Jour 1 : Les bases de Python

Sommaire

Introduction : *3*

1. **Bienvenue au jour 1** **3**
2. **Évaluations automatisées** **3**
3. **Tester son code** **3**

Partie 0 : Prérequis *4*

1. **Installation de Python** **4**
   a. macOS 4
   b. Windows 4
   c. Linux 4
2. **Choix et installation d'un IDE** **4**
   a. PyCharm 4
   b. Visual Studio Code (VS Code) 5
3. **Tester son environnement** **5**

Partie 1 : Mots clés de base *6*

1. **Fonctions et arguments** **6**
2. **Instructions conditionnelles** **6**
3. **Boucle For avec Range** **7**
4. **Boucle While avec une condition** **8**
5. **Gestion basique des erreurs** **9**
6. **Boucles imbriquées** **9**

Partie 2 : Les chaînes de caractères *11*

1. **Concaténation** **11**
2. **Formatage avec les f-strings** **11**

Partie 3 : Import, modules et packages *12*

Exercice 1 : Do-op **12**

Partie 4 : main et point d'entrée *14*

Exercice 1 : Création et utilisation d'un main **14**

Exercice 2 : Simulation de l'importation de script **15**

Partie 5 : Les différentes structures de données *16*

Exercice 1 : Création et manipulation de listes **16**

Exercice 2 : Création et manipulation de dictionnaires **17**
Exercice 3 : Création et compréhension des Tuples : collections immuables **18**
Exercice 4 : Création et manipulation d'ensembles **18**
Exercice 5 : Les compréhensions de listes **19**
Partie 6 : Fonctions natives pour manipuler les différentes structures de données ***20***
Exercice 1. Obtenir l'index et l'élément d'une structure de données **20**
Exercice 2. Itérer sur plusieurs structures de données en même temps **21**
Exercice 3. Modifier les éléments d'une structure de données **22**
Exercice 4. Filtrer les éléments contenus dans une structure de données **23**
Exercice 5. Récupérer un sous-ensemble de données **24**
Partie 7 : Manipulations de chaînes de caractères ***25***
Exercice 1 : Séparations et fusions **25**
Exercice 2 : Recherche de sous-chaînes **26**
Exercice 3 : Remplacement de sous-chaînes **26**
Exercice 4 : Conversion de casse **27**
Exercice 5 : Suppression des espaces blancs **27**
Going Further : Récapitulatif et Regex ***28***
Exercice 1 : Formatage d'IP **29**
Exercice 2 : Les bases des Regex **30**
Partie 1 : Correspondance simple 30
Partie 2 : Extraction d'adresses mail 30
Partie 3 : Trouver tous les Hashtags dans un message de réseau social 31
Partie 4 : Division de texte basée sur un pattern délimiteur 31
Partie 5 : Conversion de formats de dates 31

Introduction :

1. Bienvenue au jour 1

Bienvenue au premier jour de cette piscine Big Data. Au cours des sept prochains jours, vous allez acquérir les compétences clés nécessaires pour récupérer, manipuler, nettoyer, analyser et visualiser de grands ensembles de données. Vous commencerez par les bases de la programmation en Python (aujourd'hui) et progresserez vers des sujets plus avancés, tels que le scraping de sites web, l'interaction avec des bases de données NoSQL, ainsi que la manipulation et la visualisation des données.
Chaque jour est conçu de manière progressive et s'appuie sur ce que vous avez appris lors des exercices et jours précédents. Ainsi, à la fin de cette piscine, vous aurez de solides bases en science des données que vous pourrez appliquer à des problèmes réels.

Aujourd'hui, vous commencerez par configurer votre environnement Python, vous familiariser avec les concepts de base de la programmation Python, et commencer à manipuler des structures de données simples. L'objectif étant de vous donner une base solide en Python, qui sera essentielle pour les sujets plus avancés que nous aborderons les prochains jours.

2. Évaluations automatisées

Tous les exercices de cette piscine seront évalués automatiquement via des tests unitaires. Vos pédagos en local seront responsable du lancement de ces tests sur vos rendu ainsi que du partage des résultats obtenus. Cette évaluation automatique aidera à suivre votre progression et à garantir que vous atteignez les objectifs d'apprentissage.

3. Tester son code

Bien que vos rendus soient évalués automatiquement, il est important que vous preniez également le temps de tester vous-même votre code. Cela vous aidera non seulement à mieux comprendre certains concepts de programmation, mais aussi à garantir que votre code fonctionne dans diverses situations, avant soumission de votre rendu à la fin de la journée.

Partie 0 : Prérequis

Avant de commencer, il faut configurer votre environnement de développement. Cela implique l'installation de Python, le choix d'un environnement de développement intégré (IDE) et l'exécution de votre premier script Python.

Ces étapes vous assureront d'avoir les outils nécessaires installés et prêts à commencer à coder. Assurez-vous que tout fonctionne correctement, afin que vous soyez prêt pour les activités du premier jour.

1. Installation de Python

Pour cette piscine, il est préférable que vous installiez Python 3.12, qui est la version stable et recommandée car elle inclut les dernières fonctionnalités et améliorations de sécurité. Voici les instructions à suivre pour chaque OS :

a. macOS

Si vous ne l'avez pas déjà, installez brew qui est le gestionnaire de paquet principal de macOS. Dans un terminal, exécuter la commande « brew install python3 ».

b. Windows

Installez Python 3.12 depuis le Microsoft Store.

c. Linux

Référez-vous à la documentation de votre OS et de son gestionnaire de paquet. Voici des exemples pour Ubuntu et Fedora :

- Ubuntu -> « sudo apt install python3»
- Fedora -> Python 3.12 est installé par défaut sur la dernière version (40)

2. Choix et installation d'un IDE

Vous pouvez choisir n'importe quel IDE ou éditeur de code que vous préférez, mais voici deux IDE adapté au développement Python :

a. PyCharm

PyCharm est un IDE de la suite JetBrains, possédant divers IDE et outils très puissant pour les langages de programmation et Framework principaux. Il possède des fonctionnalités telles que

l'analyse de code, un débogueur graphique, une gestion de base de données intégré et bien d'autres.

Avec votre compte étudiant, vous pouvez avoir accès à la totalité de la suite JetBrains professionnelle gratuitement. L'inscription se fait sur ce lien.

Si vous voulez découvrir toute la suite JetBrains, il est recommandé d'installer l'application Toolbox qui permet facilement de gérer et installer tous les IDE de la suite. Sinon vous pouvez installer PyCharm tout seul depuis la page dédiée.

b. Visual Studio Code (VS Code)

VS Code est un éditeur de code léger mais puissant. Il intègre uniquement un support natif pour JavaScript, TypeScript et Node.js mais dispose d'un riche écosystème d'extensions pour d'autres langages et environnements d'exécution (tels que Python, C++, PHP, Go, …). VS Code est entièrement gratuit.

Pour développer en Python sur VS Code, vous allez avoir besoin de l'extension Python officielle, qui permet un support de Python tel que IntelliSense, du linting, du débogage, etc.
Vous pouvez accéder à la vue « Extensions » directement dans VS Code avec « Ctrl + Shift + X » sur Windows / Linux et « command + shift + X » sur macOS.

3. Tester son environnement

Dans votre IDE, créez un nouveau fichier appelé « hello.py » et écrivez-y la ligne « print("Hello, world!") ».

Exécuter le script :

- **PyCharm**, faites un clic droit sur le fichier dans l'explorateur de projets et sélectionnez « Run 'hello.py' ». Vous pouvez également utiliser le raccourci « ctrl + shift + r » si vous avez le fichier ouvert,
- **VS Code**, ouvrez le terminal (« Ctrl + ` »), et en tapant « python3 hello.py »,
- **Terminal**, déplacez-vous dans le dossier contenant votre fichier et tapez la commande « python3 hello.py ».

Partie 1 : Mots clés de base

1. Fonctions et arguments

Prototype : multiply(a: int, b: int) -> int
Rendu : partie1.py

Écrire une fonction qui prend deux entiers et retourne leur produit.

2. Instructions conditionnelles

Prototype : compare(a: int, b: int)
Rendu : partie1.py

Écrire une fonction qui prend deux entiers et vérifie si le premier est supérieur au second.
Le programme doit afficher :

- « Le premier nombre est plus grand que le second » si le premier nombre est plus grand
- « Le premier nombre est plus petit que le second » si le premier est plus petit
- « Les deux nombres sont égaux » s'ils sont égaux

Exemples :

Code de test :

```
compare(1, 1)
compare(2, 1)
compare(-2, 1)
```

Résultat :

```
→  Jour1 git:(main) ✗ python3 partie1.py | cat -e
Les deux nombres sont égaux$
Le premier nombre est plus grand que le second$
Le premier nombre est plus petit que le second$
→  Jour1 git:(main) ✗
```

Indice : « cat -e » permet d'afficher les retours à la ligne avec le signe « $ ». Cela permet de voir si vous avez autant de retour à la ligne qu'attendu mais également vos potentiels espaces blancs en fin de ligne qui ne devraient pas être présents.

3. Boucle For avec Range

Prototype : counting(x: int)
Rendu : partie1.py

Écrire une fonction qui prend un entier « x » et affiche les nombres de 1 à *x*. Les nombres doivent être séparés d'une la séquence ", ". Utiliser une boucle « for » et la fonction « range ». La fonction doit afficher uniquement les nombres impairs.

Indice : Soyez attentifs aux paramètres de « range ».

Avertissement : Vous n'avez pas le droit d'utiliser d'instructions de condition (if-elif-else).

Exemples :

```
counting(0)
counting(10)
counting(11)
```

```
→  Jour1 git:(main) ✗ python3 partie1.py | cat -e
$
1, 3, 5, 7, 9, $
1, 3, 5, 7, 9, 11, $
→  Jour1 git:(main) ✗
```

4. Boucle While avec une condition

Prototype : ask_user()
Rendu : partie1.py

Écrire une fonction qui demande continuellement à l'utilisateur de saisir un mot jusqu'à ce qu'il entre « exit ». Chaque fois qu'un mot est saisi, à l'exception du mot « exit », la fonction doit imprimer « Vous avez entré : {mot} ».

Indice : Depuis Python 3.8, il existe un nouvel opérateur appelé walrus qui permet d'assigner une valeur à une variable au sein d'une expression (conditionnelle par exemple).

Exemples :

```
→  Jour1 git:(main) ✗ python3 partie1.py | cat -e
hello
Vous avez entré : hello$
world
Vous avez entré : world$
1exit
Vous avez entré : 1exit$
EXIT
Vous avez entré : EXIT$
exit exit
Vous avez entré : exit exit$
exit
→  Jour1 git:(main) ✗
```

5. Gestion basique des erreurs

Prototype : safe_divide(a: int, b: int) -> float | None
Rendu : partie1.py

Écrire une fonction qui prend deux entiers et divise le premier par le second. Cette fonction doit gérer la division par zéro en affichant le message d'erreur « Impossible de diviser par zéro » et retourner *None*.

Avertissement : Vous n'avez pas le droit d'utiliser d'instruction de condition (if-elif-else) pour cet exercice.

Indice : Regardez la gestion des erreurs avec Python.

Exemples :

```
print(safe_divide(1, 2))
print(safe_divide(1, 0))
```

```
➜  Jour1 git:(main) ✗ python3 partie1.py | cat -e
0.5$
Impossible de diviser par zéro$
None$
➜  Jour1 git:(main) ✗
```

6. Boucles imbriquées

Prototype : display_square(size: int, char: chr)
Rendu : partie1.py

Écrire une fonction qui prend deux paramètres. La fonction doit afficher un carré de la taille « size » et du caractère « char » donnés.

Exemple 1 :

```
display_square(0, '*')
```

```
➜  Jour1 git:(main) ✗ python3 partie1.py | cat -e
➜  Jour1 git:(main) ✗
```

Exemple 2 :

```
display_square(1, '*')
display_square(5, '*')
```

```
→  Jour1 git:(main) ✗ python3 partie1.py | cat -e
*$
*****$
*****$
*****$
*****$
*****$
→  Jour1 git:(main) ✗
```

Partie 2 : Les chaînes de caractères

1. Concaténation

Prototype : concat_with_space(a: str, b: str) -> str
Rendu : partie2.py

Écrire une fonction qui prend deux chaînes de caractères, les concatènes en mettant un espace entre les deux et retourne la chaîne résultante.

Exemples :

```
print(concat_with_space('Hello', 'World!'))
print(concat_with_space('', ''))
```

```
➜  Jour1 git:(main) ✗ python3 partie2.py | cat -e
Hello World!$
 $
➜  Jour1 git:(main) ✗
```

2. Formatage avec les f-strings

Prototype : format_with_fstring(username: str, age: int) -> str
Rendu : partie2.py

Écrire une fonction qui prend deux paramètres, un nom d'utilisateur et un âge, et retourne la chaîne suivante : "Hello {nom d'utilisateur}, you are {âge} years old!" en utilisant une **f-string**.

Exemples :

```
print(format_with_fstring('Lexie', 27))
print(format_with_fstring('', 0))
```

```
➜  Jour1 git:(main) ✗ python3 partie2.py | cat -e
Hello Lexie, you are 27 years old!$
Hello , you are 0 years old!$
➜  Jour1 git:(main) ✗
```

Partie 3 : Import, modules et packages

Exercice 1 : Do-op

Prototypes :

- *add(a: int, b: int) -> int*
- *subtract(a: int, b: int) -> int*
- *do_op(a: int, b: int, op: chr) -> float | int | None*

Rendu : partie3/*

Suivre les étapes ci-dessous :

- Créer un **package** nommé ***operations***
- À l'intérieur d'*operations*, créer deux fichiers : ***basic_ops.py*** et ***advanced_ops.py***
- Dans *basic_ops.py*, créer deux fonctions :
    - ***add*** qui retourne la somme de deux nombres
    - ***subtract*** qui retourne la différence de deux nombres
- Dans advanced_ops.py, ajouter les fonctions *multiply* et *safe_divide* des exercices 1.1 et 1.5
- Créer un fichier nommé ***main.py*** au même niveau que le package *operations*
- Dans *main.py*:
    - Importer les modules *basic_ops* et *advanced_ops* du package *operations*
    - Créer une fonction ***do_op*** qui prend deux entiers et un caractère parmi ('+', '-', '*', '/') représentant l'opération à effectuer
    - Utiliser les modules/fonctions importées pour réaliser l'opération correspondant au caractère donné et retournez le résultat ou *None* si l'opération ne peut pas être effectuée ou n'est pas reconnue.

Indice : Pensez au fichier *__init__.py !*

Indice : Il existe en Python un mot clé « match » permettant d'éviter de faire de multiple « if » lorsqu'il s'agit d'exécuter un code différent en fonction de la valeur d'une variable.

Exemples :

Code de test dans *main.py* :

```
print(do_op(1, 1, '+'))
print(do_op(1, 1, '-'))
print(do_op(11, 11, '*'))
print(do_op(2, 2, '/'))
print(do_op(2, 0, '/'))
```

```
➜  Jour1 git:(main) ✗ python3 partie3/main.py | cat -e
2$
0$
121$
1.0$
Impossible de diviser par zéro$
None$
➜  Jour1 git:(main) ✗
```

Partie 4 : *main* et point d'entrée

Exercice 1 : Création et utilisation d'un main

Prototypes :

- *greet()*
- *main()*

Rendu : partie4/exercice1.py

Créer un script Python nommé **exercice1.*py***. Dans ce fichier, créer deux fonctions :

- ***greet*** qui affiche "Hello world!"
- **main** qui appelle *greet*.

Faites en sorte de vérifier si le script est l'environnement d'exécution principal (ou point d'entrée) et, si c'est le cas, appeler la fonction *main*, sinon ne faites rien.

Exemple :

```
→  Jour1 git:(main) ✗ python3 partie4/exercice1.py | cat -e
Hello world!$
→  Jour1 git:(main) ✗
```

Dans un fichier *test.py* :

```
import exercice1

print('Exercise 1 is imported!')
```

```
→  Jour1 git:(main) ✗ python3 partie4/test.py | cat -e
Exercise 1 is imported!$
→  Jour1 git:(main) ✗
```

Exercice 2 : Simulation de l'importation de script

Prototypes :

- *display_message()*
- *main()*

Rendu :

- partie4/exercice2/main.py
- partie4/exercice2/other.py

Suivre les étapes ci-dessous :

- Créer deux fichier Python : ***main.py*** et ***other.py***
- Dans *other.py*, définir une fonction nommée ***display_message*** qui affiche "Je suis la fonction display_message venant du module other"
- Créer une fonction ***main*** dans *other.py* qui affiche « Je suis le main du module other » puis appelle la fonction *display_message*. Comme précédemment, la fonction *main* doit être appelée uniquement si *other.py* est le point d'entrée du programme
- Dans *main.py*, importer *other.py* et appeler la fonction *display_message* dans un ***main***

Si vous exécutez *main.py*, seul le texte de la fonction *display_message* sera affiché. Si vous exécutez *other.py*, les textes du main et de la fonction *display_message* seront affichés.

Exemple :

```
→  Jour1 git:(main) ✗ python3 partie4/exercice2/main.py | cat -e
Je suis la fonction display_message venant du module other$
→  Jour1 git:(main) ✗
```

```
→  Jour1 git:(main) ✗ python3 partie4/exercice2/other.py | cat -e
Je suis le main du module other$
Je suis la fonction display_message venant du module other$
→  Jour1 git:(main) ✗
```

Partie 5 : Les différentes structures de données

Dans cette partie, vous allez découvrir les structures de données natives à Python. Elles ont toutes des utilisations bien distinctes que vous découvrirez à travers ces exercices. Vous trouverez toutes les informations dont vous aurez besoin pour réaliser les exercices suivants dans la documentation officielle de Python.

Exercice 1 : Création et manipulation de listes

Prototypes : *list_discovery() -> list*
Rendu : partie5.py

Écrire une fonction *list_discovery* qui utilise les 6 premiers arguments du programme[1] et effectuer les actions suivantes :

- Créer une liste ***numbers*** avec les cinq premiers arguments du programme. Les éléments insérés doivent être du type *int*, attention à bien les convertir
- Trier *numbers* dans l'ordre decroissant (n'implémentez pas votre propre tri !!)
- Supprimer le dernier élément de *numbers*
- Afficher la phrase suivante : « Numbers has {x} elements and the sum of them all is {somme}. »
- Ajouter le sixième argument du programme à la fin de *numbers*
- Retourner *numbers*

Indice : Un programme Python peut prendre des arguments lors de son exécution. Ces arguments sont stockés dans sys.argv sous forme de liste. Attention, tous les arguments sont de type *str.* Vous pouvez utiliser des fonctions de conversion comme float pour modifier le type d'un élément.

Avertissement : Vous n'avez pas le droit d'utiliser de boucles (for, while, …), ni d'instructions de condition pour cet exercice.

Exemple :

```
print(list_discovery())
```

```
→  Jour1 git:(main) ✗ python3 partie5.py 1 2 3 4 5 6 | cat -e
Numbers has 4 elements and the sum of them all is 14.$
[5, 4, 3, 2, 6]$
→  Jour1 git:(main) ✗
```

Exercice 2 : Création et manipulation de dictionnaires

Prototypes :

- *dict_creation() -> dict*
- *dict_display(my_dict: dict)*

Rendu : partie5.py

Écrire une fonction *dict_creation* qui créé un dictionnaire à partir des arguments du programme. 1 argument sur 2 doit être la clé, l'autre la valeur. Le programme prendra forcément un nombre pair d'arguments.

Écrire une fonction *dict_display* prenant en paramètre un dictionnaire et qui effectue les actions suivantes :

- Afficher toutes les clés uniquement, chacune sur une nouvelle ligne
- Afficher toutes les valeurs uniquement, chacune sur une nouvelle ligne
- Afficher toutes les paires clé-valeur sous la forme "Key: {clé} - Value: {valeur}", chacune sur une nouvelle ligne

Avertissement : Vous n'avez pas le droit d'utiliser d'instructions de condition (if-elif-else) pour cet exercice.

Exemple :

```
res = dict_creation()
dict_display(res)
```

```
→  Jour1 git:(main) ✗ python3 partie5.py 1 2 3 4 5 6 | cat -e
1$
3$
5$
2$
4$
6$
Key: 1 - Value: 2$
Key: 3 - Value: 4$
Key: 5 - Value: 6$
→  Jour1 git:(main) ✗
```

Exercice 3 : Création et compréhension des Tuples : collections immuables

Prototypes :

- *tuple_discovery(a, b, c, d) -> tuple*
- *tuple_display(tpl: tuple)*

Rendu : partie5.py

Dans un script, effectuer les tâches suivantes :

- Créer une fonction nommée *tuple_discovery* prenant 4 paramètres. Celle-ci retourne un tuple contenant les paramètres dans l'ordre suivant : d, c, b, a.
- Créer une fonction nommée *tuple_display* prenant un tuple en paramètre et affichant chaque élément sur une nouvelle ligne.

Expérience : Essayer de changer l'un des éléments du tuple et observer le comportement.

Exercice 4 : Création et manipulation d'ensembles

Prototypes : *set_discovery(l1: list, l2: list) -> tuple(set, set, set, set)*
Rendu : partie5.py

Écrire une fonction *set_discovery* qui prend 2 listes en paramètre et effectue les actions suivantes :

- Créé deux ensembles, *set*1 contenant les éléments de la première liste et *set2* ceux de la deuxième liste
- Réalise une opération d'union et stocke le résultat
- Réalise une opération d'intersection et stocke le résultat
- Réalise une opération de différence (set1 - set2) et stocke le résultat
- Réalise une opération de différence symétrique et stocke le résultat
- Retourner le résultat de chaque opération dans un tuple. Les résultats doivent être insérés dans le tuple dans le même ordre que celui ci-dessus.

Expérience : Essayer d'ajouter un élément en double dans l'un des ensembles et observer le résultat.

Indice : Il existe deux méthodes pour réaliser chaque opération. Essayez de les trouver.

Exemple :

```python
l1 = [1, 2, 3, 4, 5]
l2 = [10, 2, 4, 8, 12]
res = set_discovery(l1, l2)
for elem in res:
    print(elem)
```

```
→  Jour1 git:(main) ✗ python3 partie5.py | cat -e
{1, 2, 3, 4, 5, 8, 10, 12}$
{2, 4}$
{1, 3, 5}$
{1, 3, 5, 8, 10, 12}$
→  Jour1 git:(main) ✗
```

Exercice 5 : Les compréhensions de listes

Prototypes : *power_via_comprehension(numbers: list[int]) -> list[int]*
Rendu : partie5.py

Écrire une fonction *power_via_comprehension* qui prend en paramètre une liste de nombres entiers. En utilisant la compréhension de liste, créer une nouvelle liste qui mets au carrés les nombres négatifs et multiplie par -1 les nombres positifs.

Note : La compréhension de liste est un sucre syntaxique très pratique et répandu en Python. Mais il faut savoir que celle-ci n'est pas limitée aux listes, vous pouvez également faire des compréhensions de dictionnaires et d'ensembles. Il suffit de changer les crochets '[]' en accolades '{}'. Vous pouvez retrouver des exemples dans la documentation fournie au début de cette partie.

Exemple :

```python
l = [1, -2, -3, 4, -5]
print(power_via_comprehension(l))
```

```
→  Jour1 git:(main) ✗ python3 partie5.py | cat -e
[-1, 4, 9, -4, 25]$
→  Jour1 git:(main) ✗
```

Partie 6 : Fonctions natives pour manipuler les différentes structures de données

Maintenant que vous maitrisez les bases des différentes structures de données, il est temps d'apprendre diverses façon de les manipuler.
Dans les exercices suivants, le but n'est pas que vous réinventiez la roue. Tous les exercices ont une solution sous forme d'une ou plusieurs fonctions native à Python. On s'attend donc à ce que vous trouviez et utilisiez ces fonctions afin d'effectuer les diverses manipulations demandées. À nouveau, vous pourrez trouver toutes les informations dont vous aurez besoin dans la documentation officielle de Python.

Exercice 1. Obtenir l'index et l'élément d'une structure de données

Prototypes : *struct_index_display(elems)*
Rendu : partie6.py

Créer une fonction *struct_index_display,* prenant en paramètre une structure de données. Itérer sur cette structure et afficher chaque élément sur une nouvelle ligne dans la forme "{elem_id} = {elem}".

Exemples :

```
struct_index_display([1, 2, 3, 4, 5])
```

```
→  Jour1 git:(main) ✗ python3 partie6.py | cat -e
0 = 1$
1 = 2$
2 = 3$
3 = 4$
4 = 5$
→  Jour1 git:(main) ✗
```

Exercice 2. Itérer sur plusieurs structures de données en même temps

Prototypes :

- *combine_lists(l1: list, l2: list) -> None | list*
- *display_combined_lists(l: list)*

Rendu : partie6.py

Créer une fonction *combine_lists* prenant deux listes de longueur égale en paramètre :

- Si les listes ne sont pas de longueur égale, retourner None
- Si les listes sont de longueur égale, retourner une liste de tuple sous la forme : [(elem**1**_**l1**, elem**1**_**l2**), (elem**2**_**l1**, elem**2**_**l2**), …. (elem**N**_**l1**, elem**N**_**l2**)]

Créer une fonction *display_combined_lists* qui itère sur les éléments provenant de la fonction *combine_lists* et affiche chaque élément sur une nouvelle ligne sous la forme : "{elem_id} = {elem1} - {elem2}".

Exemples 1 :

```
res = combine_lists([1, 2, 3], [4, 5, 6])
print(res)
res = combine_lists([1, 2], [4, 5, 6])
print(res)
```

```
→  Jour1 git:(main) ✗ python3 partie6.py | cat -e
[(1, 4), (2, 5), (3, 6)]$
None$
→  Jour1 git:(main) ✗
```

Exemple 2 :

```
res = combine_lists([1, 2, 3], [4, 5, 6])
display_combined_lists(res)
```

```
→  Jour1 git:(main) ✗ python3 partie6.py | cat -e
0 = 1 - 4$
1 = 2 - 5$
2 = 3 - 6$
→  Jour1 git:(main) ✗
```

Exercice 3. Modifier les éléments d'une structure de données

Prototypes :

- *convert_to_string(numbers: list[float]) -> list[str]*
- *multiply_numbers(l: list[int], multiplicator) -> list[int]*

Rendu : partie6.py

Créer une fonction *convert_to_string* prenant en paramètre une liste de nombres décimaux et retournant ces nombres convertis en string.

Créer une fonction *multiply_numbers* prenant en paramètre une liste de nombres entiers et un multiplicateur. Multipliez chaque nombre de la liste par le multiplicateur donné.

Note : Vous avez le droit de créer des fonctions supplémentaires pour réaliser cet exercice.

Indice : Vous pouvez créer une fonction anonyme avec une lambda pour simplifier la lecture de votre code.

Indice 2 : Certaines fonctions natives retournent des objets de type *iterable*. Cependant, ces objets ne sont pas forcément d'un des types de base (*list* ou *tuple* par exemple). Vous pouvez néanmoins convertir tout objet du type *iterable* vers un type équivalent avec une fonction comme *set*.

Exemple :

```
t = convert_to_string([1.5, 2.4, 3, 4.2])
print(t)
t = multiply_numbers([1, 2, 3, 4, 5, 6], 3)
print(t)
```

```
→  Jour1 git:(main) ✗ python3 partie6.py | cat -e
['1.5', '2.4', '3', '4.2']$
[3, 6, 9, 12, 15, 18]$
→  Jour1 git:(main) ✗
```

Exercice 4. Filtrer les éléments contenus dans une structure de données

Prototypes :

- *remove_negatives(numbers: list[float]) -> list[float]*
- *keep_strings(elements: list) -> list[str]*

Rendu : partie6.py

Créer une fonction *remove_negatives* prenant en paramètre une liste de nombres décimaux. La fonction doit retourner une nouvelle liste sans aucun nombre négatif.

Créer une fonction *keep_strings* prenant en paramètre une liste contenant plusieurs types de données (int, float, str, list, …) et retournant une nouvelle liste qui contient uniquement les éléments de type "str".

Note : Vous avez le droit de créer des fonctions supplémentaires pour réaliser cet exercice.

Indice : type est une fonction qui vous permets de connaitre le type de n'importe quelle variable.

Exemple :

```
t = remove_negatives([-1, 2, 3, 4, -5, 6, 7])
print(t)
t = keep_strings(['Hello', 1, 3, "spam", 5.5, (1, 2)])
print(t)
```

```
→  Jour1 git:(main) ✗ python3 partie6.py | cat -e
[2, 3, 4, 6, 7]$
['Hello', 'spam']$
→  Jour1 git:(main) ✗
```

Exercice 5. Récupérer un sous-ensemble de données

Prototypes : *cut_in_two(numbers: list[float]) -> (list[float], list[float])*
Rendu : partie6.py

Écrire une fonction *cut_in_two* prenant en paramètre une liste de nombres décimaux. Votre fonction doit séparer cette liste en deux nouvelles listes de même longueur. Vous devez ensuite retourner ces deux nouvelles listes dans un tuple.
Si la liste possède un nombre d'éléments impair, la deuxième liste retournée doit être la plus grande des deux.

Indice : Vous pouvez utiliser le slicing pour découper tout *iterable* (str, list, tuple, …) en fonction d'indexes de début, de fin et de pas.

Indice 2 : Il existe deux types de divisions en Python, la division flottante (true division) et la division entière (floor division).

Avertissement : Vous n'avez pas le droit d'utiliser de boucles (for, while, …), ni d'instructions de conditions (if, elif, else, …) pour cet exercice.

Exemple :

```
t = cut_in_two([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(t)
```

```
→  Jour1 git:(main) ✗ python3 partie6.py | cat -e
([1, 2, 3, 4], [5, 6, 7, 8, 9])$
→  Jour1 git:(main) ✗
```

Partie 7 : Manipulations de chaînes de caractères

Note : Documentation des méthodes de la classe str

Vous devez effectuer les exercices de cette partie en utilisant uniquement les fonctions présentes dans la documentation ci-dessus. Vous êtes autorisés à utiliser les mots-clés « if / else » uniquement, les autres sont interdits (for, while, try, elif, …).

Exercice 1 : Séparations et fusions

Prototypes : *change_separator(base_str: str, split_str: str, join_str: str) -> str*
Rendu : partie7.py

Effectuer les tâches suivantes :

- Créer une fonction *change_separator* prenant en paramètre trois chaînes de caractères (base_str, split_str, join_str)
- La chaîne *split_str* contient un délimiteur. Vous devez l'utiliser pour découper la chaîne *base_str* en autant de morceaux que d'occurrences du délimiteur
- Retirer le premier et dernier morceaux grâce au slicing (vous n'avez pas besoin de gérer les erreurs)
- Utiliser ensuite la chaîne *join_str* pour fusionner les morceaux en une seule chaîne.
- Retourner la chaîne résultante

Exemple :

```
print(change_separator(
    'Hello, world, my name is Leah, how are you ?',
    ', ',
    ' - ')
)
```

```
→  Jour1 git:(main) ✗ python3 partie7.py | cat -e
world - my name is Leah$
→  Jour1 git:(main) ✗
```

Exercice 2 : Recherche de sous-chaînes

Prototypes : *sub_index(base_str: str, sub_str: str) -> str*
Rendu : partie7.py

Écrire une fonction *sub_index* qui prend en paramètre deux chaînes de caractères. Chercher la première occurrence de *sub_str* dans *base_str :*

- Si aucune occurrence n'est trouvée, retourner la chaîne *base_str*
- Si une occurrence est trouvée, retourner le reste de la chaîne *base_str* après la fin de la première occurrence.

Indice : slicing

Exemple :

```
print(sub_index(
    'Hello, world, my name is Leah, how are you ?',
    'my name is ')
)
```

```
→  Jour1 git:(main) ✗ python3 partie7.py | cat -e
Leah, how are you ?$
→  Jour1 git:(main) ✗
```

Exercice 3 : Remplacement de sous-chaînes

Prototypes : *replace_str(base_str: str, sub_str: str) -> str*
Rendu : partie7.py

Écrire une fonction *replace_str* qui remplace chaque occurrence de *sub_str* dans *base_str* par la chaîne "Egg and Spam".

Exemple :

```
print(replace_str(
    'Hello Leahh, my name is Leah, how are you ?',
    'Leah')
)
```

```
→  Jour1 git:(main) ✗ python3 partie7.py | cat -e
Hello Egg and Spamh, my name is Egg and Spam, how are you ?$
→  Jour1 git:(main) ✗
```

Exercice 4 : Conversion de casse

Prototypes : *normalize_input(base_str: str) -> (str, str)*
Rendu : partie7.py

Écrire une fonction *normalize_input* qui prend en paramètre une chaîne de caractère et retourne un *tuple* contenant la chaîne tout en minuscules et la chaîne tout en majuscules.

Exemple :

```
print(normalize_input('Hello LeAh, hOW aRE YOu ?'))
```

```
→  Jour1 git:(main) ✗ python3 partie7.py | cat -e
('hello leah, how are you ?', 'HELLO LEAH, HOW ARE YOU ?')$
→  Jour1 git:(main) ✗
```

Exercice 5 : Suppression des espaces blancs

Prototypes : *remove_white_spaces(base_str: str) -> (str, str, str)*
Rendu : partie7.py

Écrire une fonction *remove_white_spaces* qui prend en paramètre une chaîne de caractères et retourne un *tuple* contenant 3 chaînes. La première est la chaîne en paramètre sans aucun espace blanc avant et après le texte, la deuxième n'a aucun espace blanc avant le texte et la troisième n'a aucun espace blanc après le texte.

Indice : Les espaces blancs ne sont pas forcément uniquement composés du caractère " " (espace / ASCII 0x20).

Exemple :

```
print(remove_white_spaces('      \t\tMy name is Leah.\t\t\t   \t'))
```

```
→  Jour1 git:(main) ✗ python3 partie7.py | cat -e
('My name is Leah.', 'My name is Leah.\t\t\t   \t', '      \t\tMy name is Leah.')$
→  Jour1 git:(main) ✗
```

Going Further : Récapitulatif et Regex

Lors de cette piscine, vous trouvez à la fin de certains jours une partie appelée "Going Further". Celle-ci est destinée à celles et ceux qui souhaitent approfondir leur compréhension et se mettre au défi au-delà du programme de base. Ces exercices sont facultatifs et non évalués, bien que certains puissent faire partie des tests automatisés en fonction de la faisabilité technique.

Il est important de noter que ne pas réaliser cette partie n'affectera en rien votre capacité à progresser lors des jours suivants, ni vos compétences fondamentales à acquérir d'ici à la fin de la piscine. Ces exercices sont là pour enrichir votre expérience d'apprentissage, si vous choisissez de relever ce défi supplémentaire.

Exercice 1 : Formatage d'IP

Prototypes : *fix_ip (ip: str) -> str*
Rendu : going_further.py

Écrire une fonction fix_*ip* qui prend en paramètre une chaîne de caractère contenant une potentielle adresse IPV4. Vous devez gérer les erreurs et corriger le formatage au besoin. La fonction retourne l'adresse correctement formattée si elle est valide, sinon None.

Pour simplifier, on considère une adresse IPV4 valide si elle contient 4 nombres compris 0 et 255 inclus, séparés un point. Par exemple :

- « 0.0.0.0 », « 156.183.193.1 » sont valides
- « 010.184.00242.056 » est mal formatée mais peut être rendue valide
- « 000010.00000.0295.10 » est mal formatée et invalide
- « 0.0.0.324 » et « 0.0.0.0.0 » sont invalides

Note : Essayer de réutiliser un maximum de fonctions que vous avez vu tout au long de cette journée et fouillez la documentation pour en trouver d'autres qui pourraient vous être utiles. Ne réinventez pas la roue ! Vous n'avez pas besoin d'utiliser de boucles (for, while) pour cet exercice si vous utilisez les bonnes méthodes.

Exemples :

```
print(fix_ip('0.0.0.0'))
print(fix_ip('156.183.193.1'))
print(fix_ip('010.184.00242.056'))
print(fix_ip('000010.00000.0295.10'))
print(fix_ip('0.0.0.324'))
print(fix_ip('0.0.0.0.0'))
```

```
→  Jour1 git:(main) ✗ python3 going_further.py | cat -e
0.0.0.0$
156.183.193.1$
10.184.242.56$
None$
None$
None$
→  Jour1 git:(main) ✗
```

Exercice 2 : Les bases des Regex

Les expressions régulières, ou regex, sont un outil puissant pour identifier des patterns dans un texte. Elles vous permettent de spécifier un pattern de recherche, de trouver des séquences spécifiques de caractères et d'effectuer des manipulations de texte sophistiquées. En apprenant à utiliser les regex, vous pouvez rechercher, remplacer ou valider du texte de manière efficace, ce qui en fait une compétence essentielle pour analyser et traiter de grandes quantités de données textuelles.

Vous pouvez créer et tester vos regex sur le site regex101 avant d'écrire le code Python associé. Pensez à bien sélectionner les regex (« Flavor ») Python sur le site pour être sûr d'avoir la bonne syntaxe.
Python gère nativement la création de regex avec son module re.

Prototypes :

- *simple_pattern_search(phrase: str) -> bool*
- *extract_emails(phrase: str) -> list[str]*
- *extract_hashtags(message: str) -> list[str]*
- *split_on_pattern(message: str) -> list[str]*
- *date_conversion(text: str) -> str*

Rendu : going_further.py

Partie 1 : Correspondance simple

Écrire une fonction *simple_pattern_search* qui prend en paramètre une chaîne de caractère et vérifie si le mot "eggs" suivi d'un nombre de taille quelconque y est contenu. Exemple : « I love eggs203, can I have some please ? ». La fonction retourne "True" si le mot est présent, "False" sinon.

Partie 2 : Extraction d'adresses mail

Écrire une fonction *extract_emails* prenant en paramètre une chaîne de caractères et retournant une liste d'emails contenus dans celle-ci.

Exemple: "Please contact us at support@example.com or sales@example.org."

Note : Un pattern d'adresse mail peut être simplifié comme "des caractères, un symbole @, des caractères, un point, des caractères".

Partie 3 : Trouver tous les Hashtags dans un message de réseau social

Écrire une fonction *extract_hashtags* qui prend en paramètre un message et retourne une liste de hashtags contenu dans celui-ci. Un hashtag est n'importe quel mot qui commence par un `#` et est suivi de caractères alpha-numériques.

Exemple : "Loving the #Python and #regex exercises from #EDS #DataPoolDigi2!"

Partie 4 : Division de texte basée sur un pattern délimiteur

Écrire une fonction *split_on_pattern* qui prend en paramètre une phrase et la sépare pour toute occurrence du pattern suivant : un nombre de taille quelconque, suivi d'un point et d'un retour à la ligne "\n".

Exemple : « Today, Leah is 25 years old. Meaning she was born in 1994.\nMaybe I have to check my maths ! »

Partie 5 : Conversion de formats de dates

Écrire une fonction *date_conversion* qui prend un paramètre un texte contenant des dates sous la forme yyyy-mm-dd. Votre rôle est de convertir ces dates sous la forme dd/mm/yyyy sans perte d'informations. Votre fonction doit retourner le texte résultant.

Exemple : « Today's date is 2024-01-15 » devient « Today's date is 15/01/2024 ».

Indice : Capturer le jour, le mois et l'année dans des groupes séparés et les réarranger dans le pattern de remplacement.