	5	-1

Lorsqu'un contrôle reçoit la valeur -1 c'est que l'étudiant est absent. Si un étudiant est absent à au moins un contrôle, il est considéré comme absent à l'UE et ne reçoit pas de moyenne. Noter qu'en fait les notes de chaque étudiant sont accessibles via un indice i qui varie entre 1 et le nombre `n` d'étudiants, dans l'exemple c'est `n=12`. Noter que les identifiants commencent à 1 et pas à 0.

La liste des notes des étudiants de l'exemple est donnée dans la cellule ci-dessous. Le prénom des étudiants n'est pas utilisé, seul leur identifiant servira dans la suite. Noter que les notes du premier étudiant (indice 0) de la liste sont les notes de l'étudiant d'identifiant 1 (et non pas 0).

```
notes=[[12, 5,  11] ,
[18, 17, 16],
[-1, 9, -1],
[8, 12, 15],
[2, 4, 5],
[10, 10, 11],
[7, 7, 5],
[20, 20, 20],
[14, -1, -1],
[15, 18, 13],
[12, 11, -1],
[-1, 5, -1]]
```

Dans les questions ci-dessous, votre code doit être capable de traiter les situations les plus générales : un nombre quelconque d'étudiants, un nombre quelconque de contrôles.

1. Ecrire une fonction `moyenne(i, notes)` qui calcule la moyenne de l'étudiant d'identifiant donné `i` ≥ 1 dans une liste appelée `notes`. Dans l'exemple, Marc, d'identifiant 2, a 17 de moyenne donc `moyenne(2, notes)` renvoie 17.
2. Ecrire une fonction `absents(notes)` qui retourne une liste de listes, chaque liste donnant les identifiants des étudiants absents au 1er, 2e, etc contrôle. Dans l'exemple, `absents(notes)` renvoie la liste `[[3, 12], [9], [3, 9, 11, 12]]`.
3. Ecrire une fonction `moyenne_eme_controle(notes, nro)` qui renvoie la liste des id d'étudiants ayant eu au moins la note de 10 au contrôle identifié par le numéro valant `nro` (donc

numéro 1 pour le premier contrôle, numéro 2 pour le deuxième, etc). Avec l'exemple ci-dessus et `nro=2`, on devra obtenir : `[2, 4, 6, 8, 10, 11]`.

Relevé de températures

On relève la température pendant une semaine dans 4 villes ce qui donne le tableau suivant :

Ville	Lun	Mar	Mer	Jeu	Ven	Sam	Dim
Albi	28	25	29	31	33	32	35
Castres	27	26	30	29	32	34	34
Valence	24	22	27	28	28	31	31
Rodez	25	24	27	27	27	28	32

Il s'agit juste d'un exemple avec 4 villes et votre code doit fonctionner pour un nombre quelconque de villes.

Chaque ville est repérée par un indice commençant à 0 (on ne tiendra pas compte des noms de villes). Les températures relevées sont fournies dans une liste de 7 entiers. Le relevé est donné par une liste de listes. Dans l'exemple ci-dessus, cette liste est :

```
releve=[
[28, 25, 29, 31, 33, 32, 35],
[27, 26, 30, 29, 32, 34, 34],
[24, 22, 27, 28, 28, 31, 31],
[25, 24, 27, 27, 27, 28, 32]]
```

(1) Ecrire une fonction `tempMax(releve)` qui renvoie une liste `[t, j, v]` où `t` est la température maximale du relevé, `j` l'indice du jour de la semaine (entre 0 et 6) correspondant à la température maximale et `v` le numéro de la ville (à partir de 0) où cette température a été relevée. Dans l'exemple, ci-dessus, l'appel renverra la liste suivante :

```
[35, 6, 0]
```

(2) Ecrire une fonction `maxiJour2ville(releve, j)` qui à partir d'un indice `j` de jour de la semaine (entre 0 et 6) renvoie le numéro de la ville (à partir de 0) où il a fait le plus chaud le jour `j`.
Par exemple, si `j = 2` (mercredi) alors c'est à Castres qu'il a fait le plus chaud donc la réponse attendue est 1.

Transposer une liste de listes

On se donne un tableau `L` d'entiers de `n` lignes et `p` colonnes vu comme une liste de `n` listes ayant chacune `p` entiers. Par exemple, le tableau :

```
4 6 9
7 0 6
9 1 2
1 3 6
```

est représenté par la liste Python :

```
L = [[4, 6, 9], [7, 0, 6], [9, 1, 2], [1, 3, 6]]
```

On appelle *transposé* du tableau `L` le tableau obtenu en échangeant les lignes et les colonnes : la première ligne du transposé de `L` est la première colonne de `L`, et ainsi de suite. Dans l'exemple, c'est le tableau

```
4 7 9 1
6 0 1 3
9 6 2 6
```

représenté par la liste Python :

```
[[4, 7, 9, 1], [6, 0, 1, 3], [9, 6, 2, 6]]
```

Ecrire une fonction `transposer(L)` qui renvoie le tableau transposé du tableau `L`. Vérifier que si l'on transpose deux fois de suite, on retombe sur le tableau initial.

Scinder une liste en deux

On donne une liste `L` et un entier positif `m` et on cherche à contruire une liste `[L1, L2]` où `L1` est la liste des `m` premiers éléments de `L` et `L2` est la liste formée des autres éléments de `L`. Si $m \geq n$ où `n` est la longueur de la liste `L` alors on supposera que `L1` $=$ `L` et que `L2` est vide. Voici quelques exemples de comportements :

```
L = [8, 3, 8, 1, 6, 7] , m = 2 → [[8, 3], [8, 1, 6, 7]]
L = [3, 1, 4, 3, 4, 6] , m = 3 → [[3, 1, 4], [3, 4, 6]]
L = [2, 1, 5, 5, 9, 8] , m = 7 → [[2, 1, 5, 5, 9, 8], []]
```

Ecrire une fonction `scinder(L, m)` qui renvoie la liste `[L1, L2]`.

Diagonale descendante d'une grille

On vous donne un tableau 2D d'entiers sous la forme d'une liste `L` de listes, par exemple le tableau à 7 lignes et 4 colonnes suivant :

```
L=[
[95, 42, 31, 98],
[50, 98, 60, 41],
[85, 94, 15, 90],
[83, 42, 38, 33],
[69, 96, 70, 23],
[90, 27, 10, 24],
[46, 81, 55, 29]]

print(L)
```

On vous donne une position d'indice de ligne `i` et d'indice de colonne `j`, par exemple `i=3` et `j=1` dans le tableau ci-dessus et qui contient l'élément 42.

On demande de déterminer, sous forme de liste, les éléments du tableau `L` qui se trouvent sur la diagonale descendante contenant la case du tableau d'indices `i` et `j`. Avec l'exemple ci-dessus, la liste attendue est :

```
[85, 42, 70, 24]
```

On écrira une fonction `diagDesc(L, i, j)` qui renverra la liste demandée.

On pourra énumérer les éléments de la diagonale dans l'ordre de son choix.

Pour écrire le code, on utilisera que les cases d'indices `u` et `v` de la diagonale de `L`, descendante et contenant la case d'indices `i` et `j` vérifient exactement la relation `u - v = i - j`. Ainsi, dans l'exemple ci-dessus, la case d'indices `u=5` et `v=3` est bien sur la diagonale puisque `u - v = 5 - 3 = 2` et `i - j = 3 - 1 = 2`.

Matrice de covariance

Si `Z` est une liste de `n` nombres, on appelle *moyenne* de `Z` le nombre `moy(Z)` suivant

$$\texttt{moy(Z)} = \frac{1}{\texttt{n}}\left(\texttt{Z[0]} + \texttt{Z[1]} + \cdots + \texttt{Z[n-1]}\right)$$

Si `X` et `Y` sont deux listes de nombres, de même taille `n`, et de moyennes notée `mX` et `mY`, on appelle *covariance* de `X` et `Y` le nombre `cov(X, Y)` :

$$\texttt{cov(X, Y)} = \frac{1}{\texttt{n}-1}\left((\texttt{X[0]} - \texttt{mX})(\texttt{Y[0]} - \texttt{mY}) + \cdots + (\texttt{X[0]} - \texttt{mX})(\texttt{Y[0]} - \texttt{mY})\right)$$

On donne une liste `L` de `p` listes de nombres, chacune de longueur `n`. On appelle matrice de covariance de `L` la liste `C` de `p` listes, chacune de longueur `p` telle que pour tous indices $\texttt{i}, \texttt{j} = 0, \ldots, \texttt{p} - 1$:

$$\texttt{C[i][j]} = \texttt{cov(L[i], L[j])}$$

Ecrire une fonction `mat_cov(L)` qui renvoie cette matrice. On pourra écrire des fonction auxiliaires et on veillera à ne pas répéter inutilement certains calculs.

Par exemple, si `L` est la liste suivante :

```
[[ 64.  66.  68.  69.  73.]
 [580. 570. 590. 660. 600.]
 [ 29.  33.  37.  46.  55.]]
```

alors la matrice de covariance est

```
[[  11.5    50.     34.75]
 [  50.   1250.    205.  ]
 [  34.75  205.    110.  ]]
```

Triangle de Floyd vertical avec liste de listes

Soit le triangle de Floyd vertical à `n = 9` lignes :

```
1
2 10
3 11 18
4 12 19 25
5 13 20 26 31
6 14 21 27 32 36
7 15 22 28 33 37 40
8 16 23 29 34 38 41 43
9 17 24 30 35 39 42 44 45
```

Plus généralement, soit le motif consistant en un triangle formé de n lignes et de n colonnes et composé d'entiers consécutifs à partir de 1 et disposés en colonnes de la manière suivante :

— on lit les n premiers entiers à partir de 1 dans la première colonne,

— on lit les $n-1$ entiers suivants dans la 2^e colonne à partir de la 2^e ligne,

— et ainsi de suite jusqu'à la dernière colonne qui ne contient qu'un seul entier.

On demande d'écrire un code qui pour un entier `n` ≥ 1 donné construise la liste de listes d'entiers représentant le triangle de Floyd vertical à `n` lignes. Par exemple, pour `n` $= 4$, le motif est

```
1
2 5
3 6 8
4 7 9 10
```

et la liste recherchée est :

```
[[1], [2, 5], [3, 6, 8], [4, 7, 9, 10]]
```

Transformation du boustrophédon

On demande de construire un triangle de nombres dont voici les 8 premières lignes :

```
1
0   1
1   1   0
0   1   2   2
5   5   4   2   0
0   5   10  14  16  16
61  61  56  46  32  16  0
0   61  122 178 224 256 272 272
```

La construction du triangle suit le procédé d'écriture précisé ci-dessous :

— Chaque ligne contient autant de nombres que sa position, ie la `n`-ème ligne contient `n` entiers

— Les lignes de rang pair s'écrivent de gauche à droite, les lignes de rang impair de droite à gauche

— La première ligne contient uniquement 1

— Dans chaque ligne à partir de la 2^e, le premier nombre écrit est 0

— Chaque nombre de chaque ligne, à partir du 2^e, s'obtient en ajoutant le dernier que l'on a écrit et celui qui se trouve derrière et au-dessus du nombre à écrire.

Dans le tableau ci-dessus, par exemple :

— 7^e ligne : $46 = 32 + 16$

— 8^e ligne : $178 = 122 + 46$.

On écrira une fonction `boustrophedon(n)` qui affichera les `n` premières lignes du triangle.

Rotation horaire d'une matrice carrée

Ecrire une fonction `rotate(M)` qui étant donné une matrice carrée `M` renvoie la matrice déduite de `M` par rotation d'un quart de tour dans le sens des aiguilles d'une montre.

Par exemple, voici une matrice `M` et la matrice `rotate(M)` :

$$M = \begin{pmatrix} 23 & 51 & 75 & 50 & 56 & 94 \\ 52 & 93 & 97 & 17 & 99 & 79 \\ 21 & 33 & 61 & 97 & 82 & 10 \\ 42 & 69 & 20 & 18 & 80 & 69 \\ 54 & 58 & 54 & 86 & 20 & 44 \\ 23 & 12 & 41 & 53 & 27 & 78 \end{pmatrix} \quad N = \begin{pmatrix} 23 & 54 & 42 & 21 & 52 & 23 \\ 12 & 58 & 69 & 33 & 93 & 51 \\ 41 & 54 & 20 & 61 & 97 & 75 \\ 53 & 86 & 18 & 97 & 17 & 50 \\ 27 & 20 & 80 & 82 & 99 & 56 \\ 78 & 44 & 69 & 10 & 79 & 94 \end{pmatrix}$$

(cf. ci-dessous pour pouvoir récupérer les valeurs de cet exemple).

En déduire une fonction `antiRotate(M)` qui renvoie la matrice déduite de `M` par rotation d'un quart de tour dans le sens *inverse* des aiguilles d'une montre.

Ci-dessous, les valeurs utilisables dans un code Python de la matrice qui a été donnée en exemple :

```
[[23, 51, 75, 50, 56, 94],
 [52, 93, 97, 17, 99, 79],
 [21, 33, 61, 97, 82, 10],
 [42, 69, 20, 18, 80, 69],
 [54, 58, 54, 86, 20, 44],
 [23, 12, 41, 53, 27, 78]]
```

Remplir une matrice en diagonale

(1) Écrire une fonction `remplirDiagonale(A, lig, col, k)`

- qui remplit, par les entiers consécutifs à partir de l'entier `k`, la partie de la diagonale montante de `A` commençant à la position d'indices `(lig, col)`
- qui renvoie la dernière valeur inscrite dans la diagonale.

Par exemple, si `A` est la matrice nulle de taille 8×9 alors `remplirDiagonale(A, 5, 2, 81)` renverra 86 et la matrice `A` sera remplie comme suit :

```
0 0  0  0  0  0  0 86 0
0 0  0  0  0  0 85  0 0
0 0  0  0  0 84  0  0 0
0 0  0  0 83  0  0  0 0
0 0  0 82  0  0  0  0 0
0 0 81  0  0  0  0  0 0
0 0  0  0  0  0  0  0 0
0 0  0  0  0  0  0  0 0
```

On utilisera une boucle `while` dont la condition garantira que la progression sur la diagonale reste bien à l'*intérieur* de la matrice.

(2) Écrire une fonction `remplirEnDiagonales(n, p)`, qui à partir de deux entiers $n, p > 0$, construit la matrice `M` de taille $n \times p$ et remplit, suivant ses diagonales montantes, par les entiers consécutifs $0, 1, 2, \ldots, np-1$.

Par exemple, si $n = 5$ et $p = 4$, la matrice `M` à obtenir doit s'afficher comme suit :

```
 0  2  5  9
 1  4  8 13
 3  7 12 16
 6 11 15 18
10 14 17 19
```

Somme des carrés des éléments d'une matrice

1. Ecrire une fonction `f(M)` qui renvoie la somme des carrés de chacun des termes d'une matrice carrée `M`.
2. Ecrire une fonction `g(M)` qui renvoie la trace de la matrice $M\,{}^tM$ où ${}^tM$ désigne la transposée de `M`. On rappelle que la trace d'une matrice carrée est la somme des éléments de sa diagonale principale.
3. En testant sur 100 matrices carrées aléatoires `M`, de taille aléatoire `n` $\in \{1, \dots, 10\}$, vérifier que les nombres `f(M)` et `g(M)` sont égaux.

Matrice à symétrie verticale

1. Ecrire une fonction `colEgales(M, j, k)` où `M` est une matrice et qui renvoie `True` si les colonnes de `M` d'indices `j` et `k` sont égales et `False` sinon. Par exemple, si `M` est la matrice ci-dessous :

```
1 5 2 5
0 4 3 4
2 8 9 8
0 4 6 4
```

   alors `colEgales(M, 1, 3)` renvoie `True`.
2. Observez les matrices suivantes ; les deux premières admettent une symétrie verticale mais pas la dernière :

```
1 5 3 5 1
0 4 6 4 0
2 8 0 8 2
0 4 5 4 0
```

```
1 5 5 1
0 4 4 0
2 8 8 2
0 4 4 0
```

```
1 1
4 0
```

   Ecrire une fonction `estSymVert(M)` qui renvoie `True` si `M` admet une symétrie verticale et `False` sinon. Tester et afficher.

Border une matrice par des zéros

Soit `A` une matrice. Ecrire une fonction `b(A)` qui renvoie la matrice `B` obtenue à partir de `A` en bordant de `0` la matrice `A`. Par exemple, si `A` est la matrice :

```
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
```

alors `b(A)` est la matrice suivante

```
0  0  0  0  0  0 0
0 23  5  7 14 16 0
0  4  6 13 20 22 0
0 10 12 19 21  3 0
0 11 18 25  2  9 0
0  0  0  0  0  0 0
```

Retirer le bord d'une matrice

Soit `A` une matrice. Ecrire une fonction `r(A)` qui renvoie la matrice `B` obtenue à partir de `A` en retirant le pourtour de `A` ie la première et la dernière colonne et la première et la dernière ligne. Par exemple, si `A` est la matrice :

```
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
```

alors `r(A)` est la matrice suivante

```
 6 13 20
12 19 21
```

Matrice magique

On donne une matrice de taille `n` $\times$ `p`. On note `N` $=$ `np`, par exemple, si `n` $= 2$ et `p` $= 3$ alors `N` $= 6$. On dit qu'une matrice `A` de taille `n` $\times$ `p` est *bien remplie* si ses coefficients sont tous les entiers entre 1 et `N`. Par exemple, la matrice suivante, de taille 2×3 est bien remplie :

```
4 1 6
2 5 3
```

1. Ecrire une fonction qui `f(A)` qui dit si une matrice `A` est bien remplie.
2. Une matrice carrée de taille `n`$\times$ est dite *magique* si les $(2\texttt{n} + 2)$ sommes suivantes sont toutes égales entre elles :
   - la somme de n'importe quelle ligne
   - la somme de n'importe quelle colonne
   - la somme de l'une des deux diagonales

   Par exemple, la matrice suivante est magique :

   ```
   17 24  1  8 15
   23  5  7 14 16
    4  6 13 20 22
   10 12 19 21  3
   11 18 25  2  9
   ```

   Ecrire une fonction `estMagique(A)` qui dit si une matrice `A` est magique.

Demi-carré en mode texte de hauteur variable

On donne un entier $n \geq 0$. On demande d'écrire un code qui affiche un demi-carré rempli par des caractères `X` et s'étendant sur n lignes.

Par exemple, si $n = 5$, le code doit afficher :

```
X
X X
X X X
X X X X
X X X X X
```

Générer un motif en forme de croix

1. On vous donne un entier *impair* $n > 1$ et vous devez générer un motif rempli d'un zéro et de uns. Par exemple, si $n = 9$, le motif a la forme suivante :

   ```
   1 1 1 1 0 1 1 1 1
   ```

   Ecrire une fonction `ligne(n)` qui affiche le motif. On pourra supposer que `n` est toujours impair. Le motif est une ligne formée de n chiffres valant 0 ou 1. Deux chiffres seront toujours séparés d'un unique espace. Le chiffre au milieu de la ligne vaut 0 et les autres valent 1.
2. Vous allez devoir créer une fonction `croix(n)` qui devra invoquer la fonction précédente pour générer un motif en forme de croix décrit ci-après. On part d'un entier impair $n > 1$ et le motif est de forme carrée, rempli de zéros ou de uns, en sorte que les zéros forment une croix. Par exemple, si $n = 9$, le motif est généré par l'appel `croix(9)` :

   ```
   1 1 1 1 0 1 1 1 1
   1 1 1 1 0 1 1 1 1
   1 1 1 1 0 1 1 1 1
   1 1 1 1 0 1 1 1 1
   0 0 0 0 0 0 0 0 0
   1 1 1 1 0 1 1 1 1
   1 1 1 1 0 1 1 1 1
   1 1 1 1 0 1 1 1 1
   1 1 1 1 0 1 1 1 1
   ```

   Le motif est un carré formé de $n \times n$ chiffres valant 0 ou 1. Deux chiffres seront toujours séparés d'un unique espace. Les axes vertical et horizontal de symétrie du carré sont formés de 0 et les autres parties du carré sont formées de 1.

Motif triangulaire de nombres

Dans cet exercice, vous allez devoir afficher un motif dépendant d'un entier n. Par exemple, pour $n = 6$ le motif est le carré suivant :

```
1 0 0 0 0 0
1 2 0 0 0 0
1 2 3 0 0 0
1 2 3 4 0 0
1 2 3 4 5 0
1 2 3 4 5 6
```

Ecrire une procédure `motif` prenant un paramètre entier $n \geq 0$ et qui affiche une suite de n lignes telle que la k-ième ligne (pour chaque k entre 1 et n) soit formée de n nombres entiers organisés de la façon suivante :

— chaque ligne commence par la suite des k premiers entiers : 1, 2, etc. jusqu'à k
— la ligne est ensuite complétée par des zéros.

Les nombres dans chaque ligne seront séparés par une espace.
L'exemple ci-dessus correspond à ce qu'affiche `motif(6)`.

Triangle d'entiers consécutifs

On demande d'afficher un motif de n lignes, par exemple pour $n = 5$ le motif sera :

```
1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
```

Plus précisément, écrire une fonction `triangle_entiers(n)` qui affiche `n` lignes formées d'entiers disposés de la manière suivante :
la k-ème ligne énumère les entiers de 1 à `k` dans l'ordre croissant puis les énumère en décroissant jusqu'à 1.

Suite de Prouhet

La suite de Prouhet est une suite de motifs formée de 0 et de 1. Ci-dessous, les 4 premiers motifs :

```
0
0 1
0 1 1 0
0 1 1 0 1 0 0 1
```

Cette suite est une suite de motifs composés de 0 et de 1 de la manière suivante : chaque motif T s'obtient à partir du précédent U en adjoignant à U la suite V obtenue à partir de U en échangeant tout 0 en 1 et tout 1 en 0.

Par exemple, ci-dessus, la suite `0 1 1 0 1 0 0 1` est obtenue :

— à partir de la suite précédente `U = 0 1 1 0`
— en construisant, à partir de la suite `U`, la suite inversée `V = 1 0 0 1`
— en mettant bout à bout `U` et `V`

1. Ecrire une fonction `prouhet(n)` qui renvoie sous forme de liste de 0 et de 1, le i-ème motif de la suite de Prouhet. Par exemple, `prouhet(4)` vaudra la liste `[0, 1, 1, 0, 1, 0, 0, 1]`.
2. Ecrire une fonction `afficher_liste(L)` qui affiche horizontalement une liste `L` de nombres en séparant deux nombres voisins dans la liste par un espace.
3. Afficher les 5 premières lignes de la suite de Prouhet avec la fonction `afficher_liste`.

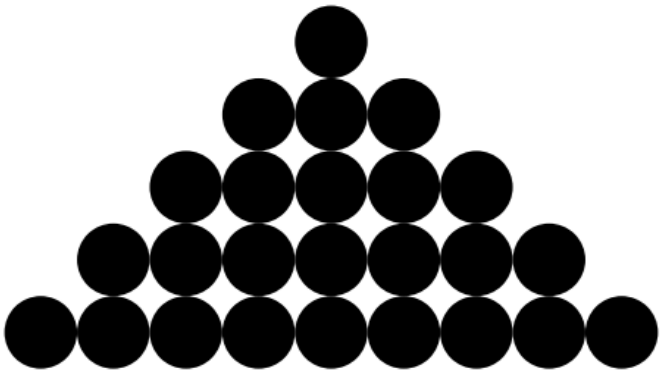
Pyramide de disques

On cherche à dessiner une pyramide en utilisant une fonction qui va afficher les lignes de la pyramide.

1. Ecrire une fonction `f` qui dessine, à partir d'une position donnée un alignement horizontal de `n` disques tangents, de couleur noire et tous de même rayon `r`. Plus précisément, `f` prendra en paramètres `x`, `y`, `n` et `r` où le couple `(x, y)` désigne les coordonnées du centre du premier

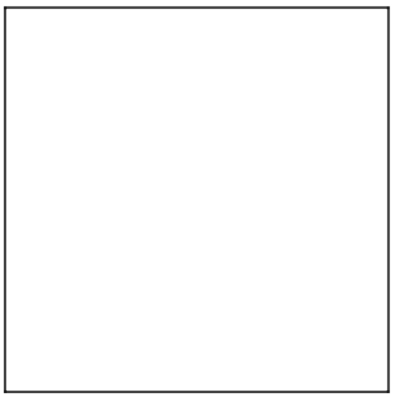
disque (le plus à gauche), `n` est le nombre de disques à dessiner et `r` le rayon de chaque disque.

(2) Ecrire une fonction `g(h, r)` qui dessine une pyramide de base horizontale, formée d'une hauteur de `h` disques, tous de rayon `r` . Par exemple, `g(5, 20)` affichera la pyramide ci-contre. La fonction `g` appelera la fonction `f`.

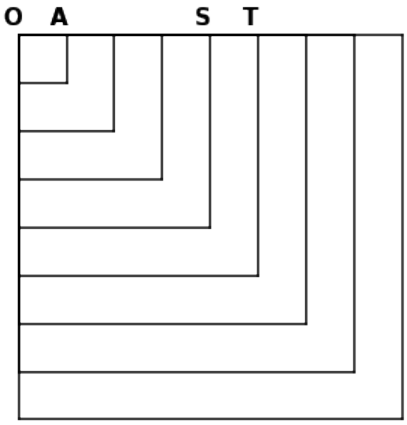


Carrés emboîtés

(1) Cette question a pour objet de faire dessiner un carré de côté de longueur donnée et dont un des sommets est $(0, 0)$. Plus précisément, écrire une fonction `carre(a)` qui prend en paramètre un entier $a > 0$ et dessine le carré de côtés parallèles aux axes, dont le sommet en haut à gauche est $(0, 0)$ et de côté de longueur a. Par exemple, un appel tel que `carre(100)` doit dessiner une figure comme ci-dessous :



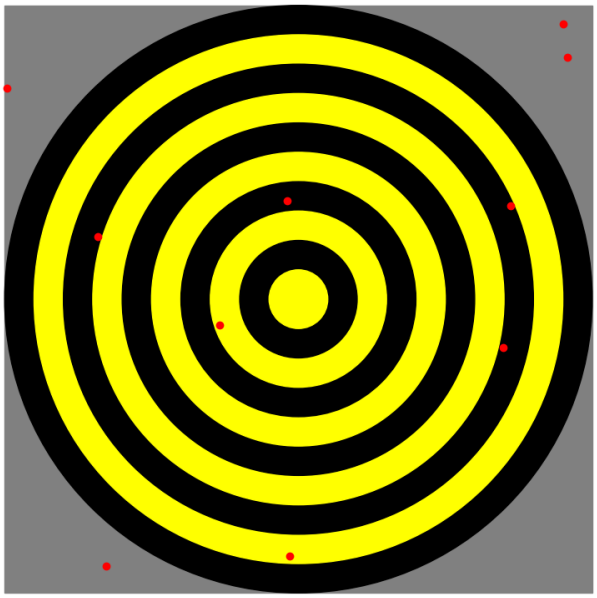
(2) Utiliser la fonction `carre(a)` pour dessiner un motif formé de n carrés emboîtés comme ci-dessous. Le côté OA du carré initial ainsi que les distances, comme la distance ST, entre deux carrés successifs, seront toutes identiques à une valeur c que vous choisirez librement. Dans le dessin ci-dessous, on a choisi $n = 8$ et $c = 15$.



Paintball

Cet exercice simule un séquence de tirs à billes sur un carton portant une cible :

1. Dessiner une cible bicolore de couleurs jaune et noir alternées constituée de n disques concentriques. Le disque central est de couleur jaune et de rayon `r` et le rayon de chaque disque augmente de `r` en même temps qu'il change de couleur. La cible est parfaitement ajustée dans un carton carré de fond gris. On écrira une fonction `cible(n, r)`.
2. Un joueur tire au hasard à l'intérieur du carton (ou sur le bord du carton) N billes de couleur rouge. Ecrire une fonction `tirs(N, R)` qui dessine le carton, la cible et les impacts des billes. On utilisera la fonction `uniforme(a, b)` du module `random` qui génère un flottant aléatoire entre les deux valeurs `a` et `b`
   Voici un exemple de sortie attendue :



(3) Un joueur effectue `N` tirs aléatoires dans le carton. On vous demande d'écrire de modifier la fonction précédente pour qu'elle montre la cible mais aussi calcule et affiche le nombre de billes qui ont atteint la cible, autrement dit qui sont dans une zone colorée.
On rappelle que la distance de l'origine $(0, 0)$ au point de coordonnées (x, y) est donnée par la formule $\sqrt{x^2 + y^2}$
Dans l'exemple présenté à la question 2, le nombre à trouver est 6.

Déplacement vertical

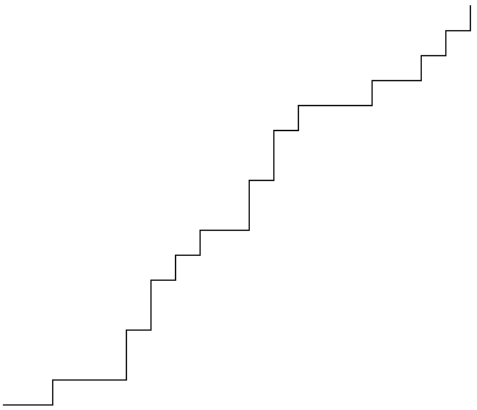
Dans cet exercice, on demande de dessiner le mouvement d'un mobile quand il suit des directions données dans une liste.

Plus précisément, on demande d'écrire une fonction `bouger(L)` qui prend en argument une liste `L` formée de nombres valant 1 ou 2. Le nombre 1 représente un déplacement vers la droite de 20 pixels et le nombre 2 représente un déplacement vers le haut de 20 pixels. Un appel `bouger(L)` provoque le déplacement du mobile suivant les différentes valeurs de `L`. Le mobile commence son déplacement en $(0, 0)$.

Pour la liste des 35 déplacements ci-dessous :

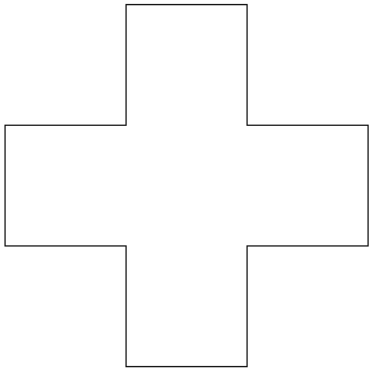
```
L=[1, 1, 2, 1, 1, 1, 2, 2, 1, 2, 2,
   1, 2, 1, 2, 1, 1, 2, 2, 1, 2, 2,
   1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 2, 1, 2]
```

la trajectoire aura la forme suivante :

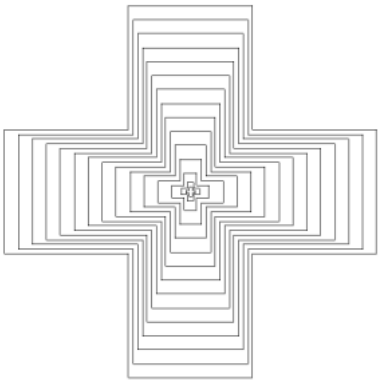


Signes *plus* emboîtés

(1) Créer une fonction `croix(r)` qui dessine une croix de centre $(0, 0)$ et dont tous les côtés mesurent une distance de `r`. Un côté est la distance entre deux sommets consécutifs et une croix est formée de 12 côtés de même longueur. Ainsi, chaque branche de la croix aura une longueur et une largeur valant `r`.

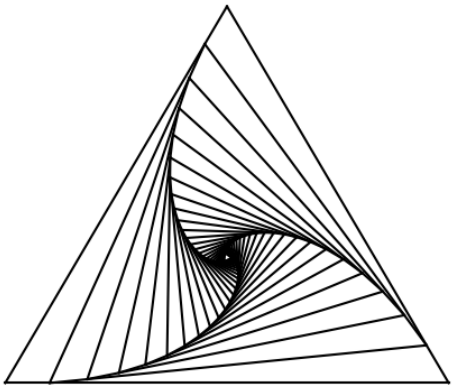


(2) Utiliser la fonction `croix` pour générer un motif de `n` croix emboîtées. L'espacement entre deux croix successives sera placé dans une variable `esp`. Les dessins ci-dessous ont été obtenus pour `n=15` et une valeur initiale de `r=200`.

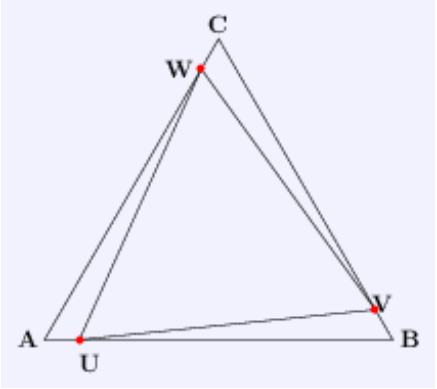


Triangles emboîtés

Construire un emboîtements de triangles comme dans la figure ci-dessous.



Le principe est simple : étant donné un triangle ABC, on construit un autre triangle UVW tel que U soit sur le segment AB et tel que $AU = AB/10$ et de même pour les autres côtés :



Il suffit ensuite de répéter cette construction avec le triangle UVW.

On aura besoin de la transformation `T(P,Q)` qui renvoie les coordonnées du point placé à 1/10 de PQ à partir de P :

```
def T(P, Q):
    return (P[0]+(Q[0]-P[0])/10, P[1]+(Q[1]-P[1])/10)
```

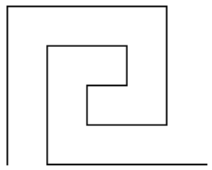
On écrira une fonction `emboiter(A, B, C)` qui étant donné un triangle `ABC` comme ci-dessus dessine le triangle `UVW` et renvoie la liste des coordonnées de `U`, `V` et `W`.

Dans le dessin ci-dessus, la fonction `emboiter` a été appelée 25 fois. Pour des raisons esthétiques, on préférera un triangle équilatéral pour triangle initial.

Dessiner une frise

Cet exercice codé en Turtle est corrigé en vidéo.

1. Ecrire une fonction `motif(x, y, e)` qui dessine la ligne polygonale ci-dessous

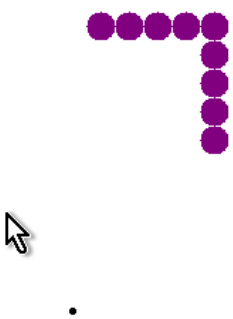


On supposera que tout écart entre deux segments parallèles voisins est constant, stocké dans la variable `e`. La largeur totale est donc de `5e` et la hauteur totale de `4e`.
Le motif commence au point de coordonnées `(x, y)`.

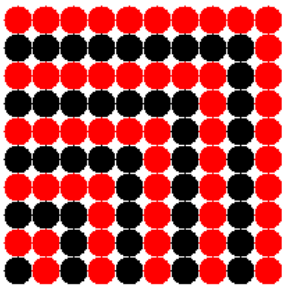
2. En déduire un programme de tracé de n frises comme ci-dessous où $n = 6$:

Equerres de disques

1. Ecrire une fonction `equerreDessiner(x,y, n, color)` une équerre formée de $2n - 1$ disques de diamètre `d=20` et de couleur `color` et dont l'angle droit est positionné au point (x, y). Exemple pour `n=5`, `(x, y)=(100, 200)` et `color=purple` (pour bien comprendre, une bulle noire a été placée en `(0,0)`) :



(2) Dessiner un motif $n \times n$ formé d'équerres remplis de disques, les équerres étant alternativement rouges et noires. On utilisera la question précédente.



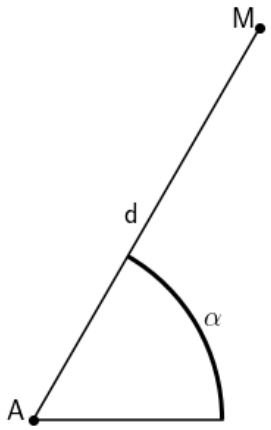
Logo Google Drive

Dessiner le logo de Google Drive :

Les couleurs sont le bleu par défaut de Matplotlib, la couleur orange et la couleur vert. Tous les angles visibles sont de 60° ou de 120°. Chaque bande est un parallélogramme dont le grand côté est de longueur double du petit. Chaque bande sera dessinée avec `Polygone` et l'option `color`.

Indication

On utilisera le résultat décrit plus bas et dont les données sont fournies par la figure suivante :



Le point `A` $= (\texttt{x}_\texttt{A}, \texttt{y}_\texttt{A})$ étant donné ainsi qu'un angle α (en radians pour Python) et une distance `d` alors le point `M` tel que `AM` $=$ `d` et $(\overrightarrow{i}, \overrightarrow{AM}) = \alpha$ a pour coordonnées :

$$\texttt{x}_\texttt{M} = \texttt{x}_\texttt{A} + \texttt{d}\cos\alpha \quad \texttt{y}_\texttt{M} = \texttt{y}_\texttt{A} + \texttt{d}\sin\alpha$$

où $\overrightarrow{i}$ est le vecteur unitaire de l'axe des abscisses.

On importera du module `math` les fonctions `cos`, `sin` et la constante `pi` et on construira dans le code Python une fonction `extremite(A, d, alpha)` renvoyant les coordonnées de `M` défini ci-dessus et qu'on utilisera pour construire la figure.

Construire un histogramme

Dans cet exercice, une *note* est un nombre entier entre 0 et 20 (observer que cela fait 21 notes possibles). Le but de l'exercice est de dessiner, sans utiliser les possibilités spécifiques qu'offre Matplotlib, un histogramme de notes.

(1) Écrire une fonction `effectifs_par_notes` qui prend en paramètre une liste `L` constituée d'un nombre indéterminé de notes et renvoie une liste `t` de 21 entiers tels que `t[v]` soit le nombre de notes valant `v` dans la liste `L`.
Ainsi, pour `L=[8,13,12,8,6,19]` on aura, par exemple, `t[12]=1` et `t[8]=2`. Autrement dit, `t[v]` est le nombre d'occurrences de la note `v` dans la liste `L`. Au départ, `t` sera une liste composée de 21 zéros.
Pour tester, on pourra utiliser la liste suivante :

```
notes=[14, 8, 17, 13, 16, 15, 12, 0, 5, 17, 10, 18,
17, 7, 16, 1, 13, 2, 14, 11, 5,
7, 7,11, 1, 10, 13, 2, 13, 12,
3, 6, 12, 14, 19]
```

(2) Tracer un axe vertical et un axe horizontal avec 21 « bulles » représentant toutes les notes possibles (de 0 à 20). L'axe vertical est là seulement pour permettre de mieux lire l'histogramme.

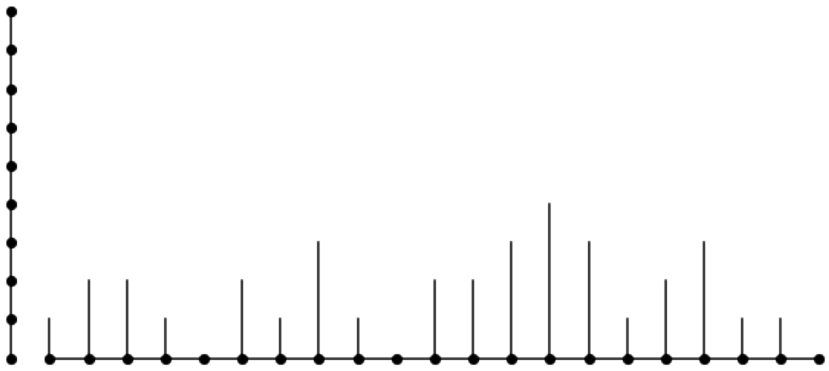
(3) Ecrire une fonction `baton` qui prend en paramètre un entier `n` et une hauteur `h` et dessine un bâton de hauteur `h` au-dessus du point d'abscisse correspondant à la note `n` (il faut au préalable tracer les axes horizontal et vertical).

3) On se donne une liste `L` de notes. Dessiner l'histogramme de `L`. On rappelle que l'histogramme de `L` est constitué des segments verticaux issus de chaque note `n` et de hauteur le nombre d'occurrences de la note `n` dans la liste `L`.

Si la liste est par exemple constituée des 35 notes suivantes :

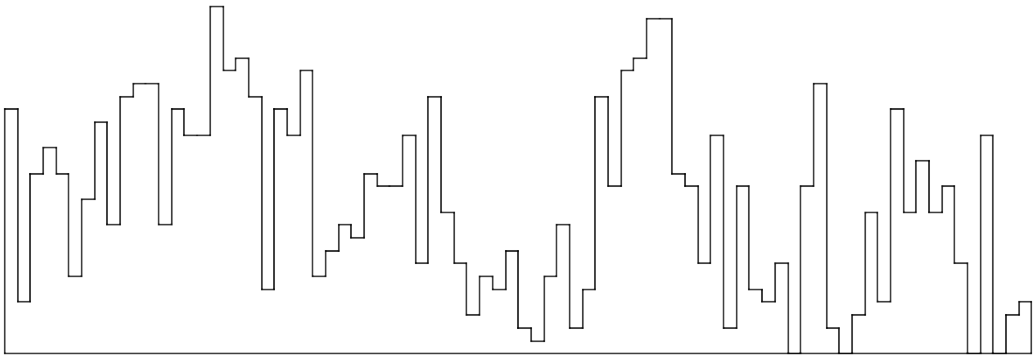
```
14, 8, 17, 13, 16, 15, 12,
0, 5, 17, 10, 18, 17, 7,
16, 1, 13, 2, 14, 11, 5,
7, 7, 11, 1, 10, 13, 2,
13, 12, 3, 6, 12, 14, 19
```

alors l'histogramme pourra ressembler à ceci



Remplissage de bassins

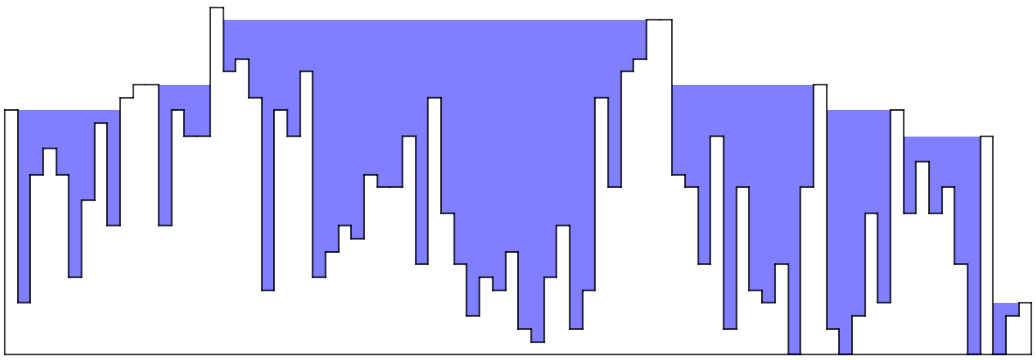
On dispose d'une succession de colonnes dont les hauteurs sont données dans une liste d'entiers positifs. Cette succession définit un relief :



Ce relief a été obtenu pour la liste suivante :

```
[19, 4, 14, 16, 14, 6, 12, 18, 10, 20, 21, 21, 10,
19, 17, 17, 27, 22, 23, 20, 5, 19, 17, 22, 6, 8, 10,
9, 14, 13, 13, 17, 7, 20, 11, 7, 3, 6, 5, 8, 2, 1, 6,
10, 2, 5, 20, 13, 22, 23, 26, 26, 14, 13, 7, 17, 2,
13, 5, 4, 7, 0, 13, 21, 2, 0, 3, 11, 4, 19, 11, 15,
11, 13, 7, 0, 17, 0, 3, 4]
```

Lorsqu'il se met à pleuvoir, l'eau peut remplir des cavités dont les parois sont des colonnes. On demande de dessiner les bassins remplis par la pluie lorsque la quantité d'eau retenue est maximale. Avec le motif précédent, le dessin à obtenir est le suivant :



Pour cela, on déterminera successivement chaque bassin par sa paroi à gauche et sa paroi à droite en parcourant la liste des hauteurs des colonnes à partir de la gauche. Plus précisément, on se

donnera une paroi de gauche d'un bassin à remplir et de contenance maximale (en supposant qu'un tel bassin existe). Cette paroi a un flanc situé entre deux hauteurs `h1` et `h2` avec `h1` $\leq$ `h2`. On cherche alors la paroi située à sa droite, de hauteur la plus haute possible, inférieure à `h2` et valant au moins `h1`.

Le code à fournir contient deux parties :

— le calcul des parois de chaque bassin

— le tracé du relief puis des bassins

Il sera utile de créer une ou plusieurs fonctions.

Cet exercice provient d'une question sur le forum Python d'OpenClassrooms. Le problème est connu sous le nom de Trapping Rain Water et admet une solution astucieuse de complexité linéaire.

Fizz-buzz en VF

Ecrire une procédure `f(n)` où `n` $\geq$ `1` est un entier et qui affiche les entiers de `1` à `n` avec les contraintes suivantes :

— si le nombre à afficher est multiple de 3, au lieu de l'afficher, la procédure affiche le mot `coco` ;

— si le nombre est multiple de 4, la procédure affiche seulement le mot `rico` ;

— si le nombre est multiple, à la fois, de 3 et de 4, la procédure affichera uniquement le mot `cocorico`.

Les affichages seront effectués sur des lignes distinctes.

Par exemple, l'appel `f(13)` doit afficher les lignes ci-dessous :

```
1
2
coco
rico
5
coco
7
rico
coco
10
11
cocorico
13
```

Cet exercice s'appelle le Fizz-buzz a été popularisé en 2007. Pour l'évaluation de l'exercice, la qualité du code, et pas seulement le résultat, sera prise en compte.

Inspirez, expirez !

La respiration carrée (*Sama Vritti*) est un exercice de respiration consistant à alterner de narine quand on respire. Le cycle est le suivant :

— on inspire par la narine droite

— on pause la respiration

— on expire par la narine gauche

— on pause la respiration
— on inspire par la narine gauche
— on pause la respiration
— on expire par la narine droite
— on pause la respiration.

On demande d'écrire une fonction `respirer(n, debut)` qui affiche une suite de `n` cycles dont on connaît l'étape de départ (une inspiration ou une expiration de côté donné). L'inspiration est codée par `I`, l'expiration par `E`, le côté droit par `D`, le côté gauche par `G`. La pause de la respiration sera affichée par `P`. On affichera une étape par ligne et la fin d'un cycle sera suivie de l'affichage d'une ligne de tirets. Par exemple, `respirer(3, "ID")` affichera

```
ID
P
EG
P
IG
P
ED
P
------
ID
P
EG
P
IG
P
ED
P
------
ID
P
EG
P
IG
P
ED
P
------
```

Suite de Syracuse

(1) Soit la fonction f définie pour $n > 0$ entier par :
   — $f(n)$ est la moitié de n si n est pair
   — $f(n)$ vaut $3n + 1$ si n est impair

   Par exemple `f(13) = 40` ou encore `f(10) = 5`.
   Coder en Python cette fonction. Tester avec $f(13)$ et $f(10)$.
(2) On itère la fonction f en partant d'un entier $n > 0$. Par exemple, si $n = 13$, on obtient :

```
13 -> 40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
```

La conjecture de Syracuse affirme que si on itère f depuis n'importe quel entier $n > 0$ alors, on tombera forcément sur 1 (ce qui se produit bien pour l'exemple ci-dessus).
Écrire une fonction `seq` qui

- prend n en paramètre,
- affiche les itérés de la suite depuis le premier itéré, à savoir n jusqu'au dernier, à savoir 1.

(3) Écrire une fonction `saut(x)` qui renvoie le nombre d'itérations pour arriver à 1 quand la suite commence avec x (si $x = 13$, il y a donc 9 itérations pour arriver à 1).

(4) Déterminer à l'aide de la fonction fonction `saut(x)` le plus petit entier n tel qu'il faille exactement 100 itérations à partir de n pour arriver à 1 (vous devez trouver $n = 107$).

Fibonacci

Voici les 12 premiers termes de la suite de Fibonacci :

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144$$

Plus généralement, les deux premiers termes de cette suite sont 1 et encore 1 et, chaque terme de la suite à partir du troisième s'obtient en faisant la somme des deux précédents. Par exemple, ci-dessus, $89 = 34 + 55$.

(1) Le but de cette question est juste de permettre facilement de visualiser les termes de la suite. Écrire une fonction `fibo_liste(n)` qui, pour un entier $n \geq 2$, construit la liste des n premiers termes de la suite de Fibonacci. On utilisera la méthode `append`. Tester en affichant la liste.

(2) *Cette question n'utilise pas la question précédente.*
Ecrire une fonction `fibo` qui renvoie le n-ième terme de la suite de Fibonacci. Par exemple, `fibo(12)` vaut 144. On observera qu'il est totalement INUTILE de *stocker* TOUS les termes de la suite antérieurs au terme à calculer, il suffit juste de disposer des deux termes qui précèdent le terme recherché.

Multiplication du paysan russe (boucle `while`)

La multiplication dite « du paysan russe » ramène une multiplication de deux entiers naturels à une suite d'additions ou de divisions par 2. En voici une illustation à travers le produit $85 \times 18 = 1530$:

```
85      18
------------------------
84      18      18
------------------------
42      36
------------------------
21      72
------------------------
20      72      90
------------------------
10     144
------------------------
5      288
------------------------
4      288     378
------------------------
2      576
------------------------
1     1152
------------------------
0     1152    1530
```

Cette multiplication est basée sur la propriété suivante : si $a, b \in \mathbb{N}$ alors

$$a \times b = \begin{cases} 0 & \text{si } a = 0 \\ a/2 \times 2b & \text{si a est pair et non nul} \\ (a-1) \times b + b & \text{sinon} \end{cases}$$

Dans l'illustration ci-dessus, la 3e colonne est remplie lorsque a est impair.

Ecrire un fonction `paysan(a, b)` qui implémente la multiplication ci-dessus en utilisant une boucle while. Tester en choisissant des entiers aléatoires.

Suite d'entiers

On définit une suite de nombres entiers de la manière suivante :

- le premier élement vaut 0
- le deuxième élément vaut 1
- pour obtenir l'élément `v` au rang `r` ≥ 3, on considère l'élément `x` au rang `r-2`; si `r` $- 1$ est un multiple de 3 alors `v` $=$ `x` et sinon `v` $=$ `x` $+ 1$.

Ainsi, les 13 premiers éléments de la suite sont donnés dans le tableau suivant :

r	1	2	3	4	5	6	7	8	9	10	11	12	13
v	1	0	1	1	1	1	2	1	2	2	2	2	3

Par exemple :

— le 12^e élément vaut 2 car le 10^e élément vaut 2 et que $12 - 1 = 11$ n'est pas un multiple de 3

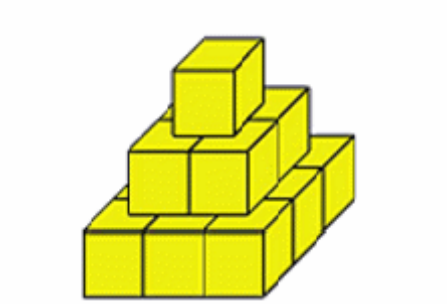
— le 13^e élément vaut 3 car $13 - 1 = 12$ est un multiple de 3 et le 11^e élément vaut 2.

On donne un rang `r` et on demande de calculer la valeur `v` de la liste qui est au rang `r`. Par exemple, si `r = 2030` on trouvera que `v = 338`. On écrira une fonction `f(r)`. Le code à écrire ne nécessite de construire aucune liste.

La suite `f(r)` est répertoriée ICI.

Pyramide de cubes

On dispose de cubes pour monter une pyramide. Voici par exemple une pyramide de hauteur 3 :



D'une manière générale, le dernier étage de la pyramide est formé d'un seul cube ; ce cube repose sur un carré de 2x2 cubes qui repose lui-même sur un carré de 3x3 cubes et ainsi de suite jusqu'au premier niveau.

On dispose de `N` cubes et on cherche à connaître :

— la hauteur de la plus haute pyramide que l'on peut construire avec ces `N` cubes

— le nombre de cubes utilisés.

On écrira une fonction `pyramide(N)` qui renverra une liste `[h, t]` où `h` est la hauteur de la pyramide et `t` le nombre total de cubes utilisés pour la construire.

Par exemple, `pyramides(42)` renvoie `[4, 30]`. En effet, une pyramide de hauteur 4 nécessite 30 cubes ce qui est moins que 42 mais une pyramide de hauteur 5 nécessite 55 cubes ce qui dépasse 42.

De même, `pyramides(5)` renvoie `[2, 5]` car une pyramide de hauteur 2 nécessite pile 5 cubes.

Cet exercice reprend l'exercice du site france-ioi intitulé Département d'architecture : construction d'une pyramide

Sous-listes d'une liste

On donne une liste `L` d'entiers entre 0 et 9, par exemple

$$\texttt{L} = [5, 5, 0, 8, 1, 4, 2, 3, 7, 8, 1, 4, 2, 3, 3]$$

On observe que cette liste contient exactement à deux reprises la sous-liste `M = [8, 1, 4, 2]` : en effet, les chiffres 8, 1, 4, 2 apparaissent les uns à la suite des autres dans `L` en exactement deux endroits.

Ecrire une fonction `nb_sous_listes(L, M)` qui renvoie le nombre de sous-listes identiques à `M` et qui sont présentes dans `L`. Avec l'exemple ci-dessus, la fonction doit renvoyer 2.

Plus petit entier non nul (découpage en fonctions)

1. Écrire une fonction `mini(L, i)` qui prend en paramètre
   - une liste `L` d'entiers
   - un indice valide positif `i` de la liste `L`

   et renvoie le plus petit des éléments de la liste `L` et ayant un indice supérieur ou égal à `i`.
   Voici quelques exemples de comportement de la fonction :

```
[40, 10, 50, 20] 0 -> 10
[40, 10, 50, 20] 1 -> 10
[40, 10, 50, 20] 2 -> 20
[40, 10, 50, 20] 3 -> 20
```

2. Écrire une fonction `plusPetitNonNul(L)` qui prend en paramètre une liste `L` d'entiers *non tous nuls* et qui renvoie le plus petit indice d'un élément non nul de `L`.
   Voici quelques exemples de comportement de la fonction :

```
[50, 10, 0, 90] -> 0
[0, 0, 70, 0, 42] -> 2
[0, 0, 70] -> 2
[42] -> 0
```

3. En utilisant les deux fonctions précédentes, écrire une fonction `miniNonNul(L)` qui prend en paramètre une liste `L` d'entiers *non tous nuls* et qui renvoie le plus petit élément *non nul* de `L`.
   Voici quelques exemples de comportement de la fonction :

```
[50, 0, 0, 90, 10] -> 10
[0, 80, 10, 60, 0] -> 10
[0, 0, 0, 40, 0, 30, 0] -> 30
[0, 0, 0, 40, 0, 50, 0] -> 40
[0, 0, 0, 40, 0, 0, 0] -> 40
[0, 0, 0, 0, 0, 0, 40] -> 40
[50, 80, 10, 60, 40] -> 10
[42] -> 42
```

Minimum des entiers pairs (découpage en fonctions et boucle `for`)

1. On donne une liste `L` d'entiers et un indice valide de la liste (l'indice est appelé `debut`) tel que `L[debut]` soit pair. Construire une fonction `mini(L, debut)` qui renvoie la valeur du plus petit des éléments **pairs** de la liste `L` et ayant un indice supérieur ou égal à `debut`. Par exemple, si `L=[81, 32, 10, 21, 42, 11, 12, 9, 12, 65, 46]` et si `debut=4` alors `mini(L, debut)` vaut 12.
2. On donne une liste `L` d'entiers contenant au moins un entier pair et on demande d'écrire une fonction `ind_elt_pair(L)` qui renvoie le plus petit indice `i` de `L` tel que `L[i]` soit pair. Par exemple, si `L=[81, 19, 31, 21, 42, 11, 12]` alors `ind_elt_pair(L)` vaudra 4. Il est attendu que votre code utilise une boucle `for`.

3. On donne une liste `L` d'entiers contenant au moins un entier pair. Ecrire une fonction `miniPairs` qui renvoie le plus petit des éléments de `L` qui soit pair. Par exemple :

```
[81, 32, 12, 9, 12, 65, 46] -> 12
[81, 65, 46] -> 46
```

Il est attendu que la fonction `miniPairs` utilise les deux fonctions des questions précédentes.

Minimum des entiers pairs (découpage en fonctions et boucle `while`)

1. On donne une liste `L` d'entiers et un indice valide de la liste (l'indice est appelé `debut`) tel que `L[debut]` soit pair. Construire une fonction `mini(L, debut)` qui renvoie la valeur du plus petit des éléments **pairs** de la liste `L` et ayant un indice supérieur ou égal à `debut`. Par exemple, si `L=[81, 32, 10, 21, 42, 11, 12, 9, 12, 65, 46]` et si `debut=4` alors `mini(L, debut)` vaut 12.
2. On donne une liste `L` d'entiers contenant au moins un entier pair et on demande d'écrire une fonction `ind_elt_pair(L)` qui renvoie le plus petit indice `i` de `L` tel que `L[i]` soit pair. Par exemple, si `L=[81, 19, 31, 21, 42, 11, 12]` alors `ind_elt_pair(L)` vaudra 4. Il est attendu que votre code utilise une boucle `while`.
3. On donne une liste `L` d'entiers contenant au moins un entier pair. Ecrire une fonction `miniPairs` qui renvoie le plus petit des éléments de `L` qui soit pair. Par exemple :

```
[81, 32, 12, 9, 12, 65, 46] -> 12
[81, 65, 46] -> 46
```

Il est attendu que la fonction `miniPairs` utilise les deux fonctions des questions précédentes.

Triangle de Floyd vertical

On donne un entier $n \geq 1$ et on demande de générer un motif dépendant de n. A titre d'illustration, si $n = 9$ le motif est le suivant

```
1
2 10
3 11 18
4 12 19 25
5 13 20 26 31
6 14 21 27 32 36
7 15 22 28 33 37 40
8 16 23 29 34 38 41 43
9 17 24 30 35 39 42 44 45
```

Le motif est un triangle formé de n lignes et de n colonnes et composé d'entiers consécutifs à partir de 1 et disposés en colonnes de la manière suivante :

- on lit les n premiers entiers à partir de 1 dans la première colonne,
- on lit les $n - 1$ entiers suivants dans la 2^e colonne à partir de la 2^e ligne,
- et ainsi de suite jusqu'à la dernière colonne qui ne contient qu'un seul entier.

Indication de résolution : on pourra remarquer que si le motif a n lignes alors

— les éléments de la 2e colonne s'obtiennent en additionnant $n-1$ à chaque élément de la 1re colonne

— les éléments de la 3e colonne s'obtiennent en additionnant $n-2$ à chaque élément de la 2e colonne

— et ainsi de suite.

Tranches de termes d'une liste

On donne un entier p et une liste `L` d'entiers et on appelle `M` la nouvelle liste formée des éléments de `L` qui ne sont ni parmi les p premiers éléments de `L` ni parmi les p derniers éléments de `L`. Ecrire une fonction `trancheElements(p, L)` qui renvoie `M`. On supposera que $0 \leq p \leq n/2$ où n est le nombre d'éléments de `L`.

Voici quelques exemples de comportement de la fonction si `L = [12, 81, 31, 65, 9, 32, 82]` :

```
[12, 81, 31, 65, 9, 32, 82] , 1 -> [81, 31, 65, 9, 32]
[12, 81, 31, 65, 9, 32, 82] , 3 -> [65]
[12, 81, 31, 9, 32, 82] , 3 -> []
[12, 81, 31, 65, 9, 32, 82] , 0 -> [12, 81, 31, 65, 9, 32, 82]
[12, 81] , 1 -> []
```

Zéros qui séparent

On donne une liste `L` d'entiers et on demande d'afficher sur une même ligne toutes les suites d'éléments consécutifs de la liste qui ne contiennent pas zéro. Par exemple, si

```
L=[0, 2, 8, 4, 1, 0, 6, 0, 0, 2, 1, 3, 0, 6, 1, 7, 0, 0]
```

alors l'affichage sera

```
2 8 4 1
6
2 1 3
6 1 7
```

Si `L` est formée uniquement de zéros, aucun affichage ne doit avoir lieu (pas même un saut de ligne).

Tranches de même signe

On donne une liste d'entiers non nuls, par exemple

`L = [-3, -7, -5, -8, 5, 4, 9, -8, 6, -2, -2, 1, 6, 2].`

Cette liste alterne des tranches de nombres ayant tous le même signe. Avec la liste précédente, il y a 6 tranches, énumérées ci- dessous :

— tranche négative : -3, -7, -5, -8,

— tranche positive : 5, 4, 9,

— tranche négative : -8,

— tranche positive : 6,

— tranche négative : -2, -2,

— tranche positive : 1, 6, 2

Les longueurs de ces différentes tranches sont : 4, 3, 1, 1, 2, 3.

Ecrire une fonction `longTranchesMemeSigne(L)` qui prend en paramètre une liste `L` d'entiers et qui renvoie la liste des longueurs des différentes tranches successives formées de nombres successifs de `L` et ayant tous le même signe.

Voici quelques exemples du comportement de la fonction :

```
[-3, -7, -5, -8, 5, 4, 9, -8, 6, -2, -2, 1, 6, 2]
                                -> [4, 3, 1, 1, 2, 3]

[-1, 1] -> [1, 1]
[-1, -1, -1, -1, -1] -> [5]
[42] -> [1]
```

Nombre de changements de signe

1. Ecrire une fonction `signeDifferent(a, b)` qui prend en paramètres deux entiers `a` et `b` non nuls et renvoie True si les nombres sont de signes contraires. Par exemple, `signeDifferent(42, 81)` vaut `True` et `signeDifferent(42, -81)` vaut `False`.
2. On donne une liste d'entiers non nuls et on demande de déterminer le nombre de changements de signe qui apparaissent quand on parcourt la liste.

Liste	Nombre de changements de signe
`[9, 2, 2, 8]`	0
`[3, 6, 8, -1, 7, -2, 8]`	4
`[6]`	0
`[-1, 3, 8, 2, 6, 8, 3]`	1
`[2, -3, -3, 1]`	2
`[9, 6, -3]`	1

Répétitions successives

Soit, par exemple, la liste `L` d'entiers :

`[3, 7, 5, 5, 8, 8, 8, 8, 2, 5, 5, 5]`

Elle contient des répétitions *successives* comme 8, 8, 8, 8 ou encore 5, 5, 5. La liste sans répétition successive est `[3, 7, 5, 8, 2, 5]`.

Plus généralement, écrire une fonction `f(L)`, où `L` est une liste non vide d'entiers, qui renvoie la liste `M` formée des éléments de `L` sans ses répétitions de termes successifs. La liste `M` doit être une *nouvelle* liste et la liste `L` ne doit pas être modifiée par la création de `M`. Voici quelques exemples de listes `L` et de listes `f(L)` :

Liste L	Liste $f(L)$ sans répétitions successives
[3, 7, 5, 5, 8, 8, 8, 8, 2, 5, 5, 5]	[3, 7, 5, 8, 2, 5]
[7, 7, 7]	[7]
[7, 0, 1]	[7, 0, 1]
[7]	[7]

Au plus deux zéros consécutifs

On donne une liste L d'entiers. On demande de générer une liste M contenant les mêmes entiers que L, dans le même ordre sauf que M ne contiendra jamais trois zéros consécutifs. Voici quelques exemples de comportements :

```
[0] -> [0]

[9, 4, 0, 0, 0, 0, 3, 6, 0, 0, 0, 8, 0, 5] ->
[9, 4, 0, 0, 3, 6, 0, 0, 8, 0, 5]

[0] -> [0]

[8, 5, 2] -> [8, 5, 2]

[5, 9, 0] -> [5, 9, 0]

[0, 2, 3, 0, 0, 0, 0, 9, 5, 6, 0, 0, 0] ->
[0, 2, 3, 0, 0, 9, 5, 6, 0, 0]
```

Liste sans doublons (version efficace)

On donne une liste L formée d'au moins 100000 entiers valant **entre 1 et 10 millions**.

Ecrire une fonction `eliminer_doublons(L)` qui partant de la liste L renvoie une liste composée des valeurs de L, mais dans laquelle les multiples valeurs égales de L (s'il y en a) n'ont été copiées qu'une seule fois. Exemples :

Liste	Liste sans doublons
[42, 81, 42, 65, 12, 81, 31, 42]	[42, 81, 65, 12, 31]
[42, 42, 42, 42, 42]	[42]
[42]	[42]

Pour tester, on utilisera une liste L d'entiers aléatoires comme ci-dessous :

```
from random import randrange
L=[randrange(10**7) for _ in range(500000)]
```

Réduction parallèle de somme

Etant donné une liste d'entiers, on veut en faire la somme par réductions successives obtenues en faisant la somme suivant des paires de voisins. Voici un exemple de comment cette réduction est réalisée :

```
 6  0  5  3  7  5  1  8  8  1  2  9  9  1
 6  .  8  . 12  .  9  .  9  . 11  . 10  .
14  .  .  . 21  .  .  . 20  .  .  . 10  .
35  .  .  .  .  .  .  . 30  .  .  .  .  .
65  .  .  .  .  .  .  .  .  .  .  .  .  .
```

On part de la liste L de la première ligne. On fait la somme des éléments deux par deux et à chaque fois, on remplace dans L le premier terme par leur somme. Ci-dessus, et uniquement pour une raison d'intelligibilité de la procédure, les termes de la somme déjà ajoutés sont remplacés par un

point. Et on recommence le procédé mais avec la ligne suivante. Quand la liste ne contient plus qu'un seul élément non pris en compte, ce terme, qui est le premier de la liste, vaut la somme de la liste initiale.

Ecrire une fonction `somme_parallele(L)` qui calcule la somme des éléments de `L` par le procédé décrit ci-dessus. On pourra résoudre suivant un des deux procédés ci-desous.

Méthode 1

— Ecrire une fonction `réduction(L)` qui renvoie la liste des sommes des termes des regroupements disjoints deux termes consécutifs. Par exemple, si `L=[15, 14, 3, 14, 12, 2]` alors `réduction(L)` est la liste `[29, 17, 14]`.

— En déduire une fonction `somme_parallele(L)` qui calcule la somme des éléments de `L` par applications successives de la réduction de la question précédente.

Méthode 2

On pourra observer qu'à la première étape, le premier terme de chaque somme est à un indice multiple de 2, à l'étape suivante à un multiple de 4, puis aux multiples de 8, etc. On pourra utiliser deux boucles `while` imbriquées.

Ce procédé dit de réduction parallèle est utilisé pour calculer une somme de termes d'une liste en programmation parallèle.

Marques sur un axe

On se donne un axe gradué par les entiers 1, 2, 3, etc. ainsi qu'une liste `L` d'entiers valant au moins 1. Initialement, l'axe ne contient aucune marque. On va examiner dans l'ordre croissant des indices, les éléments de la liste `L` et effacer et ajouter des marques sur les graduations de l'axe, comme précisé ci-après. A chaque examen d'un élément `x` de `L` :

— on supprime toutes les marques sur l'axe qui sont à une graduation inférieure ou égale à `x`

— on marque `x` sur l'axe.

On demande de déterminer la liste `P` des marques sur l'axe lorsque la liste `L` a été complètement examinée. On écrira une fonction `marquer(L)` qui renverra `P`.

Illustrons. Pour fixer les idées, on imagine que

```
L = [5, 2, 8, 9, 6, 5, 6, 4]
```

Voici les étapes :

— on marque sur l'axe le premier élément de `L`, donc 5

— l'élément suivant est 2 : il n'y a aucune marque avant 2 sur l'axe donc on n'a rien à effacer et on marque la graduation 2

— l'élément suivant est 8 : on efface les marques précédentes, 2 et 5 (car inférieures à 8), et on marque la graduation 8

— l'élément suivant est 9 : on efface la marque 8 et on marque la graduation 9

— l'élément suivant est 6 : on marque la graduation 6

— l'élément suivant est 5 : on marque la graduation 5

— l'élément suivant est 6 : on efface les marques inférieures ou égales donc 5 et 6 puis on place 6

— le dernier élément est 4 : il n'y a pas de marque avant donc on n'efface rien et on marque 4.

Au total, l'axe comporte les marques suivantes : 4, 6 et 9.

Ecrire un code qui à partir d'une liste `L` d'entiers génère la liste `P` des positions marqués sur l'axe (dans l'ordre croissant). Avec `L` comme ci-dessus, on aura `P=[4, 6, 9]`.

Cet exercice est inspiré de l'exercice Collage d'affiches sur France-IOI.

Plus petit nombre unique

On donne une liste d'entiers triés et on demande d'écrire une fonction `plusPetitUnique(L)` qui renvoie le plus petit entier de la liste qui n'apparaît qu'une seule fois ou, si cet entier n'existe pas, qui renvoie `None`.

Voici quelques comportements de la fonction :

```
[42] → 42
[42, 45] → 42
[42, 42, 45] → 45
[42, 42, 45, 45] → None
[42, 45, 48, 49] → 42
[42, 42, 45, 45, 45, 48, 49, 49, 52] → 48
```

On pourra se débarasser tout de suite des listes ayant 1 ou 2 éléments.

On pourra maintenir un drapeau `etat` qui témoignera de l'état de la liste :

— `etat = 0` lorsque l'élément courant est égal au précédent,

— `etat = 1` lorsque l'élément courant est différent du précédent et était à 0 juste avant,

— `etat = 2` lorsque l'élément courant est différent du précédent et était à 1 juste avant.

Zéros isolés

On donne une liste ayant au moins deux éléments et dont les éléments sont parmi 0 ou 1. On dit qu'un zéro de L est *isolé* si aucun de ses voisins n'est un zéro. Ci-dessous, on trouvera trois exemples.

1er exemple : la liste suivante

```
0 1 1 0 0 0 0 1 0 1 0 1 1 0 0
```

contient 3 zéros isolés : un zéro en 1re position, un zéro en 9e position et un zéro en 11e position

2e exemple : la liste suivante

```
1 0
```

contient 1 zéro isolé.

3e exemple : aucune des trois listes suivantes

```
1 1 0 0 1 0 0 0 1 1 0 0
0 0 0
1 1 1 1
```

ne contient de zéro isolé.

Ecrire une fonction `compterZerosIsoles(L)` qui renvoie le nombre de zéros isolés d'une liste L ayant au moins deux éléments et dont les éléments sont parmi 0 ou 1.

Indices des minimums des entiers pairs

On donne une liste L d'entiers. Ecrire une fonction `indiceminiPairs` qui renvoie la liste des indices des valeurs du plus petit des éléments pairs de la liste. Exemples de comportement :

```
[81, 32, 12, 9, 12, 65, 46] -> [2,4]
[81, 65, 46] -> [2]
[81, 32, 12, 9] -> [2]
[81, 33, 11, 9,  65, 47] -> []
```

Plateau le plus long

On donne une liste d'entiers et on recherche un entier qui est répété consécutivement dans la liste le plus grand nombre de fois (ce qui forme un « plateau »). Par exemple, la liste

```
[4, 8, 8, 8, 8, 7, 7, 7, 7, 7, 9, 4, 4, 4, 4, 4, 5, 5, 5]
```

montre un plateau de quatre 8, de cinq 7, de cinq 4 et trois 5. Donc, une réponse au problème est 7 ou 4. Attention, bien que 4 apparaisse 6 fois, il n'apparait consécutivement que suivant un plateau de longueur 5.

Si tous les nombres sont différents, il n'y a que des plateaux de longueur 1 et le programme peut renvoyer n'importe quel élément de la liste.

On pourra procéder comme suit :

— parcourir la liste avec une boucle `for`

— chaque fois qu'on entre dans un « plateau », on définit un compteur pour ce plateau que l'on incrémente tant qu'on reste sur le plateau

— on surveille si le compteur dépasse la taille maximale déjà atteinte, et si nécessaire, on met à jour et on mémorise le nombre dont la répétition forme le plateau.

Cet exercice provient des demi-finales 2017 du concours Algoréa.

Somme de tous les autres

Cet exercice peut nécessiter une analyse préalable avant de pouvoir être codé efficacement.

On donne une liste L d'entiers, par exemple

```
L = [3, 1, 5, 8]
```

et on demande de modifier L en sorte que chaque élément de L soit remplacé par la somme de tous les éléments de L sauf l'élément lui-même. Dans le cas de la liste ci-dessus, après opérations, L sera :

```
[14, 16, 12, 9]
```

Par exemple, `L[1]` vaut désormais 16 car dans la liste initiale, la somme de tous les éléments sauf celui d'indice `1` vaut `3+5+8=16`.

Vous créérez une fonction `somme_autre(L)` qui modifiera L et qui, en principe, ne renverra rien.

Vous testerez votre code sur une liste L très grande ayant par exemple un million d'éléments, comme la liste L générée par le code ci-dessous :

```
from random import randrange
```

```
N=10**6
L=[randrange(10) for _ in range(N)]

# Vous pouvez désactiver l'affichage
print(L)
```

Le résultat, si son calcul est bien codé, devrait s'afficher instantanément. Sinon, vous risquez de freezer votre machine, donc allez-y doucement et changez `N` par une valeur plus raisonnable.

Par ailleurs, si vous n'arrivez pas à coder le fait de **changer** les éléments de `L`, vous pouvez envisager de créer une **nouvelle** liste.

Cet exercice est la reprise de l'exercice sum of other elements sur le site Hackerrank.

Arrondi au demi point

On vous donne une note x entre 0 et 20 sous forme d'un nombre flottant et vous devez construire l'arrondi de cette note au demi point près. Voici d'abord quelques exemples d'arrondi au demi point près le plus proche :

```
12.66     -->     12.75
12.9     -->     13
18.11     -->     18.0
8.5     -->     8.5
9.88     -->     10
13.25     -->     13.25
19.80     -->     19.75
13     -->     13
```

Les arrondis de notes sont des nombres de la forme suivante : N ou encore $N + 0.5$ où N est un entier.

Si un note est situé exactement à égale distance de deux arrondis possibles au demi-point, on arrondit à la note supérieure. Par exemple, la note 12.25 qui est à égale distance de 12 et 12.50 est arrondie à la note 12.5.

Pour obtenir la partie entière d'une note (par exemple pour 12.66 c'est 12), on utilisera la fonction `int` :

```
note = 12.66
N = int(note)
print(N)
```

qui affiche 12.

Ecart minimal d'indices (tableau auxiliaire)

On donne une liste `L` d'entiers, par exemple

```
L = [16, 14, 17, 13, 15, 14, 16, 15, 13, 14, 16, 15, 13, 16]
```

Les entiers de `L` seront compris entre 1 et 1000000 inclus. On demande de déterminer l'écart minimal d'indices où se trouveraient des valeurs identiques. Cette valeur est 3 dans la liste ci-dessus ; en effet, 15 apparaît aux indices 4 et 7 et tout autre répétition de valeur, comme 13, apparaît dans la liste avec un écart d'indices valant au moins 3. On notera que l'écart minimal est aussi atteint pour la valeur 16 aux indices 10 et 13.

Si la liste ne contient que des valeurs différentes, le nombre minimal recherché sera 0.

On utilisera l'algorithme suivant, qui vous donne juste la piste à suivre :

— on construira une liste `J`, initialement remplie d'entiers valant tous par exemple -1 (ou si vous préférez, `None`) ayant `1000000 + 1` éléments.

— on parcourra par indices `i` la liste `L` et on enregistrera de manière appropriée l'indice courant `i` dans la liste `J` en fonction de la valeur de `L[i]`

Votre algorithme doit pouvoir traiter n'importe quelle liste `L`, même ayant des centaines de millions d'éléments. On écrira une fonction `ecart_mini(L)` qui renverra la valeur de l'écart minimal.

Cet exercice est inspiré de l'exercice *Distance minimale* de la demi-finale du concours Algoréa 2017.

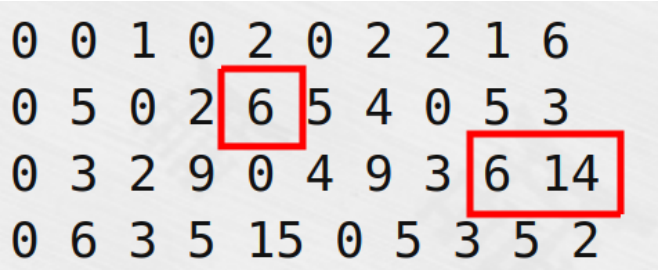
La suite de Van Eck

La séquence de Van Eck est une séquence infinie de nombres entiers positifs dont voici les 100 premiers :

```
0  0  1  0  2  0  2  2  1  6
0  5  0  2  6  5  4  0  5  3
0  3  2  9  0  4  9  3  6  14
0  6  3  5  15 0  5  3  5  2
17 0  6  11 0  3  8  0  3  3
1  42 0  5  15 20 0  4  32 0
3  11 18 0  4  7  0  3  7  3
2  31 0  6  31 3  6  3  2  8
33 0  9  56 0  3  8  7  19 0
5  37 0  3  8  8  1  46 0  6
```

(il y a 10 nombres par ligne).

Chaque élément s'obtient **à partir des précédents** par un procédé que l'on va décrire sur un exemple. Ainsi, comment s'obtient le 30e terme (14) ? Réponse, cf. dessin et explication ci-dessous :



— on regarde le terme courant, le 29e terme, donc 6,

— on cherche de combien d'étapes il faut reculer dans la séquence depuis ce nombre 6 pour retrouver 6 dans la liste. Ici, si on compte, on voit qu'il faut 14 étapes (ce qui localise le 6 situé au milieu de la 2e ligne), d'où le terme suivant de la suite : 14.

Si un élément n'est apparu qu'une seule fois, le nombre qui le suit est, par convention, 0. Par exemple, dans la suite ci-dessus, le 9 en 24e position est suivi de 0 puisqu'aucun 9 n'apparaît avant la 24e position.

La séquence commence par zéro et elle se construit, élément par élément, en suivant la règle illustrée ci-dessus.

Ecrire une fonction `vaneck(n)` qui renvoie la liste des `n` premiers éléments de la séquence.

Suite de Mazet-Saias

La suite de Mazet-Saias est une suite (infinie) d'entiers strictement positifs telle que chaque élément de la suite est diviseur ou multiple de l'élément suivant. Voici les 18 premiers termes de la suite :

```
1, 2, 6, 3, 12, 4, 20, 5, 35, 7, 56, 8, 72, 9, 90, 10, 110, 11
```

En outre, cette suite a la propriété que chaque entier strictement positif est présent une fois et une seule.

L'idée de sa construction est de placer, un par un, au plus tôt et s'ils ne sont pas déjà présents, les entiers consécutifs à partir de 1. Si on est à l'étape où tous les entiers jusqu'à n sont dans la liste, alors soit k l'indice de n dans la liste ; on cherche le plus petit entier non présent, disons N, on le place à l'indice $k+2$ et on place à l'indice $k+1$ l'entier $N \times d$.

Par exemple, la liste à l'étape où tous les entiers entre 1 et 11 sont présents est :

```
1, 2, 6, 3, 12, 4, 20, 5, 35, 7, 56, 8, 72, 9, 90, 10, 110, 11
```

On cherche le plus petit entier à partir de 11 qui n'est pas placé. On voit que c'est 13. On le place deux positions après 11 et, entre 11 et 13, on place le produit $11 \times 13 = 143$, ce qui donne la liste mise à jour suivante :

```
1, 2, 6, 3, 12, 4, 20, 5, 35, 7, 56, 8, 72, 9, 90, 10, 110, 11, 143, 13
```

Etant donné un entier strictement positif n, écrire un programme qui génère les éléments de la suite contenant tous les entiers entre 1 et n. Par exemple si $n = 12$ la liste à trouver est :

```
1, 2, 6, 3, 12, 4, 20, 5, 35, 7, 56, 8, 72, 9, 90, 10, 110, 11
```

Suite découverte ici et répertoriée sur OIES.

Numéro de jour de l'année

Dans cet exercice, l'année est supposée être non bissextile. Par ailleurs, on aura besoin de la liste ci-dessous

```
JOURS_MOIS = [None, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

et qui est telle que `JOURS_MOIS[i]` est le nombre de jours du `i`-ème mois de l'année, pour tout numéro de mois entre 1 et 12. Par exemple, on lit que `JOURS_MOIS[5]` vaut 31 car le mois de mai contient 31 jours.

Toute date de l'année sera codée sous forme d'une **liste** `[j, m]` de deux entiers (`j` pour le numéro du jour dans le mois et `m` pour le numéro du mois). Par exemple, le 14 juillet est codé sous la forme `[14, 7]`.

A titre d'illustration des deux questions suivantes, voici un tableau donnant la date d'un jour dont on connaît le rang (le numéro d'ordre) dans l'année :

Date [j, m]	Numéro dans l'année
[1, 1]	1
[31, 1]	31
[11, 2]	42
[1, 3]	60
[22, 3]	81
[10, 4]	100
[19, 7]	200
[1, 9]	244
[27, 10]	300
[30, 11]	334
[31, 12]	365

(1) Ecrire une fonction `dateVersNumero(date)` qui à partir d'une date, qui sera transmise en argument sous forme de liste, renvoie le numéro de ce jour dans l'année. Par exemple, `dateVersNumero([22,3])` vaut 81 : en effet, il y a 31 jours en janvier, 28 jours en février et 22 à comptabiliser au mois de mars ce qui fait 31+28+22=81.

(2) Ecrire une fonction `numeroVersDate(nro)` qui à partir du numéro d'un jour de l'année renvoie la date de ce jour. Par exemple, `numeroVersDate(81)` est la liste `[22, 3]`. En effet, $31 + 28 \leq 81 < 31 + 28 + 31$ donc il y a au moins les mois de janvier et février complet, mais pas assez pour inclure **tout** le mois de mars, donc $m = 3$. Le reliquat de jours est de $81 - (31 + 28) = 22$ donc $j = 22$.
[Source : examen Python de Champollion/ISM Dakar]

Le problème des déménageurs

On dispose de n objets dont les poids sont connus et qu'on doit ranger dans des boites identiques. Chaque boîte supporte un poids maximal p. On cherche à placer les n objets en minimisant le nombre de boîtes utilisées.

Pour cela, même si ce n'est pas forcément la meilleure solution, on procède de la manière suivante :

- on place les objets côte à côte
- on range le premier objet dans une boîte, disons A
- on range le 2e objet dans la boîte A si cela respecte la contrainte de poids maximal, sinon, on range l'objet dans une nouvelle boîte B
- pour l'objet suivant, on recommence : on place l'objet dans la première des boîtes A ou B qui le permet, et si aucune ne le permet, on le range dans une nouvelle boîte C
- on continue comme ça jusqu'à ce que tous les objets soient rangés.

Par exemple, si la liste des poids de 10 objets est :

```
L = [3, 6, 6, 8, 6, 3, 10, 9, 10, 8]
```

que le poids maximal supporté par chaque boîte est de $p = 20$ alors, la méthode ci-dessus nécessitera 4 boites :

```
Boîte 1 : 3, 6, 6, 3
Boîte 2 : 8, 6
```

```
Boîte 3 : 10, 9
Boîte 4 : 10, 8
```

Explications :

— Les 3 premiers objets tiennent dans la première boîte (poids = 15)
— le 4e objet, de poids 8, ne peut être placé dans la 1re boîte (car $15 + 8 > 20$)
— on le place donc dans la 2e boîte
— on peut y placer le 5e objet, de poids 6 puisque $8 + 6 = 14 \leq 20$
— le 6e objet peut être placé dans la 1re boîte
— le 7e, de poids 10, ne tient dans aucune des boîtes déjà utilisées, donc il faut une nouvelle boîte
— et ainsi de suite jusqu'au 10e objet.

Ecrire une fonction `ranger(L, p)` qui applique la méthode ci-dessus. Le fonction renverra une liste de listes de poids. Dans l'exemple ci-dessus, la fonction doit renvoyer :

```
[[3, 6, 6, 3], [8, 6], [10, 9], [10, 8]]
```

Le problème proposé est connu sous le nom de *bin-packing* et l'algorithme est dit *first-fit*.

Somme de deux éléments d'une liste croissante

On donne une liste croissante `L` d'entiers, par exemple

```
L = [10, 10, 20, 37, 37, 41, 43, 45, 46, 48, 56, 57, 65, 69, 77]
```

et on donne un entier `S`, par exemple $S = 80$ et on demande de dire s'il existe deux entiers `a` et `b` de la liste, placés à des indices distincts, et tels que $a + b = S$.

Dans le cas de la liste `L` ci-dessus :

— si $S = 80$, le problème admet pour solution $a = 37$ et $b = 43$,
— si $S = 81$, le problème n'admet aucune solution,
— si $S = 74$, le problème admet pour solution $a = 37$ et $b = 37$ (il y a deux occurrences de 37 dans la liste),
— si $S = 130$, le problème n'admet aucune solution bien que $65 + 65 = 130$ car 65 n'est présent qu'une seule fois dans la liste.

La liste `L` pouvant être très grande (par exemple un million d'entiers), on ne testera pas toutes les paires possibles d'entiers de `L` car l'exécution serait très longue.

On utilisera la croissance de la liste. On illustre sur le cas de $S = 80$ l'algorithme à appliquer :

— on essaye le premier élément de la liste, $a = 10$; on cherche donc dans la liste un élément $b = 80 - a = 70$
  — on essaye alors à partir du dernier élément de la liste, ici 77
  — cet élément est trop grand, donc, comme la liste est croissante, on essaye le voisin de 70 dans la liste, ici 69 qui lui est trop petit.
— c'est donc que $a = 10$ est exclu et on essaye le voisin à droite de `a` dans la liste.
— ce voisin vaut encore 10 donc on essaye le voisin encore, $a = 20$ et on cherche donc dans la liste un élément $b = 80 - a = 60$

- l'élément cherché est forcément inférieur au dernier élément essayé dans le cas précédent, c'est-à-dire 69
- or, 69 ne convient pas (on cherche 60), ni son voisin de gauche 65 ni son voisin de gauche encore, 57
- comme $57 < 60$ c'est que $a = 20$ n'est pas solution

- on continue de la même façon mais avec le a suivant dans la liste, ici, $a = 37$.

On pourra admettre et utiliser, si besoin, que le problème a, au plus, une solution.

On écrira une fonction `somme(L, s)` qui renverra les deux nombres cherchés sous la forme d'une liste $[a, b]$ si la somme est réalisable ou, une liste vide sinon.

Cet exercice est inspiré de l'exercice Somme à 42000000 de la demi-finale du concours Algoréa 2018.

Remplir une grille en zig-zag

Vous devez remplir un tableau par des entiers consécutifs en suivant un certain motif. Par exemple, si le tableau est de taille 4×6 et si le premier entier à placer est 10 alors le tableau doit être rempli de la manière suivante :

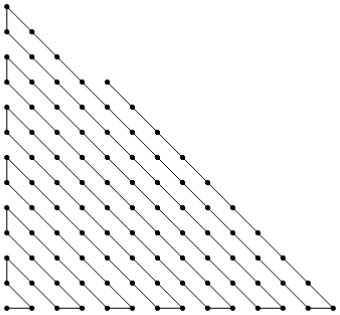
```
10 11 15 16 23 24
12 14 17 22 25 30
13 18 21 26 29 31
19 20 27 28 32 33
```

Plus précisément, écrire une fonction `remplirEnZigzag(n, p, z)` qui remplit un tableau de dimension `n` $\times$ `p` avec tous entiers consécutifs à partir de `z` en appliquant le motif suivant :

- les remplissages se font suivant les diagonales qui montent de la gauche vers la droite du tableau ou qui descendent de la droite vers la gauche ;
- quand une diagonale touche le bord du tableau, le mouvement se poursuit en reprenant à la case voisine au bord.

Déplacement en zig-zag

Ecrire un programme qui dessine le déplacement en zig-zag d'un objet mobile :



Le mobile démarre en bas à gauche et doit marquer au total n bulles noires. Dans le dessin ci-dessus, `n=100` et l'écart vertical ou horizontal entre deux bulles noires voisines est constant. On écrira une fonction `zigzag(n)`.

Conseil : on fera en sorte qu'un couple `(x,y)` enregistre la position courante du mobile et après chaque « déplacement », le couple sera mis à jour. Quasiment tous les tracés de déplacement seront placés dans une boucle `for i in range(n)`.

Détection de cycle

Soit `n` un entier strictement positif et soit une liste `L` de `n` entiers parmi $0, 1, \ldots, n-1$, pas forcément distincts, par exemple, si $n = 9$, cela pourrait être la liste :

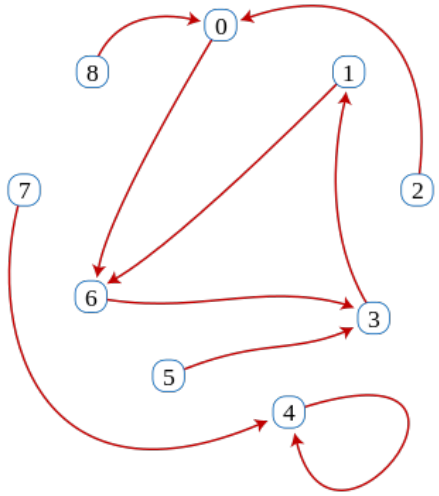
```
L = [6, 6, 0, 1, 4, 3, 3, 4, 0]
```

On peut traduire cette liste par le tableau suivant :

0	1	2	3	4	5	6	7	8
6	6	0	1	4	3	3	4	0

Ensuite, on va parcourir depuis un indice de départ `d` une suite d'indices jusqu'à ce qu'on rencontre un indice déjà rencontré, ce qui fera un cycle :

x	$f(x)$
0	6
1	6
2	0
3	1
4	4
5	3
6	3
7	4
8	0



Par exemple,

- on part de l'indice `d` = `i0` = 2,
- on lit la valeur dans `L` à l'indice `i0`, c'est `i1` = 0
- l'indice `i1` = 0 n'a pas encore été rencontré donc on s'y rend
- on recommence : on lit la valeur dans `L` à l'indice `i1` = 0, c'est `i2` = 6
- l'indice `i2` = 6 n'a pas encore été rencontré donc on s'y rend
- on recommence : on lit la valeur dans `L` à l'indice `i2` = 6, c'est `i3` = 3 qui n'a pas été rencontré
- on a `L[i3]` = 1 non encore rencontré et on pose `i4` = `L[1]` = 6
- Mais `i4` = 6 a été rencontré (`i2` = 6) donc on s'y rend et comme on a trouvé un cycle, on arrête là le parcours.

Dans cet exemple, on a trouvé un cycle de longueur 3 traduit par : 6 → 3 → 1 → 6.

D'une manière générale, on demande d'écrire une fonction `cycle(L, d)` qui renvoie la longueur du cycle détecté quand l'indice de départ est l'indice `d` (on peut démontrer facilement qu'il y a forcément un cycle). Cette longueur est le nombre de « sauts » qu'il faut accomplir avant de revenir à un indice déjà rencontré. Ce problème peut être résolu de manière efficace par l'algorithme de Floyd de détection de cycle, non demandé ici.

Dans le cas de la liste en exemple ci-dessus, l'appel `cycle(L, 2)` doit valoir 3. Si l'indice de départ `d` vérifie `L[d]=d` (avec l'exemple ci-dessus, $d = 4$) la longueur à renvoyer sera 1.

Le tri par sélection

Le *tri par sélection* est un algorithme de tri, utilisé par exemple pour trier une liste de nombres dans l'ordre croissant. Son principe est le suivant : on dispose d'une liste `t`, on en cherche le plus grand élément `m`, on le met « de côté », on recommence avec la liste restante en recherchant à nouveau son plus grand élément, puis on le place à gauche de `m`, et ainsi de suite jusqu'à ce que la liste initiale soit vide. La liste triée est alors la liste construite avec les maxima successifs.

Quand on programme un tri par sélection, pour éviter de multiples suppressions/recopies, au lieu de mettre « de côté » les maxima trouvés, on les déplace à l'intérieur même de la liste `t`. Plus précisément, soit une liste `t` de `n` entiers. Le tri par sélection consiste à :

- chercher le plus grand élément de `t`,
- l'échanger avec l'élément le plus à droite de `t`
- recommencer avec la liste formée des $n - 1$ premiers éléments de `t`.

(1) Ecrire une fonction `maxi(t, p)` qui prend en paramètre une liste d'entiers `t` de longueur `n`, et un indice `p` valide de `t` (donc tel que $0 \leq p \leq n - 1$) et qui renvoie l'**indice** d'un élément maximum de la sous-liste de `t` qui s'étend de l'indice `0` (inclus) à l'indice `p` (inclus).

(2) (a) Ecrire une procédure `echange(t, i, j)` qui prend en paramètre une liste d'entiers `t` et deux indices `i` et `j` d'éléments de `t` et qui échange les contenus dans `t` aux indices `i` et `j`.

(b) Écrire le code d'un tri par sélection avec une procédure `selection(t)` et qui utilise `maxi` et `echange`.

(c) Générer une liste `L` de 30000 entiers et lui appliquer la fonction `selection`. Que pensez-vous de la vitesse d'exécution ?

Vendredi 13

On se propose d'écrire un code qui recherche les 10 prochains vendredis 13.

On codera

- chaque mois par un entier entre 1 et 12,
- chaque jour par un entier entre 1 et 31,
- chaque jour de la semaine par un code (lundi = 1, mardi = 2, ..., dimanche=7).

(1) Ecrire une fonction `lendemain(js, j, m, a)` qui prend une date en paramètres et renvoie une liste définissant la date du lendemain. Par exemple `lendemain(2, 8, 12, 2015)` demande la date du lendemain du mardi 8 décembre 2015 et doit renvoyer la liste `[3, 9, 12, 2015]` car le lendemain est le mercredi 9 décembre 2015.
Les paramètres ont les significations suivantes :

- le jour de la semaine `js` (un entier entre 1 et 7)

— le numéro `j` du jour dans le mois (un entier entre 1 et 31)
— le numéro `m` du mois (un entier entre 1 et 12)
— l'année `a`.

La fonction traitera successivement les cas suivants :

— le 31 décembre,
— la fin des mois autres que décembre et ayant 30 ou 31 jours,
— la fin du mois de février,
— tous les autres jours.

On pourra utiliser les listes suivantes :

— `mois30 = [4, 6, 9, 11]`
— `mois31 = [1, 3, 5, 7, 8, 10]`

sachant qu'on peut tester l'appartenance d'un objet `x` dans une liste `L` par :

```
x in L
```

On rappelle qu'une année `a` est bissextile si le booléen

```
a % 400 == 0 or (a % 4 == 0 and a % 100 != 0)
```

est `True`.
Pour trouver le jour de la semaine du lendemain, on pourra utiliser un reste de division par 7 mais ce n'est pas indispensable, on peut toujours faire une instruction `if`.
On effectuera les tests suivants :

```
(2, 8, 12, 2037)  -> [3, 9, 12, 2037]
(5, 1, 1, 2038)   -> [6, 2, 1, 2038]
(4, 31, 12, 2037) -> [5, 1, 1, 2038]
(7, 31, 1, 2038)  -> [1, 1, 2, 2038]
(6, 30, 1, 2038)  -> [7, 31, 1, 2038]
(1, 29, 2, 2038)  -> [2, 1, 3, 2038]
(7, 28, 2, 2038)  -> [1, 1, 3, 2038]
(6, 28, 2, 2037)  -> [7, 1, 3, 2037]
(6, 16, 7, 2038)  -> [7, 17, 7, 2038]
```

(2) Déterminer les 10 prochains vendredis 13 à partir d'aujourd'hui. Vérifiez avec un calendrier.

IA pour un Mastermind

Les règles de bases du jeu Mastermind sont supposées être à peu près connues. Dans la suite, les 6 couleurs sont numérotées de 0 à 5. Le joueur doit découvrir 4 pions colorés. Les répétitions de couleurs sont autorisées.

Un pion de la combinaison proposée par le joueur est dit *bien placé* si à la même position, le pion de la combinaison à découvrir est de même couleur.

L'évaluation par le joueur qui connait la combinaison cachée de chaque proposition faite par le joueur se fait avec des fiches noires ou blanches de la manière suivante :

— le nombre `a` de fiches noires est le nombre de pions biens placés ; plus précisément, `a` est le nombres de positions bien placées c'est-à-dire les positions parmi les 4 possibles où le pion de la combinaison inconnue et le pion de la combinaison proposée ont même couleur ;

— le nombre b de fiches blanches est le nombre de pions *présents* mais *mal placés*; plus précisément, cela veut dire qu'on peut mettre en correspondance b positions $p_1, \cdots, p_b$ dans la combinaison inconnue et b positions $q_1, \cdots, q_b$ mal placées dans la combinaison proposée en sorte que les pions à ces positions aient même couleur autrement dit que p_1 et q_1 aient même couleur, p_2 et q_2 aient même couleur, et ainsi de suite.

Par exemple si on a la combinaison inconnue (*guess*) et la proposition (*code*) ci-dessous :

```
2 3 3 5 ← code
2 5 4 3 ← guess
```

l'évaluation de la proposition est : 1 noir et 2 blancs. Explication :

— le noir à cause des couleurs 2 en 1re colonne,

— le premier blanc à cause de la couleur 3

  — en 2e colonne dans le code

  — en 4e colonne dans guess,

— le deuxième blanc à cause de la couleur 5

  — en 4e colonne dans le code

  — en 2e colonne dans guess.

En clair, pour connaître le nombre de blancs dans l'évaluation, on retire, dans la proposition et le code à trouver, chaque jeton bien placé puis on regarde le nombre de couleurs commune entre la proposition et le code inconnu.

Toute combinaison de pions s'écrira en Python comme une liste de 4 entiers.

(1) Ecrire une fonction `get_score(code, guess)` qui renvoie une liste `[a, b]` (le score) où `a` est le nombre de fiches noires et `b` le nombre de fiches blanches.

(2) Construire une liste `guesses` de toutes les combinaisons possibles (écrire 4 boucles for imbriquées). `guesses` est une liste de listes d'entiers entre 0 et 5. On touvera 1296 combinaisons possibles.

(3) Le joueur applique la statégie suivante : on suppose qu'il dispose d'une liste `L` de combinaisons parmi lesquelles se trouve la solution. Il en choisit une au hasard (on l'appelle `my_guess`), il la propose et elle est évaluée en un score `[a, b]`. Le joueur calcule alors l'évaluation de toutes les combinaisons de la liste `L` et ne retient que celles qui sont évaluées en `[a, b]`. Ces combinaisons forment une nouvelle liste `M` moins longue que `L` et où se trouve la solution. Ecrire une fonction `reduire(L, my_guess, score)` et qui renvoie `M`.

(4) Simuler 1000 parties et examiner en combien de coups, en moyenne, le joueur parvient à deviner la solution et le nombre maximal de coups.

Alignement au Puissance 4

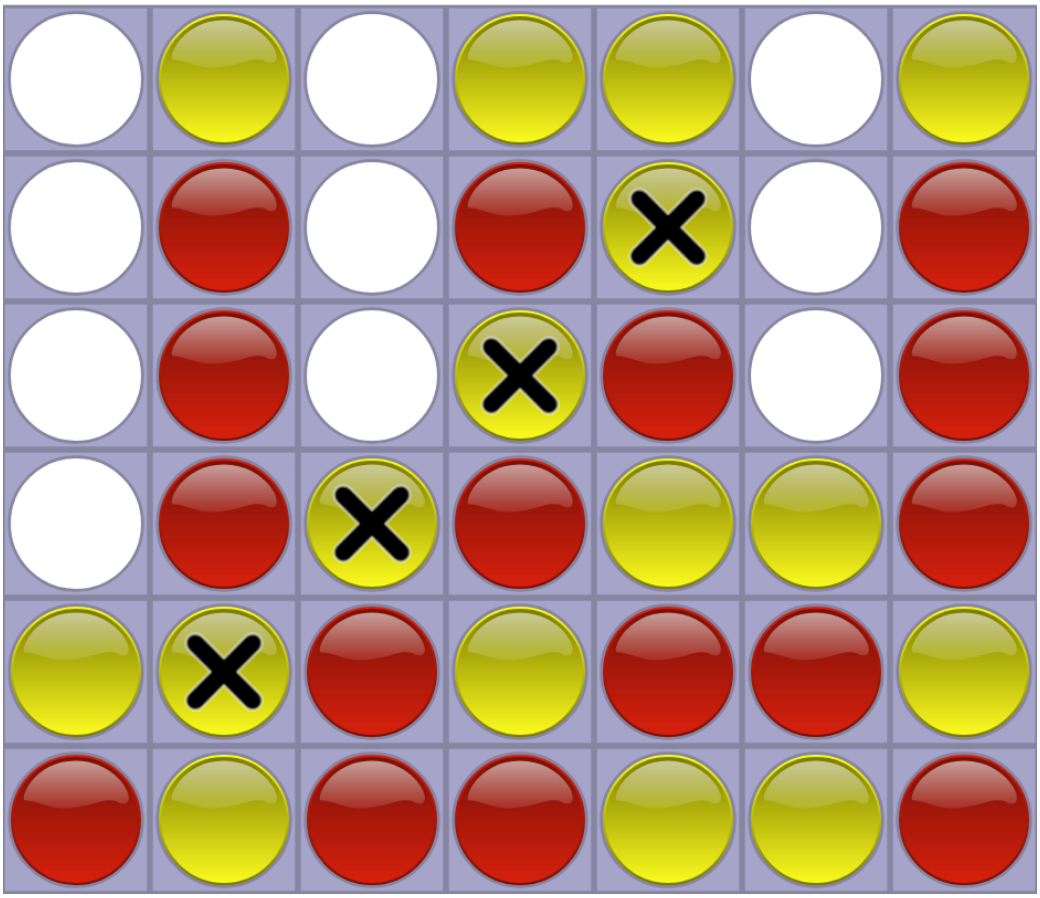
On donne le plateau d'une partie du jeu Puissance 4. La partie peut aussi bien être en cours que terminée. Le plateau est fourni par ses lignes horizontales, sous la forme d'une liste `L` de 6 listes de 7 caractères parmi les trois caractères suivants :

— le caractère `R` pour une case occupée par un jeton rouge,

— le caractère `J` pour une case occupée par un jeton jaune,

— le caractère `.` (un point) pour une case vide.

Attention, la première liste de `L` représente la ligne la plus basse du plateau, et ainsi de suite. Par exemple, le plateau de la partie (terminée) suivante :



est représenté par la liste `L` Python suivante :

```
L= [
['R', 'J', 'R', 'R', 'J', 'J', 'R'],
['J', 'J', 'R', 'J', 'R', 'R', 'J'],
['.', 'R', 'J', 'R', 'J', 'J', 'R'],
['.', 'R', '.', 'J', 'R', '.', 'R'],
['.', 'R', '.', 'R', 'J', '.', 'R'],
['.', 'J', '.', 'J', 'J', '.', 'J']
]
```

(l'image du plateau provient d'une application en ligne qui propose une IA imbattable).

(1) Ecrire une fonction `afficher(L)` qui affiche en mode texte le plateau tel que les joueurs pourraient le voir, c'est-à-dire que si une ligne est en dessous d'une autre, cela doit appaître dans la sortie affichée. Chaque case du plateau sera représentée par le caractère `.`, `J` ou `R`. On placera un espace entre deux colonnes consécutives. Par exemple, le plateau de l'image ci-dessus devra être affiché comme suit :

```
. J . J J . J
. R . R J . R
. R . J R . R
. R J R J J R
J J R J R R J
R J R R J J R
```

(2) Ecrire une fonction `connect(seq)` qui partant d'une liste `seq` formée d'un nombre indéterminé de caractères parmi `J`, `R` et `.` renvoie, si cela se produit, le premier caractère parmi `J` ou `R` qui apparaît consécutivement 4 fois de suite ou qui, sinon, renvoie le caractère `.` (le point).
Voici quelques exemples de comportements :

```
..JRRRR.J → R
JRRRR.JJJ → R
JJ.RRRR → R
RRRJ → .
JRRR → .
..... → .
..JJ.RRR.R → .
JRJR → .
```

(3) Etant donné un plateau donné sous forme de liste `L` comme ci-dessus, écrire une fonction `seq(L)` qui renvoie une liste formée des séquences suivantes du plateau :

- toutes les lignes,
- toutes les colonnes
- toutes les diagonales de longueur au moins 4.

Chaque ligne, colonne ou diagonale sera représentée par une liste de caractères parmi `J`, `R` ou `.` (un point).
Concernant les diagonales (vues de la gauche vers la droite), on pourra remarquer que

- si on fixe une diagonale montante, alors pour chaque position `(i, j)` de cette diagonale, la valeur de `j - i` est une constante que l'on déterminera,
- si on fixe une diagonale descendante, alors pour chaque position `(i, j)` de cette diagonale, la valeur de `j + i` est une constante que l'on déterminera.

(4) Ecrire une fonction `victoire(L)` qui renvoie le caractère représentant le joueur qui a éventuellement gagné la partie ou, sinon, la chaîne `"."` (un point). On utilisera les fonctions construites aux deux questions précédentes.
Par exemple, si `L` est le plateau donné en début d'énoncé, la fonction doit renvoyer `"J"`.
Si `L` est le plateau ci-dessous :

```
L=[
['R', 'J', 'R', 'R', 'J', 'J', 'R'],
['J', 'J', 'R', 'J', 'R', 'R', 'J'],
['.', 'R', '.', 'R', 'J', 'J', 'R'],
['.', 'R', '.', 'J', 'R', '.', 'R'],
['.', 'R', '.', 'R', 'J', '.', 'R'],
['.', 'J', '.', 'J', 'J', '.', 'J']]
```

le caractère `.` doit être renvoyé par fonction `victoire`.

Formule de Luhn

1. Écrire une fonction qui calcule et retourne sous forme d'entier la somme des valeurs d'une liste L passée en paramètre.
2. Écrire une fonction `reduire(n)` qui étant donné un entier n entre 0 et 18 renvoie le nombre réduit à 1 chiffre, obtenu en effectuant la somme des chiffres de n. Voici quelques exemples de comportement de la fonction :

```
0 -> 0
7 -> 7
10 -> 1 (=1+0)
18 -> 9 (=1+8)
```

3. Écrire une fonction `to_liste(n)` qui prend en paramètre un nombre entier n composé de 15 chiffres et qui retourne la liste inversée des chiffres qui le composent. On utilisera des divisions successives par 10. Tester avec le nombre suivant :

```
497010123456789 -> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 0, 7, 9, 4]
```

4. Écrire une fonction `double(L)` qui modifie une liste L passée en paramètre en multipliant par deux et en réduisant (cf. fonction de la question 1) les valeurs aux indices pairs de `L`, sachant que les indices commencent à 0. Tester avec la liste `L` suivante :

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0] --> [9, 8, 5, 6, 1, 4, 6, 2, 2, 0]
```

5. L'algorithme de Luhn est un procédé de contrôle simple utilisé en cryptographie pour valider des numéros de sécurité sociale, de compte, de carte bancaire, etc. Le principe est celui d'une somme de contrôle : il consiste à calculer un code de contrôle à partir des chiffres qui composent le numéro. Ce code est ensuite collé au numéro afin de permettre de tester facilement sa validité.
   Par exemple, dans le numéro de carte de crédit suivant :

```
3056 4931 0876 7124
```

   le `4` final est en fait un code de contrôle, obtenu à partir du numéro de carte original (`305649310876712`).
   L'algorithme de vérification est le suivant, avec comme illustration le code `3056 4931 0876 7124`

   — On sépare le numéro de la carte (`305649310876712`) du code de vérification (`4`).
   — On transforme le numéro de la carte en liste inversée des chiffres qui le composent : `[2, 1, 7, 6, 7, 8, 0, 1, 3, 9, 4, 6, 5, 0, 3]`
   — On multiplie par deux la valeur de chaque élément de la liste positionné à un indice pair (les indices commencent à 0), en prenant soin d'additionner les chiffres des valeurs supérieures à 9 afin de conserver des nombres entre 0 et 9 : `[4, 1, 5, 6, 5, 8, 0, 1, 6, 9, 8, 6, 1,`
   — On fait la somme de tous les chiffres de la liste : 4 + 1 + 5 + 6 + 5 + 8 + 0 + 1 + 6 + 9 + 8 + 6 + 1 + 0 + 6 = 66
   — On divise cette somme par 10 et on conserve le reste : 6

   Si le reste (`6`) + le code de vérification (`4`) = 10 alors le numéro est valide, sinon il s'agit probablement d'un faux numéro.
   Écrire une fonction `valider` qui prend un code de carte bancaire en paramètre (nombre entier de 16 chiffres) et qui met en oeuvre l'algorithme décrit ci-dessus. La fonction écrit

le message `"Le code est valide"` si le code est valide, ou `"Le code est invalide"` sinon.
Tester avec les codes suivants, puis avec votre propre code de carte bleue :

```
5133698711648808 -> True
3056493108767125 -> False
```

Selon D. Knuth, on doit aussi à Luhn l'invention en 1953 des tables de hachage (par chaînage), une des structures de données les plus fondamentales en informatique.

Chapitre VI

Chaînes de caractères

Les chaînes : premier contact

La notion de chaîne

Les chaînes de caractères permettent de stocker du **texte**. En première approximation, une chaîne est une succession de caractères imprimables. Cela peut être, par exemple, une phrase ou un groupe de mots, comme *Le nombre 42 est magique.*

On dit souvent *chaîne* au lieu de *chaîne de caractères.*

De la même façon qu'un programme peut travailler avec des entiers, il peut travailler avec des chaînes.

Sensibilité à la casse

La *casse* d'un caractère alphabétique est le fait que le caractère soit en majuscule ou en minuscule.

Les chaînes *bonjour* et *bonJour* sont considérées en Python comme distinctes. Si deux chaînes ont, à des positions identiques, des lettres de casses différentes, les chaînes sont considérées comme distinctes.

Chaîne littérale

Le moyen le plus immédiat d'utiliser des chaînes dans un code Python est par le biais d'une *chaîne littérale* :

```
1 s = "orange"
2 print(len(s))
3
4 t = 'rose'
5 print(len(t))
```

```
6 6
7 4
```

À la ligne 1, on lit `"orange"` : c'est une chaîne, dite *littérale.* A la ligne 2, on demande l'affichage du nombre de caractères de la chaîne. On lit à la ligne 6 la longueur de la chaîne.

Une chaîne est *littérale* lorsque

— tous ses caractères sont placés dans le **code-source**,

— la chaîne est encadrée par des caractères particuliers, appelés *quotes*, ici des guillemets anglo-

saxons (mais il existe d'autres délimiteurs possibles).

Afficher une chaîne

Pour afficher une chaîne avec une « commande » Python, on utilisera obligatoirement la fonction `print` :

```
1 fruit = "orange"
2 print(fruit)
3 print("bonjour")
```

```
4 orange
5 bonjour
```

— Lignes 2 et 4 : ici, on affiche une chaîne par l'intermédiaire d'une variable.
— Lignes 3 et 5 : affichage direct d'une chaîne littérale.
— On notera que les délimiteurs de chaîne ne sont pas affichés.

Dans certains environnement (mode interactif, cellule Jupyter Notebook), une chaîne peut être affichée sans recours à la fonction `print`.

Erreur courante de débutant

Pour afficher une chaîne littérale, on fera attention de ne pas oublier les quotes autour de la chaîne :

```
print(bonjour)
```

```
NameError: name 'bonjour' is not defined
```

La notion de caractère

L'unité fondamentale d'un texte est le caractère. Un mot, une phrase est composée de caractères. On peut disposer de certains caractères en les plaçant entre guillemets, par exemple la lettre `"Y"` qui est en fait une chaîne d'un seul caractère (alors qu'on en écrit trois dans le code si on compte les deux guillemets).

Parmi les caractères, on trouve les caractères alphabétiques majuscules ou minuscules mais aussi tous les caractères de ponctuation, les chiffres, par exemple le chiffre `"8"`.

On dispose aussi de caractères invisibles comme l'espace `" "` (taper sur la barre d'espace) ou encore le saut de ligne. Le **blanc souligné** _ est aussi un caractère très utilisé. Il existe aussi des caractères non imprimables et qui peuvent se retrouver dans une chaîne.

Python supporte l'encodage UTF-8 et donne donc accès à tous les caractères imaginables.

En pratique, dans un cours de débutant sur les chaînes de caractères, on essaye de ne pas avoir à utiliser des caractères tels que le guillemet ou l'apostrophe car ils ont rôle spécial pour construire des chaînes dites *littérales*.

La notion de chaîne littérale

Une chaîne littérale est un chaîne de caractères **saisie telle quelle dans le code-source** entre une paire de délimiteurs. Ainsi, `"orange"` est une chaîne littérale, le délimiteur étant ici un guillemet :

```
s = "orange"
print(s)
```

```
orange
```

Les délimiteurs **ne** font **pas** partie du contenu d'une chaîne littérale, comme on le voit à l'affichage.

Insistons : une chaîne littérale n'a de sens que dans un **code-source** puisque *littéral* veut dire qui s'interprète *littéralement* autrement dit, « lettre à lettre », et c'est bien comme cela qu'elle apparaît dans le code-source Python.

Le délimiteur de chaîne littérale le plus familier est le guillemet anglais ". Mais un autre délimiteur usuel est l'apostrophe :

```
1 s = "rose"
2 t = 'rose'
3 print(s)
4 print(t)
```

```
5 rose
6 rose
```

— Ligne 1 : chaîne littérale entourée de guillemets

— Ligne 2 : chaîne littérale entourée d'apostrophes

— Lignes 5-6 : les chaînes `s` et `t` sont égales.

Pour ce qui est du contenu d'une chaîne, par défaut, les caractères d'une chaîne Python (littérale ou pas d'ailleurs) appartiennent au jeu Unicode et donc peuvent comporter des accents ou être des caractères spéciaux. Par exemple, la chaîne littérale `s` suivante est valide :

```
s = "Les pièces de 5€ n'existent pas"
print(s)
```

```
Les pièces de 5€ n'existent pas
```

Opérations sur des chaînes

On peut faire de nombreuses opérations sur les chaînes. Voici juste quelques exemples et qui seront détaillés ailleurs.

Nombre de caractères

Déterminer avec la fonction standard `len` le nombre de caractères d'une chaîne, ce qu'on appelle sa *longueur* :

```
1 print(len("anticonstitutionnellement"))
```

```
2 25
```

— Ligne 1 : la fonction standard `len`, fournie par Python, effectue le calcul de la longueur.

Concaténer deux chaînes

L'opérateur + permet de mettre bout à bout deux chaînes :

```
1 print("bon" + "Jour")
```

```
2 bonJour
```

— Ligne 1 : la *concaténation* (c'est le terme consacré) est effectuée avec l'opérateur +.

— Ligne 2 : noter que les deux chaînes **bon** et **Jour** sont placées bout à bout, sans espace entre les chaînes.

Détail : l'expression `"bon" + "Jour"` est la concaténation de deux chaînes littérales et n'est pas une chaîne littérale.

Chaîne vide

Une chaîne peut être vide autrement dit ne contenir aucun caractère. Une chaîne vide peut être représentée par la chaîne littérale `""` (deux guillemets collés l'un à l'autre et qui n'entourent rien). La chaîne vide est de longueur nulle :

```
vide = ""

print(len(vide))
```

```
0
```

La chaîne vide peut aussi être notée `''` c'est-à-dire deux apostrophes côte-à-côte, sans espace entre les deux apostrophes :

```
vide = ''

# une chaîne vide a une longueur nulle
print(len(vide))
```

```
0
```

La chaîne vide a de nombreux usages (comme le chiffre zéro en a !). Par exemple, une chaîne vide peut être utilisée pour supprimer des caractères à l'intérieur d'une chaîne en remplaçant les caractères par une chaîne vide.

Le caractère *espace*

De même qu'il existe un caractère `A` désignant la lettre `A`, il existe un caractère *espace*, celui obtenu en tapant sur la barre d'espaces. Plus précisément, le contenu de la chaîne `" "` est un espace :

```
1 print("Bug" + "2038")
2 print("Bug" + " " + "2038")
3 print("Bug 2038")
```

```
4 Bug2038
5 Bug 2038
6 Bug 2038
```

— Ligne 1 : Les chaînes `"Bug"` et `"2038"` sont concaténées sans insertion d'espace.

— Lignes 2 et 3 : les chaînes affichées sont identiques : l'espace peut-être considéré comme une chaîne littérale (ligne 2) ou immergé dans la chaîne (ligne 3).

Invisible n'est pas vide

La chaîne `" "` n'est pas vide, elle contient exactement un caractère :

```
espace = " "
print(len(espace))
```

```
1
```

Le caractère *espace* sert aussi à créer une indentation en plaçant des « blancs » en début de lignes. Pour dessiner en mode texte, la figure ci-dessous :

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
*******************
          #
          #
```

il faut placer des caractères espace en début de ligne.

Le caractère *saut de ligne*

Dans beaucoup de langages de programmation, y compris Python, le codage du caractère *saut de ligne* est noté \n. L'usage de ce groupe de **deux** caractères permet effectivement de signifier un saut de ligne dans une chaîne (littérale) :

```
s = "Lundi\nMardi\nMercredi"
print(s)
```

```
Lundi
Mardi
Mercredi
```

On notera que le caractère saut de ligne \n n'a pas de raison d'être d'être suivi d'un espace littéral qui provoquerait un effet peut-être non désiré :

```
1 s = "Lundi\n Mardi\nMercredi"
2 print(s)
```

```
3 Lundi
4  Mardi
5 Mercredi
```

— Ligne 1 : noter l'espace dans le code source, dans la chaîne littérale, juste avant le mot *Mardi*.

— Ligne 4 : l'espace est compté dans la chaîne et visible à l'affichage.

Bien que le caractère littéral représentant un saut de ligne s'écrive avec **deux** caractères, ces deux caractères sont insécables et ils comptent pour un caractère dans la longueur de la chaîne vue comme objet Python. Ainsi :

```
s="rose\nbleu"
print(s)
print(len(s))
```

```
rose
bleu
9
```

Plusieurs sauts de ligne

Pour aérer un texte, on peut souhaiter y placer plusieurs sauts de ligne l'un après l'autre. Si on place côte-à-côte deux caractères \n, on effectue deux passages à la ligne, ce qui va créer une ligne blanche :

```
texte = "rose\n\nkiwi"
print(texte)
```

```
rose

kiwi
```

Une ligne blanche n'est jamais vide et est constituée d'au moins un caractère, le saut de ligne et éventuellement des espaces en début de ligne.

Concaténation et répétition de chaînes

On peut concaténer des chaînes, c'est-à-dire les mettre bout à bout. La concaténation de chaînes est effectuée avec l'opérateur + :

```
1 a = "Rose"
2 b = "Rouge"
3 c = "Rome"
4 s = a + b + c
5 print(s)
6 print(a, b, c)
```

```
7 RoseRougeRome
8 Rose Rouge Rome
```

— Ligne 4 : s est la concaténation des chaînes a, b et c.

— Lignes 5 et 8 : la concaténation des chaînes crée une **nouvelle** chaîne et préserve les chaînes initiales.

On peut aussi concaténer n fois une chaîne avec elle-même avec l'opérateur *

```
1 a = "Rose"
2 r = 3 * a
```

```
3 print(r)
4 print(a * 3)
5 print(2 * a *2)
```

```
6 RoseRoseRose
7 RoseRoseRose
8 RoseRoseRoseRose
```

— Ligne 3 ou 4 : a est répétée 3 fois

— Lignes 5 : on peut multiplier des deux côtés.

On peut aussi panacher addition et multiplication :

```
a = "Rose"
b = "Rouge"
c = "Rome"
s = 2 * (c + a + ' ') + 2 * (c + b + ' ')
print(s)
```

```
RomeRose RomeRose RomeRouge RomeRouge
```

Concaténation et produit par zéro

L'opération `0 * s` renvoie la chaîne vide et `1 * s`. On peut en déduire une astuce pour mettre un nom au pluriel en ajoutant, selon l'effectif, un `"s"` terminal :

```
for val in (0, 1, 7):
    print(val, "bille" + (val > 1)*"s")
```

```
0 bille
1 bille
7 billes
```

Voici 3 autres façons de réaliser la même opération

```
for val in (0, 1, 7):
    print(val, "bille" + ["s", ""][val < 2])

print("---------------")
for val in (0, 1, 7):
    print(val, "bille" + ("" if val < 2 else 's'))

print("---------------")
for val in (0, 1, 7):
    print(val,  f"bille{'s'[val<2:]}")
```

et qui affiche :

```
0 bille
1 bille
7 billes
---------------
0 bille
```

```
1 bille
7 billes
---------------
0 bille
1 bille
7 billes
```

Chaînes égales

L'opérateur == permet de comparer deux chaînes pour savoir si elles ont ou pas exactement les mêmes caractères et dans le même ordre :

```
1 print("ROSE" == "ROSE")
2 print("RoseRose" == "Rose" + "Rose")
3 print("Rose" == "rose")
4 print("ROSE" == 'ROSE')
```

```
5 False
6 True
7 False
8 True
```

— Ligne 1 : Malgré les apparences, les chaînes sont distinctes : à l'indice 1, l'une utilise la lettre `O` majuscule et l'autre un zéro.

— Ligne 2 : `"Rose" + "Rose"` est la concaténation de deux chaînes.

— Ligne 3 : Pour que deux caractères soient égaux, il faut déjà qu'ils aient la même casse, ce qui n'est pas le cas de la lettres initiale.

— Ligne 4 : Des délimiteurs apostrophes ou guillemets n'ont pas d'influence sur l'égalité de chaînes.

Attention que l'égalité de deux chaînes est une opération ayant une complexité cachée. En effet, pour savoir si deux chaînes sont égales ou pas, l'interpréteur Python compare terme à terme les éléments des deux chaînes jusqu'à ce que deux caractères soints distincts ou bien qu'une des chaînes ne possède plus d'élément à examiner.

Donc la comparaison de deux chaînes de longueur chacune 1000, peut nécessiter jusqu'à 1000 comparaisons. Ainsi, l'opérateur == entre deux chaînes peut être coûteux (en réalité, c'est moins simple que ça, cf. implicit Interning).

Recherche de sous-chaînes avec `in`

Soit la chaîne

```
s = "xyxABCyyyxABCxyyxABCxyx"
```

La chaîne `'ABC'` est une sous-chaîne de `s` et elle y apparaît même 3 fois. Rechercher une sous-chaîne dans une chaîne est une opération classique en programmation et appelée *recherche de motif*, en anglais *pattern matching*. Les caractères de la sous-chaîne doivent être contigus et non séparés ; ainsi la chaîne `ABC` n'est pas considérée comme étant une sous-chaine de la chaîne `xxxAxxxxBxxCxxxx`.

Une première possibilité de détection d'une sous-chaîne est d'utiliser l'opérateur `in` qui va renvoyer `True` ou `False` selon la présence ou l'absence de la sous-chaîne :

```
s = "xyxABCyyyxABCxyyxABCxyx"

print("ABC" in s)
print("AbC" in s)
```

```
True
False
```

En première approximation, voici comment se fait la recherche de la sous-chaîne : elle se poursuit jusqu'à obtention d'une occurrence de la sous-chaîne ou alors lorsque la fin de la chaîne est atteinte. Si une occurrence est découverte, la recherche s'interrompt.

Pour entrer dans les détails d'implémentation en CPython, il semble que l'opérateur `in` fasse appel aux algorithmes de Boyer-Moore et Horspool. L'examen du code-source de la fonction `PyUnicode_Contains` montre que des fonctions spécialisées de recherche sont invoquées, par exemple `ucs1lib_find`, qui font appel à la fonction `FASTSEARCH`.

Accès aux caractères d'une chaîne

On peut accéder en lecture individuellement aux caractères d'une chaîne à l'aide d'un indice entier :

```
p = "Jupiter"

# le premier caractère de la chaîne p
print(p[0])

# le troisième caractère de la chaîne p
print(p[2])

# la longueur de la chaîne
n = len(p)
print(n)

# Dernier caractère de la chaîne p
print(p[n - 1])
```

```
J
p
7
r
```

Les caractères d'une chaînes sont numérotés de la gauche vers la droite. Ces numéros sont appelés des *indices* ; la numérotation commence à 0 (et non pas à 1) et se termine à `n-1` où `n` désigne le nombre total de caractères.

On accède à chaque caractère de la liste avec l'opérateur `[]` d'indexation.

Le fait de pouvoir accéder à tous les caractères d'une chaîne par des entiers consécutifs se traduit en disant que les chaînes de caractères sont du type *séquence*.

Il est possible d'utiliser des indices négatifs, ce qui est parfois à l'origine de bugs pour ceux qui ne connaissent pas leur existence.

Dépassement d'indice dans une chaîne

Comme pour les listes, un indice de chaîne trop grand entraîne la levée d'une exception :

```
1 z = "orange"
2 c = z[10]
3
4 print(c)
```

```
5     c = z[10]
6 IndexError: string index out of range
```

— Ligne 6 : une erreur d'indice est signalée
— Ligne 2 : la variable c désignerait le caractère à l'indice 10 alors que l'indice maximal dans c est 5 puisque la chaîne z est de longueur 6.
— Ligne 2 : L'opération z[10] est en fait interdite, on tente d'accéder en lecture à un caractère qui n'existe pas.
— Lignes 5-6 : Le message d'erreur explique que l'indice 10 est en dehors de la plage d'indices possibles.

Tenter d'accéder en simple lecture à un caractère d'une chaîne avec un indice ne correspondant pas à un élément de la chaîne conduit à une erreur, de type `indexError` et qualifiée de *débordement d'indice*. C'est le même problème que pour les listes.

Boucle for : parcours de chaînes

Une chaîne est une séquence et peut être parcourue, sans indice, par une boucle `for`. Voici un exemple de parcours d'une chaîne avec une boucle for :

```
for c in "ALIBABA":
    print(c*2)
```

```
AA
LL
II
BB
AA
BB
AA
```

C'est le même principe que pour les listes, il est possible de faire un parcours sans recourir à un indice.

Il est aussi possible de parcourir la chaîne par indices :

```
s="ALIBABA"

for i in range(len(s)):
```

```
    print(s[i]*2)
```

```
AA
LL
II
BB
AA
BB
AA
```

Cela peut être justifié dans certaines situations (par exemple si on doit comparer un caractère au suivant ou au précédent) où l'usage recommande d'utiliser `enumerate`, voir cette explication.

Créer une chaîne à partir de la chaîne vide

Soit à construire une chaîne `z` dont les caractères sont exactement les consonnes d'une chaîne `s` donnée. Par exemple, si `s` est la chaîne `broccoli` alors la chaîne cherchée est `brccl`.

Pour cela, il suffit de parcourir la chaîne `s` et d'ajouter à une chaîne initialement vide (disons `z`) le caractère courant de `s` si le caractère est une consonne (et donc s'il n'est pas une voyelle). D'où le code

```
1  s= "broccoli"
2  VOYELLES ="aeiouy"
3  z = ""
4
5  for c in s:
6      if c not in VOYELLES:
7          z += c
8
9  print(s)
10 print(z)
```

```
11 broccoli
12 brccl
```

— Lignes 3, 10 et 12 : la chaîne à construire est initialement vide.

— Lignes 5-7 : la chaîne `s` est agrandi au fur et à mesure par ajout d'une consonne.

C'est le même type de construction que pour une liste initialement vide à laquelle on applique la méthode append.

Modifier une chaîne

Les caractères d'une chaîne ne sont pas modifiables :

```
1 s = "taTou"
2 s[2] = "B"
3 print(s)
```

```
4 TypeError: 'str' object does not support item assignment
```

— Ligne 2 : on tente de changer le caractère `T` de la chaîne `"taTou"` en `"B"`.

Par construction, les chaînes de caractères sont de type *immuable* : leur contenu ne peut être modifié. Non seulement, on ne peut pas changer un caractère en un autre, mais on ne peut en supprimer ni en rajouter, à quelque endroit que ce soit.

EXERCICES

Doubler les lettres

À partir d'une chaîne telle que **bali**, on veut construire la chaîne **bbaallii**.

Plus précisément, étant donné une chaîne de caractères, sans accent, construire une nouvelle chaîne telle que chaque caractère de la chaîne initiale soit doublé.

Distance de Hamming

Étant donné deux chaînes de caractères `s` et `t`, supposées de même longueur, on appelle distance de Hamming entre `s` et `t`, le nombre de positions où les chaînes ont des caractères différents. Par exemple, si `s` et `t` sont les chaînes

```
s = "pointes"
t = "voisins"
```

alors, la distance de Hamming entre ces deux chaînes est 4.

Ecrire une fonction `hamming(s, t)` qui renvoie la distance de Hamming entre deux chaînes de caractères `s` et `t` de même longueur. Par exemple, si `s` et `t` sont les chaînes ci-dessus, alors `hamming(s, t) = 4`.

Doubler les voyelles sauf certaines

On donne un mot `M` formé de lettres minuscules sans accent. On veut construire un mot `MM` qui s'obtient à partir de `M` en en gardant les mêmes lettres sauf que chaque voyelle est doublée à l'exception des deux cas suivants :

— la voyelle est la dernière lettre du mot,

— la voyelle est suivi de la lettre **n**.

Les voyelles sont parmi les lettres **a**, **e**, **i**, **o**, **u** et **y**.

Voici quelques exemples d'exécution :

```
enduire : enduuiire
ordinateur : oordinaateeuur
nsi : nsi
lenteur : lenteeuur
python : pyython
java : jaava
c : c
```

On pourra parcourir les caractères de `mot` en utilisant une boucle `for` (avec ou sans indice mais la faire sans indice oblige à mémoriser le caractère précédent).

Cet exercice provient d'une question posée sur le forum Python d'OpenClassrooms.

Changer OU en O et O en OU

On donne un mot `M` formé de lettres majuscules sans accent. On veut construire un mot `MM` qui s'obtient à partir de `M` en en gardant les mêmes lettres sauf que

— si une lettre O est suivie de la lettre U, la lettre U ne sera pas gardée (par exemple NOUNOU deviendra NONO),

— si une lettre O n'est pas suivie de la lettre U, la lettre O sera remplacée par OU (par exemple DODO deviendra DOUDOU).

Voici quelques exemples d'exécution :

```
O : OU
U : U
NOUNOU : NONO
DODO : DOUDOU
OO : OUOU
OUOU : OO
OO : OUOU
OUOUO : OOOU
UOOU : UOUO
```

On pourra parcourir les caractères de `mot` en utilisant une boucle `for` avec indice et utiliser un drapeau ou passant à `True` lorsqu'une lettre O est rencontrée et qu'elle est suivie d'un U. Il est aussi possible de parvenir au même résultat en utilisant un parcours sans indice, en gérant le caractère précédent avec une variable `prv`, initialisée à la chaîne vide (par exemple).

Cet exercice provient d'une question posée sur le forum Python d'OpenClassrooms.

Palindrome

1. Ecrire une fonction `inverser(ch)` qui prend une chaîne en argument et renvoie cette chaîne mais écrite à l'envers. Par exemple, la chaîne **CITOYEN** doit renvoyer la chaîne **NEYOTIC**.
2. Le mot **RADAR** est un *palindrome* : quand on lit ses lettres de la droite vers la gauche, le mot est inchangé. Ecrire une fonction `estPalindrome(ch)` qui prend une chaîne `ch` en argument et renvoie `True` si la chaîne `ch` est un palindrome, et `False` sinon. On utilisera la question précédente. On supposera que la chaîne `ch` ne contient que des caractères alphabétiques majuscules et sans accent. On rappelle que Python permet de comparer des chaînes.

Chaînes formées seulement de lettres parmi a ou b

On appellera *babachaîne* toute chaîne constituée de caractères uniquement parmi les deux lettres minuscules **a** ou **b**. Par exemple, la chaîne **bbabaaab** ou encore la chaîne **bb** sont des *babachaîne*, mais pas **dada** (qui contient le caractère **d**).

1. Soit une *babachaîne* C. Écrire un code Python qui renvoie le nombre d'occurrences de la lettre *a* dans la chaîne C. Par exemple,
   — si C est la chaîne **ababa**, le programme doit renvoyer 3 ;
   — si C est la chaîne **bbb**, le programme doit renvoyer 0.
2. Soit une chaîne *babachaîne* C de longueur impaire. Écrire un code Python qui renvoie la lettre la plus présente dans la chaîne C. Par exemple,
   — si C est la chaîne **ababa**, le programme doit renvoyer **a** ;
   — si C est la chaîne **bbb**, le programme doit renvoyer **b**.

Mots les plus longs

On donne une liste de mots et on demande de renvoyer la liste des mots ayant la longueur maximum dans cette liste. Par exemple, si la liste a pour contenu

```
poire
salade
pomme
vanille
salade
cerise
bégonia
radis
ananas
tomate
poireau
```

un mot le plus long a pour longueur 7 et les mots de la liste ayant cette longueur sont :

```
vanille
bégonia
poireau
```

Les mots ci-dessus sous forme de liste :

```
['poire', 'salade', 'pomme', 'vanille', 'salade', 'cerise',
'bégonia', 'radis', 'ananas', 'tomate', 'poireau']
```

Conjuguer un verbe du premier groupe

On donne un verbe régulier du premier groupe (qui se terminer par **ER**), par exemple, *danser*. Ecrire un code qui affiche la conjugaison de ce verbe au présent de l'indicatif. Par exemple, avec *danser*, le code affichera :

```
je danse
tu danses
il/elle danse
nous dansons
vous dansez
ils/elles dansent
```

On pourra créer une liste de terminaisons et une liste de pronoms.

Bijou, caillou, etc

On donne un nom commun, au singulier, par exemple le mot **POIRE**, sous forme de chaîne de caractères alphabétiques majuscules. On demande d'afficher ce mot au pluriel. On suppose que le mot satisfait une des deux situations suivantes :

— il admet un pluriel régulier, qui s'obtient en ajoutant un `s`

— il se termine par *OU* auquel cas son pluriel n'est pas régulier si le nom fait partie de la liste d'exceptions données ci-dessous et auquel cas il faut ajouter un `x` et non pas un `s`.

La liste des exceptions est la suivante :

```
BIJOU, CAILLOU, CHOU, GENOU, HIBOU, JOUJOU, POU
```

Par exemple,

— si le mot est **POIRE**, il faudra construire le mot **POIRES**

— si le mot est **CHOU**, il faudra construire le mot **CHOUX**

— si le mot est **TOUTOU**, il faudra construire le mot **TOUTOUS**

On écrira une fonction `pluriel(nom)` qui renverra le mot au pluriel.

Dernières lettres d'un mot

Ecrire une fonction `term(mot, k)` qui renvoie la partie d'un mot formée de ses `k` dernières lettres. Si `k` dépasse la longueur du mot, la fonction renverra le mot en entier.

Par exemple, si `mot = "orange"` alors `term(mot, 4)` renvoie `"ange"`.

Mots en lignes, en colonnes

On donne une liste de mots, tous de même longueur. On demande d'afficher les mots, sous forme de tableau 2D et des deux façons suivantes :

— les mots sont écrits horizontalement

— les mots sont écrits verticalement, avec un trait vertical entre chaque colonne

Par exemple, si la liste des mots est :

```
mots = ["POIRE", "SAVON", "LAVER", "LABEL"]
```

alors l'affichage horizontal sera :

```
POIRE
SAVON
LAVER
LABEL
```

et l'affichage vertical sera :

```
P|S|L|L|
O|A|A|A|
I|V|V|B|
R|O|E|E|
E|N|R|L|
```

Un trait vertical s'obtient avec la combinaison `Alt Gr + 6`.

Triangle en mode texte

On donne un entier $n \geq 2$ et on demande d'écrire un code qui dessine en mode texte un triangle tel que chaque côté soit constitué de n astérisques. Par exemple, pour $n = 4$

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

ou pour $n = 5$:

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

À gauche d'une astérique, quand il semble y avoir du vide, c'est qu'il y a un caratère espace. La base du triangle alterne un caractère espace et une astérique. **Il est attendu que le code ne contienne aucune instruction** `if`.

Découper en mots

On donne une chaîne de caractères `s` et on demande de construire la liste des mots apparaissant dans cette chaîne, en respectant leur ordre d'apparition. Par exemple, si la chaîne est

```
"l'artichaut, le chou-fleur, la poire d'italie ..."
```

la liste des mots à afficher sera :

```
['l', 'artichaut', 'le', 'chou', 'fleur', 'la', 'poire', 'd', 'italie']
```

Pour simplifier, un mot sera une suite formée uniquement de caractères alphabétiques minuscules. On pourra utiliser la chaîne suivante

```
ALPHA="azertyuiopmlkjhgfdsqwxcvbn"
```

Immatriculation valide

Voici un exemple d'immatriculation de véhicule en France : **42-TDA-08**. Une immatriculation française est constituée de 9 caractères, qui se décomposent, de la gauche vers la droite en :

— deux chiffres
— un tiret
— trois lettres majuscules,
— un tiret
— deux chiffres

Écrire un programme qui, à partir d'une suite de caractères, renvoie True ou False selon que cette suite constitue, oui ou non, une immatriculation valide.
Voici des exemples de réponses attendues

Immatriculation	Réponse
42-TDA-8	False
TDA-452-AbC	False
81-ABC-75	True

On pourra utiliser les chaînes suivantes :

```
DIGITS = "0123456789"
ALPHA = "AZERTYUIOPLKJHGFDSQWXCVBN"
```

Afficher un numéronyme

On va décrire ci-dessous un procédé de compression de mot. On donne une chaîne de caractères, par exemple `s = "Kubernetes"` et on demande d'**afficher** une nouvelle chaîne `S`, avec notre choix de `s`, ce sera `S = "K8s"`. La règle de construction de la chaîne `S` est précisée ci-dessous :

— si `s` a au moins trois éléments :

  — le premier et le dernier caractères de `S` sont identiques à ceux de `s` ;

  — les caractères de `S` strictement entre le premier et de dernier caractère de `S` représentent le nombre d'éléments de `s` **strictement** compris entre les extrémités de `s`

— sinon `S = s`.

Voici quelques exemples de comportement :

```
Kubernetes -> K8s
Python -> P4n
Internationalization -> I18n
globalization -> g11n
```

On ne demande pas de construire `S`, on demande juste de l'afficher. On rappelle que l'on peut afficher des chaînes concaténées avec la fonction `print` :

```
print("Pomme", end='')
print("Pink", end='')
print("Lady")
```

```
PommePinkLady
```

Mot de passe valide

Un mot de passe `m` est considéré comme sûr si les conditions suivantes sont réalisées :

— `m` est formé d'au moins huit caractères ;

— `m` contient au moins :

  — une lettre minuscule

  — une lettre majuscule

  — un chiffre

  — un signe de ponctuation parmi le tiret, le point d'exclamation ou d'interrogation, la virgule, le point, le point-virgule, le deux-points, l'astérisque et le tilde.

Créer une variable `mot_de_passe_valide` qui vaut `True` si le mot de passe `m` est sûr et `False` sinon.

On pourra utiliser les chaînes da caractères suivantes :

```
MIN = 'azertyuiopmlkjhgfdsqwxcvbn'
MAJ = 'AZERTYUIOPMLKJHGFDSQWXCVBN'
CHIF = '0123456789'

# Ponctuation
PONCT = '-!?,.;:*~'
```

Afficher les occurrences d'un sous-mot

On donne une chaîne `s`, de caractères majuscules et on demande d'afficher toutes les occurrences du sous-mot **LOL** dans la chaîne `s`, les autres lettres étant remplacées par des points.

Par exemple si `s = "LALAOOLOLA"` alors le programme devra afficher :

```
L...O.L...
L...O...L.
L....OL...
L....O..L.
L......OL.
..L.O.L...
..L.O...L.
..L..OL...
..L..O..L.
..L....OL.
......LOL.
```

On utilisera le fait que la chaîne à détecter est courte.

L'idée de exercice est inspirée d'un problème de Codeforces.

Métathèses

Ecrire un code qui effectue un échange aléatoire de deux lettres parmi toutes les lettres du mot sauf la première. A moins que cela soit impossible, le mot généré doit être différent du mot initial. Par exemple, le mot *programmation* pourra générer le mot *prorgammation.*

Si on applique cette modification à la phrase suivante :

La raison du plus fort est toujours la meilleure, nous l'allons montrer tout à l'heure

cela pourrait produire

La raisno du puls fotr ets toujuors la muilleere, nosu la'llons mrntreo totu à l'reuhe

Lettres découvertes

On propose à un joueur de découvrir un mot secret, par exemple **CITRONNIER**. Au début, les lettres du mot secret sont cachées par des astériques :

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

Si le joueur propose le caractère `R`, il obtient la réponse :

```
***R*****R
```

S'il continue avec `N`, il obtiendra :

```
***R*NN**R
```

S'il propose maintenant la lettre `U`, il obtiendra le même résultat que ci-dessus car `U` n'est pas dans le mot secret.

On donne une chaîne représentant un mot, par exemple le mot **CITRONNIER**. On donne une chaîne `essais`, par exemple `essais = "RNU"`. Les lettres de cette chaîne représentent les lettres successives proposées par le joueur. On demande d'afficher le mot au fur et à mesure que les lettres sont proposées.

Par exemple, si le mot est **CITRONNIER** et si la suite des lettres est `essais="RNUIB"` alors le programme devra afficher :

```
**********
***R*****R
***R*NN**R
***R*NN**R
*I*R*NNI*R
*I*R*NNI*R
```

Séparer par des majuscules

On donne une phrase écrite en minuscules, et dont les mots sont séparés par un ou plusieurs espaces, par exemple

```
phrase ="la   vie est      un long fleuve tranquille"
```

et on demande de produire une chaîne issue de la phrase et telle que les mots soient accollés les uns aux autres mais commencent par une majuscule. Avec l'exemple ci-dessus, on obtiendra

```
LaVieEstUnLongFleuveTranquille
```

Pour produire un caractère majuscule à partir d'un caractère minuscule, on utilisera la méthode `upper`, comme illustré ci-dessous :

```
a_min = "a"
a_maj = a_min.upper()

print(a_maj)
```

qui affiche

```
A
```

Pour répondre à la question, on partira d'une chaîne `reponse` contenant uniquement le premier caractère écrit en majuscule, on parcourra la phrase avec un indice, on fera croître la chaîne `reponse` en utilisant un drapeau booléen `ESP` qui indiquera si oui ou non le parcours est en train d'examiner une suite d'espaces.

Les deux premiers mots

On donne une chaîne de caractères contenant au moins un espace, par exemple

`s = "Le travail est un trésor"`.

On demande d'en extraire la sous-chaîne formée de deux premiers mots, un mot étant délimité par le début ou la fin de la chaîne ou par un espace.

Par exemple, si `s = "Le travail est un trésor"` alors la chaîne à extraire est `"Le travail"`.

On pourra parcourir avec une boucle while la chaîne `s` à la recherche du premier caractère *espace* puis avec une autre boucle while le caractère *espace* suivant ou la fin de la chaîne (on peut aussi n'utiliser qu'une boucle `while` et compter le nombre d'espaces).

Supprimer les espaces autour d'une chaîne

Ecrire une fonction `strip(s)` qui partant d'une chaîne de caractères `s` renvoie la chaîne identique à `s` sauf que les caractères *Espace* placés en début et fin de la chaîne `s` ont été retirés. Pour cela, on pourra procéder comme suit :

— chercher le plus petit indice `i` d'un caractère de `s` différent du caractère `" "` (si `s` n'est formée que d'espaces, on posera que `i` vaut la longueur de la chaîne) ;
— chercher le plus grand indice `j` $\geq$ `i` d'un caractère de `s` différent du caractère `" "`
— construire la sous-chaîne de `s` entre l'indice `i` et l'indice `j`.

On utilisera une boucle while pour les deux premières étapes.

Nettoyer un message

On donne un message sous forme d'une chaîne de caractères. On doit récrire ce message mais en ignorant :

— tout caractère qui est un espace
— toute portion du message placée entre deux astériques consécutives, astérisques incluses.

Par exemple, le message suivant

```
VI  C*DEST*TO *TREM BLER*IRE
```

sera transformé en le message

```
VICTOIRE
```

Ecrire un code qui tranforme un message donné comme indiqué ci-dessus. On parcourra la chaîne et on utilisera un drapeau indiquant si le parcours se trouve, oui ou non, dans une portion située entre deux astérisques.

Cet exercice est inspiré de la sélection au concours Prologin 2020.

Séparer par blocs de 3 chiffres

On donne un entier positif sous forme d'une chaîne de caractères formée exclusivement de chiffres entre 0 et 9, par exemple la chaîne

```
"14959787070"
```

On demande de construire une chaîne de caractères représentant le même nombre sauf que ses chiffres sont reproupés par paquets de 3 à partir du chiffres des unités, les blocs étant séparés par des caractères _ (un blanc souligné).

Voici quelques exemples de comportements du programme :

```
0 → 0
5 → 5
42 → 42
142 → 142
2038 → 2_038
149597870 → 149_597_870
14959787070 → 14_959_787_070
```

On pourra examiner le reste de `n` modulo 3 où `n` est le nombre de chiffres.

Supprimer la virgule et les zéros inutiles

On donne un entier positif sous forme d'une chaîne de caractères `s` formée de chiffres entre 0 et 9 et éventuellement d'une virgule, par exemple :

```
"00042"
"2030,0"
"2030,00000"
"2030,"
"2030"
"00000"
"000,00"
```

Le nombre donné est un entier positif donc une chaîne telle que `s = "3,14"` est exclue. On demande de représenter le même entier par une chaîne S ne contenant ni virgule ni zéros inutiles. Les zéros dits *inutiles* se trouvent en tête du nombre ou en fin de nombre, après la virgule. On écrira une fonction `supprimer(s)`. Voici quelques exemples d'exécution du programme :

```
s = 00042 → S = 42
s = 2030,0 → S = 2030
s = 2030,00000 → S = 2030
s = 2030, → S = 2030
s = 2030 → S = 2030
s = 00000 → S = 0
s = 000,00 → S = 0
```

Chaîne représentant un entier ou pas

On donne une chaîne de caractères `s` représentant un nombre positif ou nul `x`. On admettra que la chaîne contient exclusivement des chiffres entre 0 et 9 et éventuellement le caractère *virgule*, présent au plus une fois. On demande de créer un booléen qui vaut `True` si le nombre s'évalue en un nombre entier et `False` sinon. Voici quelques exemples de comportement du programme :

```
s = "42" → True
s = "2030,0" → True
s = "2030," → True
s = "2030,00000" → True
s = "3,140000" → False
```

Nom valide ou pas

On donne une chaîne de caractères alphanumériques (chiffres et lettres non accentuées minuscules ou majuscules) et on considère que la chaîne est valide si elle vérifie les trois conditions suivantes :

— elle commence par une lettre majuscule

— elle comporte au maximum 2 caractères successifs identiques

— sa taille est d'au minimum 5 caractères et, au maximum, 15 caractères (bornes incluses).

Noter que deux lettres, l'une minuscule et l'autre majuscule, sont considérées comme différentes.

On donne une chaîne et on demande de construire une fonction `isValid(s)` qui renvoie `True` si la chaîne `s` est valide et `False` autrement.

Voici quelques exemples de comportement du programme :

```
Abricot : Valide
Abricot42 : Valide
```

```
AbbriCOOot : Valide
Abri : Invalide
abricot : Invalide
42abricot : Invalide
Abricot0123456789 : Invalide
Abbbricot : Invalide
Abricot3000 : Invalide
```

On pourra utiliser la chaîne suivante :

```
ALPHA = "AZERTYUIOPLMKJHGFDSQWXCVBN"
```

Cet exercice est inspiré de l'exercice Joli nom donné en 1/2 finale de Prologin 2018.

Mot dans une phrase à une position donnée

On donne une chaîne de caratères représentant une phrase formée de mots. Un mot, par définition est entouré d'espaces ou est déterminé par le début ou la fin de la chaîne. On donne un indice valide `k` ≥ 0 de la chaîne. On demande d'écrire une fonction `isoler(s, k)` qui détermine l'unique mot dont une des lettres est à l'indice donné. Si l'indice `k` indique dans `s` un caractère espace, la fonction renverra la chaîne vide.

L'exercice a été proposé sur le forum du site OpenClassrooms.

Voici quelques exemples d'exécution sur la chaîne `" Hello You "` :

```
k = 0
lettre :
Mot :
----------
k = 1
lettre : H
Mot : Hello
----------
k = 2
lettre : e
Mot : Hello
----------
k = 3
lettre : l
Mot : Hello
----------
k = 4
lettre : l
Mot : Hello
----------
k = 5
lettre : o
Mot : Hello
----------
k = 6
lettre :
Mot :
```

```
----------
k = 7
lettre : Y
Mot : You
----------
k = 8
lettre : o
Mot : You
----------
k = 9
lettre : u
Mot : You
----------
k = 10
lettre :
Mot :
----------
```

Découper un texte en lignes

On donne un texte sous la forme d'une chaîne de caractères ne contenant aucun saut de ligne. On testera son code sur la chaîne `s` donné dans le code Python suivant :

```
s = ("Lorem ipsum dolor sit amet, consectetur ad"
     "ipiscing elit. Maecenas eu nisl id lectus "
     "malesuada dignissim. Etiam leo justo, dapi"
     "bus non purus tempus, consequat luctus jus"
     "to. Donec id tincidunt odio, in consequat "
     "metus. In hac habitasse platea dictumst. I"
     "n efficitur consectetur luctus. Fusce et a"
     "ccumsan arcu. Nam convallis at odio nec he"
     "ndrerit. Cras pulvinar, enim in viverra da"
     "pibus, ante risus lobortis nisl, vel biben"
     "dum massa velit nec nulla. Sed viverra ris"
     "us et diam pulvinar, in dignissim ex hendr"
     "erit. Sed ipsum lacus, porttitor vitae mol"
     "estie ultrices, semper in magna.")

print(s)
```

Malgré les apparences, ce texte ne contient aucun saut de ligne, cela se voit à l'affichage produit par la fonction `print`.

(1) On veut découper ce texte en lignes qui fassent *exactement* 30 caractères, sauf éventuellement la dernière ligne. Le texte doit être préservé : on ne fait qu'ajouter des caractère sauts de lignes `\n`.
Écrire un code qui renvoie une chaîne de caractères représentant le nouveau texte auquel on a ajouté des sauts de ligne. Voici les 10 premières lignes pour l'exemple ci-dessus :

```
Lorem ipsum dolor sit amet, co
nsectetur adipiscing elit. Mae
cenas eu nisl id lectus malesu
ada dignissim. Etiam leo justo
, dapibus non purus tempus, co
nsequat luctus justo. Donec id
 tincidunt odio, in consequat
metus. In hac habitasse platea
 dictumst. In efficitur consec
tetur luctus. Fusce et accumsa
```

(2) On veut cette fois découper le texte en un nombre minimum de lignes, qui fassent au plus 30 caractères et en sorte que les sauts de ligne ne coupent pas un mot en deux. Le texte doit être préservé : on ne fait qu'ajouter des sauts de lignes. On garantit qu'il n'existe aucun mot dans le texte formé de plus de 30 caractères.
Écrire un code qui renvoie une chaîne de caractères représentant le nouveau texte auquel on a ajouté des sauts de ligne. Voici les 10 premières lignes pour l'exemple ci-dessus :

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
Maecenas eu nisl id lectus
malesuada dignissim. Etiam
leo justo, dapibus non purus
tempus, consequat luctus
justo. Donec id tincidunt
odio, in consequat metus. In
hac habitasse platea
dictumst. In efficitur
```

Répéter un motif et tronquer

On donne une chaîne de caractères `s`, par exemple `Orange`. On donne un entier positif `n` et on demande de construire la chaîne `S` obtenue si on répétait bout à bout la chaîne `s` et qu'on ne conservait que les `n` premiers caractères de la répétition.

Par exemple, pour la chaîne `s = Orange` et `n = 39`, on obtient pour `S` la chaîne :

```
OrangeOrangeOrangeOrangeOrangeOrangeOra
```

Cet exercice est la version pour les chaînes de caractères d'un exercice antérieur intitulé Dupliquer une liste sur une certaine longueur et qui s'appliquait aux listes.