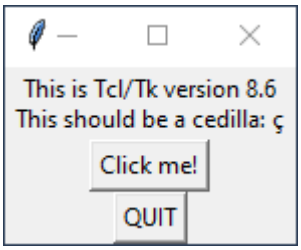
Tkinter

Pour tester son installation

- Dans votre terminal, lancer la commande suivante :

```
python -m tkinter
```

- Si la fenêtre suivante apparaît, alors **tkinter** est bien installé sur votre système.



Fonctionnement de *Tkinter*

- Toute les classes de **Tkinter** sont situées dans le module **tkinter**.
- La première étape nécessite d'importer le module :

```
import tkinter as tk
```

- L'élément visuel de base est la fenêtre. Cet élément **racine** est une instance de la classe `Tk` du module **tkinter**.

```
root = tk.Tk()
```

- Pour qu'une fenêtre s'affiche, il faut lancer la **boucle principale**.
    - Cette boucle permet de rester à l'écoute des évènements : clique, touche appuyer, etc.
    - Elle permet à la fenêtre de se mettre à jour.
    - La boucle se lance par une simple commande :

```
root.mainloop()
```

La fenêtre *Tkinter*

- La méthode `title("str")` de la classe `Tk` permet de modifier le titre de la fenêtre.

```
root.title("Ma première fenêtre tkinter")
```

- La méthode `geometry("chaine_formatée")` permet de modifier la taille de la fenêtre.
    - La chaîne en paramètre utilise le formatage (minimal) suivant :

```
{taille horizontale en px}x{taille verticale en px}
```

Les widgets Tkinter - Le fonctionnement générique

- Les **widgets** sont les éléments graphiques. Ainsi, en tkinter, tout est un widget.
    - La fenêtre est souvent appelée le **widget principal**.
- Tous les widgets sont créés à partir d'une classe dédiée.
    - Le constructeur de ces classes a toujours la même forme :

```
un_widget = tk.UnWidget(widget_principal, autres_paramètres = "valeur")
```

- La méthode `config` permet de modifier un paramètre du widget après sa création.
    - Les paramètres varient selon le type de widget.

```
un_widget.config(un_parametre='valeur')
```

- La méthode `cget` permet de récupérer la valeur d'un paramètre

```
un_widget.cget('un_parametre')
```

- Pour s'afficher, un widget doit être positionné dans le widget parent.
- Il existe trois positionnements possibles :
    - **pack** : Le widget parent sera coupé en deux. La première partie sera la zone occupée par le widget. La seconde partie sera la partie libre.
    - **place** : Permet de positionner le widget au pixel près.
    - **grid** : Permet de positionner le widget sur une grille programmée par l'utilisateur.
- Pour placer un élément sur une grille, il suffit d'appeler la méthode `grid`.

```
un_widget.grid(column=1, row=1) # Place le widget à la deuxième colonne et à la deuxième ligne
un_widget.grid() # Place le widget automatiquement
```

- Tkinter possède plusieurs paramètres de base aux widgets :
  - **foreground** : Couleur du texte du widget
  - **activeforeground** : Couleur du texte lorsque le curseur survole le widget
  - **background** : Couleur de fond du widget
  - **activebackground** : Couleur de fond du widget lorsque le curseur survole le widget
  - **padx** : Marge horizontale entre les bords du widget et de son widget parent
  - **pady** : Marge verticale entre les bords du widget et de son widget parent
  - **width** : Largeur du widget en taille de police
  - **height** : Hauteur du widget en taille de police

Les widgets Tkinter - Quelques widgets de base

Les étiquettes `Label`

- Les étiquettes sont créées à partir de la classe `Label`.
- Le paramètre `text` défini la chaîne que le widget affichera.
- Le paramètre `justify` permet de modifier l'alignement du texte (`tk.LEFT` ou `tk.RIGHT`)

```
label = tk.Label(root, text='Hello world', foreground='#892222',
                 background='#FFAAAA', activebackground="#FFBBBB", padx="10", pady="10")
```

Les boutons `Button`

- Les boutons sont créés à partir de la classe `Button`.
- Le paramètre `text` défini la chaîne qui s'affichera dans le bouton.
- Le paramètre `command` définit la fonction à exécuter lorsque l'utilisateur clique sur le bouton.

```
def foo() :
    print("click")

button = tk.Button(root, text='Cliquez sur moi', command=foo)
```

- Pour appeler une fonction avec des paramètres, il faut encapsuler la fonction et les paramètres grâce à la fonction `partial` du module `functools`.

```
from functools import partial

def bar(label, msg) :
    label.config(text=msg)

etiquette = tk.Label(root, text='Ce msg va bientôt changer')
btn = tk.Button(root, text='On change le Label', command=partial(bar, etiquette, "Le nouveau message"))
```

Les champs de saisie `Entry`

- Les champs de saisie (texte) sont créés à partir de la classe `Entry`.
- Le paramètre `textvariable` permet de récupérer la valeur du champ.
    - Il est de type `StringVar`. Le constructeur de `StringVar` prend le widget principal comme paramètre.
    - Pour récupérer le texte d'un `StringVar`, il suffit d'appeler sa méthode `get`.
    - Si la variable est modifiée, la valeur du champ est alors modifiée.

```
texte_champ = tk.StringVar(root)
champ = tk.Entry(root, textvariable=texte_champ)
```

Les cases à cocher `Checkbutton`

- Les cases à cocher sont créées à partir de la classe `Checkbutton`.
- Le paramètre `variable` permet de récupérer la valeur du champ (cocher ou non).
    - Il est de type `BooleanVar`. Le constructeur de `BooleanVar` prend le widget principal et la valeur du champ (`True` ou `False`) comme paramètres.
    - Pour récupérer le texte d'un `BooleanVar`, il suffit d'appeler sa méthode `get`.
    - Si la variable est modifiée, la valeur du champ est alors modifiée.

```
is_checked = tk.BooleanVar(root, False)
checkbox = tk.Checkbutton(root, text="Changer le champs ?", variable=is_checked)
```

Les widgets Tkinter - Les conteneurs

- Ces widgets permettent de contenir d'autre widget comme pour la fenêtre. Le placement est alors simplifié.
- Leurs utilisations pour les widgets communs reste identiques que pour une fenêtre.

`Frame`

- Possède deux attributs intéressants :
    - **relief** : type de bordure (voir les options possibles)
    - **bd** : Taille, en pixel, de la bordure (par défaut, 2 pixels)
    - **bg** : Couleur normale de fond
    - **height** : hauteur du frame
    - **width** : Largeur du frame

```
import tkinter as tk

root = tk.Tk()
root.geometry("280x280")
# Pas de bordure
f1 = tk.Frame(root, bd=0)
# Petit message
label = tk.Label(f1, text="Hello world")
# Étiquette centré
label.pack(side=tk.LEFT)

f1.pack()
root.mainloop()
```

Les évènements

- Pour « écouter » un évènement, il faut ajouter cette écoute grâce à la fonction `bind` d'un widget.
- Il existe plusieurs évènements différents :
  - `<ButtonPress>` : Pression d'un bouton de la souris ;
    - `<ButtonPress-1>` : Représente le bouton gauche;
    - `<ButtonPress-2>` : Représente le bouton du centre;
    - `<ButtonPress-3>` : Représente le bouton droit;
  - `<ButtonRelease>` : Relâchement d'un bouton de la souris ;
  - `<KeyPress>` : Pression d'un bouton du clavier ;
    - `<KeyPress-a>` : Pression du bouton **a** du clavier ;
  - `<KeyRelease>` : Relâchement d'un bouton du clavier ;
- Il existe aussi des modificateurs :
  - `Control` : la touche Control sera pressée ;
    - `<Control-KeyPress-s>` : Les touches **contrôles** et **s** ont été appuyées.
  - `Shift` : la touche Shift sera pressée ;
  - `Alt` : la touche Alt sera pressée ;
  - `Double` : double clique
  - `Triple` : triple clique
  - `Quadruple` : quadruple clique
- La fonction à appeler doit toujours prendre un évènement `event` en paramètre

```
def foo(texte, event) :
    print(texte)

root.bind("<ButtonPress-1>", partial(foo, "Click !"))
```

- Pour avoir la position du d'un évènement, il suffit d'appeler les attributs `x` et `y` de `event`.

Les couleurs en Tkinter

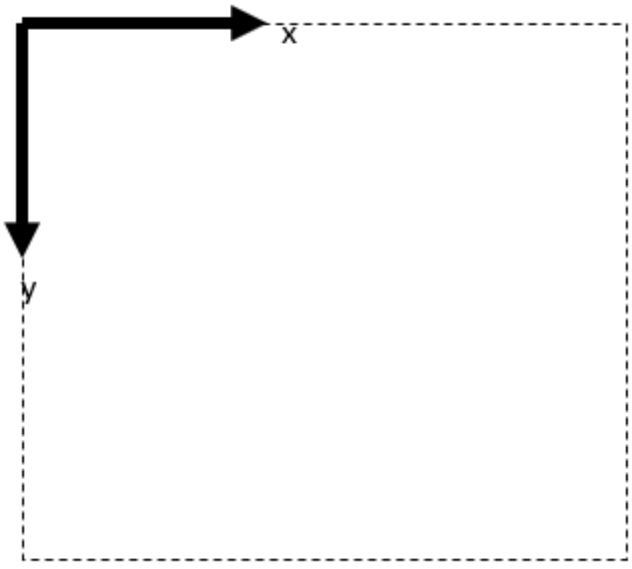
- Concernant les couleurs, Tkinter accepte les codes hexadécimaux.

    - Les codes abrégés sont aussi supportés : `#665599 == # 659`.
- La bibliothèque accepte aussi les couleurs nommées comme sur l'image suivante :

- Voir la page suivante pour le code source qui génère cette application.

Les formes, la classe `Canvas`

- La classe `Canvas` est un widget qui permet de dessiner des formes.
- Les positions sont définies par un plan cartésien.
    - L'origine est en haut à gauche.
    - La variation de l'axe `x` (largeur) est positive vers la droite.
    - La variation de l'axe `y` (hauteur) est positive vers le bas.



- La fonction `create_line((x1, y1), (x2, y2), ..., (xn, yn))` permet de dessiner une ou plusieurs lignes.

    - Les tuples `(x, y)` représentent les positions d'un sommet.
- Le code suivant dessine une ligne qui part du point `(10, 10)` et qui se rend à la position `(10, 200)`.

```
import tkinter as tk
root = tk.Tk()
root.geometry("280x280")

ligne = tk.Canvas(root, background='white')
ligne.create_line((10, 10), (200, 10))

ligne.grid()
root.mainloop()
```

- Le code suivant créer un triangle avec une base de 50 pixels et une hauteur de 50 pixels.

```
import tkinter as tk
root = tk.Tk()
root.geometry("280x280")

ligne = tk.Canvas(root, background='white')
ligne.create_line((10, 60), (60, 60), (35, 10), (10, 60), fill="blue")

ligne.grid()
root.mainloop()
```

- Il est aussi possible de créer un rectangle :

```
carre = tk.Canvas(root, background='white')
carre.create_retangle((10,10), (50,50))
```

- Il est aussi possible de lui ajouter une couleur de fond :

```
carre = tk.Canvas(root, background='white')
carre.create_retangle((10,10), (50,50), fill='red')
```

- Ainsi que des bordures :

```
carre = tk.Canvas(root, background='white')
carre.create_retangle((10,10), (50,50), width=3, outline='green')
```

- Pour ce simplifier la vie, nous pouvons utiliser la bibliothèque **C31Geometry**.

La bibliothèque `C31Geometry`

- Cette bibliothèque fournit une série de classe qui permet de dessiner plus facilement des formes de base.
    - La bibliothèque possède la classe `Forme`. Cette dernière est inutilisable. Il s'agit du comportement commun à toutes les classes de `C31Geometry`.
- Voici un exemple complet d'utilisation. Son fonctionnement sera expliqué après.

```
import tkinter as tk
```

```
from functools import partial
from c31Geometry import *

def foo(line, multiplicateur, root, event):
    line.resize(multiplicateur)
    line.draw()
    #root.update()
def bar(line, add, root, event):
    line.localRotate(add)
    line.draw()
    #root.update()

root = tk.Tk()
root.title("Les canvas en Tkinter")
root.geometry("280x280")

btn_plus = tk.Button(root, text="Augmenter")
btn_minus = tk.Button(root, text="Diminuer")

btn_plus.grid()
btn_minus.grid()

########################################
# Zone Canevas
########################################

canvas = tk.Canvas(root, background='white')

ligne = Ligne(canvas, Point(50, 50), 200, 0)
#ligne = Rectangle(canvas, Point(100, 100), 50, 25)
#ligne = Carre(canvas, Point(100, 155), 50)
#ligne = Cercle(canvas, Point(100, 100), 50)
#ligne = Croix(canvas, Point(100, 100), 50)

ligne.draw()
canvas.grid()
canvas.bind("<ButtonPress-1>", partial(foo, ligne, 1.1, root))
canvas.bind("<ButtonPress-3>", partial(foo, ligne, 0.1, root))
root.bind("<KeyPress-e>", partial(bar, ligne, 1, root))
root.bind("<KeyPress-q>", partial(bar, ligne, -1, root))
root.mainloop()
```

Fonctionnement de base (commune à toutes les classes)

- Pour toutes les classes de la bibliothèque, vous devez créer, au préalable, un widget `Canvas`. Pour chaque constructeur, il s'agit du premier paramètre à fournir.

```
canvas = tk.Canvas(root, background='white')
ligne = Ligne(canvas, ...)
```

- Les objets `Point` représentent les coordonnées dans la bibliothèque. À chaque fois qu'une position ou un déplacement est demandé, vous devez fournir un objet de type `Point`.
    - L'objet de type `Point` gère les opérations vectorielles.
    - Le constructeur de `Point` prend deux paramètres : sa position en `x` et sa position en `y`.

- Les longueurs sont exprimées par des **entiers** ou des **nombres flottants**.
- Les angles sont exprimés en **degrés**.
    - **Attention !** N'exprimez jamais les angles en radians. La bibliothèque effectue automatiquement les changements.
- La bibliothèque réalise toujours la vérification de type.
- Pour afficher une forme, vous devez toujours appeler sa fonction `draw()`.
    - Pour le premier affichage.
    - Pour les mises à jour (déplacement, rotation, etc.)
- Dans la version actuelle, les mises à jour ne sont pas supportées.

La méthode `resize`

- Cette méthode, commune à toutes les classes, permet de modifier la grosseur d'un objet sans affecter la position de son centre.
- Elle prend un **facteur multiplicateur**. C'est-à-dire une valeur par laquelle la taille sera augmentée ou diminuée.
    - Ce facteur est toujours un entier ou un nombre flottant **positif**.
    - Un facteur plus grand que **1** augmente la grosseur.
    - Un facteur plus petit que **1** diminue la grosseur.
    - Exemple :
        - Avec un facteur de **2**, la forme sera deux fois **plus grosse**.
        - Avec un facteur de **0.5**, la forme sera deux fois **plus petite**.

```
forme.resize(multiplicateur)
```

La méthode `translate`

- Cette méthode permet de déplacer l'origine de la forme.
- Cette méthode prend un type `Point` en paramètre.
- La valeur du paramètre détermine le déplacement à faire.
    - C'est-à-dire que la valeur du paramètre sera ajoutée à l'origine.

```
deplacement = Point(x, y)
forme.translate(deplacement)
```

La classe `Ligne`

- Cette classe permet de dessiner une ligne.
- Le constructeur prend quatre paramètres (en plus de canvas) :
    - `origine` : Coordonnée de départ de la ligne.
    - `distance` : Longueur, sur l'axe de la ligne, de cette dernière.
    - `angle` : rotation de la ligne.
    - `fill` : couleur de la ligne (par défaut **noir**).

```
ligne = Ligne(canvas, origine, distance, angle = 0, fill = 'black')
```

La classe `Rectangle`

- Cette classe permet de dessiner un rectangle.
- Le constructeur prend six paramètres (en plus de canvas) :
  - `origine` : Coordonnée du centre du rectangle.
  - `largeur` : Longueur sur l'axe `x` de votre rectangle.
  - `hauteur` : Longueur sur l'axe `y` de votre rectangle.
  - `fill` : couleur du fond de la forme (par défaut **transparent**).
  - `outline` : couleur de la bordure (par défaut **noir**).
  - `width` : Épaisseur [1] de la bordure.

```
rectangle = Rectangle(canvas, origine, largeur, hauteur, fill = '',
outline='black', width=1)
```

La classe `Carre`

- Cette classe permet de dessiner un carré.
- Le constructeur prend cinq paramètres (en plus de canvas) :
  - `origine` : Coordonnée du centre du carré.
  - `largeur` : Longueur d'un côté de votre carré.
  - `fill` : couleur du fond de la forme (par défaut **transparent**).
  - `outline` : couleur de la bordure (par défaut **noir**).
  - `width` : Épaisseur [1] de la bordure.

```
carre = Carre(canvas, origine, largeur, fill = '', outline='black', width=1)
```

La classe `Cercle`

- Cette classe permet de dessiner un cercle.
- Le constructeur prend cinq paramètres (en plus de canvas) :
  - `origine` : Coordonnée du centre du cercle.
  - `rayon` : Longueur du rayon de votre cercle.
  - `fill` : couleur du fond de la forme (par défaut **transparent**).
  - `outline` : couleur de la bordure (par défaut **noir**).
  - `width` : Épaisseur [1] de la bordure (par défaut **1px**).

```
cercle = Cercle(anvas, origine, rayon, fill = '', outline='black', width=1)
```

La classe `Croix`

- Cette classe permet de dessiner une croix (`X`).
- Le constructeur prend cinq paramètres (en plus de canvas) :
  - `origine` : Coordonnée du centre de la croix.
  - `largeur` : Longueur de la croix autant à l'horizontale qu'à la verticale.
  - `fill` : couleur du fond de la forme (par défaut **transparent**).
  - `width` : Épaisseur [1] de la bordure (par défaut **1px**).

```
croix = Croix(canvas, origine, largeur, fill = 'black', width = 1)
```

La classe `AfterEvent`

- La classe permet d'exécuter une fonction de votre choix à toutes les **n** millisecondes.
- Pour utiliser la classe, il suffit de lui donner une fonction, sans paramètre, à son constructeur et le délai entre deux exécutions en milliseconde.
    - Pour rappel, **1000 millisecondes correspondent à** 1 seconde**.
- Pour lancer la boucle d'exécution, il suffit d'appeler la fonction `start()`.
    - Votre méthode sera exécutée après le délai mentionné dans le constructeur.
    - Pour exécuter immédiatement votre fonction, utilisez la fonction `startImmediately()`.
        - Par la suite, votre boucle sera exécutée après le délai mentionné dans le constructeur.

```
def evenement() :
    print('Ceci est un exemple.')

e = AfterEvent(root, evenement, 500)
# La fonction evenement sera exécutée toutes les demi-secondes.
e.start()
# La fonction evenement sera exécuté dans exactement un demi-seconde.
```

---

1. L'épaisseur est toujours exprimée par un nombre **entier**. ↩ ↩ ↩ ↩