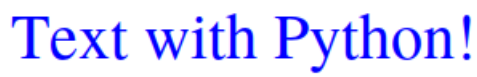
Text viewer – Markdown

You will program a simple word processor that displays paragraphs cleanly and highlights words in bold and italics.

Lesson 1 (Text with tkinter)**.**
Here's how to display text with `Python` and the graphics window module `tkinter`.



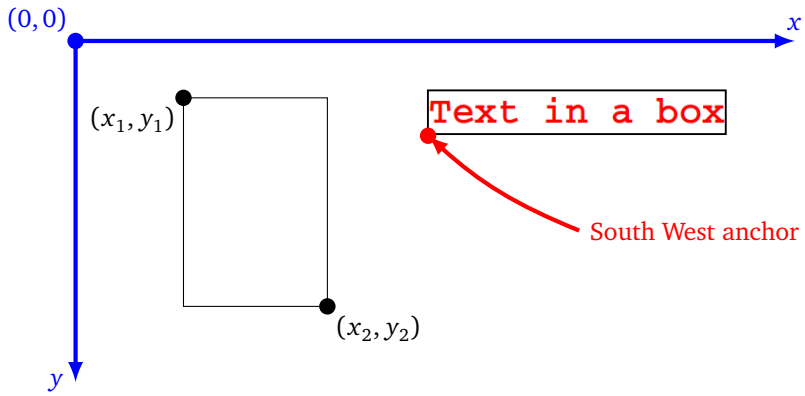
The code is:

```
from tkinter import *
from tkinter.font import Font
# tkinter window
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)
# Font
myfont = Font(family="Times", size=30)
# Some text
canvas.create_text(100,100, text="Text with Python!",
anchor=SW, font=myfont, fill="blue")
# Launch the window
root.mainloop()
```

Some explanations:

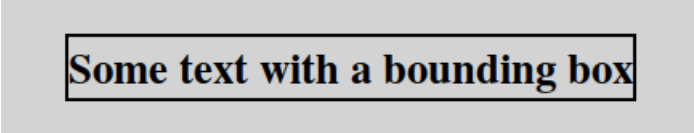
- `root` and `canvas` are the variables that define a graphic window (here of width 800 and height 600 pixels). This window is launched by the last command: `root.mainloop()`.
- We remind you that for the graphic coordinates, the y-axis is directed downwards and the origin is the top-left corner. To define a rectangle, simply specify the coordinates (x_1, y_1) and (x_2, y_2) from two opposite vertices (see figure below).
- The text is displayed by the `canvas.create_text()` command. It is necessary to specify the coordinates (x, y) of the point from which you want to display the text.
- The `text` option allows you to pass the string to display.
- The `anchor` option allows you to specify the text anchor point, `anchor=SW` means that the text box is anchored to the Southwest point (*SW*) (see figure below).

- The `fill` option allows you to specify the text color.
- The option `font` allows you to define the font (i.e. the style and size of the characters). Here are some examples of fonts, it's up to you to test them:
  - `Font(family="Times", size=20)`
  - `Font(family="Courier", size=16, weight="bold")` in **bold**
  - `Font(family="Helvetica", size=16, slant="italic")` in *italics*



Activity 1 (Display a text with tkinter).

Goal: display text with the graphics module `tkinter`.



1. (a) Define a `tkinter` window of size 800 × 600 for example.
   (b) Draw a gray rectangle (which will be our background box) with a size of `back_width` × `back_height` (for example 700 × 500).
   (c) Define several types of fonts: `title_font`, `subtitle_font`, `bold_font`, `italic_font`, `text_font`.
   (d) Display texts with different fonts.
2. Write a `text_box(word,font)` function that draws a rectangle around a text. To do this, use the `canvas.bbox(myobject)` method which returns the x_1, y_1, x_2, y_2 coordinates of the desired rectangle. (Here `myobject = canvas.create_text(...)`).
3. Write a `length_word(word,font)` function that calculates the length of a word in pixels (this is the width of the rectangle from the previous question).
4. Write a `font_choice(mode,in_bold,in_italics)` function that returns the name of an adapted font (among those defined in the first question) according to a mode (among `"title"`, `"subtitle"`, `"text"`) and according to booleans `in_bold`, `in_italics`.
   For example, `font_choice("text",True,False)` returns the font `bold_font`.

Lesson 2 (Markdown).
The *Markdown* is a simple markup language that allows you to write your own easy to read text file and possibly convert it to another format (html, pdf...).

Here is an example of a text file with the *Markdown* syntax with its graphic rendering just below.

```
# On the Origins of Species

## by Charles Darwin

When on board H.M.S. * Beagle * as naturalist, I was much struck with
certain facts in the distribution of the inhabitants of South America,
and in the geological relations of the present to the past inhabitants
of that continent. These facts seemed to me to throw some light on the
** origin of species ** that mystery of mysteries, as it has been called
by one of our greatest philosophers.

## Chapters

+ Variation under domestication.
+ Variation under nature.
+ Struggle for existence.
+ Natural selection.
+ ...
```

On the Origins of Species

by Charles Darwin

When on board H.M.S. *Beagle* as naturalist, I was much struck with certain facts in the distribution of the inhabitants of South America, and in the geological relations of the present to the past inhabitants of that continent. These facts seemed to me to throw some light on the **origin of species** that mystery of mysteries, as it has been called by one of our greatest philosophers.

Chapters

- Variation under domestication.
- Variation under nature.
- Struggle for existence.
- Natural selection.
- ...

The syntax is simple, with a clear and clean text file. Here are some elements of this syntax:

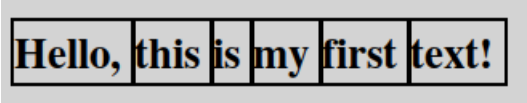
- a **bold text** is obtained by surrounding the text with two asterisks `**`;
- a *text in italics* is obtained by surrounding the text with one asterisk `*`;
- the line of a title begins with one hashtag `#`;
- the line of a subtitle begins with two hashtags `##`;
- for the elements of a list, each line starts with a special symbol, for us it will be the “plus” symbol `+`.
- There is also syntax to display links, tables, code…

In the following we will use the simplified syntax as described above.

Activity 2 (View *Markdown*)**.**

Goal: display text with our simplified Markdown *syntax.*

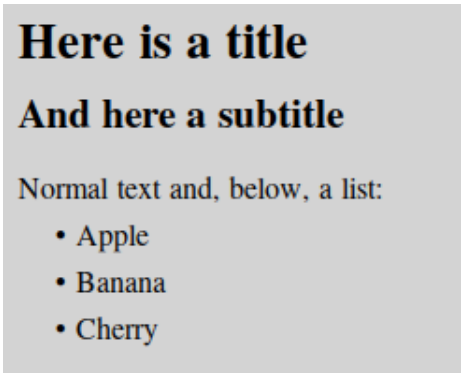
1. Write a function `print_line_v1(par,posy)` that displays *one by one* the words of a paragraph `par` (at the line of ordinate `posy`).



Hints.

- These words are obtained one by one with the command `par.split()`.
- The displayed line starts at the very left, it overflows to the right if it is too long.
- After each word we place a space and then the next word.
- In the picture above the words are framed.

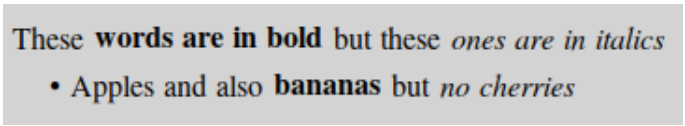
2. Improve your function in `print_line_v2(par,posy)` to take into account titles, subtitles and lists.



Hints.

- To know which mode to use when displaying the line, simply test the first characters of the line. The line of a title begins with #, that of a subtitle with ##, that of a list with +.
- For lists, you can get the bullet point character "•" by the unicode character `u'\u2022'`. You can also indent each item of the list for more readability.
- Use the `font_choice()` function from the first activity.
- In the image above, each line is produced by a call to the function. For example `print_line_v2("## And here a subtitle",100)`.

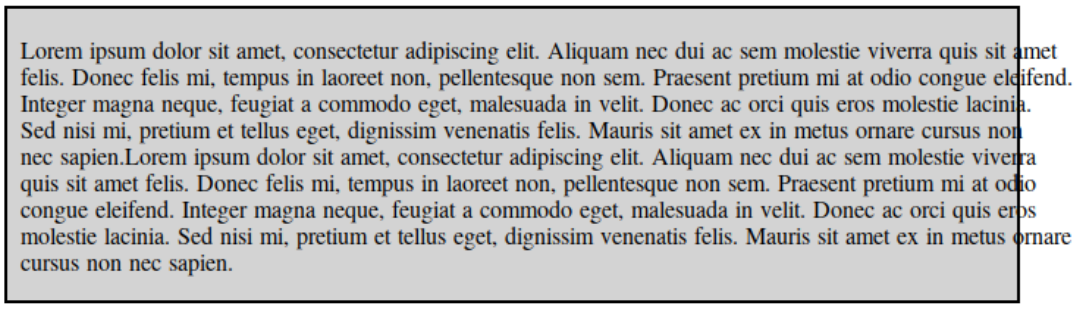
3. Further improve your function in `print_line_v3(par,posy)` to take into account the words in bold and italics in the text.



Hints.

- Words in bold are surrounded by the tag `**`, words in italics by the tag `*`. In our simplified syntax, tags are separated from words by spaces, for example: `"Words ** in bold ** and in * italics *"`
- Define a boolean variable `in_bold` that is false at the beginning; each time you encounter the tag `**` reverse the value of `in_bold` ("True" becomes "False", "False" becomes "True", you can use the `not` operator).
- Still use the `font_choice()` function from the first activity.
- In the image above, each line is produced by a call to the function. For example `print_line_v3("+ Apples and also ** bananas ** but * no cherries *",100)`

4. Further improve your function in `print_paragraph(par,posy)` which manages the display of a paragraph (i.e. a string that can be very long) on several lines.



Hints.

- As soon as you place a word that exceeds the length of the line (see those that come out of the frame in the image above), then the next word is placed on the next line.
- The function will therefore modify the variable `posy` at each line break. At the end, the function returns the new value of `posy`, which will be useful for displaying the next paragraph.

5. End with a `print_file(filename)` function that displays the paragraphs of a text file with our simplified *Markdown* syntax.

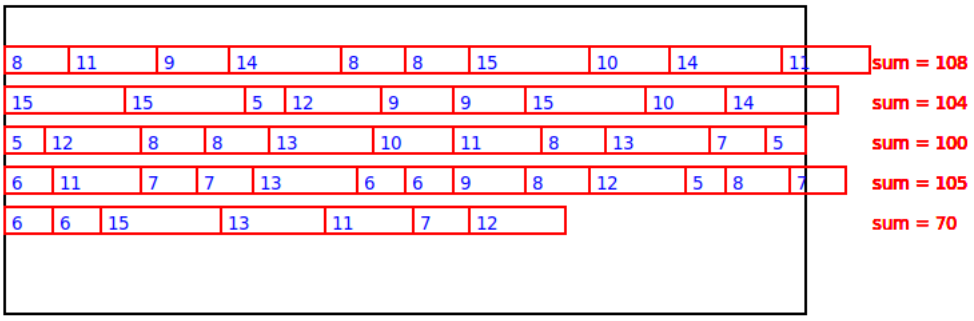
Activity 3 (Justification)**.**

Goal: understand how it is possible to "justify" a text, i.e. to ensure that the words are well aligned on the left and right sides of the page. To model the problem we work with a series of integers that represent the lengths of our words.

In this activity:

- `list_lengths` is a list of integers (for example a list of 50 integers between 5 and 15) which represent the lengths of the words;
- we set a constant `line_length` which represents the length of a line. For our examples, this length is equal to 100.

In previous activities, we moved to the next line after a word had passed the end of the line. We represent this by the following figure:



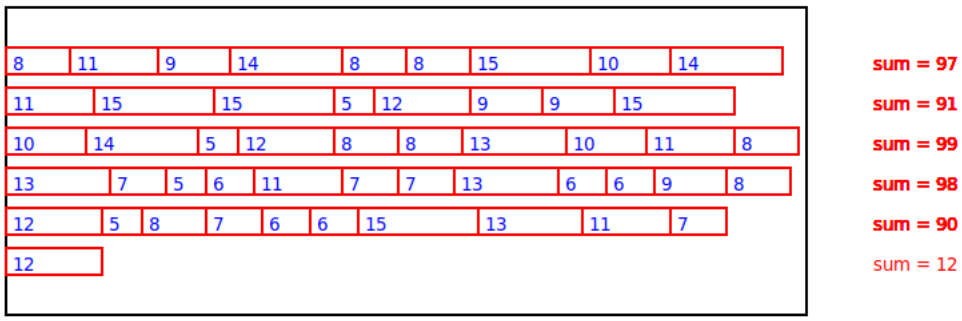
You're going to try to place the words more nicely!
The drawings are based on the example:

```
list_lengths = [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]
```

which was obtained by random integers:

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

1. Write a `justification_simple()` function that calculates the indices at which to make the text alignment corresponding to the figure below, i.e. an alignment on the left (without spaces) and without exceeding the total length of the line (here of length 100).



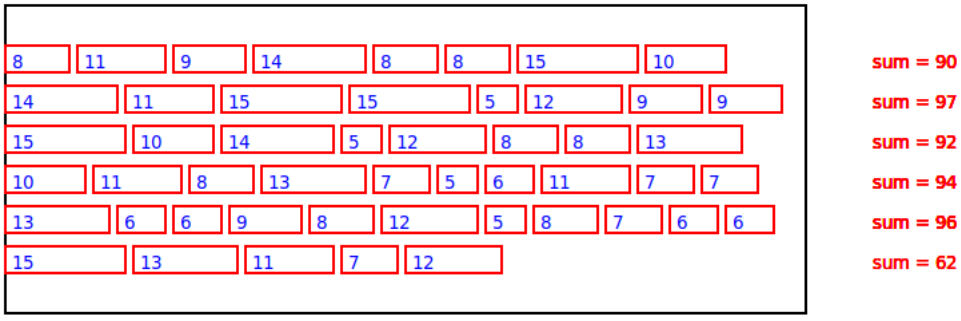
`justification_simple()`

Use: `justification_simple(list_lengths)`
Input: a sequence of lengths (a list of integers)
Output: the list of indices where to make a cut

Example: `justification_simple(list_lengths)` where `list_lengths` is the example given above, returns the list `[0, 9, 17, 27, 39, 49, 50]`. That is to say that:

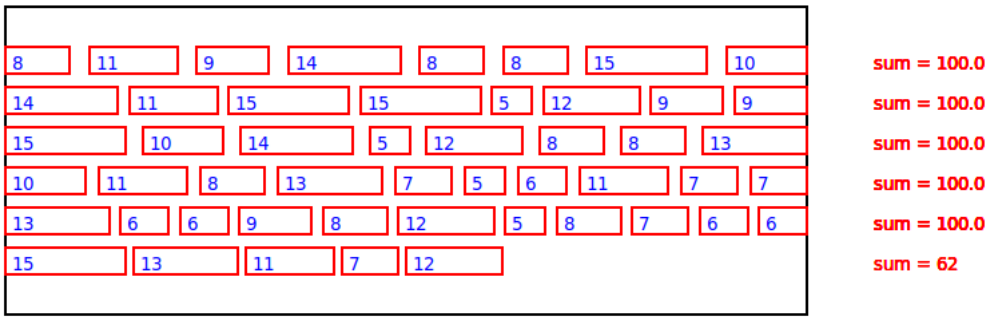
- the first line corresponds to indices 0 to 8 (given by `range(0,9)`),
- the second line corresponds to indices 9 to 16 (given by `range(9,17)`),
- the third line corresponds to indices 17 to 26 (given by `range(17,27)`),
- ...
- the last line corresponds to the index 49 (given by `range(49,50)`).

2. Modify your work into a `justification_spaces()` function that adds a space (with `space_length = 1`) between two words of the same line (but not at the beginning of the line, nor at the end of the line). This corresponds to the following drawing:



For our example, the justifications returned are `[0, 8, 16, 24, 34, 45, 50]`.

3. In order to be able to justify the text, you allow spaces to be larger than the initial length of 1. On each line, the spaces between the words are all the same length (greater than or equal to 1) so that the last word is aligned on the right. From one line to another, the length of the spaces can change.



Write a `compute_space_lengths()` function that returns the length that the spaces of each line must have in order for the text to be justified. For our example, we obtain the list `[2.43, 1.43, 2.14, 1.67, 1.40, 1.00]`, i.e. for the first line the spaces must be 2.43 long, for the second line 1.43,...
To find the right formula, simply take the results of the `justification_spaces()` function and then, for each line, count the total length of words it contains, as well as the value it lacks to arrive at a total of 100.

You now have everything you need to visualize text written with the Markdown *syntax and justify it (see Darwin's text displayed text in the lesson). It's still a lot of work! You can also improve the support of the* Markdown *syntax: code, numbered lists, sublists, bold and italic words at the same time...*