

# Programmations : Concepts fondamentaux- Pratique

Cours Python

Bachelier en informatique– Finalité Télécommunications et réseaux –  
Bloc 1.

Rédigé par :

Depreter Johan – [johan.depreter@heh.be](mailto:johan.depreter@heh.be)

Desmet Erwin – [erwin.desmet@heh.be](mailto:erwin.desmet@heh.be)



## Table des matières

1.	Introduction / Rappels des concepts .....	11
1.1	Introduction .....	11
1.1.1	Comment fonctionne un programme informatique ? .....	11
1.1.2	Les langages naturels vs les langages de programmation .....	12
1.1.3	De quoi est composé un langage ? .....	13
1.1.4	Compilation ou Interprétation .....	14
1.1.5	Que fais un interpréteur ? .....	15
1.1.6	Qu'est-ce que Python ? .....	17
1.1.7	Qui a créé Python ? .....	17
1.1.8	Les objectifs de Python .....	18
1.1.9	Pourquoi Python est si spécial et adapté pour vous ? .....	19
1.1.10	Les concurrents directs de Python au niveau prédisposition .....	20
1.1.11	Où Python semble peu porteur ? .....	20
1.2	Les différents types de Python .....	21
1.2.1	Python 2 ou Python 3 ? .....	21
1.2.2	Python aka CPython .....	22
1.2.3	Cython .....	22
1.2.4	Jython .....	23
1.2.5	PyPy et RPython .....	23
1.3	Comment obtenir Python et l'utiliser ? .....	24
1.3.1	Obtenir Python .....	24
1.3.2	Testons l'installations de Python .....	25
1.3.3	Règle importante .....	26
1.3.4	Ecrire son premier programme .....	26
2.	Types de données, variables, opérations entrées-sorties, opérateurs de base .....	27
2.1	Premier programme .....	27
2.1.1	Hello World .....	27
2.1.2	La fonction print() .....	28
2.1.3	La sémantique de print() .....	30
2.1.4	Mais à quoi sert ce print ? .....	32
2.1.5	Les instructions .....	33
2.1.6	Les caractères d'échappement et les nouvelles lignes .....	34
2.1.7	Ajoutons des arguments .....	35
2.1.8	Les mots-clés de la fonction print() .....	36
2.1.9	Exercices .....	38
2.1.10	Résumé .....	40

2.2. Les littéraux.....	41
2.2.1 Introduction .....	41
2.2.2 Faisons une expérience .....	41
2.2.3 Les entiers (integers).....	42
2.2.3 Les Réels ou nombres flottants (float) .....	44
2.2.4 La guerre des entiers et des réels .....	45
2.2.5 Le codage des nombres flottants.....	46
2.2.6 Les strings.....	47
2.2.7 Valeurs booléennes (Boolean) .....	48
2.2.8 Exercice .....	49
2.2.9 Résumé .....	50
2.3 Les opérateurs.....	51
2.3.1 Python est une calculatrice .....	51
2.3.2 Les opérateurs de bases .....	51
2.3.3 L'exponentiel .....	52
2.3.4 La multiplication .....	52
2.3.5 La division .....	52
2.3.6 La division entière .....	53
2.3.7 Le modulo .....	54
2.3.8 La division par zéro.....	55
2.3.9 L'addition .....	55
2.3.10 L'opérateur de soustraction, les opérateurs unaires et binaires ...	55
2.3.11 Les priorités des opérations.....	56
2.3.12 La liste des priorités .....	57
2.3.13 Les parenthèses .....	57
2.3.14 Résumé.....	58
2.4 Les variables .....	59
2.4.1 Introduction.....	59
2.4.2 Les noms corrects .....	60
2.4.3 Les mots-clés.....	60
2.4.4 Créaton de variables .....	61
2.4.5 Essayons d'affecter une variable déjà existante.....	63
2.4.6 Pythagore et les mathématiques.....	64
2.4.7 Exercice .....	65
2.4.8 Les opérateurs raccourcis .....	66
2.4.9 Exercices.....	67
2.4.10 Résumé.....	70
2.5 Les commentaires .....	71

2.5.1 Laisser des commentaires dans le code : pourquoi, comment et quand .....	71
2.5.2 Exercice .....	73
2.5.3 Résumé .....	74
2.6 Parlons avec notre ordinateur/programme .....	75
2.6.1 La fonction input().....	75
2.6.2 Ajoutons un argument à notre fonction .....	76
2.6.3 Le résultat de notre fonction input ( ) .....	76
2.6.4 La fonction input() - opérations interdites .....	76
2.6.5 Formatage des types (cast).....	77
2.6.6 Bonus sur input() et la conversion de type .....	78
2.6.7 Opérateurs de chaîne – introduction .....	79
2.6.8 exercices.....	81
2.6.9 Résumé .....	84
3. Valeurs booléennes et exécution conditionnelle.....	85
3.1 Les comparateurs .....	85
3.1.1 Questions et réponses.....	85
3.1.2 La comparaison ou l'opérateur d'égalité.....	85
3.1.3 L'opérateur d'inégalité .....	86
3.1.4 Opérateur de comparaison : supérieur à .....	87
3.1.5 Opérateur de comparaison : supérieur ou égal à .....	87
3.1.6 Opérateurs de comparaison : inférieure à et inférieur ou égal à .....	87
3.1.7 Que faire des réponses ? .....	88
3.1.8 Exercice.....	88
3.2 Prendre des décisions en Python .....	89
3.2.1 Les conditions et leur exécution.....	89
3.2.2 La condition – if .....	91
3.2.3 La condition – if-else.....	91
3.2.4 L'imbrication sorcellerie ou réalité ? .....	92
3.2.5 La condition – elif.....	93
3.2.6 Analyse des exemples de code .....	94
3.2.7 Pseudo-code et introduction aux boucles .....	95
3.2.8 Exercices .....	98
3.2.9 Résumé .....	101
4. Les boucles .....	104
4.1 La boucle while.....	104
4.1.1 La boucle infinie .....	105
4.1.2 Exemple bonus .....	106

4.1.3 Utilisation d'un compteur dans une boucle .....	107
4.1.4 Exercice .....	108
4.2 La boucle for .....	109
4.2.1 La boucle for et son fonctionnement.....	109
4.2.2 Encore plus de pouvoir avec 3 arguments .....	110
4.2.3 Exercice .....	112
4.2.3 Les mots-clés : break et continue .....	113
4.2.4 Quelques exemples .....	114
4.2.5 Exercices .....	115
4.2.6 Le else et la boucle while.....	118
4.2.7 Le else et la boucle for .....	118
4.2.8 Exercices .....	119
4.3 Résumé.....	121
5. La logique informatique.....	124
5.1 Introduction .....	124
5.1.1 Le et .....	124
5.1.2 Le ou .....	125
5.1.3 Le non .....	125
5.2 Les expressions logique .....	125
5.3 Valeurs logiques vs bits uniques.....	126
5.4 Opérateurs binaires.....	126
5.5 Résumé.....	128
6. Les listes.....	130
6.1 La base des listes .....	130
6.1.1 Pourquoi les listes sont-elles utiles ? .....	130
6.1.2 L'index.....	131
6.1.3 Accès au contenu d'une liste .....	132
6.1.4 Supprimer les éléments de la liste. ....	133
6.1.5 Les indices négatifs .....	133
6.1.6 Exercice .....	134
6.1.7 Fonctions vs Méthodes .....	135
6.1.8 Ajout d'éléments à une liste : append() et insert() .....	136
6.1.9 Utilisons mieux nos listes.....	137
6.1.10 Travaillons avec nos listes.....	138
6.1.11 Exercice .....	140
6.1.12 Résumé.....	141
6.2 Trier une liste simple.....	143
6.2.1 Le tri à bulles.....	143

6.2.2 Créons une version interactive du tri .....	145
6.2.3 Résumé .....	146
6.3 Les opérations sur les listes.....	147
6.3.1 la vie intérieure des listes.....	147
6.3.2 Les indices négatives dans le slice .....	148
6.3.3 Spécificités du slice, les omissions. ....	149
6.3.4 Spécificités du slice avec del .....	149
6.3.5 Les opérateurs in et not in.....	150
6.3.6 Quelques exemples de programmes.....	151
6.3.7 Exercice.....	153
6.3.8 Résumé .....	153
6.4 Listes multidimensionnelles .....	156
6.4.1 Une liste dans une liste .....	156
6.4.2 Passons en 2 D .....	157
6.4.3 Applications avancées du multidimensionnel.....	159
6.4.4 Le tridimensionnel .....	161
6.4 Résumé.....	162
7. Les fonctions .....	164
7.1 Les fonctions.....	164
7.1.1 Pourquoi en avons-nous besoin ?.....	164
7.1.2 La décomposition .....	165
7.1.3 Mais d'où viennent les fonctions ? .....	166
7.1.4 Première fonction .....	166
7.1.5 Allons plus loin dans le fonctionnement .....	168
7.1.6 Résumé .....	170
7.2 Les fonctions paramétrées .....	171
7.2.1 Transmission de paramètres positionnels.....	174
7.2.2 Passage d'arguments par mot-clé .....	175
7.2.3 Mélange d'arguments positionnels et de mots-clés .....	175
7.2.4 Fonctions paramétrées, infos complémentaires.....	177
7.2.5 Résumé .....	178
7.3 Effets et résultats des fonctions .....	180
7.3.1 L'instruction de retour.....	180
7.3.2 Quelques mots sur None .....	182
7.3.4 Listes et fonctions .....	184
7.3.5 Exercices .....	185
7.3.6 Résumé .....	190
7.4 Fonctions et portées.....	192

7.4.1 Le mot-clé global.....	193
7.4.2 L'interaction avec les arguments.....	194
7.4.3 Résumé .....	196
7.5 Création de fonctions simples .....	198
7.5.1 Création d'une fonction à deux paramètres .....	198
7.5.2 Jouons avec un triangle.....	201
7.5.3 Les factorielles.....	204
7.5.4 La suite de Fibonacci .....	205
7.5.5 La récursion .....	206
7.5.6 Résumé .....	207
8 Tuples et dictionnaire.....	208
8.1 Types de séquences et mutabilité.....	208
8.2 Qu'est-ce qu'un tuple ?.....	209
8.2.1 Comment créer un tuple ? .....	209
8.2.2 Comment utiliser un tuple?.....	210
8.3 Qu'est-ce qu'un dictionnaire?.....	212
8.3.1 Comment faire un dictionnaire? .....	213
8.3.2 Comment utiliser un dictionnaire?.....	214
8.3.4 Comment utiliser un dictionnaire: les clés().....	215
8.3.5 La fonction sorted() .....	216
8.3.6 Comment utiliser un dictionnaire : méthodes items() et values()..	216
8.3.7 Comment utiliser un dictionnaire : modification et ajout de valeurs .	217
8.3.8 Ajout d'une nouvelle clé .....	217
8.3.9 Suppression d'une clé .....	218
8.4 Les tuples et les dictionnaires peuvent fonctionner ensemble .....	218
8.5 Résumé des tuples.....	220
8.6 Résumé des dictionnaires.....	222
8.7 Résumé : tuples et dictionnaires .....	224
9. Les exceptions .....	226
9.1 Le pain quotidien du développeur.....	226
9.2 Erreur de code ou erreur de données.....	226
9.3 Quand les données ne sont pas ce qu'elles devraient.....	227
9.4 le Try-except.....	228
9.5 L'exception prouve la règle.....	229
9.6 La gestion de plusieurs erreurs possibles.....	230
9.7 L'exception par défaut .....	231
9.8 Les exceptions les plus courantes.....	231
9.8.1 ZeroDivisionError .....	231

9.8.2 ValueError.....	231
9.8.3 TypeError.....	232
9.8.4 AttributeError .....	232
9.8.5 SyntaxError.....	232
10 Les tests .....	233
10.1 Tester son code est une obligation, une loi DSTienne ! .....	233
10.2 Quand python ferme les yeux .....	234
10.3 Tests, essais et testeurs.....	235
10.3.1 Bug vs débogage.....	235
10.4 Débug d'impression .....	236
10.5 Quelques conseils utiles .....	236
10.6 Les tests unitaires -Le codage de haut niveau .....	237
10.7 Résumé chapitre 9 et 10 .....	238
10.8 Exercice.....	241
11 Les chaînes de caractères et les listes en détails .....	245
11.1 La nature des chaînes de caractères en Python.....	245
11.1.1 Les chaines en détails .....	245
11.1.2 Chaînes multilignes.....	246
11.1.3 Opérations sur les chaînes .....	247
11.1.4 Opérations sur les chaînes : ord() .....	248
11.1.5 Opérations sur les chaînes : chr() .....	248
11.1.6 Chaînes en tant que séquences: indexation .....	249
11.1.7 Chaînes en tant que séquences: itération .....	249
11.1.8 Tranches/plages .....	249
11.1.9 Les opérateurs in et not in .....	250
11.1.10 Les chaînes Python sont immuables.....	252
11.1.11 Opérations sur chaînes : min() .....	253
11.1.12 Opérations sur les chaînes : max() .....	254
11.1.13 Opérations sur les chaînes : méthode index() .....	254
11.1.14 Opérations sur les chaînes : fonction list() .....	255
11.1.15 Opérations sur les chaînes : méthode count() .....	255
11.1.16 Résumé.....	255
11.2 Les méthodes des string .....	257
11.2.1 La méthode capitalize() .....	257
11.2.2 La méthode center() .....	258
11.2.3 La méthode endswith().....	258
11.2.4 La méthode find().....	259
11.2.5 La méthode isalnum() .....	260



11.2.6 La méthode isalpha()	261
11.2.7 La méthode isdigit()	261
11.2.8 La méthode islower()	261
11.2.9 La méthode isspace()	262
11.2.10 La méthode isupper()	262
11.2.11 La méthode join()	262
11.2.12 La méthode lower()	263
11.2.13 La méthode lstrip()	263
11.2.14 La méthode replace()	264
11.2.15 La méthode rfind()	264
11.2.16 La méthode rstrip()	265
11.2.17 La méthode split()	265
11.2.18 La méthode startswith()	265
11.2.19 La méthode strip()	265
11.2.20 La méthode swapcase()	266
11.2.21 La méthode title()	266
11.2.22 La méthode upper()	266
11.2.23 Résumé	267
11.2.24 Exercice	269
11.3 Les strings en folie	270
11.3.1 Comparaison de chaînes	270
11.3.2 Classement	271
11.3.4 Chaînes et nombres	273
11.3.5 Principaux points à retenir	274
11.3.6 Exercice	275
11.4 Quelques petits projets	277
11.4.1 Le chiffre de César : chiffrer un message	277
11.4.2 Le Chiffrement César : décrypter un message	278
11.4.3 Le processeur de nombres	279
11.4.4 Le validateur IBAN	280
11.4.5 Principaux points à retenir	282
11.4.6 Exercices	283
11.5 Les erreurs, pain quotidien du programmeur	289
11.5.1 Erreurs, échecs et autres fléaux	289
11.5.2 Les exceptions	290
11.5.3 L'anatomie d'une exception	290
15.5.3 Raise	294
15.5.4 assert	296

15.5.5 Principaux points à retenir .....	297
15.6 Exception utile .....	299
15.6.1 Exceptions intégrées .....	299
IndexError.....	300
15.6.2 Exercice .....	303
15.6.3 Résumé.....	304
16 Les fichiers .....	305
16.1 Accès aux fichiers à partir du code Python .....	305
16.2 Noms de fichiers .....	305
16.3 Flux de fichiers .....	308
16.4 Gestion des fichiers .....	309
16.5 Ouverture des flux .....	311
16.5.1 Ouverture des flux : modes.....	312
16.5.2 Sélection des modes texte et binaire .....	312
16.5.3 Ouverture du flux pour la première fois.....	313
16.5.4 Flux pré-ouverts .....	314
16.5.5 Fermeture des flux .....	315
16.5.6 Diagnostic des problèmes de flux .....	315
16.5.7 Diagnostic des problèmes de flux, la suite .....	316
16.5.8 Principaux points à retenir .....	317
16.6 Le travail sur les fichiers texte .....	319
16.6.1 Traitement des fichiers texte .....	319
16.6.2 Traitement des fichiers texte : readline() .....	322
16.6.3 Traitement des fichiers texte : readlines() .....	322
16.6.4 Traitement des fichiers texte : on s'accroche ! .....	324
16.6.5 Traitement des fichiers texte : write() .....	324
16.6.6 Traitement des fichiers texte : Avec des lignes entières Mr ?.....	325
16.6.7 Exercices.....	326
16.6.8 Résumé.....	329

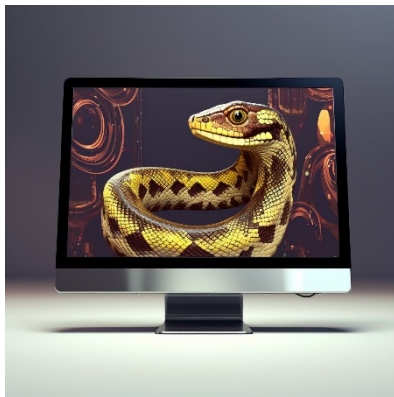
## 1. Introduction / Rappels des concepts

### 1.1 Introduction

#### 1.1.1 Comment fonctionne un programme informatique ?

Ce cours vise à vous montrer ce qu'est le langage Python et à quoi il sert. Commençons par les bases absolues.

Un programme rend un ordinateur utilisable. Sans programme, un ordinateur, même le plus puissant, n'est rien de plus qu'un objet. De même, sans joueur, un piano n'est rien de plus qu'une boîte en bois.



Les ordinateurs sont capables d'effectuer des tâches très complexes, mais cette capacité n'est pas innée. La nature d'un ordinateur est tout à fait différente. Il ne peut exécuter que des opérations extrêmement simples. Par exemple, un ordinateur ne peut pas comprendre la valeur d'une fonction mathématique compliquée par lui-même, bien que cela ne soit pas hors du domaine du possible dans un proche avenir. Les ordinateurs contemporains ne peuvent évaluer que les résultats d'opérations très fondamentales, comme l'addition ou la division, mais ils peuvent le faire très rapidement et peuvent répéter ces actions

pratiquement n'importe quel nombre de fois.

Imaginez que vous voulez connaître la vitesse moyenne que vous avez atteinte au cours d'un long voyage. Vous connaissez la distance, vous connaissez l'heure, vous avez besoin de la vitesse.

Naturellement, l'ordinateur sera capable de calculer cela, mais l'ordinateur n'est pas conscient de choses telles que la distance, la vitesse ou le temps.

Par conséquent, il est nécessaire de demander à l'ordinateur de:

- accepter un nombre représentant la distance;
- accepter un nombre représentant le temps de déplacement;
- diviser la première valeur par la seconde et stocker le résultat dans la mémoire ;
- Affichez le résultat (représentant la vitesse moyenne) dans un format lisible.

Ces quatre actions simples forment un **programme**. Bien sûr, ces exemples ne sont pas formalisés, et ils sont très loin de ce que l'ordinateur peut comprendre, mais ils sont assez bons pour être traduits dans une langue que l'ordinateur peut accepter.

**La langue** est le mot-clé.

### 1.1.2 Les langages naturels vs les langages de programmation

Une langue est un moyen (et un outil) d'exprimer et d'enregistrer des pensées. Il y a beaucoup de langues tout autour de nous. Certains d'entre eux ne nécessitent ni parler ni écrire, comme le langage corporel; Il est possible d'exprimer vos sentiments les plus profonds très précisément sans dire un mot.

Une autre langue que vous utilisez chaque jour est votre langue maternelle, que vous utilisez pour manifester votre volonté et méditer sur la réalité. Les ordinateurs ont aussi leur propre langage, appelé **langage machine**, qui est très rudimentaire.

Un ordinateur, même le plus sophistiqué techniquement, est dépourvu même d'une trace d'intelligence. On pourrait dire que c'est comme un chien bien dressé - il ne répond qu'à un ensemble prédéterminé de commandes connues.

Les commandes qu'il reconnaît sont très simples. On peut imaginer que l'ordinateur réponde à des ordres comme « prendre ce nombre, diviser par un autre et enregistrer le résultat ».

Un ensemble complet de commandes connues est appelé une **liste d'instructions**, parfois abrégée en **IL**. Différents types d'ordinateurs peuvent varier en fonction de la taille de leurs IL, et les instructions peuvent être complètement différentes dans différents modèles.

**Remarque :** les langages machine sont développés par des humains.

Aucun ordinateur n'est actuellement capable de créer une nouvelle langue. Cependant, cela pourrait bientôt changer. Tout comme les gens utilisent un certain nombre de langages très différents, les machines ont aussi de nombreux langages différents. La différence, cependant, est que les langues humaines se sont développées naturellement.

De plus, ils continuent d'évoluer, et de nouveaux mots sont créés chaque jour à mesure que les anciens disparaissent. Ces langues sont appelées langues **naturelles**.

### 1.1.3 De quoi est composé un langage ?

On peut dire que chaque langage (machine ou naturel, peu importe) se compose des éléments suivants :

- un **alphabet** : un ensemble de symboles utilisés pour construire des mots d'une certaine langue (par exemple, l'alphabet latin pour l'anglais, l'alphabet cyrillique pour le russe, le kanji pour le japonais, etc.)
- un **lexique** : (alias un dictionnaire) un ensemble de mots que la langue offre à ses utilisateurs (par exemple, le mot « computer » vient du dictionnaire de langue anglaise, alors que « mcopture » ne le fait pas ; le mot « chat » est présent dans les dictionnaires anglais et français, mais leurs significations sont différentes)
- une **syntaxe** : un ensemble de règles (formelles ou informelles, écrites ou ressenties intuitivement) utilisées pour déterminer si une certaine chaîne de mots forme une phrase valide (par exemple, « Je suis un python » est une phrase syntaxiquement correcte, alors que « Je un python suis » ne l'est pas)
- **sémantique**: un ensemble de règles déterminant si une certaine phrase a un sens (par exemple, « J'ai mangé un beignet » a du sens, mais « Un beignet m'a mangé » n'en a pas)

L'IL est, en fait, **l'alphabet d'un langage machine**. C'est l'ensemble de symboles le plus simple et le plus primaire que nous puissions utiliser pour donner des commandes à un ordinateur. C'est la langue maternelle de l'ordinateur.

Malheureusement, cette langue est loin d'être une langue maternelle humaine. Nous avons tous (ordinateurs et humains) besoin de quelque chose d'autre, d'un langage commun pour les ordinateurs et les humains, ou d'un pont entre les deux mondes différents.

Nous avons besoin d'un langage dans lequel les humains peuvent écrire leurs programmes et d'un langage que les ordinateurs peuvent utiliser pour exécuter les programmes, un langage beaucoup plus complexe que le langage machine et pourtant beaucoup plus simple que le langage naturel.

Ces langages sont souvent appelés langages de programmation de haut niveau. Ils sont au moins un peu similaires aux naturels en ce sens qu'ils utilisent des symboles, des mots et des conventions lisibles par les humains. Ces langages permettent aux humains d'exprimer des commandes à des ordinateurs beaucoup plus complexes que ceux proposés par les IL.

Un programme écrit dans un langage de programmation de haut niveau est appelé code **source** (par opposition au code machine exécuté par les ordinateurs). De même, le fichier contenant le code source est appelé fichier source.

#### 1.1.4 Compilation ou Interprétation

La programmation informatique est l'acte de composer les éléments du langage de programmation sélectionné dans l'ordre qui provoquera l'effet désiré. L'effet peut être différent dans chaque cas spécifique - cela dépend de l'imagination, des connaissances et de l'expérience du programmeur.

Bien sûr, une telle composition doit être correcte à plusieurs égards:

- **alphabétique** - un programme doit être écrit dans un script reconnaissable, tel que romain, cyrillique, etc.
- **lexicalement** - chaque langage de programmation a son dictionnaire et vous devez le maîtriser; Heureusement, c'est beaucoup plus simple et plus petit que le dictionnaire de n'importe quel langage naturel;
- **syntactiquement** - chaque langage a ses règles et elles doivent être respectées;
- **Sémantiquement** - le programme doit avoir un sens.

Malheureusement, un programmeur peut également faire des erreurs avec chacun des quatre sens ci-dessus. Chacun d'eux peut rendre le programme complètement inutile.

Supposons que vous avez écrit un programme avec succès. Comment persuadons-nous l'ordinateur de l'exécuter? Vous devez convertir votre programme en langage machine. Heureusement, la traduction peut être effectuée par un ordinateur lui-même, ce qui rend l'ensemble du processus rapide et efficace.

Il existe deux façons différentes de **transformer un programme d'un langage de programmation de haut niveau en langage machine** :

- **COMPILATION** - le programme source est traduit une fois (cependant, cet acte doit être répété chaque fois que vous modifiez le code source) en obtenant un fichier (par exemple, un fichier .exe si le code est destiné à être exécuté sous MS Windows) contenant le code machine; vous pouvez maintenant distribuer le fichier dans le monde entier; le programme qui effectue cette traduction est appelé compilateur ou traducteur;
- **INTERPRÉTATION** - vous (ou tout utilisateur du code) pouvez traduire le programme source chaque fois qu'il doit être exécuté; le programme effectuant ce type de transformation est appelé un interpréteur, car il interprète le code chaque fois qu'il est destiné à être exécuté; cela signifie également que vous ne pouvez pas simplement distribuer le code source tel quel, Parce que l'utilisateur final a également besoin de l'interpréteur pour l'exécuter.

Pour des raisons très fondamentales, un langage de programmation de haut niveau particulier est conçu pour entrer dans l'une de ces deux catégories.

Il y a très peu de langages qui peuvent être à la fois compilés et interprétés.

### 1.1.5 Que fais un interpréteur ?

Supposons une fois de plus que vous avez écrit un programme. Maintenant, il existe sous la forme d'un **fichier** informatique: un programme informatique est en fait un morceau de texte, de sorte que le code source est généralement placé dans des **fichiers texte**.

Remarque: il doit s'agir de « **texte pur** », sans aucune « décoration » comme différentes polices, couleurs, images intégrées ou autres médias. Vous devez maintenant appeler l'interpréteur et le laisser lire votre fichier source.

L'interprète lit le code source d'une manière courante dans la culture occidentale : de haut en bas et de gauche à droite. Il y a quelques exceptions - elles seront couvertes plus tard dans le cours.

Tout d'abord, l'interprète vérifie si toutes les lignes suivantes sont correctes (en utilisant les quatre aspects abordés précédemment).

Si l'interprète trouve une erreur, il termine son travail immédiatement. Le seul résultat dans ce cas est un **message d'erreur**.

L'interprète vous indiquera où se trouve l'erreur et ce qui l'a causée. Cependant, ces messages peuvent être trompeurs, car l'interprète n'est pas en mesure de suivre vos intentions exactes et peut détecter des erreurs à une certaine distance de leurs causes réelles. L'erreur n'est donc pas spécifiquement à la ligne précise retournée par l'interpréteur.

Par exemple, si vous essayez d'utiliser une entité d'un nom inconnu, cela provoquera une erreur, mais l'erreur sera découverte à l'endroit où elle tente d'utiliser l'entité, et non à l'endroit où le nom de la nouvelle entité a été introduit.

En d'autres termes, la raison réelle est généralement située un peu plus tôt dans le code, par exemple, à l'endroit où vous deviez informer l'interprète que vous alliez utiliser l'entité du nom.



Si la ligne semble bonne, l'interpréteur essaie de l'exécuter (note: chaque ligne est généralement exécutée séparément, de sorte que le trio « lecture-vérification-exécution » peut être répété plusieurs fois - plus de fois que le nombre réel de lignes dans le fichier source, car certaines parties du code peuvent être exécutées plus d'une fois).

Il est également possible qu'une partie importante du code soit exécutée avec succès avant que l'interpréteur ne trouve une erreur. Il s'agit d'un comportement normal dans ce modèle d'exécution.

Vous vous demandez peut-être maintenant: quel est le meilleur? Le modèle « compilateur » ou le modèle « interprétatif » ? Il n'y a pas de réponse évidente. S'il y en avait eu, l'un de ces modèles aurait cessé d'exister depuis longtemps. Les deux ont leurs avantages et leurs inconvénients.

	COMPILATION	INTERPRÉTATION
AVANTAGES	<ul style="list-style-type: none"> <li>• L'exécution du code traduit est généralement plus rapide ;</li> <li>• seul l'utilisateur doit avoir le compilateur - l'utilisateur final peut utiliser le code sans lui;</li> <li>• Le code traduit est stocké en langage machine - comme il est très difficile à comprendre, vos propres inventions et astuces de programmation resteront probablement votre secret.</li> </ul>	<ul style="list-style-type: none"> <li>• Vous pouvez exécuter le code dès que vous l'avez terminé - il n'y a pas de phases supplémentaires de traduction;</li> <li>• le code est stocké à l'aide du langage de programmation, pas celui de la machine - cela signifie qu'il peut être exécuté sur des ordinateurs utilisant différents langages de machine; Vous ne compilez pas votre code séparément pour chaque architecture différente.</li> </ul>
INCON-VÉNIENTS	<ul style="list-style-type: none"> <li>• La compilation elle-même peut être un processus très long - vous ne pourrez peut-être pas exécuter votre code immédiatement après une modification;</li> <li>• Vous devez avoir autant de compilateurs que de plates-formes matérielles sur lesquelles vous souhaitez que votre code soit exécuté.</li> </ul>	<ul style="list-style-type: none"> <li>• Ne vous attendez pas à ce que l'interprétation accélère votre code à grande vitesse - votre code partagera la puissance de l'ordinateur avec l'interpréteur, il ne peut donc pas être très rapide;</li> <li>• Vous et l'utilisateur final devez disposer de l'interpréteur pour exécuter votre code.</li> </ul>



Qu'est-ce que tout cela signifie pour vous ?

- Python est un **langage interprété**. Cela signifie qu'il hérite de tous les avantages et inconvénients décrits. Bien sûr, il ajoute certaines de ses caractéristiques uniques aux deux ensembles.
- Si vous souhaitez programmer en Python, vous aurez besoin de **l'interpréteur Python**. Vous ne pourrez pas exécuter votre code sans lui. Heureusement, **Python est gratuit**. C'est l'un de ses avantages les plus importants.

Pour des raisons historiques, les langages conçus pour être utilisés de manière interprétative sont souvent appelés **langages de script**, tandis que les programmes sources encodés à l'aide de ces langages sont appelés **scripts**.

#### 1.1.6 Qu'est-ce que Python ?

Python est un langage de programmation largement utilisé, interprété, orienté objet et de haut niveau avec une sémantique dynamique, utilisé pour la programmation à usage général.

Et bien que vous connaissiez peut-être le python comme un grand serpent, le nom du langage de programmation Python vient d'une vieille série de sketches comiques télévisés de la BBC appelée **Monty Python's Flying Circus**.

Au sommet de son succès, l'équipe des Monty Python présentait ses sketches à des publics du monde entier.

Et Puisque Monty Python est considéré comme l'un des deux nutriments fondamentaux pour un programmeur (l'autre étant la pizza), le créateur de Python a nommé le langage en l'honneur de l'émission de télévision.

#### 1.1.7 Qui a créé Python ?

L'une des caractéristiques étonnantes de Python est le fait qu'il s'agit en fait du travail d'une seule personne. Habituellement, les nouveaux langages de programmation sont développés et publiés par de grandes entreprises employant de nombreux professionnels, et en raison des règles du droit d'auteur, il est très difficile de nommer les personnes impliquées dans le projet. Python est une exception.



Il n'y a pas beaucoup de langues dont les auteurs sont connus par leur nom. Python a été créé par Guido van Rossum, né en 1956 à Haarlem, aux Pays-Bas. Bien sûr, il n'a pas développé et fait évoluer tous les composants lui-même.

La vitesse à laquelle Python s'est répandu dans le monde entier est le résultat du travail continu de milliers de programmeurs, testeurs, utilisateurs (beaucoup ne sont pas des informaticiens) et de passionnés (très souvent anonymes), mais il faut dire que la toute première idée (la graine à partir de laquelle Python a germé) est venue à une tête, celle de Mr Guido.

Les circonstances dans lesquelles Python a été créé sont un peu déroutantes.

Selon Guido van Rossum :

En décembre 1989, je cherchais un projet de programmation « hobby » qui m'occuperait pendant la semaine autour de Noël. Mon bureau (...) serait fermé, mais j'avais un ordinateur à la maison, et pas grand-chose d'autre entre les mains. J'ai décidé d'écrire un interpréteur pour le nouveau langage de script auquel j'avais pensé ces derniers temps : un descendant d'ABC qui plairait aux hackers Unix/C. J'ai choisi Python comme titre de travail pour le projet, étant d'humeur légèrement irrévérencieuse (et un grand fan de Monty Python's Flying Circus).

#### 1.1.8 Les objectifs de Python

En 1999, Guido van Rossum a défini ses objectifs pour Python :

- un langage **simple et intuitif** tout aussi puissant que ceux des principaux concurrents ;
- **l'open source**, afin que tout le monde puisse contribuer à son développement ;
- un code aussi **compréhensible** que l'anglais simple;
- **Convient aux tâches** quotidiennes, permettant des temps de développement courts.

Environ 20 ans plus tard, il est clair que toutes ces intentions ont été réalisées. Certaines sources disent que Python est le langage de programmation le plus populaire au monde, tandis que d'autres affirment que c'est le deuxième ou le troisième.

Quoi qu'il en soit, il occupe toujours un rang élevé dans le top dix du [PYPL Popularity of Programming Language](#) et du [TIOBE Programming Community Index](#).

Python n'est plus un langage jeune. Il est **mature et digne de confiance**. Ce n'est pas une merveille unique. C'est une étoile brillante dans le firmament de la programmation, et le temps passé à apprendre Python est un très bon investissement.



### 1.1.9 Pourquoi Python est si spécial et adapté pour vous ?

Comment se fait-il que les programmeurs, jeunes et vieux, expérimentés et novices, veuillent l'utiliser? Comment se fait-il que les grandes entreprises aient adopté Python et mis en œuvre leurs produits phares en l'utilisant?

Il y a plusieurs raisons - nous en avons déjà énuméré quelques-unes, mais énumérons-les à nouveau de manière plus pratique:

- c'est **facile à apprendre** - le temps nécessaire pour apprendre Python est plus court que pour beaucoup d'autres langages; cela signifie qu'il est possible de démarrer la programmation plus rapidement;
- il est **facile d'enseigner** - la charge de travail d'enseignement est inférieure à celle requise par d'autres langages; cela signifie que l'enseignant peut mettre davantage l'accent sur les techniques de programmation générales (indépendantes de la langue), sans gaspiller d'énergie sur des astuces exotiques, des exceptions étranges et des règles incompréhensibles;
- il est **facile à utiliser** pour écrire de nouveaux logiciels - il est souvent possible d'écrire du code plus rapidement en utilisant Python;
- c'est facile à comprendre - il est aussi souvent plus facile de comprendre le code de quelqu'un d'autre plus rapidement s'il est écrit en Python;
- il est facile à obtenir, à **installer et à déployer** - Python est gratuit, ouvert et multiplateforme; tous les langages ne peuvent pas s'en vanter.



Bien sûr, Python a aussi ses inconvénients:

- ce n'est pas un démon de vitesse - Python n'offre pas de performances exceptionnelles;
- dans certains cas, il peut être résistant à certaines techniques de test plus simples - cela peut signifier que le débogage du code de Python peut être plus difficile qu'avec d'autres langages; **Heureusement que faire des erreurs est toujours plus difficile en Python.**

Il convient également de préciser que Python n'est pas la seule solution de ce type disponible sur le marché informatique.

Il a beaucoup d'adeptes, mais il y en a beaucoup qui préfèrent d'autres langages et ne considèrent même pas Python pour leurs projets. Alors à quelle secte de programmeurs veux-tu donner ton âme ?

#### 1.1.10 Les concurrents directs de Python au niveau prédisposition

Python a deux concurrents directs, avec des propriétés et des prédispositions comparables. Il s'agit de :

- **Perl** - un langage de script écrit à l'origine par Larry Wall;
- **Ruby** - un langage de script écrit à l'origine par Yukihiro Matsumoto.

Le premier est plus traditionnel, plus conservateur que Python, et ressemble à certains des bons vieux langages dérivés du langage de programmation C classique.

En revanche, le dernier est plus innovant et plus novateurs que Python. Python lui-même se situe quelque part entre ces deux vagues. Internet regorge de forums avec des discussions infinies sur la supériorité de l'un de ces trois langages sur les autres, si vous souhaitez en savoir plus sur chacun d'eux, libre à vous de voguer sur le flot d'internet.

#### 1.1.11 Ou Python semble peu porteur ?

Malgré la popularité croissante de Python, il existe encore des niches où Python est absent, ou rarement vu :

- **programmation de bas niveau** : si vous voulez implémenter un pilote ou un moteur graphique extrêmement efficace, vous n'utiliserez pas Python;
- **applications pour appareils mobiles** : bien que ce territoire attende toujours d'être conquis par Python, cela arrivera très probablement un jour.

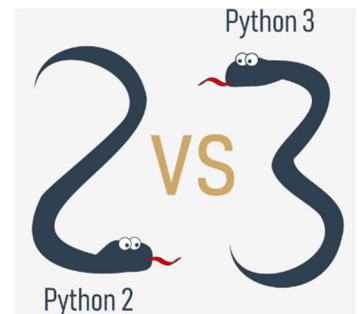
## 1.2 Les différents types de Python

### 1.2.1 Python 2 ou Python 3 ?

Il existe deux principaux types de Python, appelés Python 2 et Python 3. Python 2 est une ancienne version du Python original. Son développement a depuis été intentionnellement bloqué, bien que cela ne signifie pas qu'il n'y ait pas de mises à jour. Au contraire, les mises à jour sont publiées régulièrement, mais elles ne sont pas destinées à modifier le langage de manière significative. Elles corrigent plutôt les bugs et les failles de sécurité fraîchement découvertes. Le chemin de développement de Python 2 a déjà atteint une impasse, mais Python 2 lui-même est toujours bien vivant.

**Python 3 est la version la plus récente (pour être précis, la version actuelle) du langage. Il suit son propre chemin d'évolution, créant ses propres normes et habitudes. C'est bien entendu en Python 3 que nous travaillerons.**

Ces deux versions de Python ne sont pas compatibles entre elles. Les scripts Python 2 ne fonctionneront pas dans un environnement Python 3 et vice versa, donc si vous voulez que l'ancien code Python 2 soit exécuté par un interpréteur Python 3, la seule solution possible est de le réécrire (sauf exception), pas à partir de zéro, bien sûr, car de grandes parties du code peuvent rester intactes, mais vous devez réviser tout le code pour trouver toutes les incompatibilités possibles. Malheureusement, ce processus ne peut pas être entièrement automatisé.



Il est trop difficile, trop long, trop coûteux et trop risqué de migrer une ancienne application Python 2 vers une nouvelle plate-forme. Il est possible que la réécriture du code y introduise de nouveaux bogues. Il est plus facile et plus judicieux de laisser ces systèmes seuls et d'améliorer l'interpréteur existant, au lieu d'essayer de travailler à l'intérieur du code source déjà fonctionnel. C'est un peu comme les banques, elles ne veulent pas investir dans le déploiement d'un nouveau système bancaires alors que l'ancien fonctionne.

Python 3 n'est pas seulement une meilleure version de Python 2 - c'est un langage complètement différent, bien qu'il soit très similaire à son prédécesseur. Lorsque vous les regardez de loin, ils semblent être les mêmes, mais quand vous regardez de près, vous remarquez beaucoup de différences.

Si vous modifiez une ancienne solution Python existante, il est fort probable qu'elle ait été codée en Python 2. C'est la raison pour laquelle Python 2 est toujours utilisé. Il y a trop d'applications Python 2 existantes pour l'éliminer complètement.

Il est important de se rappeler qu'il peut y avoir des différences plus petites ou plus grandes entre les versions ultérieures de Python 3 (par exemple, Python 3.6 a introduit des clés de dictionnaire ordonnées par défaut sous l'implémentation CPython), la bonne nouvelle, est que toutes les nouvelles versions de Python 3 sont **rétrocompatibles** avec les versions précédentes de Python 3. Chaque fois que cela est significatif, nous essaierons de mettre en évidence ces différences.

### 1.2.2 Python aka CPython

En plus de Python 2 et Python 3, il existe plus d'une version de chacun.

Tout d'abord, il y a les Pythons qui sont maintenus par les personnes rassemblées autour de la PSF (Python Software Foundation), une communauté qui vise à développer, améliorer, étendre et populariser Python et son environnement. Le président du PSF est Guido van Rossum lui-même, et pour cette raison, ces pythons sont appelés **canoniques**. Ils sont également considérés comme des **pythons de référence**, car toute autre implémentation du langage doit suivre toutes les normes établies par le PSF.

Guido van Rossum a utilisé le langage de programmation « C » pour implémenter la toute première version de son langage et cette décision est toujours en vigueur. Tous les Pythons provenant de la PSF sont écrits dans le langage « C ». Il y a plusieurs raisons à cette approche et elle a de nombreuses conséquences. L'un d'eux (probablement le plus important) est que grâce à lui, Python peut être facilement porté et migré vers toutes les plates-formes avec la possibilité de compiler et d'exécuter des programmes en langage « C » (presque toutes les plates-formes ont cette fonctionnalité, ce qui ouvre de nombreuses possibilités d'expansion pour Python).

C'est pourquoi l'implémentation PSF est souvent appelée **CPython**. C'est le Python le plus influent parmi tous les Pythons du monde.

### 1.2.3 Cython

Cython est l'une des nombreuses solutions possibles au plus douloureux des traits de Python → **le manque d'efficacité**. Les calculs mathématiques volumineux et complexes peuvent être facilement codés en Python (beaucoup plus facile qu'en « C » ou tout autre langage traditionnel), mais l'exécution du code résultant peut prendre beaucoup de temps.



Comment ces deux contradictions sont-elles conciliées ? Une solution consiste à écrire vos idées mathématiques en utilisant Python, et lorsque vous êtes absolument sûr que votre code est correct et produit des résultats valides, vous pouvez le traduire en « C » qui fonctionnera beaucoup plus vite que Python pur.

C'est ce que Cython est censé faire - traduire automatiquement le code Python (propre et clair, mais pas trop rapide) en code « C » (compliqué et bavard, mais agile).

### 1.2.4 Jython

Une autre version de Python s'appelle **Jython**.

« J » est pour « Java ». Imaginez un Python écrit en Java au lieu de C. Ceci est utile, par exemple, si vous développez des systèmes volumineux et complexes écrits entièrement en Java et que vous souhaitez leur ajouter une certaine flexibilité Python. Le CPython traditionnel peut être difficile à intégrer dans un tel environnement, car C et Java vivent dans des mondes complètement différents et ne partagent pas beaucoup d'idées communes.



Jython peut communiquer plus efficacement avec l'infrastructure Java existante. C'est pourquoi certains projets le trouvent utilisable et nécessaire.

Remarque : l'implémentation actuelle de Jython suit les normes Python 2. Il n'y a pas de Jython conforme à Python 3, jusqu'à présent (début 2023).

### 1.2.5 PyPy et RPython

**PyPy** - un Python dans un Python. En d'autres termes, il représente un environnement Python écrit dans un langage de type Python nommé **RPython** (Restricted Python). C'est en fait un sous-ensemble de Python.



Le code source de PyPy n'est pas exécuté de manière interprétative, mais est traduit dans le langage de programmation C et ensuite exécuté séparément.

Ceci est utile car si vous voulez tester une nouvelle fonctionnalité qui peut être (mais n'a pas à être) introduite dans l'implémentation Python grand public, il est plus facile de la vérifier avec PyPy qu'avec CPython. C'est pourquoi PyPy est plutôt un outil pour les personnes développant Python que pour le reste des utilisateurs.

Cela ne rend pas PyPy moins important ou moins sérieux que CPython, bien sûr. De plus, PyPy est compatible avec le langage Python 3.

Il y a encore beaucoup plus de sortes de pythons différents dans le monde. Vous les trouverez si vous regardez, mais **ce cours se concentrera sur le CPython**.



### 1.3 Comment obtenir Python et l'utiliser ?

#### 1.3.1 Obtenir Python

Il existe plusieurs façons d'obtenir votre propre copie de Python 3, selon le système d'exploitation que vous utilisez.

**Les utilisateurs de Linux ont probablement déjà installé Python** - c'est le scénario le plus probable, car l'infrastructure de Python est utilisée de manière intensive par de nombreux composants du système d'exploitation Linux.

Par exemple, certains distributeurs peuvent coupler leurs outils spécifiques avec le système et beaucoup de ces outils, comme les gestionnaires de paquets, sont souvent écrits en Python. Certaines parties des environnements graphiques disponibles dans le monde Linux peuvent également utiliser Python.

Si vous êtes un utilisateur Linux, ouvrez-le terminal/la console et tapez :  
**python3**

à l'invite du shell, appuyez sur *Entrée* et patientez.

Si vous voyez quelque chose comme ceci :

```
Python 3.4.5 (par défaut, Jan 12 2017, 02:28:40)
[GCC 4.2.1 Compatible Clang 3.7.1 (tags/RELEASE_371/final)] sur
linux
Tapez « aide », « copyright », « crédits » ou « licence » pour plus
d'informations.
>>>
```

alors vous n'avez rien d'autre à faire.

Si Python 3 est absent, reportez-vous à votre documentation Linux afin de savoir comment utiliser votre gestionnaire de paquets pour télécharger et installer un nouveau paquet - celui dont vous avez besoin s'appelle **python3** ou son nom commence par cela.

Tous les utilisateurs non-Linux peuvent télécharger une copie à  
<https://www.python.org/downloads/>.

Étant donné que le navigateur indique au site que vous avez entré le système d'exploitation que vous utilisez, la seule étape que vous devez effectuer est de cliquer sur la version appropriée de Python souhaitée.

Dans ce cas, sélectionnez Python 3. Le site vous en propose toujours la dernière version.

Si vous êtes un **utilisateur Windows**, démarrez le fichier .exe téléchargé et suivez toutes les étapes.

Laissez les paramètres par défaut suggérés par le programme d'installation pour l'instant, à une exception près - regardez la case à cocher nommée **Ajouter Python 3.x à PATH** et vérifiez-la.



Si vous êtes un **utilisateur macOS**, une version de Python 2 a peut-être déjà été préinstallée sur votre ordinateur, mais comme nous travaillerons avec Python 3, vous devrez toujours télécharger et installer le fichier .pkg approprié à partir du site Python.

Dans notre formation, nous utiliserons le logiciel Pycharm. Nous vous conseillons d'installer Pycharm Education pour avoir une version aboutie lors de votre cursus : <https://www.jetbrains.com/pycharm-edu/>

### 1.3.2 Testons l'installations de Python

Maintenant que Python 3 est installé, il est temps de vérifier si cela fonctionne et d'en faire la toute première utilisation.

Ce sera une procédure très simple, mais elle devrait suffire à vous convaincre que l'environnement Python est complet et fonctionnel.

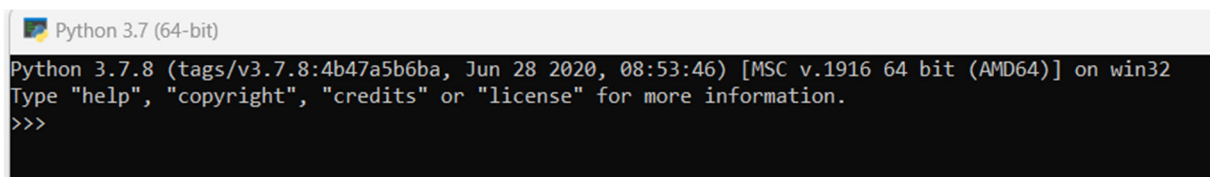
Il existe de nombreuses façons d'utiliser Python, surtout si vous décidez d'être un développeur Python.

Pour commencer votre travail, vous avez besoin des outils suivants :

- un éditeur qui vous aidera à écrire le code (il devrait avoir quelques fonctionnalités spéciales, non disponibles dans des outils simples); cet éditeur dédié vous donnera plus que l'équipement standard du système d'exploitation;
- une **console** dans laquelle vous pouvez lancer votre code nouvellement écrit et l'arrêter de force lorsqu'il devient incontrôlable;
- Un outil nommé **débogueur**, capable de lancer votre code étape par étape et vous permettant de l'inspecter à chaque moment d'exécution.

Outre ses nombreux composants utiles, l'installation standard de Python 3 contient une application très simple mais extrêmement utile nommée IDLE. **IDLE** est un acronyme : Integrated Development and Learning Environment.

Naviguez dans les menus de votre système d'exploitation, trouvez IDLE quelque part sous Python 3.x et lancez-le. Voici ce que vous devriez voir:



```
Python 3.7 (64-bit)
Python 3.7.8 (tags/v3.7.8:4b47a5b6ba, Jun 28 2020, 08:53:46) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Vous pouvez aussi directement lancer Pycharm

### 1.3.3 Règle importante

Ne définissez aucune extension pour le nom de fichier que vous allez utiliser. Python a besoin que ses fichiers aient l'extension `.py`, vous devez donc vous fier aux valeurs par défaut de la fenêtre de dialogue de votre IDLE. L'utilisation de l'extension `.py` standard permet au système d'exploitation d'ouvrir correctement ces fichiers.

### 1.3.4 Ecrire son premier programme


Dans votre fichier `.py`, encodez la ligne suivante :

```
print("ce cours est trop coooooo111")
```

Attention ! Les 6 « o » et « 3 « l » sont capitaux !

Nous n'allons pas expliquer la signification du programme pour le moment.

Regardez de plus près les guillemets. Ce sont les formes les plus simples de guillemets (neutre, droit, muet, etc.) couramment utilisées dans les fichiers sources. N'essayez pas d'utiliser des guillemets typographiques (courbes, bouclés, intelligents, etc.), utilisés par les processeurs de texte avancés, car Python ne les accepte pas.

Si vous lancez le programme avec le bouton run  ou en faisant Run→Run 'main' , votre script se lance et affiche le texte en vert.

## 2. Types de données, variables, opérations entrées-sorties, opérateurs de base

### 2.1 Premier programme

#### 2.1.1 Hello World

Il est temps de commencer à écrire du **vrai code Python** fonctionnel. Ce sera très simple pour le moment.

Comme nous allons vous montrer quelques concepts et termes fondamentaux, ces extraits de code ne seront ni sérieux ni complexes.

Prenons le code suivant :

```
print("Hello, World!")
```

Vous pouvez lancer IDLE, créer un nouveau fichier source Python, le remplir avec ce code, nommer le fichier et l'enregistrer. Maintenant, exécutez-le. Si tout se passe bien, vous verrez la ligne dans la fenêtre de la console IDLE. Le code que vous avez exécuté doit vous sembler familier. Vous avez vu quelque chose de très similaire lorsque nous vous avons guidé dans la configuration de l'environnement IDLE.

Maintenant, nous allons passer un peu de temps à vous montrer et à vous expliquer ce que vous voyez réellement, et pourquoi cela ressemble à ceci.

Comme vous pouvez le voir, le premier programme comprend les parties suivantes:

- Le mot clé **print**;
- une parenthèse ouvrante;
- un guillemet;
- une ligne de texte : **Hello, World!** ;
- un autre guillemet;
- une parenthèse fermante.

Chacun des éléments ci-dessus joue un rôle très important dans le code. Voyons-les.

### 2.1.2 La fonction print()

Regardez la ligne de code ci-dessous :

```
print("Hello, World!")
```

Le mot **print** que vous pouvez voir ici est un nom de **fonction**. Cela ne signifie pas que partout où le mot apparaît, il s'agit toujours d'un nom de fonction. Le sens du mot vient du contexte dans lequel le mot a été utilisé.

Vous avez probablement rencontré le terme fonction plusieurs fois auparavant, pendant les cours de mathématiques. Vous pouvez probablement aussi énumérer plusieurs noms de fonctions mathématiques, comme sinus ou log.

Les fonctions Python, cependant, sont plus flexibles et peuvent contenir plus de contenu que leurs frères et sœurs mathématiques.

Une fonction (dans ce contexte) est une partie distincte du code informatique capable de :

- **provoquer un certain effet** (par exemple, envoyer du texte au terminal, créer un fichier, dessiner une image, jouer un son, etc.); c'est quelque chose de complètement inconnu dans le monde des mathématiques;
- **évaluer une valeur** (par exemple, la racine carrée d'une valeur ou la longueur d'un texte donné) et la **renvoyer comme résultat de la fonction**; c'est ce qui fait des fonctions Python les parents des concepts mathématiques.
- De plus, de nombreuses fonctions Python peuvent faire les deux choses ci-dessus ensemble.

D'où viennent les fonctions ?

- Elles peuvent **provenir de Python lui-même**; la fonction d'impression est de ce type; une telle fonction est une valeur ajoutée reçue avec Python et son environnement (elle est **intégrée**); vous n'avez rien de spécial à faire si vous voulez l'utiliser;
- Elles peuvent provenir d'un ou plusieurs **modules** complémentaires nommés de Python; certains des modules sont livrés avec Python, d'autres peuvent nécessiter une installation séparée - quoi qu'il en soit, ils doivent tous être explicitement connectés à votre code (nous vous montrerons comment le faire bientôt);
- Vous pouvez **les écrire vous-même**, en plaçant autant de fonctions que vous le souhaitez et dont vous en avez besoin dans votre programme pour le rendre plus simple, plus clair et plus élégant.

Le nom de la fonction doit être **significatif** (le nom de la fonction print est évident).

Bien sûr, si vous utilisez une fonction déjà existante, vous n'avez aucune influence sur son nom, mais lorsque vous commencerez à écrire vos propres fonctions, vous devrez réfléchir attentivement à votre choix de noms.

Comme nous l'avons dit précédemment, une fonction peut avoir :

- un **effet**;
- un **résultat**.

Il y a aussi un troisième composant de fonction, très important : le(s) **argument(s)**.

Les fonctions mathématiques prennent généralement un argument, par exemple,  $\sin(x)$  prend un  $x$ , qui est la mesure d'un angle.

Les fonctions Python, en revanche, sont plus polyvalentes. Selon les besoins individuels, ils peuvent accepter n'importe quel nombre d'arguments → autant que nécessaire pour effectuer leurs tâches.

Remarque: tout nombre inclut zéro → Certaines fonctions Python n'ont besoin d'aucun argument.

Malgré le nombre d'arguments nécessaires/fournis, les fonctions Python exigent fortement la présence **d'une paire de parenthèses** - respectivement ouvertes et fermantes.

```
print("Hello, World!")
```

Si vous souhaitez fournir un ou plusieurs arguments à une fonction, placez-les **entre parenthèses**. Si vous utilisez une fonction qui ne prend aucun argument, vous devez toujours avoir les parenthèses, elles seront juste vides.

Remarque : pour distinguer les mots ordinaires des noms de fonction, placez **une paire de parenthèses vides** après leur nom, même si la fonction correspondante veut un ou plusieurs arguments. Il s'agit d'une convention standard.

La fonction dont nous parlons ici est `print()`.

La fonction `print()` de notre exemple a-t-elle des arguments ?

Bien sûr que oui, mais quels sont-ils?

Le seul argument fourni à la fonction `print()` dans cet exemple est une **chaîne de caractères** :

```
print("Hello, World!")
```

Comme vous pouvez le voir, la chaîne **est délimitée par des** guillemets → en fait, les guillemets font la chaîne de caractères, ils découpent une partie du code et lui attribuent une signification différente.

Vous pouvez imaginer que les guillemets disent quelque chose comme: le texte entre nous n'est pas du code. Il n'est pas destiné à être exécuté, et vous devriez le prendre tel quel.

Presque tout ce que vous mettez entre guillemets sera pris littéralement, non pas comme code, mais comme des **données**. Essayez de jouer avec cette chaîne particulière et modifiez-la, entrez du nouveau contenu, supprimez une partie du contenu existant, ...

Nous verrons plus tard qu'il y a plus d'une façon de spécifier une chaîne dans le code de Python, mais pour l'instant, cependant, celle-ci est suffisante.

Sans le savoir vous venez d'apprendre un type de données : les chaînes de caractères (ou String) mais on en reparle plus tard !

Ici nous ne parlions que de syntaxe mais qu'en est-il de la sémantique ?

### 2.1.3 La sémantique de print()

Le nom de la fonction (**en caractères d'imprimerie** dans ce cas) ainsi que les parenthèses et les arguments forment l'appel de la **fonction**.

```
print("Bonjour, Monde!")
```

Que se passe-t-il lorsque Python rencontre un appel comme celui-ci ci-dessous ?

```
function_name(argument)
```

Détaillons cela :

- Tout d'abord, Python vérifie si le nom spécifié est **légal** (il parcourt ses données internes afin de trouver une fonction existante avec ce nom; si cette recherche échoue, Python abandonne le code);
- deuxièmement, Python vérifie si les exigences de la fonction pour le nombre d'arguments **vous permettent d'invoquer** la fonction de cette manière (par exemple, si une fonction spécifique exige exactement deux arguments, tout appel ne fournissant qu'un seul argument sera considéré comme erroné et interrompra l'exécution du code);
- troisièmement, Python **quitte votre code pendant un moment** et saute dans la fonction que vous souhaitez invoquer; Bien sûr, il prend aussi vos arguments et les transmet à la fonction;
- quatrièmement, la fonction **exécute son code**, provoque l'effet désiré (le cas échéant), évalue-le(s) résultat(s) souhaité(s) et termine sa tâche;
- enfin, Python **retourne à votre code** (à l'endroit juste après l'appel) et reprend son exécution.

## Temps estimé

5-10 minutes

## Niveau de difficulté

Très facile

## Objectifs

- se familiariser avec la fonction `print()` et ses capacités de formatage;
- expérimenter avec du code Python.

## Scénario

La commande `print()`, qui est l'une des directives les plus simples en Python, imprime simplement une ligne à l'écran.

Dans votre premier laboratoire :

- utilisez la fonction `print()` pour imprimer la ligne Hello, mon ,nouvel ami! à l'écran. Utilisez des guillemets doubles autour de la chaîne;
- Cela fait, utilisez à nouveau la fonction `print()`, mais cette fois imprimez votre prénom;
- Supprimez les guillemets doubles et exécutez votre code. Regardez la réaction de Python. Quel genre d'erreur est généré?
- Ensuite, supprimez les parenthèses, remettez les guillemets doubles et exécutez à nouveau votre code. Quel genre d'erreur est généré cette fois-ci?
- Expérimentez autant que vous le pouvez. Remplacez les guillemets doubles par des guillemets simples, utilisez plusieurs fonctions `print()` sur la même ligne, puis sur différentes lignes. Voyez ce qui se passe.

#### 2.1.4 Mais à quoi sert ce print ?

Trois questions importantes doivent trouver réponses à vos yeux dès que possible(et cela pour toutes les fonctions que vous utiliserez) :

##### 1. Quel est l'effet de la fonction **print()** ?

L'effet est très **utile et très spectaculaire**.

La fonction :

- prend ses arguments (il peut accepter plus d'un argument et peut également accepter moins d'un argument)
- les convertit en forme lisible par l'homme si nécessaire (comme vous pouvez vous en douter, les chaînes ne nécessitent pas cette action, car la chaîne est déjà lisible)
- **Envoie les données résultantes au périphérique de sortie** (généralement la console); en d'autres termes, tout ce que vous mettez dans la fonction print() apparaîtra sur votre écran.

Il n'est donc pas étonnant qu'à partir de maintenant, vous utilisiez **print()** de manière très intensive pour voir les résultats de vos opérations et évaluations.

##### 2. Quels arguments print() attend-il?

Quelconque. Nous vous montrerons bientôt que print() est capable de fonctionner avec pratiquement tous les types de données offertes par Python. Chaînes, nombres, caractères, valeurs logiques, objets, ... n'importe lequel d'entre eux peut être passé avec succès à print().

##### 3. Quelle valeur la fonction print() renvoie-t-elle?

Aucun. Son effet(afficher) suffit.



### 2.1.5 Les instructions

Nous avons donc un programme informatique qui contient un appel de fonction. Ce qui est l'un des nombreux types possibles **d'instructions** Python.

Bien sûr, tout programme complexe contient généralement beaucoup plus d'instructions. La question est : comment coupler plus d'une instruction dans le code Python ?

La syntaxe de Python est assez spécifique dans ce domaine. Contrairement à la plupart des langages de programmation, Python exige **qu'il ne puisse y avoir plus d'une instruction dans une ligne**.

Une ligne peut être vide (c'est-à-dire qu'elle ne peut contenir aucune instruction), mais elle ne doit pas contenir deux, trois instructions ou plus. Ceci est strictement interdit.

Remarque : Python fait une exception à cette règle : il permet à une instruction de s'étaler sur plus d'une ligne (ce qui peut être utile lorsque votre code contient des constructions complexes).

Développons un peu. Exécutez le code suivant et notez ce que vous voyez dans la console.

```
print("Les étudiants parlent de plus en plus fort .")  
print("Le professeur se doit de crier encore plus fort pour les calmer.")
```

Votre console Python devrait maintenant ressembler à ceci :

```
Les étudiants parlent de plus en plus fort.  
Le professeur se doit de crier encore plus fort pour les calmer.
```

C'est une bonne occasion de faire quelques observations :

- Le programme **appelle** la **fonction** `print()` deux fois, et vous pouvez voir deux lignes distinctes dans la console - cela signifie que `print()` commence sa sortie d'affichage à partir d'une nouvelle ligne chaque fois qu'il commence son exécution; vous pouvez modifier ce comportement, mais vous pouvez également l'utiliser à votre avantage;
- Chaque invocation `print()` contient une chaîne différente, car son argument et le contenu de la console le reflètent - cela signifie que **les instructions du code sont exécutées dans le même ordre** dans lequel elles ont été placées dans le fichier source ; aucune instruction suivante n'est exécutée tant que la précédente n'est pas terminée (il y a quelques exceptions à cette règle, mais vous pouvez les ignorer pour l'instant)

Changeons un peu l'exemple, nous allons ajouter un appel de fonction `print()` vide. Nous l'appelons vide parce que nous n'avons fourni aucun argument à la fonction. Plaçons-le entre les deux autres lignes.

Que se passe-t-il ?

Si tout se passe bien, vous devriez voir quelque chose comme ceci:

Les étudiants parlent de plus en plus fort.

Le professeur se doit de crier encore plus fort pour les calmer.

Comme vous pouvez le voir, l'appel `print()` vide n'est pas aussi vide que vous auriez pu vous y attendre, il génère une ligne vide, ou (cette interprétation est également correcte) sa sortie est juste une nouvelle ligne.

Ce n'est pas la seule façon de produire une **nouvelle ligne** dans la console de sortie.

#### 2.1.6 Les caractères d'échappement et les nouvelles lignes

Modifions à nouveau le code pour obtenir :

```
print("Les étudiants parlent \nde plus en plus fort")
print()
print("Le professeur se doit de crier\n encore plus fort pour les calmer.")
```

Il y a deux changements, nous avons inséré une étrange paire de caractères. Ils ressemblent à ceci : `\n`.

Il est intéressant de constater que, alors que **vous pouvez voir deux caractères, Python n'en voit qu'un**.

La barre oblique inverse (`\`) a une signification très particulière lorsqu'elle est utilisée à l'intérieur de chaînes, c'est ce qu'on appelle **le caractère d'échappement**.

Le mot *échappement* doit être compris spécifiquement, il signifie que la série de caractères dans la chaîne s'échappe pour le moment (un moment très court) pour introduire une inclusion spéciale.

En d'autres termes, la barre oblique inverse ne signifie rien en soi, mais n'est qu'une sorte d'annonce, que le caractère suivant après la barre oblique inverse a également une signification différente.

La lettre n placée après la barre oblique inverse provient du mot *newline*.

La barre oblique inverse et le n forment un symbole spécial appelé **caractère de nouvelle ligne**, qui incite la console à démarrer une **nouvelle ligne de sortie**.

Comme vous pouvez le voir en exécutant le programme, deux nouvelles lignes apparaissent dans votre console, aux endroits où les `\n` ont été utilisés. Cette convention sur la barre oblique, a deux conséquences importantes :

1. Si vous voulez mettre une seule barre oblique inverse à l'intérieur d'une chaîne, n'oubliez pas sa nature d'échappement, vous devrez la doubler !!

par exemple, un tel appel provoquera une erreur:  
`print(« \ »)`

alors que celui-ci ne le fera pas:  
`print(« \\ »)`

2. Toutes les paires d'échappement (la barre oblique inverse couplée à un autre caractère) ne signifient pas quelque chose.

### 2.1.7 Ajoutons des arguments

Jusqu'à présent, nous avons testé le comportement de la fonction `print()` sans argument ou avec un seul argument. Il vaut également la peine d'essayer d'alimenter la fonction `print()` avec plusieurs arguments.

Regardez le code suivant :

```
print("Les étudiants" , "parlent" , "trop fort.")
```

Il existe un appel de fonction `print()`, mais il contient **trois arguments**. Tous sont des chaînes de caractères.

Les arguments sont **séparés par des virgules**. Nous les avons entourés d'espaces pour les rendre plus visibles, mais ce n'est pas vraiment nécessaire, et nous ne le ferons plus à l'avenir.

Dans ce cas, les virgules séparant les arguments jouent un rôle complètement différent de la virgule à l'intérieur de la chaîne de caractères. Le premier fait partie de la syntaxe de Python, le second est destiné à être affiché dans la console.

Si vous regardez à nouveau le code, vous verrez qu'il n'y a pas d'espaces à l'intérieur des chaînes (entre les derniers mots et les guillemets par exemple).

Exécutez le code et voyez ce qui se passe.  
La console doit maintenant afficher le texte suivant :

```
Les étudiants parlent trop fort.
```

Les espaces, supprimés des chaînes, sont réapparus. Pouvez-vous expliquer pourquoi?

Deux conclusions se dégagent de cet exemple :

- une fonction `print()` appelée avec plus d'un argument **les affiche tous sur une seule ligne** ;
- La fonction `print()` **place un espace entre les arguments sortis** de sa propre initiative.

Python fonctionne avec ce que l'on appelle la manière positionnelle, c'est-à-dire, que l'argument sortira à la place où il a été mis, le premier, suivi du deuxième, ...

### 2.1.8 Les mots-clés de la fonction print()

Python offre un autre mécanisme pour le passage d'arguments, ce qui peut être utile lorsque vous voulez spécifier à la fonction print() de changer un peu son comportement.

Nous n'allons pas l'expliquer en profondeur pour le moment. Nous prévoyons de le faire lorsque nous parlons des fonctions. Pour l'instant, nous voulons simplement vous montrer comment cela fonctionne. N'hésitez pas à l'utiliser dans vos propres programmes. Et à le tester dans tous les sens pour voir faire la main.

Le mécanisme est appelé « **Keywords arguments** » (**arguments de mot-clé**). Le nom provient du fait que la signification de ces arguments n'est pas tirée de son emplacement (position) mais du mot spécial (mot-clé) utilisé pour les identifier.

La fonction print() possède deux arguments de mot-clé que vous pouvez utiliser pour vos besoins.

Le premier d'entre eux est nommé **end**.

Dans l'éditeur, vous pouvez voir encoder l'exemple très simple d'utilisation d'un argument de mot-clé fourni ci-dessous.

```
print("My name is", "Python.", end=" ")
print("Monty Python.")
```

Pour l'utiliser, il est nécessaire de connaître quelques règles:

- Un argument de mot-clé se compose de trois éléments : un **mot-clé** identifiant l'argument (**end** dans ce cas) ; un **signe égal** (=) ; et une **valeur** affectée à cet argument ;
- Tous les arguments de mot-clé doivent être placés **après le dernier argument positionnel** (c'est **très très** important)

Dans notre exemple, nous avons utilisé l'argument de mot-clé end et l'avons défini sur une chaîne contenant un espace.

Exécutez le code pour voir comment il fonctionne.

La console doit maintenant afficher le texte suivant :

```
Je m'appelle Python. Monty Python.
```

Comme vous pouvez le voir, l'argument mot-clé end détermine les caractères que la fonction print() envoie à la sortie une fois qu'elle atteint la fin de ses arguments positionnels.

Le comportement par défaut reflète la situation dans laquelle l'argument de mot-clé end est **implicitement** utilisé de la manière suivante : **end="\n"**  
Le mot clé end peut aussi être utilisé pour annuler le caractère de saut de ligne à la fin d'un print.

Par exemple le code :

```
print("My name is ", end="")  
print("Monty Python.")
```

Nous afficherons dans la console, la phrase : My name is Monty Python.  
En effet, comme l'argument end a été défini sur rien, la fonction print() ne produit rien non plus, une fois que ses arguments positionnels ont été épuisés.

Un autre mot clé, est le mot clé **sep** qui va nous permettre de séparer les arguments de sorties autrement que par des espaces.

```
print("My", "name", "is", "Monty", "Python.", sep="-")
```

Ce code nous donnera : Mon-nom-est-Monty-Python.

Il est évident qu'il est possible de mélanger dans un même print() ces deux mots-clés.

```
print("My", "name", "is", sep="_", end="*")  
print("Monty", "Python.", sep="*", end="*\n")
```

Si nous regardons cet exemple, qui nous sommes d'accord n'a aucun sens logique, nous pouvons voir la combinaison de nos deux mots-clés et le résultat de ceux-ci en console : My\_name\_is\*Monty\*Python.\*

## 2.1.9 Exercices

## 1. Temps estimé

5-10 minutes

## Objectifs

Se familiariser avec la fonction `print()` et ses capacités de formatage; expérimenter avec du code Python.

## Scénario

Créer un code dans l'éditeur, à l'aide des mots-clés `sep` et `end`, pour qu'il corresponde à la sortie attendue. Utilisez deux fonctions `print()`.

Ne changez rien dans le deuxième appel `print()`.

Bout de code de base :

```
print("Les Profs", "sont", "génial")  
print("surtout Mr Mandoux")
```

## Résultats escomptés

Les Profs\*\*\*sont\*\*\*géniaux... surtout Mr Mandoux

## 2. Temps estimé

5-15 minutes

## Objectifs

Expérimenter avec du code Python existant, la découverte et la correction d'erreurs de syntaxe de base;

Se familiariser avec la fonction `print()` et ses capacités de formatage.

## Scénario

Nous vous encourageons fortement à jouer avec le code que nous avons écrit pour vous (faites en ce que vous souhaitez dans un premier temps pour tester votre compréhension) et à y apporter quelques modifications (peut-être même destructrices).

N'hésitez pas à modifier n'importe quelle partie du code, mais il y a une condition: apprendre de vos erreurs et tirer vos propres conclusions.

Essayez de :

- Réduire le nombre d'appels de la fonction `print()` en insérant la séquence `\n` dans les chaînes
- Rendre la flèche deux fois plus grande (mais garder les proportions)
- Dupliquer la flèche, en plaçant les deux flèches côte à côte; Remarque: une chaîne peut être multipliée en utilisant l'astuce suivante: `"string" * 2` produira « stringstring » (nous vous en dirons plus bientôt)
- Supprimez toutes les guillemets et examinez attentivement la réponse de Python; faites attention à l'endroit où Python voit une erreur - est-ce l'endroit où l'erreur existe vraiment?
- Faites de même avec certaines parenthèses;
- Changer le mot `print` en quelque chose d'autre, ne différant que dans le cas (par exemple, `Print`) - que se passe-t-il maintenant?
- Remplacer certains guillemets par des apostrophes; Observez attentivement ce qui se passe.

## 2.1.10 Résumé

1. La fonction **print()** est une fonction **intégrée**. Elle imprime / envoie un message spécifié à la fenêtre d'écran / console.
2. Les fonctions intégrées, contrairement aux fonctions définies par l'utilisateur, sont toujours disponibles et n'ont pas besoin d'être importées. Python 3.8 est livré avec 69 fonctions intégrées. Vous pouvez trouver leur liste complète fournie par ordre alphabétique sur <https://docs.python.org/3/library/functions.html>.
3. Pour appeler une fonction (ce processus est connu sous le nom **d'appel** de fonction ou d'invocation de fonction), vous devez utiliser le nom de la fonction suivi de parenthèses. Vous pouvez passer des arguments dans une fonction en les plaçant entre parenthèses. Vous devez séparer les arguments par une virgule, par exemple, `print(« Hello, », « world! »)`. Une fonction `print()` « vide » affiche une ligne vide à l'écran.
4. Les chaînes Python sont délimitées par des guillemets, par exemple, « Je suis une chaîne » (guillemets doubles) ou 'Je suis aussi une chaîne' (guillemets simples).
5. Les programmes d'ordinateurs sont des collections **d'instructions**. Une instruction est une commande pour effectuer une tâche spécifique lorsqu'elle est exécutée, par exemple, pour imprimer un certain message à l'écran.
6. Dans les chaînes Python, la **barre oblique inverse** (\) est un caractère spécial qui annonce que le caractère suivant a une signification différente, par exemple, `\n` (**le caractère de nouvelle ligne**) commence une nouvelle ligne de sortie.
7. **Les arguments positionnels** sont ceux dont le sens est dicté par leur position, par exemple, le deuxième argument est produit après le premier, le troisième est produit après le second, etc.
8. Les **arguments de mots-clés** sont ceux dont la signification n'est pas dictée par leur emplacement, mais par un mot spécial (mot-clé) utilisé pour les identifier.
9. Les paramètres **end** et **sep** peuvent être utilisés pour formater la sortie de la fonction `print()`. Le paramètre `sep` spécifie le séparateur entre les arguments générés (par exemple, `print("H", "E", "L", "L", "O", sep= "-")`), tandis que le paramètre `end` spécifie ce qu'il faut imprimer à la fin de l'instruction `print`.



## 2.2. Les littéraux

### 2.2.1 Introduction

Maintenant que vous avez un peu de connaissance de certaines des fonctionnalités de la fonction `print()`, il est temps d'en apprendre davantage sur de nouveaux problèmes et un nouveau terme important : les **littéraux**.

**Un littéral est une donnée dont les valeurs sont déterminées par le littéral lui-même.**

Comme il s'agit d'un concept difficile à comprendre, un bon exemple peut être utile.

Jetez un coup d'œil à l'ensemble de chiffres suivant :

123

Pouvez-vous deviner quelle valeur il représente? Normalement c'est une évidence et vous me répondrez tous : *cent vingt-trois*.

Mais qu'en est-il de ceci:

c

Ce symbole représente-t-il une valeur? Peut-être. Il est aussi peut être le symbole de la vitesse de la lumière, ou encore cela peut aussi être la constante de l'intégration, ou même la longueur d'une hypoténuse dans le théorème de Pythagore. Les possibilités sont nombreuses.

Vous ne pouvez pas choisir le bon sens sans quelques connaissances supplémentaires.

Et voici un indice : 123 est un littéral, et c ne l'est pas.

Vous utiliserez les littéraux **pour coder des données et les placer dans votre code**. Nous allons maintenant vous montrer quelques conventions auxquelles vous devez obéir lorsque vous utilisez Python.

### 2.2.2 Faisons une expérience

Commençons par une expérience simple :

```
print("2")  
print(2)
```

La première ligne semble familière alors que la seconde semble erronée en raison de l'absence visible de guillemets.

Essayez de l'exécuter.

Si tout s'est bien passé, vous devriez maintenant voir deux lignes identiques. Que s'est-il passé?? Qu'est-ce que cela signifie?

Dans cet exemple, vous rencontrez en fait deux types de littéraux différents :

- une **chaîne**, que vous connaissez déjà,
- et un nombre **entier**, quelque chose de complètement nouveau.

La fonction `print()` les présente exactement de la même manière. Cet exemple est évident, car leur représentation est facilement lisible par l'homme. Un 2 reste 2 à nos yeux.

Dans votre ordinateur, dans la mémoire, ces deux valeurs sont stockées de manière complètement différente, la chaîne existe sous la forme d'une simple chaîne (une série de lettres) alors que le nombre est converti en représentation machine (un ensemble de bits).

La fonction `print()` est capable de les montrer les deux éléments sous une forme lisible par les humains.

Nous allons maintenant passer du temps à discuter des littéraux numériques et de leur vie interne à la machine.

### 2.2.3 Les entiers (integers)

Vous en savez peut-être déjà un peu sur la façon dont les ordinateurs effectuent des calculs sur des nombres grâce aux cours de fonctionnement des systèmes 1 ou au cours de théorie.

Vous avez sûrement déjà entendu parler du système **binaire** et savez que c'est le système que les ordinateurs utilisent pour stocker des nombres et qu'ils peuvent effectuer n'importe quelle opération sur ceux-ci.

Nous n'explorerons pas ici les subtilités des systèmes de numération positionnelle, mais nous dirons que les nombres gérés par les ordinateurs modernes sont de deux types:

- **les entiers**, c'est-à-dire ceux qui sont dépourvus de la partie fractionnaire
- et les nombres **à virgule flottante** (ou simplement **flottants**), qui contiennent (ou sont capables de contenir) la partie fractionnaire.

Cette définition n'est pas tout à fait exacte, nous en convenons, mais tout à fait suffisante pour l'instant.

La distinction est très importante et la frontière entre ces deux types de nombres est très stricte. Ces deux types de nombres diffèrent considérablement dans la façon dont ils sont stockés dans la mémoire d'un ordinateur et dans la plage de valeurs acceptables.

La caractéristique de la valeur numérique qui détermine son type, sa plage et son application est **appelée type**.

Si vous encodez un littéral et le placez dans du code Python, la forme du littéral détermine la représentation (type) que Python utilisera pour **le stocker dans la mémoire**.

Pour l'instant, laissons de côté les nombres à virgule flottante (nous y reviendrons dans le point suivant) et considérons la question de savoir comment Python reconnaît les entiers.

Le processus est presque comme vous les écririez avec un crayon sur papier dans d'autres de vos cours. C'est simplement une chaîne de chiffres qui composent le nombre.  
Attention que Python n'accepte aucun caractère non numérique dans un entier.

Prenons, par exemple, le nombre onze millions cent onze mille cent onze. Si vous preniez un crayon dans votre main maintenant, vous écririez le nombre comme ceci: 11,111,111, ou comme ceci: 11.111.111, ou même comme ceci: 11 111 111.

Il est clair que cette disposition facilite la lecture, surtout lorsque le numéro se compose de nombreux chiffres.  
Cependant, Python n'accepte pas de telles choses. C'est **interdit**. Ce que Python permet, cependant, c'est l'utilisation de **traits de soulignement** dans les littéraux numériques.<sup>1</sup>

Par conséquent, vous pouvez écrire ce nombre comme ceci: 11111111, ou comme ceci: 11\_111\_111.

Et comment code-t-on les nombres négatifs en Python ? Comme d'habitude, en ajoutant un **moins**.  
Vous pouvez écrire : -11111111, ou -11\_111\_111.

Les nombres positifs n'ont pas besoin d'être précédés du signe **plus**, mais c'est permis, si vous souhaitez le faire. Les lignes suivantes décrivent le même nombre : +11111111 et 11111111.

Il existe deux conventions supplémentaires en Python qui sont inconnues du monde des mathématiques. La première nous permet d'utiliser des nombres dans une représentation **octale**.  
Si un nombre entier est précédé d'un préfixe 0O ou 0o (zéro-o), il sera traité comme une valeur octale. Cela signifie que le nombre doit contenir uniquement des chiffres tirés de la plage [0..7].

0o123 est un nombre **octal** avec une valeur (décimale) égale à 83.

La fonction print() effectue la conversion automatiquement. Essayez ceci :

```
print(0o123)
```

---

<sup>1</sup> Python 3.6 a introduit des traits de soulignement dans les littéraux numériques, permettant de placer des traits de soulignement uniques entre les chiffres et après les spécificateurs de base pour une meilleure lisibilité. Cette fonctionnalité n'est pas disponible dans les anciennes versions de Python.

La deuxième convention nous permet d'utiliser des **nombres hexadécimaux**. Ces nombres doivent être précédés du préfixe 0x ou 0X (zéro-x).

0x123 est un nombre **hexadécimal** avec une valeur (décimale) égale à 291. La fonction print() peut également gérer ces valeurs. Essayez ceci :

```
print(0x123)
```

### 2.2.3 Les Réels ou nombres flottants (float)

Il est maintenant temps de parler d'un autre type, conçu pour représenter et stocker les nombres qui (comme dirait un mathématicien) ont une **fraction décimale non vide**.

Ce sont les nombres qui ont (ou peuvent avoir) une partie fractionnaire après la virgule, et bien qu'une telle définition soit très pauvre, elle est suffisante pour notre besoin.

Chaque fois que nous utilisons un terme comme « *deux et demi* ou *moins zéro point quatre* », nous pensons à des nombres que l'ordinateur considère comme des nombres à **virgule flottante**:

2.5  
-0.4

Remarque: *deux et demi* semble normal lorsque vous l'écrivez dans un programme, bien que si dans notre langue maternelle, on préfère utiliser une virgule au lieu d'un point. Attention qu'en programmation, vous devez vous assurer que votre **nombre ne contiennent aucune virgule**.

Python ne l'acceptera pas, ou (dans de très rares cas) peut mal comprendre vos intentions, car la virgule elle-même a sa propre signification en Python.

Si vous souhaitez utiliser seulement une valeur de deux et demi, vous devez l'écrire comme indiqué ci-dessus.

Comme vous pouvez l'imaginer, la valeur du nombre : **zéro point quatre** pourrait être écrite en Python comme: 0.4

Mais attention, n'oubliez pas cette règle simple: vous pouvez omettre zéro lorsqu'il s'agit du seul chiffre devant ou après la virgule.

Vous pouvez donc écrire la valeur 0.4 : .4

Autre exemple, la valeur de 4,0 pourrait s'écrire comme suit : 4.

Cela ne changera ni son type ni sa valeur. Si on écrit juste 4, on écrit un entier par un réel, voyons cela en détail !

### 2.2.4 La guerre des entiers et des réels

La virgule décimale (le point en programmation) est essentiellement importante pour reconnaître les nombres à virgule flottante en Python.

Regardons ces deux chiffres :

4  
4.0

Vous devez vous dire qu'ils sont exactement les mêmes, mais en programmation et donc en Python, on les voit d'une manière complètement différente.

4 est un nombre **entier**, tandis que 4.0 est un nombre **à virgule flottante**. Le point est ce qui fait « flotter ».

D'autre part, ce ne sont pas seulement les points qui font flotter. Vous pouvez également utiliser la lettre e.

Lorsque vous souhaitez utiliser des nombres très grands ou très petits, vous pouvez utiliser **la notation scientifique**. On se rappelle des cours de Math ? Non, ok, on s'en doutait donc voici un petit rappel.

Prenons, par exemple, la vitesse de la lumière, exprimée en *mètres par seconde*. Si on l'écrit instinctivement, elle ressemblerait à : 300000000.

Pour éviter d'écrire autant de zéros (et être sûr de ne pas en oublier), les manuels de physique utilisent une forme abrégée, que vous avez probablement déjà vue:  $3 \times 10^8$ .

On peut y lire : trois fois dix à la puissance de huit.

En Python, le même effet est obtenu d'une manière légèrement différente, attention les yeux, voici la réponse : 3E8

La lettre **E** (vous pouvez également utiliser la lettre minuscule **e**, elle vient du mot **exposant**) est un enregistrement concis de la phrase *fois dix à la puissance de*.

Note:

- l'**exposant** (la valeur après E) doit être un entier;
- la **base** (la valeur devant le E) peut être un entier.

### 2.2.5 Le codage des nombres flottants

Voyons comment cette convention est utilisée pour enregistrer des nombres très petits (dans le sens de leur valeur absolue, qui est proche de zéro).

Une constante physique appelée *constante de Planck* (et notée  $h$ ) a la valeur de: **6,62607 x 10<sup>-34</sup>**.

Si vous souhaitez l'utiliser dans un programme, vous devez l'écrire de cette façon: `6.62607E-34`

Remarque: le fait que vous ayez choisi l'une des formes possibles de codage des valeurs flottantes ne signifie pas que Python le présentera de la même manière. Python peut parfois choisir **une notation différente** de la vôtre lors de l'affichage.

Par exemple, supposons que vous ayez décidé d'utiliser le littéral flottant suivant : `0.000000000000000000000001`

Lorsque vous exécutez ce littéral via Python :

```
print(0.000000000000000000000001)
```

Voici le résultat : `1e-22`

Python choisit toujours la **forme la plus économique pour la présentation du nombre**, et vous devez en tenir compte lors de la création de vos littéraux.

### 2.2.6 Les strings

Les chaînes de caractères (ou string) sont utilisées lorsque vous devez traiter du texte (comme des noms de toutes sortes, des adresses, des romans, etc.).

On en a déjà parlé un peu, par exemple, que les chaînes ont besoin de **guillemets** de la même manière que les flottants ont besoin de points.

Exemple : `«Je suis une chaîne.»`

Cependant, il y a un hic (et pas celui qui vient après une bonne bière). Le problème est de savoir comment encoder un guillemet à l'intérieur d'une chaîne qui est déjà délimitée par des guillemets.

Supposons que nous voulons imprimer un message très simple disant:

`J'aime « Monty Python »`

Comment pouvons-nous le faire sans générer d'erreur?  
Et bien deux solutions sont possibles !!

La première est basée sur le concept que nous connaissons déjà du **caractère d'évasion**, dont vous devez vous rappeler → **la barre oblique inverse**. La barre oblique inverse peut également « échapper les guillemets ».

Un guillemet précédé d'une barre oblique inverse change de signification et il n'est donc plus un délimiteur, mais juste un guillemet.

```
print("Vive \"Monty Python\"")
```

La deuxième solution peut être surprenante. Python peut utiliser une **apostrophe au lieu d'un guillemet**. L'un ou l'autre de ces caractères peut délimiter des chaînes, mais vous devez être **cohérent** (et c'est là que les problèmes arrivent).

Si vous ouvrez une chaîne avec un guillemet, vous devez le fermer avec un guillemet.

Si vous commencez une chaîne avec une apostrophe, vous devez la terminer par une apostrophe.

Cet exemple fonctionnera donc également : `print('Vive "Monty Python"')`

Et si maintenant, je souhaite incorporer un ` et un « dans ma chaîne :o Comment puis-je par exemple écrire la phrase suivante :

`J'aime les Monty Python !`

Vous avez une idée ??

Réponse assez simple si vous avez compris :

```
print('J\'aime Monty Python.')  
print("J'aime Monty Python.")
```

Petit rappel complémentaire ! Nous vous l'avons déjà dit mais une chaîne peut être vide. Pour cela il suffit d'écrire : `""` ou `" "`.

### 2.2.7 Valeurs booléennes (Boolean)

Pour conclure avec les littéraux de Python, il y en a deux supplémentaires. Ils ne sont pas aussi évidents que les précédents, car ils sont utilisés pour représenter une valeur très abstraite : **la vérité**.

Chaque fois que vous demandez à Python si un nombre est supérieur à un autre par exemple, la question entraîne la création de données spécifiques que l'on nommera valeur **booléenne**.

Le nom vient de George Boole (1815-1864), l'auteur de l'ouvrage fondamental, « *The Laws of Thought* », qui contient la définition de l'algèbre **booléenne** : Une partie de l'algèbre qui n'utilise que deux valeurs distinctes: `True` et `False`, notées `1` et `0` en informatique.

Un programmeur écrit un programme et le programme pose des questions. Python exécute le programme et fournit les réponses. Le programme doit donc pouvoir réagir en fonction des réponses reçues.

Heureusement, les ordinateurs ne connaissent que deux types de réponses :

- Oui, c'est vrai;
- Non, c'est faux.

Vous n'obtiendrez jamais une réponse comme: Je ne sais pas ou Probablement oui, mais je ne sais pas avec certitude.

Python est donc un reptile **binaire**.

Ces deux valeurs booléennes ont des dénnotations strictes en Python :

- `True`
- `False`

Vous ne pouvez rien changer - vous devez prendre ces symboles tels quels, y compris **la distinction majuscules/minuscules**.

Petit défi : Quelle sera la sortie de l'extrait de code suivant ?

```
print(Vrai > Faux)  
print(Vrai < Faux)
```



## 2.2.8 Exercice

## Temps estimé

5-10 minutes

## Objectifs

- se familiariser avec la fonction `print()` et ses capacités de formatage;
- pratiquer le codage des chaînes;
- expérimenter avec du code Python.

## Scénario

Écrivez un morceau de code d'une ligne, en utilisant la fonction `print()`, ainsi que les caractères de nouvelle ligne et d'échappement, pour correspondre au résultat attendu généré sur trois lignes.

## Résultats escomptés

```
"I'm"  
"learning"  
"Python"
```

## Mini Questions :

- Quels types de littéraux sont les deux exemples suivants?  
« Bonjour », « 007 »
- Quels types de littéraux sont les quatre exemples suivants?  
« 1.5 », 2.0, 528, faux
- Quelle est la valeur décimale du nombre binaire suivant ?  
1011

### 2.2.9 Résumé

1. **Les littéraux** sont des notations permettant de représenter certaines valeurs fixes dans le code. Python a différents types de littéraux, par exemple, un littéral peut être un nombre (littéraux numériques : `123`) ou une chaîne (littéraux de chaîne : « Je suis un littéral. »).

2. Le système **binaire** est un système de nombres qui emploie 2 comme base. Par conséquent, un nombre binaire est composé de 0 et de 1 seulement, par exemple, 1010 est 10 en décimale. De même, les systèmes de numération octale et hexadécimale utilisent respectivement 8 et 16 comme bases. Le système hexadécimal utilise les chiffres décimaux et six lettres supplémentaires.

3. **Les entiers** (ou simplement **int**) sont l'un des types numériques pris en charge par Python. Ce sont des nombres écrits sans composante fractionnaires, par exemple, 256 ou -1 (entiers négatifs).

4. Les nombres **à virgule flottante** (ou simplement **float**) sont un autre des types numériques pris en charge par Python. Ce sont des nombres qui contiennent (ou sont capables de contenir) une composante fractionnaire, par exemple, 1.27.

5. Pour encoder une apostrophe ou un guillemet à l'intérieur d'une chaîne, vous pouvez utiliser le caractère d'échappement, par exemple, `'J\'aime ce cours'`, ou ouvrez et fermez la chaîne à l'aide d'un ensemble de symboles opposés à ceux que vous souhaitez encoder, par exemple, « J'aime ce cours » pour encoder une apostrophe, et `"Il a dit « Python », pas « typhon »"` pour encoder une (double) guillemet.

6. **Les valeurs booléennes** sont les deux objets constants `True` et `False` utilisés pour représenter les valeurs de véracité (dans les contextes numériques, 1 est `True`, tandis que 0 est `False`).

Bonus :

Il y a un autre littéral spécial qui est utilisé en Python : le littéral `None`. Ce littéral est un objet appelé `NoneType`, et il est utilisé pour représenter **l'absence d'une valeur**.

## 2.3 Les opérateurs

### 2.3.1 Python est une calculatrice

Maintenant, nous allons vous montrer une toute nouvelle facette de la fonction `print()`.

Vous savez déjà que la fonction est capable de vous montrer les valeurs des littéraux qui lui sont passés par des arguments.

Mais en plus de cela, elle peut aussi réaliser des opérations :

```
print(2+2)
```

Retapez le code dans l'éditeur et exécutez-le. Pouvez-vous deviner le résultat?

Vous devriez voir le chiffre quatre. N'hésitez pas à expérimenter avec d'autres opérateurs.

Sans prendre cela trop au sérieux, vous venez de découvrir que Python peut être utilisé comme calculatrice. Pas très pratique, et certainement pas une calculatrice de poche, mais une calculatrice néanmoins.

En prenant cela plus au sérieux, nous entrons maintenant dans le chapitre des **opérateurs** et des **expressions**.

### 2.3.2 Les opérateurs de bases

Un **opérateur** est un symbole du langage de programmation, qui est capable de fonctionner sur des valeurs.

Par exemple, tout comme en arithmétique, le signe `+` (plus) est l'opérateur qui est capable d'ajouter deux nombres, donnant le résultat de l'addition.

Cependant, tous les opérateurs Python ne sont pas aussi évidents que le signe plus, alors passons en revue certains des opérateurs disponibles en Python, et nous expliquerons quelles règles régissent leur utilisation et comment interpréter les opérations qu'ils effectuent.

Nous commencerons par les opérateurs qui sont associés aux opérations arithmétiques les plus largement reconnaissables:

`+`, `-`, `*`, `/`, `//`, `%`, `**`

L'ordre de leur apparition n'est pas accidentel. Nous en reparlerons une fois que nous les aurons tous parcourus. Mais retenez-le déjà !

**N'oubliez pas** : les données et les opérateurs, lorsqu'ils sont connectés, forment des expressions. L'expression la plus simple est le littéral lui-même.

### 2.3.3 L'exponentiel

Un signe `**` (double astérisque) est un opérateur « **d'exponentiation** » (puissance). Son argument de gauche est la **base** et sa droite est l'**exposant**.

Les mathématiques classiques préfèrent la notation avec des exposants, comme ceci:  $2^3$ . Les éditeurs de texte pur n'acceptent pas cela, donc Python utilise `**` à la place, par exemple, `2 ** 3`.

Jetez un coup d'œil à nos exemples :

```
print(2 ** 3)
print(2 ** 3.)
print(2. ** 3)
print(2. ** 3.)
```

Remarque: nous avons entouré les doubles astérisques d'espaces dans nos exemples. Ce n'est pas obligatoire, mais cela améliore la **lisibilité** du code.

Exécutez le code et examinez attentivement les résultats qu'il produit. Voyez-vous une constante dans ceux-ci?

**Attention :** il est possible de formuler les règles suivantes avec ces résultats :

- Lorsque **les deux** arguments `**` sont des entiers, le résultat est également un entier ;
- **Lorsqu'au moins** un argument `**` est un float , le résultat est également un float.

**C'est une distinction importante à retenir.**

### 2.3.4 La multiplication

Un signe `*` (astérisque) est un opérateur de multiplication.

Exécutez le code ci-dessous et vérifiez si la règle integer vs float fonctionne toujours :

```
print(2 * 3)
print(2 * 3.)
print(2. * 3)
print(2. * 3.)
```

### 2.3.5 La division

Un signe `/` (barre oblique) est un opérateur **divisionnaire**.

La valeur devant la barre oblique est un **dividende**, la valeur derrière la barre oblique, un **diviseur**. Exemple :

```
print(6 / 3)
print(6 / 3.)
print(6. / 3)
print(6. / 3.)
```

Vous devriez voir qu'il y a une exception à la règle. Et oui déjà ...

**Le résultat produit par l'opérateur de division est toujours un flottant**, que le résultat semble ou non être un flottant à première vue:  $1 / 2$ , ou s'il ressemble à un entier pur:  $2 / 1$ .

Est-ce un problème? Oui. Il arrive parfois que vous ayez vraiment besoin d'une division qui fournit une valeur entière, pas une valeur flottante.

Heureusement, Python peut vous aider avec cela et oui Python sait « presque tout faire ».

### 2.3.6 La division entière

Un signe `//` (double barre oblique) est un opérateur **divisionnaire entière**. Il diffère de la norme `/` opérateur par deux détails:

- son résultat n'a pas la partie fractionnaire, ou est toujours égal à zéro (pour les flottants); Cela signifie que **les résultats sont toujours arrondis**;
- Il est conforme à la *règle des nombres entiers par rapport aux valeurs flottantes*.

Exécutez l'exemple ci-dessous et voyez les résultats :

```
print(6 // 3)
print(6 // 3.)
print(6. // 3)
print(6. // 3.)
```

Comme vous pouvez le voir, un entier *diviser par un entier* donne un **résultat entier**. Tous les autres cas produisent des flottants.

Faisons des tests plus avancés. Regardez l'extrait suivant :

```
print(6 // 4)
print(6. // 4)
```

Imaginez que nous utilisions `/` au lieu de `//`, pourriez-vous prédire les résultats?

Normalement c'est un grand : Oui et cela serait 1,5 dans les deux cas.

Mais à quels résultats faut-il s'attendre avec `//` ?

Exécutez le code et voyez par vous-même.

Ce que nous obtenons est différent, un entier et un flottant.

Le résultat de la division entière est toujours arrondi à la valeur entière la plus proche qui est inférieure au résultat réel (non arrondi).

Ceci est très important : **l'arrondi va toujours à l'entier inférieur**.

Regardez le code ci-dessous et essayons de prédire les résultats une fois de plus:

```
print(-6 // 4)
print(6. // -4)
```

Remarque : certaines valeurs sont négatives. Cela affectera évidemment le résultat. Mais comment?

Le résultat est deux en valeur négatives. Le résultat réel (non arrondi) est de -1,5 dans les deux cas. Cependant, les résultats font l'objet d'arrondis.

**L'arrondi va vers la valeur entière inférieure**, et la valeur entière inférieure est -2, d'où : -2 et -2,0.

En anglais, on parle aussi de floor division pour la division entière.

### 2.3.7 Le modulo

L'opérateur suivant est assez particulier, car il n'a pas d'équivalent parmi les opérateurs arithmétiques traditionnels.

Sa représentation graphique en Python est le signe % (pourcentage), ce qui peut sembler un peu déroutant et qui va provoquer pas mal d'erreurs chez vous.

Essayez de le considérer comme un slash (opérateur de division) accompagné de deux petits cercles amusants, oui ce n'est pas un pourcent à proprement parler.

Le résultat de l'opérateur est **un reste restant après la division entière**.

En d'autres termes, c'est la valeur restante après avoir divisé une valeur par une autre pour produire un quotient entier.

Remarque: l'opérateur est parfois appelé **remainder**.

Jetez un coup d'œil à l'extrait et essayez de prédire le résultat, puis exécutez-le:

```
print(14 % 4)
```

Comme vous pouvez le voir, le résultat est deux. Mais pourquoi:

- 14 // 4 donne 3 → c'est le **quotient entier**;
- 3 \* 4 donne 12 → à la suite de la **multiplication du quotient et du diviseur**;
- 14 - 12 donne 2 → c'est le **reste**.

Cet exemple est un peu plus compliqué :

```
print(12 % 4.5)
```

3.0 → pas 3 mais 3.0 (la règle fonctionne toujours: 12 // 4.5 donne 2.0; 2,0 \* 4,5 donne 9,0; 12 - 9,0 donne 3,0)

### 2.3.8 La division par zéro

Comme vous le savez probablement, la division par zéro ne fonctionne pas. N'essayez pas de :

- effectuer une division par zéro;
- effectuer une division entière par zéro;
- Trouvez un reste d'une division par zéro.

### 2.3.9 L'addition

L'opérateur d'addition est le signe + (plus), qui est entièrement conforme aux normes mathématiques.

Encore une fois, jetez un oeil à cet extrait et tirez vos conclusions :

```
print(-4 + 4)
print(-4. + 8)
```

### 2.3.10 L'opérateur de soustraction, les opérateurs unaires et binaires

L'opérateur de **soustraction** est évidemment le signe - (moins), bien que vous deviez vous souvenir que cet opérateur a également une autre signification → **il peut changer le signe d'un nombre.**

C'est une excellente occasion de présenter une distinction très importante entre les opérateurs **unaires** et **binaires**.

**L'opérateur moins attend deux arguments** : le gauche et le droit.

Pour cette raison, l'opérateur de soustraction est considéré comme l'un des opérateurs binaires, tout comme les opérateurs d'addition, de multiplication et de division.

Mais l'opérateur moins peut être utilisé d'une manière différente (unaire) : Analysez ce code et surtout la dernière ligne. Voici un opérateur unaire !

```
print(-4 - 4)
print(4. - 8)
print(-1.1)
```

Oups, il y avait aussi un opérateur unaire + que vous pouvez l'utiliser comme ceci:

```
print(+2)
```

L'opérateur conserve le signe de son seul argument. L'opérateur unaire n'agit donc que sur un seul opérande.

Bien qu'une telle construction soit syntaxiquement correcte, son utilisation n'a pas beaucoup de sens, et il serait difficile de trouver une bonne justification pour le faire. Vous ne notez jamais +2 mais toujours 2 dans la vie de tous les jours et bien en informatique cela sera pareil.

### 2.3.11 Les priorités des opérations

Jusqu'à présent, nous avons traité chaque opérateur comme s'il n'avait aucun lien avec les autres.

Il est évident qu'une situation aussi idéale et simple est une rareté dans la programmation réelle.

De plus, vous trouverez très souvent plus d'un opérateur dans une expression, et alors cette présomption n'est plus aussi évidente.

Considérez l'expression suivante :  $2 + 3 * 5$

Vous vous souvenez probablement avoir vu dans un cours de math que les **multiplications précèdent les additions**.

Vous vous souvenez sûrement que vous devez d'abord multiplier 3 par 5 et, tout gardant le 15 dans votre mémoire, puis d'ajouter ce nombre à 2, obtenant ainsi le résultat de 17.

Le phénomène qui pousse certains opérateurs à agir avant d'autres est connu sous le nom de **hiérarchie des priorités**.

Python définit précisément les priorités de tous les opérateurs et suppose que les opérateurs d'une priorité plus grande (supérieure) effectuent leurs opérations avant les opérateurs d'une priorité inférieure.

Donc, si vous savez que  $*$  a une priorité plus élevée que  $+$ , le calcul du résultat final devrait être évident.

De plus, les programmes fonctionnent comme les mathématiques, avec une forte liaison à gauche. C'est-à-dire que les calculs s'effectuent de la gauche vers la droite. Par exemple, si on utilise le code : `9%6 %2`, la réponse devrait être 1. Si nous utilisons une liaison à droite, nous obtiendrons une erreur. Pourquoi à votre avis ?

Il est à noter qu'il existe une exception (et oui encore) à cette liaison à gauche. Laquelle ? Et bien c'est l'exposant !

Essayons de coder le code suivant : `print(2 ** 2 ** 3)`

Quel est le résultat attendu ? 64 ? ou 256 ? Vous avez compris que si on en parle c'est car dans ce cas la liaison est à droite. On parle de « right-sided binding ».



### 2.3.12 La liste des priorités

Puisque vous êtes nouveau dans les opérateurs Python, nous ne voulons pas présenter la liste complète des priorités des opérateurs pour le moment.

Au lieu de cela, nous vous montrerons un formulaire tronqué, et nous l'étendrons de manière cohérente au fur et à mesure que nous introduirons de nouveaux opérateurs dans les différents cours de programmations de votre cursus.

Regardez le tableau ci-dessous :

Priorité Opérateur		
1	**	
2	+, - (note: les opérateurs unaires situés côté droit de l'opérateur de puissance se lient plus fortement)	unaire
3	*, /, //, %	
4	+, -	binaire

Remarque : nous avons énuméré les opérateurs dans **l'ordre des priorités les plus élevées (1) aux plus faibles (4) dans les chapitres ci-dessus, malin non ?**

Essayez de travailler l'expression suivante : `print(2 * 3 % 5)`

Les deux opérateurs (\* et %) ont la même priorité, de sorte que le résultat ne peut être deviné que lorsque vous connaissez le sens de liaison. Qu'en pensez-vous? Quel est le résultat?

### 2.3.13 Les parenthèses

Bien sûr, vous êtes toujours autorisé(obligé ? ) d'utiliser **des parenthèses**, ce qui peut changer l'ordre naturel d'un calcul.

Conformément aux règles arithmétiques, les **sous-expressions entre parenthèses sont toujours calculées en premier**. Les parenthèses ont donc une priorité presque absolue.

Vous pouvez utiliser autant de parenthèses que nécessaire, et elles sont souvent utilisées pour **améliorer la lisibilité** d'une expression, même si elles ne modifient pas l'ordre des opérations.

Voici un exemple d'expression avec plusieurs parenthèses :

```
print((5 * ((25 % 13) + 100) / (2 * 13)) // 2)
```

Essayez de calculer la valeur imprimée sur la console. Quel est le résultat ?

## 2.3.14 Résumé

1. Une **expression** est une combinaison de valeurs (ou variables, opérateurs, appels de fonctions) qui représente une certaine valeur, exemple :  $1+2$ .
2. **Les opérateurs** sont des symboles spéciaux ou des mots-clés qui sont capables d'opérer sur les valeurs et d'effectuer des opérations (mathématiques), par exemple, l'opérateur  $*$  multiplie deux valeurs:  $x * y$ .
3. Les opérateurs arithmétiques en Python:  $+$  (addition),  $-$  (soustraction),  $*$  (multiplication),  $/$  (division classique qui renvoie toujours un flottant),  $\%$  (modulo : divise l'opérande gauche par l'opérande droit et renvoie le reste de l'opération, par exemple,  $5 \% 2 = 1$ ),  $**$  (exponentiation : opérande gauche élevé à la puissance de l'opérande droit, par exemple,  $2 ** 3 = 2 * 2 * 2 = 8$ ),  $//$  (division entière : renvoie un nombre résultant de la division, mais arrondi au nombre entier inférieur le plus proche, par exemple,  $3 // 2,0 = 1,0$ )
4. Un opérateur **unaire** est un opérateur avec un seul opérande, par exemple,  $-1$  ou  $+3$ .
5. Un opérateur **binaire** est un opérateur avec deux opérandes, par exemple  $4 + 5$ , ou  $12 \% 5$ .
6. Certains opérateurs agissent avant **d'autres – la hiérarchie des priorités** :
  - l'opérateur  $**$  (exponentiation) a la priorité la plus élevée ;
  - alors l'unaire  $+$  et  $-$  (note: un opérateur unaire à droite de l'opérateur d'exponentiation se lie plus fortement, par exemple:  $4 ** -1$  est égal à  $0,25$ )
  - puis  $*$ ,  $/$ ,  $//$ , et  $\%$ ;
  - et, enfin, la priorité la plus basse: le binaire  $+$  et  $-$ .
7. Les sous-expressions entre **parenthèses** sont toujours calculées en premier, par exemple,  $15 - 1 * (5 * (1 + 2)) = 0$ .
8. L'opérateur **d'exponentiation** utilise **la liaison droite**, par exemple,  $2 ** 2 ** 3 = 256$ .

**Exercice 1**

Quelle est la sortie de l'extrait de code suivant ?

```
print((2 ** 4), (2 * 4.), (2 * 4))
```

**Exercice 2**

Quelle est la sortie de l'extrait de code suivant ?

```
print((-2 / 4), (2 / 4), (2 // 4), (-2 // 4))
```

**Exercice 3**

Quelle est la sortie de l'extrait de code suivant ?

```
print((2 % -4), (2 % 4), (2 ** 3 ** 2))
```

## 2.4 Les variables

### 2.4.1 Introduction

On sait qu'on peut utiliser des opérateurs, utiliser différents littéraux, mais comment stocker nos valeurs sans les afficher ?

C'est une question tout à fait normale de se demander comment **stocker les résultats** de ces opérations, afin de les utiliser dans d'autres opérations, et ainsi de suite.

Comment enregistrer les résultats intermédiaires et les réutiliser pour produire les suivants ?

Python vous aidera avec cela. Il propose des « boîtes » spéciales (conteneurs) à cet effet, et ces boîtes sont appelées **variables**, le nom lui-même suggère que le contenu de ces conteneurs peut être modifié de (presque) n'importe quelle façon.

Qu'est-ce que chaque variable Python a obligatoirement ?

- un nom;
- une valeur (le contenu du conteneur)



Commençons par les problèmes liés au nom d'une variable.

Les variables n'apparaissent pas automatiquement dans un programme. En tant que développeur, vous devez décider combien et quelles variables utiliser dans vos programmes.

Vous devez également les nommer.

Si vous souhaitez **donner un nom à une variable**, vous devez suivre des règles strictes :

- Le nom de la variable doit être composé de lettres majuscules ou minuscules, de chiffres et du caractère \_ (trait de soulignement)
- le nom de la variable doit commencer par une lettre;
- le caractère de soulignement est une lettre;
- les lettres majuscules et minuscules sont traitées différemment (un peu différemment que dans le monde réel : *Alice* et *ALICE* sont les mêmes prénoms, mais en Python ce sont deux noms de variables différents, et par conséquent, deux variables différentes);
- le nom de la variable ne doit pas être l'un des mots réservés de Python

### 2.4.2 Les noms corrects

Notez que les mêmes restrictions s'appliquent aux noms de fonction.

Python n'impose pas de restrictions sur la longueur des noms de variables, mais cela ne signifie pas qu'un nom de variable long est toujours meilleur qu'un nom court. Essayez de rester cohérent.

Voici quelques noms de variables corrects, mais pas toujours pratiques :

MyVariable, i, t34, Exchange\_Rate, compteur, Jour\_avant\_Noel,  
CeNomEstTellementLongQueVousFerezSurementUneErreurEnLEcrivant, \_ .

De plus, Python vous permet d'utiliser non seulement des lettres latines, mais aussi des caractères spécifiques aux langues qui utilisent d'autres alphabets.

Ces noms de variables sont également corrects :

Adiós\_Señora, OnTheFloor, Einbahnstraße, переменная.

Et maintenant pour certains **noms incorrects**:

10t (ne commence pas par une lettre), Taux de change (contient un espace)

Remarque : Le [PEP 8 -- Guide de style pour le code Python](#) recommande la convention de nommage suivante pour les variables et les fonctions en Python :

- Les noms des variables doivent être en minuscules, avec des mots séparés par des traits de soulignement pour améliorer la lisibilité (par exemple, **var**, **my\_variable**)
- Les noms de fonction suivent la même convention que les noms de variables (par exemple, **fun**, **my\_function**)
- il est également possible d'utiliser la casse mixte (par exemple, **myVariable**), mais seulement dans les contextes où c'est déjà le style dominant, pour conserver la rétrocompatibilité avec la convention adoptée.

### 2.4.3 Les mots-clés

Jetez un coup d'œil à la liste des mots qui jouent un rôle très spécial dans chaque programme Python :

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',  
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',  
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',  
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with',  
'yield']
```

Ils sont appelés **mots-clés** ou (plus précisément) **mots-clés réservés**. Ils sont réservés car vous **ne POUVEZ pas les utiliser comme noms** : ni pour vos variables, ni pour vos fonctions, ni pour quoi que ce soit d'autres.

La signification du mot-clé réservé est **prédéfinie** et ne doit en aucun cas être modifiée.

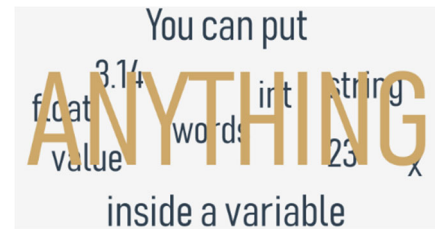
Heureusement, en raison du fait que Python est sensible à la casse, vous pouvez modifier n'importe lequel de ces mots en changeant la casse de n'importe quelle lettre, créant ainsi un nouveau mot, qui n'est plus réservé.

Par exemple, **vous ne pouvez pas nommer** votre variable comme suit : `import` Vous ne devez pas avoir une variable nommée de cette manière, c'est interdit. Mais vous pouvez faire ceci à la place : `Import`

#### 2.4.4 Créaton de variables

Que pouvez-vous mettre à l'intérieur d'une variable ?

Réponse : ...Tout.



Vous pouvez utiliser une variable pour stocker n'importe quelle valeur de l'un des types déjà présentés, et beaucoup plus de celles que nous ne vous avons pas encore montrées.

La valeur d'une variable est ce que vous y avez mis. Cela peut varier aussi souvent que vous en avez besoin ou que vous le souhaitez. Il peut s'agir d'un entier un instant et d'un flottant un instant plus tard, devenant éventuellement une chaîne. (Ceci ne marche pas dans tous les langages)

Parlons maintenant de deux choses importantes :

- comment les **variables sont créées** ;
- comment mettre des valeurs **à l'intérieur** (ou plutôt, comment leur donner ou **leur attribuer des valeurs**).

SE SOUVENIR que :

- **Une variable naît à la suite de l'attribution d'une valeur à celle-ci.** Contrairement à d'autres langages, vous n'avez pas besoin de le déclarer d'une manière particulière.
- Si vous affectez une valeur à une variable inexistante, la variable sera **automatiquement créée**. Vous n'avez rien d'autre à faire.
- La création (sa syntaxe) est extrêmement simple: **il suffit d'utiliser le nom de la variable souhaitée, puis le signe égal (=) et la valeur que vous souhaitez mettre dans la variable.**

Prenons un exemple :

```
var = 1
print(var)
```

Il se compose de deux instructions simples:

- Le premier d'entre eux crée une variable nommée `var` et attribue un littéral avec une valeur entière égale à 1.
- La seconde imprime la valeur de la variable nouvellement créée sur la console.

Remarque: `print()` a encore un autre côté : il peut également gérer des variables. Savez-vous quelle sera la sortie de l'extrait ?

Vous désirez faire un code plus important et bien vous êtes autorisé à utiliser autant de déclarations de variables que nécessaire pour atteindre votre objectif, comme ceci par exemple :

```
var = 1
account_balance = 1000.0
client_name = 'John Doe'
print(var, account_balance, client_name)
print(var)
```

Vous n'êtes pas autorisé à utiliser une variable **qui n'existe pas** (en d'autres termes, une variable à laquelle aucune valeur n'a encore été attribuée). Cet exemple **provoquera une erreur** :

```
var = 1
print(Var)
```

Nous avons essayé d'utiliser une variable nommée `Var`, qui n'a aucune valeur (note: **var** et **Var** sont des entités différentes, rappelez-vous.)

Astuce :

Vous pouvez utiliser la fonction `print()` et combiner du texte et des variables à l'aide de l'opérateur `+` pour générer des ensembles de chaînes de caractères et des variables, par exemple :

```
var = "3.8.5"
print("Python version: " + var)
```

#### 2.4.5 Essayons d'affecter une variable déjà existante

Comment attribuer une nouvelle valeur à une variable déjà créée ? De la même manière. Il vous suffit d'utiliser le signe égal (=).

Le signe égal est en fait un opérateur d'affectation. Bien que cela puisse sembler étrange, l'opérateur a une syntaxe simple et une interprétation sans ambiguïté.

Il attribue la valeur de droite à son argument de gauche, tandis que l'argument droit peut être une expression arbitrairement complexe impliquant des littéraux, des opérateurs et des variables déjà définies.

Regardez le code ci-dessous :

```
var = 1
print(var)
var = var + 1
print(var)
```

Le code envoie deux lignes à la console :

```
1
2
```

La première ligne de l'extrait crée une nouvelle variable nommée var et lui attribue 1.

L'instruction se lit comme suit : attribuez une valeur de 1 à une variable nommée var.

On peut le résumer en : attribuer 1 à var.

La troisième ligne attribue la même variable avec la nouvelle valeur tirée de la variable elle-même, additionnée à 1. En voyant un tel constat, un mathématicien protesterait probablement, aucune valeur ne peut être égale à elle-même plus un. C'est une contradiction.

Mais Python traite le signe = non pas comme « égal à », mais comme assignant une valeur.

Alors, comment lisez-vous un tel enregistrement dans le programme?

Prenez la valeur actuelle de la variable var, ajoutez-y 1 et stockez le résultat dans la variable var.

En effet, la valeur de la variable var a été incrémentée d'un, ce qui n'a rien à voir avec la comparaison de la variable avec n'importe quelle valeur.

Savez-vous quelle sera la sortie de l'extrait suivant ?

```
var = 100
var = 200 + 300
print(var)
```

500 : pourquoi? Eh bien, tout d'abord, la variable var est créée et affectée à une valeur de 100. Ensuite, une nouvelle valeur est attribuée à la même variable : le résultat de l'addition de 200 à 300, soit 500.

#### 2.4.6 Pythagore et les mathématiques

Maintenant, vous devriez être capable de construire un programme court résolvant des problèmes mathématiques simples tels que le théorème de Pythagore:

*Le carré de l'hypoténuse est égal à la somme des carrés des deux autres côtés.*

Le code suivant évalue la longueur de l'hypoténuse (c'est-à-dire le côté le plus long d'un triangle rectangle, celui opposé à l'angle droit) en utilisant le théorème de Pythagore:

```
a = 3,0
b = 4,0
c = (a ** 2 + b ** 2) ** 0,5
print("c =", c)
```

Remarque: nous devons utiliser l'opérateur \*\* pour évaluer la racine carrée comme:

$$\sqrt{x} = x^{(1/2)}$$

et

$$c = \sqrt{a^2 + b^2}$$

Pouvez-vous deviner la sortie du code? Vérifiez en exécutant le code dans l'éditeur pour confirmer vos prédictions.



## 2.4.7 Exercice

## Temps estimé

10 minutes

## Objectifs

Se familiariser avec le concept de stockage et d'utilisation de différents types de données en Python;

## Scénario

Voici une petite histoire :

Il était une fois à Appleland, un homme nommé Jean qui avait trois pommes, une femme nommée Marie qui avait cinq pommes et un garçon nommé Adam qui en avait six. Ils étaient tous très heureux et ont vécu très longtemps. Fin de l'histoire.

Votre tâche consiste à :

- Créez les variables : Jean, Marie et Adam ;
- Attribuez des valeurs aux variables. Les valeurs doivent être égales au nombre de fruits possédés respectivement par Jean, Marie et Adam;
- Après avoir stocké les nombres dans les variables, imprimez les variables sur une ligne et séparez chacune d'elles par une virgule;
- Créez maintenant une nouvelle variable nommée `total_pommes` égale à l'ajout des trois anciennes variables.
- Imprimer la valeur stockée dans `total_pommes` sur la console ;

**Expérimentez avec votre code** : créez de nouvelles variables, attribuez-leur différentes valeurs et effectuez diverses opérations arithmétiques sur celles-ci (par exemple, +, -, \*, /, //, etc.). Essayez d'imprimer une chaîne et un entier ensemble sur une ligne, par exemple, « Nombre total de pommes: » et `total_pommes`.

#### 2.4.8 Les opérateurs raccourcis

Très souvent, nous voulons utiliser une seule et même variable à droite et à gauche de l'opérateur `=`.

Par exemple, si nous devons calculer une série de valeurs successives de puissances de 2, nous pouvons utiliser une instruction comme celle-ci :

```
x = x * 2
```

Mais on peut aussi utiliser des expressions avec des variables plus longue à écrire, comme par exemple si vous n'arrivez pas à vous endormir et que vous essayez de gérer cela en utilisant de bonnes vieilles méthodes à l'ancienne :

```
moutons = moutons + 1
```

C'est laborieux et long mais heureusement Python vous offre un moyen raccourci d'écrire des opérations comme celles-ci :

```
x *= 2  
moutons += 1
```

Essayons de présenter une description générale de ces opérations.

Si **op** est un opérateur à deux arguments (c'est une condition très importante) et que l'opérateur est utilisé dans le contexte suivant :

```
variable = variable op expression
```

Il peut être simplifié et affiché comme suit :

```
variable op= expression
```

Jetez un coup d'œil aux exemples ci-dessous pour mieux comprendre :

```
i = i + 2 * j ⇒ i += 2 * j  
var = var / 2 ⇒ var /= 2  
frein = rem % 10 ⇒ rem %= 10  
j = j - (i + var + rem) ⇒ j -= (i + var + rem)  
x = x ** 2 ⇒ x **= 2
```

## 2.4.9 Exercices

## Temps estimé

10 minutes

## Objectifs

- se familiariser avec le concept des variables et travailler avec elles;
- effectuer des calculs et des conversions de base;
- expérimenter avec du code Python.

## Scénario

Les miles et les kilomètres sont des unités de longueur ou de distance.

En gardant à l'esprit que 1 mile est égal à environ 1,61 kilomètre, complétez le programme dans l'éditeur afin qu'il convertisse:

- miles à kilomètres;
- kilomètres à miles.

Ne modifiez rien dans le code existant. Écrivez votre code aux endroits indiqués par `###`. Testez votre programme avec les données que nous avons fournies dans le code source.

Portez une attention particulière à ce qui se passe à l'intérieur de la fonction `print()`. Analysez comment nous fournissons plusieurs arguments à la fonction et comment nous produisons les données attendues.

Notez que certains des arguments à l'intérieur de la fonction `print()` sont des chaînes (par exemple, « miles is », tandis que d'autres sont des variables (par exemple, miles).

## Compléments :

Il y a une autre chose intéressante qui se passe. Voyez-vous l'autre fonction à l'intérieur de la fonction `print()` ? C'est la fonction `round()`. Son travail consiste à arrondir le résultat obtenu au nombre de décimales spécifié entre parenthèses et à renvoyer un float (à l'intérieur de la fonction `round()`, vous pouvez trouver le nom de la variable, une virgule et le nombre de décimales que nous visons). Nous allons parler des fonctions très bientôt, alors ne vous inquiétez pas que tout ne soit pas encore tout à fait clair. Nous voulons juste éveiller votre curiosité et vous faire questionner votre professeur de laboratoire 😊

Une fois le laboratoire terminé, expérimentez davantage. Essayez d'écrire différents convertisseurs, par exemple, un convertisseur USD en EUR, un convertisseur de température, etc. Laissez libre cours à votre imagination pendant un petit temps de travail. Essayez de générer les résultats en combinant des chaînes et des variables.

Essayez d'utiliser et d'expérimenter plus en profondeur avec la fonction `round()` pour arrondir vos résultats à une, deux ou trois décimales. Découvrez ce qui se passe si vous ne fournissez aucun nombre de chiffres. N'oubliez pas de tester vos programmes. Expérimentez, tirez des conclusions et apprenez. **Soyez curieux.**

## Résultats escomptés

7,38 milles correspondent à 11,88 kilomètres  
12,25 kilomètres est 7,61 milles

## Code de départ pour votre travail :

```
kilometers = 12.25
miles = 7.38

miles_to_kilometers = ###
kilometers_to_miles = ###

print(miles, "miles is", round(miles_to_kilometers, 2), "kilometers")
print(kilometers, "kilometers is", round(kilometers_to_miles, 2), "miles")
```

## Temps estimé

10-15 minutes

## Objectifs

- se familiariser avec le concept de nombres, d'opérateurs et d'opérations arithmétiques en Python;
- effectuer des calculs de base.

## Scénario

Regardez le code ci-dessous: il lit une valeur flottante, la place dans une variable nommée `x` et imprime la valeur d'une variable nommée `y`. Votre tâche consiste à compléter le code afin d'évaluer l'expression suivante :

$$3x^3 - 2x^2 + 3x - 1$$

Le résultat doit être affecté à `y`.

Rappelez-vous que la notation algébrique classique aime omettre l'opérateur de multiplication, vous devez l'utiliser explicitement pour votre part. Notez comment nous modifions le type de données pour nous assurer que `x` est de type flottant.

Gardez votre code propre et lisible, et testez-le en utilisant les données que nous avons fournies, en l'affectant à chaque fois à la variable `x` (en la codant en dur).

## Données de test

Entrée :

```
x = 0
x = 1
x = -1
```

Sortie :

```
y = -1.0
y = 3.0
y = -9.0
```

## Code de base :

```
x = # hardcoded your test data here
x = float(x)
# write your code here
print("y =", y)
```

## 2.4.10 Résumé

1. Une **variable** est un emplacement nommé réservé au stockage des valeurs dans la mémoire. Une variable est créée ou initialisée automatiquement lorsque vous lui attribuez une valeur pour la première fois.

2. Chaque variable doit avoir un nom unique - un **identifiant**. Un nom valable doit être une séquence de caractères non vide, doit commencer par le trait de soulignement (\_) ou une lettre, et il ne peut pas s'agir d'un mot clé Python. Le premier caractère peut être suivi de traits de soulignement, de lettres et de chiffres. Les identificateurs en Python sont sensibles à la casse.

3. Python est un langage **typé dynamiquement**, ce qui signifie que vous n'avez pas besoin d'y *déclarer* des variables et leurs types. Pour attribuer des valeurs à des variables, vous pouvez utiliser un opérateur d'affectation simple sous la forme du signe égal (=), c'est-à-dire par exemple : `var = 1`.

4. Vous pouvez également utiliser **des opérateurs d'affectation composés** (opérateurs raccourci) pour modifier les valeurs affectées aux variables, par exemple, `var += 1` ou `var /= 5 * 2`.

5. Vous pouvez attribuer de nouvelles valeurs à des variables déjà existantes à l'aide de l'opérateur d'affectation ou de l'un des opérateurs composés, par exemple:

```
var = 2
print(var)
var = 3
print(var)
```

6. Vous pouvez combiner du texte (on parle de concaténation) et des variables à l'aide de l'opérateur `+` et utiliser la fonction `print()` pour générer des chaînes et des variables, par exemple:

```
var = "007"
print("Agent" + var)
```

**Exercice 1**

Quelle est la sortie de l'extrait de code suivant ?

```
var = 2
var = 3
print(var)
```

**Exercice 2**

Lequel des noms de variables suivants est illégal en Python ?

```
my_var
m
101
averylongvariablename
m101
m 101
Del
del
```

**Exercice 3**

Quelle est la sortie de l'extrait de code suivant ?

```
a = '1'
b = "1"
print(a + b)
```

#### Exercice 4

Quelle est la sortie de l'extrait de code suivant ?

```
a = 6
b = 3
a /= 2 * b
print(a)
```

## 2.5 Les commentaires

### 2.5.1 Laisser des commentaires dans le code : pourquoi, comment et quand

Vous voudrez peut-être écrire quelques mots qui seront adressés non pas à Python mais à des humains lisant votre code, généralement pour expliquer aux autres lecteurs du code comment fonctionnent les astuces utilisées dans le code, ou la signification des variables, et éventuellement, afin de conserver des informations stockées sur qui est l'auteur et quand le programme a été écrit.

Une remarque insérée dans le programme, qui est **omise au moment de l'exécution**, est appelée un **commentaire**.

Comment laisser ce genre de commentaire dans le code source ? Cela doit être fait d'une manière qui ne forcera pas Python à l'interpréter comme faisant partie du code.

Chaque fois que Python rencontre un commentaire dans votre programme, le commentaire est complètement transparent, du point de vue de Python, ce n'est qu'un espace (quelle que soit la longueur du commentaire réel).

En Python, un commentaire est un morceau de texte qui commence par un signe `#` (hachage) et s'étend jusqu'à la fin de la ligne.

Si vous voulez un commentaire qui s'étend sur plusieurs lignes, vous devez mettre un hachage devant tous.

Tout comme ici:

```
# Ce programme évalue l'hypoténuse c.
# a et b sont les longueurs des jambes.
a = 3,0
b = 4,0
c = (a ** 2 + b ** 2) ** 0,5 # Nous utilisons ** au lieu de racine
carrée.
print (« c = », c)
```

Les bons développeurs responsables **décrivent chaque morceau de code important**, par exemple, en expliquant le rôle des variables; bien qu'il faille préciser que la meilleure façon de commenter les variables est de les nommer de manière non ambiguë. Certaines normes actuelles prônent également le « no comment », car le code doit être assez lisible pour ne pas être commenté.

Une autre manière de faire du commentaire sur plusieurs lignes est le docstring,

comme ci-dessous :

```
"""
Ceci est un commentaire sur plusieurs lignes grâce aux docstrings

print("Bonjour le monde")
print("Bonjour l'univers")
print("Bonjour tout le monde")
"""
print("Bonjour les campers")
```

Par exemple, si une variable particulière est conçue pour stocker une zone d'un carré unique, le nom `square_area` sera évidemment meilleur que `nain jaune`.

Nous dirons dans ce cas que la variable est **auto-commentée**.

Les commentaires peuvent être utiles à un autre égard, vous pouvez les utiliser pour **marquer un morceau de code qui n'est actuellement pas nécessaire** pour une raison quelconque.

Regardez l'exemple ci-dessous, si vous **supprimez les commentaires** de la 4ème ligne, cela affectera la sortie du code :

```
# Ceci est un programme de test.
x = 1
y = 2
# y = y + x
imprimer (x + y)
```

Ceci est souvent fait lors du test d'un programme, afin d'isoler l'endroit où une erreur pourrait être cachée.



## 2.5.2 Exercice

## Temps estimé

5 minutes

## Objectifs

- se familiariser avec le concept de commentaires en Python;
- utiliser et ne pas utiliser les commentaires;
- remplacer les commentaires par du code;
- expérimenter avec du code Python.

## Scénario

Le code ci-dessous contient des commentaires. Essayez de l'améliorer : ajoutez ou supprimez des commentaires là où vous le jugez approprié (oui, parfois supprimer un commentaire peut rendre le code plus lisible), et changez les noms des variables lorsque vous pensez que cela améliorera la compréhension du code.

## Code source :

```
#Ce programme calcule le nombre de secondes dans un nombre d'heures données  
# ce programme a été écrit il y a deux jours  
  
a = 2 # Nombre d'heures  
seconds = 3600 # Nombre de secondes dans une heure  
  
print("Hours: ", a) #imprime le nombre d'heures  
# print(« Seconds in Hours: « , a * seconds) # impression du nombre de  
secondes dans un nombre d'heures données  
  
  
#Ici nous devrions aussi imprimer « Goodbye », mais un programmeur n'a pas  
eu le temps d'écrire du code  
#c'est la fin du programme qui calcule le nombre de secondes en 3 heures
```

## Remarque :

- Les commentaires sont très importants. Ils sont utilisés non seulement pour rendre vos programmes **plus faciles à comprendre**, mais aussi pour **désactiver les morceaux de code qui ne sont actuellement pas nécessaires** (par exemple, lorsque vous devez tester certaines parties de votre code uniquement et en ignorer d'autres). Les bons programmeurs donnent des **noms auto-commentés** aux variables.
- Il est bon d'utiliser des noms de variables **lisibles**, et parfois il est préférable **de diviser votre code** en morceaux nommés (par exemple, des fonctions). Dans certaines situations, c'est une bonne idée d'écrire les étapes de calcul de manière plus claire.
- Attention aux erreurs dans les commentaires, ça n'impacte pas le fonctionnement mais ça peut détruire une maintenance.

### 2.5.3 Résumé

1. Les commentaires peuvent être utilisés pour laisser des informations supplémentaires dans le code. Ils sont omis lors de l'exécution. Les informations laissées dans le code source sont adressées à des lecteurs humains. En Python, un commentaire est un morceau de texte qui commence par `#`. Le commentaire s'étend jusqu'à la fin de la ligne.

2. Si vous souhaitez placer un commentaire qui s'étend sur plusieurs lignes, vous devez placer `#` devant toutes les lignes (ou utiliser un docstring). De plus, vous pouvez utiliser un commentaire pour marquer un morceau de code qui n'est pas nécessaire pour le moment.

3. Chaque fois que cela est possible et justifié, vous devriez donner des noms **auto-commentés** aux variables, par exemple, si vous utilisez deux variables pour stocker une longueur et une largeur de quelque chose, les noms de variables `longueur` et `largeur` sont un meilleur choix que `myvar1` et `myvar2`.

4. Il est important d'utiliser des commentaires pour rendre les programmes plus faciles à comprendre et d'utiliser des noms de variables lisibles et significatifs dans le code. Cependant, il est tout aussi important de **ne pas utiliser** de noms de variables qui prêtent à confusion, ou de laisser des commentaires qui contiennent des informations erronées ou incorrectes!

5. Les commentaires peuvent être importants lorsque *vous* lisez votre propre code après un certain temps (croyez-nous, les développeurs oublient ce que fait leur propre code), et lorsque d'autres lisent votre code (ils peuvent les aider à comprendre ce que font vos programmes et comment ils le font, beaucoup plus rapidement qu'en tentant de vous déchiffrer).

#### Exercice 1

Quelle est la sortie de l'extrait de code suivant ?

```
# print("Chaîne #1")
print("Chaîne #2")
```

#### Exercice 2

Que se passe-t-il lorsque vous exécutez le code suivant ?

```
# This is
a multiline
comment. #

print("Hello!")
```

## 2.6 Parlons avec notre ordinateur/programme

### 2.6.1 La fonction input()

Nous allons maintenant vous présenter une toute nouvelle fonction, qui semble être un reflet miroir de la bonne vieille fonction `print()`.

Pourquoi? Eh bien, `print()` envoie des données à la console.  
La nouvelle fonction en tire des données.

`print()` n'a aucun résultat utilisable. La signification de la nouvelle fonction est de **renvoyer un résultat utilisable**.

La fonction est nommée `input()`. Le nom de cette fonction dit tout, enfin si vous parlez un peu anglais.

La fonction `input()` est capable de lire les données saisies par l'utilisateur et de renvoyer les mêmes données au programme en cours d'exécution.

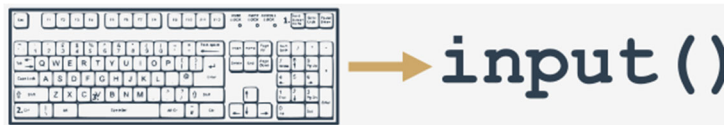
Le programme peut manipuler ces données, rendant le code vraiment interactif.

Pratiquement tous les programmes **lisent et traitent les données**. Un programme qui n'obtient pas d'entrée d'un utilisateur est un programme **sourd**.

Jetez un coup d'œil à notre exemple :

```
print("Dis-moi n'importe quoi... ")
n_importe_quoi = input()
print("Humm... " , n_importe_quoi, " ... Vraiment? " )
```

Il montre un cas très simple d'utilisation de la fonction **`input()`**.



Note:

- Le programme **invite l'utilisateur à entrer certaines** données à partir de la console (très probablement à l'aide d'un clavier, bien qu'il soit également possible de saisir des données en utilisant la voix ou l'image);
- la fonction `input()` est invoquée sans arguments (c'est la façon la plus simple d'utiliser la fonction); la fonction **basculera la console en mode d'entrée**; vous verrez un curseur clignotant, et vous pourrez entrer des caractères, en terminant en appuyant sur la touche *Entrée*; toutes les données saisies seront **envoyées à votre programme** via le résultat de la fonction;
- Remarque : vous devez affecter le résultat à une variable ; Ceci est crucial, manquer cette étape entraînera la perte des données saisies;
- Ensuite, nous utilisons la fonction `print()` pour sortir les données que nous obtenons, avec quelques infos supplémentaires.

### 2.6.2 Ajoutons un argument à notre fonction

La fonction `input()` peut faire autre chose : elle peut inviter l'utilisateur à écrire sans avoir besoin de la fonction `print()`.

Nous avons un peu modifié notre exemple, regardez le code:

```
anything = input("Tell me anything...")
print("Hmm...", anything, "...Really?")
```

Note:

- La fonction `input()` est appelée avec un argument : c'est une chaîne contenant un message;
- Le message sera affiché sur la console avant que l'utilisateur n'ait la possibilité d'entrer quoi que ce soit;
- `input()` fera alors son travail.
- Cette variante de l'appel `input()` simplifie le code et le rend plus clair.

### 2.6.3 Le résultat de notre fonction `input()`

Nous l'avons déjà dit, mais il faut le répéter sans ambiguïté, et un professeur à l'habitude de répéter au moins 3x : le **résultat de la** fonction `input()` **est une chaîne**.

Chaîne contenant tous les caractères que l'utilisateur saisit à partir du clavier. Ce **n'est pas** un entier ou un flottant.

Cela signifie que vous **ne devez pas l'utiliser comme argument d'une opération arithmétique**, par exemple, vous ne pouvez pas utiliser ces données pour les multiplier, les diviser par quoi que ce soit ou encore diviser quoi que ce soit par elles.

```
anything = input("Entrez un nombre: ")
quelque_chose = n_importe_quoi ** 2.0
print(n_importe_quoi, « à la puissance de 2 est égal à », quelque_chose)
```

### 2.6.4 La fonction `input()` - opérations interdites

Regardez le code suivant. Exécutez-le, entrez n'importe quel nombre et appuyez sur *Entrée*.

```
# Testing TypeError message.

anything = input("Enter a number: ")
something = anything ** 2.0
print(anything, "to the power of 2 is", something)
```

Que se passe-t-il ?

Python aurait dû vous donner la sortie suivante:

Retraçage (dernier appel le plus récent) :  
Fichier « .main.py », ligne 4, dans <module>  
quelque chose = n'importe quoi \*\* 2.0  
TypeError : type(s) d'opérande non pris en charge pour \*\* ou pow(): 'str' et 'float'

La dernière ligne de la phrase explique tout ce qu'il faut savoir : vous avez essayé d'appliquer l'opérateur \*\* à 'str' (chaîne) accompagné de 'float'. Ceci est interdit.

Cela devrait être évident pour vous avec ce qu'on a vu, pouvez-vous prédire la valeur de « être ou ne pas être » élevée à la puissance de 2?

On ne peut pas. Python ne peut pas non plus.

Sommes-nous tombés dans une impasse? Y a-t-il une solution à ce problème? Bien sûr qu'il y en a.

#### 2.6.5 Formatage des types (cast)

Python offre deux fonctions simples pour spécifier un type de données et résoudre ce problème : `int()` et `float()`.

Leurs noms se commentent eux-mêmes :

- la fonction `int()` prend un argument (par exemple, une chaîne : `int(string)`) et essaie de le convertir en entier ; si elle échoue, l'ensemble du programme échouera aussi (il existe une solution de contournement pour cette situation, mais nous vous le montrerons un peu plus tard) ;
- La fonction `float()` prend un argument (par exemple, une chaîne : `float(string)`) et tente de le convertir en float (le reste est le même).

C'est très simple et très efficace. De plus, vous pouvez appeler n'importe laquelle des fonctions en leur passant directement les résultats `input()`. Il n'est pas nécessaire d'utiliser une variable comme stockage intermédiaire.

Nous avons implémenté l'idée dans le code ci-dessous, jetez-y un coup d'œil.

```
variable = float(input("Entrer un nombre: "))  
variable2 = variable ** 2.0  
print(variable, " à la puissance 2 est", variable2)
```

Pouvez-vous imaginer comment la chaîne saisie par l'utilisateur passe de `input()` à `print()` ?

### 2.6.6 Bonus sur input() et la conversion de type

Avoir une équipe composée du trio input()-int()-float() ouvre déjà beaucoup de nouvelles possibilités.

Vous pourrez éventuellement écrire des programmes complets, accepter des données sous forme de nombres, les traiter et afficher les résultats.

Bien sûr, ces programmes seront très primitifs et peu utilisables, car ils ne peuvent pas prendre de décisions et, par conséquent, ne sont pas capables de réagir différemment à différentes situations.

Ce n'est pas vraiment un problème, cependant; Nous vous montrerons comment le surmonter bientôt.

Notre exemple suivant fait référence au programme précédent pour trouver la durée d'une hypoténuse. Réécrivons-le et rendons-le capable de lire les longueurs des cotés à partir de la console. Voici à quoi cela ressemble maintenant.

```
side_a = float(input("Input first side length: "))
side_b = float(input("Input second side length: "))
hypo = (side_a**2 + side_b**2) ** .5
print("Hypotenuse length is", hypo)
```

Le programme demande à l'utilisateur la taille des deux cotés, évalue l'hypoténuse et imprime le résultat.

Exécutez-le et essayez d'entrer des valeurs négatives.

Le programme (malheureusement) ne réagit pas à cette erreur évidente.

Ignorons cette faiblesse pour l'instant.

Notez que dans le programme que vous pouvez voir ci-dessus, la variable hypo n'est utilisée que dans un seul but: enregistrer la valeur calculée entre l'exécution de la ligne de code adjacente.

Comme la fonction print() accepte une expression comme argument, vous pouvez **supprimer la variable** du code.

```
side_a = float(input("Input first side length: "))
side_b = float(input("Input second side length: "))
print("Hypotenuse length is", (side_a**2 + side_b**2) ** .5)
```

On apprend enfin à vraiment manipuler des données !

### 2.6.7 Opérateurs de chaîne – introduction

Il est temps de revenir à ces deux opérateurs arithmétiques : + et \*.

Nous voulons vous montrer qu'ils ont une deuxième fonction. Ils sont capables de faire quelque chose de plus que simplement **ajouter** et **multiplier**.

Nous les avons vus en action où leurs arguments sont des nombres (flottants ou entiers, peu importe). Maintenant, nous allons vous montrer qu'ils peuvent également gérer les chaînes de caractères, bien que d'une manière très spécifique.

#### Concaténation

Le signe + (plus), lorsqu'il est appliqué à deux chaînes, devient un opérateur de **concaténation** :

chaîne + chaîne

Il **concatène** (colle) deux chaînes de caractères en une seule. Bien sûr, comme son frère arithmétique, il peut être utilisé plus d'une fois dans une expression, et dans un tel contexte, il se comporte selon la liaison à gauche.

Contrairement à son frère arithmétique, l'opérateur de concaténation **n'est pas commutatif**, c'est-à-dire que « ab » + « ba » n'est pas la même chose que « ba » + « ab ».

N'oubliez pas, si vous voulez que le signe + soit un **concaténeur**, pas un additionneur, vous devez vous assurer que **ses deux arguments sont des chaînes de caractères**.

Vous ne pouvez pas mélanger les types.

Ce programme simple montre le signe + dans sa deuxième utilisation:

```
fnam = input("Puis-je avoir votre prénom,svp? ")
lnam = input("Puis-je avoir votre nom,svp? ")
print("Merci.")
print("\nVotre nom est " + fnam + " " + lnam + ".")
```

Remarque : l'utilisation de + pour concaténer des chaînes vous permet de construire la sortie de manière plus précise qu'avec une fonction print() pure , même si elle est enrichie des arguments de mots-clés end= et sep=. Exécutez le code et voyez si la sortie correspond à vos prédictions.

### Réplication

Le signe \* (astérisque), lorsqu'il est appliqué à une chaîne et à un nombre (ou à un nombre et à une chaîne, car il reste commutatif) devient un opérateur de **réplication** :

```
chaîne * nombre  
nombre * chaîne
```

Il réplique la chaîne le même nombre de fois que spécifié par le nombre.

Par exemple :

« James » \* 3 donne « JamesJamesJames »

3 \* « an » donne « ananan »

5 \* « 2 » (ou « 2 » \* 5) donne « 22222 » (**pas 10!**)

**Remarque :** Un nombre inférieur ou égal à zéro produit une **chaîne vide**.

Ce programme simple « dessine » un rectangle, en utilisant un opérateur (+) dans un nouveau rôle:

```
print("+" + 10 * "-" + "+")  
print(("|" + " " * 10 + "|\\n") * 5, end="")  
print("+" + 10 * "-" + "+")
```

Notez la façon dont nous avons utilisé les parenthèses dans la deuxième ligne du code. Essayez de vous entraîner pour créer d'autres formes ou vos propres œuvres d'art!

### Conversion type str()

Vous savez déjà comment utiliser les fonctions **int()** et **float()** pour convertir une chaîne en nombre.

Ce type de conversion n'est pas une voie à sens unique. Vous pouvez également **convertir un nombre en chaîne**, ce qui est beaucoup plus facile et plus sûr - cette opération est toujours possible.

La fonction capable de le faire est appelée **str()**:

```
str(nombre)
```

Pour être honnête, il peut faire beaucoup plus que simplement transformer des nombres en chaînes, mais cela peut attendre plus tard.



## 2.6.8 exercices

## 1. Temps estimé

5-10 minutes

## Objectifs

- se familiariser avec la saisie et la sortie de données en Python;
- évaluer des expressions simples.

## Scénario

Votre tâche consiste à compléter le code afin d'évaluer les résultats de quatre opérations arithmétiques de base.

Les résultats doivent être imprimés sur la console.

Vous ne pourrez peut-être pas protéger le code d'un utilisateur qui souhaite diviser par zéro. Ce n'est pas grave, ne vous inquiétez pas pour l'instant.

Testez votre code : produit-il les résultats que vous attendez?

Nous ne vous montrerons aucune donnée de test : ce serait trop simple.

## Code source

```
# input a float value for variable a here
# input a float value for variable b here

# output the result of addition here
# output the result of subtraction here
# output the result of multiplication here
# output the result of division here

print("\nThat's all, folks!")
```

## 2 Temps estimé

20 minutes

### Objectifs

- se familiariser avec le concept de nombres, d'opérateurs et d'opérations arithmétiques en Python;
- comprendre la priorité et l'associativité des opérateurs Python, ainsi que l'utilisation correcte des parenthèses.

### Scénario

Votre tâche consiste à compléter le code afin d'évaluer l'expression suivante :

$$\frac{1}{x + \frac{1}{x + \frac{1}{x + \frac{1}{x}}}}$$

Le résultat doit être affecté à y. Soyez prudent, surveillez les opérateurs et gardez leurs priorités à l'esprit. N'hésitez pas à utiliser autant de parenthèses que nécessaire.

Vous pouvez utiliser des variables supplémentaires pour raccourcir l'expression (mais ce n'est pas nécessaire). Testez votre code avec soin.

### Code source

```
x = float(input("Entrer valeur pour x: "))  
  
# Write your code here.  
  
print("y =", y)
```

### Données de test

Exemple d'entrée : 1  
Résultats escomptés :  
y = 0,600000000000000001

Exemple d'entrée : 10  
Résultats escomptés :  
y = 0,09901951266867294

Exemple d'entrée : 100  
Résultats escomptés :  
y = 0,009999000199950014

Exemple d'entrée : -5  
Résultats escomptés :  
y = -0,19258202567760344

### 3 Temps estimé

15-20 minutes

### Objectifs

- améliorer la capacité d'utiliser des nombres, des opérateurs et des opérations arithmétiques en Python;
- utiliser les capacités de formatage de la fonction print();
- Apprendre à exprimer les phénomènes de la vie quotidienne en termes de langage de programmation.

### Scénario

Votre tâche consiste à préparer un code simple capable d'évaluer **l'heure de fin** d'une période, donnée en nombre de minutes (il pourrait être arbitrairement grand). L'heure de début est donnée sous forme de paire d'heures (0..23) et de minutes (0..59). Le résultat doit être imprimé sur la console.

Par exemple, si un événement commence à **12:17** et dure **59 minutes**, il se terminera à **13:16**.

Ne vous inquiétez pas des imperfections de votre code, ce n'est pas grave s'il accepte une heure non valide à ce moment de votre apprentissage, le plus important est que le code produise des résultats valides pour les données d'entrée valides.

Astuce : l'utilisation de l'opérateur % peut être la clé du succès.

### Données de test

Exemple d'entrée : 12 17 59  
Résultats escomptés : 13:16  
Exemple d'entrée : 23 58 642  
Résultats escomptés : 10:40  
Exemple d'entrée : 0 1 2939  
Produit attendu : 1:0

## 2.6.9 Résumé

1. La fonction `print()` **envoie des données à la console**, tandis que la fonction `input()` **obtient les données de la console**.

2. La fonction `input()` est livrée avec un paramètre facultatif : **la chaîne d'invitation à écrire**. Il vous permet d'écrire un message avant l'entrée de l'utilisateur, par exemple: `name = input("Entrez votre nom:")`

3. Lorsque la fonction `input()` est appelée, le flux du programme est arrêté, le symbole d'invite continue de clignoter (il invite l'utilisateur à prendre des mesures lorsque la console passe en mode d'entrée) jusqu'à ce que l'utilisateur ait entré une entrée et / ou appuyé sur la touche *Entrée*.

Astuce: la fonction mentionnée ci-dessus de la fonction `input()` peut être utilisée pour inviter l'utilisateur à mettre fin à un programme. Regardez le code ci-dessous :

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")

print("\nPress Enter to end the program.")
input()
print("THE END.")
```

4. Le résultat de la fonction `input()` est une chaîne. Vous pouvez ajouter des chaînes les unes aux autres à l'aide de l'opérateur de concaténation (+). Découvrez ce code :

```
num_1 = input("Enter the first number: ") # Enter 12
num_2 = input("Enter the second number: ") # Enter 21

print(num_1 + num_2) # the program returns 1221
```

5. Vous pouvez également répliquer (\*) des chaînes, par exemple :

```
my_input = input("Enter something: ") # Example input: hello
print(my_input * 3) # Expected output: hellohellohello
```

**Exercice 1**

```
x = int(input("Enter a number: ")) # The user enters 2
print(x * "5")
```

**Exercice 2**

Quelle est la sortie attendue de l'extrait de code suivant ?

```
x = input("Enter a number: ") # The user enters 2
print(type(x))
```

### 3. Valeurs booléennes et exécution conditionnelle

#### 3.1 Les comparateurs

##### 3.1.1 Questions et réponses

Un programmeur écrit un programme et **le programme pose des questions**.

Un ordinateur exécute le programme et **fournit les réponses**. Le programme doit pouvoir **réagir en fonction des réponses reçues**.

On se rappelle qu'on a vu plus tôt que les ordinateurs ne connaissent que deux types de réponses :

- Oui, c'est vrai;
- Non, c'est faux.

**Pour poser des questions, Python utilise un ensemble d'opérateurs très spéciaux.** Passons-les en revue l'un après l'autre, illustrant leurs effets sur quelques exemples simples.

##### 3.1.2 La comparaison ou l'opérateur d'égalité

Question : **deux valeurs sont-elles égales ?**

Pour poser cette question, vous utilisez l'opérateur `==` (égal égal).

N'oubliez pas cette distinction importante :

- `=` est un opérateur d'affectation, par exemple, `a = b` assigne `a` avec la valeur de `b`;
- `==` La question est-elle de *savoir si ces valeurs sont égales ?* ; `A == B` compare `A` et `B`.

C'est un opérateur binaire avec liaison côté gauche. Il a besoin de deux arguments et vérifie s'ils sont égaux.

**Question #1:** Quel est le résultat de la comparaison suivante?

`2 == 2`

**Question #2:** Quel est le résultat de la comparaison suivante?

`2 == 2.`

**Question #3:** Quel est le résultat de la comparaison suivante?

`1 == 2`

L'opérateur `==` (égal à) compare les valeurs de deux opérandes. S'ils sont égaux, le résultat de la comparaison est `True`. S'ils ne sont pas égaux, le résultat de la comparaison est `Faux`.

Regardez la comparaison d'égalité ci-dessous, quel est le résultat de cette opération?

```
var == 0
```

On aura deviné qu'il nous est impossible de trouver la réponse si nous ne savons pas quelle valeur est actuellement stockée dans la variable **var**.

Si la variable a été modifiée plusieurs fois au cours de l'exécution de votre programme, ou si sa valeur initiale est entrée à partir de la console, la réponse à cette question ne peut être donnée que par Python et uniquement au moment de l'exécution.

Maintenant, imaginez un programmeur qui souffre d'insomnie et doit compter les moutons noirs et blancs séparément jusqu'à ce qu'il y ait exactement deux fois plus de moutons noirs que de moutons blancs.

La question sera la suivante :

```
black_sheep == 2 * white_sheep
```

En raison de la faible priorité de l'opérateur `==`, la question doit être traitée comme équivalente à celle-ci :

```
black_sheep == (2 * white_sheep)
```

Alors, essayons maintenant de bien comprendre l'opérateur `==` : pouvez-vous deviner la sortie du code ci-dessous?

```
var = 0 # Affectation de 0 à var
print(var == 0)
```

```
var = 1 # Affectation de 1 à var
print(var == 0)
```

### 3.1.3 L'opérateur d'inégalité

L'opérateur `!=` (non égal à) compare également les valeurs de deux opérandes.

Voici la différence : s'ils sont égaux, le résultat de la comparaison est `Faux`. S'ils ne sont pas égaux, le résultat de la comparaison est `True`.

Jetez un coup d'œil à la comparaison des inégalités ci-dessous - pouvez-vous deviner le résultat de cette opération?

```
var = 0 # Affectation de 0 à var
print(var != 0)
```

```
var = 1 # Affectation de 1 à var
print(var != 0)
```

### 3.1.4 Opérateur de comparaison : supérieur à

Vous pouvez également poser une question de comparaison à l'aide de l'opérateur `>` (supérieur à).

Si vous voulez savoir s'il y a plus de moutons noirs que de moutons blancs, vous pouvez l'écrire comme suit:

```
black_sheep > white_sheep # Supérieur à
```

True le confirme; False le nie.

### 3.1.5 Opérateur de comparaison : supérieur ou égal à

L'opérateur « *supérieur à* » a une autre variante spéciale et **non stricte**, mais il est noté différemment de la notation arithmétique classique: `>=` (supérieur ou égal à).

Ces deux opérateurs (stricts et non stricts), ainsi que les deux autres discutés dans la section suivante, sont des **opérateurs binaires avec liaison à gauche**, et leur **priorité est supérieure à celle indiquée par `==` et `!=`**.

Si nous voulons savoir si nous devons ou non porter un chapeau chaud, nous posons la question suivante:

```
temperature_ext ≥ 0,0 # Supérieur ou égal à
```

### 3.1.6 Opérateurs de comparaison : inférieure à et inférieur ou égal à

Comme vous l'avez probablement déjà deviné, les opérateurs utilisés dans ce cas sont : l'opérateur `<` (inférieur à ) et son frère non strict : `<=` (inférieur ou égal à).

Regardez cet exemple simple :

```
vitesse_en_kmh < 90 # Moins de  
vitesse_en_kmh ≤ 90 # Inférieur ou égal à
```

Imaginons vouloir vérifier s'il y a un risque d'amende (la première question est stricte, la seconde ne l'est pas).

### 3.1.7 Que faire des réponses ?

Que pouvez-vous faire avec la réponse (c'est-à-dire le résultat d'une opération de comparaison) que vous obtenez de l'ordinateur?

Il y a au moins deux possibilités: d'abord, vous pouvez le mémoriser (**le stocker dans une variable**) et l'utiliser plus tard. Comment procédez-vous? Eh bien, vous utiliseriez une variable arbitraire comme celle-ci:

```
reponse = nbr_d_etudiants >= nbr_de_profs
```

Le contenu de la variable vous indiquera la réponse à la question posée.

La deuxième possibilité est plus pratique et beaucoup plus courante: vous pouvez utiliser la réponse que vous obtenez pour **prendre une décision sur l'avenir du programme**.

Vous avez besoin d'une instruction spéciale à cet effet, et c'est ce que nous allons faire juste après.

Nous devons maintenant mettre à jour notre **tableau des priorités** et y intégrer tous les nouveaux opérateurs. Il se présente maintenant comme suit :

Priorité	Opérateur	
1	+, -	unaire
2	**	
3	*, /, //, %	
4	+, -	binaire
5	<, <=, >, >=	
6	==, !=	

### 3.1.8 Exercice

#### Temps estimé

5-10 minutes

#### Objectifs

- se familiariser avec la fonction `input()`;
- se familiariser avec les opérateurs de comparaison en Python.

#### Scénario

À l'aide de l'un des opérateurs de comparaison en Python, écrivez un programme simple de deux lignes qui prend le paramètre `n` comme entrée, qui est un entier, et imprime `False` si `n` est inférieur à 100 et `True` si `n` est supérieur ou égal à 100.



Ne créez pas de blocs if (vous ne savez pas ce que c'est ! Bien, vous le savez ? faites comme si vous ne le saviez pas ). Testez votre code à l'aide des données suivantes :

## Données de test

Exemple d'entrée : 55

Résultat attendu : Faux

Exemple d'entrée : 99

Résultat attendu : Faux

Exemple d'entrée : 100

Résultat attendu : Vrai

Exemple d'entrée : 101

Résultat attendu : Vrai

Exemple d'entrée : -5

Résultat attendu : Faux

Exemple d'entrée : +123

Résultat attendu : Vrai

## 3.2 Prendre des décisions en Python

### 3.2.1 Les conditions et leur exécution

Vous savez déjà comment poser des questions en Python, mais vous ne savez toujours pas comment faire un usage raisonnable des réponses. Vous devez avoir un mécanisme qui vous permettra de faire quelque chose si **une condition est remplie, et de ne pas le faire si ce n'est pas le cas**.

C'est comme dans la vraie vie : vous faites certaines choses ou vous ne le faites pas lorsqu'une condition spécifique est remplie ou non, par exemple, vous allez vous promener s'il fait beau, ou restez à la maison s'il fait humide et froid.

Pour prendre de telles décisions, Python propose une instruction spéciale. En raison de sa nature et de son application, on l'appelle une instruction conditionnelle (on dira instruction ou statement en anglais).

Il en existe plusieurs variantes. Nous allons commencer par le plus simple, en augmentant lentement la difficulté vers les cas complexes.

La première forme d'une déclaration conditionnelle, que vous pouvez voir ci-dessous, est écrite de manière très informelle mais figurative:

```
if vrai_ou_faux:
    faire_ceci_si_vrai
```

Cette instruction conditionnelle se compose des éléments suivants, strictement nécessaires, dans cet ordre et dans celui-ci uniquement :

- le mot-clé **if** ;
- un ou plusieurs espaces blancs;
- une expression (une question ou une réponse) dont la valeur sera interprétée uniquement en termes de `Vrai` (lorsque sa valeur est non nulle) et de Faux (lorsqu'elle est égale à zéro);
- un **signe deux-points** suivi d'une nouvelle ligne;
- une instruction en retrait ou un ensemble d'instructions (au moins une instruction est absolument requise); l'indentation peut être obtenue de deux manières: en insérant un nombre particulier d'espaces (il est recommandé d'utiliser quatre espaces d'indentation) ou en utilisant le caractère de *tabulation*; note: s'il y a plus **d'une** instruction dans la partie **en retrait**, l'indentation doit être la même dans toutes les lignes;

Comment cette déclaration fonctionne-t-elle?

- Si l'expression `vrai_ou_faux` représente la vérité (c'est-à-dire que sa valeur n'est pas égale à zéro), **la ou les instructions en retrait seront exécutées** ;
- Si l'expression `vrai_ou_faux` **ne représente pas la vérité** (c'est-à-dire que sa valeur est égale à zéro), **la ou les instructions en retrait seront omises** (ignorées) et l'instruction suivante exécutée sera celle qui suivra le niveau d'indentation d'origine.

Dans la vraie vie, nous exprimons souvent un désir comme ceci :

*S'il fait beau, nous irons nous promener*

*Ensuite, nous déjeunerons*

Comme vous pouvez le constater, déjeuner **n'est pas une activité** conditionnelle et ne dépend pas de la météo.

En sachant quelles conditions influencent notre comportement, et en supposant que nous avons les fonctions sans paramètre `allons_marcher()` et `manger()`, nous pouvons écrire l'extrait suivant:

```
if la_meteo_est_bonne:
    aller_marcher()
manger()
```

### 3.2.2 La condition – if

Si un certain étudiant développeur Python sans sommeil s'endort alors qu'il compte 120 moutons, et que la procédure induisant le sommeil peut être implémentée comme une fonction spéciale nommée `sleep_and_dream()`, le code entier de celle-ci prendrait la forme suivante:

```
if sheep_counter >= 120: # Évaluer une expression de test
    sleep_and_dream() # Exécuter si l'expression de test est True
```

Vous pouvez lire ce code comme: si `sheep_counter` est supérieur ou égal à 120, alors endormez-vous et rêvez (exécutez la fonction `sleep_and_dream`).

Nous avons dit que les **déclarations exécutées sous condition doivent être mises en retrait**. Cela crée une structure très lisible, démontrant clairement tous les chemins d'exécution possibles dans le code et c'est une des grandes forces de Python.

Jetez un coup d'œil au code suivant :

```
if sheep_counter >= 120:
    make_a_bed()
    take_a_shower()
    sleep_and_dream()
feed_the_sheepdogs()
```

Comme vous pouvez le voir, faire un lit, prendre une douche, s'endormir et rêver sont tous **exécutés de manière conditionnelle** lorsque `sheep_counter` atteint la limite souhaitée.

L'alimentation des chiens de berger, cependant, est toujours **effectuée** (c'est-à-dire que la fonction `feed_the_sheepdogs()` n'est pas indentée et n'appartient pas au bloc if, ce qui signifie qu'elle est toujours exécutée.)

### 3.2.3 La condition – if-else

Nous avons commencé par une phrase simple qui disait : *S'il fait beau, nous irons nous promener*.

Note : il n'y a pas d'explication sur ce qui se passe si le temps est mauvais. Nous savons seulement que nous n'irons pas à l'extérieur, mais ce que nous ferions à la place n'est pas connu. Nous pouvons également planifier quelque chose en cas de mauvais temps.

Nous pouvons dire, par exemple: *s'il fait beau, nous irons nous promener, sinon nous irons jouer à la console*.

Maintenant, nous savons ce que nous ferons si **les conditions sont remplies**, et nous savons ce que nous ferons **si tout ne se passe pas comme nous le voulons**. En d'autres termes, nous avons un « plan B ».

Python nous permet d'exprimer de tels plans alternatifs. Cela se fait avec une deuxième forme, légèrement plus complexe, de l'instruction conditionnelle, l'instruction *if-else*:

```
if true_or_false_condition:
    perform_if_condition_true
else:
    perform_if_condition_false
```

Ainsi, il y a un nouveau mot: `else` → c'est un mot-clé (comme `if`).

La partie du code qui commence par `else` indique quoi faire si la condition spécifiée pour le `if` n'est pas remplie (notez les **deux-points** après le mot).

L'exécution *if-else* se déroule comme suit :

- si la condition a la valeur **True** (sa valeur n'est pas égale à zéro), l'instruction `perform_if_condition_true` est exécutée et l'instruction conditionnelle prend fin ;
- si la condition a la valeur **False** (elle est égale à zéro), l'instruction `perform_if_condition_false` est exécutée et l'instruction conditionnelle prend fin.

### 3.2.4 L'imbrication sorcellerie ou réalité ?

Tout d'abord, considérons le cas où **l'instruction placée après le `if` est un autre `if`**.

Analysons un petit texte : S'il fait beau, nous irons nous promener. Si nous trouvons un bon restaurant, nous y déjeunerons. Sinon, nous mangerons un sandwich. Si le temps est mauvais, nous irons au théâtre. S'il n'y a pas de billets, nous irons faire du shopping dans le centre commercial le plus proche.

Écrivons la même chose en Python :

```
if the_weather_is_good:
    if nice_restaurant_is_found:
        have_lunch()
    else:
        eat_a_sandwich()
else:
    if tickets_are_available:
        go_to_the_theater()
    else:
        go_shopping()
```

Voici deux points importants :

- Cette utilisation de l'instruction `if` est connue sous le nom **d'imbrication**; rappelez-vous que `else` fait référence au `if` qui se trouve **au même niveau d'indentation**; vous devez le savoir pour déterminer comment les `if` et `else` s'apparient;
- Regardez comment **l'indentation améliore la lisibilité** et rend le code plus facile à comprendre et à suivre.

### 3.2.5 La condition – elif

Le deuxième cas particulier introduit un autre nouveau mot-clé Python : **elif**. Comme vous vous en doutez probablement, c'est une forme plus courte pour le terme **else if**.

**elif** est utilisé pour **vérifier plus d'une condition**, et pour s'arrêter lorsque la première déclaration qui est vraie est trouvée.

Notre exemple s'exprimera comme suit:

S'il fait beau, nous irons nous promener, sinon si nous obtenons des billets, nous irons au théâtre, sinon s'il y a des tables libres au restaurant, nous irons déjeuner; Si tout le reste échoue, nous retournerons à la maison et jouerons aux échecs.

Avez-vous remarqué combien de fois nous avons utilisé le mot *Sinon*? C'est l'étape où le mot-clé **elif** joue son rôle.

Écrivons le même scénario en utilisant Python :

```
if the_weather_is_good:
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_theater()
elif table_is_available:
    go_for_lunch()
else:
    play_chess_at_home()
```

La façon d'assembler les instructions *if-elif-else* suivantes est parfois appelée **cascade**.

Remarquez à nouveau comment l'indentation améliore la lisibilité du code (Oui on insiste sur ce point mais c'est important car cela n'est pas obligatoire dans tous les langages mais ça le devrait).

Une attention supplémentaire doit être accordée dans ce cas:

- Vous **ne devez pas utiliser** **else** **sans un** **if**;
- **else** est toujours le **dernière branche de la cascade**, que vous ayez utilisé **elif** ou pas;
- **else** est une partie **facultative** de la cascade et peut être omis;
- S'il n'y a pas de branche **else**, il est possible qu'aucune des branches disponibles ne soit exécutée.

Cela peut sembler un peu déroutant, mais j'espère que quelques exemples simples aideront à faire la lumière sur ceci.

### 3.2.6 Analyse des exemples de code

Maintenant, nous allons vous montrer quelques programmes simples mais complets. Nous ne les expliquerons pas en détail, car nous considérons les commentaires (et les noms de variables) à l'intérieur du code comme des informations suffisantes.

Tous les programmes résolvent le même problème, ils **trouvent le plus grand nombre et l'impriment**.

#### Exemple 1 :

Nous allons commencer par le cas le plus simple, **comment identifier le plus grand des deux nombres**:

```
# Read two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))

# Choose the larger number
if number1 > number2:
    larger_number = number1
else:
    larger_number = number2
# Print the result
print("The larger number is:", larger_number)
```

L'extrait ci-dessus doit être clair, il lit deux valeurs entières, les compare et trouve laquelle est la plus grande.

#### Exemple 2 :

Voici le même exemple mais avec une astuce de programmeur :

```
# Read two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))

# Choose the larger number
if number1 > number2: larger_number = number1
else: larger_number = number2

# Print the result
print("The larger number is:", larger_number)
```

Remarque: si l'une des branches *if-elif-else* ne contient qu'une seule instruction, vous pouvez la coder sous une forme plus simplifiée (vous n'avez pas besoin de faire une ligne en retrait après le mot-clé, mais continuez simplement la ligne après les deux-points).

Ce style, cependant, peut être trompeur, et nous n'allons pas l'utiliser dans nos futurs programmes, mais il vaut vraiment la peine de savoir si vous voulez lire et comprendre les programmes de quelqu'un d'autre. Et vous l'utiliserez sûrement plus tard.

**Exemple 3 :**

Il est temps de compliquer le code, trouvons le plus grand des trois nombres. Nous supposons que la première valeur est la plus grande. Ensuite, nous vérifions cette hypothèse avec les deux valeurs restantes :

```
# Read three numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))

# We temporarily assume that the first number
# is the largest one.
# We will verify this soon.
largest_number = number1

# We check if the second number is larger than current largest_number
# and update largest_number if needed.
if number2 > largest_number:
    largest_number = number2

# We check if the third number is larger than current largest_number
# and update largest_number if needed.
if number3 > largest_number:
    largest_number = number3

# Print the result
print("The largest number is:", largest_number)
```

Cette méthode est beaucoup plus simple que d'essayer de trouver le plus grand nombre en comparant toutes les paires de nombres possibles (c.-à-d. premier avec deuxième, deuxième avec troisième, troisième avec premier).

### 3.2.7 Pseudo-code et introduction aux boucles

Vous devriez maintenant être capable d'écrire un programme qui trouve le plus grand des quatre, cinq, six ou même dix nombres.

Vous connaissez déjà le schéma, donc étendre l'ampleur du problème ne sera pas particulièrement complexe.

Mais que se passe-t-il si nous vous demandons d'écrire un programme qui trouve le plus grand de deux cents nombres? Pouvez-vous imaginer le code? Vous aurez besoin de deux cents variables. Si deux cents variables ne suffisent pas, essayez d'imaginer la recherche du plus grand nombre d'un million.

Imaginez un code qui contient 199 instructions conditionnelles et deux cents appels de la fonction `input()`. Heureusement, vous n'avez pas besoin de faire face à cela. Il existe une approche plus simple.



Nous allons ignorer les exigences de la syntaxe Python pour l'instant, et essayer d'analyser le problème sans penser à la vraie programmation. En d'autres termes, nous essaierons d'écrire **l'algorithme**, oui ce vilain truc donc vous parle le prof de théorie régulièrement, et quand nous en serons satisfaits, nous l'implémenterons.

Dans ce cas, nous utiliserons une sorte de notation qui n'est pas un langage de programmation réel (il ne peut être ni compilé ni exécuté), mais il est formalisé, concis et lisible. C'est ce qu'on appelle **du pseudo-code**.

Regardons notre pseudo-code ci-dessous:

```
largest_number = -999999999
number = int(input())
if number == -1:
    print(largest_number)
    exit()
if number > largest_number:
    largest_number = number
# Go to line 02
```

Que se passe-t-il là-dedans?

Tout d'abord, nous avons attribuer à la variable `largest_number` une valeur qui sera inférieure à n'importe lequel des nombres saisis. Nous utiliserons -999999999 pour cela.

Deuxièmement, nous supposons que notre algorithme ne saura pas à l'avance combien de numéros seront livrés au programme. Nous nous attendons à ce que l'utilisateur entre autant de nombres qu'il le souhaite, l'algorithme fonctionnera bien avec cent nombres ou avec mille nombres. Mais comment faire ?

Nous concluons un accord avec l'utilisateur: lorsque la valeur -1 est entrée, ce sera un signe qu'il n'y a plus de données et que le programme devrait terminer son travail.

Sinon, si la valeur entrée n'est pas égale à -1, le programme lira un autre nombre, et ainsi de suite.

L'astuce est basée sur l'hypothèse que n'importe quelle partie du code peut être exécutée plus d'une fois, et précisément, autant de fois que nécessaire.



L'exécution d'une certaine partie du code plus d'une fois est appelée **boucle**. Les lignes 02 à 08 font une boucle. Nous les parcourrons **autant de fois que nécessaire** pour examiner toutes les valeurs saisies. C'est à cet égard que notre commentaire existe. Oui nous ne savons pas encore faire de boucles. Infos supplémentaires :

Python est livré avec beaucoup de fonctions intégrées qui feront le travail pour vous. Par exemple, pour trouver le plus grand nombre de tous, vous pouvez utiliser une fonction intégrée Python appelée `max()`. Vous pouvez l'utiliser avec plusieurs arguments. Analysez le code ci-dessous :

```
# Read three numbers.
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))

# Check which one of the numbers is the greatest
# and pass it to the largest_number variable.
largest_number = max(number1, number2, number3)

# Print the result.
print("The largest number is:", largest_number)
```

De la même manière, vous pouvez utiliser la fonction `min()` pour renvoyer le nombre le plus bas.

Nous allons bientôt parler de ces fonctions (et de bien d'autres). Pour le moment, nous nous concentrerons sur l'exécution conditionnelle et les boucles pour vous permettre de gagner plus de confiance en programmation et vous enseigner les compétences qui vous permettront de comprendre et d'appliquer pleinement les deux concepts de votre code. Nous n'avons donc pas pris de raccourcis dans ce but, mais en tant que bon programmeur, il faudra que vous appreniez à le faire.

## 3.2.8 Exercices

## Temps Estimé

5-15 minutes

## Objectifs

- se familiariser avec la fonction `input()`;
- se familiariser avec les opérateurs de comparaison en Python;
- se familiariser avec le concept d'exécution conditionnelle.

## Scénario

Mr Depreter communément connu sous le nom de Er-Han ou Apprentis des ténèbres, est l'un des professeurs les plus populaires car il apporte connaissances, joies, rires et affonds dans l'école.

Imaginez que votre programme informatique aime ce professeur. Chaque fois qu'il reçoit une entrée sous la forme du mot Depreter, il crie involontairement à la console la chaîne suivante: « Mr Depreter est le meilleur professeur de tous les temps! »

Ecrivez un programme qui utilise le concept d'exécution conditionnelle, prend une chaîne comme entrée et :

- Imprime la phrase « Oui – Mr Depreter est le meilleur professeur de tous les temps! » à l'écran si la chaîne saisie est « Depreter » (Majuscule)
- Imprime « Non, je veux le vrai Mr Depreter! » si la chaîne saisie est « depreter » (minuscule)
- Imprime «Mr Depreter! Pas [input]! » dans les autres cas. Remarque : [input] est la chaîne prise comme entrée.

## 2 Temps estimé

10-20 minutes

## Objectifs

Familiariser l'élève avec :

- l'utilisation de l'instruction *if-e/se* pour brancher le chemin de contrôle ;
- Construire un programme complet qui résout des problèmes simples de la vie réelle.

## Scénario

Il était une fois une terre, une terre de bières et de chips, habitée par des gens heureux et prospères. Les gens payaient des impôts, bien sûr, leur bonheur avait des limites. L'impôt le plus important, appelé *impôt sur le revenu des particuliers (IRPP)* devait être payé une fois par an et était évalué selon la règle suivante:

- Si le revenu du citoyen ne dépassait pas 85 528 Funcoins, l'impôt était égal à 18% du revenu moins 556 Funcoins et 2 cents (c'était le soi-disant *allégement fiscal*)
- Si le revenu était supérieur à ce montant, l'impôt était égal à 14 839 Funcoins et 2 cents, plus 32% de l'excédent sur 85 528 Funcoins.

Votre tâche consiste à rédiger un **calculateur d'impôt**.

- Il devrait accepter une valeur à virgule flottante : le revenu.
- Ensuite, il devrait imprimer la taxe calculée, arrondie aux Funcoins complets. Il y a une fonction nommée `round()` qui fera l'arrondi pour vous

Note : ce pays heureux ne rend jamais d'argent à ses citoyens. Si la taxe calculée est inférieure à zéro, cela signifie seulement qu'il n'y a pas d'impôt du tout (la taxe est égale à zéro). Tenez-en compte lors de vos calculs.

Regardez le code ci-dessous de notre ancien informaticien, il ne lit qu'une seule valeur d'entrée et génère un résultat, vous devez donc le compléter avec des calculs intelligents.

Testez votre code à l'aide des données que nous avons fournies.

## Données de test

Exemple d'entrée: 10000 Résultat attendu: La taxe est: 1244.0 Funcoins

Exemple d'entrée : 100000 Résultat attendu: La taxe est: 19470.0 Funcoins

Exemple d'entrée: 1000 Résultat attendu : La taxe est de : 0,0 Funcoins

Exemple d'entrée : -100 Résultat attendu : La taxe est de : 0,0 Funcoins

## Code source :

```
income = float(input("Enter the annual income: "))

#
# Write your code here.

tax = round(tax, 0)
print("The tax is:", tax, "Funcoins")
```

### 3 Temps estimé

10-25 minutes

### Objectifs

Familiariser l'élève avec :

- à l'aide de l'instruction if-elif-else;
- trouver la bonne mise en œuvre des règles définies verbalement;
- Test du code à l'aide d'exemples d'entrée et de sortie.

### Scénario

Comme vous le savez sûrement, pour des raisons astronomiques, les années peuvent être *bissextiles* ou *courantes*. Les premiers durent 366 jours, tandis que les seconds durent 365 jours.

Depuis l'introduction du calendrier grégorien (en 1582), la règle suivante est utilisée pour déterminer le type d'année:

- Si le numéro de l'année n'est pas divisible par quatre, il s'agit d'une *année commune*;
- sinon, si le numéro de l'année n'est pas divisible par 100, il s'agit d'une *année bissextile*;
- sinon, si le numéro de l'année n'est pas divisible par 400, il s'agit d'une *année commune*;
- Sinon, c'est une *année bissextile*.

Notre ancien informaticien nous a juste fourni ce bout de code :

```
year = int(input("Enter a year: "))  
  
# Write your code here.
```

Le code doit générer l'un des deux messages possibles, qui sont Année bissextile ou Année commune, selon la valeur entrée.

Il serait bon de vérifier si l'année saisie tombe dans l'ère grégorienne, et d'émettre un avertissement sinon: Pas dans la période du calendrier grégorien. Conseil : utilisez les opérateurs != et %. Testez votre code à l'aide des données que nous avons fournies.

### Données de test

Exemple d'entrée : 2000 Résultats escomptés : Année bissextile

Exemple d'entrée : 2015 Résultats escomptés: Année commune

Exemple d'entrée : 1999 Résultats escomptés: Année commune

Exemple d'entrée : 1996 Résultats escomptés : Année bissextile

Exemple d'entrée : 1580 Résultats escomptés : Non compris dans la période du calendrier grégorien

## 3.2.9 Résumé

1. Les opérateurs **de comparaison** (ou les opérateurs dits relationnels) sont utilisés pour comparer les valeurs. Le tableau ci-dessous illustre le fonctionnement des opérateurs de comparaison, en supposant que  $x = 0$ ,  $y = 1$  et  $z = 0$  :

Opérateur	Description	Exemple
<code>==</code>	renvoie si les valeurs des opérandes sont égales et Faux dans le cas contraire.	<code>x == et # Faux</code> <code>x == avec # Vrai</code>
<code>!=</code>	renvoie Vrai si les valeurs des opérandes ne sont pas égales et Faux dans le cas contraire.	<code>x != et # Vrai</code> <code>x != avec # Faux</code>
<code>&gt;</code>	Vrai si la valeur de l'opérande gauche est supérieure à celle de l'opérande droit, et Faux dans le cas contraire	<code>x &gt; et # Faux</code> <code>et &gt; avec # Vrai</code>
<code>&lt;</code>	True si la valeur de l'opérande gauche est inférieure à la valeur de l'opérande droit, et Faux dans le cas contraire	<code>x &lt; et # Vrai</code> <code>et &lt; avec # Faux</code>
<code>&gt;=</code>	True si la valeur de l'opérande gauche est supérieure ou égale à la valeur de l'opérande droit, et Faux dans le cas contraire	<code>x &gt;= et # Faux</code> <code>x &gt;= avec # Vrai</code> <code>et &gt;= avec # Vrai</code>
<code>&lt;=</code>	True si la valeur de l'opérande gauche est inférieure ou égale à la valeur de l'opérande droit, et Faux dans le cas contraire	<code>x &lt;= et # Vrai</code> <code>x &lt;= avec # Vrai</code> <code>et &lt;= avec # Faux</code>

2. Lorsque vous souhaitez exécuter du code uniquement si une certaine condition est remplie, vous pouvez utiliser une **instruction conditionnelle** :

- une seule instruction if, par exemple :

```
x = 10
if x == 10: # condition
    print("x is equal to 10") # Executed if the condition is True.
```

- Une série de if, par exemple :

```
x = 10
if x > 5: # condition one
    print("x is greater than 5") # Executed if condition one is True.
if x < 10: # condition two
    print("x is less than 10") # Executed if condition two is True.
if x == 10: # condition three
    print("x is equal to 10") # Executed if condition three is True.
```

Chaque instruction if est testée séparément.

- Une instruction if-else, par exemple :

```
x = 10
if x < 10: # Condition
    print("x is less than 10") # Executed if the condition is True.
else:
    print("x is greater than or equal to 10") # Executed if the condition
is False.
```

- une série d'instructions `if` suivies d'un `else`, par exemple:

```
x = 10
if x > 5: # True
    print("x > 5")
if x > 8: # True
    print("x > 8")
if x > 10: # False
    print("x > 10")
else:
    print("else will be executed")
```

Chaque `if` est testé séparément. Le corps du `else` est exécuté si le dernier `if` est Faux.

- L'instruction `if-elif-else`, par exemple :

```
x = 10
if x == 10: # True
    print("x == 10")
if x > 15: # False
    print("x > 15")
elif x > 10: # False
    print("x > 10")
elif x > 5: # True
    print("x > 5")
else:
    print("else will not be executed")
```

Si la condition du `if` est Faux, le programme vérifie les conditions des blocs `elif` suivants - le premier bloc `elif` qui est True est exécuté. Si toutes les conditions sont Faux, le bloc `else` sera exécuté.

- Instructions conditionnelles imbriquées, par exemple :

```
x = 10
if x > 5: # True
    if x == 6: # False
        print("nested: x == 6")
    elif x == 10: # True
        print("nested: x == 10")
    else:
        print("nested: else")
else:
    print("else")
```

### Exercice 1

Quelle est la sortie de l'extrait de code suivant ?

```
x = 5
y = 10
z = 8
print(x > y)
print(y > z)
```

**Exercice 2**

Quelle est la sortie de l'extrait de code suivant ?

```
x, y, z = 5, 10, 8
print(x > z)
print((y - 5) == x)
```

**Exercice 3**

Quelle est la sortie de l'extrait de code suivant ?

```
x, y, z = 5, 10, 8
x, y, z = z, y, x
print(x > z)
print((y - 5) == x)
```

**Exercice 4**

Quelle est la sortie de l'extrait de code suivant ?

```
x = 10
if x == 10:
    print(x == 10)
if x > 5:
    print(x > 5)
if x < 10:
    print(x < 10)
else:
    print("else")
```

**Exercice 5**

Quelle est la sortie de l'extrait de code suivant ?

```
x = "1"
if x == 1:
    print("one")
elif x == "1":
    if int(x) > 1:
        print("two")
    elif int(x) < 1:
        print("three")
    else:
        print("four")
if int(x) == 1:
    print("five")
else:
    print("six")
```

**Exercice 6**

Quelle est la sortie de l'extrait de code suivant ?

```
x = 1
y = 1.0
z = "1"
if x == y:
    print("one")
if y == int(z):
    print("two")
elif x == y:
    print("three")
else:
    print("four")
```

## 4. Les boucles

### 4.1 La boucle while

Êtes-vous d'accord avec le texte présenté ci-dessous?

```
while there is something to do
    do it
```

Notez aussi que ce code déclare également que s'il n'y a rien à faire, rien du tout ne se passera.

En général, en Python, une boucle peut être représentée comme suit :

```
while conditional_expression:
    instruction
```

Si vous remarquez des similitudes avec l'instruction *if*, c'est très bien. En effet, la différence syntaxique est simple : vous utilisez le mot `while` au lieu du mot `if`.

La différence sémantique est plus importante : lorsque la condition est remplie, *if* n'effectue ses instructions **qu'une seule fois** ; *pendant que while l'exécute tant que la condition à la valeur Vrai*.

Remarque: toutes les règles concernant **l'indentation** sont également applicables ici.

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    instruction_n
```

Il est important de se rappeler que :

- Si vous souhaitez exécuter **plus d'une instruction dans un** même `while`, vous devez (comme avec `if`) **mettre en retrait** toutes les instructions de la même manière ;
- Une instruction ou un ensemble d'instructions exécutées à l'intérieur de la boucle `while` est appelé corps de la **boucle**;
- Si la condition est Faux (égale à zéro) dès le premier test, le corps n'est pas exécuté une seule fois
- Le corps devrait être capable de changer la valeur de la condition, car si la condition est vraie au début, le corps peut être appelé à l'infini



#### 4.1.1 La boucle infinie

Une boucle infinie, également appelée **boucle sans fin**, elle est une séquence d'instructions dans un programme qui se répètent indéfiniment.

Voici un exemple de boucle qui ne parvient pas à terminer son exécution :

```
while True:
    print("Je suis coincé dans une boucle. " )
```

Cette boucle imprimera à l'infini « Je suis coincé à l'intérieur d'une boucle », à l'écran.

Revenons à l'esquisse de l'algorithme que nous vous avons montré ci-dessus.

Nous allons vous montrer comment utiliser cette boucle nouvellement apprise pour trouver le plus grand nombre à partir d'un grand ensemble de données saisies.

Analysez attentivement le programme. Voyez où commence la boucle (ligne 8).

Localisez le corps de la boucle et découvrez **comment sortir de la boucle**:

```
# Stockez le plus grand nombre actuel ici.
largest_number = -999999999

# Entrez la première valeur.
number = int(input("Enter a number or type -1 to stop: "))

# Si le nombre différent de -1 on continue.
while number != -1:
    # Le nombre est-il plus grd que largest_number ?
    if number > largest_number:
        # Oui, on mets à jour.
        largest_number = number
    # Entrer un autre nombre
    number = int(input("Enter a number or type -1 to stop: "))

# Afficher le largest_number
print("The largest number is:", largest_number)
```

#### 4.1.2 Exemple bonus

Regardons un autre exemple utilisant la boucle while.  
Suivez les commentaires pour découvrir l'idée et la solution.

```
# Un programme qui lit une séquence de nombres
# et compte combien de nombres sont pairs et combien sont impairs.
# Le programme se termine lorsque zéro est entré.
odd_numbers = 0
even_numbers = 0

# Lisez le premier numéro.
number = int(input("Enter a number or type 0 to stop: "))

# 0 termine la boucle.
while number != 0:
    # Vérifiez si le nombre est impair.
    if number % 2 == 1:
        # Augmentez le compteur odd_numbers.

        odd_numbers += 1
    else:
        #Augmentez le compteur even_numbers.
        even_numbers += 1

    # Lisez le numéro suivant.
    number = int(input("Enter a number or type 0 to stop: "))

# Imprimer les résultats.
print("Odd numbers count:", odd_numbers)
print("Even numbers count:", even_numbers)
```

Certaines expressions peuvent être simplifiées sans modifier le comportement du programme.

Essayez de vous rappeler comment Python interprète la vérité d'une condition, et notez que ces deux formes sont équivalentes:

```
while number != 0: et while number:
```

La condition qui vérifie si un nombre est impair peut également être codée sous ces formes équivalentes:

```
if nom % 2 == 1: et if le nom %2 :
```

#### 4.1.3 Utilisation d'un compteur dans une boucle

Regardez l'extrait ci-dessous:

```
counter = 5
while counter != 0:
    print ("À l'intérieur de la boucle.", counter)
    counter -= 1
print("En dehors de la boucle.", counter)
```

Ce code est destiné à imprimer la chaîne « À l'intérieur de la boucle ». et la valeur stockée dans la variable compteur au cours d'une boucle donnée exactement cinq fois.

Une fois que la condition n'a pas été remplie ( la variable compteur a atteint 0), la boucle est fermée et le message « En dehors de la boucle ». ainsi que la valeur stockée dans le compteur sont imprimés.

Mais il y a une chose qui peut être écrite de manière plus compacte, c'est l'état de la boucle while.

Voyez-vous la différence?

```
counter = 5
while counter:
    print("À l'intérieur de la boucle.", counter)
    counter -= 1
print("En dehors de la boucle.", counter)
```

Ce code est-il plus compact qu'auparavant ? Un peu, oui. Est-il plus lisible pour autant ? C'est discutable.

Important :

Ne vous sentez pas obligé de coder vos programmes d'une manière qui est toujours la plus courte et la plus compacte. La lisibilité peut être un facteur plus important. Gardez votre code prêt pour un nouveau programmeur.

## 4.1.4 Exercice

## Temps estimé

15 minutes

## Objectifs

Familiariser l'élève avec :

- en utilisant la boucle pendant que;
- reflétant des situations réelles dans le code informatique.

## Scénario

Un magicien junior a choisi un numéro secret. Il l'a caché dans une variable nommée `secret_number`. Il veut que tous ceux qui exécutent son programme jouent au jeu **Devinez le nombre secret**, et devinent quel numéro il a choisi pour eux. Ceux qui ne devinent pas le nombre seront coincés dans une boucle sans fin pour toujours! Malheureusement, il ne sait pas comment compléter le code, il est magicien pas développeur, le bougre.

Votre tâche est d'aider le magicien à compléter le code dans l'éditeur de manière à ce que le code:

- demandera à l'utilisateur d'entrer un nombre entier;
- utilisera une boucle **while**;
- vérifiera si le numéro saisi par l'utilisateur est le même que le numéro choisi par le magicien. Si le numéro choisi par l'utilisateur est différent du numéro secret du magicien, l'utilisateur devrait voir le message « Ha ha! Tu es coincé dans ma boucle! » et être invité à entrer à nouveau un numéro. Si le numéro entré par l'utilisateur correspond au numéro choisi par le magicien, le numéro doit être imprimé à l'écran et le magicien doit dire les mots suivants: « Bien joué, novice! Tu es libre maintenant. Ton savoir s'approche de celui de Maître Mandoux. »

•

Le magicien compte sur vous ! Ne le décevez pas.

## Code de base

```
secret_number = 777
print(
    """
+=====+
| Welcome to my game, muggle!    |
| Enter an integer number        |
| and guess what number I've     |
| picked for you.                |
| So, what is the secret number? |
+=====+
    """)
```

Tips : Regarde on a su imprimer sur plusieurs lignes :D quand on parlait de magicien...

## 4.2 La boucle for

Un autre type de boucle disponible en Python vient de l'observation qu'il est parfois plus important de **compter les « tours » de la boucle que de vérifier des conditions**.

Imaginez que le corps d'une boucle doive être exécuté exactement cent fois. Si vous souhaitez utiliser la boucle while pour le faire, on aura quelque chose comme:

```
i = 0
while i < 100:
    # do_something()
    i += 1
```

Ce n'est quand même pas super pratique de devoir créer son propre compteur... Et bien sûr que c'est le cas, il y a une boucle spéciale pour ce genre de tâches, et elle porte le nom de boucle for.

En fait, la boucle for est conçue pour effectuer des tâches plus complexes, **elle peut « parcourir » de grandes collections de données élément par élément**.

### 4.2.1 La boucle for et son fonctionnement

Jetez un coup d'œil au code suivant :

```
for i in range(100):
    # do_something()
    pass
```

Il y a de nouveaux éléments :

- Le mot-clé for ouvre la boucle for ; Remarque : il n'y a pas de condition après ; vous n'avez pas à penser aux conditions, car elles sont vérifiées en interne, sans aucune intervention de votre part;
- La variable après le mot-clé for est la **variable de contrôle** de la boucle ; elle compte les tours de la boucle et cela automatiquement;
- Le mot-clé in introduit un élément de syntaxe décrivant la plage de valeurs possibles affectées à la variable de contrôle ;
- La fonction range() est responsable de la génération de toutes les valeurs souhaitées de la variable de contrôle; dans notre exemple, la fonction va créer (on peut même dire qu'elle alimentera le boucle avec) des valeurs ultérieures à partir de l'ensemble suivant: 0, 1, 2 .. 97, 98, 99; Remarque: dans ce cas, la fonction range() commence son travail à partir de 0 et le termine une étape (un nombre entier) avant la valeur souhaitée;
- Notez le mot-clé pass à l'intérieur du corps de la boucle, il ne fait rien du tout; c'est une instruction **vide**, nous le mettons ici parce que la syntaxe de la boucle for exige au moins une instruction à l'intérieur du corps (soit dit en passant : if, elif, else et while le demande aussi)

Nos prochains exemples seront un peu plus modestes en nombre de répétitions en boucle.

Jetez un coup d'œil à l'extrait ci-dessous. Pouvez-vous prédire sa sortie?

```
for i in range(10):  
    print("La valeur de i est actuellement : ", i)
```

Exécutez le code pour vérifier si vous aviez raison.

Note:

- La boucle a été exécutée dix fois (c'est l'argument de la fonction range())
- La valeur de la dernière variable de contrôle est 9 (et non 10, car **elle commence à partir de 0, pas 1**)

L'appel de la fonction range() peut être équipé de deux arguments, et non d'un seul :

```
for i in range(2, 8):  
    print("La valeur de i est actuellement : ", i)
```

Dans ce cas, le premier argument détermine la valeur initiale de la variable de contrôle.

Le dernier argument indique la première valeur à laquelle la variable de contrôle ne sera pas affectée.

Remarque : la fonction range() **n'accepte que des entiers comme arguments** et génère des séquences d'entiers.

La première valeur affichée sera 2 (tirée du premier argument de range().)

Le dernier est 7 (bien que le deuxième argument de range() soit 8).

#### 4.2.2 Encore plus de pouvoir avec 3 arguments

La fonction range() peut également accepter **trois arguments**

```
for i in range(2, 8, 3):  
    print("La valeur de i est actuellement : ", i)
```

Le troisième argument est un incrément, c'est une valeur ajoutée pour contrôler la variable à chaque tour de boucle (comme vous pouvez vous en douter, la valeur **par défaut de l'incrément est 1**).

Pouvez-vous nous dire combien de lignes apparaîtront dans la console et quelles valeurs elles contiendront ?

Vous devriez pouvoir voir les lignes suivantes dans la fenêtre de console :

```
La valeur de i est actuellement de 2  
La valeur de i est actuellement de 5
```

Savez-vous pourquoi? Le premier argument passé à la fonction range() nous indique quel est le **départ** de la séquence (ici 2 ).

Le deuxième argument indique à la fonction où **arrêter** la séquence (la fonction génère des nombres jusqu'au nombre indiqué par le deuxième argument, mais ne l'inclut pas). Enfin, le troisième argument indique l'étape, ce qui signifie en fait la différence entre chaque nombre dans la séquence de nombres générée par la fonction.

2 (nombre de départ)  $\rightarrow$  5 (2 incréments de 3 égalent 5 - le nombre est compris entre 2 et 8)  $\rightarrow$  8 (5 incréments de 3 est égal à 8 - le nombre n'est pas compris entre 2 et 8, car le paramètre stop n'est pas inclus dans la séquence de nombres générée par la fonction.)

Remarque : si l'ensemble généré par la fonction `range()` est vide, la boucle n'exécutera pas du tout son corps.

Tout comme ici, il n'y aura pas de sortie:

```
for i in range(1, 1):  
    print("La valeur de i est actuellement ", i)
```

Remarque : l'ensemble généré par la fonction `range()` doit être trié par ordre croissant. Il n'y a aucun moyen de forcer le `range()` à créer un ensemble sous une forme différente lorsque la fonction `range()` accepte exactement deux arguments. Cela signifie que le deuxième argument de `range()` doit être supérieur au premier.

Ainsi, il n'y aura pas non plus de sortie ici:

```
for i in range(2, 1):  
    print("La valeur de i est actuellement ", i)
```

Jetons un coup d'œil à un programme court dont la tâche est d'écrire quelques-unes des premières puissances de deux:

```
power = 1  
for expo in range(16):  
    print("2 à la puissance de", expo, "est", power)  
    power *= 2
```

La variable `expo` est utilisée comme variable de contrôle pour la boucle et indique la valeur actuelle de l'exposant. L'exponentiation elle-même est remplacée par la multiplication par deux. Puisque  $2^0$  est égal à 1, alors  $2 \times 1$  est égal à  $2^1$ ,  $2 \times 2^1$  est égal à  $2^2$ , et ainsi de suite. Quel est le plus grand exposant pour lequel notre programme imprime encore le résultat?

## 4.2.3 Exercice

## Temps estimé

5-15 minutes

## Objectifs

Familiariser l'élève avec :

- à l'aide de la boucle `for`;
- reflétant des situations réelles dans le code informatique.

## Scénario

Savez-vous ce qu'est le Mississippi? Eh bien, c'est le nom de l'un des États et d'une rivière le traversant aux États-Unis. Le fleuve Mississippi mesure environ 2 340 milles de long, ce qui en fait le deuxième plus long fleuve des États-Unis (le plus long étant le fleuve Missouri). C'est tellement long qu'une seule goutte d'eau a besoin de 90 jours pour parcourir toute sa longueur!

Le mot *Mississippi* est également utilisé dans un but légèrement différent: compter les **mississippily**.

Si vous n'êtes pas familier avec la phrase, nous sommes là pour vous expliquer ce qu'elle signifie: elle est utilisée pour compter les secondes.

L'idée derrière cela est que l'ajout du mot Mississippi à un nombre lors du comptage des secondes à haute voix les rend plus proches de la réalité, et donc « un Mississippi, deux Mississippi, trois Mississippi » prendra environ trois secondes! Il est souvent utilisé par les enfants qui jouent à cache-cache pour s'assurer que le chercheur fait un compte juste.

Votre tâche est très simple ici: écrire un programme qui utilise une boucle `for` pour compter jusqu'à cinq. Après avoir compté jusqu'à cinq, le programme devrait imprimer à l'écran le message final « Prêt ou pas, me voilà! »

Utilisez le squelette suivant :

```
import time

# Write a for loop that counts to five.
# Body of the loop - print the loop iteration number and the word
"Mississippi".
# Body of the loop - use: time.sleep(1)

# Write a print function with the final message.
```

## INFOS SUPPLÉMENTAIRES

Notez que le code ci-dessus contient deux éléments qui peuvent ne pas être tout à fait clairs pour vous en ce moment : l'instruction `import time` et la méthode `sleep()`. Ce n'est pas grave utilisez le sans savoir exactement le but pour le moment, nous devons juste que vous dire que nous avons importé le module de `time` et utilisé la méthode `sleep()` pour suspendre l'exécution de chaque fonction `print()` dans la boucle `for` pendant une seconde, de sorte que le message envoyé à la console ressemble à un comptage réel.



#### 4.2.3 Les mots-clés : break et continue

Jusqu'à présent, nous avons traité le corps de la boucle comme une séquence indivisible et inséparable d'instructions qui sont exécutées complètement à chaque passage dans celle-ci. Cependant, en tant que développeur, vous pourriez être confronté aux choix suivants :

- il semble qu'il soit inutile de poursuivre la boucle dans son ensemble; vous devez vous abstenir de toute exécution ultérieure du corps de la boucle et aller plus loin dans votre code;
- Il semble que vous deviez commencer le tour suivant de la boucle sans terminer l'exécution du tour en cours.

Python fournit deux instructions spéciales pour la mise en œuvre de ces deux tâches. Disons par souci de précision que leur existence dans le langage n'est pas nécessaire, un programmeur expérimenté est capable de coder n'importe quel algorithme sans ces instructions et il faudrait vraiment pouvoir vous en passer. De tels ajouts, qui n'améliorent pas le pouvoir expressif du langage, mais simplifient seulement le travail du développeur, sont parfois appelés **bonbons syntaxiques**. Et on le sait tous, les bonbons c'est bon mais clairement pas indispensable !

Ces deux instructions sont les suivantes :

- **break** - quitte immédiatement la boucle et met fin inconditionnellement au fonctionnement de la boucle; le programme commence à exécuter l'instruction la plus proche après le corps de la boucle;
- **continue** - se comporte comme si le programme avait soudainement atteint l'extrémité du corps; le tour suivant est lancé et l'expression de la condition est testée immédiatement.

```
# break - example
```

```
print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")
```

```
# continue - example
```

```
print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

Amusez-vous avec ce code, transformez-le c'est la meilleure manière d'apprendre.

#### 4.2.4 Quelques exemples

Revenons à notre programme qui reconnaît le plus grand des nombres inscrits.

Nous allons le convertir deux fois, en utilisant les instructions `break` et `continue`.

Analysez le code et jugez si et comment vous utiliseriez l'un ou l'autre.

- La variante avec **break** :

```
largest_number = -99999999
counter = 0
while True:
    number = int(input("Entrez un nombre ou tapez -1 pour terminer le
programme: "))
    if number == -1:
        break
    counter += 1
    if number > largest_number:
        largest_number = number

if counter != 0:
    print("Le plus grand est ", largest_number)
else:
    print("Vous n'avez entré aucun nombre.")
```

- La variante **continue**:

```
largest_number = -99999999
counter = 0

number = int(input("Entrez un nbr ou -1 pour terminer le programme: "))

while number != -1:
    if number == -1:
        continue
    counter += 1

    if number > largest_number:
        largest_number = number
    number = int(input("Entrez un nbr ou -1 pour fermer le
programme: "))

if counter:
    print("Le plus grand est ", largest_number)
else:
    print("Vous n'avez entré aucun nombre.")
```

Regardez attentivement, l'utilisateur entre le premier numéro **avant** que le programme n'entre dans la boucle `while`. Le numéro suivant est entré lorsque le programme est **déjà dans la boucle**.

## 4.2.5 Exercices

## 1 Temps estimé

10-20 minutes

## Objectifs

Familiariser l'élève avec :

- utilisation de l'instruction **break** dans les boucles;
- reflétant des situations réelles dans le code informatique.

## Scénario

L'instruction **break** est utilisée pour quitter/terminer une boucle.

Concevez un programme qui utilise une boucle **while** et demande continuellement à l'utilisateur d'entrer un mot à moins que l'utilisateur n'entre « Chupacabra » comme mot de sortie secret, auquel cas le message « Vous avez quitté la boucle avec succès ». doit être imprimé à l'écran et la boucle terminée.

N'imprimez aucun des mots saisis par l'utilisateur. Utilisez le concept d'exécution conditionnelle et l'instruction **break**.

## 2 Temps estimé

10-20 minutes

## Objectifs

Familiariser l'élève avec :

- utilisation de l'instruction continue dans les boucles;
- reflétant des situations réelles dans le code informatique.

## Scénario

Votre tâche ici est très spéciale: vous devez concevoir un mangeur de voyelles! Écrivez un programme qui utilise :

- une boucle **for**;
- Le concept d'exécution conditionnelle (*if-elif-else*)
- L'instruction continue.

Votre programme doit :

- demander à l'utilisateur d'entrer un mot;
- utilisez `user_word = user_word.upper()` pour convertir le mot entré par l'utilisateur en majuscules; nous parlerons des méthodes dites de **chaîne** et de la méthode `upper()` très bientôt, ne vous inquiétez pas;
- utiliser l'exécution conditionnelle et l'instruction continue pour « manger » les voyelles suivantes *A, E, I, O, U* du mot saisi ;
- Imprimez les lettres non consommées à l'écran, chacune d'elles sur une ligne séparée.

## Données d'essai

Exemple d'entrée : Johan

Résultats escomptés :

J  
H  
N

Exemple d'entrée : Fabrice

Résultats escomptés :

F  
B  
R  
C

Exemple d'entrée : AUIOE

Résultats escomptés :

## Code source

```
# Prompt the user to enter a word
# and assign it to the user_word variable.
for letter in user_word:
    # Complete the body of the for loop.
```

## 3 Temps estimé

5-15 minutes

## Objectifs

Familiariser l'élève avec :

- utilisation de l'instruction **continue** dans les boucles;
- modifier et mettre à niveau le code existant;
- reflétant des situations réelles dans le code informatique.

## Scénario

Votre tâche ici est encore plus spéciale qu'avant: vous devez redessiner le (laid) mangeur de voyelles de l'exercice précédent et en créer un meilleur (joli)! Écrivez un programme qui utilise :

- Une boucle **for**;
- Le concept d'exécution conditionnelle (*if-elif-else*)
- **continue**.

Votre programme doit :

- Demander à l'utilisateur d'entrer un mot;
- Utilisez `user_word = user_word.upper()` pour convertir le mot entré par l'utilisateur en majuscules;
- Utiliser l'exécution conditionnelle et l'instruction continue pour « manger » les voyelles suivantes *A, E, I, O, U* du mot saisi ;
- Affectez les lettres non consommées à la variable `word_without_vowels` et imprimez la variable à l'écran.

Regardez le code ci-dessous. Nous avons créé `word_without_vowels` et lui avons attribué une chaîne vide. Utilisez l'opération de concaténation pour demander à Python de combiner les lettres sélectionnées en une chaîne plus longue lors des tours de boucle suivants et de l'affecter à la variable `word_without_vowels`.

```
word_without_vowels = ""

# Prompt the user to enter a word
# and assign it to the user_word variable.

for letter in user_word:
    # Complete the body of the loop.

# Print the word assigned to word_without_vowels.
```

## Données d'essai

Exemple d'entrée : Johan

Résultats escomptés : JHN

Exemple d'entrée : Reussite

Résultats escomptés :RSST

Exemple d'entrée : IOUEA

Résultats escomptés :

#### 4.2.6 Le else et la boucle while

Les deux boucles, `while` et `for`, ont une fonctionnalité intéressante (et rarement utilisée).

Nous allons vous montrer comment cela fonctionne, essayez de juger par vous-même si la fonctionnalité est précieuse et utile, ou s'il s'agit simplement de bonbons syntaxique.

Jetez un coup d'œil à l'extrait suivant, il y a quelque chose d'étrange à la fin, le mot-clé `else` :

```
i = 1
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)
```

La branche `else` de la boucle **est toujours exécutée une fois, que la boucle soit entrée dans son corps ou non.**

Modifions un peu l'extrait afin que la boucle n'ait aucune chance d'exécuter son corps une seule fois :

```
i = 1
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)
```

La condition de la boucle `while` est Faux dès le début. Nous n'aurons donc que le `else` d'exécuté.

#### 4.2.7 Le else et la boucle for

La boucle `for` se comportent un peu différemment, jetez un coup d'œil à l'extrait dans suivant :

```
for i in range(5):
    print(i)
else:
    print("else:", i)
```

La sortie peut être un peu surprenante. La variable `i` conserve sa dernière valeur.

Modifions un peu le code pour effectuer une autre expérience.

```
i = 111
for i in range(2, 1):
    print(i)
else:
    print("else:", i)
```

Le corps de la boucle ne sera pas du tout exécuté ici.

Lorsque le corps de la boucle n'est pas exécuté, la variable de contrôle conserve la valeur qu'elle avait avant la boucle.

Note: **Si la variable de contrôle n'existe pas avant le démarrage de la boucle, elle n'existera pas lorsque l'exécution atteindra le else.**

#### 4.2.8 Exercices

### 1 Temps estimé

20-30 minutes

### Objectifs

Familiariser l'élève avec :

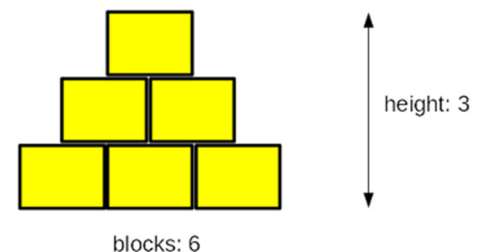
- à l'aide de la boucle while;
- trouver la bonne mise en œuvre des règles définies verbalement;
- reflétant des situations réelles dans le code informatique.

### Scénario

Écoutez cette histoire : un garçon et son père, programmeur informatique, jouent avec des blocs de bois. Ils construisent une pyramide. Leur pyramide est un peu bizarre, car c'est en fait un mur en forme de pyramide, elle est en 2D. La pyramide est empilée selon un principe simple: chaque couche inférieure contient un bloc de plus que la couche supérieure.

La figure illustre la règle utilisée par nos constructeurs :

Votre tâche consiste à écrire un programme qui lit le nombre de blocs dont disposent les constructeurs et affiche la hauteur de la pyramide qui peut être construite à l'aide de ces blocs.



Remarque: la hauteur est mesurée par le nombre de lignes entièrement terminées, si les constructeurs n'ont pas un nombre suffisant de blocs et ne peuvent pas terminer l'étage suivant, ils terminent leur travail immédiatement.

### Données de test

Exemple d'entrée : 6

Résultats escomptés : La hauteur de la pyramide: 3

Exemple d'entrée : 1000

Résultats escomptés : La hauteur de la pyramide: 44

Exemple d'entrée : 2

Expected output: The height of the pyramid: 1

## 2 Temps estimé

20 minutes

## Objectives

Familiariser l'élève avec :

- en utilisant la boucle `while`;
- convertir des boucles définies verbalement en code Python réel.

## Scenarior

En 1937, un mathématicien allemand nommé Lothar Collatz a formulé une hypothèse intrigante (elle reste encore non prouvée) qui peut être décrite de la manière suivante:

- Prendre n'importe quel nombre entier non négatif et non nul et le nommer `c0`;
- S'il est pair, évaluez un nouveau `c0` comme  $c0 \div 2$ ;
- Sinon, si c'est impair, évaluez un nouveau `c0` comme  $3 \times c0 + 1$ ;
- Si  $c0 \neq 1$ , passez au point 2.
- L'hypothèse dit que quelle que soit la valeur initiale de `c0`, il finira à 1.

Bien sûr, c'est une tâche extrêmement complexe d'utiliser un ordinateur afin de prouver l'hypothèse de tout entier naturel (cela peut même nécessiter une intelligence artificielle), mais vous pouvez utiliser Python pour vérifier certains nombres individuels. Peut-être trouverez-vous celui qui réfute l'hypothèse.

Écrire un programme qui lit un entier naturel et exécute les étapes ci-dessus tant que `c0` reste différent de 1. Nous voulons également que vous comptiez les étapes nécessaires pour atteindre l'objectif. Votre code doit générer toutes les valeurs intermédiaires de `c0`.

## Données de test

Exemple d'entrée : 15

Résultats escomptés : 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1  
étapes = 17

Exemple d'entrée : 16

Résultats escomptés : 8 4 2 1  
étapes = 4

Exemple d'entrée : 1023

Résultats escomptés : 3070 1535 4606 2303 6910 3455 10366 5183 15550  
7775 23326 11663 34990 17495 52486 26243 78730 39365 118096 59048  
29524 14762 7381 22144 11072 5536 2768 1384 692 346 173 520 260 130 65  
196 98 49 148 74 37 112 56 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2  
1  
étapes = 62



### 4.3 Résumé

#### 1. Il existe deux types de boucles en Python : while et for:

- Le while exécute une instruction ou un ensemble d'instructions tant qu'une condition booléenne spécifiée est vraie, par exemple :

```
# Exemple 1
while True:
    print("Stuck in an infinite loop.")

# Exemple 2
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

- Le boucle for exécute un ensemble d'instructions plusieurs fois ; elle est utilisée pour itérer sur une séquence (par exemple, une liste, un dictionnaire, un tuple ou un ensemble) ou d'autres objets itérables (par exemple, des chaînes). Vous pouvez utiliser la boucle for pour itérer sur une séquence de nombres à l'aide de la fonction de range intégrée .

```
# Exemple 1
word = "Python"
for letter in word:
    print(letter, end=" ")

# Exemple 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

#### 2. Vous pouvez utiliser le break et continue pour modifier le flux d'une boucle :

- Vous utilisez break pour quitter une boucle, par exemple :

```
text = "HEH love Python "
for letter in text:
    if letter == "P":
        break
    print(letter, end=" ")
```

- Vous utilisez continue pour ignorer l'itération en cours et continuer avec l'itération suivante, par exemple :

```
text = "pyxpyxpyx"
for letter in text:
    if letter == "x":
        continue
    print(letter, end=" ")
```

3. Les boucles while et for peuvent également avoir une clause else en Python. La clause else s'exécute après la fin de l'exécution de la boucle tant qu'elle n'a pas été terminée par break, par exemple :

```
n = 0
while n != 3:
    print(n)
    n += 1
else:
    print(n, "else")

print()
for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```

4. La fonction range() génère une séquence de nombres. Elle accepte les entiers et renvoie les valeurs de la plage. La syntaxe de range() se présente comme suit : range (début, arrêt, étape) où :

- début est un paramètre facultatif spécifiant le numéro de début de la séquence (0 par défaut)
- arrêt est un paramètre optionnel spécifiant la fin de la séquence générée (il n'est pas inclus),
- étape(ou pas) est un paramètre facultatif spécifiant la différence entre les nombres de la séquence (1 par défaut).

Exemple de code :

```
for i in range(3):
    print(i, end=" ") # Outputs: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ") # Outputs: 6, 4, 2
```

### Exercice 1

Créer un for qui compte de 0 à 10 et imprime les nombres impairs à l'écran.

Utilisez le squelette ci-dessous:

```
for i in range(1, 11):
    # Ligne de code.
    # Ligne de code.
```

### Exercice 2

Créez une boucle while qui compte de 0 à 10 et imprime les nombres impairs à l'écran. Utilisez le squelette ci-dessous:

```
x = 1
while x < 11:
    # Ligne de code.
    # Ligne de code.
    # Ligne de code.
```

**Exercice 3**

Créer un programme avec un for et un break . Le programme doit itérer sur les caractères d'une adresse e-mail, quitter la boucle lorsqu'il atteint le @ et imprimez les éléments avant @ sur une seule ligne. Utilisez le squelette ci-dessous:

```
for ch in "john.smith@pythoninstitute.org":  
    if ch == "@":  
        # Ligne de code.  
        # Ligne de code.
```

**Exercice 4**

Créer un programme avec un for et un continue. Le programme doit itérer sur une chaîne de chiffres, remplacer chaque 0 avec x, puis imprimez la chaîne modifiée à l'écran. Utilisez le squelette ci-dessous:

```
for chiffre in "0165031806510":  
    if chiffre == "0":  
        # Ligne de code.  
        # Ligne de code.  
        # Ligne de code.
```

**Exercice 5**

Quelle est la sortie du code suivant ?

```
n = 3  
  
while n > 0:  
    print(n + 1)  
    n -= 1  
else:  
    print(n)
```

**Exercice 6**

Quelle est la sortie du code suivant ?

```
n = range(4)  
  
for num in n:  
    print(num - 1)  
else:  
    print(num)
```

**Exercice 7**

Quelle est la sortie du code suivant ?

```
for i in range(0, 6, 3):  
    print(i)
```

## 5. La logique informatique

### 5.1 Introduction

Avez-vous remarqué que les conditions que nous avons utilisées jusqu'à présent ont été très simples, pour ne pas dire assez primitives? Vous avez dû voir bien plus dans un autre cours.

Les conditions que nous utilisons dans la vie réelle sont beaucoup plus complexes. Regardons cette phrase:

*Si nous avons du temps libre et qu'il fait beau, nous irons boire un verre.*

Nous avons utilisé la conjonction **et**, ce qui signifie que le verre dépend du respect simultané de ces deux conditions. Dans le langage de la logique, une telle connexion de conditions est appelée une **conjonction**. Un autre exemple:

*Si tu es dans le centre commercial ou si je le suis, l'un de nous achètera un cadeau pour notre professeur préféré.*

L'apparition du mot **ou** signifie que l'achat dépend d'au moins une de ces conditions. En logique, un tel composé est appelé une **disjonction**.

Il est clair que Python doit avoir des opérateurs pour construire des conjonctions et des disjonctions. Sans eux, le pouvoir expressif du langage serait considérablement affaibli. Ils s'appellent les **opérateurs logiques**.

#### 5.1.1 Le et

L'opérateur de conjonction logique en Python est le mot-clé **and**. Il s'agit d'un **opérateur binaire avec une priorité inférieure à celle exprimée par les opérateurs de comparaison**. Il nous permet de coder des conditions complexes sans utiliser de parenthèses comme ceci:

```
counter > 0 and value == 100
```

Le résultat fourni par l'opérateur **and** peut être déterminé sur la base de sa **table de vérité**.

Si l'on considère la conjonction de **A et B**, l'ensemble des valeurs possibles des arguments et des valeurs correspondantes de la conjonction se présente comme suit :

Argument A	Argument B	A et B
Faux	Faux	Faux
Faux	Vrai	Faux
Vrai	Faux	Faux
Vrai	Vrai	Vrai

### 5.1.2 Le ou

Un opérateur de disjonction est le mot **or**. C'est un **opérateur binaire avec une priorité inférieure à and** (tout comme **+** par rapport à **\***). Son tableau de vérité est le suivant :

Argument A	Argument B	A ou B
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	Faux	Vrai
Vrai	Vrai	Vrai

### 5.1.3 Le non

Il existe un autre opérateur qui peut être appliqué pour les conditions de construction. C'est un **opérateur unaire effectuant une négation logique**. Son fonctionnement est simple : il transforme la vérité en mensonge et le mensonge en vérité.

Cet opérateur est écrit **not**, et sa **priorité est très élevée: la même que pour les comparateurs unaires + et -**. Sa table de vérité est simple:

Argument	not d'argument
Faux	Vrai
Vrai	Faux

## 5.2 Les expressions logique

Créons une variable nommée **var** et assignons-lui la valeur **1** . Les conditions suivantes sont **équivalentes**:

```
# Example 1:
print(var > 0)
print(not (var <= 0))
```

```
# Example 2:
print(var != 0)
print(not (var == 0))
```

Vous connaissez peut-être les lois de De Morgan qui dit :

*La négation d'une conjonction est la disjonction des négations.  
La négation d'une disjonction est la conjonction des négations.*

Écrivons la même chose en utilisant Python:

```
not (p and q) == (not p) or (not q)
not (p or q) == (not p) and (not q)
```

On a utilisé les parenthèses pour améliorer la lisibilité.

Nous devons ajouter qu'aucun de ces opérateurs à deux arguments ne peut être utilisé sous la forme abrégée comme **op=**.

### 5.3 Valeurs logiques vs bits uniques

Les opérateurs logiques prennent leurs arguments dans leur ensemble, quel que soit le nombre de bits qu'ils contiennent. Les opérateurs ne connaissent que la valeur : zéro (lorsque tous les bits sont réinitialisés) signifie Faux; non nul (lorsqu'au moins un bit est défini) signifie Vrai.

Le résultat de leurs opérations est l'une de ces deux valeurs. Cela signifie que cet extrait affectera la valeur Vrai à la variable `j` si `i` n'est pas égal à zéro ; sinon, il sera Faux. Comme `i` commence à 1, il vaudra bien Vrai.

```
i = 1
j = not not i
```

### 5.4 Opérateurs binaires

Il existe quatre opérateurs qui vous permettent de **manipuler des bits de données uniques**. Ils sont appelés **opérateurs binaires**.

Ils couvrent toutes les opérations que nous avons mentionnées précédemment dans le contexte logique, ainsi qu'un opérateur supplémentaire.

C'est le **xor (ou exclusif)**, et il est désigné par **^** (caret).

Les voici tous :

- **&** (ampersand) : conjonction binaire;
- **|** (bar) : disjonction binaire;
- **~** (tilde) : négation binaire;
- **^** (caret) : bit exclusive ou (xor).

Opérations binaires (&,   et ^)				
Argument A	Argument B	A & B	Un   B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Opérateurs binaires (~)	
Argument	~ Argument
0	1
1	0

Facilitons-nous la tâche :

- **&** nécessite exactement deux 1 pour fournir 1 comme résultat;
- **|** exige au moins un 1 pour fournir 1 comme résultat;
- **^** nécessite exactement un 1 pour fournir 1 comme résultat.

Ajoutons une remarque importante: les arguments de ces opérateurs **doivent être des entiers**; nous ne devons pas utiliser de flottants ici.

La différence dans le fonctionnement des opérateurs logiques et binaires est importante : **les opérateurs logiques ne pénètrent pas dans le niveau binaire de son argument.** Ils ne s'intéressent qu'à la valeur entière finale.

Les opérateurs binaires sont plus stricts : ils traitent **chaque bit séparément.** Si nous supposons que la variable entière occupe 64 bits (ce qui est courant dans les systèmes informatiques modernes), vous pouvez imaginer l'opération binaire comme une évaluation de l'opérateur logique 64 fois pour chaque paire de bits des arguments. Cette analogie est évidemment imparfaite, car dans le monde réel, toutes ces 64 opérations sont effectuées en même temps.

Nous allons maintenant vous montrer un exemple de la différence de fonctionnement entre les opérations logiques et binaires. Supposons que les tâches suivantes ont été effectuées :

```
i = 15
j = 22
```

Si nous supposons que les entiers sont stockés avec 32 bits, l'image binaire des deux variables sera la suivante:

```
i: 000000000000000000000000000001111
j: 000000000000000000000000000010110
```

L'affection est :

```
log = i and j
```

Il s'agit ici d'une conjonction logique. Retraçons le cours des calculs. Les variables *i* et *j* ne sont pas des zéros, elles seront donc considérées comme représentant True. En consultant la table de vérité pour l'opérateur `and`, nous pouvons voir que le résultat sera True. Aucune autre opération n'est effectuée et donc `log` sera égale à Vrai

Maintenant, réalisons ce calcul en opération binaire:

```
bit = i & j
```

L'opérateur **&** fonctionnera avec chaque paire de bits correspondants séparément, produisant les valeurs des bits résultants. Par conséquent, le résultat sera le suivant :

i	000000000000000000000000000001111
j	000000000000000000000000000010110
bit = i & j	000000000000000000000000000001110

Ces bits correspondent à la valeur entière de six.





**Exercice 1**

Quelle sera la sortie du code suivant :

```
x = 1
y = 0

z = ((x == y) and (x == y)) or not(x == y)
print(not(z))
```

**Exercice 2**

Quelle sera la sortie du code suivant :

```
x = 4
y = 1

a = x & y
b = x | y
c = ~x # tricky!
d = x ^ 5
e = x >> 2
f = x << 2

print(a, b, c, d, e, f)
```

## 6. Les listes

### 6.1 La base des listes

#### 6.1.1 Pourquoi les listes sont-elles utiles ?

Il peut arriver que vous deviez lire, stocker, traiter, imprimer, ... des dizaines, peut-être des centaines, voire même des milliers de nombres.

Avez-vous besoin de créer une variable distincte pour chaque valeur ?

Devrez-vous passer de longues heures à écrire des déclarations comme celle-ci?

```
var1 = int(input())
var2 = int(input())
var3 = int(input())
var4 = int(input())
var5 = int(input())
var6 = int(input())
:
```

Si vous ne pensez pas que c'est une tâche compliquée, prenez un morceau de papier et écrivez un programme qui:

- lit cinq chiffres,
- les imprime dans l'ordre du plus petit au plus grand (NB, ce type de traitement est appelé **tri (sorting)**).

Vous devriez constater que vous n'avez même pas assez de place pour terminer la tâche.

Jusqu'à présent, vous avez appris à déclarer des variables capables de stocker exactement une valeur donnée à la fois. De telles variables sont parfois appelées **scalaires** par analogie avec les mathématiques. Toutes les variables que vous avez utilisées jusqu'à présent sont en fait des scalaires.

Pensez à quel point il serait pratique de déclarer une variable qui pourrait **stocker plus d'une valeur**. Par exemple, cent, mille ou même dix mille. Ce serait toujours une seule et même variable, mais très large et vaste. Cela semble attrayant? Peut-être, mais comment gérer un tel conteneur de valeurs différentes? Comment choisir uniquement celle dont vous avez besoin?

Et si vous pouviez simplement les numéroté? Et puis donner des ordres comme: *donnez-moi la deuxième valeur; attribuer la valeur 15; Augmenter la valeur de 10000.*

Pour prouver que c'est possible, nous le ferons avec l'exemple que nous venons de suggérer. Nous écrirons un **Programme qui trie une séquence de nombres**. Nous ne serons pas particulièrement ambitieux, nous supposons qu'il y a exactement cinq chiffres.

Créons une variable appelée nombres; elle est affectée non pas à un seul nombre, mais à une liste composée de cinq valeurs (note: la liste commence par **un crochet ouvert et se termine par un crochet fermé**; l'espace entre les parenthèses est rempli de cinq nombres séparés par des virgules).

```
nombres = [10, 5, 7, 2, 1]
```

Utilisons une terminologie adéquate pour nous exprimer: nombres **est une liste composée de cinq valeurs, toutes des nombres**. Nous pouvons également dire que cette déclaration crée une liste de longueur égale à cinq.

Les éléments à l'intérieur d'une liste **peuvent avoir différents types**. Certains d'entre eux peuvent être des entiers, d'autres des flottants, et d'autres encore peuvent être des listes.

Python a adopté une convention stipulant que les éléments d'une liste sont **toujours numérotés à partir de zéro**. Cela signifie que l'élément stocké au début de la liste aura le numéro zéro (l'indice). Comme il y a cinq éléments dans notre liste, le dernier d'entre eux se voit attribuer le numéro quatre.

Avant d'aller plus loin dans notre discussion, nous devons déclarer ce qui suit : notre **liste est une collection d'éléments, mais chaque élément est un scalaire**.

### 6.1.2 L'index

Comment peut-on modifier la valeur d'un élément choisi dans la liste ?  
Allons **Affectez une nouvelle valeur 111 au premier élément** de la liste. Nous procéderons de cette façon :

```
numbers = [10, 5, 7, 2, 1]
print("La liste original est :", numbers)

numbers[0] = 111
print("La nouvelle liste est : ", numbers)
```

Et maintenant, nous voulons **copier la valeur du cinquième élément dans le deuxième élément**, Pouvez-vous deviner comment faire?

```
nombres = [10, 5, 7, 2, 1]
print("La liste original est:", nombres)

nombres[0] = 111
print("\n Contenu de la liste précédente: ", nombres)

numbers[1] = numbers[4] # Copie valeur du 5ème élément vers le second.
print("Nouvelle liste :", numbers)
```

La valeur entre crochet qui sélectionne un élément de la liste est **appelée index**, tandis que l'opération de sélection d'un élément de la liste est appelée **indexation**.

Nous allons utiliser la fonction `print()` pour imprimer le contenu de la liste chaque fois que nous apportons les modifications. Cela nous aidera à suivre chaque étape plus attentivement et à voir ce qui se passe après modification.

Remarque: tous les indices utilisés jusqu'à présent sont littéraux. Leurs valeurs sont fixes au moment de l'exécution, mais **n'importe quelle expression peut également être l'index**. Cela ouvre beaucoup de possibilités.

### 6.1.3 Accès au contenu d'une liste

Partons d'un code de base :

```
numbers = [10, 5, 7, 2, 1]
print("Original list content:", numbers)

numbers[0] = 111
print("\nPrevious list content:", numbers)

numbers[1] = numbers[4]
print("Previous list content:", numbers)

print("\nList length:", len(numbers))
```

Chacun des éléments de la liste est accessible séparément. Par exemple, il peut être imprimé :

```
print(numbers[0])
```

En supposant que toutes les opérations précédentes ont été effectuées avec succès, l'extrait de code enverra 111 à la console. Comme vous pouvez le voir, la liste peut également être imprimée dans son ensemble:

```
print(numbers) # Printing the whole list.
```

Comme vous l'avez probablement déjà remarqué, Python décore la sortie d'une manière qui suggère que toutes les valeurs présentées forment une liste. La sortie de l'exemple d'extrait ci-dessus ressemble à ceci :

```
[111, 1, 7, 2, 1]
```

*The len() function*

La **longueur d'une liste** peut varier au cours de l'exécution. De nouveaux éléments peuvent être ajoutés à la liste, tandis que d'autres peuvent en être retirés. Cela signifie que la liste est une entité dynamique.

Si vous souhaitez vérifier la longueur actuelle de la liste, vous pouvez utiliser une fonction nommée **len()**.

La fonction prend le **nom de la liste comme argument et renvoie le nombre d'éléments actuellement stockés** dans la liste (en d'autres termes, la longueur de la liste).

#### 6.1.4 Supprimer les éléments de la liste.

N'importe lequel des éléments de la liste peut être **supprimé** à tout moment, cela se fait avec une instruction nommée **del** (supprimer).

Remarque: c'est une **instruction**, pas une fonction.

Vous devez pointer sur l'élément à supprimer, il disparaîtra de la liste et la longueur de la liste sera réduite d'une unité.

Regardez l'extrait ci-dessous :

```
del numbers[1]
print(len(numbers))
print(numbers)
```

**Vous ne pouvez pas accéder à un élément qui n'existe pas**, Vous ne pouvez ni obtenir sa valeur ni lui attribuer une valeur.

Ces deux instructions provoqueront des erreurs d'exécution maintenant:

```
print(numbers[4])
numbers[4] = 1
```

Ajoutez l'extrait ci-dessus après la dernière ligne de code dans le code ci-dessous, exécutez le programme et vérifiez ce qui se passe.

```
numbers = [10, 5, 7, 2, 1]
print("Original list content:", numbers)
numbers[0] = 111
print("\nPrevious list content:", numbers)
numbers[1] = numbers[4] second.
print("Previous list content:", numbers)
print("\nList's length:", len(numbers))
del numbers[1]
print("New list's length:", len(numbers))
print("\nNew list content:", numbers)
```

Remarque: nous avons supprimé l'un des éléments de la liste, il n'y a plus que quatre éléments dans la liste. Cela signifie que l'élément numéro quatre n'existe pas. Donc l'élément avec l'indice 4 n'existe plus.

#### 6.1.5 Les indices négatifs

```
numbers = [111, 7, 2, 1]
print(numbers[-1])
print(numbers[-2])
```

Cela peut sembler étrange, mais les indices négatifs sont autorisés et peuvent être très utiles.

Un élément dont l'index est égal à **-1** est par exemple **le dernier de la liste**.

```
print(numbers[-1])
```

De même manière, l'élément dont l'indice est égal à -2 est l'**avant-dernier**.

```
print(numbers[-2])
```

Le dernier élément accessible de notre liste est numbers[-4] (le premier)!

#### 6.1.6 Exercice

### Temps estimé

5 minutes

### Objectifs

Familiariser l'élève avec :

- utiliser des instructions de base relatives aux listes;
- Création et modification de listes.

### Scénario

Il était une fois un chapeau. Le chapeau ne contenait pas de lapin, mais une liste de cinq chiffres: 1, 2, 3, 4, 5. Et chaque chiffre représentait une maison chez notre ami magicien à la cicatrice. La dernière étant la maison inconnue du grand public : HEH

Votre tâche est:

- écrire une ligne de code qui invite l'utilisateur à remplacer le numéro du milieu de la liste par un nombre entier entré par l'utilisateur (étape 1)
- écrire une ligne de code qui supprime le dernier élément de la liste (étape 2)
- écrire une ligne de code qui imprime la longueur de la liste existante (étape 3).

Prêt à relever ce défi ?

### Code de base

```
hat_list = [1, 2, 3, 4, 5]

# Step 1: write a line of code that prompts the user
# to replace the middle number with an integer number entered by the user.

# Step 2: write a line of code that removes the last element from the list.

# Step 3: write a line of code that prints the length of the existing list.

print(hat_list)
```

### 6.1.7 Fonctions vs Méthodes

Une **méthode est un type spécifique de fonction**, elle se comporte comme une fonction et ressemble à une fonction, mais diffère dans la façon dont elle agit et dans son style d'invocation.

Une **fonction n'appartient à aucune donnée**, elle obtient des données, elle peut créer de nouvelles données et elle produit (généralement) un résultat.

Une méthode fait toutes ces choses, mais est également capable **de changer l'état d'une entité sélectionnée**.

**Une méthode appartient aux données pour lesquelles elle travaille, tandis qu'une fonction appartient à l'ensemble du code.**

Cela signifie également que l'appel d'une méthode nécessite une spécification des données à partir desquelles la méthode est appelée.

Cela peut sembler déroutant ici, mais nous le traiterons en profondeur lorsque nous nous plongerons dans la programmation orientée objet au Q2.

En général, un appel de fonction typique peut ressembler à ceci :

```
result = function(arg)
```

La fonction prend un argument, fait quelque chose et renvoie un résultat.

Un appel de méthode typique ressemble généralement à ceci :

```
result = data.method(arg)
```

Remarque : le nom de la méthode est précédé du nom des données propriétaires de la méthode. Ensuite, vous ajoutez un **point**, suivi du nom de la **méthode** et d'une paire de **parenthèses entourant les arguments**.

La méthode se comportera comme une fonction, mais peut faire quelque chose de plus, elle peut **changer l'état interne des données** à partir desquelles elle a été appelée.

Vous vous demandez peut-être: pourquoi parlons-nous de méthodes, pas de listes? Simplement car pour comprendre la suite des chapitres, cette nuance doit être acquise.

### 6.1.8 Ajout d'éléments à une liste : `append()` et `insert()`

Un nouvel élément peut être *collé* à la fin de la liste existante :

```
list.append(value)
```

Une telle opération est effectuée par une méthode nommée **`append()`**. Il prend la valeur de son argument et la place **à la fin de la liste** qui utilise la méthode. La longueur de la liste augmente ensuite d'une unité.

L'autre méthode, **`insert()`** est un peu plus intelligente, elle peut ajouter un nouvel élément **à n'importe quel endroit de la liste**, pas seulement à la fin.

```
list.insert(location, value)
```

Il faut deux arguments :

- le premier indique l'emplacement requis de l'élément à insérer; Tous les éléments existants qui occupent des emplacements à droite du nouvel élément (y compris celui à la position indiquée) sont déplacés vers la droite, afin de faire de la place pour le nouvel élément ;
- le second est l'élément à insérer.

Regardez le code suivant. Comprenez-vous comment nous utilisons les méthodes `append()` et `insert()`. Faites attention à ce qui se passe après l'utilisation de `insert()` : l'ancien premier élément est maintenant le deuxième, le second le troisième, et ainsi de suite.

```
numbers = [111, 7, 2, 1]
print(len(numbers))
print(numbers)
###
numbers.append(4)
print(len(numbers))
print(numbers)
###
numbers.insert(0, 222)
print(len(numbers))
print(numbers)
#
```

Ajoutez l'extrait de code suivant après la dernière ligne de code dans le code:

```
numbers.insert(1, 333)
```

Imprimez le contenu de la liste finale à l'écran et voyez ce qui se passe. L'extrait ci-dessus insère 333 dans la liste en indice 1, ce qui en fait le deuxième élément.

Vous pouvez « **commencer la vie d'une liste** » **en la vidant** (cela se fait avec une paire de crochets vides).



Jetez un coup d'œil à l'extrait ci-dessous. Essayez de deviner sa sortie après l'exécution de la boucle **for**.

```
my_list = []
for i in range(5):
    my_list.append(i + 1)
print(my_list)
```

Il s'agira d'une séquence de nombres entiers consécutifs de 1 (vous en ajoutez ensuite une à toutes les valeurs ajoutées) à 5.

Nous avons un peu modifié l'extrait :

```
my_list = []
for i in range(5):
    my_list.insert(0, i + 1)
print(my_list)
```

Que va-t-il se passer maintenant?

Vous devriez obtenir la même séquence, mais dans **l'ordre inverse** (c'est le fait d'utiliser la méthode insert()).

#### 6.1.9 Utilisons mieux nos listes

La boucle for a une variante qui peut **traiter les listes** efficacement.

Supposons que vous souhaitiez **calculez la somme de toutes les valeurs stockées dans la liste my\_list**.

Vous avez besoin d'une variable dont la somme sera stockée et initialement affectée à la valeur 0, son nom sera `total`. (Remarque: nous n'allons pas la nommer `sum` car Python utilise le même nom pour l'une de ses fonctions intégrées, `sum()`. **L'utilisation du même nom est généralement considérée comme une mauvaise pratique.**)

Ensuite, vous y ajoutez tous les éléments de la liste à l'aide de la boucle for.

```
my_list = [10, 1, 8, 3, 5]
total = 0
for i in range(len(my_list)):
    total += my_list[i]

print(total)
```

Commentons cet exemple :

- La liste se voit attribuer une séquence de cinq valeurs entières ;
- La variable `i` prend les valeurs 0, 1, 2, 3 et 4, puis il indexe la liste, en sélectionnant les éléments suivants: le premier, le deuxième, le troisième, le quatrième et le cinquième;
- Chacun de ces éléments est additionné par l'opérateur `+=` à `total`, donnant le résultat final à la fin de la boucle;
- Notez la façon dont la fonction `len()` a été utilisée, elle rend le code indépendant de tout changement possible dans le contenu de la liste.

Mais la boucle for peut faire beaucoup plus. Elle peut masquer toutes les actions liées à l'indexation de la liste et fournir tous les éléments de la liste de manière pratique.

Cet extrait modifié montre comment cela fonctionne :

```
my_list = [10, 1, 8, 3, 5]
total = 0
for i in my_list:
    total += i

print(total)
```

- L'instruction for spécifie la variable utilisée pour parcourir la liste (ici i) suivie du mot-clé in et du nom de la liste en cours de traitement (my\_list).
- La variable i se voit attribuer les valeurs de tous les éléments de la liste suivante, et le processus se produit autant de fois qu'il y a d'éléments dans la liste ;
- Cela signifie que vous utilisez la variable i comme copie des valeurs des éléments et que vous n'avez pas besoin d'utiliser d'index ;
- La fonction len() n'est donc pas nécessaire.

#### 6.1.10 Travaillons avec nos listes

Examinons un problème intrigant pour tenter de mieux comprendre nos listes. Imaginez que vous ayez besoin de réorganiser les éléments d'une liste, c'est-à-dire d'inverser l'ordre des éléments: le premier et le cinquième ainsi que les deuxième et quatrième éléments seront échangés. Le troisième restera le même.

Question : comment échanger les valeurs de deux variables ?

```
variable_1 = 1
variable_2 = 2

variable_2 = variable_1
variable_1 = variable_2
```

Si vous faites quelque chose comme ça, vous **perdrez la valeur précédemment stockée** dans `variable_2`. Changer l'ordre des affectations n'aidera pas. Vous avez besoin d'une **troisième variable qui sert de stockage auxiliaire ou variable tampon**.

Voici comment vous pouvez le faire:

```
variable_1 = 1
variable_2 = 2

tampon = variable_1
variable_1 = variable_2
variable_2 = tampon
```

Python est magique(encore) et offre un moyen plus pratique de faire l'échange :

```
variable_1 = 1
variable_2 = 2

variable_1, variable_2 = variable_2, variable_1
```

Clair, efficace et intelligent comme votre professeur non ?

Vous pouvez maintenant facilement **permuter** les éléments de la liste pour **inverser leur ordre** grâce à cette astuce:

```
my_list = [10, 1, 8, 3, 5]

my_list[0], my_list[4] = my_list[4], my_list[0]
my_list[1], my_list[3] = my_list[3], my_list[1]

print(my_list)
```

Si vous exécutez l'extrait de code. Sa sortie devrait ressembler à ceci:  
[5, 3, 8, 1, 10]

C'est sympa mais ... on est sur un nombre d'éléments fixes là. Que feriez-vous avec 100 éléments ? 1000000 ?

Pouvez-vous utiliser la boucle for pour faire la même chose automatiquement, quelle que soit la longueur de la liste ? Et bien oui !

Voici comment :

```
my_list = [10, 1, 8, 3, 5]
length = len(my_list)

for i in range(length // 2): my_list[i], my_list[length - i - 1] =
my_list[length - i - 1], my_list[i]

print(my_list)
```

Commentaires :

- Nous avons attribué à la variable length, la longueur de la liste actuelle (cela rend notre code un peu plus clair et plus court)
- Nous avons lancé la boucle for pour parcourir son corps length // 2 fois (cela fonctionne bien pour les listes de longueurs paires et impaires)
- Nous avons échangé le i<sup>ème</sup> élément (du début de la liste) avec celui avec un index égal à (longueur - i - 1) (à partir de la fin de la liste); dans notre exemple, pour i égal à 0, le (length - i - 1) donne 4; pour i égal à 1, cela donne 3.

## 6.1.11 Exercice

## Temps estimé

10-15 minutes

## Objectifs

Familiariser l'élève avec :

- créer et modifier des listes simples;
- utilisation de méthodes pour modifier des listes.

## Scénario

Les Beatles étaient l'un des groupes de musique les plus populaires des années 1960 et le groupe le plus vendu de l'histoire. Certaines personnes les considèrent comme le groupe le plus influent de l'ère du rock. Mais aviez-vous entendu parlé des IsiPotes ? Ce groupe composé de Johan le Prêtre, Denis seins doux, Antoine Mal à l'aise et Yoan l'imprononçable faisait ravage dans les bals musettes du borinage. Les line-up du groupe changeant parfois avec les apparitions de Erwin le schmet, Joakim Synagogue ou encore David Art nouveau.

Écrivez un programme qui reflète ces changements et vous permet de vous entraîner avec le concept de listes. Votre tâche consiste à :

- Étape 1: Créez une liste vide nommée groupe;
- étape 2: utilisez la méthode `append()` pour ajouter les membres suivants du groupe à la liste: Johan le Preteur, Denis seins doux et Yoan l'imprononçable;
- Étape 3: Utilisez la boucle `for` et la méthode `append()` pour inviter l'utilisateur à ajouter les membres suivants du groupe à la liste: Joakim Synagogue et David Art Nouveau;
- Étape 4 : Utilisez l'instruction `del` pour retirer David Art nouveau et Denis seins doux de la liste;
- Étape 5 : Utilisez `insert()` pour ajouter Erwin le schmet au début de la liste.

Au fait, êtes-vous un fan des IsiPotes? (Ils sont l'un des groupes préférés d'Anas. Mais attendez... qui est Anas...?)

## Code test

```
# step 1
print("Step 1:", groupe)
# step 2
print("Step 2:", groupes)
# step 3
print("Step 3:", groupes)
# step 4
print("Step 4:", groupes)
# step 5
print("Step 5:", groupes)

# testing list length
print("The Fab", len(groupes))
```

## 6.1.12 Résumé

1. La **liste** est un **type de données** en Python utilisé pour **stocker plusieurs objets**. Il s'agit d'une **collection ordonnée et mutable** d'éléments séparés par des virgules entre crochets, par exemple :

```
my_list = [1, None, True, "I am a string", 256, 0]
```

2. Les listes peuvent être **indexées et mises à jour**, par exemple :

```
my_list = [1, None, True, 'I am a string', 256, 0]
print(my_list[3]) # outputs: I am a string
print(my_list[-1]) # outputs: 0

my_list[1] = '?'
print(my_list) # outputs: [1, '?', True, 'I am a string', 256, 0]

my_list.insert(0, "first")
my_list.append("last")
print(my_list) # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']
```

3. Les listes peuvent être **imbriquées** :

```
my_list = [1, 'a', ["list", 64, [0, 1], False]]
```

4. Les éléments de liste et les listes peuvent être **Supprimés** :

```
my_list = [1, 2, 3, 4]
del my_list[2]
print(my_list) # outputs: [1, 2, 4]

del my_list # deletes the whole list
```

5. Les listes peuvent être **Réitéré** grâce à l'utilisation de la boucle for :

```
my_list = ["white", "purple", "blue", "yellow", "green"]

for color in my_list:
    print(color)
```

6. `len()` peut être utilisée pour **Vérifier la longueur de la liste** :

```
my_list = ["white", "purple", "blue", "yellow", "green"]
print(len(my_list)) # outputs 5

del my_list[2]
print(len(my_list)) # outputs 4
```

7. L'appel d'une **fonction** se présente comme suit : `resultat = fonction(arg)`, tandis que pour une **méthode** l'invocation est : `resultat = data.methode(arg)`.

**Exercice 1**

Quelle est la sortie de l'extrait de code suivant ?

```
lst = [1, 2, 3, 4, 5]
lst.insert(1, 6)
del lst[0]
lst.append(1)

print(lst)
```

**Exercice 2**

Quelle est la sortie de l'extrait de code suivant ?

```
lst = [1, 2, 3, 4, 5]
lst_2 = []
add = 0

for number in lst:
    add += number
    lst_2.append(add)

print(lst_2)
```

Vérifier

**Exercice 3**

Que se passe-t-il lorsque vous exécutez l'extrait de code suivant ?

```
lst = []
del lst
print(lst)
```

**Exercice 4**

Quelle est la sortie de l'extrait de code suivant ?

```
lst = [1, [2, 3], 4]
print(lst[1])
print(len(lst))
```

## 6.2 Trier une liste simple

### 6.2.1 Le tri à bulles

Maintenant que vous pouvez jongler efficacement avec les éléments des listes, il est temps d'apprendre à les trier. De nombreux algorithmes de tri ont été inventés jusqu'à présent, qui diffèrent beaucoup par leur vitesse et leur complexité. Nous allons vous montrer un algorithme très simple, facile à comprendre, mais malheureusement pas trop efficace non plus. Il est utilisé très rarement, et certainement pas pour les grandes et longues listes.

Disons qu'une liste peut être triée de deux manières :

- croissant (ou plus précisément, non décroissant) : si dans chaque paire d'éléments adjacents, le premier élément n'est pas supérieur au second;
- décroissant (ou plus précisément, non croissant) : si dans chaque paire d'éléments adjacents, le premier élément n'est pas inférieur au second.

Dans les sections suivantes, nous allons trier la liste par ordre croissant, afin que les numéros soient classés du plus petit au plus grand.

Voici la liste :

8	10	6	2	4
---	----	---	---	---

Nous allons essayer d'utiliser l'approche suivante: nous allons prendre le premier et le deuxième élément et les comparer; Si nous déterminons qu'ils sont dans le mauvais ordre (c'est-à-dire que le premier est supérieur au second), nous les échangerons; Si pas, on ne fait rien.

Un coup d'œil à notre liste confirme ce point, les éléments 01 et 02 sont dans le bon ordre,  $8 < 10$ .

Regardez maintenant les deuxième et troisième éléments. Ils sont dans les mauvaises positions. Nous devons les échanger:

8	6	10	2	4
---	---	----	---	---

Nous allons plus loin et examinons les troisième et quatrième éléments. Encore une fois, ce n'est pas ce que c'est censé être. Nous devons les échanger:

8	6	2	10	4
---	---	---	----	---

Maintenant, nous vérifions les quatrième et cinquième éléments. Oui, eux aussi sont dans les mauvaises positions. Un autre échange se produit :

8	6	2	4	10
---	---	---	---	----

Le premier passage dans la liste est déjà terminé. Nous sommes encore loin d'avoir terminé notre travail, mais quelque chose de curieux s'est produit entre-temps. L'élément le plus important, 10, est déjà allé à la fin de la liste. Notez que c'est l'endroit **souhaité** pour cela. Les autres éléments forment un désordre.

Maintenant, pendant un moment, essayez d'imaginer la liste d'une manière légèrement différente, à savoir, comme ceci:

10
4
2
6
8

Regardez 10 est en haut. On pourrait dire qu'il a flotté du fond vers la surface, tout comme la **bulle dans une coupe de champagne**. La méthode de tri tire son nom de la même observation.

Maintenant, nous commençons par le deuxième passage à travers la liste. Nous examinons les premier et deuxième éléments, un échange est nécessaire:

6	8	2	4	10
---	---	---	---	----

Au tour des deuxième et troisième éléments: nous devons les échanger aussi:

6	2	8	4	10
---	---	---	---	----

Maintenant, les troisième et quatrième éléments, et la deuxième passe est terminée, car 8 est déjà en place:

6	2	4	8	10
---	---	---	---	----

Nous commençons immédiatement le prochain passage. Surveillez attentivement le premier et le deuxième éléments - un autre échange est nécessaire:

2	6	4	8	10
---	---	---	---	----

Maintenant, 6 doit être mis en place. Nous échangeons les deuxième et troisième éléments:

2	4	6	8	10
---	---	---	---	----

La liste est déjà triée. Nous n'avons plus rien à faire. C'est exactement ce que nous voulons.

Comme vous pouvez le constater, l'essence de cet algorithme est simple : nous **comparons les éléments adjacents, et en échangeant certains d'entre eux, nous atteignons notre objectif**.

Codons en Python toutes les actions effectuées au cours d'un seul passage dans la liste, puis nous examinerons le nombre de passes dont nous avons réellement besoin pour l'effectuer.

De combien de passage avons-nous donc besoin pour trier toute la liste?

Nous résolvons ce problème de la manière suivante: **Nous introduisons une autre variable;**

Sa tâche est d'observer si un échange a été effectué pendant le passage ou non ; S'il n'y a pas d'échange, la liste est déjà triée et rien de plus n'est à faire. Nous créons une variable nommée **swapped**, et nous lui affectons une valeur de Faux, pour indiquer qu'il n'y a pas d'échanges. Partons du code d'origine.



```
my_list = [8, 10, 6, 2, 4] # list to sort

for i in range(len(my_list) - 1): # we need (5 - 1) comparisons
    if my_list[i] > my_list[i + 1]: # compare adjacent elements
        my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i] # If
we end up here, we have to swap the elements.
```

Vous devriez être capable de lire et de comprendre ce programme sans aucun problème:

```
my_list = [8, 10, 6, 2, 4] # list to sort
swapped = True # It's a little fake, we need it to enter the while
loop.

while swapped:
    swapped = False # no swaps so far
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True # a swap occurred!
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print(my_list)
```

### 6.2.2 Créons une version interactive du tri

Ci-dessous, vous pouvez voir un programme complet, enrichi par une conversation avec l'utilisateur, et permettant à l'utilisateur d'entrer et d'imprimer les éléments de la liste: **Le tri à bulles - version interactive.**

```
my_list = []
swapped = True
num = int(input("How many elements do you want to sort: "))

for i in range(num):
    val = float(input("Enter a list element: "))
    my_list.append(val)

while swapped:
    swapped = False
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print("\nSorted:")
print(my_list)
```

Python, cependant, a ses propres mécanismes de tri. Personne n'a besoin d'écrire ses propres types, car il y a un nombre suffisant d'outils **prêts à l'emploi.**

Nous vous avons expliqué ce système de tri car il est important d'apprendre à traiter le contenu d'une liste et de vous montrer comment un tri réel peut fonctionner.

Si vous voulez que Python trie votre liste, vous pouvez le faire comme ceci:

```
my_list = [8, 10, 6, 2, 4]
my_list.sort()
print(my_list)
```

Comme vous pouvez le voir, toutes les listes ont une méthode nommée **sort()**, qui les trie le plus rapidement possible.

### 6.2.3 Résumé

1. Vous pouvez utiliser la méthode **sort()** pour trier les éléments d'une liste, par exemple :

```
lst = [5, 3, 1, 2, 4]
print(lst)

lst.sort()
print(lst)  # outputs: [1, 2, 3, 4, 5]
```

2. Il existe également une méthode de liste appelée **reverse()**, que vous pouvez utiliser pour inverser la liste, par exemple :

```
lst = [5, 3, 1, 2, 4]
print(lst)
lst.reverse()
print(lst)  # outputs: [4, 2, 1, 3, 5]
```

#### Exercice 1

Quelle est la sortie de l'extrait de code suivant ?

```
lst = ["D", "F", "A", "Z"]
lst.sort()
print(lst)
```

#### Exercice 2

Quelle est la sortie de l'extrait de code suivant ?

```
a = 3
b = 1
c = 2
lst = [a, c, b]
lst.sort()
print(lst)
```

#### Exercice 3

Quelle est la sortie de l'extrait de code suivant ?

```
a = "A"
b = "B"
c = "C"
d = " "
lst = [a, b, c, d]
lst.reverse()
print(lst)
```

## 6.3 Les opérations sur les listes

### 6.3.1 la vie intérieure des listes

Maintenant, nous voulons vous montrer une caractéristique importante, et très surprenante, des listes, qui les distingue fortement des variables ordinaires.

Nous voulons que vous le mémorisiez, cela peut affecter vos programmes futurs et causer de graves problèmes si c'est oublié ou négligé.

Jetez un coup d'œil au code suivant :

```
list_1 = [1]
list_2 = list_1
list_1[0] = 2
print(list_2)
```

Le programme :

- Crée une liste d'un élément nommée list\_1;
- l'affecte à une nouvelle liste nommée list\_2;
- change le seul élément de list\_1;
- imprime list\_2.

La partie surprenante est le fait que le programme produira: [2], pas [1], ce qui semblait être la solution évidente.

Les listes (et de nombreuses autres entités complexes Python) sont stockées de différentes manières que les variables ordinaires (scalaires).

On pourrait dire que :

- Le nom d'une variable ordinaire est le **nom de son contenu**;
- Le nom d'une liste est le nom d'un **emplacement mémoire où la liste est stockée**.

Lisez ces deux lignes une fois de plus, la différence est essentielle pour comprendre de quoi nous allons parler ensuite.

L'assignation: list\_2 = list\_1 copie le nom de la liste, pas son contenu. En effet, les deux noms (list\_1 et list\_2) identifient le même emplacement dans la mémoire de l'ordinateur. Modifier l'un d'eux affecte l'autre, et vice versa.

Heureusement, la solution est à portée de main, son nom est le **slice (la tranche)**. C'est un élément de la syntaxe Python qui vous permet de faire une toute nouvelle copie d'une liste **ou de parties d'une liste**.

Il copie en fait le contenu de la liste, pas le nom de celle-ci.

C'est exactement ce dont nous avons besoin.

```
list_1 = [1]
list_2 = list_1[:]
list_1[0] = 2
print(list_2)
```

Cela imprimera [1].

Cette astuce de code décrite comme [:] est capable de produire une toute nouvelle liste.

L'une des formes les plus générales du slice se présente comme suit:

```
my_list[start:end]
```

Comme vous pouvez le voir, cela ressemble à l'indexation, mais le deux-points à l'intérieur fait une grande différence.

Un slice de cette forme **crée une nouvelle liste (cible), en prenant des éléments de la liste source : les éléments des indices du début à end-1.**

Utilisation de valeurs négatives :

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:3]
print(new_list)
```

La liste new\_list aura des éléments de end - start (3 - 1 = 2) - ceux avec des indices égaux à 1 et 2 donc mais pas 3.

```
# Copying the entire list.
list_1 = [1]
list_2 = list_1[:]
list_1[0] = 2
print(list_2)
# Copying some part of the list.
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:3]
print(new_list)
```

La sortie de l'extrait est : `[8, 6]`

### 6.3.2 Les indices négatives dans le slice

Regardez l'extrait ci-dessous:

```
my_list[start:end]
```

Pour rappeler :

- **start** est l'indice du premier élément **inclus dans la tranche**;
- **end** est l'index du premier élément **non inclus dans la tranche**.

Voici comment les **indices négatifs** fonctionnent :

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:-1]
print(new_list)
```

La sortie de l'extrait est : `[8, 6, 4]`

Si le début spécifie un élément situé plus loin que celui décrit par end(du point de vue du début de la liste), la tranche sera **vide**:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[-1:1]
print(new_list)
```

La sortie de l'extrait est : []

### 6.3.3 Spécificités du slice, les omissions.

Si vous omettez le **début** dans votre tranche, il est supposé que vous souhaitez obtenir une tranche commençant à l'élément avec l'index **0**.

En d'autres termes, la tranche de ce formulaire :

`my_list[:end]` est un équivalent plus compact que : `my_list[0:end]`

Regardez l'extrait ci-dessous:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[:3]
print(new_list)
```

Sa sortie est : [10, 8, 6].

De même, si vous omettez **end** de votre tranche, il est supposé que vous souhaitez que la tranche se termine à l'élément avec l'index **len(my\_list)**. En d'autres termes, la tranche de ce formulaire :

`my_list[start:]` est un équivalent à : `my_list[start:len(my_list)]`

Regardez l'extrait suivant :

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[3:]
print(new_list)
```

Son résultat est : [4, 2].

### 6.3.4 Spécificités du slice avec del

L'instruction `del` est capable de **Supprimer plus qu'un élément de liste à la fois, il peut également supprimer des tranches**:

```
my_list = [10, 8, 6, 4, 2]
del my_list[1:3]
print(my_list)
```

Remarque: dans ce cas, la tranche **ne produit aucune nouvelle liste!**

La sortie est : [10, 4, 2]

Supprimer **tous les éléments** à la fois est également possible:

```
my_list = [10, 8, 6, 4, 2]
del my_list[:]
print(my_list)
```

La liste devient vide et la sortie est : `[]`.

Attention à la manière d'écrire votre code, l'omission de la tranche comme ci-dessous changera drastiquement votre résultat :

```
my_list = [10, 8, 6, 4, 2]
del my_list
print(my_list)
```

L'instruction **del** **supprimera la liste elle-même, pas son contenu**.  
Le `print()` provoquera alors une erreur d'exécution.

### 6.3.5 Les opérateurs `in` et `not in`

Python propose deux opérateurs très puissants, capables de parcourir la liste **afin de vérifier si une valeur spécifique est stockée dans celle-ci ou non**.

Ces opérateurs sont :

```
elem in my_list
elem not in my_list
```

Le premier d'entre eux (**in**) vérifie si un élément donné (son argument gauche) est actuellement stocké quelque part dans la liste (l'argument droit), l'opérateur renvoie **True** dans ce cas.

Le second (**not in**) vérifie si un élément donné (son argument gauche) est absent dans une liste, l'opérateur renvoie **True** dans ce cas.

Regardez le code ci-dessous, l'extrait montre les deux opérateurs en action.  
Quel sera le résultat ?

```
my_list = [0, 3, 12, 8, 2]

print(5 in my_list)
print(5 not in my_list)
print(12 in my_list)
```

### 6.3.6 Quelques exemples de programmes

Maintenant, nous voulons vous montrer quelques programmes simples utilisant des listes.

- Le premier d'entre eux essaie de trouver la plus grande valeur dans la liste. Regardez et analysez le code ci-dessous.

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]

for i in range(1, len(my_list)):
    if my_list[i] > largest:
        largest = my_list[i]

print(largest)
```

Le concept est plutôt simple, nous supposons temporairement que le premier élément est le plus grand et vérifions l'hypothèse par rapport à tous les éléments restants de la liste. Le code génère 17 (comme prévu).

Ce code peut être réécrit pour utiliser la nouvelle forme du for boucle:

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]

for i in my_list:
    if i > largest:
        largest = i

print(largest)
```

Le programme ci-dessus effectue une comparaison inutile, lorsque le premier élément est comparé à lui-même, mais ce n'est pas un problème du tout. Les sorties de code 17, aussi.

Si vous avez besoin d'économiser de l'énergie de l'ordinateur, vous pouvez utiliser une tranche :

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]

for i in my_list[1:]:
    if i > largest:
        largest = i

print(largest)
```

La question est : laquelle de ces deux actions consomme le plus de ressources informatiques, une seule comparaison, ou découper presque tous les éléments d'une liste ?

Trouvons maintenant l'emplacement d'un élément donné dans une liste:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
to_find = 5
found = False

for i in range(len(my_list)):
    found = my_list[i] == to_find
    if found:
        break

if found:
    print("Element found at index", i)
else:
    print("absent")
```

Remarque:

- La valeur cible est stockée dans la variable `to_find` ;
- L'état actuel de la recherche est stocké dans la variable `found` (Vrai/Faux)
- quand `found` devient Vrai le `for` est terminé.

Supposons que vous ayez choisi les numéros suivants à la loterie: 3, 7, 11, 42, 34, 49.

Les chiffres qui ont été tirés sont les suivants : 5, 11, 9, 42, 3, 49.

La question est: combien de chiffres avez-vous en commun?

Ce programme vous donnera la réponse:

```
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
    if number in drawn:
        hits += 1

print(hits)
```

Remarque :

- La liste `drawn` stocke tous les numéros tirés;
- La liste `bets` stocke vos paris ;
- La variable `hits` compte vos nombres communs.
- Le programme donne : 4.



## 6.3.7 Exercice

## Temps estimé

10-15 minutes

## Objectifs

Familiariser l'élève avec :

- indexation des listes;
- en utilisant les opérateurs `in` et `not in`.

## Scénario

Imaginez une liste de matricule d'étudiants, pas très longue, pas très compliquée, juste une simple liste contenant des nombres entiers. Certains de ces chiffres peuvent être répétés. Nous ne voulons pas de répétitions. Nous voulons qu'ils soient supprimés.

Votre tâche est d'écrire un programme qui supprime toutes les répétitions de numéros de la liste. L'objectif est d'avoir une liste dans laquelle tous les numéros n'apparaissent pas plus d'une fois. Un étudiant ne pouvant être inscrit qu'une fois.

Remarque: supposons que la liste sources soit codée en dur à l'intérieur du code, vous n'avez pas besoin de l'entrer à partir du clavier. Bien sûr, vous pouvez améliorer le code et ajouter une partie qui peut mener une conversation avec l'utilisateur et obtenir toutes les données mais simplifions-nous la vie.

Astuce: nous vous encourageons à créer une nouvelle liste en tant que tampon. Nous n'avons fourni aucune donnée de test, car ce serait trop facile. Vous pouvez utiliser notre squelette à la place.

```
my_list = [1, 2, 4, 4, 1, 4, 2, 6, 2, 9]
#
# Write your code here.
#
print("The list with unique elements only:")
print(my_list)
```

## 6.3.8 Résumé

1. Si vous avez une liste `l1`, alors l'affectation suivante : `l2 = l1` ne fait pas une copie de la liste `l1`, mais fait pointer les variables `l1` et `l2` **vers une seule et même liste en mémoire**. Par exemple:

```
vehicles_one = ['car', 'bicycle', 'motor']
print(vehicles_one) # sortie: ['car', 'bicycle', 'motor']

vehicles_two = vehicles_one
del vehicles_one[0] # supprime 'car'
print(vehicles_two) # sortie: ['bicycle', 'motor']
```

2. Si vous souhaitez copier une liste ou une partie de la liste, vous pouvez le faire en effectuant **une tranche**:

```
colors = ['rouge', vert, 'orange']

copy_whole_colors = colors[:]
copy_part_colors = colors[0:2]
```

3. Vous pouvez utiliser les **indices négatifs** pour vos tranches. Par exemple:

```
sample_list = ["A", "B", "C", "D", "E"]
new_list = sample_list[2:-1]
print(new_list) # outputs: ['C', 'D']
```

4. start et end, les paramètres de vos tranches sont **optionnels**. Lors de l'exécution d'une tranche : liste[start:end].Par exemple :

```
my_list = [1, 2, 3, 4, 5]
slice_one = my_list[2: ]
slice_two = my_list[ :2]
slice_three = my_list[-2: ]

print(slice_one) # sortie: [3, 4, 5]
print(slice_two) # sortie: [1, 2]
print(slice_three) # sortie: [4, 5]
```

5. Vous pouvez **supprimer des tranches** à l'aide de l'instruction del :

```
my_list = [1, 2, 3, 4, 5]
del my_list[0:2]
print(my_list) # sortie: [3, 4, 5]

del my_list[:]
print(my_list) # sortie: []
```

6. Vous pouvez tester si certains éléments **existe dans une liste ou non** Utilisation des mots-clés in et not in.Par exemple :

```
my_list = ["A", "B", 1, 2]

print("A" in my_list) # sortie: True
print("C" not in my_list) # sortie: True
print(2 not in my_list) # sortie: False
```

### Exercice 1

Quelle est la sortie de l'extrait de code suivant ?

```
list_1 = ["A", "B", "C"]
list_2 = list_1
list_3 = list_2

del list_1[0]
del list_2[0]

print(list_3)
```

**Exercice 2**

Quelle est la sortie de l'extrait de code suivant ?

```
list_1 = ["A", "B", "C"]
list_2 = list_1
list_3 = list_2
```

```
del list_1[0]
del list_2
```

```
print(list_3)
```

**Exercice 3**

Quelle est la sortie de l'extrait de code suivant ?

```
list_1 = ["A", "B", "C"]
list_2 = list_1
list_3 = list_2
```

```
del list_1[0]
del list_2[:]
```

```
print(list_3)
```

**Exercice 4**

Quelle est la sortie de l'extrait de code suivant ?

```
list_1 = ["A", "B", "C"]
list_2 = list_1[:]
list_3 = list_2[:]
```

```
del list_1[0]
del list_2[0]
```

```
print(list_3)
```

**Exercice 5**

Insérer in ou not in à la place de ??? afin que le code produise le résultat attendu.

```
my_list = [1, 2, "in", True, "ABC"]

print(1 ??? my_list) # sortie True
print("A" ??? my_list) # sortie True
print(3 ??? my_list) # sortie True
print(False ??? my_list) # sortie False
```

## 6.4 Listes multidimensionnelles

### 6.4.1 Une liste dans une liste

Les listes peuvent être constituées de scalaires et d'éléments d'une structure beaucoup plus complexe (vous avez déjà vu des exemples tels que des chaînes, des booléens ou même d'autres listes).

Examinons de plus près le cas où un **Les éléments d'une liste ne sont que des listes**.

Nous trouvons souvent **de tels tableaux** dans nos vies. Le meilleur exemple est probablement un **échiquier**.

Un échiquier est composé de lignes et de colonnes. Il y a huit lignes et huit colonnes. Chaque colonne est marquée des lettres A à H. Chaque ligne est marquée d'un nombre de un à huit.

L'emplacement de chaque pièce est identifié par des paires lettre-chiffre. Ainsi, nous savons que le coin inférieur gauche du plateau (celui avec la tour blanche) est A1, tandis que le coin opposé est H8.

Supposons que nous soyons capables d'utiliser les nombres sélectionnés pour représenter n'importe quelle pièce d'échecs. Nous pouvons également supposer que **chaque ligne de l'échiquier est une liste**.

Regardez le code ci-dessous :

```
row = []

for i in range(8):
    row.append(WHITE_PAWN)
```

Il construit une liste contenant huit éléments représentant la deuxième rangée de l'échiquier, celle remplie de pions (supposons que `WHITE_PAWN` est un **symbole prédéfini** représentant un pion blanc).

Le même effet peut être obtenu au moyen d'une **list comprehension**, la syntaxe spéciale utilisée par Python afin de remplir des listes massives. Une « list comprehension » est en fait une liste, mais **créé à la volée pendant l'exécution du programme et non pas décrit statiquement**.

```
row = [WHITE_PAWN for i in range(8)]
```

La partie du code placée entre parenthèses précise :

- les données à utiliser pour remplir la liste (`WHITE_PAWN`)
- la clause spécifiant combien de fois les données apparaissent dans la liste (`for i in range(8)`)

Laissez-nous vous montrer d'autres **Exemples de « list comprehension »** :

Exemple #1 :

```
squares = [x ** 2 for x in range(10)]
```

L'extrait produit une liste de dix éléments qui sont les carrés de dix nombres entiers à partir de zéro (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

Exemple #2 :

```
twos = [2 ** i for i in range(8)]
```

L'extrait crée un tableau à huit éléments contenant les huit premières puissances de deux (1, 2, 4, 8, 16, 32, 64, 128)

Exemple #3 :

```
odds = [x for x in squares if x % 2 != 0]
```

L'extrait fait une liste avec seulement les éléments impairs de la liste **squares** .

#### 6.4.2 Passons en 2 D

Supposons également qu'un symbole prédéfini nommé **EMPTY** (vide) désigne un champ vide sur l'échiquier.

Donc, si nous voulons créer une liste de listes représentant l'ensemble de l'échiquier, cela peut être fait de la manière suivante:

```
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)
```

Remarques:

- la partie intérieure de la boucle crée une ligne composée de huit éléments (chacun d'eux égal à **VIDE**) et l'ajoute à la liste du `board`;
- la partie extérieure le répète huit fois;
- au total, la liste du tableau se compose de 64 éléments (tous égaux à **EMPTY**)

Ce modèle imite parfaitement le véritable échiquier, qui est en fait une liste de huit éléments, tous étant des rangées uniques.

Résumons nos observations :

- Les éléments des lignes sont des champs, huit par ligne ;
- Les éléments de l'échiquier sont des rangées, huit par échiquier.

La variable **board** est maintenant une **Tableau bidimensionnel**. Elle est également appelée, par analogie aux termes algébriques, une **matrice**.

Comme les « list comprehension » peuvent être **imbriquées**, nous pouvons raccourcir la création du tableau de la manière suivante:

```
board = [[EMPTY for i in range(8)] for j in range(8)]
```

La partie interne crée une ligne et la partie externe crée une liste de lignes.

L'accès à la case sélectionnée du plateau nécessite deux index, le premier sélectionne la ligne et le second, le numéro de case à l'intérieur de la ligne, qui est de facto un numéro de colonne.

Jetez un coup d'œil à l'échiquier. Chaque champ contient une paire d'index qui doivent être donnés pour accéder au contenu du champ:

	A	B	C	D	E	F	G	H	
8	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]	8
7	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]	7
6	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[2][6]	[2][7]	6
5	[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[3][6]	[3][7]	5
4	[4][0]	[4][1]	[4][2]	[4][3]	[4][4]	[4][5]	[4][6]	[4][7]	4
3	[5][0]	[5][1]	[5][2]	[5][3]	[5][4]	[5][5]	[5][6]	[5][7]	3
2	[6][0]	[6][1]	[6][2]	[6][3]	[6][4]	[6][5]	[6][6]	[6][7]	2
1	[7][0]	[7][1]	[7][2]	[7][3]	[7][4]	[7][5]	[7][6]	[7][7]	1
	A	B	C	D	E	F	G	H	

En jetant un coup d'œil à la figure ci-dessus, plaçons quelques pièces d'échecs sur le plateau. Tout d'abord, ajoutons toutes les tours:

```
board[0][0] = ROOK
board[0][7] = ROOK
board[7][0] = ROOK
board[7][7] = ROOK
```

Si vous souhaitez ajouter un chevalier à C4, procédez comme suit :

```
board[4][2] = KNIGHT
```

Et maintenant un pion en E5 :

```
board[3][4] = PAWN
```

Et maintenant, expérimentez avec le code suivant :

```
EMPTY = "-"
ROOK = "ROOK"
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)

board[0][0] = ROOK
board[0][7] = ROOK
board[7][0] = ROOK
board[7][7] = ROOK

print(board)
```

#### 6.4.3 Applications avancées du multidimensionnel

Approfondissons la nature multidimensionnelle des listes. Pour trouver un élément d'une liste bidimensionnelle, vous devez utiliser deux *coordonnées* :

- vertical (numéro de ligne)
- horizontal (numéro de colonne).

Imaginez que vous développiez un logiciel pour une station météo automatique. L'appareil enregistre la température de l'air sur une base horaire et le fait tout au long du mois. Cela vous donne un total de  $24 \times 31 = 744$  valeurs. Essayons de concevoir une liste capable de stocker tous ces résultats.

Tout d'abord, vous devez décider quel type de données serait adéquat pour cette application.

Dans ce cas, un `float` serait préférable, car ce thermomètre est capable de mesurer la température avec une précision de 0,1°C.

Ensuite, vous prenez une décision arbitraire, que les lignes enregistreront les lectures toutes les heures (donc la ligne aura 24 éléments) et chacune de ces lignes sera affectée à un jour du mois (supposons que chaque mois a 31 jours, donc vous avez besoin de 31 lignes).

Voici la paire appropriée :

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

Toute la matrice est remplie de zéros.

Vous pouvez supposer qu'elle est mise à jour automatiquement à l'aide d'agents. La chose que vous devez faire est d'attendre que la matrice soit remplie de mesures.

Il est maintenant temps de déterminer la température moyenne mensuelle à midi. Additionnez les 31 lectures enregistrées à midi et divisez la somme par 31. Vous pouvez par exemple, supposer que la température de minuit est stockée en premier:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# la magie de l'update de la matrice se place ici.
#

total = 0.0

for day in temps:
    total += day[11]

average = total / 31
print("Température moyenne à midi:", average)
```

Remarque: la variable de `day` utilisée par la boucle `for` n'est pas un scalaire, chaque passage à travers la matrice `temps` l'assigne avec les lignes suivantes de la matrice : une liste. Elle doit être indexé à `11` pour accéder à la valeur de température mesurée à midi.

Maintenant, trouvez la température la plus élevée pendant tout le mois:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# la magie de l'update de la matrice se place ici.
#

highest = -100.0

for day in temps:
    for temp in day:
        if temp > highest:
            highest = temp

print("La t° la plus élevée était :", highest)
```

Remarque:

- La variable `day` parcourt toutes les lignes de la matrice `temps` ;
- La variable `temp` parcourt toutes les mesures prises en une journée.

Maintenant, comptez les jours où la température à midi était d'au moins 20 °C:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# la magie de l'update de la matrice se place ici.
#

hot_days = 0

for day in temps:
    if day[11] > 20.0:
        hot_days += 1

print(hot_days, "jours chaud.")
```



#### 6.4.4 Le tridimensionnel

Python ne limite pas la profondeur de l'inclusion de liste dans une autre liste.

Imaginez un hôtel. C'est un immense hôtel composé de trois bâtiments de 15 étages chacun. Il y a 20 chambres à chaque étage. Pour cela, vous avez besoin d'un tableau capable de collecter et de traiter des informations sur les chambres occupées/libres.

- Première étape : le type des éléments du tableau. Dans ce cas, une valeur booléenne (True/False) conviendrait.
- Deuxième étape : analyse calme de la situation. Résumez les informations disponibles : trois bâtiments, 15 étages, 20 chambres.

Vous pouvez maintenant créer le tableau :

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]
```

Le premier index (0 à 2) sélectionne l'un des bâtiments; le second (0 à 15) sélectionne l'étage, le troisième (0 à 19) sélectionne le numéro de pièce. Toutes les chambres sont initialement vides.

Maintenant, vous pouvez réserver une chambre pour deux jeunes mariés: dans le deuxième bâtiment, au dixième étage, chambre 14:

```
rooms[1][9][13] = True
```

et libérer la deuxième chambre au cinquième étage située dans le premier bâtiment:

```
rooms[0][4][1] = False
```

Vérifie-s'il y a des chambres vacantes au 15e étage du troisième bâtiment :

```
vacancy = 0
```

```
for room_number in range(20):  
    if not rooms[2][14][room_number]:  
        vacancy += 1
```

La variable de `vacancy` contient 0 si toutes les chambres sont occupées, ou le nombre de chambres disponibles dans le cas contraire.

## 6.4 Résumé

1. **List Comprehension** vous permet de créer de nouvelles listes à partir de listes existantes de manière concise et élégante. La syntaxe d'une List comprehension se présente comme suit :

```
[expression for element in list if conditional]
```

qui est en fait un équivalent du code suivant :

```
for element in list:
    if conditional:
        expression
```

Voici un exemple de code qui crée une liste de cinq éléments remplie avec les cinq premiers nombres naturels élevés à la puissance 3 :

```
cubed = [num ** 3 for num in range(5)]
print(cubed) # sortie: [0, 1, 8, 27, 64]
```

2. Vous pouvez utiliser des **listes imbriquées** en Python pour créer des **Matrices** (c.-à-d. listes bidimensionnelles). Par exemple:

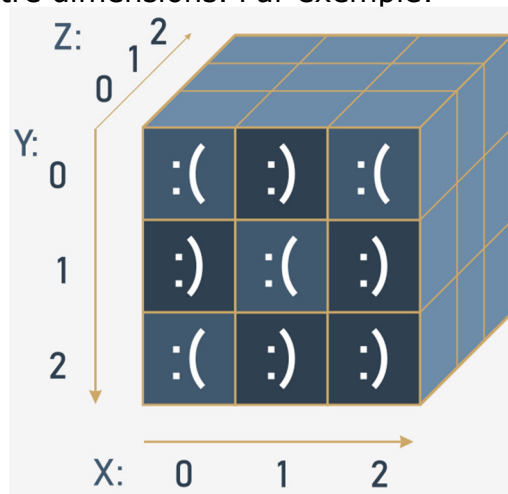
Y: 0	: (	: )	: (	: )
1	: )	: (	: )	: )
2	: (	: )	: )	: (
3	: )	: )	: )	: (
	X: 0	1	2	3

```
# A four-column/four-row table - a two dimensional array (4x4)
```

```
table = [[:("(", ":)"), ":(", ":)"),
          [":)", ":((", ":)", ":)"),
          [":(", ":)", ":)", ":("),
          [":)", ":)", ":)", ":(")]
```

```
print(table)
print(table[0][0]) # sortie: ':(('
print(table[0][3]) # sortie: ':)'
```

3. Vous pouvez imbriquer autant de listes dans des listes que vous le souhaitez, et donc créer des listes à n dimensions, par exemple, des tableaux à trois, quatre ou même soixante-quatre dimensions. Par exemple:



```
# Cube - a three-dimensional array (3x3x3)
```

```
cube = [[[':( ', 'x', 'x'],
          [':) ', 'x', 'x'],
          [':( ', 'x', 'x']],

        [[':) ', 'x', 'x'],
          [':( ', 'x', 'x'],
          [':) ', 'x', 'x']],

        [[':( ', 'x', 'x'],
          [':) ', 'x', 'x'],
          [':) ', 'x', 'x']]]
```

```
print(cube)
print(cube[0][0][0]) # sortie: ':( '
print(cube[2][2][0]) # sortie: ':) '
```

## 7. Les fonctions

### 7.1 Les fonctions

#### 7.1.1 Pourquoi en avons-nous besoin ?

Vous avez rencontré des **fonctions** à plusieurs reprises jusqu'à présent, mais le point de vue que nous vous en avons donné était plutôt unilatéral. Vous n'avez invoqué les fonctions qu'en les utilisant comme des outils pour vous faciliter la vie et simplifier les tâches fastidieuses et chronophages.

Lorsque vous souhaitez que certaines données soient imprimées sur la console, vous utilisez `print()`. Lorsque vous souhaitez lire la valeur d'une variable, vous utilisez `input()`, couplé avec `int()` ou `float()`.

Vous avez également utilisé certaines **méthodes**, qui sont en fait des fonctions, mais déclarées d'une manière très spécifique. Maintenant, vous allez apprendre à écrire vos propres fonctions et à les utiliser. Nous allons écrire plusieurs fonctions ensemble, du très simple au plus complexe.

Il arrive souvent qu'un morceau de code particulier soit **répété plusieurs fois dans votre programme**. Il est répété soit littéralement, soit avec seulement quelques modifications mineures, consistant en l'utilisation d'autres variables dans le même algorithme.

Il arrive également qu'un programmeur ne puisse résister à simplifier le travail et commence à cloner de tels morceaux de code.

Cela peut finir par être frustrant quand soudainement il s'avère qu'il y avait une erreur dans le code cloné. Le programmeur aura alors beaucoup de mal pour trouver tous les endroits qui ont besoin de corrections. Il y a aussi un risque élevé que les corrections provoquent des erreurs.

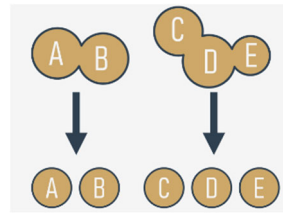
Nous pouvons définir la première condition qui peut vous aider à décider quand commencer à écrire vos propres fonctions : **si un fragment particulier du code commence à apparaître à plus d'un endroit, envisagez la possibilité de l'isoler sous la forme d'une fonction**.

Il peut arriver que l'algorithme que vous allez implémenter soit si complexe que votre code commence à croître de manière incontrôlée, et soudainement vous remarquez que vous n'êtes plus capable de naviguer dedans si facilement. Vous pouvez essayer de résoudre le problème en commentant le code de manière approfondie, mais vous constaterez rapidement que cela aggrave considérablement votre situation, **trop de commentaires rendent le code plus volumineux et plus difficile à lire**.

Certains disent qu'une **fonction bien écrite devrait être vue entièrement en un coup d'œil**.

Un bon et attentif **développeur divise le code** (ou plus précisément : le problème) en morceaux bien isolés, et **encode chacun d'eux sous la forme d'une fonction**.

Cela simplifie considérablement le travail du programmeur, car chaque morceau de code peut être encodé séparément et testé séparément. Le processus décrit ici est souvent appelé **décomposition**.



Nous pouvons maintenant énoncer la deuxième condition: si un **morceau de code devient si volumineux que sa lecture et sa sous-estimation peuvent provoquer un problème, envisagez de le diviser en problèmes distincts et plus petits, et implémentez chacun de ceux-ci sous la forme d'une fonction distincte.**

### 7.1.2 La décomposition

Il arrive souvent que le problème soit si vaste et complexe qu'il ne peut pas être attribué à un seul développeur, et une **équipe de développeurs** doit travailler dessus. Le problème doit être réparti entre plusieurs développeurs de manière à assurer leur coopération efficace et transparente.



Il semble inconcevable que plus d'un programmeur écrive le même morceau de code en même temps, de sorte que le travail doit être réparti entre tous les membres de l'équipe.

Ce type de décomposition a un objectif différent de celui décrit précédemment, il ne s'agit pas seulement de partager **le travail**, mais aussi de **partager la responsabilité** entre de nombreux développeurs.

Chacun d'eux écrit un ensemble de fonctions clairement définies et décrites qui, une fois **combinées dans le module** (nous vous en parlerons un peu plus tard), donneront le produit final.

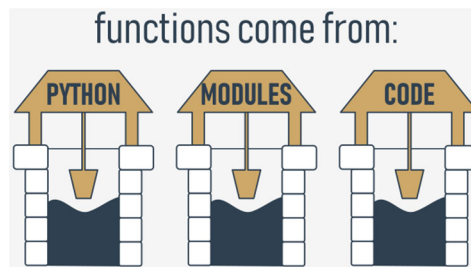
Cela nous amène directement à la troisième condition: si vous divisez le travail entre plusieurs programmeurs, **décomposez le problème pour permettre au produit d'être implémenté comme un ensemble de fonctions écrites séparément et emballées ensemble dans différents modules.**

### 7.1.3 Mais d'où viennent les fonctions ?

En général, les fonctions proviennent d'au moins trois endroits:

- De Python lui-même, de nombreuses fonctions (comme `print()`) font **partie intégrante de Python** et sont toujours disponibles sans effort supplémentaire de la part du programmeur; nous appelons ces fonctions **des fonctions intégrées**;
- A partir des modules **préinstallés** de Python, beaucoup de fonctions, très utiles, mais utilisées beaucoup moins souvent que celles intégrées, sont disponibles dans un certain nombre de modules installés avec Python; l'utilisation de ces fonctions nécessite quelques étapes supplémentaires de la part du programmeur afin de les rendre entièrement accessibles;
- **Directement à partir de votre code**, vous pouvez écrire vos propres fonctions, les placer dans votre code et les utiliser librement;

Il y a une autre possibilité, mais elle liée aux classes, nous allons donc l'omettre pour l'instant et la laisser pour le Q2.



### 7.1.4 Première fonction

Jetez un coup d'œil à l'extrait de code :

```
print("Enter a value: ")
a = int(input())

print("Enter a value: ")
b = int(input())

print("Enter a value: ")
c = int(input())
```

C'est assez simple, mais nous voulons seulement que ce soit un exemple de **transformation d'une partie répétitive d'un code en fonction**.

Les messages envoyés à la console par la fonction `print()` sont toujours les mêmes. Bien sûr, il n'y a rien de vraiment mauvais dans un tel code, mais essayez d'imaginer ce que vous auriez à faire si votre patron vous demandait de modifier le message pour le rendre plus poli, par exemple, de le commencer par « S'il vous plaît ».

Il semble que vous devriez passer un peu de temps à changer toutes les occurrences du message (vous utiliseriez un presse-papiers, bien sûr, mais cela ne vous faciliterait pas la vie).

Il est évident que vous feriez probablement des erreurs pendant le processus d'amendement, et vous (et votre patron) seriez un peu frustré.

Est-il possible de séparer une telle partie *reproductible* du code, de la nommer et de la rendre réutilisable ? Cela signifierait qu'un **changement effectué une fois au même endroit serait propagé à tous les endroits où il est utilisé**.

Bien sûr, un tel code ne devrait fonctionner que lorsqu'il est explicitement lancé. Et évidemment c'est possible. C'est exactement à cela que servent les fonctions.

Comment faire une telle fonction?

Vous devez **la définir**. Le mot *define* sera utilisé et est significatif ici.

Voici à quoi ressemble la définition de fonction la plus simple :

```
def function_name():  
    function_body
```

- Toujours commencer par le **mot-clé** `def` (pour *définir*)
- Suive après `def` le **nom de la fonction** (les règles pour nommer les fonctions sont exactement les mêmes que pour nommer les variables)
- Après le nom de la fonction, il y a une place pour une paire de **parenthèses** (elles ne contiennent rien ici, mais cela changera bientôt)
- La ligne doit être terminée par deux **points**;
- La ligne directement après `def` commence le **corps** de la fonction, un couple (au moins un) d'instructions est **imbriquées**, elles seront exécutées chaque fois que la fonction est appelée;

Remarque: la fonction se **termine là où l'imbriication se termine**, vous devez donc être prudent.

Nous sommes prêts à définir notre fonction **d'impression**. Nous allons la nommer `message`, la voici:

```
def message():  
    print("Enter a value: ")
```

La fonction est extrêmement simple, mais **utilisable**. Nous l'avons nommé `message`, mais vous pouvez l'étiqueter selon vos goûts.

Notre code contient maintenant la définition de la fonction :

```
def message():  
    print("Enter a value: ")  
  
print("We start here.")  
print("We end here.")
```

Remarque: nous n'utilisons pas du tout la fonction - il n'y a pas **d'invocation** de celle-ci dans le code.

Lorsque vous l'exécutez, vous voyez la sortie suivante :

```
Nous commençons ici.
Nous nous arrêtons ici.
```

Cela signifie que Python lit les définitions de la fonction et s'en souvient, mais ne lancera aucune d'entre elles sans votre permission.

Nous avons modifié le code maintenant, nous avons inséré **l'appel** de la fonction entre les messages de début et de fin:

```
def message():
    print("Enter a value: ")

print("We start here.")
message()
print("We end here.")
```

La sortie est différente maintenant:

```
Nous commençons ici.
Entrez une valeur :
Nous nous arrêtons ici.
```

#### 7.1.5 Allons plus loin dans le fonctionnement

Regardez l'image ci-dessous:



Cette image essaie de vous montrer l'ensemble du processus:

- lorsque vous **appelez** une fonction, Python se souvient de l'endroit où cela s'est produit et **saute** dans la fonction invoquée ;
- le corps de la fonction est ensuite **exécuté** ;
- atteindre la fin de la fonction force Python à **revenir** à l'endroit directement après le point d'appel.

Il y a deux informations très importantes. Voici le premier d'entre eux :

**Vous ne devez pas invoquer une fonction qui n'est pas connue au moment de l'appel.**

N'oubliez pas, Python lit votre code de haut en bas. Il ne va pas regarder vers l'avenir afin de trouver une fonction que vous avez oublié de mettre au bon endroit (« bon » signifie « avant l'invocation ».)

Nous avons inséré une erreur dans ce code, voyez-vous la différence?

```
print("We start here.")
message()
print("We end here.")
def message():
    print("Enter a value: ")
```



Nous avons déplacé la fonction à la fin du code. Python est-il capable de le trouver lorsque l'exécution atteint l'invocation ?

Non, ce n'est pas le cas. Le message d'erreur se lit comme suit :

```
NameError: name 'message' is not defined
```

N'essayez pas de forcer Python à rechercher des fonctions que vous n'avez pas encodées au bon moment.

La deuxième information semble un peu plus simple:

**Vous ne devez pas avoir une fonction et une variable du même nom.**

L'extrait de code suivant est erroné :

```
def message():  
    print("Enter a value: ")  
  
message = 1
```

L'attribution d'une valeur à la variable message fait oublier à Python son rôle précédent. La fonction nommée message devient indisponible.

Heureusement, vous êtes libre de mélanger votre code avec les fonctions, vous n'êtes pas obligé de mettre toutes vos fonctions en haut de votre fichier source. Regardez l'extrait :

```
print("We start here.")  
  
def message():  
    print("Enter a value: ")  
message()  
print("We end here.")
```

Cela peut sembler étrange, mais c'est tout à fait correct et fonctionne comme prévu.

Revenons à notre exemple principal, et utilisons la fonction pour le bon travail, comme ici:

```
def message():  
    print("Enter a value: ")  
  
message()  
a = int(input())  
message()  
b = int(input())  
message()  
c = int(input())
```

La modification du message d'impression est maintenant facile et claire - vous pouvez le faire en **changeant le code en un seul endroit** - à l'intérieur du corps de la fonction.

### 7.1.6 Résumé

1. Une fonction est un bloc de code qui effectue une tâche spécifique lorsque la fonction est appelée (invoquée). Vous pouvez utiliser des fonctions pour rendre votre code réutilisable, mieux organisé et plus lisible. Les fonctions peuvent avoir des paramètres et des valeurs de retour.

2. Il existe au moins quatre types de fonctions de base dans Python :

- **fonctions intégrées** qui font partie intégrante de Python (telles que la fonction `print()`). Vous pouvez voir une liste complète des fonctions intégrées de Python à <https://docs.python.org/3/library/functions.html>.
- ceux qui proviennent de **modules préinstallés**
- **fonctions définies par l'utilisateur** qui sont écrites par les utilisateurs pour les utilisateurs.
- les fonctions **lambda** (On verra cela plus tard).

3. Vous pouvez définir votre propre fonction en utilisant le mot-clé `def` et la syntaxe suivante:

```
def your_function(optional_parameters):
    # the body of the function
```

Vous pouvez définir une fonction qui ne prend aucun argument, par exemple :

```
def message():      # defining a function
    print("Hello")   # body of the function

message()           # calling the function
```

Vous pouvez également définir une fonction qui prend des arguments, tout comme la fonction à un paramètre ci-dessous :

```
def hello(name):    # defining a function
    print("Hello,", name) # body of the function

name = input("Enter your name: ")
hello(name)
```

#### Exercice 1

La fonction `input()` est un exemple de:

- fonction définie par l'utilisateur
- fonction intégrée

#### Exercice 2

Que se passe-t-il lorsque vous essayez d'appeler une fonction avant de la définir?

```
hi()
def hi():
    print("hi!")
```

#### Exercice 3

Que se passera-t-il lorsque vous exécuterez le code ci-dessous ?

```
def hi():
    print("hi")
hi(5)
```

## 7.2 Les fonctions paramétrées

La pleine puissance de la fonction se révèle lorsqu'elle peut être équipée d'une interface capable d'accepter les données fournies par l'appelant. Ces données peuvent modifier le comportement de la fonction, la rendant plus flexible et adaptable aux conditions changeantes.

Un paramètre est en fait une variable, mais il y a deux facteurs importants qui rendent les paramètres différents et spéciaux :

- Les paramètres n'existent qu'à l'intérieur des fonctions dans lesquelles ils ont été définis, et le seul endroit où le paramètre peut être défini est un espace entre une paire de parenthèses dans l'instruction **def**;
- L'attribution d'une valeur au paramètre se fait au moment de l'appel de la fonction, en spécifiant l'argument correspondant.

```
def fonction(parametre) :  
    ###
```

N'oubliez pas:

- Les paramètres vivent à l'intérieur des fonctions (c'est leur environnement naturel)
- Les arguments existent en dehors des fonctions et sont porteurs de valeurs passées aux paramètres correspondants.

Il existe une frontière claire et sans ambiguïté entre ces deux mondes.

Enrichissons la fonction ci-dessus avec un seul paramètre, nous allons l'utiliser pour montrer à l'utilisateur le numéro d'une valeur demandée par la fonction.

Nous devons reconstruire la déclaration **def** : voici à quoi elle ressemble maintenant:

```
def message(number) :  
    ###
```

La définition précise que notre fonction fonctionne sur un seul paramètre nommé **number**. Vous pouvez l'utiliser comme une variable ordinaire, mais seulement à l'intérieur de la fonction, elle n'est visible nulle part ailleurs.

Améliorons maintenant le corps de la fonction:

```
def message(number) :  
    print("Entrez un nombre:", number)
```

Nous avons utilisé le paramètre.

**Remarque :** nous n'avons attribué aucune valeur au paramètre.

Une valeur pour le paramètre arrivera de l'environnement de la fonction.

N'oubliez pas : la spécification d'un ou plusieurs paramètres dans la définition d'une fonction est également une exigence, et vous devez la remplir lors de l'appel. Vous devez fournir autant d'arguments qu'il y a de paramètres définis.

Si vous ne le faites pas, une erreur sera commise.

Essayez d'exécuter le code ci-dessous :

```
def message(number):  
    print("Enter a number:", number)  
  
message()
```

Voici ce que vous verrez dans la console :

```
TypeError : message() manquant 1 argument positionnel requis :  
'number'
```

Ceci a l'air mieux:

```
def message(number):  
    print("Enter a number:", number)  
  
message(1)
```

Le code se comportera mieux, il produira la sortie suivante :

```
Entrez un nombre : 1
```

Pouvez-vous comprendre comment cela fonctionne? La valeur de l'argument utilisé lors de l'appel (1) a été passée dans la fonction, définissant la valeur initiale du paramètre nommé number.

Nous devons vous sensibiliser à un contexte important.

Il est légal, et possible, d'avoir une **variable nommée de la même manière que le paramètre d'une fonction**.

L'extrait ci-dessous illustre le phénomène :

```
def message(number):  
    print("Enter a number:", number)  
  
number = 1234  
message(1)  
print(number)
```

Une situation comme celle-ci active un mécanisme appelé **ombrage** :

- Le paramètre x fait de l'ombre à toute variable du même nom, mais...  
... uniquement à l'intérieur de la fonction définissant le paramètre.

Le paramètre nommé number est une entité complètement différente de la variable nommée number.

Cela signifie que l'extrait ci-dessus produira la sortie suivante :

```
Entrez un nombre : 1  
1234
```

Une fonction peut avoir autant de paramètres **que vous le souhaitez**, mais plus vous avez de paramètres, plus il est difficile de mémoriser leurs rôles et leurs objectifs.



Modifions la fonction pour qu'elle ait deux paramètres :

```
def message(what, number):  
    print("Enter", what, "number", number)  
  
# invoke the function
```

Cela signifie également que l'appel de la fonction nécessitera **deux arguments**.

Le premier nouveau paramètre est destiné à porter le nom de la valeur souhaitée :

```
def message(what, number):  
    print("Enter", what, "number", number)  
  
message("telephone", 11)  
message("price", 5)  
message("number", "number")
```

Voici la sortie de cet extrait :

```
Enter telephone number 11  
Enter price number 5  
Enter number number number
```

Exécutez le code, modifiez-le, ajoutez d'autres paramètres et voyez comment cela affecte la sortie.

### 7.2.1 Transmission de paramètres positionnels

Une technique qui assigne le  $i^{\text{ème}}$  argument (premier, deuxième, etc.) au  $i^{\text{ème}}$  paramètre de fonction (premier, deuxième, etc.) est appelée **passage de paramètre positionnel**, tandis que les arguments passés de cette manière sont appelés arguments **positionnels**.

Vous avez déjà utilisé ce principe, mais Python peut offrir beaucoup plus.

```
def my_function(a, b, c):  
    print(a, b, c)
```

```
my_function(1, 2, 3)
```

Remarque: la transmission de paramètres de position est utilisée intuitivement par les gens dans de nombreuses occasions sociales. Par exemple, il peut être généralement admis que lorsque nous nous présentons, nous mentionnons notre/nos prénom(s) avant notre nom de famille, par exemple, « Je m'appelle John Doe ».

Par contre, les Hongrois par exemple le font dans l'ordre inverse.

Implémentons cette coutume sociale en Python. La fonction suivante sera responsable de l'introduction de quelqu'un:

```
def introduction(first_name, last_name):  
    print("Hello, my name is", first_name, last_name)
```

```
introduction("Luke", "Skywalker")  
introduction("Jesse", "Quick")  
introduction("Clark", "Kent")
```

Imaginez maintenant que la même fonction soit utilisée en Hongrie. Dans ce cas, le code ressemblerait à ceci :

```
def introduction(first_name, last_name):  
    print("Hello, my name is", first_name, last_name)
```

```
introduction("Skywalker", "Luke")  
introduction("Quick", "Jesse")  
introduction("Kent", "Clark")
```

La question est maintenant : Pouvez-vous rendre la fonction plus indépendante de la culture?

### 7.2.2 Passage d'arguments par mot-clé

Python offre une autre convention pour passer des arguments, où **la signification de l'argument est dictée par son nom**, pas par sa position : c'est ce qu'on appelle la transmission d'argument **de mot-clé**.

Jetez un coup d'œil à l'extrait:

```
def introduction(first_name, last_name):  
    print("Hello, my name is", first_name, last_name)  
  
introduction(first_name = "James", last_name = "Bond")  
introduction(last_name = "Skywalker", first_name = "Luke")
```

Le concept est clair - les valeurs transmises aux paramètres sont précédées des noms des paramètres cibles, suivis du signe **=**.

La position n'a pas d'importance ici, la valeur de chaque argument connaît sa destination sur la base du nom utilisé.

Vous devriez être en mesure de prédire la sortie du code ci-dessus.

Bien sûr, vous **ne devez pas utiliser un nom de paramètre inexistant**.

L'extrait de code suivant provoquera une erreur d'exécution :

```
def introduction(first_name, last_name):  
    print("Hello, my name is", first_name, last_name)  
  
introduction(surname="Skywalker", first_name="Luke")
```

Voici ce que l'interpréteur Python vous dira :

```
TypeError: introduction() a un argument de mot-clé inattendu  
'surname'
```

### 7.2.3 Mélange d'arguments positionnels et de mots-clés

Vous pouvez mélanger les deux modes si vous voulez - il n'y a qu'une seule règle incassable: vous devez mettre les arguments **positionnels avant les arguments de mots-clés**.

Si vous réfléchissez un instant, vous devinerez certainement pourquoi.

Pour vous montrer comment cela fonctionne, nous utiliserons la fonction simple à trois paramètres suivants:

```
def adding(a, b, c):  
    print(a, "+", b, "+", c, "=", a + b + c)
```

Son but est d'évaluer et de présenter la somme de tous ses arguments.

La fonction, lorsqu'elle est appelée de la manière suivante :

```
adding(1, 2, 3)
```

produira :  $1 + 2 + 3 = 6$

C'est - comme vous pouvez vous en douter - un pur exemple de **passage d'argument positionnel**.

Bien sûr, vous pouvez remplacer un tel appel par une variante de mot-clé pur, comme ceci:

```
adding(c = 1, a = 2, b = 3)
```

Notre programme produira une ligne comme celle-ci:

$2 + 3 + 1 = 6$

Attention, regardez bien l'ordre des valeurs.

Essayons de mélanger les deux styles maintenant. Regardez bien l'appel de fonction ci-dessous:

```
adding(3, c = 1, b = 2)
```

Analysons-le ensemble :

- L'argument (3) pour le paramètre `a` est passé en utilisant la méthode positionnelle;
- Les arguments pour `c` et `b` sont spécifiés en tant que mots-clés.

Voici ce que vous verrez dans la console :

$3 + 2 + 1 = 6$

Soyez prudent et méfiez-vous des erreurs. Si vous essayez de passer plus d'une valeur à un argument, tout ce que vous obtiendrez est une erreur d'exécution.

Regardez l'invocation ci-dessous:

```
adding(3, a = 1, b = 2)
```

Réponse de Python :

*`TypeError: adding() a plusieurs valeurs pour l'argument 'a'`*

Regardez le petit bout de code ci-dessous. Un code comme celui-ci est tout à fait correct, mais cela n'a pas beaucoup de sens :

```
adding(4, 3, c = 2)
```

Tout est correct, mais laisser un seul argument par mot-clé semble un peu bizarre non ? Qu'en pensez-vous?



#### 7.2.4 Fonctions paramétrées, infos complémentaires

Il arrive parfois que les valeurs d'un paramètre particulier soient utilisées plus souvent que d'autres. Ces arguments peuvent voir leurs **valeurs par défaut (prédéfinies)** prises en considération lorsque leurs arguments correspondants ont été omis.

On dit que le nom de famille anglais le plus populaire est *Smith*. Essayons d'en tenir compte.

La valeur du paramètre par défaut est définie à l'aide d'une syntaxe claire et illustrée :

```
def introduction(first_name, last_name="Smith"):
    print("Salut mon nom est", first_name, last_name)
introduction(first_name, last_name="Smith"):
    print("Salut mon nom est", first_name, last_name)
```

Il vous suffit d'étendre le nom du paramètre avec le signe `=`, suivi de la valeur par défaut.

Appelons la fonction comme d'habitude :

```
introduction("James", "Doe")
```

Pouvez-vous deviner le résultat du programme? Exécutez-le et vérifiez si vous aviez raison. Et? Tout se ressemble, mais lorsque vous invoquez la fonction d'une manière qui semble un peu suspecte à première vue, comme ceci:

```
introduction("Henry")
```

ou ceci : `introduction(first_name="William")`

Il n'y aura pas d'erreur et les deux appels réussiront, tandis que la console affichera la sortie suivante :

```
Salut mon nom est Henry Smith
Salut mon nom est William Smith
```

Vous pouvez encore aller plus loin si c'est utile. Les deux paramètres ont maintenant leurs valeurs par défaut, regardez le code ci-dessous:

```
def introduction(first_name="John", last_name="Smith"):
    print("Salut mon nom est", first_name, last_name)
```

Cela rend l'invocation suivante valide : `introduction()`

Et voici le résultat attendu: `Salut mon nom est John Smith`

Si vous utilisez un argument par mot-clé, le reste prendra la valeur par défaut :

```
introduction(last_name="Hopkins")
```

Le résultat est le suivant : `Salut mon nom est John Hopkins`

## 7.2.5 Résumé

1. Vous pouvez transmettre des informations aux fonctions à l'aide de paramètres. Vos fonctions peuvent avoir autant de paramètres que vous le souhaitez.

Exemple de fonction à un paramètre :

```
def hi(name):
    print("Hi,", name)

hi("Greg")
```

Exemple de fonction à deux paramètres :

```
def hi_all(name_1, name_2):
    print("Hi,", name_2)
    print("Hi,", name_1)

hi_all("Sébastien", "Pierre")
```

Exemple de fonction à trois paramètres :

```
def address(street, city, postal_code):
    print("Votre adresse est:", street, "St.", city, postal_code)

s = input("Rue: ")
p_c = input("Code Postal: ")
c = input("Ville: ")
address(s, c, p_c)
```

2. Vous pouvez passer des arguments à une fonction à l'aide des techniques suivantes :

- **Passage d'argument positionnelle** dans laquelle l'ordre des arguments a passé compte (Ex. 1),
- **Passage par mot-clé** dans lequel l'ordre des arguments passés n'a pas d'importance (Ex. 2),
- un mélange d'arguments positionnels et par mots clés (Ex. 3).

**Ex. 1**

```
def subtra(a, b):
    print(a - b)
subtra(5, 2)      # outputs: 3
subtra(2, 5)      # outputs: -3
```

**Ex. 2**

```
def subtra(a, b):
    print(a - b)
subtra(a=5, b=2)   # outputs: 3
subtra(b=2, a=5)   # outputs: 3
```

**Ex. 3**

```
def subtra(a, b):
    print(a - b)
subtra(5, b=2)     # outputs: 3
subtra(5, 2)       # outputs: 3
```

Il est important de se rappeler que les arguments **positionnels ne doivent pas suivre les arguments par mots clés**. C'est pourquoi, si vous essayez d'exécuter l'extrait de code suivant :

```
def subtra(a, b):  
    print(a - b)  
  
subtra(5, b=2)      # outputs: 3  
subtra(a=5, 2)      # Syntax Error
```

Python ne vous laissera pas le faire en signalant une erreur de syntaxe.

3. Vous pouvez utiliser la technique de passage d'argument par **mot-clé pour prédéfinir** une valeur pour un argument donné :

```
def name(first_name, last_name="Smith"):  
    print(first_name, last_name)  
  
name("Andy")        # outputs: Andy Smith  
name("Betty", "Johnson")  # outputs: Betty Johnson
```

### Exercice 1

Quelle est la sortie de l'extrait de code suivant ?

```
def intro(a="James Bond", b="Bond"):  
    print("My name is", b + ".", a + ".")  
  
intro()
```

### Exercice 2

Quelle est la sortie de l'extrait de code suivant ?

```
def intro(a="James Bond", b="Bond"):  
    print("My name is", b + ".", a + ".")  
  
intro(b="Sean Connery")
```

### Exercice 3

Quelle est la sortie de l'extrait de code suivant ?

```
def intro(a, b="Bond"):  
    print("My name is", b + ".", a + ".")  
  
intro("Susan")
```

### Exercice 4

Quelle est la sortie de l'extrait de code suivant ?

```
def add_numbers(a, b=2, c):  
    print(a + b + c)  
  
add_numbers(a=1, c=3)
```

## 7.3 Effets et résultats des fonctions

### 7.3.1 L'instruction de retour

Toutes les fonctions présentées précédemment ont une sorte d'effet - elles produisent du texte et l'envoient à la console. Dans la littérature, c'est ce que l'on appelle une procédure. Quand on cherchera à avoir un retour, on utilisera ici bel et bien le mot fonction et comme leurs frères et sœurs mathématiques, elles peuvent avoir des résultats.

Pour obtenir **des fonctions qui renvoient une valeur** (mais pas seulement à cette fin), vous utilisez l'instruction **return**.

Ce mot vous donne une image complète de ses capacités.

Remarque: c'est un **mot-clé** Python.

L'instruction **return** a **deux variantes différentes**

#### *Retour sans expression*

Le premier consiste dans le mot-clé lui-même, sans rien à suivre. Lorsqu'il est utilisé à l'intérieur d'une fonction, il provoque la **fin immédiate de l'exécution de la fonction et un retour instantané (d'où le nom) au point d'appel**.

Remarque: si une fonction n'est pas destinée à produire un résultat, **l'utilisation de l'instruction return n'est pas obligatoire**, elle sera exécutée implicitement à la fin de la fonction( ou procédure si on veut être précis).

Quoi qu'il en soit, vous pouvez **l'utiliser pour mettre fin aux activités d'une fonction à la demande**.

Considérons la fonction suivante:

```
def happy_new_year(wishes = True):  
    print("Three...")  
    print("Two...")  
    print("One...")  
    if not wishes:  
        return  
  
    print("Happy New Year!")
```

Lorsqu'elle est invoquée sans argument :

```
happy_new_year()
```

La fonction provoque la sortie suivante :

```
Trois...  
Deux...  
Un...  
Bonne année !!
```

Si on fournit False comme argument :

```
happy_new_year(False)
```

On modifiera par cet appel le comportement de la fonction, l'instruction `return` provoquera son arrêt juste avant les souhaits, voici la sortie:

```
Trois...  
Deux...  
Un...
```

#### *Retour avec une expression*

La deuxième variante `return` est **étendue avec une expression** :

```
def fonction():  
    return expression
```

Son utilisation a deux conséquences :

- elle provoque **l'arrêt immédiat de l'exécution de la fonction** (rien de nouveau par rapport à la première variante)
- De plus, la fonction **évaluera la valeur de l'expression et la retournera (d'où le nom une fois de plus) comme résultat de la fonction.**

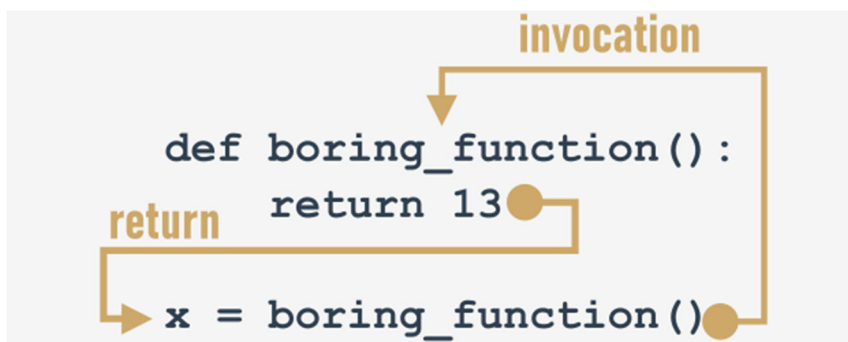
Oui, nous le savons déjà, cet exemple n'est pas vraiment sophistiqué:

```
def boring_function():  
    return 123  
  
x = boring_function()  
  
print("La boring_function a renvoyé son résultat. C'est:", x)
```

L'extrait de code écrit le texte suivant dans la console :

```
La boring_function a renvoyé son résultat. C'est: 123
```

Analysons plus en détail celle-ci.



L'instruction `return`, enrichie de l'expression (l'expression est très simple ici), « transport » de la valeur de l'expression à l'endroit où la fonction a été appelée. Le résultat peut être librement utilisé ici, par exemple pour être affecté à une variable.

Il peut également être complètement ignoré et perdu sans laisser de trace.

Notez que nous ne sommes pas trop polis ici, car la fonction renvoie une valeur, et nous l'ignorons (nous ne l'utilisons en aucune façon nulle part):

```
def boring_function():
    print("'Mode Ennui' ON.")
    return 123

print("Cette leçon est intéressante!")
boring_function()
print("Cette leçon est ennuyeuse... ")
```

Le programme produit les extraits suivants :

```
Cette leçon est intéressante!
'Mode Ennui' activé.
Cette leçon est ennuyeuse...
```

Est-ce que pour vous le fait de ne pas stocker notre résultat est punissable ? Et bien non, le seul inconvénient est que le résultat a été irrémédiablement perdu.

N'oubliez pas :

- Vous êtes toujours **autorisé à ignorer le résultat** de la fonction et à être satisfait de l'effet de la fonction (si la fonction en a un)
- Si une fonction est destinée à renvoyer un résultat utile, elle doit contenir la deuxième variante de l'instruction `return`.

Attendez une minute chers étudiants, cela signifie-t-il qu'il y a aussi des résultats inutiles?

Oui Oui Oui, dans un certain sens en tout cas. Voyons cela

### 7.3.2 Quelques mots sur `None`

Laissez-nous vous présenter une valeur très curieuse (pour être honnête, une valeur nulle) nommée `None`.

Elle ne représente aucune valeur, en fait, ce n'est pas une valeur; Par conséquent, il **ne doit prendre part à aucune expression**.

Par exemple, un extrait comme celui-ci :

```
print(None + 2)
```

provoquera une erreur d'exécution, décrite par le message de diagnostic suivant :

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Remarque : `None` n'est un **mot-clé**.

Il n'y a que deux types de circonstances dans lesquelles `None` peut être utilisé en toute sécurité :

- lorsque vous **l'affectez à une variable** (ou la renvoyez en tant que **résultat d'une fonction**)
- lorsque vous **le comparez à une variable** pour diagnostiquer son état interne.

Tout comme ici :

```
value = None
if value is None:
    print("Désolé, vous n'avez aucune valeur")
```

N'oubliez pas ceci : si une fonction ne retourne pas une certaine valeur à l'aide d'une expression `return`, on suppose qu'elle **renvoie implicitement** `None`.

Jetez un coup d'œil au code suivant :

```
def strange_function(n):
    if (n % 2 == 0):
        return True
```

Il est évident que la fonction `strange_function` renvoie `True` lorsque son argument est pair.

Qu'est-ce qu'il renvoie dans les autres cas ?

Nous pouvons utiliser le code suivant pour le vérifier :

```
print(strange_function(2))
print(strange_function(1))
```

Voici ce que nous voyons dans la console :

```
True
None
```

Ne soyez pas surpris la prochaine fois que vous verrez **None** comme résultat de fonction, cela peut être le symptôme d'une erreur subtile à l'intérieur de celle-ci.

### 7.3.4 Listes et fonctions

Il y a deux questions supplémentaires auxquelles il faut répondre.

La première est : **une liste peut-elle être envoyée à une fonction en tant qu'argument ?**

Bien sûr que oui ! Toute entité reconnaissable par Python peut jouer le rôle d'un argument de fonction, bien qu'il faille s'assurer que la fonction est capable de s'en occuper.

Donc, si vous passez une liste à une fonction, la fonction doit la gérer comme une liste.

Une fonction comme celle-ci :

```
def list_sum(lst):  
    s = 0  
  
    for elem in lst:  
        s += elem  
    return s
```

et invoqué comme ceci : `print(list_sum([5, 4, 3]))`

retournera `12`, mais vous devriez vous attendre à des problèmes si vous l'invoquez de cette manière :

```
print(list_sum(5))
```

La réponse de Python sera sans équivoque :

```
TypeError: 'int' object is not iterable
```

Cela est dû au fait **qu'une seule valeur entière ne peut pas être itérée par la boucle `for`**.

La deuxième question est la suivante : **une liste peut-elle être un résultat de fonction ?**

Oui bien sûr ! Toute entité reconnaissable par Python peut être un résultat de fonction. Regardez le code suivant :

```
def strange_list_fun(n):  
    strange_list = []  
    for i in range(0, n):  
        strange_list.insert(0, i)  
    return strange_list  
  
print(strange_list_fun(5))
```

Le résultat du programme sera comme ceci : `[4, 3, 2, 1, 0]`

Vous pouvez maintenant écrire des fonctions avec et sans résultats. Nous allons nous plonger plus profondément dans les problèmes liés aux variables dans les fonctions. Ceci est essentiel pour créer des fonctions efficaces et sûres.



## 7.3.5 Exercices

## 1 Temps estimé

10-15 minutes

## Objectifs

Familiariser l'élève avec :

- projection et écriture de fonctions paramétrées;
- l'utilisation de la déclaration de retour;
- Test des fonctions.

## Scénario

Votre tâche consiste à écrire et à tester une fonction qui prend un argument (une année) et retourne True si l'année est *bissextile*, ou False dans le cas contraire.

La base de la fonction vous êtes déjà donnée ci-dessous si vous avez des problèmes.

Remarque: nous avons également préparé un code de test court, que vous pouvez utiliser pour tester votre fonction.

Le code utilise deux listes : l'une avec les données de test et l'autre contenant les résultats attendus. Le code vous indiquera si l'un de vos résultats n'est pas valide.

## Code aide :

```
def is_year_leap(year):  
    #  
    # Votre code ici  
    #  
  
test_data = [1900, 2000, 2016, 1987]  
test_results = [False, True, True, False]  
for i in range(len(test_data)):  
    yr = test_data[i]  
    print(yr, "->", end="")  
    result = is_year_leap(yr)  
    if result == test_results[i]:  
        print("OK")  
    else:  
        print("KO")
```

## 2 Temps estimé

15-20 minutes

## Objectifs

Familiariser l'élève avec :

- projection et écriture de fonctions paramétrées;
- l'utilisation de la déclaration de retour;
- en utilisant les propres fonctions de l'élève.

## Scénario

Votre tâche consiste à écrire et à tester une fonction qui prend deux arguments (une année et un mois) et retourne le nombre de jours pour la paire mois/année donnée (Seul février est sensible à la valeur de l'année, mais votre fonction doit être universelle).

La première partie de la fonction est prête grâce à votre exercice précédent. Maintenant, convainquez la fonction de renvoyer None si ses arguments n'ont pas de sens ou sont erronés.

Bien sûr, vous pouvez (et devez) utiliser la fonction précédemment écrite et testée à l'exercice précédent. Nous vous encourageons à utiliser une liste remplie avec les longueurs des mois. Vous pouvez la créer à l'intérieur de la fonction, cette astuce raccourcira considérablement le code.

Nous avons préparé un code de test. Développez-le pour inclure d'autres cas de test.

```
def is_year_leap(year):  
    #  
    # Exercice précédent  
  
def days_in_month(year, month):  
    #  
    # Votre code ici  
  
test_years = [1900, 2000, 2016, 1987]  
test_months = [2, 2, 1, 11]  
test_results = [28, 29, 31, 30]  
for i in range(len(test_years)):  
    yr = test_years[i]  
    mo = test_months[i]  
    print(yr, mo, "->", end="")  
    result = days_in_month(yr, mo)  
    if result == test_results[i]:  
        print("OK")  
    else:  
        print("KO")
```

## 3 Temps estimé

20-30 minutes

## Objectifs

Familiariser l'élève avec :

- projection et écriture de fonctions paramétrées;
- l'utilisation de la déclaration de retour;
- la création d'un ensemble de fonctions utilitaires;
- en utilisant les propres fonctions de l'élève.

## Scénario

Votre tâche consiste à écrire et à tester une fonction qui prend trois arguments (une année, un mois et un jour du mois) et renvoie le jour correspondant de l'année, ou `None` si l'un des arguments n'est pas valide.

Utilisez les fonctions précédemment écrites et testées. Ajoutez des cas de test au code.

```
def is_year_leap(year):  
    #  
    # Code exo 1  
    #  
  
def days_in_month(year, month):  
    #  
    # Code exo 2  
    #  
  
def day_of_year(year, month, day):  
    #  
    # Votre code ici  
    #  
  
print(day_of_year(2000, 12, 31))
```

## 4 Temps estimé

15-20 minutes

## Objectifs

- familiariser l'élève avec les notions classiques et les algorithmes;
- améliorer les compétences de l'élève dans la définition et l'utilisation des fonctions.

## Scénario

Un entier naturel est **premier** s'il est supérieur à 1 et n'a pas de diviseurs autres que 1 et lui-même. Compliqué? Pas du tout, ce ne sont que des math ! Par exemple, 8 n'est pas un nombre premier, car vous pouvez le diviser par 2 et 4 (nous ne pouvons pas utiliser des diviseurs égaux à 1 et 8, car la définition l'interdit).

D'autre part, 7 est un nombre premier, car nous ne pouvons pas trouver de diviseurs.

Votre tâche consiste à écrire une fonction vérifiant si un nombre est premier.

La fonction :

- s'appelle `is_prime`;
- prend un argument (la valeur à vérifier)
- renvoie `True` si l'argument est un nombre premier, et `False` dans le cas contraire.

Astuce: essayez de diviser l'argument par toutes les valeurs suivantes (à partir de 2) et vérifiez le reste, s'il est nul, votre nombre ne peut pas être un nombre premier; Réfléchissez bien au moment où vous devriez arrêter le processus.

Si vous avez besoin de connaître la racine carrée d'une valeur, vous pouvez utiliser l'opérateur `**`. Rappelez-vous: la racine carrée de  $x$  est la même que  $x^{0,5}$ .

```
def is_prime(num):  
    #  
    # Votre code ici  
  
    for i in range(1, 20):  
        if is_prime(i + 1):  
            print(i + 1, end=" ")  
    print()
```

## Résultats escomptés

```
2 3 5 7 11 13 17 19
```

## 5 Temps estimé

10-15 minutes

## Objectifs

- améliorer les compétences de l'élève dans la définition, l'utilisation et le test des fonctions.

## Scénario

La consommation de carburant d'une voiture peut être exprimée de différentes manières. Par exemple, en Europe, on indique la quantité de carburant consommée par 100 kilomètres.

Aux États-Unis, on indique le nombre de kilomètres parcourus par une voiture utilisant un gallon de carburant.

Votre tâche consiste à écrire une paire de fonctions convertissant l/100km en mpg, et vice versa.

Les fonctions :

- sont nommés `liters_100km_to_miles_gallon` et `miles_gallon_to_liters_100km`;
- prennent un argument (la valeur correspondant à leurs noms)

Complétez le code suivant :

```
def liters_100km_to_miles_gallon(liters):  
    # Votre code ici  
  
def miles_gallon_to_liters_100km(miles):  
    # Votre code ici  
  
print(liters_100km_to_miles_gallon(3.9))  
print(liters_100km_to_miles_gallon(7.5))  
print(liters_100km_to_miles_gallon(10.))  
print(miles_gallon_to_liters_100km(60.3))  
print(miles_gallon_to_liters_100km(31.4))  
print(miles_gallon_to_liters_100km(23.5))
```

Voici quelques informations pour vous aider :

- 1 mille américain = 1609,344 mètres;
- 1 gallon américain = 3,785411784 litres.

## Résultats escomptés

```
60.31143162393162  
31.361944444444444  
23.521458333333333  
3.9007393587617467  
7.490910297239916  
10.009131205673757
```

### 7.3.6 Résumé

1. Vous pouvez utiliser le mot-clé `return` pour indiquer à une fonction de renvoyer une valeur. L'instruction `return` quitte la fonction, par exemple :

```
def multiply(a, b):  
    return a * b  
  
print(multiply(3, 4))    # outputs: 12  
  
def multiply(a, b):  
    return  
  
print(multiply(3, 4))    # outputs: None
```

2. Le résultat d'une fonction peut être facilement affecté à une variable, par exemple:

```
def wishes():  
    return "Happy Birthday!"  
  
w = wishes()  
print(w)    # outputs: Happy Birthday!
```

Regardez la différence de sortie dans les deux exemples suivants :

```
# Example 1  
def wishes():  
    print("My Wishes")  
    return "Happy Birthday"  
  
wishes()    # outputs: My Wishes  
  
# Example 2  
def wishes():  
    print("My Wishes")  
    return "Happy Birthday"  
  
print(wishes())  
  
# outputs: My Wishes  
#           Happy Birthday
```

3. Vous pouvez utiliser une liste comme argument d'une fonction, par exemple :

```
def hi_everybody(my_list):  
    for name in my_list:  
        print("Hi,", name)  
  
hi_everybody(["Anass", "Nathan", "Luca"])
```

4. Une liste peut également être un résultat de fonction, par exemple:

```
def create_list(n):  
    my_list = []  
    for i in range(n):  
        my_list.append(i)  
    return my_list  
print(create_list(5))
```

### Exercice 1

Quelle est la sortie de l'extrait de code suivant ?

```
def hi():  
    return  
    print("Hi!")  
hi()
```

### Exercice 2

Quelle est la sortie de l'extrait de code suivant ?

```
def is_int(data):  
    if type(data) == int:  
        return True  
    elif type(data) == float:  
        return False  
print(is_int(5))  
print(is_int(5.0))  
print(is_int("5"))
```

### Exercice 3

Quelle est la sortie de l'extrait de code suivant ?

```
def even_num_lst(ran):  
    lst = []  
    for num in range(ran):  
        if num % 2 == 0:  
            lst.append(num)  
    return lst  
print(even_num_lst(11))
```

### Exercice 4

Quelle est la sortie de l'extrait de code suivant ?

```
def list_updater(lst):  
    upd_list = []  
    for elem in lst:  
        elem **= 2  
        upd_list.append(elem)  
    return upd_list  
foo = [1, 2, 3, 4, 5]  
print(list_updater(foo))
```

## 7.4 Fonctions et portées

Commençons par une définition :

La **portée** d'un nom (par exemple, un nom de variable) est la partie d'un code où le nom est correctement reconnaissable.

Par exemple, la portée du paramètre d'une fonction est la fonction elle-même. Le paramètre est inaccessible en dehors de celle-ci.

Vérifions cela, à votre avis que va-t-il se produire ?

```
def scope_test():  
    x = 123  
scope_test()  
print(x)
```

Le programme échouera lors de son exécution. Le message d'erreur se lit comme suit :

```
NameError: name 'x' is not defined
```

Nous allons mener quelques expériences avec vous pour vous montrer comment Python construit des scopes (portées/étendues), et comment vous pouvez utiliser cela à votre avantage.

Commençons par vérifier si une variable créée en dehors d'une fonction est visible à l'intérieur des fonctions. En d'autres termes, le nom d'une variable se propage-t-il dans le corps d'une fonction ?

```
def my_function():  
    print("Est-ce que je connais la variable?", var)  
  
var = 1  
my_function()  
print(var)
```

Le résultat du test peut être considéré comme positif, le code sort:

```
Est-ce que je connais la variable? 1  
1
```

La réponse est : une **variable existant à l'extérieur d'une fonction a une portée à l'intérieur des corps de fonctions**.

Cette règle comporte une exception très importante. **Essayons de le trouver.**



Apportons une petite modification au code :

```
def my_function():  
    var = 2  
    print("Est-ce que je connais la variable?", var)  
var = 1  
my_function()  
print(var)
```

Le résultat a changé, le code produit maintenant une sortie légèrement différente:

```
Est-ce que je connais cette variable? 2  
1
```

Que s'est-il passé?

- La variable `var` créée à l'intérieur de la fonction n'est pas la même que lorsqu'elle est définie à l'extérieur de celle-ci, il semble qu'il existe deux variables différentes du même nom;
- De plus, la variable de la fonction fait de l'ombre à la variable venant de l'extérieur.

Nous pouvons rendre la règle précédente plus précise et juste:

**Une variable existant en dehors d'une fonction a une portée à l'intérieur du corps des fonctions, à l'exclusion de ceux d'entre eux qui définissent une variable du même nom.**

Cela signifie également que la **portée d'une variable existant en dehors d'une fonction n'est prise en charge que lors de l'obtention de sa valeur** (lecture). L'attribution d'une valeur force la création de la propre variable de la fonction.

#### 7.4.1 Le mot-clé `global`

Vous devriez maintenant arriver à la question suivante: cela signifie-t-il qu'une fonction n'est pas capable de modifier une variable définie en dehors d'elle-même? Cela créerait beaucoup pas mal de soucis. Heureusement, la réponse est *non*.

Il existe une méthode Python spéciale qui peut **étendre la portée d'une variable de manière à inclure les corps de fonctions** (que vous vouliez lire les valeurs ou les modifier).

Un tel effet est provoqué par le mot-clé `global`:

```
global name  
global name1, name2, ...
```

L'utilisation de ce mot-clé à l'intérieur d'une fonction avec le nom (ou les noms séparés par des virgules) d'une variable(s), force Python à s'abstenir de créer une nouvelle variable à l'intérieur de la fonction, celle accessible de l'extérieur sera utilisée à la place.

En d'autres termes, ce nom devient global (il a une **portée globale**, et peu importe qu'il soit le sujet de lecture ou d'affectation).

Regardez le code suivant :

```
def my_function():
    global var
    var = 2
    print("Est-ce que je connais la variable?", var)

var = 1
my_function()
print(var)
```

Nous avons ajouté **global** à la fonction.

Le code génère maintenant :

```
Est-ce que je connais cette variable? 2
2
```

Cela devrait être une preuve suffisante pour montrer que le mot-clé `global` fait ce qu'il promet.

#### 7.4.2 L'interaction avec les arguments

Voyons maintenant comment la fonction interagit avec ses arguments.

Le code suivant devrait vous apprendre quelque chose. Comme vous pouvez le voir, la fonction modifie la valeur de son paramètre. Le changement affecte-t-il l'argument?

```
def my_function(n):
    print("I got", n)
    n += 1
    print("I have", n)

var = 1
my_function(var)
print(var)
```

La sortie du code est la suivante :

```
J'en ai obtenu 1
J'en ai 2
1
```

La conclusion est évidente : **changer la valeur du paramètre ne se propage pas en dehors de la fonction** (en tout cas, pas lorsque la variable est un scalaire, comme dans l'exemple).

Cela signifie également qu'une fonction reçoit **la valeur de** l'argument, et non l'argument lui-même. Cela est vrai pour les scalaires.

Cela vaut-il la peine de vérifier comment cela fonctionne avec les listes (vous souvenez-vous des particularités de l'attribution de tranches de liste par rapport à l'affectation de listes dans leur ensemble?).

L'exemple suivant éclairera votre mémoire :

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    my_list_1 = [0, 1]
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)

my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

La sortie du code est la suivante :

```
Impression #1: [2, 3]
Impression #2: [2, 3]
Impression #3: [0, 1]
Impression #4: [2, 3]
Impression #5: [2, 3]
```

Il semble que l'ancienne règle fonctionne toujours.

Enfin, pouvez-vous voir la différence dans l'exemple ci-dessous:

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    del my_list_1[0] # Regarder cette ligne avec attention !!!
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)

my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

Nous ne changeons pas la valeur du paramètre `my_list_1` (nous savons déjà que cela n'affectera pas l'argument), mais modifions plutôt la liste identifiée par celui-ci.

Le résultat peut être surprenant. Exécutez le code et vérifiez :

```
Impression #1: [2, 3]
Impression #2: [2, 3]
Impression #3: [3]
Impression #4: [3]
Tirage #5: [3]
```

Pouvez-vous l'expliquer?

Essayons :

- si l'argument est une liste, la modification de la valeur du paramètre correspondant n'affecte pas la liste (rappelez-vous: les variables contenant des listes sont stockées d'une manière différente des scalaires),
- Mais si vous modifiez une liste identifiée par le paramètre (note : la liste, pas le paramètre !), la liste reflétera le changement.

#### 7.4.3 Résumé

1. Une variable qui existe en dehors d'une fonction a une portée à l'intérieur du corps de la fonction (exemple 1), sauf si la fonction définit une variable du même nom (exemple 2 et exemple 3), par exemple :

Exemple 1 :

```
var = 2

def mult_by_var(x):
    return x * var

print(mult_by_var(7))    # outputs: 14
```

Exemple 2 :

```
def mult(x):
    var = 5
    return x * var

print(mult(7))    # outputs: 35
```

Exemple 3 :

```
def mult(x):
    var = 7
    return x * var

var = 3
print(mult(7))    # outputs: 49
```

2. Une variable qui existe à l'intérieur d'une fonction a une portée à l'intérieur du corps de la fonction, par exemple :

```
def adding(x):
    var = 7
    return x + var

print(adding(4))    # outputs: 11
print(var)          # NameError
```

3. Vous pouvez utiliser le mot-clé global suivi d'un nom de variable pour rendre la portée de la variable globale, par exemple :

```
var = 2
print(var)      # outputs: 2

def return_var():
    global var
    var = 5
    return var

print(return_var()) # outputs: 5
print(var)         # outputs: 5
```

### Exercice 1

Que se passe-t-il lorsque vous essayez d'exécuter le code suivant ?

```
def message():
    alt = 1
    print("Hello, World!")

print(alt)
```

### Exercice 2

Quelle est la sortie de l'extrait de code suivant ?

```
a = 1

def fun():
    a = 2
    print(a)

fun()
print(a)
```

### Exercice 3

Quelle est la sortie de l'extrait de code suivant ?

```
a = 1

def fun():
    global a
    a = 2
    print(a)

fun()
a = 3
print(a)
```

## 7.5 Création de fonctions simples

### 7.5.1 Création d'une fonction à deux paramètres

Commençons par une fonction pour évaluer l'indice de masse corporelle (IMC).

$$\text{BMI} = \frac{(\text{weight in kilograms})}{\text{height in meters}^2}$$


Comme vous pouvez le voir, la formule a besoin de deux valeurs :

- Poids (à l'origine en kilogrammes)
- hauteur (à l'origine en mètres)

Il semble que cette nouvelle fonction aura donc **deux paramètres**. Son nom sera **bmi**, mais si vous préférez un autre nom, utilisez-le.

Codons la fonction:

```
def bmi(weight, height):  
    return weight / height ** 2  
  
print(bmi(52.5, 1.65))
```

Le résultat produit par l'exemple se présente comme suit :

19.283746556473833

La fonction répond à nos attentes, mais c'est un peu simple, elle suppose que les valeurs des deux paramètres sont toujours significatives. Cela vaut pourtant la peine de vérifier s'ils sont dignes de confiance. Et cela vous devriez le retenir pour votre avenir, ne pas faire confiance à un utilisateur !

Vérifions-les tous les deux et retournons **None** si l'un des deux semble suspect.

```
def bmi(weight, height):  
    if height < 1.0 or height > 2.5 or \  
    weight < 20 or weight > 200:  
        return None  
  
    return weight / height ** 2  
  
print(bmi(352.5, 1.65))
```

Tout d'abord, l'appel de test garantit que la **sécurité** fonctionne correctement et la sortie est donc la suivante :

**None**

Deuxièmement, jetez un coup d'œil à la façon dont le symbole **de barre oblique inverse** (\) est utilisé. Si vous l'utilisez dans du code Python et que vous terminez une ligne avec, il indiquera à Python de continuer la ligne de code dans la ligne de code suivante.

Cela peut être particulièrement utile lorsque vous devez gérer de longues lignes de code et que vous souhaitez améliorer la lisibilité du code.

D'accord, mais il y a quelque chose que nous avons omis trop facilement, les mesures impériales. Cette fonction n'est pas trop utile pour les personnes habituées aux livres, aux pieds et aux pouces. Que peut-on faire pour eux?

Nous pouvons écrire deux fonctions simples pour **convertir les unités impériales en unités métriques**. Commençons par les livres.

C'est un fait bien connu que  $1 \text{ lb} = 0.45359237 \text{ kg}$ . Nous l'utiliserons dans notre nouvelle fonction.

C'est notre fonction, sera nommée `lb_to_kg`:

```
def lb_to_kg(lb):  
    return lb * 0.45359237  
  
print(lb_to_kg(1))
```

Le résultat de l'appel de test semble bon: 0.45359237

Et maintenant, il est temps de faire de même pour les pieds et les pouces:  $1 \text{ ft} = 0.3048 \text{ m}$ , et  $1 \text{ in} = 2.54 \text{ cm} = 0.0254 \text{ m}$ .

La fonction que nous avons écrite s'appelle `ft_and_inch_to_m` :

```
def ft_and_inch_to_m(ft, inch):  
    return ft * 0.3048 + inch * 0.0254  
  
print(ft_and_inch_to_m(1, 1))
```

Le résultat d'un test rapide est : 0.3302

Remarque: nous voulions nommer le deuxième paramètre juste `in` , pas `inch`, mais nous ne pouvons pas. Savez-vous pourquoi?

`in` est un **mot-clé** Python, il ne peut pas être utilisé comme nom de variable.

Convertissons *six pieds* en mètres:

```
print(ft_and_inch_to_m(6, 0))
```

Et voici le résultat: 1.8288000000000002

Il est tout à fait possible que parfois vous vouliez utiliser seulement des pieds sans pouces. Python vous aidera-t-il ? Bien sûr que oui, vous vous souvenez du chapitre précédent ?

Nous avons un peu modifié le code :

```
def ft_and_inch_to_m(ft, inch = 0.0) :  
    return ft * 0.3048 + inch * 0.0254  
  
print(ft_and_inch_to_m(6))
```

Maintenant, le paramètre inch a sa valeur par défaut égale à 0,0.

Le code produit la sortie suivante : 1.8288000000000002

Enfin, le code est en mesure de répondre à la question: quel est l'IMC d'une personne mesurant 5'7" et pesant 176 lb?

Voici le code que nous avons construit ensemble en résumé :

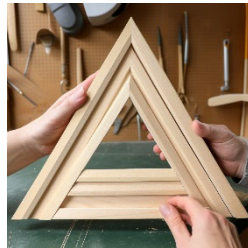
```
def ft_and_inch_to_m(ft, inch = 0.0):  
    return ft * 0.3048 + inch * 0.0254  
  
def lb_to_kg(lb):  
    return lb * 0.45359237  
  
def bmi(weight, height):  
    if height < 1.0 or height > 2.5 or weight < 20 or weight > 200:  
        return None  
  
    return weight / height ** 2  
  
print(bmi(weight = lb_to_kg(176), height = ft_and_inch_to_m(5, 7)))
```

Et la réponse est : 27.565214082533313



## 7.5.2 Jouons avec un triangle

Jouons avec les triangles maintenant. Nous allons commencer par une fonction pour vérifier si trois côtés de longueurs données peuvent construire un triangle.



Nous savons par notre cher prof de Math que vous aimez tant que la somme de deux côtés arbitraires doit être plus longue que le troisième côté.

Ce ne sera pas un défi difficile. La fonction aura **trois paramètres**, un pour chaque côté.

Elle retournera True si les côtés peuvent construire un triangle, et False dans le cas contraire. Dans ce cas, `is_a_triangle` est un bon nom pour une telle fonction.

```
def is_a_triangle(a, b, c):  
    if a + b <= c:  
        return False  
    if b + c <= a:  
        return False  
    if c + a <= b:  
        return False  
    return True  
  
print(is_a_triangle(1, 1, 1))  
print(is_a_triangle(1, 1, 3))
```

Il semble que cela fonctionne bien, voici les résultats:

```
True  
False
```

Pouvons-nous le rendre plus compact? Cela a l'air un peu trop verbeux.

```
def is_a_triangle(a, b, c):  
    if a + b <= c or b + c <= a or c + a <= b:  
        return False  
    return True  
  
print(is_a_triangle(1, 1, 1))  
print(is_a_triangle(1, 1, 3))
```

Pouvons-nous le compacter encore plus?

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

print(is_a_triangle(1, 1, 1))
print(is_a_triangle(1, 1, 3))
```

Nous avons nié la condition (inversé les opérateurs relationnels et remplacés or par and, ce qui donne une **expression universelle pour tester les triangles**).

Vous pouvez maintenant créer une fonction dans un programme plus grand qui demandera à l'utilisateur trois valeurs et utilisera celle-ci.

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

a = float(input('Entrer le taille du premier côté : '))
b = float(input('Entrer le taille du second côté : '))
c = float(input('Entrer la taille du troisième côté : '))

if is_a_triangle(a, b, c):
    print('Oui, je suis un triangle.')
else:
    print('Non, je ne peux pas être un triangle.')
```

Dans un deuxième temps, nous allons essayer de nous assurer qu'un certain triangle est un **triangle rectangle**.

Nous devons utiliser le **théorème de Pythagore**:  $c^2 = a^2 + b^2$

Comment reconnaître lequel des trois côtés est l'hypoténuse?

**Simple, l'hypoténuse est le côté le plus long.**

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

def is_a_right_triangle(a, b, c):
    if not is_a_triangle(a, b, c):
        return False
    if c > a and c > b:
        return c ** 2 == a ** 2 + b ** 2
    if a > b and a > c:
        return a ** 2 == b ** 2 + c ** 2

print(is_a_right_triangle(5, 3, 4))
print(is_a_right_triangle(1, 3, 4))
```

Regardez comment nous testons la relation entre l'hypoténuse et les côtés restants, nous choisissons le côté le plus long et **appliquons le théorème de Pythagore** pour vérifier si tout est correct. Cela nécessite trois contrôles au total.

*Et maintenant un petit calcul d'air*

Nous pouvons également évaluer l'aire d'un triangle. **La formule de Heron** sera pratique ici:

$$s = \frac{a+b+c}{2}$$

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

Nous allons utiliser l'opérateur d'exponentiation pour trouver la racine carrée - cela peut sembler étrange, mais cela fonctionne:

$$\sqrt{X} = X^{\frac{1}{2}}$$

Voici le code résultant :

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

def heron(a, b, c):
    p = (a + b + c) / 2
    return (p * (p - a) * (p - b) * (p - c)) ** 0.5

def area_of_triangle(a, b, c):
    if not is_a_triangle(a, b, c):
        return None
    return heron(a, b, c)

print(area_of_triangle(1., 1., 2. ** .5))
```

Nous l'essayons ici avec un triangle à angle droit qui est la moitié d'un carré avec un côté égal à 1. Cela signifie que sa superficie doit être égale à 0,5.

C'est étrange - le code produit la sortie suivante: 0.49999999999999983

C'est très proche de 0,5, mais ce n'est pas exactement 0,5. Qu'est-ce que cela signifie? Est-ce une erreur?

Non, ce n'est pas le cas. Ce sont **les spécificités des calculs en virgule flottante**. Nous vous en dirons plus bientôt.

### 7.5.3 Les factorielles

Une autre fonction assez simple et connue est la **factorielle**. Vous souvenez-vous comment une factorielle est définie ?

$0! = 1$  (oui! c'est vrai) ;  $1! = 1$  ;  $2! = 1 * 2$  ;  $3! = 1 * 2 * 3$  ;  $4! = 1 * 2 * 3 * 4$  ; ... ;  $n! = 1 * 2 * 3 * 4 * ... * n-1 * n$

Elle est représentée par un **point d'exclamation** et est égale au **produit** de tous les entiers naturels d'un à son argument.

Écrivons notre code. Nous allons créer une fonction et l'appeler `factorial_function`. Voici le code :

```
def factorial_function(n):  
    if n < 0:  
        return None  
    if n < 2:  
        return 1  
  
    product = 1  
    for i in range(2, n + 1):  
        product *= i  
    return product  
  
for n in range(1, 6): # test  
    print(n, factorial_function(n))
```

Remarquez comment nous respectons étape par étape la définition mathématique et comment nous utilisons la boucle `for` pour **trouver le produit**.

Nous ajoutons un code de test simple, et voici les résultats que nous obtenons:

```
1 1  
2 2  
3 6  
4 24  
5 120
```

## 7.5.4 La suite de Fibonacci

Connaissez-vous les nombres de **Fibonacci** ?

Il s'agit d'une **suite de nombres** entiers construite à l'aide d'une règle très simple:

- le premier élément de la séquence est égal à un (**Fib<sub>1</sub> = 1**)
- le second est également égal à un (**Fib<sub>2</sub> = 1**)
- chaque nombre suivant est la somme des deux nombres précédents: (Fib i = Fib i-1 + Fib i-2)

Voici quelques-uns des premiers nombres de Fibonacci:

```
fib_1 = 1
fib_2 = 1
fib_3 = 1 + 1 = 2
fib_4 = 1 + 2 = 3
fib_5 = 2 + 3 = 5
fib_6 = 3 + 5 = 8
fib_7 = 5 + 8 = 13
```

Que pensez-vous de **la mise en œuvre de cette fonction** ?

Créons notre fonction fib et testons-la. Le voilà:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1

    elem_1 = elem_2 = 1
    the_sum = 0
    for i in range(3, n + 1):
        the_sum = elem_1 + elem_2
        elem_1, elem_2 = elem_2, the_sum
    return the_sum

for n in range(1, 10): # test
    print(n, "->", fib(n))
```

Analysez attentivement le corps de la boucle **for** et découvrez comment nous **déplaçons** les variables **elem\_1** **et** **elem\_2** **à travers les nombres de Fibonacci suivants**.

La partie test du code produit la sortie suivante :

```
1 -> 1
2 -> 1
3 -> 2
4 -> 3
5 -> 5
6 -> 8
7 -> 13
8 -> 21
9 -> 34
```

### 7.5.5 La récursion

Il y a encore une chose que nous voulons vous montrer c'est la **récurtivité**.

Ce terme peut décrire de nombreux concepts différents, mais l'un d'entre eux est particulièrement intéressant, c'est celui qui se réfère à la programmation informatique. Dans ce domaine, la récurtivité est une **technique où une fonction qui s'invoque elle-même**.

Ces deux cas semblent être les meilleurs pour illustrer le phénomène - factorielles et nombres de Fibonacci. Surtout ce dernier.

**La définition des nombres de Fibonacci est un exemple clair de récurtivité.**

Nous vous avons déjà dit que :

**Fib i = Fib i-1 + Fib i-2**

La définition du  $i^{\text{ème}}$  nombre se réfère au nombre  $i-1$ , et ainsi de suite, jusqu'à ce que vous atteigniez les deux premiers.

Peut-il être utilisé ce principe dans le code? Oui, c'est possible. Cela peut également rendre le code plus court et plus clair.

La deuxième version de notre fonction fib() utilise directement cette définition :

```
def fib(n):  
    if n < 1:  
        return None  
    if n < 3:  
        return 1  
    return fib(n - 1) + fib(n - 2)
```

Le code est beaucoup plus clair non ?

Mais est-ce vraiment sûr? Y a-t-il un risque?

Oui, il y a un petit risque en effet. **Si vous oubliez de considérer les conditions qui peuvent arrêter la chaîne d'invocations récursives, le programme peut entrer dans une boucle infinie.** Il faut faire attention.

La factorielle a aussi un second côté **récurtif**.

Regarder:  $n! = 1 \times 2 \times 3 \times \dots \times n-1 \times n$

Il est évident que :  $1 \times 2 \times 3 \times \dots \times n-1 = (n-1)!$

Donc, finalement, le résultat est:  $n! = (n-1)! \times n$

```
def factorial_function(n):  
    if n < 0:  
        return None  
    if n < 2:  
        return 1  
    return n * factorial_function(n - 1)
```

## 7.5.6 Résumé

1. Une fonction peut appeler d'autres fonctions ou même elle-même. Lorsqu'une fonction s'appelle elle-même, cette situation est connue sous le nom de **récurtivité**, et la fonction qui s'appelle elle-même et contient une condition de terminaison spécifiée (c'est-à-dire le cas de base, une condition qui ne dit pas à la fonction de faire d'autres appels à cette fonction mais de stopper) est appelée une fonction **récursive**.

2. Vous pouvez utiliser des fonctions récursives en Python pour écrire du **code propre et élégant et le diviser en morceaux plus petits et organisés**. D'un autre côté, vous devez être très prudent car il pourrait être **facile de faire une erreur et de créer une fonction qui ne se termine jamais**. Vous devez également vous rappeler que les **appels récursifs consomment beaucoup de mémoire** et peuvent donc parfois être inefficaces.

Lorsque vous utilisez la récurtivité, vous devez prendre en compte tous ses avantages et inconvénients.

La fonction factorielle est un exemple classique de la façon dont le concept de récurtivité peut être mis en pratique :

*# Implémentation récursive de la fonction factorielle.*

```
def factorial(n):  
    if n == 1:      # Cas de base  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
print(factorial(4)) # 4 * 3 * 2 * 1 = 24
```

**Exercice 1**

Que se passe-t-il lorsque vous tentez d'exécuter l'extrait de code suivant et pourquoi ?

```
def factorial(n):  
    return n * factorial(n - 1)
```

```
print(factorial(4))
```

**Exercice 2**

Quelle est la sortie de l'extrait de code suivant ?

```
def fun(a):  
    if a > 30:  
        return 3  
    else:  
        return a + fun(a + 3)  
  
print(fun(25))
```

## 8 Tuples et dictionnaire

### 8.1 Types de séquences et mutabilité

Avant de commencer à parler de **tuples** et de **dictionnaires**, nous devons introduire deux concepts importants: **les types de séquence** et **la mutabilité**.

Un type de séquence est **un type de données en Python qui est capable de stocker plus d'une valeur (ou moins d'une, car une séquence peut être vide), et ces valeurs peuvent être parcourues séquentiellement (d'où le nom), élément par élément.**

Comme la boucle `for` est un outil spécialement conçu pour itérer à travers les séquences, nous pouvons exprimer la définition comme suit: **une séquence est une donnée qui peut être balayée par la boucle `for`.**

Vous avez rencontré une séquence Python jusqu'à présent : la liste. La liste est un exemple classique d'une séquence Python, bien qu'il y ait d'autres séquences qui méritent d'être mentionnées, et nous allons vous les présenter maintenant.

La deuxième notion à maîtriser est la **mutabilité**, c'est une propriété de toutes les données de Python qui décrit sa volonté d'être librement modifiée pendant l'exécution du programme. Il existe deux types de données Python : **mutables** et **immuables**.

**Les données mutables peuvent être librement mises à jour à tout moment**, nous appelons une telle opération **in situ**.

*In situ* est une phrase latine qui se traduit littéralement par *en position ou dans le lieu précis*. Par exemple, l'instruction suivante modifie les données **in situ** :

*liste. Ajouter(1)*

**Les données immuables ne peuvent pas être modifiées de cette manière.**

Imaginez qu'une liste ne peut être attribuée et relue. Vous ne pourrez ni y ajouter un élément, ni en supprimer. Cela signifie que l'ajout d'un élément à la fin de la liste nécessiterait la recreation de la liste à partir de zéro.

Vous devrez créer une liste complètement nouvelle, composée de tous les éléments de la liste déjà existante, plus le nouvel élément.

Le type de données dont nous voulons vous parler maintenant est un **tuple**. **Un tuple est un type de séquence immuable**. Il peut se comporter comme une liste, mais il ne doit/peut pas être modifié *in situ*.



## 8.2 Qu'est-ce qu'un tuple ?

La première et la plus claire distinction entre les listes et les tuples est la syntaxe utilisée pour les créer : les **tuples préfèrent utiliser des parenthèses**, tandis que les **listes aiment voir des crochets**, bien qu'il soit également **possible de créer un tuple juste à partir d'un ensemble de valeurs séparées par des virgules**.

Regardez l'exemple :

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125
```

Il y a deux tuples, tous deux contenant **quatre éléments**.

Imprimons-les :

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125
print(tuple_1)
print(tuple_2)
```

Voici ce que vous devriez voir dans la console:

```
(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125)
```

**Remarque :** **chaque élément de tuple peut être d'un type différent** (virgule flottante, entier ou tout autre type de données non encore introduit).

### 8.2.1 Comment créer un tuple ?

Il est possible de créer un tuple vide, des parenthèses sont alors nécessaires :

```
empty_tuple = ()
```

Si vous souhaitez créer un **tuple** à un élément, vous devez prendre en considération le fait que, pour des raisons de syntaxe (un tuple doit être distingué d'une valeur unique ordinaire), vous devez terminer la valeur par une virgule :

```
one_element_tuple_1 = (1, )
one_element_tuple_2 = 1.,
```

La suppression des virgules ne fera pas planter le programme dans un sens syntaxique, mais vous obtiendrez plutôt deux variables simples, pas des tuples.

### 8.2.2 Comment utiliser un tuple?

Si vous souhaitez obtenir les éléments d'un tuple afin de les lire, vous pouvez utiliser les mêmes conventions que lors de l'utilisation de listes.

```
my_tuple = (1, 10, 100, 1000)
```

```
print(my_tuple[0])
print(my_tuple[-1])
print(my_tuple[1:])
print(my_tuple[:-2])

for elem in my_tuple:
    print(elem)
```

Le programme devrait produire la sortie suivante - exécutez-la et vérifiez:

```
1
1000
(10, 100, 1000)
(1, 10)
1
10
100
1000
```

Les similitudes peuvent être trompeuses : **n'essayez pas de modifier le contenu d'un tuple!** Ce n'est pas une liste!!!

Toutes ces instructions (à l'exception de la première) provoqueront une erreur d'exécution :

```
my_tuple = (1, 10, 100, 1000)

my_tuple.append(10000)
del my_tuple[0]
my_tuple[1] = -10
```

Voici le message que Python vous donnera dans la fenêtre de la console :

```
AttributeError: 'tuple' object has no attribute 'append'
```

Mais la question qu'il faut se poser c'est qu'est-ce que les tuples peuvent faire d'autre pour vous?

- La fonction `Len()` accepte les tuples et renvoie le nombre d'éléments contenus à l'intérieur ;
- L'opérateur `+` peut joindre des tuples ensemble (nous vous l'avons déjà montré)
- L'opérateur `*` peut multiplier les tuples, tout comme les listes ;
- Les opérateurs `in` et `not in` fonctionnent de la même manière que dans les listes.

```
my_tuple = (1, 10, 100)

t1 = my_tuple + (1000, 10000)
t2 = my_tuple * 3

print(len(t2))
print(t1)
print(t2)
print(10 in my_tuple)
print(-10 not in my_tuple)
```

Le résultat doit se présenter comme suit :

```
9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
```

L'une des propriétés de tuple les plus utiles est leur capacité à **apparaître sur le côté gauche de l'opérateur d'affectation**. Vous avez vu ce phénomène il y a quelque temps, lorsqu'il a fallu trouver un outil élégant pour échanger les valeurs de deux variables.

Jetez un œil à l'extrait ci-dessous:

```
var = 123

t1 = (1, )
t2 = (2, )
t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)
```

Il montre trois tuples en interaction, en effet, les valeurs qui y sont stockées « circulent » - t1 devient t2, t2 devient t3 et t3 devient t1.

Note: l'exemple présente un autre fait important: les **éléments d'un tuple peuvent être des variables**, pas seulement des littéraux. De plus, ils peuvent être des expressions s'ils se trouvent du côté droit de l'opérateur d'affectation.

### 8.3 Qu'est-ce qu'un dictionnaire?

Le **dictionnaire** est une autre structure de données Python. Ce n'est **pas** un type de séquence (mais peut être facilement adapté au traitement de séquence) et il est **mutable**.

Pour expliquer ce qu'est réellement le dictionnaire Python, il est important de comprendre qu'il s'agit littéralement d'un dictionnaire.

Le dictionnaire Python fonctionne de la même manière **qu'un dictionnaire bilingue**. Par exemple, vous avez un mot anglais (par exemple, cat) et avez besoin de son équivalent français. Vous parcourez le dictionnaire afin de trouver le mot (vous pouvez utiliser différentes techniques pour le faire, cela n'a pas d'importance ou en tout cas pas dans ce cours mais chez Mr Depreter au Q2 cela sera plus important ! ) et finalement vous l'obtenez. Ensuite, vous vérifiez l'homologue français et c'est (très probablement, un mauvais dico ?) le mot « chat ».



Dans le monde de Python, le mot que vous recherchez est nommé **clé**. Le mot que vous obtenez du dictionnaire est appelé une **valeur**.

Cela signifie qu'un dictionnaire est un ensemble de paires **clé-valeur** :

- Chaque clé doit être **unique** : il n'est pas possible d'avoir plus d'une clé de la même valeur;
- Une clé peut être **n'importe quel type d'objet immuable** : il peut s'agir d'un nombre (entier ou flottant), ou même d'une chaîne, mais pas d'une liste ;
- un dictionnaire n'est pas une liste : une liste contient un ensemble de valeurs numérotées, tandis **qu'un dictionnaire contient des paires de valeurs**;
- La fonction `Len()` fonctionne également pour les dictionnaires : elle renvoie les nombres d'éléments clé-valeur dans le dictionnaire;
- un dictionnaire est un **outil à sens unique** : si vous avez un dictionnaire anglais-français, vous pouvez rechercher des équivalents français de termes anglais, mais pas l'inverse.

### 8.3.1 Comment faire un dictionnaire?

Si vous souhaitez affecter des paires initiales à un dictionnaire, utilisez la syntaxe suivante :

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
phone_numbers = {'Deckers': 5551234567, 'Naizy': 22657854310}
empty_dictionary = {}

print(dictionary)
print(phone_numbers)
print(empty_dictionary)
```

Dans le premier exemple, le dictionnaire utilise des clés et des valeurs qui sont toutes deux des chaînes.

Dans le second, les clés sont des chaînes, mais les valeurs sont des entiers. La disposition inverse (clés → nombres, valeurs → chaînes) est également possible, ainsi que la combinaison nombre-nombre.

La liste des paires est **entourée d'accolades**, tandis que les paires elles-mêmes sont **séparées par des virgules** et les clés et les **valeurs par deux-points**.

Le premier de nos dictionnaires est un dictionnaire anglais-français très simple. Le second - un très petit annuaire téléphonique.

Les dictionnaires vides sont construits par une **paire vide d'accolades**, rien d'inhabituel.

Le dictionnaire dans son ensemble peut être imprimé avec un seul appel `print()`.

L'extrait peut produire la sortie suivante :

```
{'dog': 'chien', 'horse': 'cheval', 'cat': 'chat'}
{'Naizy': 5557654321, 'Deckers': 5551234567}
{}
```

Avez-vous remarqué quelque chose de surprenant? L'ordre des paires imprimées est différent de celui de l'affectation initiale. Qu'est-ce que cela signifie?

Tout d'abord, c'est une confirmation que **les dictionnaires ne sont pas des listes**, ils ne conservent pas l'ordre de leurs données, car l'ordre est complètement dénué de sens (contrairement aux vrais dictionnaires papier).

L'ordre dans lequel un dictionnaire **stocke ses données est complètement hors de votre contrôle et de vos attentes**. C'est normal.

Pour votre culture, en Python, les dictionnaires 3.6x sont devenus **des collections ordonnées** par défaut. Vos résultats peuvent varier en fonction de la version de Python que vous utilisez.

### 8.3.2 Comment utiliser un dictionnaire?

Si vous souhaitez obtenir l'une des valeurs, vous devez fournir une clé valide :

```
print(dictionary['cat'])  
print(phone_numbers['Naizy'])
```

Obtenir la valeur d'un dictionnaire ressemble à l'indexation, notamment grâce aux crochets entourant la valeur de la clé.

Note:

- si la clé est une chaîne, vous devez la spécifier en tant que chaîne ;
- **Les clés sont sensibles à la casse** : « Naizy » est différent de « naizy ».

L'extrait génère deux lignes de texte :

```
Chat  
22657854310
```

Et maintenant, la nouvelle la plus importante: vous **ne pouvez pas utiliser une clé inexistante**. Essayez quelque chose comme ceci:

```
print(phone_numbers['president'])
```

provoquera une erreur d'exécution.

Heureusement, il existe un moyen simple d'éviter une telle situation. L'opérateur `in`, avec son compagnon, `not in`, peuvent sauver votre code de cette situation.

Le code suivant recherche en toute sécurité certains mots français :

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
words = ['cat', 'lion', 'horse']  
  
for word in words:  
    if word in dictionary:  
        print(word, "->", dictionary[word])  
    else:  
        print(word, "n'est pas dans le dico")
```

La sortie du code se présente comme suit :

```
cat -> chat  
lion n'est pas dans le dico  
horse -> cheval
```

Lorsque vous écrivez une expression « volumineuse ou longue », il peut être judicieux de la garder alignée verticalement dans votre code. Voici comment vous pouvez rendre votre code plus lisible et plus convivial pour les programmeurs, par exemple :

```
# Example 1:
dictionary = {
    "cat": "chat",
    "dog": "chien",
    "horse": "cheval"
}

# Example 2:
phone_numbers = {'boss': 5551234567,
                  'Suzy': 22657854310
}
```

De telles façons de formater le code sont appelées **retraits suspendus**. Encore un peu de culture 😊

### 8.3.4 Comment utiliser un dictionnaire: les clés()

Les dictionnaires peuvent-ils être **parcours** à l'aide de la boucle for, comme les listes ou les tuples ?

Non et oui.

Non, car un dictionnaire **n'est pas un type de séquence** : la boucle for est inutile avec elle.

Oui, car il existe des outils simples et très efficaces qui peuvent **adapter n'importe quel dictionnaire aux exigences de la boucle for** (en d'autres termes, construire un lien intermédiaire entre le dictionnaire et une entité de séquence temporaire).

La première d'entre elles est une méthode nommée **keys()**, qui existe dans chaque dictionnaire. Elle **renvoie un objet itérable constitué de toutes les clés rassemblées dans le dictionnaire**. Avoir un groupe de clés vous permet d'accéder à l'ensemble du dictionnaire d'une manière facile et pratique.

Exemple :

```
dictionary = {"cat": "chat", "dog": "chien", "horse":
"cheval"}

for key in dictionary.keys():
    print(key, "->", dictionary[key])
```

La sortie du code se présente comme suit :

```
horse -> cheval
dog -> chien
cat -> chat
```

### 8.3.5 La fonction sorted()

Vous voulez que votre dictionnaire **soit trié**? Il suffit d'enrichir la boucle for avec cette astuce :

```
for key in sorted(dictionary.keys()):
```

Exemple :

```
dictionary = {"cat": "chat", "dog": "chien", "horse":  
"cheval"}
```

```
for key in dictionary.keys():  
    print(key, "->", dictionary[key])
```

La fonction **sorted()** fera de son mieux et la sortie ressemblera à ceci:

```
cat -> chat  
dog -> chien  
horse -> cheval
```

### 8.3.6 Comment utiliser un dictionnaire : méthodes items() et values()

Une autre méthode est basée sur l'utilisation de la méthode d'un dictionnaire nommée items(). La méthode **renvoie des** tuples (c'est le premier exemple où les tuples sont quelque chose de plus qu'un simple exemple d'eux-mêmes) où **chaque tuple est une paire clé-valeur**.

Voici comment cela fonctionne :

```
dictionary = {"cat": "chat", "dog": "chien", "horse":  
"cheval"}
```

```
for english, french in dictionary.items():  
    print(english, "->", french)
```

Notez la façon dont le tuple a été utilisé comme variable de la boucle for. L'exemple imprime :

```
cat -> chat  
dog -> chien  
horse -> cheval
```

Il existe également une méthode nommée **values()**, qui fonctionne de la même manière que **keys()**, mais **retourne les valeurs**.

Voici un exemple simple :

```
dictionary = {"cat": "chat", "dog": "chien", "horse":  
"cheval"}
```

```
for french in dictionary.values():  
    print(french)
```



Comme le dictionnaire n'est pas capable de trouver automatiquement une clé pour une valeur donnée, le rôle de cette méthode est plutôt limité.

Voici le résultat :

```
cheval
chien
chat
```

#### 8.3.7 Comment utiliser un dictionnaire : modification et ajout de valeurs

L'attribution d'une nouvelle valeur à une clé existante est simple, comme les dictionnaires sont entièrement **mutables**, il n'y a aucun obstacle à leur modification.

Nous allons remplacer la valeur « chat » par « minou ».

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['cat'] = 'minou'
print(dictionary)
```

Le résultat est le suivant :

```
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```

#### 8.3.8 Ajout d'une nouvelle clé

L'ajout d'une nouvelle paire clé-valeur à un dictionnaire est aussi simple que de modifier une valeur, il vous suffit d'attribuer une valeur à une nouvelle **clé auparavant inexistante**.

Remarque : il s'agit d'un comportement très différent par rapport aux listes, qui ne vous permettent pas d'attribuer des valeurs à des index inexistants.

Ajoutons une nouvelle paire de mots au dictionnaire :

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['swan'] = 'cygne'
print(dictionary)
```

L'exemple produit :

```
{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval', 'swan': 'cygne'}
```

Un petit bonus pour vous aider à aller plus loin, vous pouvez également insérer un élément dans un dictionnaire à l'aide de la méthode `update()`, par exemple :

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.update({"duck": "canard"})
print(dictionary)
```

### 8.3.9 Suppression d'une clé

Pouvez-vous deviner comment supprimer une clé d'un dictionnaire?

Remarque : la suppression d'une clé entraînera toujours la **suppression de la valeur** associée. **Les valeurs ne peuvent pas exister sans leurs clés.**

Cela se fait avec l'instruction **del**.

Voici l'exemple :

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

del dictionary['dog']
print(dictionary)
```

Remarque : la **suppression d'une clé inexistante provoque une erreur**.

Pour supprimer le dernier élément d'un dictionnaire, vous pouvez utiliser la méthode **popitem()** :

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.popitem()
print(dictionary)      # outputs: {'cat': 'chat', 'dog': 'chien'}
```

Dans les anciennes versions de Python, c'est-à-dire avant 3.6.7, la méthode **popitem()** supprime un élément aléatoire dans un dictionnaire.

## 8.4 Les tuples et les dictionnaires peuvent fonctionner ensemble

Nous vous avons préparé un exemple simple et parlant pour vous, montrant comment les tuples et les dictionnaires peuvent fonctionner ensemble.

Imaginons le problème suivant :

- vous avez besoin d'un programme pour évaluer les notes moyennes des élèves;
- le programme demande le nom de l'élève, suivi de ses points (unique);
- les noms peuvent être inscrits dans n'importe quel ordre;
- la saisie d'un « nom vide » termine la saisie des données (remarque : la saisie d'un score vide déclenchera l'exception `ValueError`, mais ne vous inquiétez pas maintenant, vous verrez comment gérer de tels cas lorsque nous parlerons d'exceptions dans ce cursus)
- Une liste de tous les noms, ainsi que le score moyen évalué, devraient ensuite être émis.

Voici notre code :

```
school_class = {}

while True:
    name = input("Entrer le nom d'un étudiant: ")
    if name == '':
        break

    score = int(input("Entrer les points (0-10): "))
    if score not in range(0, 11):
        break

    if name in school_class:
        school_class[name] += (score,)
    else:
        school_class[name] = (score,)

for name in sorted(school_class.keys()):
    adding = 0
    counter = 0
    for score in school_class[name]:
        adding += score
        counter += 1
    print(name, ":", adding / counter)
```

Maintenant, analysons-le ligne par ligne:

- **Ligne 1:** Créez un dictionnaire vide pour les données d'entrée; le nom de l'élève est utilisé comme clé, tandis que tous les scores associés sont stockés dans un tuple (le tuple peut être une valeur de dictionnaire - ce n'est pas un problème du tout)
- **Ligne 3 :** Entrez dans une boucle « infinie » (ne vous inquiétez pas, elle se cessera au bon moment)
- **Ligne 4 :** Encodage le nom de l'élève;
- **ligne 5-6 :** si le nom est une chaîne vide (), stopper la boucle ;
- **Ligne 8 :** Demandez les points de l'élève (un entier compris entre 0 et 10)
- **ligne 9-10 :** si le score saisi n'est pas entre 0 et 10, stopper la boucle;
- **Ligne 12-13 :** Si le nom de l'élève est déjà dans le dictionnaire, allongez le tuple associé avec le nouveau score (notez l'opérateur +=)
- **ligne 14-15 :** s'il s'agit d'un nouvel élève (inconnu du dictionnaire), créez une nouvelle entrée, sa valeur est un tuple d'un élément contenant le score saisi;
- **ligne 17 :** parcourir les noms triés des élèves;
- **Ligne 18-19 :** Initialiser les données nécessaires à l'évaluation de la moyenne (somme et compteur)
- **ligne 20-22:** nous itérons à travers le tuple, en prenant toutes les valeurs et en mettant à jour la somme, avec le compteur;
- **Ligne 23 :** Évaluez et imprimez le nom et la note moyenne des élèves.

Voici un exemple d'utilisation du code:

```
Entrer le nom d'un étudiant: Anass
Entrer les points (0-10): 7
Entrer le nom d'un étudiant: Nathan
Entrer les points (0-10) : 3
Entrer le nom d'un étudiant: Anass
Entrer les points (0-10) : 2
Entrer le nom d'un étudiant: Nathan
Entrer les points (0-10) : 10
Entrer le nom d'un étudiant: Nathan
Entrer les points (0-10) : 3
Entrer le nom d'un étudiant: Anass
Entrer les points (0-10) : 9
Entrer le nom d'un étudiant:
Nathan: 5.333333333333333
Anass : 6.0
```

## 8.5 Résumé des tuples

**1. Les tuples** sont des collections de données ordonnées et immuables. Ils peuvent être considérés comme des listes immuables. Ils sont écrits entre parenthèses :

```
my_tuple = (1, 2, True, "a string", (3, 4), [5, 6], None)
print(my_tuple)
```

```
my_list = [1, 2, True, "a string", (3, 4), [5, 6], None]
print(my_list)
```

Chaque élément d'un tuple peut être d'un type différent (entiers, chaînes, booléens, etc.). De plus, les tuples peuvent contenir d'autres tuples ou listes (et inversement).

**2.** Vous pouvez créer un tuple vide comme ceci:

```
empty_tuple = ()
```

**3.** Un tuple à un élément peut être créé comme suit:

```
one_elem_tuple_1 = ("one", )      # Parenthèses et virgule
one_elem_tuple_2 = "one",         # Pas de parenthèses mais une virgule.
```

Si vous supprimez la virgule, vous direz à Python de créer une **variable**, pas un tuple :

```
my_tuple_1 = 1,
print(type(my_tuple_1))      # outputs: <class 'tuple'>
my_tuple_2 = 1
print(type(my_tuple_2))      # outputs: <class 'int'>
```

**4.** Vous pouvez accéder aux éléments tuple en les indexant :

```
my_tuple = (1, 2.0, "string", [3, 4], (5, ), True)
print(my_tuple[3])           # outputs: [3, 4]
```

5. Les tuples sont **immuables**, ce qui signifie que vous ne pouvez pas changer leurs éléments (vous ne pouvez pas ajouter, modifier ou supprimer des éléments de tuple). L'extrait de code suivant provoquera une exception :

```
my_tuple = (1, 2.0, "string", [3, 4], (5, ), True)
my_tuple[2] = "guitar"      # L'exception TypeError sera levée.
```

Toutefois, vous pouvez supprimer un tuple dans son ensemble :

```
my_tuple = 1, 2, 3,
del my_tuple
print(my_tuple)      # NameError: name 'my_tuple' is not defined
```

6. Vous pouvez parcourir en boucle un élément tuple (exemple 1), vérifier si un élément spécifique est (non) présent dans un tuple (exemple 2), utiliser la fonction `len()` pour vérifier combien d'éléments il y a dans un tuple (exemple 3), ou même joindre/multiplier des tuples (exemple 4) :

```
# Example 1
tuple_1 = (1, 2, 3)
for elem in tuple_1:
    print(elem)

# Example 2
tuple_2 = (1, 2, 3, 4)
print(5 in tuple_2)
print(5 not in tuple_2)

# Example 3
tuple_3 = (1, 2, 3, 5)
print(len(tuple_3))

# Example 4
tuple_4 = tuple_1 + tuple_2
tuple_5 = tuple_3 * 2
print(tuple_4)
print(tuple_5)
```

Un petit bonus est que vous pouvez également créer un tuple à l'aide d'une fonction Python intégrée appelée `tuple()`. Ceci est particulièrement utile lorsque vous souhaitez convertir un certain itérable (par exemple, une liste, une plage, une chaîne, etc.) en un tuple :

```
my_tuple = tuple((1, 2, "string"))
print(my_tuple)

my_list = [2, 4, 6]
print(my_list)      # outputs: [2, 4, 6]
print(type(my_list)) # outputs: <class 'list'>
tup = tuple(my_list)
print(tup)          # outputs: (2, 4, 6)
print(type(tup))    # outputs: <class 'tuple'>
```

De la même manière, lorsque vous souhaitez convertir un itérable en liste, vous pouvez utiliser une fonction Python intégrée appelée `list()`:

## 8.6 Résumé des dictionnaires

1. Les dictionnaires sont des collections de données non ordonnées\*, modifiables (mutables) et indexées. (\*Dans Python, les dictionnaires 3.6x sont classés par défaut.

Chaque dictionnaire est un ensemble de *paires clé:valeur*. Vous pouvez le créer à l'aide de la syntaxe suivante :

```
my_dictionary = {  
    key1: value1,  
    key2: value2,  
    key3: value3,  
}
```

2. Si vous souhaitez accéder à un élément du dictionnaire, vous pouvez le faire en faisant référence à sa clé à l'intérieur d'une paire de crochets (ex. 1) ou en utilisant la méthode `get()` (ex. 2) :

```
fr_eng_dictionary = {  
    "fleur": "flower",  
    "eau": "water",  
    "feu": "fire"  
}
```

```
item_1 = fr_eng_dictionary["feu"]    # ex. 1  
print(item_1)    # outputs: fire
```

```
item_2 = fr_eng_dictionary.get("eau")  
print(item_2)    # outputs: water
```

3. Si vous souhaitez modifier la valeur associée à une clé spécifique, vous pouvez le faire en vous référant au nom de la clé de l'élément de la manière suivante :

```
dc_SW_dictionary = {  
    "IronMan": "Maul",  
    "Thor": "Vador",  
    "Strange": "Sidious"  
}
```

```
dc_SW_dictionary["IronMan"] = "Savage"  
item = dc_SW_dictionary["IronMan"]  
print(item)    # outputs: Savage
```

4. Pour ajouter ou supprimer une clé (et la valeur associée), utilisez la syntaxe suivante :

```
phonebook = {}    # dico vide
```

```
phonebook["Adam"] = 3456783958    # creation/ajout pair clé/valeur  
print(phonebook)    # outputs: {'Adam': 3456783958}
```

```
del phonebook["Adam"]  
print(phonebook)    # outputs: {}
```

Vous pouvez également insérer un élément dans un dictionnaire à l'aide de la méthode `update()` et supprimer le dernier élément à l'aide de la méthode `popitem()`, par exemple :

```
fr_eng_dictionary = {"fleur": "flower"}

fr_eng_dictionary.update({"feu": "fire"})
print(fr_eng_dictionary)  # outputs: {'fleur': 'flower', 'feu': 'fire'}

fr_eng_dictionary.popitem()
print(fr_eng_dictionary)  # outputs: {'fleur': 'flower'}
```

5. Vous pouvez utiliser la boucle `for` pour parcourir un dictionnaire :

```
fr_eng_dictionary = {
    "chateau": "castle",
    "eau": "water",
    "feu": "fire"
}

for item in fr_eng_dictionary:
    print(item)
```

6. Si vous souhaitez parcourir en boucle les clés et les valeurs d'un dictionnaire, vous pouvez utiliser la méthode `items()`, par exemple:

```
fr_eng_dictionary = {
    "chateau": "castle",
    "eau": "water",
    "feu": "fire"
}

for key, value in fr_eng_dictionary.items():
    print("Fr /Eng ->", key, ":", value)
```

7. Pour vérifier si une clé donnée existe dans un dictionnaire, vous pouvez utiliser le mot-clé `in`:

```
fr_eng_dictionary = {
    "chateau": "castle",
    "eau": "water",
    "feu": "fire"
}

if "chateau" in fr_eng_dictionary:
    print("Yes")
else:
    print("No")
```

8. Vous pouvez utiliser le mot-clé `del` pour supprimer un élément spécifique ou supprimer un dictionnaire. Pour supprimer tous les éléments du dictionnaire, vous devez utiliser la méthode `clear()` :

```
pol_eng_dictionary = {
    "chateau": "castle",
    "eau": "water",
    "feu": "fire"
}

print(len(fr_eng_dictionary))    # outputs: 3
del fr_eng_dictionary["zamek"]   # remove an item
print(len(fr_eng_dictionary))    # outputs: 2

fr_eng_dictionary.clear()       # removes all the items
print(len(fr_eng_dictionary))    # outputs: 0

del fr_eng_dictionary          # removes the dictionary
```

9. Pour copier un dictionnaire, utilisez la méthode `copy()` :

```
pol_eng_dictionary = {
    "chateau": "castle",
    "eau": "water",
    "feu": "fire"
}

copy_dictionary = pol_eng_dictionary.copy()
```

## 8.7 Résumé : tuples et dictionnaires

### Exercice 1

Que se passe-t-il lorsque vous tentez d'exécuter l'extrait de code suivant ?

```
my_tup = (1, 2, 3)
print(my_tup[2])
```

### Exercice 2

Quelle est la sortie de l'extrait de code suivant ?

```
tup = 1, 2, 3
a, b, c = tup

print(a * b * c)
```

### Exercice 3

Complétez le code pour utiliser correctement la méthode `count()` pour trouver le nombre de doublons de 2 dans le tuple suivant.

```
tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9
duplicates = # Write your code here.

print(duplicates)    # outputs: 4
```



**Exercice 4**

Écrivez un programme qui va « coller » les deux dictionnaires (d1 et d2) ensemble et en créer un nouveau (d3).

```
d1 = {'Johan Depreter': 'A', 'Erwin Desmet': 'B+'}  
d2 = {'Fabrice Scopel': 'A', 'Joakim Chapelle': 'C'}  
d3 = {}  
for item in (d1, d2):  
    # Ecrire ici.  
  
print(d3)
```

**Exercice 5**

Écrivez un programme qui convertira la liste my\_list en tuple.

```
my_list = ["car", "Ford", "flower", "Tulip"]  
t = # Ecrire ici.  
print(t)
```

**Exercice 6**

Écrivez un programme qui convertira le tuple de couleurs en dictionnaire.

```
colors = ("green", "#008000"), ("blue", "#0000FF")  
# Écrivez votre code ici.  
print(colors_dictionary)
```

**Exercice 7**

Que se passe-t-il lorsque vous exécutez le code suivant ?

```
my_dictionary = {"A": 1, "B": 2}  
copy_my_dictionary = my_dictionary.copy()  
my_dictionary.clear()  
  
print(copy_my_dictionary)
```

**Exercice 8**

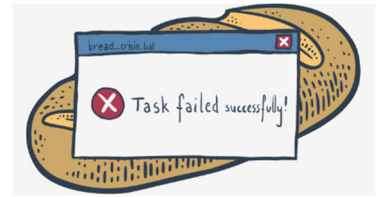
Quel est le résultat du programme suivant?

```
colors = {  
    "white": (255, 255, 255),  
    "grey": (128, 128, 128),  
    "red": (255, 0, 0),  
    "green": (0, 128, 0)  
}  
for col, rgb in colors.items():  
    print(col, ":", rgb)
```

## 9. Les exceptions

### 9.1 Le pain quotidien du développeur

Il semble indiscutable que tous les programmeurs (y compris vous et même nous !) veulent écrire du code sans erreur et faire de leur mieux pour atteindre cet objectif. Malheureusement, rien n'est parfait dans ce monde et les logiciels ne font pas exception. Faites attention au mot **exception** car nous le reverrons très bientôt dans un sens qui n'a rien de commun avec celui que vous connaissez.



L'erreur est humaine. Il est impossible de ne pas faire d'erreurs, et il est impossible d'écrire du code sans erreur. Ne vous méprenez pas, nous ne voulons pas vous convaincre que l'écriture de programmes désordonnés et défectueux est une vertu. Nous voulons plutôt expliquer que même le programmeur le plus prudent n'est pas en mesure d'éviter les défauts mineurs ou majeurs. Ce ne sont que ceux qui ne font rien qui ne font pas d'erreurs.

Paradoxalement, accepter cette vérité difficile peut faire de vous un meilleur programmeur et peut améliorer la qualité de votre code. « Comment cela pourrait-il être possible ? », vous demandez-vous et bien on va tenter de le démontrer.

### 9.2 Erreur de code ou erreur de données

Le traitement des erreurs de programmation a (au moins) deux côtés. Celui qui apparaît lorsque vous avez des ennuis parce que votre code, apparemment correct, est alimenté avec de mauvaises données.

Par exemple, vous vous attendez à ce que le code reçoive une valeur entière, mais votre utilisateur négligent entre des lettres aléatoires à la place, le fameux interface chaise-clavier.

Il peut arriver que votre code soit alors terminé et que l'utilisateur se retrouve seul avec un message d'erreur laconique et ambigu à l'écran. L'utilisateur sera insatisfait, et vous devriez être insatisfait aussi.

Nous allons vous montrer comment protéger votre code de ce genre d'échec et comment ne pas provoquer la colère de l'utilisateur.

L'autre aspect de la gestion des erreurs de programmation se révèle lorsqu'un comportement indésirable du code est causé par des erreurs que vous avez commises lors de l'écriture de votre programme. Ce type d'erreur est communément appelé un « bug », qui est une manifestation d'une croyance bien établie que si un programme fonctionne mal, il doit être causé par des bugs(insectes) malveillants qui vivent à l'intérieur du matériel informatique et provoquent des courts-circuits ou d'autres interférences.

Cette idée n'est pas aussi folle qu'elle puisse paraître, de tels incidents étaient courants à une époque où les ordinateurs occupaient de grandes salles, consommaient des kilowatts d'électricité et produisaient d'énormes quantités de chaleur. Heureusement ou non, ces temps sont révolus pour toujours et les seuls bugs qui peuvent gâcher votre code sont ceux que vous avez semés dans le code vous-même. Par conséquent, nous allons essayer de vous montrer comment trouver et éliminer vos bugs, en d'autres termes, comment déboguer votre code. Commençons ici le voyage à travers le pays des erreurs et des bugs.

### 9.3 Quand les données ne sont pas ce qu'elles devraient

Écrivons un morceau de code extrêmement trivial, il lira un entier naturel (un entier non négatif) et imprimera son réciproque. De cette façon, 2 se transformera en 0,5 ( $1/2$ ) et 4 en 0,25 ( $1/4$ ). Voici le programme :

```
value = int(input('Entrer un nombre naturel: '))
print('Le réciproque de', value, 'est', 1/value)
```

Y a-t-il quelque chose qui peut mal tourner dans ce code? Le code est si bref et si compact qu'il ne semble pas que nous puissions y trouverons le moindre problème.

Mais je suis sûr que vous voyez pourquoi on vous ennuie avec ça ! Et oui, vous avez raison (enfin j'espère), entrer des données qui ne sont pas un entier (ce qui inclut également de ne rien entrer du tout) ruinera complètement l'exécution du programme.

Voici ce que l'utilisateur du code verra :

```
Traceback (most recent call last):
  File "code.py", line 1, in
    value = int(input('Entrer un nombre naturel: '))
ValueError: invalid literal for int() with base 10: ''
```

Toutes les lignes que Python vous montre dans une erreur sont significatives et importantes, mais la dernière ligne semble être la plus précieuse souvent. Le premier mot de la ligne est le nom de **l'exception** qui provoque l'arrêt de votre code. C'est **ValueError** ici. Le reste de la ligne n'est qu'une brève explication qui spécifie plus précisément la cause de l'exception survenue.

Comment vous pourriez vous y prendre pour empêcher ceci ? Comment protégez-vous votre code de la résiliation, l'utilisateur de la déception et vous-même de l'insatisfaction de l'utilisateur ?

La toute première pensée qui peut vous venir à l'esprit est de vérifier si les données fournies par l'utilisateur sont valides et de refuser à votre code de coopérer si les données sont incorrectes. Dans ce cas, la vérification peut s'appuyer sur le fait que nous nous attendons à ce que la chaîne d'entrée ne contienne que des chiffres.

Vous devriez déjà être en mesure de mettre en œuvre cette vérification et de l'écrire vous-même !

Il est également possible de vérifier si le type de la variable de **valeur** est un **int** (Python a un moyen spécial pour ce type de vérifications, c'est l'opérateur nommé **is**. La vérification elle-même peut ressembler à ceci :

```
type(value) is int
```

et donne la valeur **true** si le type de la variable de valeur actuelle est **int**. Si vous voulez plus d'infos sur cet opérateur, vous pouvez vous référer à la documentation officielle ou attendre le q2 :-)

Revenons à notre code et soyez surpris car nous ne voulons pas que vous fassiez de validation préliminaire des données.

Pourquoi? Parce que ce n'est pas la façon dont Python recommande ce travail, dans d'autres langages cela sera plus opportun.

#### 9.4 le Try-except

Dans le monde étrange de Python, il y a une règle qui dit:

**« Il vaut mieux demander pardon que de demander la permission ».**

Mais !? Arrêtons-nous ici un instant et parlons un peu de ce concept.

Ne vous méprenez pas, nous ne voulons pas que vous appliquiez la règle dans votre vie quotidienne (surtout pas d'ailleurs).

Ne prenez pas la voiture de quelqu'un sans permission dans l'espoir d'être si convaincant que vous éviterez une condamnation par exemple. Cette règle concerne juste vos codes Python.

En fait, la règle se lit comme suit: **« il vaut mieux gérer une erreur quand elle se produit que d'essayer de l'éviter ».**

Et là vous allez répondre au prof qui est devant vous :

**« D'accord, me direz-vous maintenant, mais comment devrais-je demander pardon quand le programme est terminé et qu'il ne reste plus rien à faire ? C'est malin comme idée, encore.»**

Et bien c'est là que l'exception entre en jeu.

```
try:
    # It's a place where
    # you can do something
    # without asking for permission.
except:
    # It's a spot dedicated to
    # solemnly begging for forgiveness.
```

Vous pouvez voir deux branches, deux chemins dans ce code:

- Tout d'abord, en commençant par le mot-clé **try**, c'est l'endroit où vous mettez le code que vous soupçonnez risqué et qui peut être terminé en cas d'erreur;  
note: ce type d'erreur est appelé une exception, tandis que l'occurrence d'exception est dite **levée**, nous pouvons donc dire qu'une exception est (ou a été) levée;
- Deuxièmement, la partie du code commençant par le mot-clé **except** est conçue pour gérer l'exception; c'est à vous de décider ce que vous voulez y faire: vous pouvez « nettoyer le désordre » ou vous pouvez simplement balayer le problème sous le tapis (bien que nous préférerions la première solution et que vous serez évalué selon celle-ci).

Donc, nous pourrions dire que ces deux blocs fonctionnent comme ceci:

- Le mot-clé **try** marque l'endroit où vous essayez de faire quelque chose sans autorisation;
- Le mot-clé **except** commence l'endroit où vous pouvez montrer vos talents en excuse.

Comme vous pouvez le constater, cette approche accepte les erreurs (les traite comme une partie normale de la vie du programme) au lieu d'intensifier les efforts pour les éviter.

### 9.5 L'exception prouve la règle

Comme on le dit dans le dicton bien connu, l'exception fait règle.

Réécrivons donc le code pour l'adapter à l'approche Python de la vie:

- Toute partie du code placée entre try et except est exécutée d'une manière spéciale, toute erreur qui se produit ici **ne mettra pas fin à l'exécution du programme**. Au lieu de cela, le contrôle passera immédiatement à la première ligne située après le mot-clé except et aucune autre partie de la branche try ne sera exécutée ;
- Le code de la branche EXCEPT est activé uniquement lorsqu'une exception a été rencontrée dans le bloc TRY. Il n'y a aucun moyen d'y arriver par d'autres moyens;
- Lorsque le bloc try ou le bloc except est exécuté avec succès, le contrôle revient au chemin d'exécution normal et tout code situé au-delà dans le fichier source est exécuté comme si de rien n'était.

Maintenant, nous voulons vous poser une question « innocente » : **ValueError** est-il la seule façon dont le contrôle pourrait tomber dans la branche except ?

```
try:
    value = int(input('Entrer un nombre naturel: '))
    print('Le réciproque de', value, 'est', 1/value)
except:
    print('Je ne sais pas quoi faire...')
```

## 9.6 La gestion de plusieurs erreurs possibles

La réponse à la question ci-dessus est évidemment « non », il y a plus d'une façon possible de lever une exception.

Par exemple, un utilisateur peut entrer zéro en entrée, au vu de vos cours de Math, vous pouvez prédire ce qu'il va arriver.

La division placée à l'intérieur de l'appel de la fonction `print()` déclenchera l'erreur **ZeroDivisionError**.

Comme vous pouvez vous y attendre, le comportement du code sera le même que dans le cas précédent, l'utilisateur verra « Je ne sais pas quoi faire... », ce qui semble correcte dans ce contexte, mais il est également possible que vous souhaitiez gérer ce genre de problème d'une manière un peu différente.

Est-ce possible? Bien sûr. Il y a au moins deux approches que vous pouvez mettre en œuvre :

La première est simple et compliquée à la fois: vous pouvez simplement ajouter deux blocs **try** distincts, l'un incluant l'appel de la fonction `input()` où la `ValueError` peut être levée, et le second consacré à la gestion des problèmes éventuels induits par la division. Ces deux blocs d'essai auraient leurs propres branches **except** et, en effet, vous aurez le contrôle sur deux erreurs différentes.

Cette solution est bonne, mais elle est un peu longue, le code devient inutilement volumineux. De plus, ce n'est pas le seul danger qui vous attend.

Notez que laisser le premier bloc `try-except` laisse beaucoup d'incertitude, vous devrez ajouter du code supplémentaire pour vous assurer que la valeur que l'utilisateur a entrée est sûre à utiliser dans la division. C'est ainsi qu'une solution apparemment simple devient en fait trop compliquée.

Heureusement, Python offre un moyen plus simple de faire face à ce genre de défi.

### *Deux exceptions après un try*

Regardez le code suivant :

```
try:
    value = int(input('Entrer un nombre naturel: '))
    print('Le réciproque de', value, 'est', 1/value)
except ValueError:
    print('Je ne sais pas quoi faire...')
except ZeroDivisionError:
    print('La division par zero n\'est pas valide dans notre univers.')
```

Comme vous pouvez le voir, nous venons d'introduire une deuxième branche à l'exception. Ce n'est pas la seule différence, notez que les deux branches ont des noms d'exception spécifiés. Dans cette variante, chacune des exceptions attendues a sa propre façon de gérer l'erreur, mais il faut souligner qu'une seule de toutes les branches peut intercepter le contrôle, si l'une des branches est exécutée, toutes les autres branches restent inactives.

De plus, le nombre de branches à l'exception n'est pas limité – vous pouvez en spécifier autant ou aussi peu que vous le souhaitez, mais aucune des exceptions ne peut être spécifiée plus d'une fois.

## 9.7 L'exception par défaut

Changeons encore un petit peu notre code :

```
try:
    value = int(input('Entrer un nombre naturel: '))
    print('Le réciproque de', value, 'est', 1/value)
except ValueError:
    print('Je ne sais pas quoi faire...')
except ZeroDivisionError:
    print('La division par zero n\'est pas valide dans notre univers.')
except:
    print('Quelque chose de bizarre ce passe...désolé!')
```

Nous avons ajouté une troisième branche à l'exception, mais cette fois, elle n'a pas de nom d'exception spécifié, nous pouvons l'appeler l'exception « anonyme » ou (ce qui est plus proche de son rôle réel) c'est « la valeur par défaut ». Vous pouvez vous attendre à ce que lorsqu'une exception soit déclenchée et qu'il n'y a pas de branche d'exception dédiée à celle-ci, elle soit gérée par la branche par défaut.

Remarques : La branche par défaut doit être la dernière branche. Toujours!!

## 9.8 Les exceptions les plus courantes

Discutons plus en détail de certaines exceptions utiles (ou plutôt les plus courantes) que vous pouvez rencontrer.

### 9.8.1 ZeroDivisionError

Elle apparaît lorsque vous essayez de forcer Python à effectuer une opération qui provoque une division dans laquelle le diviseur est nul ou impossible à distinguer de zéro. Notez qu'il existe plusieurs opérateurs Python qui peuvent provoquer le déclenchement de cette exception → /, //, et %.

### 9.8.2 ValueError

Attendez-vous à cette exception lorsque vous avez affaire à des valeurs qui peuvent être utilisées de manière inappropriée dans un certain contexte. En général, cette exception est déclenchée lorsqu'une fonction (comme `int()` ou `float()`) reçoit un argument d'un type approprié, mais que sa valeur est inacceptable.

### 9.8.3 TypeError

Cette exception apparaît lorsque vous essayez d'appliquer une donnée dont le type ne peut pas être accepté dans le contexte actuel. Regardez l'exemple :

```
short_list = [1]
one_value = short_list[0.5]
```

Vous n'êtes pas autorisé à utiliser une valeur flottante comme index de liste (la même règle s'applique également aux tuples). **TypeError** est un nom adéquat pour décrire le problème et une exception adéquate à lever.

### 9.8.4 AttributeError

Cette exception arrive, entre autres occasions, lorsque vous essayez d'activer une méthode qui n'existe pas dans un élément que vous traitez. Par exemple:

```
short_list = [1]
short_list.append(2)
short_list.depend(3)
```

La troisième ligne de notre exemple tente d'utiliser une méthode qui n'est pas contenue dans les listes. C'est l'endroit où **AttributeError** est déclenché.

### 9.8.5 SyntaxError

Cette exception est déclenchée lorsque le contrôle atteint une ligne de code qui viole la grammaire de Python. Cela peut sembler étrange, mais certaines erreurs de ce type ne peuvent pas être identifiées sans exécuter au préalable le code. Ce type de comportement est typique des langages interprétés, l'interprète travaille toujours à la hâte et n'a pas le temps de scanner tout le code source. Il se contente de vérifier le code en cours d'exécution.

C'est une mauvaise idée de gérer cette exception dans vos programmes. Vous devez produire du code exempt d'erreurs de syntaxe, au lieu de masquer les erreurs que vous avez causées.



## 10 Les tests

### 10.1 Tester son code est une obligation, une loi DSTienne !

Bien que nous allions conclure nos considérations sur les exceptions ici, ne pensez pas que c'est tout ce que Python peut offrir pour vous aider à demander pardon.

La machinerie des exceptions de Python est beaucoup plus complexe et ses capacités vous permettent de créer des stratégies étendues de gestion des erreurs. Nous reviendrons sur ces questions, nous vous le promettons, Mrs Depreter et Desmet seront intransigeant sur ces points.

N'hésitez pas à mener vos expériences et à plonger vous-même dans les exceptions entre temps.

Maintenant, nous voulons vous parler du deuxième côté de la lutte sans fin avec les erreurs, le dur destin inévitable de la vie d'un développeur.

Comme vous n'êtes pas en mesure d'éviter de faire des bugs dans votre code, vous devez toujours être prêt à les rechercher et à les détruire. Ne faites pas l'autruche, ignorer les erreurs ne les fera pas disparaître au contraire.

Une tâche importante pour les développeurs est de tester le code nouvellement créé, mais vous ne devez pas oublier que **le test n'est pas un moyen de prouver que le code est exempt d'erreurs**. Paradoxalement, la seule preuve que les tests peuvent fournir est que votre code contient des erreurs. Ne pensez pas que vous pouvez vous détendre après un test réussi.

Le deuxième aspect important des tests logiciels est strictement psychologique. C'est une vérité connue depuis des années que les auteurs, même ceux qui sont fiables et conscients d'eux-mêmes, **ne sont pas en mesure d'évaluer et de vérifier objectivement leurs œuvres**.

C'est pourquoi chaque romancier a besoin d'un éditeur et chaque programmeur a besoin d'un testeur.

Certains disent, un peu méchamment mais honnêtement, que les développeurs testent le code pour montrer leur perfection, pas pour trouver des problèmes qui peuvent les frustrer.

Les testeurs sont exempts de tels dilemmes, et c'est pourquoi leur travail est plus efficace et rentable.

Bien sûr, cela ne vous dispense pas d'être attentif et prudent. Testez votre code du mieux que vous ne le pouvez. Ne rendez pas le travail des testeurs trop facile. Votre tâche principale est de vous **assurer que vous avez vérifié tous les chemins d'exécution que** votre code peut emprunter. Cela vous semble-t-il mystérieux encore mystérieux ? Vous n'avez pas été assez au cours de Mr Depreter !

### *Suivi des chemins d'exécution*

Regardez le code ci-dessous. Supposons que vous veniez de finir de l'écrire.

```
temperature = float(input('Entrer la temperature actuelle:'))

if temperature > 0:
    print("au dessus de zero")
elif temperature < 0:
    print("en dessous de zero")
else:
    print("Zero")
```

Il y a trois chemins d'exécution indépendants dans le code. Ils sont déterminés par les instructions **if-elif-else**. Bien sûr, les chemins d'exécution peuvent être construits par de nombreuses autres instructions, comme des boucles, ou même des blocs try-except .

Si vous voulez tester votre code équitablement et que vous voulez dormir profondément et rêver sans cauchemars (les cauchemars sur les bugs peuvent être dévastateurs pour les performances d'un développeur, on l'a vécu, on vous l'assure), vous êtes obligé de préparer un ensemble de données de test qui forcera votre code à négocier tous les chemins possibles.

Dans notre exemple, l'ensemble doit contenir au moins trois valeurs flottantes : une positive, une négative et zéro.

### 10.2 Quand python ferme les yeux

Reprenons notre code :

```
temperature = float(input('Entrer la temperature actuelle:'))

if temperature > 0:
    print("au dessus de zero")
elif temperature < 0:
    prin("en dessous de zero")
else:
    print("Zero")
```

Nous avons intentionnellement introduit une erreur dans le code, nous espérons que vos yeux vigilants l'ont immédiatement remarquée. Oui, nous n'avons supprimé qu'une seule lettre et en effet, l'invocation valide de la fonction print() se transforme en clause invalide « prin() ». Il n'y a pas de fonction telle que « prin() » dans la portée de notre programme, mais est-ce vraiment évident pour Python?

Exécutez le code et entrez la valeur 0.

Comme vous pouvez le voir, le code termine son exécution sans aucun obstacle. Comment est-ce possible? Pourquoi Python **néglige-t-il** une erreur de développeur aussi évidente ?

Pouvez-vous trouver les réponses à ces questions fondamentales?

### 10.3 Tests, essais et testeurs

La réponse est plus simple que vous ne le pensez, et un peu décevante aussi. Python, comme vous le savez avec certitude, est un langage **interprété**. Cela signifie que le code source est analysé et exécuté au besoin. Par conséquent, Python peut ne pas avoir le temps d'analyser les lignes de code qui ne sont pas sujettes à exécution. Comme le dit un vieux dicton de développeur: « *c'est une fonctionnalité, pas un bug* » (s'il vous plaît n'utilisez pas cette phrase pour justifier le comportement étrange de votre code, mais vous l'entendrez...souvent).

Comprenez-vous maintenant pourquoi passer par toutes les voies d'exécution est si vital et inévitable ?

Supposons que vous terminiez votre code et que les tests que vous avez effectués réussissent. Vous livrez votre code aux testeurs et, heureusement !, ils y trouve quelques bugs. Nous utilisons le mot « *heureusement* » tout à fait consciemment. Vous devez accepter que, premièrement, les testeurs sont les meilleurs amis du développeur, ne traitez pas les bugs qu'ils découvrent comme une offense ou une malignité; et, deuxièmement, chaque bug trouvé par les testeurs est un bug qui n'affectera pas les utilisateurs. Ces deux facteurs sont précieux et méritent votre attention. Vous savez déjà que votre code contient un ou plusieurs bugs (ce dernier est plus probable). Comment les localisez-vous et comment corrigez-vous votre code ?

#### 10.3.1 Bug vs débogage

La mesure de base qu'un développeur peut utiliser contre les bugs est un **débogueur**, tandis que le processus au cours duquel les bugs sont supprimés du code est appelé **débogage**. Selon une vieille blague, le débogage est un jeu de mystère compliqué dans lequel vous êtes à la fois le meurtrier, le détective et, la partie la plus douloureuse de l'intrigue, la victime. Êtes-vous prêt à jouer tous ces rôles ? Ensuite, vous devez vous armer d'un débogueur.



Un débogueur est un logiciel spécialisé qui peut contrôler la façon dont votre programme est exécuté. À l'aide du débogueur, vous pouvez exécuter votre code ligne par ligne, inspecter tous les états des variables et modifier leurs valeurs à la demande sans modifier le code source, arrêter l'exécution du programme lorsque certaines conditions sont ou ne sont pas remplies et effectuer de nombreuses autres tâches utiles.

On peut dire que chaque IDE est équipé d'un débogueur plus ou moins avancé.

## 10.4 Débug d'impression

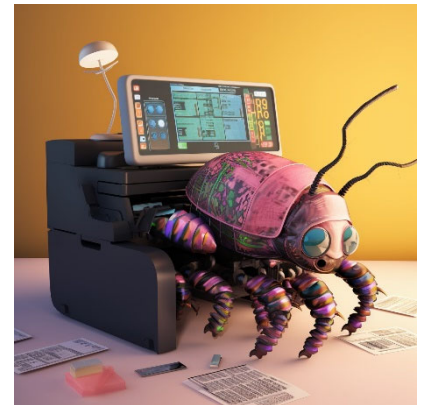
Cette forme de débogage, qui peut être appliquée à votre code à l'aide de n'importe quel type de débogueur, est parfois appelée **débogage interactif**. La signification du terme est explicite, le processus nécessite votre interaction (celle du développeur) pour être effectué.

D'autres techniques de débogage peuvent être utilisées pour chasser les bugs. Il est possible que vous ne puissiez pas ou ne souhaitiez pas utiliser un débogueur.

Vous pouvez utiliser l'une des tactiques de débogage les plus simples et les plus anciennes (mais toujours utiles) connues sous le nom de **débogage d'impression**. Le nom

parle de lui-même, il vous suffit d'insérer plusieurs invocations **print()** supplémentaires dans votre code pour générer des données qui illustrent le chemin que votre code négocie actuellement. Vous pouvez générer les valeurs des variables qui peuvent affecter l'exécution.

Ces impressions peuvent générer du texte significatif tel que « *Je suis ici* », « *Je suis entré dans la fonction foo()* », « *Le résultat est 0* », ou elles peuvent contenir des séquences de caractères qui ne sont lisibles que par vous. S'il vous plaît, n'utilisez pas de mots obscènes ou indécents à cette fin, même si vous ressentez une forte tentation, si nous avons à faire à ce genre de comportement vous serez sanctionné.



Comme vous pouvez le voir, ce type de débogage n'est pas vraiment interactif du tout, ou n'est interactif que dans une faible mesure, lorsque vous décidez d'appliquer la fonction `input()` pour arrêter ou retarder l'exécution du code. Une fois les bugs trouvés et supprimés, les impressions supplémentaires peuvent être commentées ou supprimées, c'est à vous de décider. Ne les laissez pas être exécutés dans le code final, ils peuvent confondre à la fois les testeurs et les utilisateurs, et vous apporter un mauvais karma.

## 10.5 Quelques conseils utiles

Voici quelques conseils qui peuvent vous aider à trouver et à éliminer les bugs. Aucun d'entre eux n'est ultime ou définitif. Utilisez-les de manière flexible et fiez-vous à votre intuition. Ne vous croyez pas, non dans ce cas jamais !, vérifiez tout deux fois.

- **Essayez de dire à quelqu'un** (par exemple, votre ami ou collègue) ce que votre code est censé faire et comment il se comporte réellement. Soyez concret et n'omettez pas de détails. Répondez à toutes les questions posées par votre nouvel assistant. Vous réaliserez probablement la cause du problème en racontant votre histoire, car parler active des parties de votre cerveau qui restent inactives pendant le codage. Si aucun humain ne peut vous aider avec le problème, utilisez plutôt un canard en caoutchouc jaune. Nous ne plaisantons pas, cela existe, pour votre culture consultez l'article de Wikipedia pour en savoir plus sur cette technique couramment utilisée: [le débogage de canard en caoutchouc](https://en.wikipedia.org/wiki/Rubber_duck_debugging) ([https://en.wikipedia.org/wiki/Rubber\\_duck\\_debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging)).

- **Essayez d'isoler le problème.** Vous pouvez extraire la partie de votre code problématique et l'exécuter séparément. Vous pouvez commenter les parties du code qui masquent le problème. Attribuez des valeurs concrètes aux variables au lieu de les lire à partir de l'entrée clavier. Testez vos fonctions en appliquant des valeurs d'argument prévisibles. Analysez attentivement le code. Lisez-le à haute voix.
- Si le bug est apparu récemment et n'est pas apparu plus tôt, **analysez toutes les modifications que vous avez introduites dans votre code**, l'une d'entre elles peut en être la raison, vous apprendrez le versionning au Q2.
- **Faites une pause**, buvez une tasse de café, allez-vous promener, lisez un bon livre pendant un moment, téléphonez à votre meilleur ami, vous serez surpris de voir combien de fois cela aide de se changer les idées.
- **Soyez optimiste** – vous finirez par trouver le bug;

### 10.6 Les tests unitaires -Le codage de haut niveau

Il existe également une technique de programmation importante et largement utilisée que vous devrez adopter dans votre cursus et au cours de votre carrière de développeur: c'est ce qu'on appelle les tests unitaires.

Le nom peut être un peu déroutant, car il ne s'agit pas seulement de tester le logiciel, mais aussi (et surtout) de la façon dont le code est écrit.

Pour faire court, les tests unitaires supposent que les tests sont des parties inséparables du code et que la préparation des données de test est une partie inséparable du codage.

Cela signifie que lorsque vous écrivez une fonction ou un ensemble de fonctions coopérantes, vous êtes également obligé de créer un ensemble de données pour lesquelles le comportement de votre code est prévisible et connu.

De plus, vous devez équiper votre code d'une interface utilisable par un environnement de test automatisé. Dans cette approche, toute modification apportée au code (même la moins significative) doit être suivie de l'exécution de tous les tests unitaires accompagnant votre source.

Pour standardiser cette approche et la rendre plus facile à appliquer, Python fournit un module dédié nommé **unittest**. Nous n'allons pas en discuter ici – c'est un sujet vaste et complexe mais le verrez au cours de votre cursus. Au moment d'écrire ce cours, le cours d'unittest se déroule en bac 2.

## 10.7 Résumé chapitre 9 et 10

1. En Python, il existe une distinction entre deux types d'erreurs: **erreurs de syntaxe** (syntax ou parsing), qui se produisent lorsque l'analyseur rencontre une instruction incorrecte. Par exemple:

Tentative d'exécution de la ligne suivante :

```
print("Hello, World!)
```

provoquera une *erreur* `SyntaxError` et entraînera l'affichage du message suivant (ou similaire) dans la console :

```
File "main.py", line 1
```

```
    print("Hello, World!")
                        ^
```

```
SyntaxError: EOL while scanning string literal
```

Faites attention à la flèche, elle indique l'endroit où l'analyseur Python a rencontré des problèmes. Dans notre cas, c'est le guillemet double manquant. L'aviez-vous remarqué?

**L**

**es exceptions**, qui se produisent même lorsqu'une instruction / expression est syntaxiquement correcte; ce sont les erreurs qui sont détectées lors de l'exécution lorsque votre code entraîne une erreur qui n'est pas toujours fatale. Par exemple:

Tentative d'exécution de la ligne suivante :

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

provoquera une exception `ZeroDivisionError` et entraînera l'affichage du message suivant (ou similaire) dans la console :

```
Traceback (most recent call last):
```

```
  File "main.py", line 1, in
```

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

```
ZeroDivisionError: division by zero
```

Faites attention à la dernière ligne du message d'erreur, elle vous indique en fait ce qui s'est passé. Il existe de nombreux types d'exceptions, telles que `ZeroDivisionError`, `NameError`, `TypeError` et bien d'autres ; et cette partie du message vous informe du type d'exception qui a été déclenché. Les lignes précédentes vous montrent le contexte dans lequel l'exception s'est produite.

2. Vous pouvez « attraper » (catch) et gérer les exceptions en Python en utilisant le bloc *try-except* . Donc, si vous soupçonnez qu'un extrait particulier peut déclencher une exception, vous pouvez écrire le code qui le gèrera et n'interrompra pas le programme. Regardez l'exemple :

```
while True:
    try:
        number = int(input("Enter an integer number: "))
        print(number/2)
        break
    except:
        print("Warning: the value entered is not a valid number. Try again...")
```

Le code ci-dessus demande à l'utilisateur d'entrer une valeur jusqu'à ce qu'il entre un nombre entier valide. Si l'utilisateur entre une valeur qui ne peut pas être convertie en int, le programme imprimera l'avertissement, puis demandera à l'utilisateur d'entrer à nouveau un nombre.

Le bloc try est exécuté. L'utilisateur entre une valeur erronée, par exemple : hello! .

Une exception se produit et le reste de la clause try est ignoré. Le programme accède au bloc d'exception, l'exécute, puis continue à s'exécuter après le bloc try-except.

Si l'utilisateur entre une valeur correcte et qu'aucune exception ne se produit, les instructions suivantes du bloc try sont exécutées.

3. Vous pouvez gérer plusieurs exceptions dans votre bloc de code. Regardez les exemples suivants :

```
while True:
    try:
        number = int(input("Enter an int number: "))
        print(5/number)
        break
    except ValueError:
        print("Wrong value.")
    except ZeroDivisionError:
        print("Sorry. I cannot divide by zero.")
    except:
        print("I don't know what to do...")
```

Vous pouvez utiliser plusieurs blocs « *except* » dans une instruction *try* et spécifier des noms d'exception particuliers. Si l'une des branches **except** est exécutée, les autres branches seront ignorées. N'oubliez pas : vous ne pouvez spécifier une exception intégrée particulière qu'une seule fois. N'oubliez pas non plus que l'exception **par défaut** (ou générique), c'est-à-dire celle sans nom spécifié, doit être placée en **bas de la branche** (utilisez d'abord les exceptions les plus spécifiques et les exceptions plus générales en dernier).

Vous pouvez également spécifier et gérer plusieurs exceptions intégrées au sein d'une même clause *except* :

```
while True:
    try:
        number = int(input("Enter an int number: "))
        print(5/number)
        break
    except (ValueError, ZeroDivisionError):
        print("Wrong value or No division by zero rule broken.")
    except:
        print("Sorry, something went wrong...")
```

4. Certaines des exceptions intégrées Python les plus utiles sont: *ZeroDivisionError*, *ValueError*, *TypeError*, *AttributeError* et *SyntaxError*. Une autre exception qui, à notre avis, mérite votre attention est l'exception *KeyboardInterrupt*, qui est déclenchée lorsque l'utilisateur appuie sur la touche d'interruption (CTRL-C ou Suppr). Exécutez le code ci-dessus et appuyez sur la combinaison de touches pour voir ce qui se passe. Pour en savoir plus sur les exceptions intégrées de Python, consultez la documentation officielle de Python.

5. Enfin et surtout, vous devez vous rappeler de tester et de déboguer votre code. Utilisez des techniques de débogage telles que le débogage *d'impression*; si possible, demandez à quelqu'un de lire votre code et de vous aider à y trouver des bugs ou à l'améliorer; essayez d'isoler le fragment de code qui est problématique et susceptible d'erreurs: **testez vos fonctions** en appliquant des valeurs d'argument prévisibles et essayez de **gérer** les situations où quelqu'un entre des valeurs incorrectes; Enfin, faites des pauses et revenez à votre code après un certain temps avec une nouvelle paire d'yeux.

### Exercice

Quelle est la sortie du programme suivant si l'utilisateur entre 0 ?

```
try:
    value = int(input("Enter a value: "))
    print(value/value)
except ValueError:
    print("Bad input...")
except ZeroDivisionError:
    print("Very bad input...")
except:
    print("Booo!")
```



## 10.8 Exercice

### Temps estimé

30-120 minutes

### Objectifs

- perfectionner les compétences de l'étudiant dans l'utilisation de Python pour résoudre des problèmes complexes;
- Intégrer des techniques de programmation dans un programme composé de nombreuses parties différentes.

### Scénario

Votre tâche est d'écrire **un programme simple qui prétend jouer au *tic-tac-toe* avec l'utilisateur**. Pour vous faciliter la tâche, nous avons décidé de simplifier le jeu. Voici nos hypothèses :

- l'ordinateur (c'est-à-dire votre programme) doit jouer au jeu en utilisant des « X » ;
- l'utilisateur (par exemple, vous) doit jouer au jeu en utilisant des « O » ;
- le premier mouvement appartient à l'ordinateur - il met toujours son premier « X » au milieu du tableau;
- Tous les carrés sont numérotés ligne par ligne en commençant par 1 (voir l'exemple ci-dessous pour référence)
- l'utilisateur saisit son déplacement en entrant le numéro du carré qu'il choisit, le nombre doit être valide, c'est-à-dire qu'il doit être un entier, il doit être supérieur à 0 et inférieur à 10, et il ne peut pas pointer vers un champ déjà occupé;
- Le programme vérifie si le jeu est terminé, il y a quatre verdicts possibles: le jeu doit continuer, le jeu se termine par un match nul, vous gagnez ou l'ordinateur gagne;
- l'ordinateur répond par son déplacement et la vérification est répétée;
- N'implémentez aucune forme d'intelligence artificielle, un choix de champ aléatoire fait par l'ordinateur est suffisant pour ce jeu.

L'exemple de session avec le programme peut se présenter comme suit :

1	2	3
4	X	6
7	8	9

Entrez votre chiffre: 1

0	2	3
4	X	6
7	8	9

0	X	3
4	X	6
7	8	9

Entrez votre chiffre: 8

0	X	3
4	X	6
7	0	9

0	X	3
4	X	X
7	0	9

Entrez votre chiffre: 4

0	X	3
0	X	X
7	0	9
0	X	X
0	X	X
7	0	9

Entrez votre chiffre: 7

0	X	X
0	X	X
0	0	9

You won!

## Exigences

Implémentez les fonctionnalités suivantes :

- Le tableau doit être stocké sous la forme d'une liste à trois éléments, tandis que chaque élément est une autre liste à trois éléments (les listes internes représentent des lignes) afin que tous les carrés soient accessibles à l'aide de la syntaxe suivante :  
`board[row][column]`
- chacun des éléments de la liste intérieure peut contenir 'O', 'X' ou un chiffre représentant le nombre du carré (ce carré est libre)
- L'apparence du tableau doit être exactement la même que celle présentée.
- implémenter les fonctions définies pour vous dans l'éditeur.

Le choix d'un nombre entier aléatoire peut être effectué en utilisant une fonction Python appelée `randrange()`. L'exemple de programme ci-dessous montre comment l'utiliser (le programme imprime dix nombres aléatoires de 0 à 8).

```
from random import randrange

for i in range(10):
    print(randrange(8))
```

Remarque : l'instruction `from-import` permet d'accéder à la fonction `randrange` définie au sein d'un module Python externe appelé `random`.

Voici un début de code :

```
def display_board(board):
    #La fonction accepte un paramètre contenant l'état actuel
    # de la carte et l'imprime sur la console.

def enter_move(board):
    # La fonction accepte l'état actuel du tableau, interroge
    #l'utilisateur sur son mouvement, vérifie l'entrée et met
    # à jour le tableau en fonction de la décision de
    #l'utilisateur.

def make_list_of_free_fields(board):
    #La fonction parcourt le tableau et construit une liste de
    #toutes les cases libres ;
    # la liste se compose de tuples, tandis que chaque tuple
    # est une paire de numéros de ligne et de colonne.

def victory_for(board, sign):
    # La fonction analyse l'état de la carte afin de vérifier
    #si le joueur utilisant des 'O'ou des 'X' a gagné la partie

def draw_move(board):
    # La fonction dessine le mouvement de l'ordinateur et met
    à jour le tableau.
```

## 11 Les chaînes de caractères et les listes en détails

### 11.1 La nature des chaînes de caractères en Python

#### 11.1.1 Les chaînes en détails

Faisons un bref rappel de la nature des chaînes de caractères en Python.

Tout d'abord, les chaînes de Python (ou simplement les chaînes, car nous n'allons pas discuter des chaînes d'un autre langage) sont des **séquences immuables**.

Il est très important de le noter, car cela signifie que vous devez vous attendre à un comportement qui vous est familier.

Analysons un code pour comprendre de quoi nous parlons:

```
# Example 1
```

```
word = 'by'  
print(len(word))
```

```
# Example 2
```

```
empty = ''  
print(len(empty))
```

```
# Example 3
```

```
i_am = 'I\'m'  
print(len(i_am))
```

- Jetez un coup d'œil à **l'exemple 1**. La fonction `len()` utilisée pour les chaînes renvoie un certain nombre de caractères contenus dans les arguments. L'extrait de code génère 2.
- N'importe quelle chaîne peut être vide. Sa longueur est alors 0, **exemple 2**.
- N'oubliez pas qu'une barre oblique inverse (`\`) utilisée comme caractère d'échappement n'est pas incluse dans la longueur totale de la chaîne. Le code de **l'exemple 3** génère donc 3.

### 11.1.2 Chaînes multilignes

Il est temps de vous montrer une autre façon de spécifier des chaînes dans le code source Python. Notez que la syntaxe que vous connaissez déjà ne vous permettra pas d'utiliser une chaîne occupant plus d'une ligne de texte.

Pour cette raison, ce code est erroné :

```
multiline = 'Line #1
Line #2'

print(len(multiline))
```

Heureusement, pour ces types de chaînes, Python offre une syntaxe distincte, pratique et simple.

Regardez le code :

```
multiline = '''Line #1
Line #2'''

print(len(multiline))
```

Comme vous pouvez le voir, la chaîne commence par **trois apostrophes**, pas une. La même apostrophe triplée est utilisée pour y mettre fin. Le nombre de lignes de texte placées à l'intérieur d'une telle chaîne est arbitraire.

L'extrait de code générera 15.

Comptez soigneusement les caractères. Ce résultat est-il correct ou non? Cela semble correct à première vue, mais quand vous comptez les caractères, ce n'est pas le cas.

La ligne #1 contient sept caractères. Deux de ces lignes comportent 14 caractères. Avons-nous perdu un caractère en route? Où? Comment?

Non non il est bien là !

**Le caractère manquant est tout simplement invisible, c'est un espace blanc.**

Il est situé entre les deux lignes de texte.

Il est noté comme suit : \n.

Est-ce que vous vous en souvenez? C'est un caractère spécial (contrôle) utilisé pour **forcer un saut de ligne** (d'où son nom : LF). Vous ne pouvez pas le voir, mais il compte.

Les chaînes multilignes peuvent également être délimitées par **des guillemets triples**, comme ici :

```
multiline = """Line #1
Line #2"""

print(len(multiline))
```

Choisissez la méthode qui vous convient le mieux. Les deux fonctionnent de la même manière.

### 11.1.3 Opérations sur les chaînes

Comme d'autres types de données, les chaînes ont leur propre ensemble d'opérations autorisées, bien qu'elles soient plutôt limitées par rapport aux nombres.

En général, les chaînes peuvent être :

- **concaténé** (joint)
- **répliqué**.

La première opération est effectuée par l'opérateur + (note : ce n'est pas une addition) tandis que la seconde par l'opérateur \* (remarque encore : ce n'est pas une multiplication).

La possibilité d'utiliser le même opérateur avec des types de données complètement différents (comme des nombres ou des chaînes) est appelée **surcharge** (car un opérateur est surchargé de tâches différentes, on verra ça au Q2).

Analysez l'exemple :

```
str1 = 'a'
str2 = 'b'
print(str1 + str2)
print(str2 + str1)
print(5 * 'a')
print('b' * 4)
```

- L'opérateur + utilisé entre deux chaînes ou plus produit une nouvelle chaîne contenant tous les caractères de ses arguments (note: l'ordre compte contrairement à sa version numérique, il **n'est pas commutatif**)
- L'opérateur \* a besoin d'une chaîne et d'un nombre comme arguments ; dans ce cas, l'ordre n'a pas d'importance, vous pouvez mettre le nombre avant la chaîne, ou vice versa, le résultat sera le même.

L'extrait produit la sortie suivante :

```
ab
ba
aaaaa
bbbb
```

Remarque : les variantes de raccourci des opérateurs ci-dessus sont également applicables aux chaînes (+ = et \* =).

#### 11.1.4 Opérations sur les chaînes : ord()

Si vous **souhaitez connaître la valeur du point de code ASCII/UNICODE d'un caractère spécifique**, vous pouvez utiliser une fonction nommée **ord()** (comme dans *ordinal*).

La fonction a besoin d'une **chaîne d'un caractère comme** argument, la violation de cette exigence provoque une exception `TypeError` et renvoie un nombre représentant le point de code de l'argument.

Examinez le code et exécutez-le.

```
char_1 = 'a'
char_2 = ' ' # space

print(ord(char_1))
print(ord(char_2))
```

L'extrait de code génère :

97

32

Attribuez maintenant différentes valeurs à `char_1` et `char_2`, par exemple,  $\alpha$  (alpha grec) et  $\text{ę}$  (une lettre de l'alphabet polonais); puis exécutez le code et voyez quel résultat il obtient. Réalisez vos propres expériences.

#### 11.1.5 Opérations sur les chaînes : chr()

Si vous connaissez la valeur en codage (nombre) et que vous souhaitez obtenir le caractère correspondant, vous pouvez utiliser une fonction nommée `chr()`.

La fonction **prend une valeur de codage et renvoie son caractère**.

L'invoquer avec un argument non valide (par exemple, une valeur de codage négatif ou non valide) provoque des exceptions `ValueError` ou `TypeError`.

Exécutez le code dans l'éditeur.

```
print(chr(97))
print(chr(945))
```

L'exemple d'extrait de code génère :

a  
ą



### 11.1.6 Chaînes en tant que séquences: indexation

Nous vous avons déjà dit que **les chaînes Python sont des séquences**. Il est temps de vous montrer ce que cela signifie réellement.

Les chaînes ne sont pas des listes, mais **vous pouvez les traiter comme des listes dans de nombreux cas particuliers**.

Par exemple, si vous souhaitez accéder à l'un des caractères d'une chaîne, vous pouvez le faire à l'aide de **l'indexation**, comme dans l'exemple ci-dessous. Exécutez le programme :

```
the_string = 'petite promenade'

for ix in range(len(the_string)):
    print(the_string[ix], end=' ')

print()
```

Soyez prudent, n'essayez pas de dépasser les limites d'une chaîne, cela provoquera une exception. Il est possible d'utiliser des indices négatifs.

La sortie de l'exemple donnera : p e t i t e p r o m e n a d e

### 11.1.7 Chaînes en tant que séquences: itération

**L'itération à travers les chaînes** fonctionne aussi.

Regardez l'exemple ci-dessous :

```
the_string = 'petite promenade'

for character in the_string:
    print(character, end=' ')

print()
```

La sortie est la même que précédemment.

### 11.1.8 Tranches/plages

De plus, tout ce que vous savez sur **les plages** est toujours utilisable. Nous avons rassemblé quelques exemples montrant comment fonctionnent les plages dans le monde des string. Examinez le code et analysez-le.

```
alpha = "abdefg"

print(alpha[1:3])
print(alpha[3:])
print(alpha[:3])
print(alpha[3:-2])
print(alpha[-3:4])
print(alpha[:2])
print(alpha[1::2])
```

Vous ne verrez rien de nouveau dans l'exemple, mais nous voulons que vous soyez sûr que vous pouvez expliquer toutes les lignes du code.

La sortie du code est la suivante :

```
bd
efg
abd
e
e
adf
beg
```

#### 11.1.9 Les opérateurs `in` et `not in`

##### *L'opérateur `in`*

L'opérateur **`in`** ne devrait pas vous surprendre lorsqu'il est appliqué à des chaînes, il vérifie simplement si son argument gauche (une chaîne) peut être trouvé n'importe où dans l'argument de droite (une autre chaîne).

Le résultat de la vérification est simplement `True` ou `False`.

Regardez l'exemple de programme ci-dessous:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" in alphabet)
print("F" in alphabet)
print("l" in alphabet)
print("ghi" in alphabet)
print("Xyz" in alphabet)
```

L'exemple donne :

```
True
False
False
True
```

```
False
```

*L'opérateur not in*

Comme vous vous en doutez probablement, l'opérateur **not in** est également applicable ici.

Voici comment cela fonctionne :

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" not in alphabet)
print("F" not in alphabet)
print("1" not in alphabet)
print("ghi" not in alphabet)
print("Xyz" not in alphabet)
```

L'exemple de sortie est :

```
False
True
True
False
True
```

### 11.1.10 Les chaînes Python sont immuables

Nous vous avons également dit que **les chaînes de Python sont immuables**. C'est une caractéristique très importante. Qu'est-ce que cela signifie? Cela signifie principalement que les similitudes des chaînes et des listes est limitée. Tout ce que vous pouvez faire avec une liste ne peut pas être fait avec une chaîne.

La première différence importante, **vous ne pouvez pas utiliser l'instruction del pour supprimer quoi que ce soit d'une chaîne**.

L'exemple ne fonctionnera pas :

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
del alphabet[0]
```

La seule chose que vous pouvez faire avec del et une chaîne est de **supprimer la chaîne dans son ensemble**.

Les chaînes Python **n'ont pas la** méthode append(), vous ne pouvez en aucun cas l'utiliser.

L'exemple ci-dessous est erroné :

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet.append("A")
```

**La** méthode insert() **est également illégale** :

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet.insert(0, "A")
```

Ne pensez pas que l'immuabilité d'une chaîne se limite à votre capacité à faire des opérations sur les chaînes de caractères.

La seule conséquence, est que vous devez vous en souvenir et implémenter votre code d'une manière légèrement différente, regardez l'exemple de code :

```
alphabet = "bcdefghijklmnopqrstuvwxy"

alphabet = "a" + alphabet
alphabet = alphabet + "z"

print(alphabet)
```

Cette forme de code est tout à fait acceptable, fonctionnera sans enfreindre les règles de Python et apportera l'alphabet latin complet à votre écran.

Vous vous demandez si la **création d'une nouvelle copie d'une chaîne à chaque fois que vous modifiez son contenu aggrave l'efficacité du code**. Et bien oui c'est le cas. Mais ce n'est pas un problème.

## 11.1.11 Opérations sur chaînes : min()

Maintenant que vous comprenez que les chaînes sont des séquences, nous pouvons vous montrer des capacités que possède les séquences. Nous les présenterons à l'aide de chaînes, mais n'oubliez pas que les listes peuvent également adopter les mêmes astuces (et oui ce sont des séquences aussi).

Commençons par une fonction nommée **min()**.

La fonction **trouve l'élément minimum de la séquence passée en argument**. Il y a une condition, la séquence (chaîne, liste, peu importe) **ne peut pas être vide**, sinon vous obtiendrez une exception **ValueError**.

```
print(min("aAbByYzZ"))
```

Les extraits du programme **de l'exemple** sont les suivants :

A

Remarque : Il s'agit d'un A majuscule. Pourquoi? Rappelez-vous la table ASCII vu en théorie, quelles lettres occupent les premiers emplacements?

Nous avons préparé deux autres exemples à analyser :

```
t = 'The Knights Who Say "Ni!'"
print('[' + min(t) + ']')
```

```
t = [0, 1, 2]
print(min(t))
```

Comme vous pouvez le constater, ils présentent plus que de simples string. Le résultat attendu est le suivant :

```
[ ]
0
```

Remarque: nous avons utilisé les crochets pour éviter que l'espace ne soit négligé sur votre écran.

## 11.1.12 Opérations sur les chaînes : max()

De même, une fonction nommée **max()** trouve l'élément maximum de la séquence.

Regardez l'exemple :

```
print(max("aAbByYzZ"))
```

Le résultat : z

Voyons maintenant la fonction max() appliquée aux mêmes données que précédemment. Regardez les exemples suivants :

```
t = 'The Knights Who Say "Ni!'"
print('[' + max(t) + ']')
```

```
t = [0, 1, 2]
print(max(t))
```

Le résultat attendu est :

```
[y]
2
```

## 11.1.13 Opérations sur les chaînes : méthode index()

La méthode **index()** (c'est une méthode, pas une fonction, on en reparlera) recherche la séquence depuis le début, afin de trouver le premier élément de la valeur spécifiée dans son argument.

Remarque : l'élément recherché doit se trouver dans la séquence, son absence entraînera une exception **ValueError**.

La méthode renvoie l'index de la première occurrence de l'argument (ce qui signifie que le résultat le plus bas possible est 0, tandis que le plus élevé est la longueur de l'argument décrétementée de 1).

```
print("aAbByYzZaA".index("b"))
print("aAbByYzZaA".index("z"))
print("aAbByYzZaA".index("A"))
```

Par conséquent, l'exemple dans l'éditeur génère :

```
2
7
1
```

## 11.1.14 Opérations sur les chaînes : fonction list()

La fonction **list()** prend en argument (une chaîne) et crée une nouvelle liste contenant tous les caractères de la chaîne, un par élément de liste.

Remarque: ce n'est pas strictement une fonction de chaîne, list() est capable de créer une nouvelle liste à partir de nombreuses autres entités (par exemple, à partir de tuples et de dictionnaires).

Jetez un coup d'œil :

```
print(list("abcabc"))
```

L'exemple produit : ['a', 'b', 'c', 'a', 'b', 'c']

## 11.1.15 Opérations sur les chaînes : méthode count()

La méthode **count()** compte toutes les occurrences de l'élément à l'intérieur de la séquence. L'absence de tels éléments ne pose aucun problème.

Regardez :

```
print("abcabc".count("b"))
print('abcabc'.count("d"))
```

Résultat :

2  
0

## 11.1.16 Résumé

1. Les chaînes Python sont **des séquences immuables** et peuvent être indexées, découpées et itérées comme n'importe quelle autre séquence, tout en étant soumises aux opérateurs in et not in. Il existe deux types de chaînes en Python :

- **chaînes** d'une ligne, qui ne peuvent pas franchir les limites de ligne, nous les désignons en utilisant des apostrophes ('string') ou des guillemets ("string")
- ☐ Chaînes multilignes, qui occupent plus d'une ligne de code source, délimitées par des trigraphes :

```
'''
string
'''

Ou
"""
string
"""
```

2. La longueur d'une chaîne est déterminée par la fonction **len()**. Le caractère d'échappement (\) n'est pas compté. Par exemple:

```
print(len("\n\n"))
```

3. Les chaînes peuvent être **concaténées** à l'aide de l'opérateur + et **répliquées** à l'aide de l'opérateur \* . Par exemple:

```
asterisk = '*'
plus = "+"
decoration = (asterisk + plus) * 4 + asterisk
print(decoration)
```

4. La paire de fonctions chr() et ord() peut être utilisée pour créer un caractère à l'aide de son codage, et pour déterminer un point de code correspondant à un caractère. Les deux expressions suivantes sont toujours vraies :

```
chr(ord(character)) == character
ord(chr(codepoint)) == codepoint
```

5. Voici d'autres fonctions qui peuvent être appliquées aux chaînes :

- list() : créer une liste composée de tous les caractères de la chaîne ;
- max() : trouve le caractère avec le point de code maximal ;
- min() : recherche le caractère avec le point de code minimal.

6. La méthode nommée index() trouve l'index d'une sous-chaîne donnée à l'intérieur de la chaîne.

### Exercice 1

Quelle est la longueur de la chaîne suivante en supposant qu'il n'y a pas d'espaces entre les guillemets ?

```
"""
"""
```

### Exercice 2

Quel est le résultat attendu du code suivant ?

```
s = 'yesteryears'
the_list = list(s)
print(the_list[3:6])
```

### Exercice 3

Quel est le résultat attendu du code suivant ?

```
for ch in "abc":
    print(chr(ord(ch) + 1), end='')
```



## 11.2 Les méthodes des string

### 11.2.1 La méthode capitalize()

Passons en revue quelques méthodes de chaîne standard. Nous allons les passer en revue par ordre alphabétique, pour être honnête, tout ordre a autant d'inconvénients que d'avantages, donc le choix peut aussi bien être aléatoire.

La méthode **capitalize()** fait exactement ce qu'elle dit, elle **crée une nouvelle chaîne remplie de caractères tirés de la chaîne source**, mais elle essaie de les modifier de la manière suivante:

- **Si le premier caractère à l'intérieur de la chaîne est une lettre** (remarque : le premier caractère est un élément avec un index égal à 0, pas seulement le premier caractère visible), **il sera converti en majuscules ;**
- **Toutes les lettres restantes de la chaîne seront converties en minuscules.**

N'oubliez pas que :

- la chaîne d'origine (à partir de laquelle la méthode est invoquée) n'est en aucun cas modifiée (l'immutabilité d'une chaîne doit être respectée sans réserve)
- La chaîne modifiée (en majuscule dans ce cas) est renvoyée en conséquence, si vous ne l'utilisez pas (l'affecter à une variable ou la passer à une fonction / méthode), elle disparaîtra sans laisser de trace.

Remarque : les méthodes ne doivent pas être invoquées à partir de variables uniquement. Elles peuvent être invoquées directement à partir de littéraux de chaîne. Nous allons utiliser cette convention régulièrement, cela simplifiera les exemples, car les aspects les plus importants ne disparaîtront pas parmi les affectations inutiles.

Jetez un coup d'œil à l'exemple :

```
print('aBcD'.capitalize())
```

Voici ce qu'il imprime:

Abcd

Essayez des exemples plus avancés et testez leur résultat

```
print("Alpha".capitalize())
print('ALPHA'.capitalize())
print(' Alpha'.capitalize())
print('123'.capitalize())
print("αβγδ".capitalize())
```

### 11.2.2 La méthode center()

La méthode center() effectue une copie de la chaîne d'origine, en essayant de la centrer dans un champ d'une largeur spécifiée.

Le centrage se fait en **fait en ajoutant quelques espaces avant et après la chaîne**.

Ne vous attendez pas à ce que cette méthode démontre des compétences sophistiquées. C'est assez simple.

```
print([' ' + 'alpha'.center(10) + ' '])
```

Sa sortie se présente comme suit : [ alpha ]

Si la longueur du champ cible est trop petite pour tenir la chaîne, la chaîne d'origine est renvoyée.

Vous pouvez voir la méthode center() dans plus d'exemples :

```
print([' ' + 'Beta'.center(2) + ' '])
print([' ' + 'Beta'.center(4) + ' '])
print([' ' + 'Beta'.center(6) + ' '])
```

**La variante à deux paramètres de center() utilise le caractère du deuxième argument, au lieu d'un espace.** Analysez l'exemple ci-dessous :

```
print([' ' + 'gamma'.center(20, '*') + ' '])
```

La sortie ressemble maintenant à ceci:

```
[*****gamma*****]
```

### 11.2.3 La méthode endswith()

La méthode **endswith()** vérifie si la chaîne donnée se termine par l'argument spécifié et renvoie True ou False, selon le résultat de la vérification.

Remarque: la sous-chaîne doit respecter le dernier caractère de la chaîne, il ne peut pas simplement être située quelque part près de la fin de la chaîne.

```
if "epsilon".endswith("on"):
    print("yes")
else:
    print("no")
```

### 11.2.4 La méthode find()

La méthode find() est similaire à index(), que vous connaissez déjà, **elle recherche une sous-chaîne et renvoie l'index de la première occurrence de cette sous-chaîne**, mais :

- c'est plus sûr, il **ne génère pas d'erreur pour un argument contenant une sous-chaîne inexistante** (il retourne -1 alors)
- Il **ne fonctionne qu'avec des chaînes**, n'essayez pas de l'appliquer à une autre séquence.

```
print("Eta".find("ta"))  
print("Eta".find("mma"))
```

le résultat sera :

```
1  
-1
```

Remarque: n'utilisez pas find() si vous voulez seulement vérifier si un seul caractère apparaît dans une chaîne, l'opérateur in sera nettement plus rapide.

Voici un autre exemple :

```
t = 'theta'  
print(t.find('eta'))  
print(t.find('et'))  
print(t.find('the'))  
print(t.find('ha'))
```

Si vous souhaitez effectuer la recherche, non pas à partir du début de la chaîne, mais **à partir de n'importe quelle position**, vous pouvez utiliser une **variante à deux paramètres** de la méthode find(). Regardez l'exemple :

```
print('kappa'.find('a', 2))
```

Le deuxième argument **spécifie l'index auquel la recherche sera lancée** (il n'est pas nécessaire qu'il soit dans la chaîne).

Parmi les deux lettres *a*, on ne trouvera que la seconde dans notre cas.

Vous pouvez utiliser la méthode find() pour rechercher toutes les occurrences de la sous-chaîne, comme ici :

```
the_text = """A variation of the ordinary lorem ipsum  
text has been used in typesetting since the 1960s  
or earlier, when it was popularized by advertisements  
for Letraset transfer sheets. It was introduced to  
the Information Age in the mid-1980s by the Aldus Corporation,  
which employed it in graphics and word-processing templates  
for its desktop publishing program PageMaker (from Wikipedia)"""  
  
fnd = the_text.find('the')  
while fnd != -1:  
    print(fnd)
```

```
fnd = the_text.find('the', fnd + 1)
```

Le code imprime les index de toutes les occurrences de l'article « *the* », et sa sortie ressemble à ceci:

```
15
80
198
221
238
```

Il existe également une **variante à trois paramètres** de la **méthode** `find()`, le troisième argument **pointe vers le premier index qui ne sera pas pris en considération lors de** la recherche (c'est en fait la limite supérieure de la recherche).

Regardez notre exemple ci-dessous:

```
print('kappa'.find('a', 1, 4))
print('kappa'.find('a', 2, 4))
```

Le deuxième argument spécifie l'index auquel la recherche sera lancée.

Par conséquent, l'exemple modifié génère :

```
1
-1
```

#### 11.2.5 La méthode `isalnum()`

La méthode sans paramètre nommée **`isalnum()`** vérifie si la chaîne contient uniquement des chiffres ou des caractères alphabétiques (lettres) et renvoie la valeur `True` ou `False` en fonction du résultat.

```
print('lambda30'.isalnum())
print('lambda'.isalnum())
print('30'.isalnum())
print('@'.isalnum())
print('lambda_30'.isalnum())
print('').isalnum())
```

Remarque : tout élément de chaîne qui n'est pas un chiffre ou une lettre fait renvoyer `False` à la méthode. Une chaîne vide le fait aussi.

La sortie est :

```
True
True
True
False
False
False
```

Trois autres exemples :

```
t = 'Six lambdas'  
print(t.isalnum())
```

```
t = 'AβΓδ'  
print(t.isalnum())
```

```
t = '20E1'  
print(t.isalnum())
```

Alors ? Quel est votre opinion ?

#### 11.2.6 La méthode isalpha()

La méthode isalpha() est plus spécialisée, elle ne s'intéresse qu'aux lettres.

```
print("Moooo".isalpha())  
print('Mu40'.isalpha())
```

Regardez l'exemple :

Vrai  
Faux

#### 11.2.7 La méthode isdigit()

À son tour, la méthode isdigit() ne regarde que les chiffres, tout le reste produit False comme résultat.

```
print('2023'.isdigit())  
print("Year2023".isdigit())
```

Regardez l'exemple :

Vrai  
Faux

#### 11.2.8 La méthode islower()

La méthode islower() est une variante pointilleuse de isalpha(), elle n'accepte **que les lettres minuscules**.

```
print("Moooo".islower())  
print('moooo'.islower())
```

Regardez l'exemple:

Faux  
Vrai

11.2.9 La méthode `isspace()`

La méthode `isspace()` identifie uniquement les espaces blancs, elle ne tient pas compte de tout autre caractère (le résultat est `False` alors).

```
print(' \n '.isspace())
print(" ".isspace())
print("mooo mooo mooo".isspace())
```

Regardez l'exemple 2 dans l'éditeur - le résultat est:

```
True
True
False
```

11.2.10 La méthode `isupper()`

La méthode `isupper()` est la version majuscule de `islower()`, elle se concentre uniquement sur les lettres majuscules.

```
print("Moooo".isupper())
print('moooo'.isupper())
print('MOOOO'.isupper())
```

Résultat :

```
False
False
True
```

11.2.11 La méthode `join()`

La méthode `join()` est assez compliquée, alors laissez-nous vous guider étape par étape:

- comme son nom l'indique, la méthode **effectue une jointure**, elle attend un argument sous forme de liste; il faut s'assurer que tous les éléments de la liste sont des chaînes, la méthode déclenche sinon une exception `TypeError`;
- Tous les éléments de la liste seront **jointés en une seule chaîne** mais...
- ... la chaîne à partir de laquelle la méthode a été appelée est **utilisée comme séparateur**, placée parmi les chaînes ;
- La chaîne nouvellement créée est renvoyée en conséquence.

Jetez un coup d'œil à l'exemple et analysons-le :

- La méthode `join()` est appelée à partir d'une chaîne contenant une virgule (la chaîne peut être arbitrairement longue ou vide)
- L'argument de jointure est une liste contenant trois chaînes ;
- La méthode renvoie une nouvelle chaîne.

```
print(", ".join(["omicron", "pi", "rho"]))
```

Voilà le résultat : `omicron,pi,rh`

### 11.2.12 La méthode lower()

La méthode **lower()** effectue une copie d'une chaîne source et remplace toutes les lettres majuscules par leurs équivalents minuscules, ensuite elle renvoie la chaîne comme résultat. Encore une fois, la chaîne source reste intacte. Si la chaîne ne contient pas de caractères majuscules, la méthode renvoie la chaîne d'origine.

Remarque : La méthode lower() ne prend aucun paramètre.

```
print("PRoFfesseur=DesMET".lower())
```

L'exemple donne :

```
professeur = desmet
```

### 11.2.13 La méthode lstrip()

La méthode **lstrip()** sans paramètre renvoie une chaîne nouvellement créée formée à partir de la chaîne d'origine en supprimant tous les espaces de début de chaîne.

```
print("[ " + " tau " .lstrip() + " ]")
```

Les crochets ne font pas partie du résultat, ils sont là pour montrer les limites du résultat.

L'exemple produit :

```
[tau ]
```

La méthode lstrip() à un paramètre fait la même chose que sa version sans paramètre, mais supprime tous les caractères inscrits dans son argument (une chaîne), pas seulement les espaces :

```
print("www.cisco.com".lstrip("w."))
```

La sortie se présente comme suit:

```
cisco.com
```

Et si je vous demande de tester avec .com au lieu de w. ?

11.2.14 La méthode `replace()`

La méthode `replace()` à **deux paramètres et retourne une copie de la chaîne d'origine dans laquelle toutes les occurrences du premier argument ont été remplacées par le second argument.**

Regardez l'exemple :

```
print("www.heh.be".replace("heh.be", "supercours.org"))
print("This is it!".replace("is", "are"))
print("Jus de pomme".replace("Jus de ", ""))
```

L'exemple produit :

```
www.supercours.org
Thare are it!
Pomme
```

Si le deuxième argument est une chaîne vide, remplacer supprime en fait la chaîne du premier argument. Quel genre de magie se produirait si le premier argument est une chaîne vide ?

La variante `replace()` à **trois paramètres** utilise le troisième argument (un nombre) pour **limiter le nombre de remplacements.**

Regardez l'exemple de code modifié ci-dessous :

```
print("This is it!".replace("is", "are", 1))
print("This is it!".replace("is", "are", 2))
```

Testez ce petit exemple de vous-mêmes.

11.2.15 La méthode `rfind()`

Les méthodes à un, deux et trois paramètres nommées `rfind()` font presque les mêmes choses que leurs homologues (celles dépourvues du préfixe `r`), mais **commencent leurs recherches à partir de la fin de la chaîne**, pas du début (d'où le préfixe *r*, pour *droite en anglais*).

Jetez un coup d'œil à l'exemple de code et essayez de prédire sa sortie. Exécutez le code pour vérifier si vous aviez raison.

```
print("tau tau tau".rfind("ta"))
print("tau tau tau".rfind("ta", 9))
print("tau tau tau".rfind("ta", 3, 9))
```



### 11.2.16 La méthode `rstrip()`

Les deux variantes de la méthode **`rstrip()`** font presque la même chose que **`lstrip()`**, mais **affectent le côté opposé de la chaîne**.

Regardez l'exemple de code dans l'éditeur. Pouvez-vous deviner sa sortie?  
Exécutez le code pour vérifier vos suppositions.

```
print("[ " + " Petitmot ".rstrip() + " ]")  
print("heh.be".rstrip(".be"))
```

Comme d'habitude, nous vous encourageons à expérimenter avec vos propres exemples.

### 11.2.17 La méthode `split()`

La méthode `split()` fait ce qu'elle dit littéralement, elle **divise la chaîne et construit une liste de toutes les sous-chaînes détectées**.

La méthode **suppose que les sous-chaînes sont délimitées par des espaces blancs**, les espaces ne participent pas à l'opération et ne sont pas copiés dans la liste résultante.

Si la chaîne est vide, la liste résultante est également vide.

Regardez le code : `print("phi chi\npsi".split())`

L'exemple produit la sortie suivante :

```
['Phi', 'Chi', 'PSI']
```

Remarque: l'opération inverse peut être effectuée par la méthode `join()`.

### 11.2.18 La méthode `startswith()`

La méthode `startswith()` est une réflexion miroir de `endswith()`, elle vérifie si une chaîne donnée commence par la sous-chaîne spécifiée.

```
print("omega".startswith("meg"))  
print("omega".startswith("om"))
```

Le résultat sera `False` puis `True`

### 11.2.19 La méthode `strip()`

La méthode `strip()` combine les effets causés par `rstrip()` et `lstrip()`, elle **crée une nouvelle chaîne dépourvue de tous les espaces de début et de fin**.

```
print("[ " + "   Alaïde   ".strip() + " ]")
```

Le résultat sera `[Alaïde]`

### 11.2.20 La méthode swapcase()

La méthode swapcase() **crée une nouvelle chaîne en permutant la casse de toutes les lettres de la chaîne source** : les caractères minuscules deviennent des majuscules, et vice versa. Tous les autres caractères restent intacts.

```
print("Je sais que Je ne sais rien.".swapcase())
```

Le résultat sera : jE SAIS QUE jE NE SAIS RIEN.

### 11.2.21 La méthode title()

La méthode title() remplit une fonction quelque peu similaire, elle **change la première lettre de chaque mot en majuscules, transformant toutes les autres en minuscules**.

```
print("Je sais que Je ne sais rien.".title())
```

Voici le résultat : Je Sais Que Je Ne Sais Rien.

### 11.2.22 La méthode upper()

Enfin, la méthode upper() **effectue une copie de la chaîne source, remplace toutes les lettres minuscules par leurs équivalents majuscules** et renvoie la chaîne comme résultat.

```
print("Je sais que Je ne sais rien.".upper())
```

Le résultat : JE SAIS QUE JE NE SAIS RIEN.

Hooouuuuraaaa! Nous sommes arrivés à la fin de cette section infernale. Prenez quelques minutes pour examiner toutes ces méthodes.

## 11.2.23 Résumé

1. Certaines des méthodes offertes par les chaînes sont:

`capitalize()` : remplace toutes les lettres de chaîne en majuscules ;  
`center()` : centre la chaîne à l'intérieur du champ d'une longueur connue ;  
`count()` : compte les occurrences d'un caractère donné ;  
`join()` : joint tous les éléments d'un tuple/liste en une seule chaîne ;  
`lower()` : convertit toutes les lettres de la chaîne en lettres minuscules ;  
`lstrip()` : supprime les caractères blancs du début de la chaîne ;  
`replace()` : remplace une sous-chaîne donnée par une autre ;  
`rfind()` : trouve une sous-chaîne commençant à la fin de la chaîne ;  
`rstrip()` : supprime les espaces blancs de fin de chaîne ;  
`split()` : divise la chaîne en une sous-chaîne à l'aide d'un délimiteur donné ;  
`strip()` : supprime les espaces blancs de début et de fin ;  
`swapcase()` : permute la casse des lettres (du bas vers le haut et vice versa)  
`title()` : rend la première lettre de chaque mot majuscule ;  
`upper()` : convertit toutes les lettres de la chaîne en lettres majuscules.

2. Le contenu des chaînes peut être déterminé à l'aide des méthodes suivantes (toutes renvoient des valeurs booléennes) :

`endswith()` : La chaîne se termine-t-elle par une sous-chaîne donnée ?  
`isalnum()` : La chaîne se compose-t-elle uniquement de lettres et de chiffres ?  
`isalpha()` : La chaîne est-elle composée uniquement de lettres ?  
`islower()` : La chaîne se compose-t-elle uniquement de lettres minuscules ?  
`isspace()` : La chaîne se compose-t-elle uniquement d'espaces blancs ?  
`isupper()` : La chaîne se compose-t-elle uniquement de lettres majuscules ?  
`startswith()` : La chaîne commence-t-elle par une sous-chaîne donnée ?

**Exercice 1**

Quel est le résultat attendu du code suivant ?

```
for ch in "abc123XYX":
    if ch.isupper():
        print(ch.lower(), end='')
    elif ch.islower():
        print(ch.upper(), end='')
    else:
        print(ch, end='')
```

**Exercice 2**

Quel est le résultat attendu du code suivant ?

```
s1 = 'Où sont les profs d'antan?'
s2 = s1.split()
print(s2[-2])
```

**Exercice 3**

Quel est le résultat attendu du code suivant ?

```
the_list = ['Où', 'sont', 'les', 'profs', 'd'antan?']  
s = '*'.join(the_list)  
print(s)
```

**Exercice 4**

Quel est le résultat attendu du code suivant ?

```
s = 'Ce cours est tellement facile'  
s = s.replace('facile', 'difficile').replace('tellement', '')  
print(s)
```

## 11.2.24 Exercice

## Temps estimé

20-25 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des chaînes de caractères;
- à l'aide de méthodes de chaîne Python intégrées.

## Scénario

Vous savez déjà comment fonctionne `split()`. Maintenant, nous voulons que vous le prouviez.

Votre tâche est **d'écrire votre propre fonction, qui se comporte presque exactement comme la méthode `split()` originale**, c'est-à-dire :

- il devrait accepter exactement un argument - une chaîne;
- il doit renvoyer une liste de mots créés à partir de la chaîne, divisée aux endroits où la chaîne contient des espaces ;
- si la chaîne est vide, la fonction doit renvoyer une liste vide ;
- Son nom devrait être `mysplit()`

Vous pouvez partir de ce modèle :

```
def mysplit(strng):  
    #  
    # Code ici  
    #  
  
print(mysplit("Être ou ne pas être, telle est la question"))  
print(mysplit("Être ou ne pas être,telle est la question "))  
print(mysplit("  "))  
print(mysplit(" abc "))  
print(mysplit(""))
```

## Résultats escomptés

```
['Etre','ou', 'ne', 'pas', 'être', 'telle', 'est', 'la', 'question']  
['Etre','ou', 'ne', 'pas', 'être,telle', 'est', 'la', 'question'] []  
['abc']  
[]
```

### 11.3 Les strings en folie

#### 11.3.1 Comparaison de chaînes

Les chaînes en Python **peuvent être comparées en utilisant le même ensemble d'opérateurs** qui sont utilisés avec les nombres.

Jetez un coup d'œil à ces opérateurs :

- ==
- !=
- >
- >=
- <
- <=

Il y a un « mais », les résultats de telles comparaisons peuvent parfois être un peu surprenants. N'oubliez pas que Python n'est pas conscient (il ne peut en aucun cas l'être) des problèmes linguistiques subtils, il **compare simplement les valeurs des points de code**, caractère par caractère.

Deux chaînes sont égales lorsqu'elles sont composées des mêmes caractères dans le même ordre. De la même manière, deux chaînes ne sont pas égales lorsqu'elles ne sont pas composées des mêmes caractères dans le même ordre.

Les deux comparaisons donnent True comme résultat:

```
'alpha' == 'alpha'  
'alpha' != 'Alpha'
```

La relation finale entre les chaînes est déterminée en **comparant le premier caractère différent dans les deux chaînes** (gardez à l'esprit les points de code ASCII/UNICODE à tout moment que vous avez vu en théorie).

Lorsque vous comparez deux chaînes de longueurs différentes et que la plus courte est identique au début de la plus longue, la **chaîne la plus longue est considérée comme plus grande (pas en termes de longueur !)**.

Tout comme ici: `'alpha' < 'alphabet'`

La relation est vraie.

La comparaison de chaînes est toujours sensible à la casse (**les lettres majuscules sont considérées comme inférieures aux minuscules**).

```
'beta' > 'Beta'
```

L'expression est Vraie

Même **si une chaîne ne contient que des chiffres, ce n'est toujours pas un nombre**. Elle est interprétée telle quelle, comme toute autre chaîne régulière, et son aspect numérique (potentiel) n'est en aucun cas pris en considération.

Regardez les exemples :

```
'10' == '010'  
'10' > '010'  
'10' > '8'  
'20' < '8'  
'20' < '80'
```

Ils produisent les résultats suivants :

Faux

Vrai

Faux

Vrai

Vrai

**Comparer des chaînes de caractères à des nombres est généralement une mauvaise idée.**

Les seules comparaisons que vous pouvez effectuer en toute impunité sont celles symbolisées par les opérateurs `==` et `!=`. Le premier donne toujours `False`, tandis que le second produit toujours `True`.

L'utilisation de l'un des opérateurs de comparaison restants déclenchera une exception `TypeError`.

Vérifions:

```
'10' == 10  
'10' != 10  
'10' == 1  
'10' != 1  
'10' > 10
```

Les résultats dans ces cas sont les suivants :

Faux

Vrai

Faux

Vrai

Exception `TypeError`

### 11.3.2 Classement

La comparaison est étroitement liée au tri (ou plutôt, le tri est en fait un cas très sophistiqué de comparaison, vous vous rappelez des tris ?).

C'est une bonne occasion de vous montrer deux façons possibles de **trier les listes contenant des chaînes**. Une telle opération est très courante dans le monde réel, chaque fois que vous voyez une liste de noms, de biens, de titres ou de villes, vous vous attendez à ce qu'ils soient triés.

Supposons que vous souhaitiez trier la liste suivante :

```
greek = ['omega', 'alpha', 'pi', 'gamma']
```

En général, Python offre deux façons différentes de trier les listes.

La première est implémentée sous **la forme d'une fonction nommée** `sorted()`. La fonction prend un argument (une liste) et **retourne une nouvelle liste**, remplie avec les éléments de l'argument trié. (Remarque: cette description est un peu simplifiée par rapport à l'implémentation réelle.)

La liste originale reste elle, inchangée.

```
first_greek = ['omega', 'alpha', 'pi', 'gamma']
first_greek_2 = sorted(first_greek)

print(first_greek)
print(first_greek_2)
```

Examinez le code et exécutez-le. L'extrait produit la sortie suivante :

```
['omega', 'alpha', 'pi', 'gamma']

['alpha', 'gamma', 'omega', 'pi']
```

La deuxième méthode affecte la liste elle-même, **aucune nouvelle liste n'est créée**. L'ordre est effectué in situ par la méthode nommée `sort()`.

```
second_greek = ['omega', 'alpha', 'pi', 'gamma']
print(second_greek)

second_greek.sort()
print(second_greek)
```

La sortie n'a pas change mais nous n'avons plus qu'une liste :

```
['omega', 'alpha', 'pi', 'gamma']

['alpha', 'gamma', 'omega', 'pi']
```

Si vous avez besoin d'un ordre autre que croissant, vous devez convaincre la fonction/méthode de changer ses comportements par défaut. Nous en discuterons bientôt.



#### 11.3.4 Chaînes et nombres

Il y a deux questions supplémentaires qui devraient être discutées ici: **comment convertir un nombre (un entier ou un flottant) en une chaîne, et vice versa**. Il peut être nécessaire d'effectuer une telle transformation. De plus, c'est un moyen de traiter les données d'entrée / sortie.

La conversion numération-chaîne est simple, car elle est toujours possible. Cela se fait par une fonction nommée `str()`.

Tout comme ici:

```
itg = 13
flt = 1.3
si = str(itg)
sf = str(flt)

print(si + ' ' + sf)
```

Le code produit :

13 1.3

La transformation inverse (nombre-chaîne) est possible lorsque et seulement lorsque la chaîne représente un nombre valide. Si la condition n'est pas remplie, attendez-vous à une exception `ValueError`.

Utilisez la fonction `int()` si vous souhaitez obtenir un entier et `float()` si vous avez besoin d'une valeur à virgule flottante.

Tout comme ici:

```
si = '13'
sf = '1.3'
itg = int(si)
flt = float(sf)

print(itg + flt)
```

Voici ce que vous verrez dans la console :

14.3

### 11.3.5 Principaux points à retenir

1. Les chaînes peuvent être comparées à des chaînes à l'aide d'opérateurs de comparaison généraux, mais les comparer à des nombres ne donne aucun résultat raisonnable, car **aucune chaîne ne peut être égale** à un nombre. Par exemple:

- `string == number` is always False;
- `string != number` is always True;
- `string >= number` **déclenche toujours une exception**.

2. Le tri des listes de chaînes peut être effectué par:

- une fonction nommée `sorted()`, créant une nouvelle liste triée ;
- une méthode nommée `sort()`, qui trie la liste in situ

3. Un nombre peut être converti en chaîne à l'aide de la fonction `str()`.

4. Une chaîne peut être convertie en nombre (mais pas toutes les chaînes) à l'aide de la fonction `int()` ou `float()`. La conversion échoue si une chaîne ne contient pas d'image numérique valide (une exception est alors déclenchée).

#### Exercice 1

Quelles lignes décrivent une **condition réelle**?

```
'smith' > 'Smith'  
'Smiths' < 'Smith'  
'Smith' > '1000'  
'11' < '8'
```

#### Exercice 2

Quel est le résultat attendu du code suivant ?

```
s1 = 'Ou était l'étudiant hier ?'  
s2 = s1.split()  
s3 = sorted(s2)  
print(s3[1])
```

#### Exercice 3

Quel est le résultat attendu du code suivant ?

```
s1 = '12.8'  
i = int(s1)  
s2 = str(i)  
f = float(s2)  
print(s1 == s2)
```

## 11.3.6 Exercice

## Temps estimé

30 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des strings;
- Utilisation de chaînes pour représenter des données non textuelles.

## Scénario

Vous avez sûrement vu un *affichage à sept segments* chez Mr Arnaud.

C'est un dispositif (parfois électronique, parfois mécanique) conçu pour présenter un chiffre décimal en utilisant un sous-ensemble de sept segments.

Votre tâche consiste à écrire un **programme capable de simuler le travail d'un appareil à sept segments**, bien que vous utilisiez des LED simples au lieu de segments.

Chaque chiffre est construit à partir de 13 LED (certaines allumées, d'autres sombres, bien sûr), c'est ainsi que nous l'imaginons:

#	###	###	#	#	###	###	###	###	###	###	###	###
#		#		#	#	#		#	#	#	#	#
#	###	###	###	###	###	###	#	###	###	#	#	
#	#		#		#	#	#	#	#		#	#
#	###	###	#	###	###		#	###	###	###	###	

Remarque: le chiffre 8 montre toutes les lumières LED allumées.

Votre code doit *afficher* tout nombre entier non négatif entré par l'utilisateur.

Astuce : l'utilisation d'une liste contenant des modèles des dix chiffres peut être très utile.

## Données d'essai

Exemple d'entrée :  
123

Exemple de sortie :

#	###	###
#		#
#	###	###
#	#	
#	###	###

Exemple d'entrée :

9081726354

Exemple de sortie :

###	###	###	#	###	###	###	###	###	#	#
#	#	#	#	#	#	#	#	#	#	#
###	#	#	###	#	#	###	###	###	###	###
#	#	#	#	#	#	#	#	#	#	#
###	###	###	#	#	###	###	###	###	#	#

```
chiffres = [ '1111110', # 0
             '0110000', # 1
             '1101101', # 2
             '1111001', # 3
             '0110011', # 4
             '1011011', # 5
             '1011111', # 6
             '1110000', # 7
             '1111111', # 8
             '1111011', # 9
             ]
```

## 11.4 Quelques petits projets

### 11.4.1 Le chiffre de César : chiffrer un message

Nous allons vous montrer quatre programmes simples afin de présenter certains aspects du traitement des chaînes en Python. Ils sont délibérément simples, mais les exercices seront plus compliqués.

Le premier problème que nous voulons vous montrer s'appelle le **chiffre de César** - plus de détails ici: [https://en.wikipedia.org/wiki/Caesar\\_cipher](https://en.wikipedia.org/wiki/Caesar_cipher).

Ce chiffre a (probablement) été inventé et utilisé par Gaius Julius Caesar et ses troupes pendant la guerre des Gaules. L'idée est assez simple - chaque lettre du message est remplacée par sa conséquence la plus proche (A devient B, B devient C, etc.). La seule exception est Z, qui devient A.

Le programme ci-dessous est une implémentation très simple (mais fonctionnelle) de l'algorithme.

```
text = input("Enter your message: ")
cipher = ''
for char in text:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) + 1
    if code > ord('Z'):
        code = ord('A')
    cipher += chr(code)

print(cipher)
```

Nous l'avons écrit en utilisant les hypothèses suivantes:

- il n'accepte que les lettres latines (note: les Romains n'utilisaient ni espaces ni chiffres)
- toutes les lettres du message sont en majuscules (note: les Romains ne connaissaient que les lettres capitales)

Analysons le code:

- ligne 02 : demande à l'utilisateur le message (non chiffré), d'une ligne;
- Ligne 03 : Préparer une chaîne pour un message chiffré (vide pour l'instant)
- ligne 04 : commencez l'itération à travers le message ;
- Ligne 05 : si le caractère actuel n'est pas alphabétique...
- Ligne 06: ... l'ignorer;
- Ligne 07 : convertir la lettre en majuscules (il est préférable de le faire aveuglément, plutôt que de vérifier si c'est nécessaire ou non)
- ligne 08: obtenez le code de la lettre et incrémentez-le d'un;
- ligne 09 : si le code résultant a « quitté » l'alphabet latin (supérieur au Z)...
- Ligne 10: ... remplacez-le par le code A ;
- ligne 11 : ajoutez le caractère reçu à la fin du message chiffré ;
- Ligne 13 : Imprimez le chiffrement.

Le code, alimenté avec ce message :

AVE CAESAR

Donnera :  
BWFDBFTBS

#### 11.4.2 Le Chiffrement César : décrypter un message

La transformation inverse devrait maintenant être claire pour vous, nous allons simplement vous présenter le code tel quel, sans aucune explication.

Regardez le code ci-dessous. Vérifiez soigneusement si cela fonctionne. Utilisez le cryptogramme du programme précédent.

```
cipher = input('Enter your cryptogram: ')
text = ''
for char in cipher:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) - 1
    if code < ord('A'):
        code = ord('Z')
    text += chr(code)

print(text)
```

### 11.4.3 Le processeur de nombres

Le troisième programme montre une méthode simple vous permettant d'entrer une ligne remplie de nombres et de les traiter facilement.

Remarque : la fonction `input()` de routine, combinée avec les fonctions `int()` ou `float()`, n'est pas adaptée à cette fin.

Le traitement sera extrêmement facile, nous voulons que les chiffres soient additionnés.

Regardez le code et Analysons-le.

```
line = input("Enter a line of numbers-separate them with  
spaces: ")  
strings = line.split()  
total = 0  
try:  
    for substr in strings:  
        total += float(substr)  
    print("The total is:", total)  
except:  
    print(substr, "is not a number.")
```

L'utilisation de vos connaissances sur les listes peut rendre le code plus mince. Vous pouvez le faire si vous voulez.

Présentons notre version :

- Ligne 03 : demandez à l'utilisateur d'entrer une ligne remplie d'un nombre quelconque de nombres (les nombres peuvent être flottants)
- ligne 04: diviser la ligne recevant une liste de sous-chaînes;
- ligne 05 : initier la somme totale à zéro ;
- Ligne 06 : comme la conversion string-float peut soulever une exception, il est préférable de continuer avec la protection du bloc try-except ;
- Ligne 07 : Parcourir la liste...
- Ligne 08: ... et essayer de convertir tous ses éléments en nombres flottants; si cela fonctionne, augmentez la somme;
- ligne 09: tout va bien jusqu'à présent, alors imprimez la somme;
- ligne 10 : le programme se termine ici en cas d'erreur ;
- Ligne 11 : Imprimez un message de diagnostic indiquant à l'utilisateur la raison de l'échec.

Le code a une faiblesse importante: il affiche un faux résultat lorsque l'utilisateur entre une ligne vide. Pouvez-vous le réparer?

#### 11.4.4 Le validateur IBAN

Le quatrième programme implémente (sous une forme légèrement simplifiée) un algorithme utilisé par les banques européennes pour spécifier les numéros de compte. La norme nommée **IBAN** (International Bank Account Number) fournit une méthode simple et assez fiable pour valider les numéros de compte contre les fautes de frappe simples qui peuvent se produire lors de la réécriture du numéro, par exemple, à partir de documents papier, comme des factures.

Vous pouvez trouver plus de détails ici:

[https://en.wikipedia.org/wiki/International\\_Bank\\_Account\\_Number](https://en.wikipedia.org/wiki/International_Bank_Account_Number).

Un numéro de compte compatible IBAN se compose de :

- un code de pays à deux lettres tirées de la norme ISO 3166-1 (par exemple, *BE* pour la Belgique, *GB* pour la Grande-Bretagne, *DE* pour l'Allemagne, ...)
- deux chiffres de contrôle utilisés pour effectuer les contrôles de validité, tests rapides et simples, mais pas entièrement fiables, montrant si un nombre est invalide (faussé par une faute de frappe) ou semble être bon;
- le numéro de compte réel (jusqu'à 30 caractères alphanumériques, la longueur de cette partie dépend du pays)

La norme dit que la validation nécessite les étapes suivantes (selon Wikipedia):

- (étape 1) Vérifiez que la longueur totale de l'IBAN est correcte selon le pays (ce programme ne le fera pas, mais vous pouvez modifier le code pour répondre à cette exigence si vous le souhaitez; note: vous devez enseigner au code toutes les longueurs utilisées en Europe)
- (étape 2) Déplacez les quatre caractères initiaux à la fin de la chaîne (c.-à-d. le code du pays et les chiffres de contrôle)
- (étape 3) Remplacez chaque lettre de la chaîne par deux chiffres, développant ainsi la chaîne, où  $A = 10$ ,  $B = 11 \dots Z = 35$ ;
- (étape 4) Interprétez la chaîne comme un entier décimal et calculez le reste de ce nombre sur la division par 97; Si le reste est égal à 1, le test du chiffre de contrôle est réussi et l'IBAN peut être valide.



Regardez le code et Analysons-le :

```
iban = input("Entrer un IBAN, svp: ")
iban = iban.replace(' ', '')

if not iban.isalnum():
    print("Vous avez entré un caractère non valide.")
elif len(iban) < 15:
    print("IBAN entré est trop court.")
elif len(iban) > 31:
    print("IBAN entré est trop long.")
else:
    iban = (iban[4:] + iban[0:4]).upper()
    iban2 = ''
    for ch in iban:
        if ch.isdigit():
            iban2 += ch
        else:
            iban2 += str(10 + ord(ch) - ord('A'))
    iban = int(iban2)
    if iban % 97 == 1:
        print("IBAN entré est valide.")
    else:
        print("IBAN entré est invalide.")
```

- ligne 03 : demandez à l'utilisateur d'entrer l'IBAN (le numéro peut contenir des espaces, car ils améliorent considérablement la lisibilité des numéros...
- Ligne 04: ... mais supprimez-les immédiatement)
- ligne 05: l'IBAN saisi doit être composé uniquement de chiffres et de lettres, s'il ne le fait pas...
- Ligne 06: ... affichage du message;
- ligne 07 : l'IBAN ne doit pas comporter moins de 15 caractères (c'est la variante la plus courte, utilisée en Norvège)
- ligne 08 : si elle est plus courte, l'utilisateur en est informé ;
- ligne 09: de plus, l'IBAN ne peut pas dépasser 31 caractères (c'est la variante la plus longue, utilisée à Malte)
- ligne 10 : si elle est plus longue, faites une annonce;
- ligne 11 : commencer le traitement proprement dit;
- Ligne 12 : Déplacez les quatre caractères initiaux à la fin du nombre et convertissez toutes les lettres en majuscules (étape 02 de l'algorithme)
- Ligne 13 : Il s'agit de la variable utilisée pour compléter le nombre, créée en remplaçant les lettres par des chiffres (selon l'étape 03 de l'algorithme)
- ligne 14 : itérer à travers l'IBAN ;
- Ligne 15 : si le caractère est un chiffre...
- ligne 16: il suffit de le copier;
- Ligne 17 : sinon...
- Ligne 18: ... convertissez-le en deux chiffres (notez la façon dont c'est fait ici)
- ligne 19: la forme convertie de l'IBAN est prête, faites-en un entier;
- Ligne 20 : Le reste de la division d'iban2 par 97 est-elle égal à 1 ?

- ligne 21 : Si oui, alors succès;
- ligne 22 : Sinon...
- Ligne 23: ... Le numéro n'est pas valide.

Ajoutons quelques données de test (tous ces nombres sont valides - vous pouvez les invalider en changeant n'importe quel caractère).

- Britannique : GB72 HBZU 7006 7212 1253 00
- Français : FR76 30003 03620 00020216907 50
- Allemand : DE02100100100152517108

Vous pouvez utiliser votre propre numéro de compte pour les tests.

#### 11.4.5 Principaux points à retenir

1. Les chaînes sont des outils clés dans le traitement moderne des données, car la plupart des données utiles sont en fait des chaînes. Par exemple, l'utilisation d'un moteur de recherche Web (ce qui semble assez trivial de nos jours) utilise un traitement de chaînes extrêmement complexe, impliquant des quantités inimaginables de données.

2. Comparer les chaînes de manière stricte (comme le fait Python) peut être très insatisfaisant lorsqu'il s'agit de recherches avancées (par exemple lors de requêtes de base de données étendues). En réponse à cette demande, un certain nombre d'algorithmes de comparaison de chaînes *floues* ont été créés et implémentés. Ces algorithmes sont capables de trouver des chaînes qui ne sont pas égales au sens de Python, mais qui sont **similaires**.

L'un de ces concepts est la **distance de Hamming**, qui est utilisée pour déterminer la similitude de deux string. Si ce problème vous intéresse, vous pouvez en savoir plus à ce sujet ici:

[https://en.wikipedia.org/wiki/Hamming\\_distance](https://en.wikipedia.org/wiki/Hamming_distance).

Une autre solution du même genre, mais basée sur une hypothèse différente, est la **distance de Levenshtein** décrite ici:

[https://en.wikipedia.org/wiki/Levenshtein\\_distance](https://en.wikipedia.org/wiki/Levenshtein_distance).

3. Une autre façon de comparer les string est de trouver leur similitude acoustique, ce qui signifie un processus conduisant à déterminer si deux string sonnent similaires (comme « raise » et « race »). Une telle similitude doit être établie pour chaque langue (ou même dialecte) séparément.

Un algorithme utilisé pour effectuer une telle comparaison pour la langue anglaise s'appelle **Soundex** et a été inventé, vous ne le croirez pas, en 1918. Vous pouvez en savoir plus à ce sujet ici: <https://en.wikipedia.org/wiki/Soundex>.

4. En raison de la précision limitée des données flottantes et entières natives, il est parfois raisonnable de stocker et de traiter d'énormes valeurs numériques sous forme de chaînes. C'est la technique utilisée par Python lorsque vous le forcez à fonctionner sur un nombre entier composé d'un très grand nombre de chiffres.

## 14.4.6 Exercices

## 1 Temps estimé

30-90 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des chaînes de caractères;
- conversion des caractères en code ASCII, et vice versa.

## Scénario

Vous connaissez déjà le chiffrement de César, et c'est pourquoi nous voulons que vous amélioriez le code que nous vous avons montré ci-dessus.

Le chiffre original de César décale chaque caractère de un : *a* devient *b*, *z* devient *a*, et ainsi de suite. Rendons les choses un peu plus difficiles et laissons la valeur décalée provenir de la plage 1..25 inclus.

De plus, laissez le code préserver la casse des lettres (les lettres minuscules resteront minuscules) et tous les caractères non alphabétiques doivent rester intacts.

Votre tâche consiste à écrire un programme qui:

- demande à l'utilisateur une ligne de texte à chiffrer;
- demande à l'utilisateur une valeur de décalage (nombre entier de la plage 1..25. Remarque: Vous devez forcer l'utilisateur à entrer une valeur de décalage valide (n'abandonnez pas et ne laissez pas les mauvaises données vous tromper!))
- imprimer le texte codé.
- Tester le code à l'aide des données que nous avons fournies.

## Données d'essai

Exemple d'entrée :  
abcxyzABCxyz 123  
2

Exemple de sortie :  
cdezabCDEzab 123

Exemple d'entrée :  
Les dés sont jetés  
25

Exemple de sortie :  
Sgd chd hr bzrs

## 2. Temps estimé

15-30 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des chaînes de caractères;
- Encourager l'élève à chercher des solutions non évidentes.

## Scénario

Savez-vous ce qu'est un palindrome ?

C'est un mot qui se ressemble lorsqu'il est lu en avant et en arrière. Par exemple, « kayak » est un palindrome, alors que « loyal » ne l'est pas.

Votre tâche consiste à écrire un programme qui:

- demande du texte à l'utilisateur;
- Vérifie si le texte saisi est un palindrome et imprime le résultat.

Note:

- Supposons qu'un string vide n'est pas un palindrome;
- Traiter les lettres majuscules et minuscules comme égales;
- Les espaces ne sont pas pris en compte lors du contrôle, traitez-les comme inexistants;
- Il y a plus de quelques solutions correctes, essayez d'en trouver plus d'une.
- Testez votre code à l'aide des données que nous avons fournies.

## Données d'essai

Exemple d'entrée :

La mariée ira mal

Exemple de sortie :

C'est un palindrome

Exemple d'entrée :

Le marié ira bien

Exemple de sortie :

Ce n'est pas un palindrome

## 3 Temps estimé

30-60 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des chaînes de caractères;
- convertir des chaînes en listes, et vice versa.

## Scénario

Une anagramme est un nouveau mot formé en réarrangeant les lettres d'un mot, en utilisant toutes les lettres originales exactement une fois. Par exemple, les expressions « éternité » et « étreinte » sont des anagrammes.

Votre tâche consiste à écrire un programme qui :

- Demande à l'utilisateur deux textes distincts;
- Vérifie si les textes saisis sont des anagrammes et imprime le résultat.

Note :

- Supposons que deux chaînes vides ne sont pas des anagrammes;
- Traiter les lettres majuscules et minuscules comme égales;
- Les espaces ne sont pas pris en compte lors du contrôle
- Testez votre code à l'aide des données que nous avons fournies.

## Données d'essai

Exemple d'entrée :

Gare  
rage

Exemple de sortie :  
Anagrammes

Exemple d'entrée :  
moderne  
normand

Exemple de sortie :  
Pas d'anagrammes

## 4 Temps estimé

15-30 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des chaînes de caractères;
- convertir des entiers en chaînes, et vice versa.

## Scénario

Certains disent que le chiffre *de la vie* est un chiffre évalué en utilisant l'anniversaire de quelqu'un. C'est simple, il vous suffit d'additionner tous les chiffres de la date. Si le résultat contient plus d'un chiffre, vous devez répéter l'addition jusqu'à ce que vous obteniez exactement un chiffre. Par exemple :

- 1er janvier 2023 = 2023 01 01
- $2 + 0 + 2 + 3 + 0 + 1 + 0 + 1 = 9$
- 

9 est le chiffre que nous avons trouvé.

Votre tâche consiste à écrire un programme qui:

- demande à l'utilisateur son anniversaire (au format AAAAMMJJ, ou AAAAJMM, ou MMJDYYYY - en fait, l'ordre des chiffres n'a pas d'importance)
- génère le *chiffre de vie* pour la date.

## Données d'essai

Exemple d'entrée :  
19991229

Exemple de sortie :  
6

Exemple d'entrée :  
20000101

Exemple de sortie :  
4

## 5 Temps estimé

30-45 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des chaînes de caractères;
- à l'aide de la méthode `find()` pour rechercher des chaînes.

## Scénario

Jouons à un jeu. Nous vous donnerons deux chaînes: l'une étant un mot (par exemple, « chien ») et la seconde étant une combinaison de n'importe quel caractère.

Votre tâche consiste à écrire un programme qui répond à la question suivante : **les caractères composant la première chaîne sont-ils cachés à l'intérieur de la seconde chaîne ?**

Par exemple :

- Si la deuxième chaîne est donnée comme « `vcxzxhduyfdsobiwefngas` », la réponse est oui;
- Si la deuxième chaîne est « `vcxzxdcybfidsthbywuefsnas` », la réponse est non (car il n'y a ni les lettres « c », « h », « i », « e » ou « n », dans cet ordre)

Conseils :

- Vous devez utiliser les variantes à deux arguments des fonctions `pos()` dans votre code ;
- Ne vous inquiétez pas de la sensibilité à la casse.

Testez votre code à l'aide des données que nous avons fournies.

## Données d'essai

Exemple d'entrée :

dons

Nabuchodonosor

Exemple de sortie :

Oui

Exemple d'entrée :

donneur

Nabuchodonosor

Exemple de sortie :

Non

## 6 Temps estimé

60-90 minutes

## Objectifs

- améliorer les compétences de l'élève à utiliser des chaînes et des listes;
- conversion de chaînes en listes.

## Scénario

Comme vous le savez probablement, *le Sudoku* est un puzzle de placement de nombres joué sur un plateau 9x9. Le joueur doit remplir le plateau d'une manière très spécifique:

- Chaque ligne du tableau doit contenir tous les chiffres de 0 à 9 (l'ordre n'a pas d'importance)
- Chaque colonne du tableau doit contenir tous les chiffres de 0 à 9 (encore une fois, l'ordre n'a pas d'importance)
- Chacune des neuf « tuiles » 3x3 (nous les nommerons « sous-carrés ») du tableau doit contenir tous les chiffres de 0 à 9.

Si vous avez besoin de plus de détails, vous pouvez les trouver [ici](#).

Votre tâche consiste à écrire un programme qui:

- lit 9 rangées du Sudoku, chacune contenant 9 chiffres (vérifiez soigneusement si les données saisies sont valides)
- Sorties Oui si le Sudoku est valide, et Non dans le cas contraire.

Testez votre code à l'aide des données que nous avons fournies.

## Données d'essai

Exemple d'entrée : 295743861 431865927 876192543 387459216 612387495 549216738 763524189 928671354 154938672 Exemple de sortie : Oui	Exemple d'entrée : 195743862 431865927 876192543 387459216 612387495 549216738 763524189 928671354 254938671 Exemple de sortie : Non
---	---



## 11.5 Les erreurs, pain quotidien du programmeur

### 11.5.1 Erreurs, échecs et autres fléaux

#### Tout ce qui peut mal tourner ira mal.

C'est la loi de Murphy, et elle fonctionne partout et toujours. L'exécution de votre code peut également mal tourner. Si c'est possible, il le fera.

Regardez le code.

```
import math

x = float(input("Entrer x: "))
y = math.sqrt(x)

print("La racine carrée de ", x, "vaut", y)
```

Il y a au moins deux façons possibles de « mal tourner ». Pouvez-vous les voir?

- Comme un utilisateur est capable d'entrer une chaîne de caractères complètement arbitraire, **il n'y a aucune garantie que la chaîne puisse être convertie en une valeur flottante**, c'est la première vulnérabilité;
- La seconde est que la fonction `sqrt()` **échoue si elle obtient un argument négatif**.

Vous pouvez obtenir l'un des messages d'erreur suivants.

Quelque chose comme ceci:

```
Entrer x: Abracadabra

Traceback (most recent call last):

  File "sqrt.py", line 3, in <module>

    x = float(input("Entrer x: "))

ValueError: could not convert string to float: 'Abracadabra'
```

Ou quelque chose comme ceci:

```
Enter x: -1

Traceback (most recent call last):

  File "sqrt.py", line 4, in <module>

    y = math.sqrt(x)

ValueError: math domain error
```

Pouvez-vous vous protéger de telles surprises? Bien sûr que vous pouvez. De plus, vous devez le faire pour être considéré comme un bon programmeur.

### 11.5.2 Les exceptions

Chaque fois que votre code essaie de faire quelque chose de mal / stupide / irresponsable / fou / inapplicable, Python fait deux choses:

- il **arrête votre programme**;
- Il crée un type spécial de données, appelé **exception**.

Ces deux activités sont appelées **lever une exception**. Nous pouvons dire que Python lève toujours une exception (ou qu'une exception a **été levée**) lorsqu'il n'a aucune idée de ce qu'il faut faire avec votre code.

Que se passe-t-il ensuite?

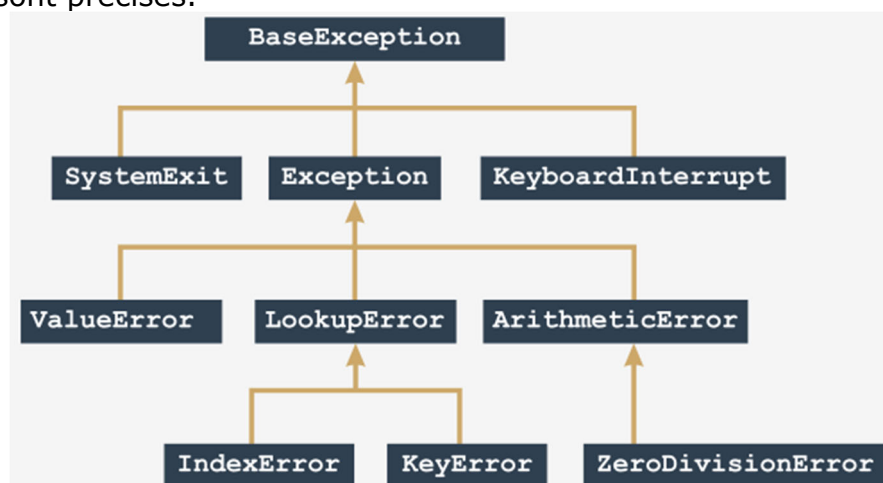
- l'exception soulevée s'attend à ce que quelqu'un ou quelque chose le remarque et s'en occupe;
- si rien ne se passe pour résoudre l'exception soulevée, le programme sera **arrêté de force** et vous verrez un **message d'erreur** envoyé à la console par Python;
- Sinon, si l'exception est prise en charge et **gérée** correctement, le programme suspendu peut être repris et son exécution peut continuer.

Python fournit des outils efficaces qui vous permettent **d'observer les exceptions, de les identifier et de les gérer** efficacement. Cela est possible en raison du fait que toutes les exceptions potentielles ont leurs noms sans ambiguïté, vous pouvez donc les catégoriser et réagir de manière appropriée, Nous en avons vu beaucoup lors du chapitre 9 mais il en reste quelques autres.

### 11.5.3 L'anatomie d'une exception

Python 3 définit **63 exceptions intégrées**, et toutes forment une **hiérarchie en forme d'arbre**, bien que l'arbre soit un peu bizarre car sa racine est située au-dessus.

Certaines des exceptions intégrées sont plus générales (elles incluent d'autres exceptions) tandis que d'autres sont complètement concrètes (elles ne représentent qu'elles-mêmes). Nous pouvons dire que plus **une exception est proche de la racine, plus elle est générale (abstraite)**. À leur tour, les exceptions situées aux extrémités des branches (nous pouvons les appeler **feuilles**) sont précises.

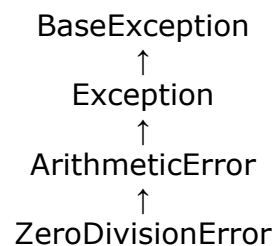


Il montre une petite section de l'arborescence complète des exceptions. Commençons par examiner l'arborescence à partir de la feuille `ZeroDivisionError`.

Note:

- `ZeroDivisionError` est un cas particulier d'une exception générale nommée `ArithmeticError` ;
- `ArithmeticError` est un cas particulier d'une exception plus générale nommée simplement `Exception` ;
- `Exception` est un cas particulier d'une plus générale nommée `BaseException` ;

Nous pouvons le décrire de la manière suivante (notez la direction des flèches, elles pointent toujours vers l'entité plus générale):



Regardez le code. C'est un exemple simple pour commencer. Exécutez-le.

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Oooppsss...")

print("THE END.")
```

La sortie que nous nous attendons à voir ressemble à ceci:

```
Oooppsss...
THE END.
```

Maintenant, regardez le code ci-dessous:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Oooppsss...")

print("THE END.")
```

Quelque chose a changé, nous avons remplacé `ZeroDivisionError` par `ArithmeticError`.

Vous savez déjà que `ArithmeticError` est une classe générale incluant (entre autres) l'exception `ZeroDivisionError`.

Ainsi, la sortie du code reste inchangée. Testez-le.

Cela signifie également que le remplacement du nom de l'exception par `Exception` ou `BaseException` ne changera pas le comportement du programme.

Résumons :

- chaque exception levée **relève de la première branche correspondante**;
- La branche correspondante n'a pas besoin de spécifier exactement la même exception, il suffit que l'exception soit plus **générale** (plus abstraite) que l'exception soulevée.

Regardez le code. Que va-t-il se passer ici?

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Zero Division!")
except ArithmeticError:
    print("Arithmetic problem!")

print("THE END.")
```

La première branche correspondante est celle contenant `ZeroDivisionError`. Cela signifie que la console affichera :

Zero division!  
THE END.

Cela changera-t-il quelque chose si nous échangeons les deux branches? Tout comme ci-dessous:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Arithmetic problem!")
except ZeroDivisionError:
    print("Zero Division!")

print("THE END.")
```

Le changement est radical, la sortie du code est maintenant:

Problème arithmétique!  
LA FIN.

Pourquoi, si l'exception levée est la même que précédemment?

L'exception est la même, mais l'exception plus générale est maintenant répertoriée en premier, elle couvrira également toutes les divisions par zéro. Cela signifie également qu'il n'y a aucune chance qu'une exception atteigne la branche `ZeroDivisionError`. Cette branche est maintenant complètement inaccessible.

Se souvenir que:

- L'ordre des branches est important!
- Ne mettez pas d'exceptions plus générales avant des exceptions plus concrètes;
- Cela rendra ce dernier inaccessible et inutile;
- De plus, cela rendra votre code désordonné et incohérent;
- Python ne générera aucun message d'erreur concernant ce problème.

Si vous souhaitez **gérer deux exceptions ou plus** de la même manière, vous pouvez utiliser la syntaxe suivante :

```
try:
    :
except (excl, exc2):
    :
```

Il vous suffit de mettre tous les noms d'exception engagés dans une liste séparée par des virgules et de ne pas oublier les parenthèses.

Si une **exception est déclenchée à l'intérieur d'une fonction**, elle peut être gérée :

- à l'intérieur de la fonction;
- en dehors de la fonction;

Commençons par la première variante :

```
def bad_fun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("Arithmetic Problem!")
    return None

bad_fun(0)

print("THE END.")
```

L'exception `ZeroDivisionError` (étant un cas concret de la classe d'exception `ArithmeticError`) est levée à l'intérieur de la fonction `bad_fun()`, et elle ne quitte pas la fonction, la fonction elle-même s'en charge.

Les extraits du programme sont les suivants :

```
Arithmetic Problem!  
THE END.
```

Il est également possible de laisser l'exception se propager **en dehors de la fonction**. Testons-le maintenant.

Regardez le code ci-dessous :

```
def bad_fun(n):  
    return 1 / n  
  
try:  
    bad_fun(0)  
except ArithmeticError:  
    print("What happened? An exception was raised!")  
  
print("THE END.")
```

Le problème doit être résolu par l'appelant (ou par l'appelant de l'appelant, etc.).

Les extraits du programme sont les suivants :

```
What happened? An exception was raised!  
THE END.
```

Remarque : l'exception **levée peut traverser les limites de fonction** et de module, et parcourir la chaîne d'appel à la recherche d'une clause d'exception correspondante capable de la gérer.

S'il n'y a pas de telle clause, l'exception reste non gérée et Python résout le problème de sa manière standard, en **mettant fin à votre code et en émettant un message de diagnostic**.

### 15.5.3 Raise

L'instruction **raise** déclenche l'exception spécifiée nommée exc comme si elle était levée de manière normale (naturelle) :

```
raise exc
```

L'instruction vous permet de :

- **simuler** l'augmentation des exceptions réelles (par exemple, pour tester votre stratégie de manipulation)
- gérer partiellement une **exception** et confier à une autre partie du code la responsabilité de terminer le traitement (séparation des préoccupations).

Regardez le code. C'est comme cela que vous pouvez l'utiliser dans la pratique.

```
def bad_fun(n):  
    raise ZeroDivisionError  
  
try:  
    bad_fun(0)  
except ArithmeticError:  
    print("What happened? An error?")  
  
print("THE END.")
```

L'extrait du programme demeure inchangé.

De cette façon, vous pouvez **tester votre routine de gestion des exceptions** sans forcer le code à faire des choses stupides.

Cette instruction peut également être utilisée de la manière suivante (notez l'absence du nom de l'exception) :

```
raise
```

Il y a une restriction sérieuse: ce type d'instruction ne peut être utilisé **qu'à l'intérieur de la branche** except; son utilisation dans tout autre contexte provoque une erreur.  
L'instruction lèvera immédiatement la même exception que celle actuellement traitée.

Grâce à cela, vous pouvez répartir la gestion des exceptions entre différentes parties du code.

Regardez le code. Exécutez-le.

```
def bad_fun(n):  
    try:  
        return n / 0  
    except:  
        print("I did it again!")  
        raise  
  
try:  
    bad_fun(0)  
except ArithmeticError:  
    print("I see!")  
  
print("THE END.")
```

Le ZeroDivisionError est déclenché deux fois :

- Tout d'abord, à l'intérieur de la partie try du code (ceci est causé par la division par zéro)
- Deuxièmement, à l'intérieur de la partie except par l'instruction raise.

En effet, le code produit :

```
I did it again!  
I see!  
THE END.
```

## 15.5.4 assert

C'est maintenant le bon moment pour vous montrer une autre instruction Python, nommée **assert**. C'est un mot-clé évidemment.

```
assert expression
```

Comment fonctionne-t-il ?

- Il évalue l'expression;
- Si l'expression a la valeur **True**, ou une valeur numérique différente de **zéro**, ou une chaîne non vide, ou toute autre valeur différente de **None**, elle ne fera rien d'autre ;
- Sinon, il déclenche automatiquement et immédiatement une exception nommée **AssertionError** (dans ce cas, nous disons que l'assertion a échoué)

Comment peut-il être utilisé?

- Vous voudrez peut-être le mettre dans votre code là où vous voulez être absolument à **l'abri des données manifestement fausses**, et où vous n'êtes pas absolument sûr que les données ont été soigneusement examinées auparavant (par exemple, à l'intérieur d'une fonction utilisée par quelqu'un d'autre)
- Le déclenchement d'une exception **AssertionError** empêche votre code de produire des résultats non valides et indique clairement la nature de l'échec ;
- **Les assertions ne remplacent pas les exceptions et ne valident pas les données**, elles sont leurs compléments.

Si les exceptions et la validation des données sont comme une conduite prudente, les assertions peuvent jouer le rôle d'un airbag.

Voyons l'instruction **assert** en action. Regardez le code. Exécutez-le.

```
import math

x = float(input("Enter a number: "))
assert x >= 0.0

x = math.sqrt(x)

print(x)
```

Le programme fonctionne parfaitement si vous entrez une valeur numérique valide supérieure ou égale à zéro; sinon, il s'arrête et émet le message suivant :

```
Traceback (most recent call last):
  File ".main.py", line 4, in  assert x >= 0.0 AssertionError
```



### 15.5.5 Principaux points à retenir

1. Vous ne pouvez pas ajouter plus d'une branche anonyme (sans nom) après les branches `except`.

```
# Ce code va aussi fonctionner.

try:

    # Code à risques.

except Except_1:
    # Première gestion de crise
except Except_2:
    # Ici nous sauvons le monde
except:
    # Les autres issues sont ici.

# Retour à la normal.
```

2. Toutes les exceptions Python prédéfinies forment une hiérarchie, c'est-à-dire que certaines d'entre elles sont plus générales (celle nommée `BaseException` est la plus générale) tandis que d'autres sont plus ou moins concrètes (par exemple, `IndexError` est plus concret que `LookupError`).

Vous ne devrez pas mettre d'exceptions plus concrètes avant les exceptions plus générales à l'intérieur de la même séquence de branches :

```
try:
    # code à risques.
except IndexError:
    # Prends soins des listes.
except LookupError:
    # Prends soins d'autres erreurs de recherche.
```

mais ne le faites pas (sauf si vous êtes absolument sûr que vous voulez qu'une partie de votre code soit inutile)

```
try:
    # code à risques.
except LookupError:
    # Erreurs de recherche.
except IndexError:
    # Vous n'arriverez jamais ici.
```

3. L'instruction Python `raise ExceptionName` peut déclencher une exception à la demande. La même instruction, mais sans `ExceptionName`, ne peut être utilisée qu'à l'intérieur de la branche `try` et déclenche la même exception que celle actuellement traitée.

4. L'expression `assert` de l'instruction Python évalue l'expression et déclenche l'exception `AssertionError` lorsque l'expression est égale à zéro, une chaîne vide ou `None`. Vous pouvez l'utiliser pour protéger certaines parties critiques de votre code contre les données inadéquates.

**Exercice 1**

Quel est le résultat attendu du code suivant ?

```
try:
    print(1/0)
except ZeroDivisionError:
    print("zero")
except ArithmeticError:
    print("arith")
except:
    print("some")
```

**Exercice 2**

Quel est le résultat attendu du code suivant ?

```
try:
    print(1/0)
except ArithmeticError:
    print("arith")
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```

**Exercice 3**

Quel est le résultat attendu du code suivant ?

```
def foo(x):
    assert x
    return 1/x

try:
    print(foo(0))
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```

## 15.6 Exception utile

### 15.6.1 Exceptions intégrées

Nous allons vous montrer une courte liste des exceptions les plus utiles. Bien qu'il puisse sembler étrange d'appeler « utile » une chose ou un phénomène qui est un signe visible d'échec ou de revers, comme vous le savez, l'erreur est humaine et si quelque chose peut mal tourner, cela ira mal. Les exceptions sont aussi routinières et normales que tout autre aspect de la vie d'un programmeur.

Pour chaque exception, nous vous montrerons :

- son nom;
- son emplacement dans l'arborescence des exceptions ;
- une brève description;
- un extrait de code concis montrant les circonstances dans lesquelles l'exception peut être soulevée.

Il y a beaucoup d'autres exceptions à explorer, nous n'avons tout simplement pas le temps pour les parcourir toutes durant les labos.

#### *ArithmeticError*

**Emplacement :** BaseException ← exception ← ArithmeticError

**Description :** exception abstraite incluant toutes les exceptions causées par des opérations arithmétiques telles que la division par zéro ou le domaine non valide d'un argument.

#### *AssertionError*

**Emplacement :** BaseException ← Exception ← AssertionError

**Description :** exception concrète déclenchée par l'instruction `assert` lorsque son argument est évalué à `False`, `None`, `0` ou une chaîne vide.

#### **Code:**

```
from math import tan, radians
angle = int(input('Enter integral angle in degrees: '))

# We must be sure that angle != 90 + k * 180
assert angle % 180 != 90
print(tan(radians(angle)))
```

#### *BaseException*

**Emplacement :** BaseException

**Description:** la plus générale (abstraite) de toutes les exceptions Python - toutes les autres exceptions sont incluses dans celle-ci; on peut dire que les deux branches suivantes sont équivalentes: `sauf:` et `sauf BaseException:`.

## IndexError

**Emplacement :** BaseException ← exception ← LookupError ← IndexError

**Description :** exception concrète déclenchée lorsque vous tentez d'accéder à un élément de séquence inexistant (par exemple, l'élément d'une liste)

**Code:**

```
# Le code montre une manière extravagante  
# de quitter la boucle.
```

```
the_list = [1, 2, 3, 4, 5]  
ix = 0  
do_it = True  
  
while do_it:  
    try:  
        print(the_list[ix])  
        ix += 1  
    except IndexError:  
        do_it = False  
  
print('Done')
```

## KeyboardInterrupt

**Emplacement :** BaseException ← KeyboardInterrupt

**Description :** **exception** concrète déclenchée lorsque l'utilisateur utilise un raccourci clavier conçu pour mettre fin à l'exécution d'un programme (*Ctrl-C* dans la plupart des systèmes d'exploitation) ; si la gestion de cette exception n'entraîne pas l'arrêt du programme, le programme poursuit son exécution.

Remarque : cette exception n'est pas dérivée de la classe Exception.

**Code:**

```
# Ce code ne peut pas être terminé  
# en appuyant sur Ctrl-C.  
  
from time import sleep  
  
seconds = 0  
  
while True:  
    try:  
        print(seconds)  
        seconds += 1  
        sleep(1)  
    except KeyboardInterrupt:  
        print("Don't do that!")
```

*LookupError*

**Emplacement :** BaseException ← Exception ← LookupError

**Description :** exception abstraite incluant toutes les exceptions causées par des erreurs résultant de références non valides à différentes collections (listes, dictionnaires, tuples, etc.)

*MemoryError (Erreur de mémoire)*

**Emplacement :** BaseException ← exception ← MemoryError

**Description :** exception concrète déclenchée lorsqu'une opération ne peut pas être terminée en raison d'un manque de mémoire libre.

**Code:**

```
# Ce code provoque l'exception MemoryError .
# Attention: l'exécution de ce code peut affecter votre OS.
# Ne l'exécutez pas dans des environnements de production !
```

```
string = 'x'
try:
    while True:
        string = string + string
        print(len(string))
except MemoryError:
    print('This is not funny!')
```

*OverflowError*

**Emplacement :** BaseException ← exception ← ArithmeticError ← OverflowError

**Description :** une exception concrète déclenchée lorsqu'une opération produit un nombre trop important pour être stocké avec succès

**Code:**

```
# Le code imprime les valeurs suivantes
# de exp(k), k = 1, 2, 4, 8, 16, ...
```

```
from math import exp
```

```
ex = 1
```

```
try:
    while True:
        print(exp(ex))
        ex *= 2
except OverflowError:
    print('The number is too big.')
```

*ImportError*

**Emplacement :** BaseException ← exception ← StandardError ← ImportError

**Description :** exception concrète déclenchée en cas d'échec d'une opération d'importation

**Code:**

```
# L'une de ces importations échouera, laquelle?

try:
    import math
    import time
    import abracadabra

except:
    print('One of your imports has failed.')
```

*KeyError*

**Emplacement :** BaseException ← Exception ← LookupError ← KeyError

**Description :** exception concrète déclenchée lorsque vous tentez d'accéder à un élément inexistant d'une collection (par exemple, l'élément d'un dictionnaire)

**Code:**

```
# Comment abuser du dictionnaire et comment y faire face?

dictionary = { 'a': 'b', 'b': 'c', 'c': 'd' }
ch = 'a'

try:
    while True:
        ch = dictionary[ch]
        print(ch)
except KeyError:
    print('No such key:', ch)
```

Nous en avons fini avec les exceptions pour l'instant, mais elles reviendront lorsque nous discuterons de la programmation orientée objet en Python. Vous pouvez les utiliser pour protéger votre code contre les accidents graves, mais vous devez également apprendre à vous y plonger, en explorant les informations qu'ils contiennent.

Les exceptions sont en fait des objets, cependant, nous ne pouvons rien vous dire sur cet aspect jusqu'à ce que nous vous présentions des classes, des objets, etc.

Pour le moment, si vous souhaitez en savoir plus sur les exceptions par vous-même, vous consultez la bibliothèque Python standard sur

<https://docs.python.org/3.6/library/exceptions.html>.

## 15.6.2 Exercice

## Temps estimé

15-25 minutes

## Objectifs

- améliorer les compétences de l'élève dans la définition des fonctions;
- utiliser des exceptions afin de fournir un environnement d'entrée sécurisé.

## Scénario

Votre tâche consiste à écrire une **fonction capable d'entrer des valeurs entières et de vérifier si elles se trouvent dans une plage spécifiée**.

La fonction devrait :

- Accepter trois arguments : une promptitude, une limite acceptable basse et une limite acceptable élevée;
- Si l'utilisateur entre une chaîne qui n'est pas une valeur entière, la fonction doit émettre le message `Erreur: saisie incorrecte` et demander à l'utilisateur d'entrer à nouveau la valeur ;
- Si l'utilisateur entre un nombre qui se situe en dehors de la plage spécifiée, la fonction doit émettre le message `Erreur : la valeur n'est pas dans la plage autorisée (min.. max)` et demander à l'utilisateur de saisir à nouveau la valeur;
- Si la valeur d'entrée est valide, renvoyez-la en conséquence.

```
def read_int(prompt, min, max):  
    #  
    # Code ici.  
    #  
  
v = read_int("Entrer un nombre entre -10 et 10: ", -10, 10)  
  
print("Le nombre est:", v)
```

## Données d'essai

Voici comment la fonction doit réagir à l'entrée de l'utilisateur :

```
Entrez un nombre compris entre -10 et 10 : 100  
Erreur : la valeur n'est pas dans la plage autorisée (-10..10)  
Entrez un nombre compris entre -10 et 10 : asd  
Erreur : saisie incorrecte  
Entrez le nombre de -10 à 10: 1  
Le nombre est : 1
```

### 15.6.3 Résumé

1. Certaines exceptions abstraites intégrées à Python sont:

- `ArithmeticError`,
- `BaseException`,
- `LookupError`.

2. Certaines exceptions Python intégrées concrètes sont:

- `AssertionError`,
- `ImportError`,
- `IndexError`,
- `KeyboardInterrupt`,
- `KeyError`,
- `ErreurMémoire`,
- `OverflowError`.

#### Exercice 1

Laquelle des exceptions utiliserez-vous pour empêcher votre code d'être interrompu par l'utilisation du clavier ?

#### Exercice 2

Quel est le nom de la plus générale de toutes les exceptions Python ?

#### Exercice 3

Lesquelles des exceptions seront soulevées par l'évaluation infructueuse suivante?

```
valueTest = 1E250 ** 2
```



## 16 Les fichiers

### 16.1 Accès aux fichiers à partir du code Python

L'un des problèmes les plus courants dans le travail du développeur est de **traiter les données stockées dans** des fichiers alors que les fichiers sont généralement stockés physiquement à l'aide de périphériques de stockage : disques durs, optiques, réseau ou SSD.

Il est facile d'imaginer un programme qui trie 20 numéros, et il est tout aussi facile d'imaginer l'utilisateur de ce programme entrant ces vingt numéros directement à partir du clavier.

Il est beaucoup plus difficile d'imaginer la même tâche lorsqu'il y a 20 000 numéros à trier et qu'il n'y a pas un seul utilisateur capable d'entrer ces numéros sans faire d'erreur.

Il est beaucoup plus facile d'imaginer que ces nombres sont stockés dans un fichier qui est lu par le programme. Le programme trie les nombres et ne les envoie pas à l'écran, mais crée un nouveau fichier et y enregistre la séquence de nombres triée.

Si nous voulons implémenter une base de données simple, la seule façon de stocker les informations entre les exécutions du programme est de les enregistrer dans un fichier (ou des fichiers si votre base de données est plus complexe).

En principe, tout problème de programmation non simple repose sur l'utilisation de fichiers, qu'il s'agisse de traiter des images (stockées dans des fichiers), de multiplier les matrices (stockées dans des fichiers) ou de calculer les salaires et les impôts (lecture de données stockées dans des fichiers).



Vous vous demandez peut-être pourquoi nous avons attendu jusqu'à maintenant pour vous montrer ce point.

La réponse est très simple: la façon dont Python accède et traite les fichiers est implémentée à l'aide d'un ensemble cohérent d'objets. Et comme nous n'avons pas encore vu ce qu'est un objet, c'est un petit peu complexe de l'aborder facilement.

### 16.2 Noms de fichiers

Différents systèmes d'exploitation peuvent traiter les fichiers de différentes manières. Par exemple, Windows utilise une convention de nommage différente de celle adoptée dans les systèmes Unix/Linux.

Si nous utilisons la notion de nom de fichier canonique (un nom qui définit de manière unique l'emplacement du fichier quel que soit son niveau dans l'arborescence des répertoires), nous pouvons nous rendre compte que ces noms sont différents sous Windows et sous Unix/Linux :

Windows

```
C:\directory\file
```

Linux

```
/directory/files
```

Comme vous pouvez le voir, les systèmes dérivés d'Unix/Linux n'utilisent pas la lettre de lecteur de disque (par exemple, C:) et tous les répertoires se développent à partir d'un répertoire racine appelé /, tandis que les systèmes Windows reconnaissent le répertoire racine comme \.

En outre, les noms de fichiers système Unix/Linux sont sensibles à la casse. Les systèmes Windows stockent la casse des lettres utilisées dans le nom de fichier, mais ne font aucune distinction entre leurs casses.

Cela signifie que ces deux chaînes: `ThisIsTheNameOfTheFile` et `thisisthenamethatthefile` décrivent deux fichiers différents dans les systèmes Unix/Linux, mais portent le même nom pour un seul fichier dans les systèmes Windows.

La différence principale et la plus frappante est que vous devez utiliser **deux séparateurs différents pour les noms de répertoire**: \ sous Windows et / sous Unix/Linux.

Cette différence n'est pas très importante pour l'utilisateur normal, mais est **très importante lors de l'écriture de programmes en Python**.

Pour comprendre pourquoi, essayez de rappeler le rôle très spécifique joué par le \ à l'intérieur des chaînes Python.

Supposons que vous soyez intéressé par un fichier particulier situé dans le répertoire `dir` et nommé « fichier ».

Supposons également que vous souhaitiez affecter à une chaîne le nom du fichier.

Dans les systèmes Unix/Linux, elle peut se présenter comme suit :

```
name = « /dir/fichier »
```

Mais si vous essayez de coder pour le système Windows:

```
name = « \dir\fichier »
```

Vous aurez une mauvaise surprise: soit Python générera une erreur, soit l'exécution du programme se comportera étrangement, comme si le nom du fichier avait été déformé d'une manière ou d'une autre.

En fait, ce n'est pas étrange du tout, mais tout à fait évident et naturel. Python utilise le \ comme caractère d'échappement (comme \n).

Cela signifie que les noms de fichiers Windows doivent être écrits comme suit :

```
name = « \\dir\\fichier »
```

Heureusement, il existe également une autre solution. Python est assez intelligent pour pouvoir convertir les barres obliques en barres obliques inverses chaque fois qu'il découvre que cela est requis par le système d'exploitation.

Cela signifie que l'une des affectations suivantes :

```
name = « /dir/fichier »
```

```
name = « c:/dir/fichier »
```

fonctionnera également avec Windows.

Tout programme écrit en Python (et pas seulement en Python, car cette convention s'applique à pratiquement tous les **langages** de programmation) ne communique pas directement avec les fichiers, mais à travers certaines entités abstraites nommées différemment dans différents langages ou environnements, les termes les plus utilisés sont les **descripteurs** ou les flux (nous les utiliserons comme synonymes ici).

Le programmeur, ayant un ensemble plus ou moins riche de fonctions/méthodes, est capable d'effectuer certaines opérations sur le flux, qui affectent les fichiers réels en utilisant les mécanismes contenus dans le noyau du système d'exploitation.

De cette façon, vous pouvez implémenter le processus d'accès à n'importe quel fichier, même lorsque le nom du fichier est inconnu au moment de l'écriture du programme.

Les opérations effectuées avec le flux abstrait reflètent les activités liées au fichier physique.

Pour connecter (lier) le flux au fichier, il est nécessaire d'effectuer une opération explicite.

L'opération de connexion du flux à un fichier est appelée ouverture du fichier, tandis que la déconnexion de ce lien est appelée **fermeture du fichier**.

Par conséquent, la conclusion est que la toute première opération effectuée sur le flux est toujours **open** et la dernière est **close**. Le programme, en effet, est libre de manipuler le flux entre ces deux événements et de manipuler le fichier associé.

Cette liberté est limitée, bien sûr, par les caractéristiques physiques du fichier et la manière dont le fichier a été ouvert.

Disons à nouveau que l'ouverture du flux peut échouer, et cela peut arriver pour plusieurs raisons: la plus courante est l'absence d'un fichier avec le nom spécifié. Il peut également arriver que le fichier physique existe, mais que le programme ne soit pas autorisé à l'ouvrir. Il y a aussi le risque que le programme ait ouvert trop de flux, et le système d'exploitation spécifique peut ne pas permettre l'ouverture simultanée de plus de n fichiers (par exemple, 200).

Un programme bien écrit doit détecter ces ouvertures défectueuses et réagir en conséquence.

### 16.3 Flux de fichiers

L'ouverture du flux n'est pas seulement associée au fichier, mais doit également déclarer la manière dont le flux sera traité. Cette déclaration est appelée « **open mode** ».

Si l'ouverture réussit, le **programme ne sera autorisé à effectuer que les opérations compatibles avec le mode ouvert déclaré.**

Deux opérations de base sont effectuées sur le flux :

- **Lire** à partir du flux : les parties des données sont extraites du fichier et placées dans une zone mémoire gérée par le programme (par exemple, une variable);
- **Écrire** dans le flux : les parties des données de la mémoire (par exemple, une variable) sont transférées dans le fichier.

Trois modes de base sont utilisés pour ouvrir le flux :

- **Mode lecture** : un flux ouvert dans ce mode autorise **uniquement les opérations de lecture** ; toute tentative d'écriture dans le flux entraînera une exception (l'exception est nommée `UnsupportedOperation`, qui hérite d'`OSError` et de `ValueError` et provient du module `io`) ;
- **Mode d'écriture** : un flux ouvert dans ce mode n'autorise **que les opérations d'écriture** ; toute tentative de lecture du flux entraînera l'exception mentionnée ci-dessus ;
- **Mode de mise à jour** : un flux ouvert dans ce mode permet **à la fois les écritures et les lectures.**

Avant de discuter de la façon de manipuler les flux, nous vous devons quelques explications. **Le flux se comporte presque comme un magnétophone.**

Lorsque vous lisez quelque chose à partir d'un flux, une tête virtuelle se déplace sur le flux en fonction du nombre d'octets transférés à partir du flux.

Lorsque vous écrivez quelque chose dans le flux, la même tête se déplace le long du flux en enregistrant les données de la mémoire.

Chaque fois que nous parlons de lire et d'écrire dans le flux, essayez d'imaginer cette analogie. Les livres de programmation se réfèrent à ce mécanisme comme la **position actuelle du fichier**, et nous utiliserons également ce terme.



Il est maintenant nécessaire de vous montrer l'objet responsable de la représentation des flux dans les programmes.

## 16.4 Gestion des fichiers

Python suppose que **chaque fichier est caché derrière un objet d'une classe adéquate**.

Bien sûr, il est difficile de ne pas se demander comment interpréter le mot *adéquat*. Et encore plus de parler d'objet et de classe mais ces deux mots on y reviendra.

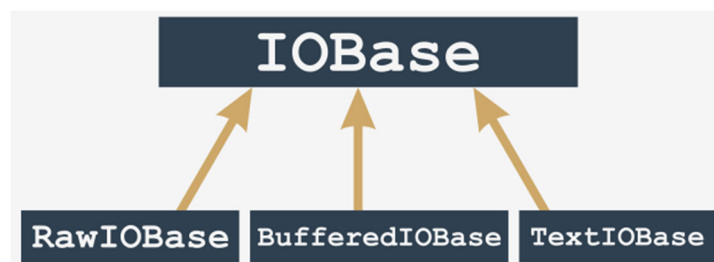
Les fichiers peuvent être traités de différentes manières, certaines d'entre elles dépendent du contenu du fichier, d'autres des intentions du programmeur.

Dans tous les cas, différents fichiers peuvent nécessiter différents ensembles d'opérations et se comporter de différentes manières.

Un objet d'une classe adéquate est **créé lorsque vous ouvrez le fichier et l'annihilez au moment de la fermeture**.

Entre ces deux événements, vous pouvez utiliser l'objet pour spécifier les opérations à effectuer sur un flux particulier. Les opérations que vous êtes autorisé à utiliser sont imposées par **la façon dont vous avez ouvert le fichier**. Ok, ce n'est pas encore très clair, mais retenez qu'un objet c'est un peu comme une variable avec des fonctions qui lui sont propres (ou une liste).

En général, l'objet provient de l'une des classes présentées ici :



Remarque : vous n'utilisez jamais de constructeurs (voir Q2) pour donner vie à ces objets. La seule façon de **les obtenir est d'invoquer la fonction** nommée `open()`.

La fonction analyse les arguments que vous avez fournis et crée automatiquement l'objet requis.

Si vous souhaitez **vous débarrasser du fichier, vous appelez la méthode** nommée `close()`.

L'appel rompra la connexion à l'objet et au fichier et supprimera l'objet.

Pour nos besoins, nous nous intéresserons uniquement aux flux représentés par les objets **BufferIOBase** et **TextIOBase**. Vous comprendrez pourquoi bientôt. En raison du type de contenu du flux, **tous les flux sont divisés en flux texte et binaires**.

Les flux de texte sont structurés en lignes; c'est-à-dire qu'ils contiennent des caractères typographiques (lettres, chiffres, ponctuation, etc.) disposés en colonnes (lignes), comme on le voit à l'œil nu lorsque vous regardez le contenu du fichier dans l'éditeur.

Ce fichier est écrit (ou lu) principalement caractère par caractère, ou ligne par ligne.

Les flux binaires ne contiennent pas de texte mais une séquence d'octets de n'importe quelle valeur. Cette séquence peut être, par exemple, un programme exécutable, une image, un clip audio ou vidéo, un fichier de base de données, etc.

Étant donné que ces fichiers ne contiennent pas de lignes, les lectures et écritures concernent des parties de données de toute taille. Par conséquent, les données sont lues / écrites octet par octet, ou bloc par bloc, où la taille du bloc varie généralement d'une à une valeur choisie arbitrairement.

Vient ensuite un problème subtil. Dans les systèmes Unix/Linux, les extrémités de ligne sont marquées par un seul caractère nommé LF (code ASCII 10) désigné dans les programmes Python comme `\n`.



D'autres systèmes d'exploitation, en particulier ceux dérivés du système préhistorique CP/M (qui s'applique également aux systèmes de la famille Windows) utilisent une convention différente: la fin de ligne est marquée par une paire de caractères, CR et LF (codes ASCII 13 et 10) qui peuvent être codés comme `\r\n`.

Cette ambiguïté peut avoir diverses conséquences désagréables.

Si vous créez un programme responsable du traitement d'un fichier texte, et qu'il est écrit pour Windows, vous pouvez reconnaître les extrémités des lignes en trouvant les caractères `\r\n`, mais le même programme exécuté dans un environnement Unix/Linux sera complètement inutile, et vice versa.

De telles fonctionnalités indésirables du programme, qui empêchent ou entravent l'utilisation du programme dans différents environnements, sont appelées **non-portable**.

De même, la caractéristique du programme permettant l'exécution dans différents environnements est appelée **portabilité**. Un programme doté d'un tel trait est appelé un programme **portable**.

Étant donné que les problèmes de portabilité étaient (et sont toujours) très graves, une décision a été prise pour résoudre définitivement le problème d'une manière qui n'engage pas l'attention du développeur.



Cela a été fait au niveau des classes, qui sont responsables de la lecture et de l'écriture des caractères vers et depuis le flux. Il fonctionne de la manière suivante:

- Lorsque le flux est ouvert et que les données du fichier associé sont traitées sous forme de texte, il passe en mode texte ;
- Lors de la **lecture/écriture** de lignes **de/vers** le fichier associé, rien de spécial ne se produit dans l'environnement Unix, mais lorsque les mêmes opérations sont effectuées dans l'environnement Windows, un processus appelé traduction des caractères de nouvelle ligne se produit : lorsque vous lisez une ligne du fichier, chaque paire de caractères `\r\n` est remplacée par un seul caractère `\n`, et vice versa;

- Le mécanisme est complètement **transparent** pour le programme, qui peut être écrit comme s'il était destiné au traitement de fichiers texte Unix/Linux uniquement; le code source exécuté dans un environnement Windows fonctionnera également correctement;
- Lorsque le flux est ouvert, son contenu est pris tel quel, **sans aucune conversion**, aucun octet n'est ajouté ou omis.

### 16.5 Ouverture des flux

**L'ouverture du flux** est effectuée par une fonction qui peut être invoquée de la manière suivante :

```
stream = open(file, mode = 'r', encoding = None)
```

Analysons-la ligne :

- le nom de la fonction (open) parle de lui-même ; si l'ouverture réussit, la fonction renvoie un objet stream ; sinon, une exception est déclenchée (par exemple, FileNotFoundError **si le fichier que vous allez lire n'existe pas**) ;
- le premier paramètre de la fonction (file) spécifie le nom du fichier à associer au flux ;
- le deuxième paramètre (mode) spécifie le mode ouvert utilisé pour le flux; c'est une chaîne remplie d'une séquence de caractères, et chacun d'eux a sa propre signification particulière (plus de détails bientôt);
- le troisième paramètre (encoding) spécifie le type de codage (par exemple, UTF-8 lorsque vous travaillez avec des fichiers texte)

L'ouverture doit être la toute première opération effectuée sur le flux.

Remarque: les arguments de mode et d'encodage peuvent être omis, leurs valeurs par défaut sont alors supposées. Le mode d'ouverture par défaut est la lecture en mode texte, tandis que l'encodage par défaut dépend de la plateforme utilisée.

### 16.5.1 Ouverture des flux : modes

r : lecture

- Le flux sera ouvert en **mode lecture** ;
- Le fichier associé au flux doit **exister** et doit être lisible, sinon la fonction `open()` déclenche une exception.

w : écriture

- Le flux sera ouvert en **mode écriture** ;
- Le fichier associé au flux n'a pas besoin d'exister ; s'il n'existe pas, il sera créé ; s'il existe, il sera tronqué à la longueur zéro (effacé) ; si la création n'est pas possible (par exemple, en raison d'autorisations système), la fonction `open()` déclenche une exception.

a : Ajouter

- Le flux sera ouvert en **mode append** ;
- Le fichier associé au flux **n'a pas besoin d'exister** ; s'il n'existe pas, il sera créé ; s'il existe, la tête d'enregistrement virtuelle sera définie à la fin du fichier (le contenu précédent du fichier reste intact).

r+ : lecture et mise à jour

- Le flux sera ouvert en **mode lecture et mise à jour** ;
- Le fichier associé au flux doit **exister et doit être accessible en écriture**, sinon la fonction `open()` déclenche une exception ;
- Les opérations de lecture et d'écriture sont autorisées pour le flux.

w+ : écriture et mise à jour

- Le flux sera ouvert en **mode écriture et mise à jour** ;
- le fichier associé au flux **n'a pas besoin d'exister** ; s'il n'existe pas, il sera créé ; le contenu précédent du fichier reste intact ;
- Les opérations de lecture et d'écriture sont autorisées pour le flux.

### 16.5.2 Sélection des modes texte et binaire

S'il y a une lettre b à la fin de la chaîne de mode, cela signifie que le flux doit être ouvert en mode **binaire**.

Si la chaîne de mode se termine par une lettre t, le flux est ouvert en **mode texte**.

Le mode texte est le comportement par défaut supposé lorsqu'aucun spécificateur de mode binaire/texte n'est utilisé.

Enfin, l'ouverture réussie du fichier définira la position actuelle du fichier (la tête de lecture/écriture virtuelle) avant le premier octet du fichier si le mode **n'est pas a** et après le dernier octet du fichier **si le mode est défini sur a**.



Mode texte	Mode binaire	Description
Rt	Rb	lire
Wt	Wb	écrire
a	De	ajouter
R+T	R+B	Lire et mettre à jour
W+T	W+B	Écrire et mettre à jour

### 16.5.3 Ouverture du flux pour la première fois

Imaginez que nous voulions développer un programme qui lit le contenu du fichier texte nommé : C:\Users\User\Desktop\file.txt.

Comment ouvrir ce fichier pour la lecture? Voici l'extrait de code pertinent :

```
try:
    stream = open("C:\Users\User\Desktop\file.txt", "rt")
    # Le traitement à lieu ici.
    stream.close()
except Exception as exc:
    print("Cannot open the file:", exc)
```

Qu'est-ce qui se passe?

- Nous utilisons le bloc try-except car nous voulons gérer les erreurs d'exécution en douceur;
- Nous utilisons la fonction open() pour essayer d'ouvrir le fichier spécifié (notez la façon dont nous avons spécifié le nom du fichier)
- Le mode ouvert est défini comme du texte à lire (comme le **texte est le paramètre par défaut**, nous pouvons ignorer la chaîne t en mode)
- En cas de succès, nous obtenons un objet de la fonction open() et nous l'affectons à la variable stream;
- Si open() échoue, nous gérons l'exception en imprimant les informations d'erreur complètes (il est utile de savoir ce qu'il s'est passé exactement)

#### 16.5.4 Flux pré-ouverts

Nous avons dit précédemment que toute opération de flux doit être précédée de l'appel de la fonction `open()`. Il y a trois exceptions bien définies à la règle.

Lorsque notre programme commence, les trois flux sont déjà ouverts et ne nécessitent aucune préparation supplémentaire. De plus, votre programme peut utiliser ces flux explicitement si vous prenez soin d'importer le module `sys` :

```
import sys
```

Parce que c'est dans ce module que la déclaration des trois flux est placée.

Les noms de ces flux sont : **`sys.stdin`**, **`sys.stdout`** et **`sys.stderr`**.

Analysons-les :

- `sys.stdin`
  - `stdin` (en tant qu'entrée standard)
  - Le flux `stdin` est normalement associé au clavier, pré-ouvert pour la lecture et considéré comme la source de données principale pour les programmes en cours d'exécution;
  - La fonction `input()` lit les données de `Stdin` par défaut.
- `sys.stdout`
  - `stdout` (en sortie standard)
  - Le flux `stdout` est normalement associé à l'écran, pré-ouvert pour l'écriture, considéré comme la cible principale pour la sortie de données par le programme en cours d'exécution;
  - La fonction `print()` envoie les données au flux `stdout`.
- `sys.stderr`
  - `stderr` (en tant que sortie d'erreur standard)
  - Le flux `stderr` est normalement associé à l'écran, pré-ouvert à l'écriture, considéré comme le lieu principal où le programme en cours d'exécution doit envoyer des informations sur les erreurs rencontrées au cours de son travail;
  - Nous n'avons présenté aucune méthode pour envoyer les données à ce flux (nous le ferons bientôt, promis)
  - La séparation des `stdout` (résultats utiles produits par le programme) des `stderr` (messages d'erreur, indéniablement utiles mais ne donnant pas de résultats) donne la possibilité de rediriger ces deux types d'informations vers les différentes cibles. Une discussion plus approfondie de cette question dépasse la portée de notre cours. Le manuel du système d'exploitation fournira plus d'informations sur ces questions.

### 16.5.5 Fermeture des flux

La dernière opération effectuée sur un flux (cela n'inclut pas les flux `stdin` , `stdout` et `stderr` qui n'en ont pas besoin) est de les **fermer**.

Cette action est effectuée par une méthode appelée à partir de l'objet ouvert dans le flux : `stream.close()`.

- Le nom de la fonction est auto-commenté: **`close()`**
- La fonction attend exactement zéro argument ; Le flux n'a pas besoin d'être ouvert
- La fonction ne renvoie rien mais déclenche l'exception `IOError` en cas d'erreur ;
- La plupart des développeurs pensent que la fonction `close()` réussit toujours et qu'il n'est donc pas nécessaire de vérifier si elle a fait sa tâche correctement.  
Cette croyance n'est que partiellement justifiée. Si le flux a été ouvert pour l'écriture et qu'une série d'opérations d'écriture ont été effectuées, il peut arriver que les données envoyées au flux n'aient pas encore été transférées vers le périphérique physique (en raison d'un mécanisme appelé ***mise en cache*** ou ***mise en mémoire tampon***).  
Étant donné que la fermeture du flux force les tampons à se vider, il se peut que cela échoue et donc que le `close()` échoue aussi.

Nous avons déjà mentionné les défaillances causées par des fonctions fonctionnant avec des flux, mais nous n'avons pas parlé de la façon dont nous pouvons identifier exactement la cause de cette défaillance. La possibilité de faire un diagnostic existe et est fournie par l'une des composantes d'exception des flux. Wait and see, cela arrive.

### 16.5.6 Diagnostic des problèmes de flux

L'objet **`IOError`** est équipé d'une propriété nommée **`errno`** (le nom vient de l'expression *numéro d'erreur*) et vous pouvez y accéder comme suit :

```
try:
    # Some stream operations.
except IOError as exc:
    print(exc.errno)
```

Jetons un coup d'œil à quelques **constantes sélectionnées utiles pour détecter les erreurs de flux**:

- `errno.EACCES` → Autorisation refusée

L'erreur se produit lorsque vous essayez, par exemple, d'ouvrir un fichier avec l'attribut *lecture seule* pour l'écriture.

- `errno.EBADF` → Numéro de fichier incorrect

L'erreur se produit lorsque vous essayez, par exemple, d'utiliser un flux non ouvert.

- `errno.EEXIST` → Fichier existe

L'erreur se produit lorsque vous essayez, par exemple, de renommer un fichier avec son nom précédent.

- `errno.EFBIG` → Fichier trop volumineux

L'erreur se produit lorsque vous essayez de créer un fichier qui est plus grand que le maximum autorisé par le système d'exploitation.

- `errno.EISDIR` → est un répertoire  
L'erreur se produit lorsque vous essayez de traiter un nom de répertoire comme le nom d'un fichier ordinaire.

- `errno.EMFILE` → Trop de fichiers ouverts

L'erreur se produit lorsque vous essayez d'ouvrir simultanément plus de flux que ce qui est acceptable pour votre système d'exploitation.

- `errno.ENOENT` → Aucun fichier ou répertoire de ce type

L'erreur se produit lorsque vous essayez d'accéder à un fichier/répertoire inexistant.

- `errno.ENOSPC` → Pas d'espace sur l'appareil

L'erreur se produit lorsqu'il n'y a pas d'espace libre sur le média.

La liste complète est beaucoup plus longue (elle inclut également des codes d'erreur non liés au traitement du flux). Mais nous pensons que cela est déjà bien pour un cours d'introduction.

#### 16.5.7 Diagnostic des problèmes de flux, la suite

Si vous êtes un programmeur très prudent, vous pouvez ressentir le besoin d'utiliser la séquence d'instructions similaire à celles présentées ci-dessous.

```
import errno

try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Proccessus ici.
    s.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file doesn't exist.")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files.")
    else:
        print("The error number is:", exc.errno)
```

Heureusement, il existe une fonction qui peut simplifier considérablement le **code de gestion des erreurs** que vous pouvez voir.

Son nom est **strerror()**, et elle vient du module **os** et **n'attend qu'un seul argument : un numéro d'erreur**.

Son rôle est simple : vous donnez un numéro d'erreur et obtenir une chaîne décrivant la signification de l'erreur.

Remarque: si vous passez un code d'erreur inexistant (un nombre qui n'est lié à aucune erreur réelle), la fonction déclenchera l'exception `ValueError`.

Nous pouvons donc simplifier notre code de la manière suivante:

```
from os import strerror

try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Actual processing goes here.
    s.close()
except Exception as exc:
    print("The file could not be opened:", strerror(exc.errno))
```

#### 16.5.8 Principaux points à retenir

1. Un fichier doit être **ouvert** avant de pouvoir être traité par un programme, et il doit être **fermé** lorsque le traitement est terminé.

L'ouverture du fichier l'associe au **flux**, et est une représentation abstraite des données physiques stockées sur le support. La façon dont le flux est traité est appelée **open mode**. Trois modes ouverts existent :

- **Mode de lecture** : seules les opérations de lecture sont autorisées;
- **Mode d'écriture** : seules les opérations d'écriture sont autorisées;
- **Mode de mise à jour** : les écritures et les lectures sont autorisées.

2. Selon le contenu du fichier physique, différentes classes Python peuvent être utilisées pour traiter les fichiers. En général, **BufferedIOBase** est capable de traiter n'importe quel fichier, tandis que **TextIOBase** est une classe spécialisée dédiée au traitement des fichiers texte (c'est-à-dire des fichiers contenant des textes visibles par l'homme divisés en lignes à l'aide de marqueurs de nouvelle ligne). Ainsi, les flux peuvent être divisés en **binares** et **textes**.

3. La syntaxe suivante de la fonction `open()` est utilisée pour ouvrir un fichier :

```
open(file_name, mode=open_mode, encoding=text_encoding)
```

L'appel crée un objet stream et l'associe au fichier nommé `file_name`, en utilisant l'`open_mode` spécifié et en définissant le `text_encoding` spécifié, où il **déclenche une exception en cas d'erreur**.

4. Trois volets prédéfinis sont déjà ouverts au démarrage du programme :

- `sys.stdin` : entrée standard;
- `sys.stdout` : sortie standard;
- `sys.stderr` : sortie d'erreur standard.

4. L'objet d'exception **IOError**, créé lorsque des opérations de fichier échouent (y compris les opérations ouvertes), contient une propriété nommée `errno`, qui contient le code d'achèvement de l'action ayant échoué. Utilisez cette valeur pour diagnostiquer le problème.

### Exercice 1

Comment encoder l'argument `mode` d'une fonction `open()` si vous allez créer un nouveau fichier texte que vous voulez remplir uniquement avec un texte ?

### Exercice 2

Quelle est la signification de la valeur représentée par `errno.EACCES` ?

### Exercice 3

Quelle est la sortie attendue du code suivant, en supposant que le fichier nommé *file* n'existe pas ?

```
import errno

try:
    stream = open("file", "rb")
    print("exists")
    stream.close()
except IOError as error:
    if error.errno == errno.ENOENT:
        print("absent")
    else:
        print("unknown")
```

## 16.6 Le travail sur les fichiers texte

### 16.6.1 Traitement des fichiers texte

Dans ce chapitre, nous allons préparer un fichier texte simple avec un contenu court et simple.

Nous allons vous montrer quelques techniques de base que vous pouvez utiliser pour **lire le contenu du fichier** afin de les **traiter**.

Le traitement sera très simple, vous allez copier le contenu du fichier sur la console et compter tous les caractères que le programme a lus.

Mais rappelez-vous que notre compréhension d'un fichier texte est très stricte. Dans notre vision des choses, il s'agit d'un fichier texte brut, il ne peut contenir que du texte, sans aucune décoration supplémentaire (formatage, polices différentes, etc.).

C'est pourquoi vous devriez éviter de créer le fichier à l'aide d'un traitement de texte avancé comme MS Word, LibreOffice Writer, ou un logiciel du genre. Utilisez ceux de bases de votre système d'exploitation: bloc-notes, vim, gedit, etc.

Si vos fichiers texte contiennent des caractères nationaux non couverts par le jeu de caractères ASCII standard, vous aurez peut-être besoin d'une étape supplémentaire. Votre invocation de fonction `open()` peut nécessiter un argument désignant un codage de texte spécifique.

Par exemple, si vous utilisez un système d'exploitation Unix/Linux configuré pour utiliser UTF-8 comme paramètre à l'échelle du système, la fonction `open()` peut se présenter comme suit :

```
stream = open('file.txt', 'rt', encoding='utf-8')
```

où l'argument d'encodage doit être défini sur une valeur qui est une chaîne représentant le codage de texte approprié (UTF-8, dans notre cas).

Exemple :

```
# Opening tzop.txt in read mode, returning it as a file
object:
stream = open("tzop.txt", "rt", encoding = "utf-8")

print(stream.read()) # printing the content of the file
```

La lecture du contenu d'un fichier texte peut être effectuée à l'aide de plusieurs méthodes différentes, aucune d'entre elles n'est meilleure ou pire qu'une autre. C'est à vous de décider laquelle vous préférez.

Certains d'entre elles seront parfois plus pratiques, et parfois plus gênantes. Faites preuve de souplesse. N'ayez pas peur de changer vos préférences.

La plus basique de ces méthodes est celle offerte par la fonction `read()`, que vous avez pu voir en action précédemment.

Si elle est appliquée à un fichier texte, la fonction est capable de :

- lire le nombre souhaité de caractères (dont un seul) dans le fichier et les renvoyer sous forme de chaîne ;
- lire tout le contenu du fichier et le renvoyer sous forme de chaîne ;
- S'il n'y a plus rien à lire (la tête de lecture virtuelle atteint la fin du fichier), la fonction renvoie une chaîne vide.

Nous allons commencer par la variante la plus simple et utiliser un fichier nommé text.txt. Le fichier a le contenu suivant :

Beau vaut mieux que laid.

Explicite vaut mieux qu'implicite.

Simple vaut mieux que complexe.

Complexe vaut mieux que compliqué

Depreter, Desmet, Scopel que choisir?

Ça vous rappelle quelque chose ??

Maintenant, regardez le code, et analysons-le.

```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    ch = s.read(1)
    while ch != '':
        print(ch, end='')
        cnt += 1
        ch = s.read(1)
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

Le code est plutôt simple à reproduire :

- Utilisez le mécanisme try-except et ouvrez le fichier du nom prédéterminé (texte.txt dans notre cas)
- Essayez de lire le tout premier caractère du fichier (ch = s.read(1))
- Si vous réussissez (ceci est prouvé par un résultat positif de la vérification de l'état while), sortez le caractère (notez l'argument end=, c'est important! Vous ne voulez pas passer à une nouvelle ligne après chaque caractère!);
- mettre à jour le compteur (cnt);
- Essayez de lire le caractère suivant et le processus se répète.



Allons un peu plus loin

Si vous êtes absolument sûr que la longueur du fichier est bonne et que vous pouvez lire le fichier entier dans la mémoire, vous pouvez le faire, la fonction `read()`, invoquée sans aucun argument ou avec un argument qui donne la valeur `None`, fera le travail pour vous.

N'oubliez pas que **la lecture d'un fichier de téraoctets à l'aide de cette méthode peut corrompre votre système d'exploitation.**

Ne vous attendez pas à des miracles, la mémoire de l'ordinateur n'est pas extensible.

Regardez le code ci-dessous. Qu'en pensez-vous ?

```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    content = s.read()
    for ch in content:
        print(ch, end='')
        cnt += 1
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

Analysons-le :

- ouvrez le fichier comme précédemment;
- lire son contenu par un appel de fonction `read()`;
- Ensuite, traitez le texte, en l'itérant avec une boucle `for` et en mettant à jour la valeur du compteur à chaque tour de la boucle;

Le résultat sera exactement le même que précédemment.

### 16.6.2 Traitement des fichiers texte : readline()

Si vous souhaitez traiter le contenu du fichier comme un **ensemble de lignes**, et non comme un tas de caractères, la méthode **readline()** vous y aidera.

La méthode tente **de lire une ligne complète de texte à partir du fichier** et la renvoie sous forme de chaîne en cas de réussite. Sinon, il retourne une chaîne vide.

Cela ouvre de nouvelles opportunités à vos talents, maintenant vous pouvez également compter facilement les lignes, pas seulement les caractères.

Profitons-en. Regardez le code ci-dessous.

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    line = s.readline()
    while line != '':
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
        line = s.readline()
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:      ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

Comme vous pouvez le voir, l'idée générale est exactement la même que dans les deux exemples précédents.

### 16.6.3 Traitement des fichiers texte : readlines()

Une autre méthode, qui traite le fichier texte comme un ensemble de lignes, et non comme une suite de caractères, est **readlines()**.

La méthode **readlines()**, lorsqu'elle est appelée sans arguments, tente de **lire tout le contenu du fichier et renvoie une liste de chaînes, un élément par ligne de fichier**.

Si vous n'êtes pas sûr que la taille du fichier soit suffisamment petite et que vous ne voulez pas tester le système d'exploitation, vous pouvez convaincre la méthode **readlines()** de ne pas lire plus d'un nombre spécifié d'octets à la fois (la valeur renvoyée reste la même, c'est une liste de chaînes).

N'hésitez pas à expérimenter avec l'exemple de code suivant pour comprendre le fonctionnement de la méthode **readlines()** :

```
s = open("text.txt")
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
s.close()
```

La taille maximale de tampon d'entrée acceptée est transmise à la méthode en tant qu'argument.

Vous pouvez vous attendre à ce que `readlines()` puisse traiter le contenu d'un fichier plus efficacement que `readline()`, car il peut être nécessaire de l'appeler moins de fois.

Remarque : lorsqu'il n'y a rien à lire dans le fichier, la méthode renvoie une liste vide. Utilisez-la pour détecter la fin du fichier.

En ce qui concerne la taille du tampon, vous pouvez vous attendre à ce que son augmentation puisse améliorer les performances d'entrée, mais il n'y a pas de règle d'or pour cela, essayez de trouver vous-même les valeurs optimales.

Regardez le code ci-dessous. Nous l'avons modifié pour vous montrer comment utiliser `readlines()`.

```
from os import strerror

try:
    cnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readlines(20)
    while len(lines) != 0:
        for line in lines:
            lcnt += 1
            for ch in line:
                print(ch, end='')
                cnt += 1
            lines = s.readlines(10)
    s.close()
    print("\n\nCharacters in file:", cnt)
    print("Lines in file:      ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

Nous avons décidé d'utiliser une mémoire tampon de 15 octets. Ne pensez pas que c'est une recommandation, c'est simplement un choix.

Nous avons utilisé une telle valeur pour éviter la situation dans laquelle le premier appel de `readlines()` consomme l'intégralité du fichier. Nous voulons que la méthode soit forcée de travailler plus en profondeur et de démontrer ses capacités.

Il y a **deux boucles imbriquées dans le code**: l'externe utilise le résultat de `readlines()` pour itérer à travers elle, tandis que l'interne imprime les lignes caractère par caractère.

#### 16.6.4 Traitement des fichiers texte : on s'accroche !

Le dernier exemple que nous pouvons vous présenter montre un trait très intéressant de l'objet retourné par la fonction `open()` en mode texte. Ok, on ne sait toujours pas ce qu'est un objet mais ce n'est pas grave, par contre la phrase suivante, nous pensons qu'elle peut vous surprendre : **l'objet est une instance de la classe itérable.**

Étrange? Incompréhensible ? Pas du tout. Utilisable? Oui, absolument.

Le **protocole d'itération défini pour l'objet** fichier est très simple, sa méthode(sa fonction dans notre langage actuel) `__next__` renvoie simplement **la ligne suivante lue à partir du fichier.**

De plus, vous pouvez vous attendre à ce que l'objet appelle automatiquement `close()` lorsque l'une des lectures de fichier atteint la fin du fichier. Regardez le code ci-dessous et voyez à quel point le code est simple et clair. Même sans connaître l'objet, on peut le comprendre et l'appliquer.

```
from os import strerror

try:
    ccnt = lcnt = 0
    for line in open('text.txt', 'rt'):
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:      ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

#### 16.6.5 Traitement des fichiers texte : `write()`

L'écriture de fichiers texte semble être plus simple, car il existe en fait qu'une méthode qui peut être utilisée pour effectuer cette tâche.

La méthode est nommée **`write()`** et elle n'attend qu'un seul argument, une chaîne qui sera transférée vers un fichier ouvert (n'oubliez pas, le mode d'ouverture doit refléter la façon dont les données sont transférées, **écrire dans fichier ouvert en mode lecture ne réussira pas**).

Aucun caractère de nouvelle ligne n'est ajouté à l'argument de `write()`, vous devez donc l'ajouter vous-même si vous souhaitez que le fichier soit rempli avec un certain nombre de lignes.

L'exemple montre un code très simple qui crée un fichier nommé `newtext.txt` (note: le mode ouvert `w` garantit que **le fichier sera créé à partir de zéro**, même s'il existe et contient des données) puis y met dix lignes.

La chaîne à enregistrer se compose du mot ligne, suivi du numéro de ligne. Nous avons décidé d'écrire le contenu de la chaîne caractère par caractère (cela se fait par la boucle `for` interne) mais vous n'êtes pas obligé de le faire de cette façon.

Nous voulions juste vous montrer que **write()** est capable de fonctionner sur des caractères uniques.

Le code crée un fichier rempli avec le texte suivant :

Ligne #1  
Ligne #2  
Ligne #3  
Ligne #4  
Ligne #5  
Ligne #6  
Ligne #7  
Ligne #8  
Ligne #9  
Ligne #10

```
from os import strerror

try:
    fo = open('newtext.txt', 'wt') # A new file is created.
    for i in range(10):
        s = "ligne #" + str(i+1) + "\n"
        for ch in s:
            fo.write(ch)
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

#### 16.6.6 Traitement des fichiers texte : Avec des lignes entières Mr ?

Regardez l'exemple. Nous avons modifié le code précédent pour écrire des lignes entières dans le fichier texte. Le contenu du fichier nouvellement créé est le même.

Remarque: vous pouvez utiliser la même méthode pour écrire dans le flux stderr, mais n'essayez pas de l'ouvrir, car il est toujours ouvert implicitement.

Par exemple, si vous souhaitez envoyer une chaîne de message à stderr pour le distinguer de la sortie normale du programme, elle peut ressembler à ceci :

```
from os import strerror

try:
    fo = open('newtext.txt', 'wt') # A new file is created.
    for i in range(10):
        s = "line #" + str(i+1) + "\n"
        for ch in s:
            fo.write(ch)
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

## 16.6.7 Exercices

## 1. Temps estimé

30-60 minutes

## Objectifs

- améliorer les compétences de l'élève dans l'utilisation des dossiers (lecture)
- utiliser des collectes de données pour compter de nombreuses données.

## Scénario

Un fichier texte contient du texte (rien d'inhabituel) mais nous devons savoir à quelle fréquence chaque lettre apparaît dans le texte. Une telle analyse peut être utile en cryptographie par exemple, nous voulons ici pouvoir le faire en référence à l'alphabet latin.

Votre tâche consiste à écrire un programme qui:

- Demander à l'utilisateur le nom du fichier d'entrée ;
- Lire le fichier (si possible) et compter toutes les lettres latines (les lettres minuscules et majuscules sont traitées de manière égale)
- Imprimer un histogramme simple par ordre alphabétique (seuls les nombres non nuls doivent être présentés)

Créez un fichier de test pour votre code et vérifiez si votre histogramme contient des résultats valides.

En supposant que le fichier de test ne contient qu'une seule ligne remplie par :  
abc

Le résultat attendu devrait être le suivant :

a -> 1

b -> 1

C -> 1

**Conseil :** Nous pensons qu'un dictionnaire est un moyen de collecte de données parfait pour stocker les valeurs. Les lettres peuvent être des clés tandis que les compteurs peuvent être les valeurs.

## 2. Temps estimé

15-30 minutes

### Objectifs

- améliorer les compétences de l'élève dans l'utilisation de fichiers (lecture/écriture)
- Utilisation de lambdas pour modifier l'ordre de tri.

### Scénario.

Le code précédent doit être amélioré. C'est correct, mais il faut que ce soit mieux.

Votre tâche consiste à apporter quelques modifications, qui génèrent les résultats suivants:

- L'histogramme de sortie sera trié en fonction de la fréquence des caractères (le plus grand compteur doit être présenté en premier)
- L'histogramme doit être envoyé à un fichier portant le même nom que celui d'entrée, mais avec le suffixe '.hist' (il doit être concaténé au nom d'origine)

En supposant que le fichier d'entrée ne contient qu'une seule ligne remplie par :  
cBabAa

Le résultat attendu devrait être le suivant :

a -> 3

b -> 2

c -> 1

**Conseil** : Utilisez un lambda pour modifier l'ordre de tri.

Tentons un exo un peu fou ....

### 3 Temps estimé

30-90 minutes

### Objectifs

- améliorer les compétences de l'élève dans l'utilisation de fichiers (lecture)
- perfectionner les capacités de l'élève à définir et à utiliser des exceptions et des dictionnaires auto-définis.

### Scénario

Le professeur MicMic dirige des cours avec les étudiants et prend régulièrement des notes dans un fichier texte sur eux. Chaque ligne du fichier contient trois éléments : le prénom de l'élève, le nom de famille de l'élève et le nombre de points que l'élève a reçus lors de certains cours.

Les éléments sont séparés par des espaces blancs. Chaque étudiant peut apparaître plus d'une fois dans le dossier du professeur.

Le fichier peut se présenter comme suit :

John	Tuche	5
Anna	Boule	4.5
John	Tuche	2
Anna	Boule	11
Jean	Petard	1,5

Votre tâche consiste à écrire un programme qui:

- Demande à l'utilisateur le nom de fichier du professeur;
- Lit le contenu du fichier et compte la somme des points reçus pour chaque élève;
- Imprime un rapport simple (mais trié), comme celui-ci :

Luca	Laporta	1,5
Nathan	Vinetot	15,5
Thibaud	Danneau	7.0

Note:

- votre programme doit être entièrement protégé contre toutes les défaillances possibles : l'inexistence du fichier, le vide du fichier ou toute défaillance des données d'entrée ; rencontrer une erreur de données devrait entraîner l'arrêt immédiat du programme, et l'erreur devra être présentée à l'utilisateur;

**Conseil** : Utilisez un dictionnaire pour stocker les données des élèves.



## 16.6.8 Résumé

1. Pour lire le contenu d'un fichier, les méthodes de flux suivantes peuvent être utilisées :

- `read(number)` : lit les nombres de caractères/octets du fichier et les renvoie sous forme de chaîne ; est capable de lire le fichier entier à la fois ;
- `readline()` : lit une seule ligne du fichier texte ;
- `readlines(number)` : lit les lignes numériques du fichier texte ; est capable de lire toutes les lignes à la fois ;

2. Pour écrire du nouveau contenu dans un fichier, les méthodes de flux suivantes peuvent être utilisées :

- `write(string)` – écrit une chaîne dans un fichier texte ;

3. La méthode `open()` retourne un objet itérable qui peut être utilisé pour itérer toutes les lignes du fichier à l'intérieur d'une boucle `for`. Par exemple :

```
for line in open("file", "rt"):  
    print(line, end='')
```

Le code copie le contenu du fichier dans la console, ligne par ligne. Remarque : le flux se ferme automatiquement lorsqu'il atteint la fin du fichier.

**Exercice 1**

Qu'attendons-nous de la méthode `readlines()` lorsque le flux est associé à un fichier vide ?

**Exercice 2**

Quel est l'objectif du code suivant ?

pour la ligne dans `open(« file », « rt »)`:

Pour le char en ligne :

Si `char.lower()` n'est pas dans « aeiouy »:  
`print(char, end="")`