

Visualiseur de texte – Markdown

Tu vas programmer un traitement de texte tout simple qui affiche proprement des paragraphes et met en évidence les mots en gras et en italiques.

[Vidéo ■ Visualiseur de texte - Markdown - partie 1](#)

[Vidéo ■ Visualiseur de texte - Markdown - partie 2](#)

[Vidéo ■ Visualiseur de texte - Markdown - partie 3](#)

Cours 1 (Texte avec tkinter).

Voici comment afficher du texte avec Python et le module des fenêtres graphiques tkinter.

Du texte avec Python !

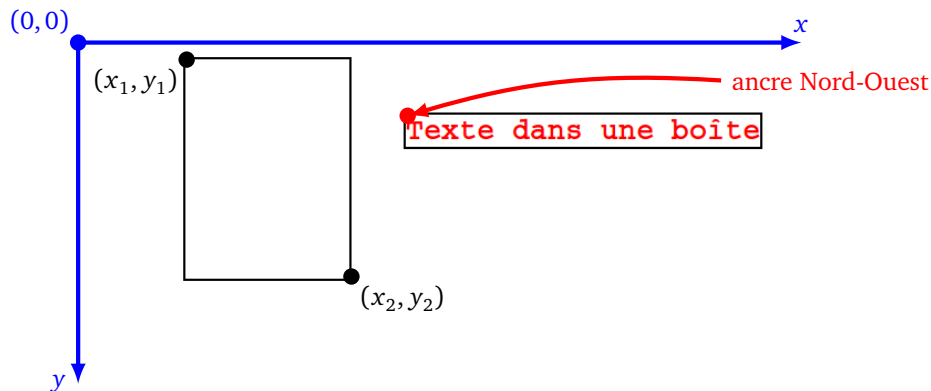
Le code est :

```
from tkinter import *
from tkinter.font import Font
# Fenêtre tkinter
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)
# Fonte
mafonte = Font(family="Times", size=20)
# Le texte
canvas.create_text(100,100, text="Du texte avec Python !",
anchor=NW, font=mafonte, fill="blue")
# Ouverture de la fenêtre
root.mainloop()
```

Quelques explications :

- root et canvas sont les variables qui définissent une fenêtre graphique (ici de largeur 800 et de hauteur 600 pixels). Cette fenêtre est visualisée par la commande de la fin : root.mainloop().
- On rappelle que pour le repère graphique l'axe des ordonnées est dirigé vers le bas. Pour définir un rectangle, il suffit de préciser les coordonnées (x_1, y_1) et (x_2, y_2) de deux sommets opposés (voir la figure ci-dessous).
- Le texte est affiché par la commande canvas.create_text(). Il faut préciser les coordonnées (x, y) du point à partir duquel on souhaite afficher le texte.
- L'option text permet de passer la chaîne de caractères à afficher.

- L'option `anchor` permet de préciser le point d'ancrage du texte, `anchor=NW` signifie que la zone de texte est ancrée au point Nord-Ouest (NW) (voir la figure ci-dessous).
- L'option `fill` permet de préciser la couleur du texte.
- L'option `font` permet de définir la fonte (c'est-à-dire le style et la taille des caractères). Voici des exemples de fontes, à toi de les tester :
 - `Font(family="Times", size=20)`
 - `Font(family="Courier", size=16, weight="bold")` en **gras**
 - `Font(family="Helvetica", size=16, slant="italic")` en *italique*



Activité 1 (Afficher un texte avec tkinter).

Objectifs : afficher du texte avec le module graphique tkinter.

Du texte avec Python

1. (a) Définis une fenêtre tkinter de taille 800×600 par exemple.
 (b) Dessine un rectangle gris (qui sera notre zone de texte) de taille largeur \times hauteur (par exemple 700×500).
 (c) Définis plusieurs types de fontes : `fonte_titre`, `fonte_sous_titre`, `fonte_gras`, `font_italique`, `fonte_texte`.
 (d) Affiche des textes avec différentes fontes.
2. Écris une fonction `encadre_mot(mot, fonte)` qui dessine un rectangle autour d'un texte. Pour cela utilise la méthode `canvas.bbox(monobjet)` qui renvoie les coordonnées x_1, y_1, x_2, y_2 du rectangle voulu. (Ici `monobjet = canvas.create_text(...)`).
3. Écris une fonction `longueur_mot(mot, fonte)` qui calcule la longueur d'un mot en pixels (c'est la largeur du rectangle de la question précédente).
4. Écris une fonction `choix_fonte(mode, en_gras, en_italique)` qui renvoie le nom d'une fonte adaptée (parmi celles définies en première question) selon un mode (parmi "titre", "sous_titre", "texte") et selon des booléens `en_gras`, `en_italique`.
 Par exemple `choix_fonte("texte", True, False)` renvoie la fonte `fonte_gras`.

Cours 2 (Markdown).

Le *Markdown* est un langage de balisage simple qui permet d'écrire un fichier texte propre et éventuellement de le convertir vers un autre format (html, pdf...).

Voici un exemple de fichier texte avec une syntaxe *Markdown* avec juste en dessous son rendu graphique.

```
# L'Origine des Espèces
## par Charles Darwin

Les rapports géologiques qui existent entre la ** faune
actuelle ** et la ** faune éteinte ** de l'Amérique
méridionale, ainsi que certains faits relatifs à la
distribution des êtres organisés qui peuplent ce continent,
m'ont profondément frappé lors de mon voyage à bord du
navire le * Beagle * en qualité de naturaliste.

** Chapitres **

+ De la variation des espèces à l'état domestique.
+ De la variation à l'état de nature.
+ La lutte pour l'existence.
+ La sélection naturelle ou la persistance du plus apte.
+ ...
```

L'Origine des Espèces
par Charles Darwin

Les rapports géologiques qui existent entre la **faune actuelle** et la **faune éteinte** de l'Amérique méridionale, ainsi que certains faits relatifs à la distribution des êtres organisés qui peuplent ce continent, m'ont profondément frappé lors de mon voyage à bord du navire le *Beagle* en qualité de naturaliste.

Chapitres

- De la variation des espèces à l'état domestique.
- De la variation à l'état de nature.
- La lutte pour l'existence.
- La sélection naturelle ou la persistance du plus apte.
- ...

La syntaxe est simple, avec un fichier texte bien lisible. Voici quelques éléments de cette syntaxe :

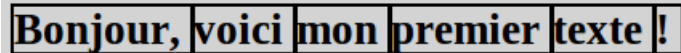
- un **texte en gras** s'obtient en entourant le texte par deux astérisques ** ;
- un *texte en italique* s'obtient en entourant le texte par un astérisque * ;
- la ligne d'un titre commence par dièse # ;
- la ligne d'un sous-titre commence par deux dièses ## ;
- pour les éléments d'une liste, chaque ligne commence par un symbole spécial, pour nous ce sera le symbole « plus » +.
- Il existe aussi une syntaxe pour afficher des liens, des tableaux, du code...

Dans la suite nous utiliserons la syntaxe simplifiée comme elle est décrite ci-dessus.

Activité 2 (Visualiser du Markdown).

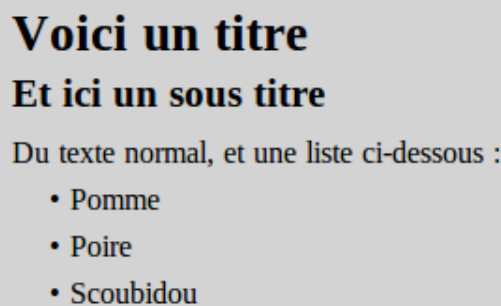
Objectifs : visualiser du texte avec la syntaxe Markdown simplifiée.

1. Écris une fonction `afficher_ligne_v1(par, posy)` qui affiche *un par un* les mots d'un paragraphe `par` (sur la ligne d'ordonnée `posy`).



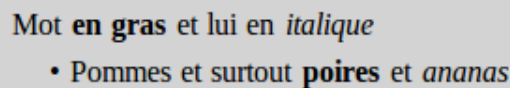
Indications :

- Ces mots sont obtenus grâce à la commande `par.split()`.
 - La ligne affichée commence tout à gauche, elle déborde à droite si elle trop longue.
 - Après chaque mot on place une espace puis le mot suivant.
 - Sur l'image ci-dessus les mots sont encadrés.
2. Améliore ta fonction en `afficher_ligne_v2(par, posy)` pour tenir compte des titres, sous-titres et listes.



Indications :

- Pour savoir dans quel mode il faut afficher la ligne, il suffit de tester les premiers caractères de la ligne. La ligne d'un titre commence par `#`, celle d'un sous-titre par `##`, celle d'une liste par `+`.
 - Pour les listes, tu peux obtenir le caractère « `•` » par le caractère unicode `u'\u2022'`. Tu peux aussi indenter les éléments de la liste pour plus de lisibilité.
 - Utilise la fonction `choix_fonte()` de la première activité.
 - Sur l'image ci-dessus, chaque ligne est produite par un appel à la fonction. Par exemple `afficher_ligne_v2("## Et ici un sous titre", 100)`
3. Améliore encore ta fonction en `afficher_ligne_v3(par, posy)` pour tenir compte des mots en gras et en italique dans le texte.



Indications :

- Les mots en gras sont entourés par la balise `**`, les mots en italique par la balise `*`. Dans notre syntaxe simplifiée, les balises sont séparées des mots par des espaces, par exemple : `"Mot ** en gras ** et lui en * italique *"`.
- Définis une variable booléenne `en_gras` qui est fausse au départ ; chaque fois que tu rencontres la balise `**` alors inverse la valeur de `en_gras` (« vrai » devient « faux », « faux » devient « vrai »). Tu peux utiliser l'opérateur `not`.
- Utilise encore la fonction `choix_fonte()` de la première activité.
- Sur l'image ci-dessus, chaque ligne est produite par un appel à la fonction. Par exemple `afficher_ligne_v3("+ Pommes et surtout ** poires ** et * ananas *", 100)`

4. Améliore encore ta fonction en `afficher_paragraphe(par, posy)` qui gère l’affichage d’un paragraphe (c’est-à-dire une chaîne de caractères qui peut être très longue) sur plusieurs lignes.

[illegible]

Indications :

- Dès que l'on place un mot qui dépasse la longueur de la ligne (voir ceux qui sortent du cadre sur l'image ci-dessus), alors le mot suivant est placé sur la ligne suivante.
 - La fonction va donc modifier la variable `posy` à chaque saut de ligne. À la fin, la fonction retourne la nouvelle valeur de `posy`, qui sera utile pour afficher le paragraphe suivant.
5. Termine par une fonction `afficher_fichier(nom)` qui visualise les paragraphes d'un fichier texte ayant la syntaxe *Markdown* simplifiée.

Activité 3 (Justification).

Objectifs : comprendre comment il est possible de « justifier » un texte, c'est-à-dire de faire en sorte que les mots soient bien alignés sur la partie gauche et la partie droite de la page. Pour modéliser le problème nous travaillons avec une suite d'entiers qui représente les longueurs de nos mots.

Dans cette activité :

- `longueurs` est une liste d'entiers (par exemple une liste de 50 entiers compris entre 5 et 15) qui représentent les longueurs des mots ;
- on fixe une constante `longueur_ligne` qui représente la longueur d'une ligne. Pour nos exemples, cette longueur vaut 100.

Dans les activités précédentes, nous passions à la ligne suivante après qu'un mot ait dépassé la fin de ligne. Nous représentons ceci par la figure suivante :

8	11	9	14	8	8	15	10	14	13	somme = 108			
15	15	5	12	9	9	15	10	14		somme = 104			
5	12	8	8	13	10	11	8	13	7	5	somme = 100		
6	11	7	7	13	6	6	9	8	12	5	8	7	somme = 105
6	6	15	13	11	7	12							somme = 70

Tu vas essayer de placer les mots plus joliment !

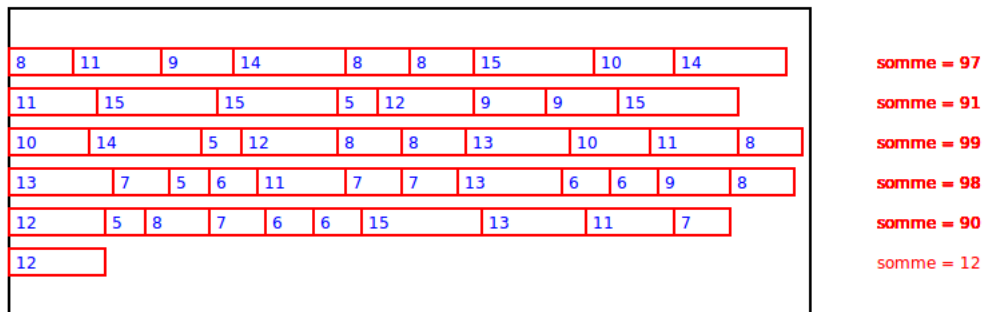
Les dessins sont construits sur la base de l'exemple :

```
longueurs = [8, 11, 9, 14, 8, 8, 15, 10, 14, 11, 15, 15, 5, 12, 9, 9, 15, 10, 14, 5, 12, 8, 8, 13, 10, 11, 8,
             13, 7, 5, 6, 11, 7, 7, 13, 6, 6, 9, 8, 12, 5, 8, 7, 6, 6, 15, 13, 11, 7, 12]
```

qui a été obtenu par un tirage aléatoire grâce aux commandes :

```
from random import randint
longueurs = [randint(5,15) for i in range(50)]
```

1. Écris une fonction `coupures_simples()` qui calcule les indices permettant de réaliser les coupures correspondant à la figure ci-dessous, c'est-à-dire un alignement à gauche (sans espaces) et sans dépasser la longueur totale de la ligne (ici de longueur 100).



coupures_simples()

Usage : `coupures_simples(long)`

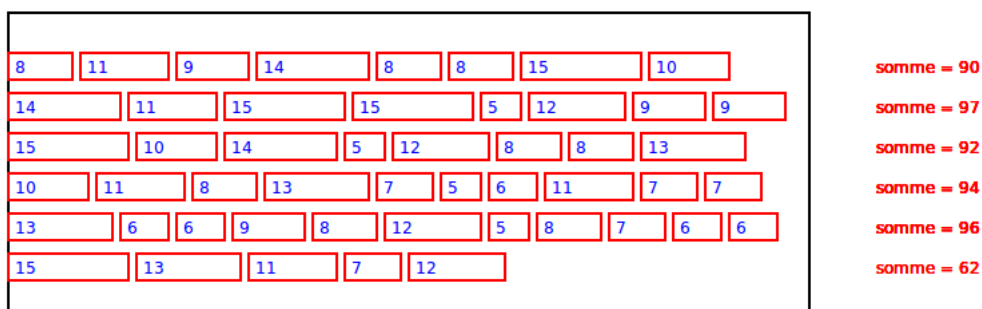
Entrée : une suite de longueurs (une liste d'entiers)

Sortie : la liste des indices où effectuer une coupure

Exemple : `coupures_simples(longueurs)` où `longueurs` est l'exemple donné ci-dessus, renvoie la liste `[0, 9, 17, 27, 39, 49, 50]`. C'est-à-dire que :

- la première ligne correspond aux indices 0 à 8 (donnés par `range(0,9)`),
- la deuxième ligne correspond aux indices 9 à 16 (donnés par `range(9,17)`),
- la troisième ligne correspond aux indices 17 à 26 (donnés par `range(17,27)`),
- ...
- la dernière ligne correspond à l'indice 49 (donné par `range(49,50)`).

2. Modifie ton travail en une fonction `coupures_espaces()` qui rajoute une espace (avec `longueur_espace = 1`) entre deux mots d'une même ligne (mais pas au début de ligne, ni à la fin de la ligne). Cela correspond au dessin suivant :



Pour notre exemple, les coupures renvoyées sont [0, 8, 16, 24, 34, 45, 50].

3. Afin de pouvoir justifier le texte, tu autorises les espaces à être plus grandes que la longueur initiale de 1. Sur chaque ligne, les espaces entre les mots sont toutes de la même longueur (supérieure ou égale 1) de sorte que le dernier mot soit aligné à droite. D'une ligne à l'autre, la longueur des espaces peut changer.

8	11	9	14	8	8	15	10	somme = 100.0
14	11	15	15	5	12	9	9	somme = 100.0
15	10	14	5	12	8	8	13	somme = 100.0
10	11	8	13	7	5	6	11	somme = 100.0
13	6	6	9	8	12	5	8	somme = 100.0
15	13	11	7	12				somme = 62

Écris une fonction `calcul_longueur_espaces()` qui renvoie la longueur que doivent avoir les espaces de chaque ligne pour que le texte soit justifié. Pour notre exemple, on obtient la liste [2.43, 1.43, 2.14, 1.67, 1.40, 1.00], c'est-à-dire que pour la première ligne les espaces doivent être de longueur 2.43, pour la seconde 1.43,...

Pour trouver la formule qui convient, il suffit de reprendre les résultats de la fonction `coupures_espaces()` puis, pour chaque ligne, compter le nombre de mots qu'elle contient, ainsi que la valeur qu'il manque pour arriver au total de 100.

Tu as maintenant tout en main pour visualiser du texte écrit avec la syntaxe Markdown et le justifier. Cela représente quand même encore du travail ! Tu peux aussi améliorer la prise en charge de la syntaxe Markdown : prendre en charge le code, les listes numérotées, les sous-listes, les mots en gras et en italique en même temps...