Challenges Python

(Discord « Docstring »)

@bucdany

2023

Table des matières

Avant propos

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Challenge N° 1

Compter le nombre de voyelles

1.1 Énoncé

Ce premier challenge est très simple, il est de niveau « débutant », mais si vous avez plus d'expérience, vous pouvez essayer de trouver de belles astuces pour un code propre, rapide et concis.

Ici, il va nous falloir créer une fonction `nb_voyelles(phrase: str)->int` qui retourne le résultat du nombre total de voyelles dans une phrase passée en paramètre.

Conditions

- Les voyelles sont : `aeiou`, `y` n'est pas pris en compte.
- Les voyelles accentuées ne sont pas prises en compte.
- La phrase passée en paramètre doit être écrite en minuscule.
- Une chaîne vide, passée en paramètre, doit renvoyer `0`.

Exemples

- `nb_voyelles("bonjour, comment allez-vous ?")` doit retourner `9`.
- `nb_voyelles("je vais à paris")` doit retourner `5`.
- `nb_voyelles("docstring")` doit retournet `2`.
- `nb_voyelles("")` doit retourner `0`.

1.2 Solution et explications

Voici donc ma solution [1] :

```
def nb_voyelles(phrase: str)->int:
    return sum(phrase.count(el) for el in "aeiou")
```

- La phrase doit toujours être en minuscule, donc pas besoin de la méthode `lower()`.
- Ici, on compte chaque voyelle dans la phrase à l'aide de la méthode `count`.
- La fonction `sum()` renvoie ensuite la somme du résultat obtenu.
- Si l'on passe un générateur ou une liste de compréhension dans la fonction `sum()`, la paire de crochets supplémentaire peut-être éliminée [2]. De cette manière :

  `sum([phrase.count(el) for el in "aeiou"])`

  est l'équivalent de :

  `sum(phrase.count(el) for el in "aeiou")`

Voici aussi le code pour mes tests unitaires :

```
import pytest

@pytest.mark.parametrize("sentence, expected", [
    ("", 0),
    ("docstring", 2),
    ("bonjour comment allez-vous ?", 9),
    ("je vais à paris", 5),
    ("vas-y !", 1),
])
def test_should_return_the_sum(sentence, expected):
    got = nb_voyelles(sentence)
    assert got == expected
```

`@OsKaR31415` a par ailleurs apporté plusieurs solutions pour résoudre ce challenge [3].

1. Fil de discussion de ce challenge: https://discord.com/channels/396825382009044994/1142617945139335189
2. Attention, car cela n'est par contre pas compatible avec la fonction `len()`. Pour plus d'information on se reportera au PEP-289: https://peps.python.org/pep-0289/#the-details.
3. https://discord.com/channels/396825382009044994/1142617945139335189/1144591818516860998

Challenge N° 2

Jeu du « Pierre - Papier - Ciseaux »

2.1 Énoncé

On va jouer un peu en développant un petit jeu très simple.

Le but du challenge est de développer le célèbre jeu « *pierre - papier - ciseaux* »[1] en essayant de trouver un algorithme astucieux et un code à la fois simple, propre et efficace.

Étapes

1. Générer un choix aléatoire pour votre session de jeu : « pierre », « papier » ou « ciseaux ».
2. Demander au joueur d'écrire son choix entre trois propositions : « pierre », « papier » ou « ciseaux ».
3. Afficher qui a gagné en dévoilant le choix aléatoire du point n°1.

Conditions

- L'affichage, le prompt et la réponse seront affichées par écrit sur la console.
- Le fonctionnement du jeu est simple : la pierre gagne sur les ciseaux, les ciseaux gagnent sur le papier, le papier gagne sur la pierre, deux éléments identiques correspondent à une égalité.
- Toutes les chaînes de caractères, « pierre », « papier » et « ciseaux » doivent toujours être entrées en minuscule, le joueur devra donc écrire correctement ces mots, sinon vous devrez lui demander de redéfinir son choix.
- S'il y a égalité, vous devrez relancer automatiquement votre programme (en regénérant un nouveau choix aléatoire pour la nouvelle session de jeu), jusqu'à ce qu'il y ait un gagnant à la partie.

1. https://fr.wikipedia.org/wiki/Pierre-papier-ciseaux

Exemples

- Le choix aléatoire donne « pierre » et le joueur a choisi « papier » -> `Vous avez gagné ! Le papier enveloppe la pierre`
- Le choix aléatoire donne « ciseaux » et le joueur a choisi « papier » -> `Vous avez perdu ! Les ciseaux coupent le papier`
- Le choix aléatoire donne « pierre » et le joueur a choisi « pierre » -> `Égalité ! Recommencez...`

2.2 Ma solution

```
from random import randint

BDD = {
    "element": ["papier", "pierre", "ciseaux"],
    "gagnant": ["10", "21", "02"],
    "phrase": ["Le papier enveloppe la pierre",
               "Les ciseaux coupent le papier",
               "La pierre casse les ciseaux"]
}

while True:
    joueur = input("pierre, papier ou ciseaux: ")
    if joueur in BDD["element"]:
        choix, joueur = (randint(0, 2),
                         BDD["element"].index(joueur))

        if choix != joueur: break
        print("Ègalité, recommencez...")

    else: print("Faute de frappe !")

print(f'Vous avez {"gagné" if f"{choix}{joueur}" in BDD["g
agnant"] else "perdu"} ! {BDD["phrase"][choix + joueur - 1]
} !')
```

2.3 Explication de l'algorithme

Contexte

- Trois éléments : « pierre », « papier » ou « ciseaux ».
- Un choix aléatoire fait par la machine et un joueur qui entre son choix au clavier.

Il suffit donc de réfléchir à un algorithme sympathique pour présenter le code de manière élégante et éviter bien sûr les répétitions.

Rangement des données

Chaque élément est rangé dans cet ordre particulier dans la liste, afin de les faire correspondre en triade logique.

- papier = index(0)
- pierre = index(1)
- ciseaux = index(2)

Algorithme pour un choix triangulaire

Si on fait l'addition `0+1`, on obtient `1`, alors le jeu se fait entre « papier » et « pierre ». On cherche ensuite la phrase dans `phrase` en faisant juste un calcul grâce à la somme `-1` des deux éléments. Donc en index: `1-0 = 0`, et on trouve donc la chaîne de caractères `"Le papier enveloppe la pierre"`.

Si on fait l'addition `1+2`, on obtient `3`, alors le jeu se fait entre la « pierre » et les « ciseaux ». Donc en index, on obtient `3-1`, soit `2`. On trouve donc la chaîne de caractères `"La pierre casse les ciseaux"`.

De la même façon, si on fait l'addition `2+0`, on obtient `2` et le jeu se fait entre les « ciseaux » et le « papier ». En index cela donne `2-1`, soit `1`, et on tombe sur la chaîne de caractères `"Les ciseaux coupent le papier"`.

The Winner is...

Pour connaître qui gagne, il suffit de convertir en *string* et de joindre les deux caractères d'index du choix et du joueur. Ainsi, `10` dans `gagnant` veut dire que le choix aléatoire donne la « pierre » (index `1`) et que le joueur a saisi le « papier » (index `0`). La « pierre » contre le « papier » fait donc gagner le joueur.

On affiche ainsi `"gagné"` puis la phrase qui suit s'obtient grâce à l'index de la liste `phrase` de la `BDD`, calculée par l'addition des deux index: `1+(0-1)`, ce qui nous donne `0`, ce qui correspond à la chaîne de caractères `"Le papier enveloppe la pierre"`.

De la même façon pour `21` et `02`, cela représente la combinatoire complète des choix gagnants pour le joueur par rapport au choix aléatoire.

Conclusion

On utilise le calcul de la somme `-1` qui renvoie un objet de type `int` et qui pemret l'association des deux chaînes de caractères (*string*) pour connaître le gagnant.

Challenge N° 3

Couleur complémentaire

3.1 Énoncé

Pour ce challenge j'ai choisi un niveau intermédiaire. Les débutants pourront cependant résoudre la première étape en s'aidant de librairies.

Le but de ce challenge est de trouver la **couleur complémentaire** [1]

Étapes

1. Créer la fonction `get_color_types(color:str)->dict` qui permet de convertir le format *RVB hexadécimal* d'une couleur au format *RVB décimal* et *TSL* [2] (anglais : ***H**ue**L**ight**S**aturation*).
   - **color : [string]** : la couleur RVB codée en hexadécimal, envoyée en paramètre.
   - **dict : [dict]** : contient le résultat de la conversion en différents styles d'écriture, contenant les clés et valeurs suivantes :

     hex : [str] : valeur hexadécimale de la couleur passée en paramètre.

     rvb : [list] : valeurs de chaque éléments RVB en décimal.

     tsl_norm : [tuple] : valeurs de chaque élément TSL (teinte en degrés (360°), saturation et luminosité en pourcentage).

     tsl : [tuple] : valeurs de chaque élément TSL (teinte, saturation et luminosité au format [0-1], soit de type `float`).

2. Afficher le contenu du dictionnaire retourné par cette fonction.

3. Créer la fonction `get_complementary(color:str)->str` pour trouver la couleur complémentaire et la retourne au format hexadécimal.

1. https://fr.wikipedia.org/wiki/Couleur_complÃl'mentaire
2. https://fr.wikipedia.org/wiki/Teinte_saturation_lumiÃÍre

Conditions

- L'affichage se fera via la console.
- Les valeurs hexadécimales sont précédées du symbole « # » et les lettres sont en minuscules.

Exemples

```
- get_color_types("#19021e") -> {'hex': '#19021e', 'rvb': [25, 2, 30],
                                 'tsl_norm': ('289°', '88%', '6%'),
                                 'tsl': (0.8035714285714285, 0.875,
                                         0.06274509803921569)}
- get_complementary("#19021e") -> "#071e02"
```

Ressource

Vous pouvez vous aider du site `colorpicker`[3] pour vos tests.

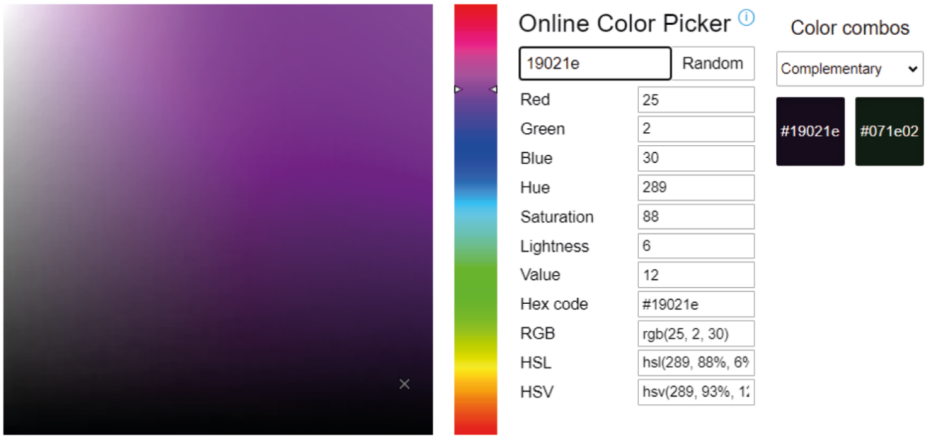

FIGURE 3.1 – `colorpicker`

Indices

- Le nombre hexadécimal d'une couleur représente ses valeurs *RougeVertBleu* codées avec six nombres. Les deux premiers correspondent à la couleur rouge, les deux suivants au vert et les deux derniers au bleu. Pour transformer cette valeur hexadécimale en décimal, il vous suffit de convertir chacune des paires de ce nombre.
- Vous pouvez vous aider de la librairie `colorsys` pour vous permettre de réaliser facilement les conversions.
- Ce sont principalement les fonctions `colorsys.rgb_to_hls` et `colorsys.hls_to_rgb` qui pourront être utilisées.

3. https://colorpicker.me/#00ee7b

- Pour chercher la complémentaire d'une couleur, il faut passer par le format *TSL* en faisant une rotation de 180° sur la teinte et trouver ainsi la position de la couleur diamétralement opposée.
- Pour faire une rotation, il faudra bien sûr penser à normaliser la valeur de la teinte *TSL* qui par défaut est de `[0, 1]` en `[0°, 360°]` puis faire la rotation en additionnant avec 180°.

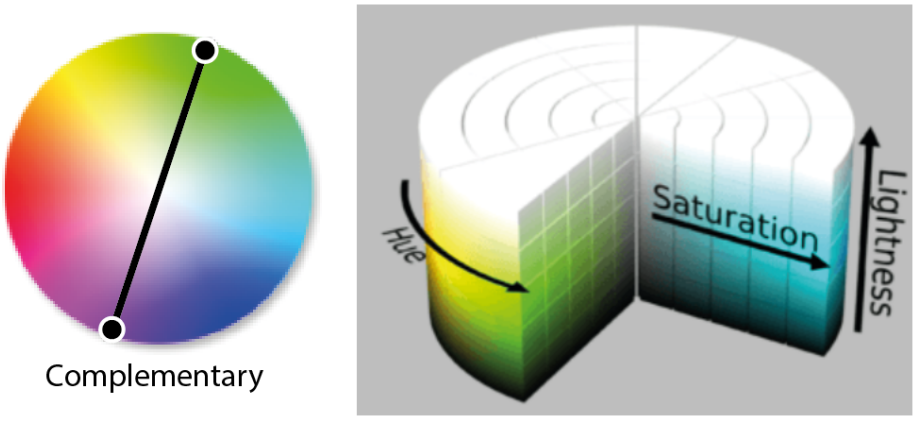

FIGURE 3.2 – Couleur complémetaire et TSL

3.2 Mon code

```python
from colorsys import rgb_to_hls, hls_to_rgb


def get_color_types(color: str) -> dict:
    rvb = [int(color[i:i+2], 16) for i in
           range(1, len(color), 2)]
    tsl = list(rgb_to_hls(*list(map(lambda x: x/255,
                                    rvb))))
    tsl.insert(-1, tsl.pop())
    tsl_norm = (f"{round(tsl[0]*360)}°",
                *[f"{el:.0%}" for el in tsl[1:3]])
    return {"hex": color, "rvb": rvb,
            "tsl_norm": tsl_norm, "tsl": tuple(tsl)}


def get_complementary(color: str) -> str:
    t, s, l = get_color_types(color)["tsl"]
    return f'#{"".join([f"{round(el*255):02x}" for el
                        in hls_to_rgb(t+.5 % 1, l, s)])}'
```

3.3 Explications

Contexte

Il s'agit soit de faire tous les calculs à la main pour les conversions entre les formats de couleur, soit trouver une bibliothèque qui nous permet de nous simplifier la vie. Mon choix s'est porté sur la bibliothèque `colorsys` qui est plutôt sympathique pour cela. Elle nous permet de passer au format `TSL` via `RGB` et vice et versa.

Le but est aussi de trouver la couleur complémentaire, qui est la couleur diamétralement opposée sur le cercle chromatique. Pour cela `TSL`, nous permet de faire une rotation et trouver directement cette valeur sans difficulté particulière.

Différentes problématiques

- **Le dièze « `#` »... gênant hein !** : il y a donc plusieurs façon de faire, soit un simple `lstrip()` et hop viré, soit débuter son index à `1`.
- **Conversion d'un format hexadécimal à un format décimal pour le `RVB`** : un simple `int(..., 16)` nous fait le calcul directement !
- **Normaliser ses valeurs** : avec la bibliothèque `colorsys`, attention à toujours rester dans un domaine de valeurs comprises entre `0` et `1`, comme stipulé dans la documentation officielle de `colorsys`[4].
- **`HLS` vs `TSL`** : attention à bien penser à inverser la saturation de la luminosité si vous utilisez la bibliothèque `colorsys`.
- ***Tuple*, *liste*** : l'énoncé précise de bien respecter les différents types dans le dictionnaire renvoyé par la fonction.
- **Utilisation des fonctions déjà crées** : on a déjà développé la fonction `get_color_type()` alors pourquoi ne pas l'utiliser pour créer la fonction `get_complementary()` afin d'avoir directement la valeur `TSL`.
- **Trouver la valeur diamétralement opposée** : une rotation de `180°` ou de `0.5` dans la plage de limites [`0` - `1`] puis un *modulo* de `1` à appliquer pour rester dans cette plage de valeurs.

Explication du code et astuces d'optimisation / *refactoring*

```
1 rvb = [int(color[i:i+2], 16) for i in
2             range(1, len(color), 2)]
```

On pointe avec un `range` sur la chaîne de caractères `color` avec un index de `1` jusqu'à sa taille totale suivant un pas de `2`, puis avec `[i:i+2]` on vient chercher deux par deux les caractères hexadécimaux de chaque couleur à convertir en décimal.

4. https://docs.python.org/3/library/colorsys.html : « `the coordinates are all between 0 and 1...` »

```
tsl = list(rgb_to_hls(*list(map(lambda x: x/255, rvb))))
```

Pour constituer notre `TSL`, on utilise la fonction `colorsys.rgb_to_hls()`. J'ai volontairement utilisé la fonction `map()` pour vous permettre de voir de nouvelles choses en plus de la compréhension de liste. Ceci `list(map(lambda x: x/255, rvb))` est la même chose que cela : `[x/255 for x in rvb]`. C'est juste une autre façon de faire ! Donc on vient diviser chaque couleur `R`, `V` et `B` par `255` pour respecter la norme `0` à `1` qui sera nécessaire pour l'argument de la fonction `rgb_to_hls()`. Notez l'astérisque (`*`) qui permet de faire un *unpacking* pour dispatcher les éléments `RVB` en trois arguments, la fonction `rgb_to_hls()` nécessitant trois arguments pour fonctionner. Et l'on convertit en `list()` car nous avons ensuite besoin d'intervertir les deux derniers éléments, luminosité et saturation étant inversés par rapport à l'énoncé du challenge.

```
tsl.insert(-1, tsl.pop())
```

C'est la phrase magique qui permet d'inverser les places des deux derniers éléments de la liste. `pop()` permet de retirer le dernier élément, `insert()` permet d'insérer ce que `pop()` renvoie à l'index `-1`, qui est l'avant dernière position... `-1` étant la dernière position mais l'insertion se passe juste avant la position indiquée, donc juste avant `-1`. La liste `tsl` sera alors modifiée directement.

```
tsl_norm = (f"{round(tsl[0]*360)}°",
                *[f"{el:.0%}" for el in tsl[1:3]])
```

On utilise le résultat de la conversion `TSL` pour obtenir `TSL_norm`:

- La teinte est définie en degrés, comprise dans une limite entre 0 et 360°. `round()` permet d'arrondir pour avoir une valeur sans chiffre après la virgule.
- La luminosité et la saturation quant à elles sont exprimées en pourcentage grâce à l'utilisation d'une *f-string* qui avec « `%` » permet à la fois de multiplier par `100` et d'afficher le pourcentage. Le `.0` permet de supprimer les chiffres après la virgule. `tsl[1:3]` c'est la lecture des élément `1` et `2` de la liste `tsl`. On utilise encore l'astérisque (`*`) pour avoir en tout trois arguments pour `tsl_norm`.

```
return {"hex": color, "rvb": rvb,
            "tsl_norm": tsl_norm, "tsl": tuple(tsl)}
```

Juste un simple `return` pour renvoyer un dictionnaire. Notez le `tuple()` qui permet de convertir notre précédente liste en tuple afin de suivre l'énoncé du challenge.

```
t, s, l = get_color_types(color)["tsl"]
```

On envoie le résultat *TSL* du dictionnaire de la fonction précédente dans les variables `t`, `s` et `l`. Il est important d'utiliser `tsl` plutôt que `tsl_norm` afin de travailler sur des valeurs non arrondies et avoir ainsi plus de précision.

```
return f'#{"".join([f"{round(el*255):02x}" for el
                        in hls_to_rgb(t+.5 % 1, l, s)])}'
```

Ce `return` peut paraître un peu complexe mais tout deviendra simple une fois décomposé :

```
t+.5 % 1
```

On effectue une rotation de la teinte de `180°` mais comme on se trouve dans un espace compris entre 0 à 1, alors cela devient `180/360 = 1/2 = 0.5`. Pour rester dans la norme, on fait un *modulo* `1`.

```
hls_to_rgb(t+.5 % 1, l, s)
```

On convertit en *RVB* via *TSL*, avec la teinte, la luminosité et la saturation.

```
[f"{round(el*255):02x}" for el in hls_to_rgb(t+.5 % 1, l, s)]
```

Cette compréhension de liste nous fournit le *RVB* au format hexadécimal. Il faut pour ça changer la norme et passer de `[0-1]` à `[0-255]` en multipliant par `255` puis convertir en hexadécimal via `02x`, ce qui nous donne deux chiffres (les valeurs de 0 à 9 seront écrites à l'aide de deux chiffres : 01 02 03 04... 09), `x` permet la conversion grâce à une *f-string*.

```
f'#{"".join([f"{round(el*255):02x}" for el
                        in hls_to_rgb(t+.5 % 1, l, s)])}'
```

`"".join()` permet de réunir le résultat de chaque paire *RVB*/*hexadécimal* et enfin le « # », la cerise sur le gâteau !

Mon fichier d'*unitests*

```
import pytest

@pytest.mark.parametrize("hexcode, expected", [
    ("#19021e", {'hex': '#19021e', 'rvb': [25, 2, 30],
                 'tsl_norm': ('289°', '88%', '6%'),
                 'tsl': (0.8035714285714285, 0.875,
                         0.06274509803921569)}),
    ("#aedd5f", {'hex': '#aedd5f', 'rvb': [174, 221, 95],
```

```
                  'tsl_norm': ('82°', '65%', '62%'),
                  'tsl': (0.22883597883597884,
                          0.6494845360824743,
                          0.6196078431372549)}),
    ("#ff0000", {'hex': '#ff0000', 'rvb': [255, 0, 0],
                  'tsl_norm': ('0°', '100%', '50%'),
                  'tsl': (0.0, 1.0, 0.5)})
])
def test_different_colour_writing_formats(
        hexcode, expected):
    got = get_color_types(hexcode)

    assert got == expected

@pytest.mark.parametrize("hexcode, expected", [
    ("#19021e", "#071e02"),
    ("#aedd5f", "#8e5fdd"),
    ("#ff0000", "#00ffff"),
])
def test_for_complementary_color_in_hex(
        hexcode, expected):
    got = get_complementary(hexcode)
```

Quelques conseils

- Faire fonctionner son code le plus tôt possible en évitant les problèmes.
- Coder simplement pour obtenir un premier jet et optimiser/refactoriser par la suite
- Ne pas hésiter pas à s'aider de bibliothèques, mais il est possible de réaliser tous les calculs à la main pour s'entraîner si on le souhaite.
- Le but n'est pas d'obtenir le code le plus court, mais un code lisible et optimisé avec un algorithme efficace. Penser qu'un code n'est pas seulement pour son usage personnel, surtout si on travaille en *open source*, donc veiller a ce qu'il soit clair. Ne pas hésiter à ajouter des commentaires, cela nous servira tout autant quand nous reverrons notre code trois mois plus tard sans y avoir touché !

Challenge N° 4

Calculatrice romaine

4.1 Énoncé

Le but de ce challenge est de développer la fonction:

`add_romans(calculate:str)->str`

...qui permet de calculer la somme de deux ou plusieurs nombres écrits en chiffres romains [1].

Conditions

- L'affichage du prompt se fera via la console.
- L'opération est simple : deux ou plusieurs nombres écrits sur une seule ligne en chiffres romains séparés par un signe `+`.
- Le résultat de l'opération sera bien sûr donné en chiffres romains dans une plage comprise entre `I` et `MMMCMXCIX`.
- Vérifier que les chiffres romains et l'opération entrés par l'utilisateur sont valides. Les lettres doivent toujours êtres écrites en majuscules.
- Les chiffres romains sont : `IVXLCDM`.
- `I`, `X` et `C` ne peuvent pas être répétés plus de trois fois.
- `V`, `L` et `D` ne peuvent pas apparaître plus d'une fois.

Exemples

```
- X + IV -> XIV
- MXCIX + CIV -> MCCIII
- II + II -> IV
- MMMCX + DCII -> MMMDCCXII
```

1. https://fr.wikipedia.org/wiki/Num%C3%A9ration_romaine)

- MCMXCIX + I -> MM
- IVM + I -> erreur (IVM n'est pas valide)
- VVI -> erreur (2x V n'est pas valide)
- XXXX -> erreur (4x X n'est pas valide)
- TX -> erreur (T ne fait pas partie de la liste des chiffres romains)
- XX - IV -> erreur (seule l'addition est permise)
- IV -> erreur (2 éléments minimum)
- MMMCMXC + X -> erreur (débordement, doit respecter la plage de I à MMMCMXCIX inclus)

Ressource

Voir le site `Roman numerals calculator` pour réaliser des tests[2].

4.2 Ma proposition

```python
from re import search

ROMANS = {"I": 1, "V": 5, "X": 10, "L": 50,
          "C": 100, "D": 500, "M": 1000}
SPEC = {4: ("IV", "XL", "CD"), 9: ("IX", "XC", "CM")}
list_romans = list(ROMANS.keys())


def to_dec(roman_nb: str) -> int:
    nb = [ROMANS[el] for el in roman_nb]
    return sum([nb[i] - 2 * nb[i - 1] if nb[i] > nb[i - 1]
                else nb[i] for i in
                range(1, len(nb))]) + nb[0]


def n_to_9(nb: int, lv: int) -> str:
    base, v = list_romans[lv * 2], list_romans[1:6:2][lv]
    return f"{base * nb}" if nb < 4 else v if nb == 5 \
        else SPEC[nb][lv] if nb in SPEC \
        else f"{v}{base * (nb - 5)}"


def to_roman(nb: int) -> str:
    nb = [int(i) for i in reversed(str(nb))]
    return "".join([f"{'M' * nb[3]}" if i == 3
                    else n_to_9(nb[i], i) for i in
                    range(0, len(nb))][::-1])
```

2. https://www.hackmath.net/en/calculator/roman-numerals

```
28
29
30  def add_romans(calculate: str) -> str:
31      if [el for el in "-*/" if el in calculate]:
32          return "Seule l'addition est permise"
33
34      el_list = calculate.replace(" ", "").split("+")
35      if len(el_list) < 2:
36          return "2 éléments minimum"
37
38      calc = []
39      for el in el_list:
40          err = [lt for lt in el if lt not in ROMANS]
41          if err:
42              return (f"{err[0][0]} ne fait pas partie de "
43                      f"la liste des chiffres romains")
44
45          if search("IL|IC|ID|IM|XD|XM|VL|VC|VD|VM", el):
46              return f"{el} n'est pas valide"
47
48          err = search("V.*?V|L.*?L|D.*?D", el)
49          if err:
50              return f"2x {err[0][0]} n'est pas valide"
51
52          err = search(
53              rf'([{"".join(list_romans[0:5:2])}])\1{{3,}}',
54              el)
55          if err:
56              return f"4x {err[0][0]} n'est pas valide"
57
58          err = search(
59              rf'({"|".join(SPEC[4] + SPEC[9])}).*?\1', el)
60          if err:
61              return f"2x {err[0]} n'est pas valide"
62
63          calc.append(to_dec(el))
64      result = sum(calc)
65      return to_roman(result) if result < 4000 else \
66          ("Débordement, doit respecter la plage de I à "
67           "MMMCMXCIX inclus")
68
69
70  if __name__ == '__main__':
71      print(add_romans(input("une opération : ")))
```

Mon fichier d'*unitests*

```python
import pytest

@pytest.mark.parametrize("calculate, expected", [
    ("X + IV", "XIV"),
    ("MXCIX + CIV", "MCCIII"),
    ("II + II", "IV"),
    ("MMMCX + DCII", "MMMDCCXII"),
    ("MCMXCIX + I", "MM"),
    ("IVM + I", "IVM n'est pas valide"),
    ("VVI + I", "2x V n'est pas valide"),
    ("XXXX + I", "4x X n'est pas valide"),
    ("TX + I", "T ne fait pas partie de la liste des "
               "chiffres romains"),
    ("XX - IV", "Seule l'addition est permise"),
    ("IV", "2 éléments minimum"),
    ("MMMCMXC + X", "Débordement, doit respecter la plage "
                    "de I à MMMCMXCIX inclus"),
])
def test_should_return_the_sum_of_roman_numbers(calculate,
                                                expected):
    got = add_romans(calculate)

    assert got == expected
```

4.3 Explications

Contexte

Il s'agit de trouver un moyen pour additionner des chiffres romains entre eux. Pour cela, il est préférable de rester dans un domaine de nombres arabes pour effectuer l'opération arithmétique puis reconvertir la sommes finales en chiffres romains.

Les différentes problématiques

- **Conversion chiffres romains en chiffres arabes :** Il faut trouver une astuce pour faire correspondre ces chiffres romains en chiffres arabes. La solution est dans cette phrase : « *Un nombre écrit en chiffres romains se lit de gauche à droite. En première approximation, sa valeur se détermine en faisant la somme des valeurs individuelles de chaque symbole, sauf quand l'un des symboles précède un symbole de valeur supérieure ; dans ce cas, on soustrait la valeur du premier symbole au deuxième* »[3].

3. Source Wikipédia: https://fr.wikipedia.org/wiki/NumÃ‰ration_romaine

- **Conversion du résultat en chiffres romains :** Il y a plusieurs façons de faire, soit faire correspondre chaque symbole romain, soit reconstituer le nombre par dizaine, centaine et millier.
- **Gestion des erreurs :** Pour ce challenge, il s'agissait de simplement suivre la liste dans les exemples, car plusieurs écritures sont possible en fonction de la version romaine utilisée. Dans ce cas, faire un simple `return` du texte est accepté, sinon faire l'habituel `raise` pour *lever l'erreur*.

Explication du code

Mon programme se divise donc en trois, voire en quatre parties :

La conversion des chiffres romains en chiffres arabes

Avec la fonction :

`to_dec(roman_nb:str)->int`

Ici, de simples additions tant qu'on ne rencontre pas de paires contenant un élément de valeur supérieur, auquel cas, il faudra réaliser une soustraction. La conversion s'opère principalement par ces lignes :

```
1 def to_dec(roman_nb: str) -> int:
2     nb = [ROMANS[el] for el in roman_nb]
3     return sum([nb[i] - 2 * nb[i - 1] if nb[i] > nb[i - 1]
4                 else nb[i] for i in
5                 range(1, len(nb))]) + nb[0]
```

La conversion des chiffres arabes en chiffres romains

Avec la fonction :

`to_roman(nb:int)->str`

Là, c'est plus compliqué. Ici je décompose mon nombre en unité, dizaine, centaine et millier et m'occupe de chaque éléments un par un. Les éléments particulier 4, 5 et 9 sont traités dans une sous-fonction :

```
1 def n_to_9(nb: int, lv: int) -> str:
2     base, v = list_romans[lv * 2], list_romans[1:6:2][lv]
3     return f"{base * nb}" if nb < 4 else v if nb == 5 \
4         else SPEC[nb][lv] if nb in SPEC \
5         else f"{v}{base * (nb - 5)}"
```

Le code de la fonction est le suivant :

```python
def to_roman(nb: int) -> str:
    nb = [int(i) for i in reversed(str(nb))]
    return "".join([f"{'M' * nb[3]}" if i == 3
                    else n_to_9(nb[i], i) for i in
                    range(0, len(nb))][::-1])
```

La fonction principale

`add_romans(calculate:str)->str`

Corps du programme dans lequel la gestion des erreurs se fait à l'aide principalement des *expressions régulières* (*regex*).

```python
def add_romans(calculate: str) -> str:
    if [el for el in "-*/" if el in calculate]:
        return "Seule l'addition est permise"

    el_list = calculate.replace(" ", "").split("+")
    if len(el_list) < 2:
        return "2 éléments minimum"

    calc = []
    for el in el_list:
        err = [lt for lt in el if lt not in ROMANS]
        if err:
            return (f"{err[0][0]} ne fait pas partie de "
                    f"la liste des chiffres romains")

        if search("IL|IC|ID|IM|XD|XM|VL|VC|VD|VM", el):
            return f"{el} n'est pas valide"

        err = search("V.*?V|L.*?L|D.*?D", el)
        if err:
            return f"2x {err[0][0]} n'est pas valide"

        err = search(
            rf'([{"".join(list_romans[0:5:2])}])\1{{3,}}',
            el)
        if err:
            return f"4x {err[0][0]} n'est pas valide"

        err = search(
            rf'({"|".join(SPEC[4] + SPEC[9])}).*?\1', el)
        if err:
            return f"2x {err[0]} n'est pas valide"

```

```
34          calc.append(to_dec(el))
35      result = sum(calc)
36      return to_roman(result) if result < 4000 else \
37          ("Débordement, doit respecter la plage de I à "
38           "MMMCMXCIX inclus")
```

Notes sur les *regexes* utilisées

Le *regex* est un langage à part entière, il permet de chercher des occurrences plus ou moins complexes dans un bloc de texte. C'est grâce à la bibliothèque `re` que l'on peut l'intégrer facilement à *Python*.

`search("IL|IC|ID|IM|XD|XM|VL|VC|VD|VM", el)`

`search` cherche soit `IL`, `IC`, `ID`, `IM`... le caractère « | » représente le « `or` ».

`search("V.*?V|L.*?L|D.*?D", el)`

Cela représente n'importe quel caractère (`*?` signifie « *jusqu'à* ». Donc de `V` jusqu'à `V`, ce qui permet d'indiquer que deux `V` peuvent être présents. De même à l'aide de « or » pour chercher aussi deux `L` et deux `D`.

`search(rf'([{"".join(list_romans[0:5:2])}])\1{{3,}}', el)`

`list_romans[0:5:2]` correspond à `["I", "X", "C"]`. On réalise ici un `.join()`, ce qui nous donne `IXC`. Ce `IXC` est entre parenthèses car nous faisons ainsi un groupe qui enregistre le caractère trouvé entre `I` ou `X` ou `C`. Le `\1` permet de récupérer ce groupe et recherche alors les deux mêmes caractères qui se suivent. `{3,}` veut dire : répéter au moins trois ou plus. Donc en tout, on cherche la répétition du même caractère quatre fois de suite (une fois dans le premier groupe, puis trois fois à l'aide de `\1`).

`search(rf'({"|".join(SPEC[4]+SPEC[9])}).*?\1', el)`

On cherche si `SPEC[4]` est dans `("IV", "XL", "CD")` ou si `SPEC[9]` est dans ("IX", "XC", "CM"), et s'ils sont présents deux fois. Pour information, le `r` avant le `f` du *f-string* permet d'ignorer le caractère d'échappement « `\` », car sans ce `r` il aurait fallu écrire « `\\1` » plutôt que « `\1` ».

4.4 En complément

- A noter deux vidéos existantes sur le sujet présentes sur `Youtube`: *Convertir des nombres entiers en chiffres romains*[4] et *Convertir des chiffres romains en nombres entiers*[5].
- *Comment vérifier si une liste est vide en Python ?*[6]

Maintenant je pense que pour vous, c'est très facile de lire cette date, n'est-ce pas?

4. https://www.youtube.com/watch?v=fD9aw0dZtjc
5. https://www.youtube.com/watch?v=WFyrryNO9Nk
6. https://flexiple.com/python/check-if-list-is-empty-python

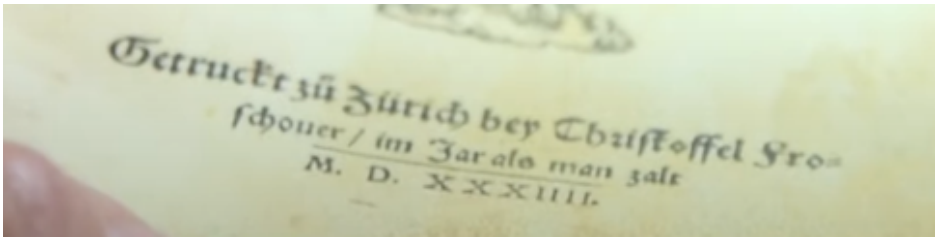

FIGURE 4.1 – Extrait de la première page d'une bible imprimée

Challenge N° 5
Affichage digital 7 segments

5.1 Énoncé

Le but de ce challenge est de développer la fonction `digital_time()->str` qui retourne l'heure actuelle au format d'affichage digital à sept segments [1].

Étapes

1. Transcrire les chiffres de 0 à 9 en sept segments digitaux.
2. Transcrire plusieurs chiffres à la suite sur une seule ligne.
3. Afficher l'heure en la retranscrivant.

Conditions

- L'affichage se fait via la console
- L'heure doit être affichée au format 24h.
- L'heure et les minutes sont séparées par deux points, le tout sur une même ligne d'affichage avec deux espaces de séparation.
- Le résultat de l'affichage de l'heure actuelle doit correspondre aux exemples ci-dessous à l'aide du caractère « # ».
- Les sept segments `a`, `b`, `c`, `d`, `e`, `f`, `g` sont les suivants :

1. https://fr.wikipedia.org/wiki/Affichage_%C3%A0_sept_segments

```
     a
     v
    ####
f-> #  # <- b
    #### <- g
e-> #  # <- c
    ####
     ^
     d
```

- Pour des raisons d'ergonomie et d'esthétisme, les segments se superposent et s'affichent de manière plus allongée, ainsi le chiffre `2` s'affichera de la façon suivante:

```
             ####                          ##
                #                            #
comme ceci : ####   et pas comme cela :  ##
             #                           #
             ####                         ##
```

Exemples

- À `17h26`, le résultat de `digital_time()` donne :

```
#  ####     ####  ####
#     #  #     #  #
#     #     ####  ####
#     #  #  #     #  #
#     #     ####  ####
```

- À `23h59`, le résultat de `digital_time()` donne :

```
####  ####     ####  ####
   #     #  #  #     #  #
####  ####     ####  ####
#        #  #     #     #
####  ####     ####  ####
```

- À `4h08`, le résultat de `digital_time()` donne : -

```
#  #     ####  ####
#  #  #  #  #  #  #
####     #  #  ####
   #  #  #  #  #  #
   #     ####  ####
```

5.2 Ma proposition avec l'utilisation de **Numpy**

```
from numpy import hstack, delete, full, ndarray
from datetime import datetime

SEG_MAP = (((0, 0), (0, 1), (0, 2), (0, 3)),  # a
           ((0, 3), (1, 3), (2, 3)),  # b
           ((2, 3), (3, 3), (4, 3)),  # c
           ((4, 0), (4, 1), (4, 2), (4, 3)),  # d
           ((4, 0), (3, 0), (2, 0)),  # e
           ((2, 0), (1, 0), (0, 0)),  # f
           ((2, 0), (2, 1), (2, 2), (2, 3)),  # g
           ((1, 0), (3, 0)))


def display(el: str) -> ndarray:
    digit = full((5, 6), [" "], dtype=str)
    match el:
        case " ": return digit
        case ":":
            s = [*[0]*7, 1]
            digit = delete(digit, slice(3), 1)
        case _:
            b = [int(b) for b in f"{int(el):04b}"][::-1]
            s = (b[3] or b[1] or b[0] and b[2] or not b[0]
                 and not b[2],
                 b[0] and b[1] or not b[1] and not b[0]
                 or not b[2],
                 b[2] or b[3] or not b[1] or b[0],
                 b[3] or b[1] and not b[0] or not b[2]
                 and not b[0] or not b[1] and b[0] and b[2]
                 or b[1] and not b[2],
                 b[1] and not b[0] or not b[0]
                 and not b[2],
                 b[3] or not b[0] and b[2] or not b[1]
                 and b[2] or not b[1] and not b[0],
                 b[3] or b[1] and not b[0] or not b[1]
                 and b[2] or not b[2] and b[1])
    digit[*zip(*[(r, c) for i, el in enumerate(s) if el
                 for r, c in SEG_MAP[i]])] = "#"
    return digit


def digital_time():
    time = datetime.now().strftime("%H:%M").lstrip("0")
    return "\n".join("".join(r) for r in
                     hstack([display(el) for el in
```

```
46                                  (time if len(time) == 5 else
47                                   " " + time)]))
48
49
50 if __name__ == "__main__":
51     print(digital_time())
```

5.3 Explications

Contexte

Il s'agit de trouver le moyen d'afficher un nombre sous forme digitale en sept segments comme on peut le voir sur certaines horloges numériques, puis d'afficher l'heure actuelle.

Différentes problématiques

- **Conversion d'un chiffre en sept segments pour simuler un affichage digital**: Il s'agit de repérer la position de chacun des segments puis « d'allumer » les segments spécifiques au chiffre à afficher.
- **Afficher plusieurs chiffres sur une ligne**: L'étape suivante est d'afficher plusieurs chiffres sur une même ligne en transposant les « matrices » des chiffres à afficher.
- **Ajouter le séparateur heure/minute**: Il nous faut donc ajouter le caractère « : » entre les heures et les minutes.
- **Formatage particulier**: Les heures de 0 à 9 ne doivent pas afficher un zéro mais laisser l'espace d'un *digit* à gauche.

Explication du code

Le code se divise en trois, voire quatre parties :

- **Définir les segments à allumer en fonction du chiffre à afficher**

Ma méthode ici consiste à chercher des solutions combinatoires pour chaque segment en fonction du chiffre à afficher. Les chiffres sont au nombre de 10 et peuvent varier de 0 à 9. Ce qui nécessite quatre *bits* pour les coder en binaire :

```
0000 -> 0
0001 -> 1
0010 -> 2
...
1000 -> 8
1001 -> 9
```

Il y a ensuite sept solutions combinatoires à rechercher pour chacun des segments de « a » à « g ».

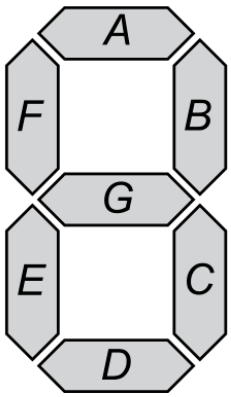

FIGURE 5.1 – Représentation visuelle des sept solutions combinatoires

Ainsi il faut allumer les segments `B` et `C` pour coder le chiffre `1`, `BCFG` pour le chiffre `4`, `ABCDEF` pour le chiffre `0`, etc. On peut donc poser tous ces résultat dans une *table de vérité* :

	b3	b2	b1	b0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

FIGURE 5.2 – *Table de vérité*

A gauche les chiffres transcrits en quatre *bits* de `b0` à `b3` et à droite les segments à allumer.

La table de vérité est incomplète dans le sens où on n'utilise pas la totalité des possibilités en quatre *bits*. Ainsi, on pourra être libre de choisir n'importe quels états durant la simplification de nos équations à l'aide des *tableaux de Karnaugh* (ces combinaisons seront notées en rouge dans les tableaux).

Le chiffre est d'abord converti en binaire sur quatre *bits* pour procéder au calcul avec l'*algèbre de Boole* :

```
b = [int(b) for b in f"{int(el):04b}"][::-1]
```

Pour faire bien correspondre à mes tableaux je suis obliger de faire une inversion via `[::-1]`.

Pour simplifier chacune des sept solutions on posera nos *tableaux de Karnaugh*[2] de cette manière :

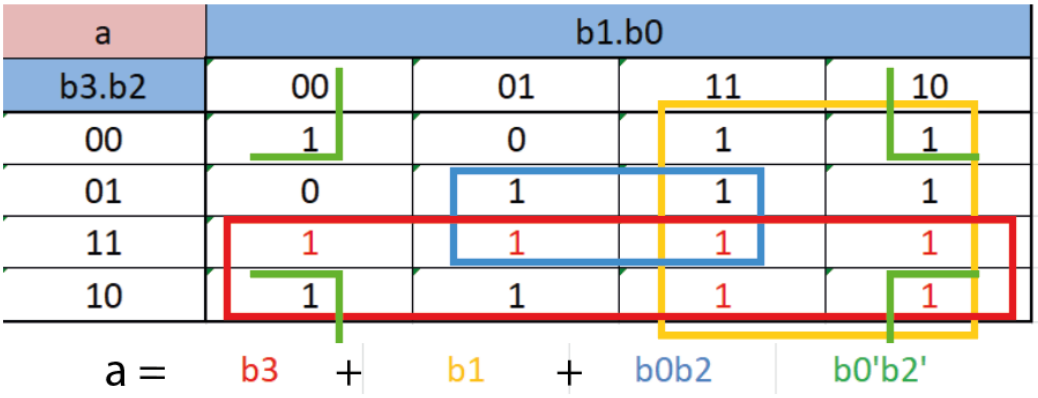

FIGURE 5.3 – *Tableau de Karnaugh* - 1

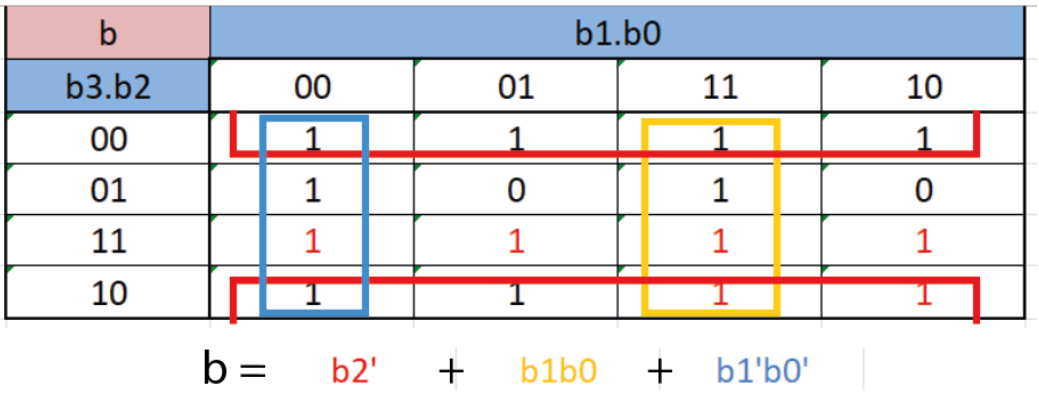

FIGURE 5.4 – *Tableau de Karnaugh* - 2

2. *Tableaux de Karnaugh* - simplification d'expression en *algèbre de Boole*: https://www.youtube.com/watch?v=2tULBk6V9ZE

c	b1.b0			
b3.b2	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

c = b1' + b0 + b2 + b3

De Morgan

FIGURE 5.5 – *Tableau de Karnaugh* - 3

d	b1.b0			
b3.b2	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	1	1	1	1
10	1	1	1	1

d = b1b0' + b2'b0' + b1'b0b2 + b3 + b1b2'

FIGURE 5.6 – *Tableau de Karnaugh* - 4

e	b1.b0			
b3.b2	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	1	1	1	1
10	1	0	1	1

e = b0'b2' + b1b0'

FIGURE 5.7 – *Tableau de Karnaugh* - 5

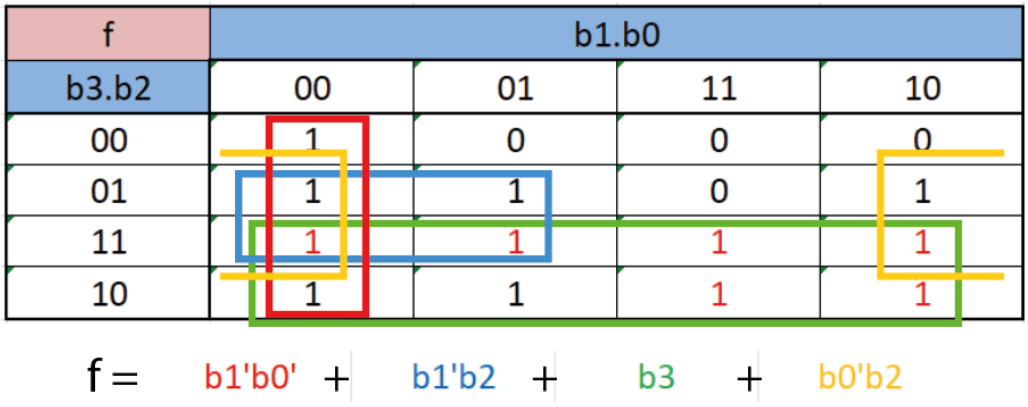

FIGURE 5.8 – *Tableau de Karnaugh* - 6

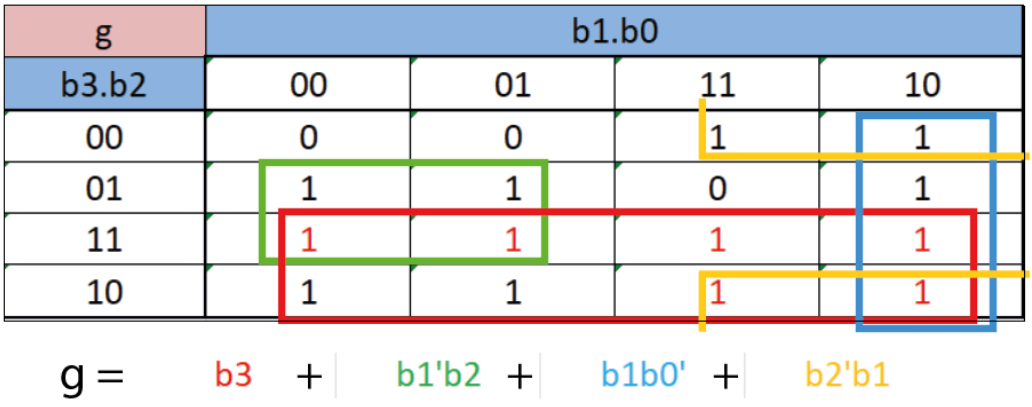

FIGURE 5.9 – *Tableau de Karnaugh* - 7

On peut donc poser les solutions sous forme de listes qu'on pourra interpréter respectivement via les index de `0` à `6` pour les segments de `a` à `g`.

```
s = (b[3] or b[1] or b[0] and b[2] or not b[0] and not
     b[2],
     b[0] and b[1] or not b[1] and not b[0] or not b[2],
     b[2] or b[3] or not b[1] or b[0],
     b[3] or b[1] and not b[0] or not b[2] and not b[0]
     or not b[1] and b[0] and b[2] or b[1] and not b[2],
     b[1] and not b[0] or not b[0] and not b[2],
     b[3] or not b[0] and b[2] or not b[1] and b[2]
     or not b[1] and not b[0],
     b[3] or b[1] and not b[0] or not b[1] and b[2]
     or not b[2] and b[1])
```

Puis la liste `SEG_MAP` me permet de fixer la position des segments en posant leur coordonnées dans la matrice d'un chiffre digital :

```
SEG_MAP = (((0, 0), (0, 1), (0, 2), (0, 3)), # a
           ((0, 3), (1, 3), (2, 3)), # b
```

```
3               ((2, 3), (3, 3), (4, 3)), # c
4               ((4, 0), (4, 1), (4, 2), (4, 3)), # d
5               ((4, 0), (3, 0), (2, 0)), # e
6               ((2, 0), (1, 0), (0, 0)), # f
7               ((2, 0), (2, 1), (2, 2), (2, 3)), # g
8               ((1, 0), (3, 0))) # :
```

Notez que la dernière ligne correspond au caractère de séparation « : » entre l'heure et les minutes.

- **Affichage d'un *digit***

La variable `digit` est un tableau *Numpy* initialisé avec des espaces :

```
digit = full((5, 6), [" "],dtype=str)
```

Je teste ici avec `match case` les caractères à afficher, soit `" "` pour un *digit* « éteint », puis les deux points (« : ») pour le séparateur et le reste pour les chiffres de 0 à 9.

```
match el:
        case " ": ...
        case ":":
            ...
        case _:
            ...
```

Une fois les solutions de ma table de vérité trouvées, alors on récupère les coordonnées pour afficher un « # » à chaque fois pour former les segments à allumer :

```
1 digit[*zip(*[(r, c) for i, el in enumerate(s) if el
2                  for r, c in SEG_MAP[i]])] = "#"
3     return digit
```

Cette fonction peut être éclatée de cette manière pour plus de lisibilité :

```
1 pos = [(r, c) for i, el in enumerate(s) if el
2            for r, c in SEG_MAP[i]]
3 r, c = zip(*pos)
4 digit[r, c] = "#"
```

Ce n'est qu'une simple transposition afin de réunir les coordonnées `x` et `y` et les appliquer d'un coup et ainsi remplir le tableau *Numpy* `digit`.

- **Afficher sur une ligne**

Pour ce faire on doit procéder à une lecture ligne par ligne de chaque `digit`. On utilise pour cela la fonction `hstack()`[3] qui permet de concaténer ligne par ligne mes tableaux *Numpy*.

Tout se fait à cette ligne :

```
hstack([display(el) for el in time])
```

- **Afficher l'heure en suivant l'énoncé**

L'affichage de l'heure n'est pas bien complexe: heure sur un chiffre et les minutes avec deux chiffres. Il faut penser à effacer le premier zéro de l'heure si l'heure est entre 0 et 9 puis d'ajouter un *digit* blanc pour l'affichage.

```
1 time = datetime.now().strftime("%H:%M").lstrip("0")
2 ...
3 ...time if len(time) == 5 else " "+time...
```

Pour cela je cherche la taille de l'heure et si l'on a que cinq caractères alors on se trouve dans le cas d'un affichage avec deux *digits* pour l'heure sinon ce sera une longueur de quatre :

- 3:24 <- une longueur de quatre
- 15h58 <- une longueur de cinq

Le caractère spécial « : » est calculé au sein de la fonction `display()` en ajoutant une solution supplémentaire pour afficher les positions de mon « deux-points » à la fin de la liste `SEG_MAP`.

3. https://numpy.org/doc/stable/reference/generated/numpy.hstack.html

Challenge N° 6

Cryptographie symétrique

6.1 Énoncé

Un peu d'encryptage pour vos communications privées !

Le but de ce challenge est de développer un codeur/décodeur suivant la logique de la cryptographie symétrique[1] qui utilise une clef numérique très simple pour chiffrer et déchiffrer.

Étapes

1. Envoyer la phrase à chiffrer : « *Salut, je suis ici pour apprendre Python* » et la clef de codage est égale à `6`.

2. On remplace d'abord tous les espace par des « `_` », ce qui nous donne alors la phrase : « *Salut,_je_suis_ici_pour_apprendre_Python* ».

3. Comme la clef est la valeur `6`, on découpe la phrase en lignes de six caractères, les cases manquantes sont comblées par des « `*` »:

```
Salut,
_je_su
is_ici
_pour_
appren
dre_Py
thon**
```

4. On transpose en lisant le texte par colonne (de haut en bas et de gauche à droite), c'est-à-dire en suivant cet ordre :

1. https://fr.wikipedia.org/wiki/Cryptographie_sym%C3%A9trique

```
1  8 15 22
2  9 16 23
3 10 17 24
4 11 18 25
5 12 19 26
6 13 20 27
7 14 21 28
```

La phrase chiffrée sera donc la suivante : « `S_i_adtajspprhle_opeou_iur_ntscreP*,ui_ny*` ».

Pour le décodage, il suffira de suivre les étapes dans le sens inverse...

Conditions

- L'affichage se fait via la console.
- La phrase a chiffrer est a envoyer en argument, accompagnée de sa clef au format `int`, le résultat doit retourner la phrase codée.
- Pour le décodeur, la phrase chiffrée est envoyée accompagnée de la clef, le résultat doit retourner la phrase en clair.
- La clef doit être la même pour un même chiffrage et déchiffrage.
- Vous êtes libre de coder en fonctionnel ou bien de constituer vos classes à l'aide de la POO.

Exemples

- **Chiffrage**

- « *Lorem ipsum dolor sit amet, consectetur adipiscing elit.* », clef : `12` -> `Ldetnoot ugrl,r_eo__emrcal__odiisnitpisp.stei*u_cs*matc*_mei*`
- « *La première machine programmable a été réalisé en 1801.* », clef : `3` -> `Lpmrmher rmlatrlén8.arieai_oaae_éei__0*_eè_cnpgmb_é_ase11*`

- **Déchiffrage**

- `Lag_eeelelm*_l_aa*ced_i*hnesn*`, clef : `5` -> « *Le challenge de la semaine* »
- `dctigosrn*`, clef : `2` -> « *docstring* »

6.2 Diverses solutions

Code de *@Arnadu*

J'aimerais mettre en avant la meilleure façon d'aborder ce challenge. Et c'est le participant ***@Arnadu*** qui l'a trouvé ! Après quelques étapes de *refactoring*, voici le code résultant final que je trouve absolument brillant de simplicité et d'ingéniosité.

```python
def encode(sentence:str, key:int)->str:
    sentence += "*"*(key - len(sentence) % key) \
        if len(sentence) % key else ""
    return "".join(sentence[i::key]
                   for i in range(key)).replace(" ", "_")


def decode(sentence_encrypt:str, key:int)->str:
    cutter = len(sentence_encrypt)//key
    return ("".join(sentence_encrypt[i::cutter]
                   for i in range(cutter))
            .replace("_", " ").rstrip("*"))


sentences_to_encode = (
    ("Lorem ipsum dolor sit amet, consectetur adipiscing "
     "elit.", 12),
    ("La première machine programmable a été réalisé en "
     "1801.", 3)
)

sentences_to_decode = (
    ("Lag_eeelelm*_l_aa*ced_i*hnesn*", 5),
    ("dctigosrn*", 2)
)

for sentence, key in sentences_to_encode:
    print(f"Sentence to encode : {sentence}")
    print(f"Sentence encoded : {encode(sentence, key)}")

for sentence, key in sentences_to_decode:
    print(f"Sentence to decode : {sentence}")
    print(f"Sentence decoded : {decode(sentence, key)}")
```

Pourquoi son code est excellent ? Car en une seule boucle, il a été capable de réaliser à la fois le découpage et la transposition de la phrase à coder ou à décoder :

```
...sentence[i::key] for i in range(key)
```

On récupère donc chaque groupe de caractères à chaque découpage , pour chaque ligne en utilisant la boucle `for` et `range(key)` avec l'utilisation d'un pas : `[i::key]`.

Ceci est valable aussi pour le décodage :

```
cutter = len(sentence_encrypt)//key
...sentence_encrypt[i::cutter] for i in range(cutter)
```

Le code de *@Hugo*

Voici une autre solution intéressante avec Numpy en fonctionnel, que je trouve aussi très sympathique. C'est une version sur laquelle nous avons pas mal bossé avec ***@Hugo*** :

```
import numpy as np


def init_matrix(sentence: str, key: int, decodage=False) \
        -> str:
    x, y = np.ceil(len(sentence)/key).astype(int), key
    if decodage:
        x, y = y, x

    matrix = np.full((x, y), '*')
    for i, c in enumerate(sentence):
        matrix[i//y, i % y] = c
    return ''.join(np.ravel(np.transpose(matrix)))


def code(sentence: str, key: int) -> str:
    return init_matrix(sentence, key).replace(' ','_')


def decode(coded_sentence: str, key: int) -> str:
    return (init_matrix(coded_sentence, key, True)
            .replace('_', ' ').rstrip('*'))


if __name__ == "__main__":
    key = 6
    print(decode(code("Salut, je suis ici pour apprendre "
                      "Python", key), key))
```

`x` et `y` permettent de calculer la dimension de la matrice ; comme le chiffrage et le déchiffrage sont simplement l'inverse, cela peut se faire via une simple permutation des variables pour le décodage.

Pour le reste, on initialise une matrice que l'on rempli au fur et à mesure avec la phrase à coder ou à décoder.

Une transposition avec `transpose()` de *Numpy* et `ravel()` pour aplatir la matrice dans le but d'afficher son contenu.

Ma solution

Voici maintenant ma proposition que je vous laisse analyser et qui est une autre façon de faire avec quelques subtilités intéressantes au niveau du formatage avec *f-*

string et de l'utilisation de la fonction `zip()` :

```
def encode(text: str, key: int)-> str:
    t = [f"{text[i:i + key]:*<{key}}"
         for i in range(0, len(text), key)]
    return "".join("".join(el)
                   for el in zip(*t)).replace(" ", "_")


def decode(text: str, key: int)-> str:
    row = len(text) // key
    t = [text[i:i + row] for i in range(0, len(text), row)]
    return ("".join("".join(el)
                   for el in zip(*t)).rstrip("*").
            replace("_", " "))


key = 6
print(decode(encode("Salut, je suis ici pour apprendre "
                    "Python", key), key))
```

Ici, la phrase est découpée avec un *pas* de `key` en utilisant `range(0, len(text), key)`, puis les caractères sont récupérés ligne par ligne entre l'*index* et l'*index* auquel on ajoute la valeur de `key`, grâce à `text[i:i + key]`.

On procède ensuite à la transposition à l'aide de la fonction `zip()` et l'*unpack* (la petite astérisque qui dispatche tous les éléments...) de la compréhension de liste issue de la ligne de code précédente. C'est à ce moment-là que les espaces sont remplacés par des « _ ». À noter que pour compléter les espaces vides avec des astérisques, j'utilise ici le formatage à l'aide des *f-string*[2] qui permet de réaliser la fonction via `:*<key`.

Le décodage se fait de la même façon, dans le sens inverse.

La solution de *@OsKaR31415*

« *Voici ma solution avec le paradigme orienté objet :*

```
import numpy as np
from abc import ABC, abstractmethod


def read_by_columns(str_array: np.ndarray[str]) -> str:
    """Retourner une chaîne de caractère de tous les
    caractères du tableau 2D, lus par colonnes.
    Exemples:
        >>> read_by_columns(np.array([["s", "u"],
```

2. Pour plus d'information sur les *f-string*: https://he-arc.github.io/livre-python/fstrings/index.html

```
        ["a", "t"], ["l", "!"]]))
        'salut!'
    """
    return ''.join(np.ravel(np.transpose(str_array)))

class Codeur(ABC):
    """Classe abstraite pour un objet qui encode ou décode
    un texte avec une clef. Cette classe gère la
    manipulation de la clef (initialisation, accesseur et
    mutateur). Elle pose la nécessité de définir la méthode
    `__call__` dans toutes les classes enfant.
    """
    def __init__(self, clef):
        self.clef = clef

    @property
    def clef(self) -> int:
        return self._clef

    @clef.setter
    def clef(self, valeur: int) -> None:
        if not isinstance(valeur, int):
            raise TypeError(f"La clef doit être un entier"
                            f", pas un {type(valeur)}.")
        self._clef = valeur

    @abstractmethod
    def __call__(self, texte: str) -> str:
        """Une méthode abstraite n'est pas définie.
        Elle sert à dire que toutes les classes filles
        doivent définir cette méthode."""
        ...


class Encodeur(Codeur):
    """Classe pour encoder du texte.
    Puisqu'elle hérite de `Codeur`, il n'est pas nécessaire
    de s'occuper de la gestion de l'attribut `clef`, ni de
    définir le `__init__`."""
    def __call__(self, texte: str) -> str:
        # remplacement des espaces + ajout des "*" à la fin
        texte = (texte.replace(' ', '_') +
                 "*" * (self.clef - len(texte)
                        % self.clef))
        return read_by_columns(np.reshape(np.array(
            list(texte)), (len(texte)//self.clef,
                           self.clef)))

```

```python
class Decodeur(Codeur):
    """Classe pour décoder du texte."""
    def __call__(self, texte: str) -> str:
        return (read_by_columns(np.reshape(np.array(
            list(texte)), (self.clef, len(texte)//
                           self.clef)))
                .rstrip('*').replace('_', ' '))


chiffrer = Encodeur(12)  # un encodeur avec la clef 12
assert chiffrer("Lorem ipsum dolor sit amet, consectetur "
                "adipiscing elit.") == ("Ldetnootugrl,r_e"
                                        "o__emrcal__odii"
                                        "snitpisp.stei*u_"
                                        "cs*matc*_mei*")
chiffrer.clef = 3  # on change la clef
assert (chiffrer("La première machine programmable a été "
                 "réalisé en 1801.") ==
        ("Lpmrmherrmlatrlén8.arieai_oaae_éei__0*_eè_cnpg"
         "mb_é_ase11*"))

# on peut également définir le Codeur à la volée :
assert (Decodeur(5)("Lag_eeelelm*_l_aa*ced_i*hnesn*") ==
        "Le challenge de la semaine")

# en définissant l'encodeur et le décodeur à la volée :
texte = "un texte au hasard avec un * pour le piège"
clef = 7
assert Decodeur(clef)(Encodeur(clef)(texte)) == texte
```

J'ai choisi de faire des objets avec une clef fixée, mais qui peuvent encoder ou décoder n'importe quel texte. Les objets `Encode` *et* `Decode` *sont en fait des sortes de fonctions (des* `_callable_`*) que l'on appelle avec comme argument un texte, et qui vont retourner le texte encodé ou décodé avec leur clef.* »

6.3 Quelques remarques sur les codes des participants au challenge, par *@OsKaR31415*

Quelques remarques sur les codes que j'ai vu :

Parmi les solutions, certaines utilisaient parfois trois boucles, et beaucoup de calculs intermédiaires. C'est une bonne pratique de diviser un problème en petits sous-problèmes (« *diviser chacune des difficultés [...] en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre* » comme disait Descartes).

Mais il ne faut pas en abuser : trop diviser son code pose problème. D'abord, cela ajoute de la complexité puisqu'il faut réunir les différentes parties divisées. Mais surtout, c'est problématique car on est souvent obligés, pour pouvoir le diviser, de rendre le problème plus complexe qu'il ne l'est.

C'est une question de cacher le code (par exemple le mettre dans une fonction à part), ou bien de le subordonner (ne pas le séparer, mais faire en sorte que sa délimitation soit claire, avec des parenthèses ou des variables). Par exemple, **@bucdany** à tendance à faire des lignes plutôt longues car il est à l'aise avec le code subordonné : une compréhension de liste dans une fonction, suivie d'un `.replace`... Si on est habitué, c'est plus lisible qu'un code dans lequel chaque étape est séparée dans une fonction à part.

J'ai vu plusieurs solutions utilisant la programmation orientée objet. C'est un paradigme très utile et pratique, mais il faut aussi comprendre sa logique. Par exemple, dans ce challenge, il fallait gérer un texte et une clef pour l'encodage et le décodage. Certaines solutions ont été de faire un objet qui contient un texte, et de proposer de (dé/en)coder ce texte avec n'importe quelle clef. Cela peut être utile dans certains contexte, mais généralement, il semble plus logique de choisir une clef une fois, et ensuite d'encoder plusieurs textes. Pour cela, mes classes ne stockent pas le texte, mais le transforment simplement.

Il faut penser à ce genre d'enjeux lorsque l'on définit la structure de nos objets. Il faut aussi repérer les parties du code qui seront communes à plusieurs classes pour en faire une classe parente.

Enfin, les classes abstraites peuvent être utiles lorsque l'on veut définir un `_contrat_`, c'est à dire une sorte de structure que toutes les classes filles d'une classe doivent respecter. Dans mon code, la classe `Codeur` est une classe abstraite (elle ne sert pas à créer des objets, mais d'autres classes peuvent hériter d'elle), et elle pose le contrat suivant : toutes les classes filles doivent contenir la méthode `__call__`.

6.4 Partie *bonus* du challenge

Le mot clé n'est plus de type `int` mais de type `string` de `n` caractères. On prendra ici le mot clé de six caractères : `python`.

Étapes

1. On fera alors bien correspondre chaque lettre de ce dernier, à notre phrase à coder que l'on découpera en `n` caractères (la longueur de notre mot clé) de cette manière :

```
p y t h o n
S a l u t ,
_ j e _ s u
i s _ i c i
_ p o u r _
a p p r e n
```

```
d r e _ P y
t h o n * *
```

2. On range ensuite chaque lettre du mot clé par ordre alphabétique : `h n o p t y`. Chaque lettre de la phrase à coder sera alors mélangée de sorte qu'elle suive toujours sa correspondance avec la lettre du mot clé qui lui a été associée dans la phase précédente. On aura donc ce résultat :

```
h n o p t y
u , t S l a
_ u s _ e j
i i c i _ s
u _ r _ o p
r n e a p p
_ y P d e r
n * * t o h
```

3. Pour finir, on lira le tout par colonne (en veillant bien sûr à enlever du traitement, la première ligne, c'est-à-dire, notre mot clé) pour obtenir la phrase codée suivante :

```
u_iur_n,ui_ny*tscreP*S_i_adtle_opeoajspprh
```

Bien sûr, je vous invite à créer une méthode de décodage en repassant toutes les étapes dans l'ordre inverse...

Là, ça devient un algorithme de cryptage plutôt intéressant et vraiment pas facile à décoder sans la clé !

NOTA: Afin d'éviter d'avoir des problèmes au décodage, je vous invite à faire vos tests en veillant à ne pas choisir un mot clé qui possède des récurrences dans sa constitution : par exemple, pour le mot clé : `"probleme"`, les deux `"e"` contenus dans ce mot peuvent être rangés dans un ordre alphabétique différent, laissant alors deux résultats possibles...

Bonne chance !

Challenge N° 7

Extraction des données

7.1 Énoncé

Aie ! Je ne retrouve plus ma liste de contact !

Le but de ce challenge est d'extraire les noms, prénoms, téléphones et emails à partir d'un paragraphe, puis de créer une base de données afin de les afficher clairement à l'écran.

Étapes

1. Trouver le moyen de repérer les différentes limites d'extractions.
2. Ranger les données dans une base de données en suivant cet ordre : `Nom & Prénom - Mail - Numéro`.
3. Afficher le résultat formaté.

Conditions

- L'affichage se fait via la console.
- Vous êtes libre d'utiliser (ou pas) toutes les librairies de votre choix.
- Vous pouvez coder votre programme aussi bien en fonctionnel qu'en orienté objet.
- L'affichage des données doit respecter le format visuel suivant (voir ci-dessous dans résultat).
- Utilisez les trois paragraphes ci-dessous pour procéder à l'extraction.

Texte dans lequel extraire les données

Récemment, j'ai eu l'opportunité de rencontrer des entrepreneurs exceptionnels lors d'une conférence. Par exemple, j'ai été inspiré par l'histoire de M. Thomas Bernard, qui a démarré sa propre entreprise dans la Silicon Valley. Vous pouvez le contacter à l'adresse email `thomas.b@alphamail.com` *ou au numéro suivant +33 1 12 34 56 78. Une autre personne fascinante était Mme Claire Martin, la fondatrice d'une startup technologique innovante. Elle est joignable à* `cmartin@betainbox.org`*, son numéro de téléphone est le 09 01 23 45 67. Ensuite, il y avait monsieur Lucas Petit, un innovateur dans le domaine de la construction durable, contactable à* `lp@experimentalpost.net`*, son téléphone est le 0890 12 34 56.*

En parcourant mon ancien annuaire, je suis tombé sur quelques contacts intéressants. Par exemple, j'ai redécouvert le contact de Mlle Sophie Martin. Son numéro de téléphone est 07 89 01 23 45, et elle est facilement joignable à l'adresse `sophie@prototypemail.com`*. Un autre contact noté était celui du Dr Lucas Dupont. Je me souviens avoir eu plusieurs discussions avec lui. Son numéro est 06 78 90 12 34 et son mail est* `drdupont@randominbox.org`*. C'est fascinant de voir comment certains contacts peuvent rapidement nous rappeler des souvenirs passés.*

Lors de notre dernière réunion, Madame Jennifer Laroche, joignable au 05.67.890.123 ou par e-mail à `laroche@trialmail.net`*, a exprimé sa satisfaction concernant les avancées du projet. Elle a insisté sur la pertinence du feedback fourni par M. Sébastien Girard, qui peut être contacté au 0456789012 ou par email à* `sebastieng@demomail.org`*. De plus, notre consultante externe, mademoiselle Chloé Lefebvre, dont le numéro est le 03.45.67.89.01 et l'e-mail est* `lefebvre_chloe@testinbox.net`*, a fourni un rapport détaillé qui a été bien reçu par l'équipe.*

Résultat en sortie de console

```
|   Nom & Prénom   |            Mail              |      Numéro      |
|------------------|------------------------------|------------------|
| Thomas Bernard   | thomas.b@alphamail.com       | 33.1.12.34.56.78 |
| Claire Martin    | cmartin@betainbox.org        | 09.01.23.45.67   |
| Lucas Petit      | lp@experimentalpost.net      | 08.90.12.34.56   |
| Sophie Martin    | sophie@prototypemail.com     | 07.89.01.23.45   |
| Lucas Dupont     | drdupont@randominbox.org     | 06.78.90.12.34   |
| Jennifer Laroche | laroche@trialmail.net        | 05.67.89.01.23   |
| Sébastien Girard | sebastieng@demomail.org      | 04.56.78.90.12   |
| Chloé Lefebvre   | lefebvre_chloe@testinbox.net | 03.45.67.89.01   |
|------------------|------------------------------|------------------|
```

7.2 Diverses solutions

Le code de *@Krystof26*

Je vous présente maintenant un des meilleurs code des participants sur lequel nous avons pas mal travaillé. ***@Krystof26*** a proposé une belle solution que nous avons

refactorisé et optimisé ensemble :

```
class ExtractContacts:
    MR_AND_MRS = ("M.", "Mme", "monsieur", "Mlle",
                  "Dr", "Madame", "mademoiselle")
    DB_KEYS = ("Nom & Prénom", "Mail", "Numéro")
    CODE_ZERO = "0"  # usage numéro en national

    def __init__(self, file):
        # Création de la base de données sous forme d'un
        # dictionnaire :
        self._all_contacts = (
            ExtractContacts._extract_all_datas(
                ExtractContacts._extract_txt(file)))

    @staticmethod
    def _extract_txt(file) -> str:
        """Extraction du contenu du fichier texte sous forme
        de chaîne de caractères"""
        with open(file, "r", encoding="utf-8") as textfile:
            return textfile.read()

    @staticmethod
    def _format_phone_number(number: str) -> str:
        """Formatage selon que le numéro de téléphone
        comporte 11 ou 10 chiffres."""
        match number[0]:
            case ExtractContacts.CODE_ZERO:
                # 10 chiffres : 0x.xx.xx.xx.xx (usage
                # en national)
                return ".".join(number[i:i + 2] for i in
                                range(0, len(number), 2))
            case _:  # 11 chiffres: xx.x.xx.xx.xx.xx
                # (usage en international)
                return (f"{number[0:2]}.{number[2]}." + ".".
                        join(number[i:i + 2] for i in
                             range(3, len(number), 2)))

    @staticmethod
    def _extract_all_datas(text: str) -> dict:
        """Extraction des données : listes de nom, de mails,
        de numéros de téléphone dans un dictionnaire."""
        bdd = {key: [] for key in ExtractContacts.DB_KEYS}

        for index, word in enumerate(text.split()):
            if word in ExtractContacts.MR_AND_MRS:
                # Pour les noms
                bdd[ExtractContacts.DB_KEYS[0]].append(
```

```
                    ' '.join(text.split()[
                                index + 1:index + 3])[:-1])
            elif "@" in word:  # Pour les mails
                bdd[ExtractContacts.DB_KEYS[1]].append(
                    word.rstrip('.').rstrip(','))

        # Pour les numéros de téléphone :
        list_numbers = [n for n in text if n.isdigit()]
        while list_numbers:
            term_index = 10 if (list_numbers[0] ==
                                ExtractContacts.CODE_ZERO) \
                else 11
            bdd[ExtractContacts.DB_KEYS[2]].append(
                ExtractContacts._format_phone_number(
                    "".join(list_numbers[:term_index])))
            list_numbers = list_numbers[term_index:]
        return bdd

    def __str__(self):
        longest = [len(max(c, key=len))+2 for c in
                   self._all_contacts.values()]

        # Création des diverses 'strings' nécessaires à la
        # construction du tableau de données :
        line_str = "\n|" + "".join(f"{'-' * long}|"
                                   for long in longest)
        header = "\n|" + "".join(f"{key_db:^{long}}|"
                                 for key_db, long in
                                 zip(ExtractContacts.DB_KEYS,
                                     longest))
        contacts = "\n|" + "\n|".join("".join(
            f" {contact:<{long-1}}|" for contact,
            long in zip(infos, longest)) for infos
                                      in zip(
            *self._all_contacts.values()))
        # Tableau complet avec les données :
        return "".join(t + line_str
                       for t in [header, contacts])


if __name__ == '__main__':
    print(ExtractContacts("text.txt"))
```

Son code est très propre, clair et esthétique. Il n'a pas utilisé les *regexes* ce qui fait que son programme demande juste un tout petit peu plus de logique, mais cela est fort bien structuré en une classe et quatre méthodes.

Ma solution

Voici ma proposition qui comprend une version avec *regex* et l'autre sans :

```
from re import compile

FILE = "data.txt"
TITLE = ("Nom & Prénom", "Mail", "Numéro")
PREFIX = ("monsieur", "Madame", "mademoiselle",
          "Mme", "Mlle", "M.", "Dr")
REGEX = (compile(r) for r in (
             r'(?:monsieur|Madame|mademoiselle|Mme|Mlle|M.'
             r'|Dr)\s((?:[A-Z]\w+\s?){2})',
             r'[\w.]+@\w+\.\w{2,}',
             r'[+0]\d+'))


def printl(func):
    def wrapper(*args):
        result = func(*args)
        print("|" + result)
        return result
    return wrapper


def get_text(file: str) -> str:
    with open(file, encoding="utf8") as f_content:
        return f_content.read()


def format_tel(tel: str) -> str:
    tel_ = ("".join(tel[i:i+2] + "."
                    for i in range(0, len(tel), 2)).
            rstrip("."))
    return tel_[1:2] + tel_[3:1:-1] + tel_[4:] \
        if tel_.startswith("+") else tel_


def extract_regex(txt: str) -> dict:
    return {t: next(REGEX).findall(txt) if i < 2 else \
               list(map(format_tel, next(REGEX).findall(
                   "".join(t for t in txt
                           if t not in " ."))) \
            for i, t in enumerate(TITLE)}


def extract_no_regex(txt: str) -> dict:
    bdd = {k: [] for k in TITLE}

```

```
    txt_ = "".join(t for i, t in enumerate(txt)
                   if t not in " ."
                   or not txt[i-1].isdigit()).split()
    for j, l in enumerate(txt_):
        if l in PREFIX:
            bdd[TITLE[0]].append(" ".join(
                txt_[j+1:j+3])[:-1])
        elif "@" in l:
            bdd[TITLE[1]].append(
                l.rstrip(",").rstrip("."))
        elif any(n.isdigit() for n in l):
            bdd[TITLE[2]].append(
                format_tel(l[:12]
                           if l.startswith("+")
                           else l[:10]))
    return bdd


@printl
def t_title(*args):
    return "".join(f'{t:^{length}}|'
                   for t, length in zip(TITLE, args[0]))


@printl
def t_sep_line(*args):
    return "".join(f'{"-"*length}|' for length in args[0])


@printl
def t_content(*args):
    return "\n|".join("".join(f" {el:<{length-1}}|"
                             for el, length
                             in zip(el_, args[0]))
                      for el_ in zip(*args[1].values()))


def show(bdd: dict) -> None:
    length = [len(max(bdd, key=len))+2
              for bdd in bdd.values()]

    for text_function in [t_title, t_content]:
        text_function(length, bdd)
        t_sep_line(length)


if __name__ == "__main__":
    DATA = get_text(FILE)
```

```
94
95      show(extract_regex(DATA))
96      show(extract_no_regex(DATA))
```

7.3 Explication du code

Il y a beaucoup de chose à dire ici, on va donc procéder étape par étape.

Contexte

Il s'agit de trouver le moyen d'extraire les données `nom`, `prénom`, `mail` et `téléphone` présentes dans un paragraphe, les stoker temporairement et les formater pour les afficher dans un tableau structuré.

Différentes problématiques

- **Extraction des données**

Il y a plusieurs façons de s'y prendre, soit à l'aide des *regexes* ou soit dans une logique *builtin* de *Python* grâce à quelques boucles et un peu de logique.

- **Sans regex**

Tout se passe dans la fonction :

`extract_no_regex(txt: str) -> dict}`

On déclare déjà le moyen de stockage des données à l'aide d'un dictionnaire (`bdd`) qui contiendra les clefs : `"Nom & Prénom"`, `"Mail"`, `"Numéro"`.

```
1 bdd = {k: [] for k in TITLE}
```

L'idée est de tout d'abord nettoyer les données pour permettre d'avoir des choses homogènes. Les numéros de téléphones sont en effet dans des formats très exotiques avec des points, des espaces et au nombre de chiffres différents en fonction d'un numéro contenant le préfixe international ou pas.

Toutes les extractions sont centrées dans une seule boucle : `for j, l in enumerate(txt_)`.

Les `noms` et `prénoms` sont extraits au travers de cette condition :

```
1 if l in PREFIX:
2   bdd[TITLE[0]].append(" ".join(txt_[j+1:j+3])[:-1])
```

Si l'on trouve un `PREFIX` tel que `"monsieur"`, `"Madame"`, `"mademoiselle"`, `"Mme"`, `"Mlle"`, `"M."` ou `"Dr"` alors on récupère les deux mots suivants que sont `nom` et `prénom`.

Les `emails` sont extraits en cherchant un `"@"` :

```
elif "@" in l:
  bdd[TITLE[1]].append(l.rstrip(",").rstrip("."))
```

De plus on nettoie le dernier caractère qui peut contenir un point ou une virgule.

Pour finir, l'extraction des `numéros de téléphone` se fait via :

```
elif any(n.isdigit() for n in l):
  bdd[TITLE[2]].append(format_tel(l[:12] if l.startswith("+")
    else l[:10]))
```

On cherche une chaîne de caractères contenant au moins un chiffre avec la fonction `any()` puis on récupère douze éléments s'il y a un `"+"` ou bien seulement dix chiffres.

Notez qu'au préalable que les données sont nettoyées en enlevant les points et les espaces entre les séries de chiffres des numéros de téléphone par cette condition :

```
...if t not in " ." or not txt[i-1].isdigit()...
```

- **Au moyen des *regexes***

Je vous invite à tester vos *regex* via ce site, sans oublier de cocher Python (à gauche) : https://regex101.com/.

La fonction qui permet de prendre en compte les *regex* se trouve ici, en une seule ligne `return` :

```
def extract_regex(txt: str)-> dict:
    return {t: next(REGEX).findall(txt) if i < 2 else \
               list(map(format_tel, next(REGEX).findall(
               "".join(t for t in txt if t not in " ."))) \
            for i, t in enumerate(TITLE)}
```

Les *regex* sont tout d'abord compilées puis utilisées avec `re.findall()` afin de pouvoir tout extraire d'un coup et éviter de surcharger le code de boucles inutiles.

La compilation [1] se fait directement à la déclaration des constantes, via l'utilisation d'un simple générateur :

```
REGEX = (compile(r) for r in (
             r'(?:monsieur|Madame|mademoiselle|Mme|Mlle|M.'
             r'|Dr)\s((?:[A-Z]\w+\s?){2})',
             r'[\w.]+@\w+\.\w{2,}',
             r'[+0]\d+'))
```

1. Pour plus d'information sur la compilation des regex : https://pynative.com/python-regex-compile/

Les `noms et prénoms` sont extraits par cette *regex* :

```
r'(?:monsieur|Madame|mademoiselle|Mme|Mlle|M.|Dr)\s((?:[A-Z]\w+\s?){2})'
```

- Un premier bloc qui me permet de chercher la civilité de la personne :

  `(?:monsieur|Madame|mademoiselle|Mme|Mlle|M.|Dr)`

  Notez le `?:` qui permet d'utiliser des parenthèses associatives sans en extraire le contenu. Ainsi le résultat de la recherche n'enverra pas ces données.

- Le `\s` pour chercher une espace.

- Un jeu de parenthèses pour permettre d'extraire proprement les données et d'éviter aussi des répétitions dans la *regex* :

  `((?:[A-Z]\w+\s?){2})`

Ainsi, on cherche ici ce pattern :

```
A-Z]\w+\s?){2}
```

qui est donc répété deux fois via `2`. Notez encore un `?` qui permet de définir le caractère optionnel espace noté `\s` qui est présent entre les noms et prénoms mais pas à la fin de la chaîne à extraire, cela devient donc facultatif.

Avec `[A-Z]` on cherche une majuscule puis un mot avec `\w+` et finalement une espace avec `\s`. Encore une fois le `?:` permet d'éviter d'avoir des doublons dans le résultat de la *regex*.

Les `emails` sont extraits grâce à cette *regex* :.

```
r'[\w.]+@\w+\.\w{2,}'
```

On cherche :

- Un mot pouvant contenir un point.
- Le caractère `"@"`.
- Un autre mot derrière.
- Un point.
- Un mot de deux lettres minimum pour l'extension de l'adresse `mail`.

Pour finir l'extraction des **numéros de téléphone** se fait via cette *regex* :

```
r'[+0]\d+'
```

- On cherche le signe `"+"` ou le chiffre `"0"` `[+0]`.
- Suivi de plusieurs chiffres `\d+`.

Au niveau du code dans mon `return`, j'utilise la fonction `next()` qui me permet d'incrément le pointeur du générateur et passer à la *regex* suivante. Ainsi mes deux premières extractions se font via : `next(REGEX).findall(txt)`. La dernière, elle, pour les numéros de téléphone se pose sur le texte nettoyé des points et des virgules, grâce à cette boucle :

```
"".join(t for t in txt if t not in " .")
```

Stokage temporaire des données

Avec ou sans *regex*, les fonctions d'extraction retournent toutes les deux un dictionnaire contenant les données suivantes :

```
{'Mail': ['thomas.b@alphamail.com',
          'cmartin@betainbox.org',
          'lp@experimentalpost.net',
          'sophie@prototypemail.com',
          'drdupont@randominbox.org',
          'laroche@trialmail.net',
          'sebastieng@demomail.org',
          'lefebvre_chloe@testinbox.net'],
 'Nom & Prénom': ['Thomas Bernard',
                  'Claire Martin',
                  'Lucas Petit',
                  'Sophie Martin',
                  'Lucas Dupont',
                  'Jennifer Laroche',
                  'Sébastien Girard',
                  'Chloé Lefebvre'],
 'Numéro': ['33.1.12.34.56.78',
            '09.01.23.45.67',
            '08.90.12.34.56',
            '07.89.01.23.45',
            '06.78.90.12.34',
            '05.67.89.01.23',
            '04.56.78.90.12',
            '03.45.67.89.01']}
```

Formatage des données

Les données des numéros téléphoniques sont à reformatées afin de respecter l'énoncer et c'est par l'appel de cette fonction que ça se passe :

```
1 def format_tel(tel: str) -> str:
2     tel_ = ("".join(tel[i:i+2] + "."
3                     for i in range(0, len(tel), 2)).
4             rstrip("."))
5     return tel_[1:2] + tel_[3:1:-1] + tel_[4:] \
6         if tel_.startswith("+") else tel_
```

Les numéros sont tout d'abord formatées dans un premier temps pour les numéros nationaux commençant par `0` :

```
1 tel_ = ("".join(tel[i:i+2] + "."
2                 for i in range(0, len(tel), 2)).rstrip("."))
```

On ajoute d'abord un point tous les deux chiffres pour formater un numéro national.

Quant aux numéraux internationaux (commençant par le symbole `"+"`) alors on doit intervertir un point et un chiffre : `tel_[3:1:-1]`. En d'autres termes, transformer ce format `+3.31.12.34.56.78` en celui-ci `33.1.12.34.56.78`.

Affichage

Là, c'est une grosse partie et je voudrais pour cela vous présenter, dans un but pédagogique, un peu les décorateurs qui ont leur importance en Python.

L'affichage est géré par la fonction `show(bdd: dict)-> None` qui fait appel à trois autres fonctions, chacune décorée de `@printl` :

```
def printl(func):
    def wrapper(*args):
        result = func(*args)
        print("|" + result)
        return result
    return wrapper

...

@printl
def t_title(*args):
    return "".join(f'{t:^{length}}|'
                   for t, length in zip(TITLE, args[0]))


@printl
def t_sep_line(*args):
    return "".join(f'{"-"*length}|' for length in args[0])


@printl
def t_content(*args):
    return "\n|".join("".join(f" {el:<{length-1}}|"
                             for el, length
                             in zip(el_, args[0]))
                      for el_ in zip(*args[1].values()))
```

En termes simples, un décorateur permet d'encapsuler une fonction, qu'il prend en paramètre, pour intégrer d'autres actions autour d'elle.

```
def printl(func):
    def wrapper(*args):
        result = func(*args)
        print("|" + result)
```

```
5           return result
6       return wrapper
```

Ici, je fais un `print("|" + result)`, cela me permet de simplifier la fonction `show()` et les formatages des contenus du tableau.

`*args` permet de récupérer les arguments de la fonction et si besoin de les utiliser dans le décorateur. Ici j'affiche ce que me renvoi la fonction en ajoutant un `"|"` au début.

Pour ajouter un décorateur, il suffit d'ajouter une ligne au-dessus de la fonction à décorer, Python s'occupe ensuite du reste.

```
1 @printl
2 def t_title(*args):
3     return "".join(f'{t:^{length}}|'
4                    for t, length in zip(TITLE, args[0]))
```

L'appel de cette fonction fait donc au final cela :

```
1 print("|" + "".join(f'{t:^{length}}|'
2                     for t, length in zip(TITLE, args[0])))
```

Ce serait donc équivalent à cela sans le décorateur :

```
1 def t_title(length:list)-> None:
2     print("|" + "".join(f'{t:^{l}}|'
3         for t, l in zip(TITLE, length)))
```

Au passage, cette fonction réunie les titres avec la taille des colonnes grâce à la fonction `zip()` puis on centre le tout avec les fonctions de formatage de *f-string*[2] et `:^` (pour centrer).

La deuxième fonction pour les lignes de séparation fonctionne de la même façon :

```
1 @printl
2 def t_sep_line(*args):
3     return "".join(f'{"-"*length}|' for length in args[0])
```

On affiche des `"-"` que l'on multiplie en fonction de la taille des colonnes.

Pour finir le contenu du tableau se fait par l'intermédiaire de deux boucles et pour le coup deux fois `zip()` (oui j'aime beaucoup les `zip()`) :

```
1 @printl
```

2. Pour plus d'informations sur les *f-strings*: https://discord.com/channels/396825382009044994/1155460501409628230

```
def t_content(*args):
    return "\n|".join("".join(f" {el:<{length-1}}|"
                              for el, length
                              in zip(el_, args[0]))
                      for el_ in zip(*args[1].values()))
```

- La première boucle `for el_ in zip(*args[1].values())` permet de lire chaque ligne du contenu du tableau en rangeant les données par personne.
- La deuxième permet, comme dans la fonction des titres, de créer un tuple pour associer les tailles de colonnes du tableau.
- Le formatage permet d'aligner à gauche (`:<`) et ajoute des espaces pour formater les données correctement dans les colonnes.

Pour revenir à la fonction `show()` :

```
def show(bdd: dict)-> None:
    length = [len(max(bdd, key = len))+2
              for bdd in bdd.values()]

    for text_function in [t_title, t_content]:
        text_function(length, bdd)
        t_sep_line(length)
```

La taille de chaque colonne est calculée par l'intermédiaire de `max(..., key = len)` qui est très pratique pour éviter de faire encore des boucles. Ainsi l'élément de la liste de taille la plus longue est alors directement renvoyée... C'est la fonction `len()` qui permet de récupérer la taille en question sous forme d'un `int`. `length` est donc une liste contenant la taille de chaque colonne du tableau.

Cette dernière fonctionnalité est un peu bonus pour éviter les répétitions de la ligne séparatrice du tableau :

```
    for text_function in [t_title, t_content]:
        text_function(length, bdd)
        t_sep_line(length)
```

J'itère donc sur les deux types d'éléments à ajouter dans le tableau : `t_title` et `t_content`, qui sont appelés avec les arguments `length` et `bdd`.

Puis après chaque type d'élément du tableau, une ligne séparatrice est ajoutée : `t_line()`.