GAJAN Antoine (TC03)
SAINT-MARC Arman (TC03)

UV : INF2

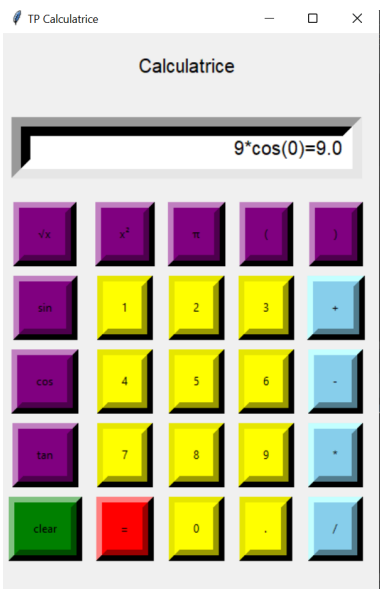
24/11/2021

INF2 - Rapport du TP5

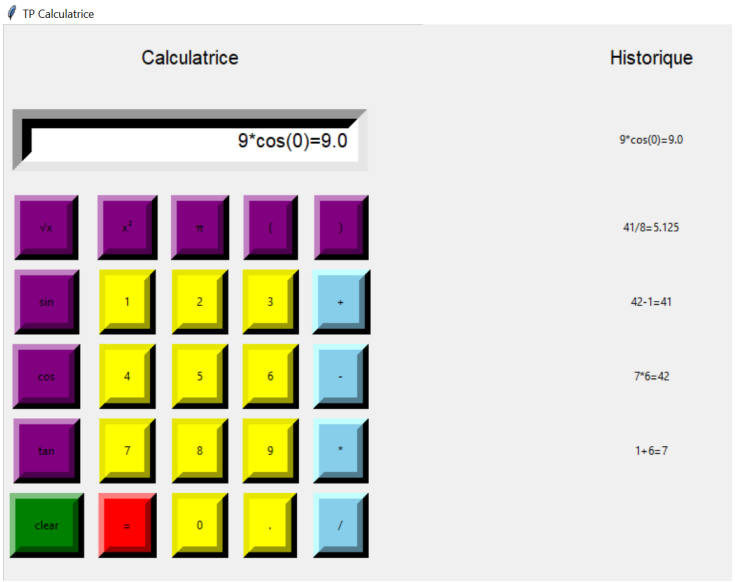
Consignes :

Travail général :

→ Expliquer le problème, le raisonnement et la solution
→ Libre sur la manière de faire (mais expliquer)



Fenêtre réduite

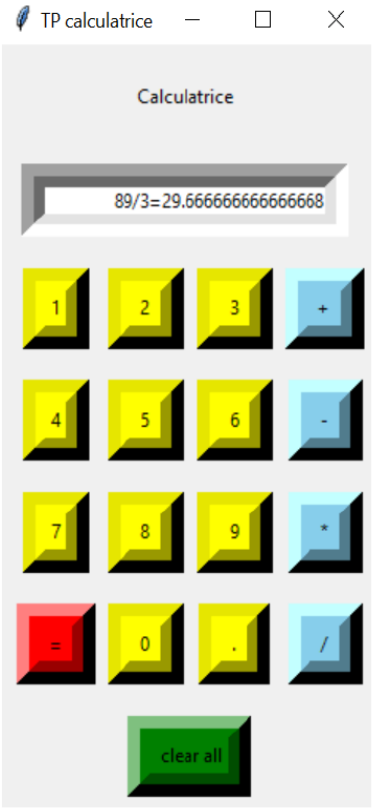


Fenêtre en plein écran

Problème général

L'objectif de ce TP est de créer une calculatrice graphique en python à l'aide de la librairie Tkinter. Cette calculatrice devra permettre à l'utilisateur de réaliser des calculs basiques (addition, soustraction, multiplication, division) mais également de faire des opérations plus complexes ayant recours aux fonctions sinus, cosinus, tangente, carré et racine carré. Une touche pour la constante pi est également demandée. Il faudra enfin ajouter un historique qui permettra à l'utilisateur de se souvenir des derniers calculs qu'il a effectué.

Un exemple de calculatrice proposé est le suivant :



Ainsi, nous avons choisi de diviser ce problème en plusieurs étapes comme suit :

- la création de l'aspect graphique de la calculette
- la mise en place de fonctions pour le calcul
- la mise en place d'un historique **(disponible uniquement lorsque la fenêtre est en plein écran)**

Aspect graphique de la calculatrice

★ Esquisse de la calculatrice

Tout d'abord, nous avons réfléchi à l'aspect graphique de notre calculatrice. Nous avons d'abord pensé au modèle suivant :

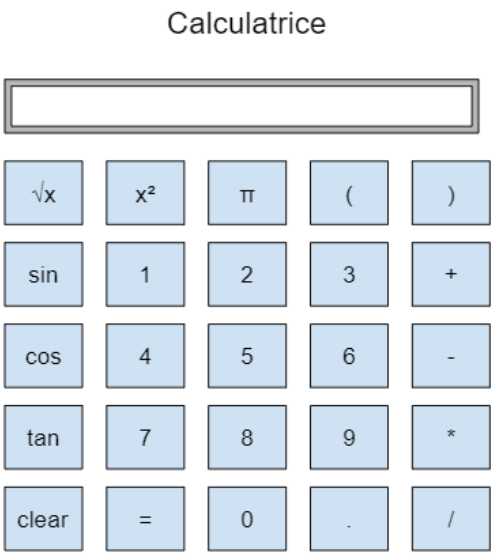


Figure 1 : Esquisse d'aspect graphique de la calculatrice à réaliser

★ Création de la première fenêtre

La réalisation de cette interface graphique se fera à l'aide de la librairie Tkinter que l'on importe comme suit :

```
#Import du module Tkinter
from tkinter import *
```

Ensuite, afin de simplifier le travail, il nous a semblé intéressant de créer une classe Calculatrice, qui héritera de la classe Tk du module Tkinter afin de pouvoir bénéficier d'un maximum de fonctionnalités.

On crée la classe comme suit :

```
#Création de la classe
class Calculatrice(Tk):

   #Initialisatin de la fenêtre
   def __init__(self, largeur = 400, hauteur = 600):
      #Instanciation de la classe mère
      super().__init__()
```

```
    #Autres méthodes à créer...
```

Comme toute application, nous avons voulu lui donner un titre afin que l'utilisateur comprenne mieux l'intérêt de cette interface. Pour cela, nous avons utilisé la méthode title() comme suit :

```
#Ajout d'un titre à la fenêtre dans le __init__()
self.title('TP Calculatrice')
```

L'aspect graphique doit permettre à l'utilisateur de trouver simple et efficace l'application. Nous avons donc voulu jouer sur la taille de l'écran et son positionnement au lancement de l'application Pour cela, nous avons utilisé plusieurs méthodes qui nous permettront non seulement de récupérer les informations sur l'écran que possède l'utilisateur, mais également de positionner la fenêtre sur l'écran comme suit :

```
#Definition de la taille de la fenêtre
self.largeur_fenetre = largeur
self.hauteur_fenetre = hauteur

#Récupération des informations de l'écran de l'utilisateur
self.largeur_ecran = self.winfo_screenwidth()
self.hauteur_ecran = self.winfo_screenheight()

#Calcul de la position à laquelle placer la fenêtre sur l'écran
de l'utilisateur
self.posx = self.largeur_ecran // 2 - self.largeur_fenetre // 2
self.posy = self.hauteur_ecran // 2 - self.hauteur_fenetre // 2 -
50

#Positionnement de la fenêtre sur l'écran de l'utilisateur
self.geometry(f"{self.largeur_fenetre}x{self.hauteur_fenetre}+{se
lf.posx}+{self.posy}")
```

Au regard de la variable self.posx, on remarque que la fenêtre n'est pas parfaitement centrée sur l'écran. Cela est fait exprès, car en général, l'utilisateur n'est pas plein écran et a par conséquent la barre du système d'exploitation en bas de son écran, d'où ce choix.

★ Ajout des labels et des boutons

Grâce au travail préparatoire réalisé en amont, nous avons compris que nous allions avoir besoin de créer 25 boutons (sans compter ceux de l'historique) et plusieurs labels pour représenter le titre "Calculatrice", le titre "Historique" mais aussi l'écran de la calculatrice.

La tâche étant répétitive, la meilleure solution est de créer des méthodes nous permettant de créer des boutons et des labels. Nous appellerons respectivement createLabel() et createButton() ces méthodes.

Après lecture de la documentation Tkinter, nous avons mis en place ces méthodes comme suit :

➔ **Pour créer un label**

```
"""Création d'un label ayant un nom d'identification, un texte à
afficher, une largeur de bordure, un nombre de colonnes sur lequel
il s'étend, un alignement dans le label, une couleur
d'arrière-plan, un certain relief, se positionnant sur la grille à
l'emplacement (ligne, colonne)"""

def createLabel(self, nom, texte, ligne, colonne, largeur,
border_largeur = 2, colonnespan = 1, alignement = CENTER, bgcolor
= '#F0F0F0', relief = FLAT):

   #Création du label avec les différents paramètres
   self.labels[nom] = Label(self, text = texte, bg = bgcolor,
relief = relief, width = largeur, borderwidth = border_largeur,
anchor = alignement, font = tkinter.font.Font(family =
"TkDefaultFont", size = 15))

   #Placement du label sur la grille de la fenêtre
   self.labels[nom].grid(row = ligne, column = colonne,
columnspan = colonnespan, padx = 10, pady = 20)
```

➔ **Pour créer un bouton**

```
   """Création d'un bouton ayant un nom d'identification, un texte
à afficher, une couleur d'arrière-plan, se positionnant sur la
grille à l'emplacement (ligne, colonne) et ayant une commande à
activer"""

def createButton(self, nom_bouton, ligne, colonne, commande,
bgcolor = '#F0F0F0', colonnespan = 1, relief = RAISED):

   #Création du bouton
   self.boutons[nom_bouton] = Button(self, text = nom_bouton,
padx = 12, pady = 12, bg = bgcolor, borderwidth = 12, relief =
relief, command = commande)

   #Positionnement du bouton sur la grille
   self.boutons[nom_bouton].grid(row = ligne, column = colonne,
pady = 5, columnspan = colonnespan)
```

NB : anchor → postion du texte si le widget dispose de plus de place
borderwidth → largeur de la bordure du widget
bg → couleur de l'arrière-plan
columnspan → nombre de colonnes où ça s'étend
font → police et taille du texte
relief → type de bordure
padx, pady → espace horizontal/vertical à insérer dans l'étiquette

Il ne reste plus qu'à appeler ces deux fonctions dans le __init__() afin de créer les boutons nécessaires sur la fenêtre.

Algorithme de calcul

L'aspect graphique étant réalisé, il faut à présent que notre calculatrice soit opérationnelle. Pour cela, il faut définir des fonctions de calcul, qui peuvent récupérer les calculs souhaités par l'utilisateur et les exécuter.

Ces fonctions seront exécutées lorsque les boutons sont pressés. On distinguera la fonction appelée en fonction du type de bouton pressé (si c'est un égal, on fait le calcul, sinon on le modifie juste). Dans le cas où la touche "clear" a été choisie, il faudra remettre à vide l'affichage de la calculatrice.

Pour cela, nous avons créé 3 méthodes : modifier_calcul(), valider_calcul() et supprimer()

★ Modifier le calcul

Notre raisonnement algorithmique pour la méthode modifier_calcul() est le suivant :

- On modifie la chaine de l'expression à afficher sur l'écran de la calculatrice en rajoutant la valeur mathématique du bouton sélectionné par l'utilisateur
- On modifie la chaine de l'expression mathématique en rajoutant la valeur mathématique convertie en chaine de caractère du bouton sélectionné par l'utilisateur

Ainsi, après avoir initialisé 2 variables self.affichage et self.expression à vide dans le __init__() qui contiennent respectivement l'expression à afficher sur l'écran et l'expression mathématique contenant le calcul "informatique". On a donc les bouts de code suivants :

```
#A rajouter dans le __init__():
#Initialisation de la variable contenant l'expression
mathématique du calcul souhaité par l'utilisateur
self.expression = ''
#Initialisation de la variable contenant l'expression
mathématique du calcul souhaité par l'utilisateur
self.affichage = ''



def modifier_calcul(self, valeur_affichee, valeur_mathematique):
   "Mise à jour de l'affichage et de l'expression mathématique"
   #Modification de la chaine à afficher sur l'écran de la
calculatrice
   self.affichage += valeur_affichee
   #Modification de l'expression mathématique en rajoutant la
valeur mathématique choisie par l'utilisateur
   self.expression += valeur_mathematique
   #Mise à jour de l'affichage de la calculatrice
   self.labels['resultat'].config(text = self.affichage)
```

Lors de l'appel de cette fonction callback dans les différents boutons, puisqu'il faut lui donner 2 paramètres que sont la valeur affichée et la valeur mathématique, nous utiliserons la fonction partial du module functools qui permet de donner des paramètres à la fonction callback(). Par exemple, on aura la ligne de code suivante pour le bouton relatif à la touche "1" :

```
self.createButton('1', 4, 1, commande =
partial(self.modifier_calcul, '1', '1'), bgcolor = 'yellow')
```

★ Valider le calcul

Lorsque l'utilisateur appuie sur le bouton "=", il faut déclencher le calcul.

Notre raisonnement algorithmique pour la méthode valider_calcul() est le suivant :

Deux cas sont à distinguer :

- **Si le calcul souhaité est possible (pas d'erreur de syntaxe, de domaine de définition,...)**
- On évalue le résultat de l'expression mathématique grâce à la fonction eval() de Python.
- On affiche dans l'écran de la calculatrice le résultat du calcul.
- Préparation des variables d'affichage et d'expression mathématique avec la valeur du résultat en vu d'un prochain calcul

Traduit en Python, cela nous donne le résultat suivant :

```
def valider_calcul(self):

   "Calcul de l'expression mathématique demandée par
l'utilisateur"

   #Evaluation de la valeur de l'expression mathématique pour
obtenir le résultat si possible
   try:
       #Calcul du résultat
       self.resultat = str(eval(self.expression))

       #Mise à jour de l'affichage de la calculatrice
       self.labels['resultat'].config(text=f"{self.affichage} =
{self.resultat}")
```

- **Si le calcul souhaité est impossible (erreur rencontrée dans le calcul demandé)**
- Si l'erreur rencontrée est une division par 0, alors on affiche "ERREUR DIVISION PAR 0" dans l'écran de la calculatrice

- Si l'erreur rencontrée est liée à une valeur incorrecte (comme la demande d'une racine carrée d'un nombre négatif, alors, on affiche "ERREUR DE VALEUR" dans l'écran de la calculatrice
- Si l'erreur rencontrée est liée à un problème de syntaxe (ex : "cos34" sans parenthèse ou "3(4)+2))") on affiche "ERREUR DE SYNTAXE" dans l'écran de la calculatrice
- Enfin, on remet à vide l'affichage et l'expression mathématique en vue du prochain calcul

On peut alors rajouter cette gestion des erreurs dans notre fonction valider_calcul().

```
#En cas de division par 0
except ZeroDivisionError:
   #Affichage de l'erreur sur l'écran de la calculatrice
   self.labels['resultat'].config(text = "ERREUR DIVISION PAR 0")
   #Remise à 0 du texte l'expression mathématique et de
l'affichage en vue du prochain calcul
   self.expression = ''
   self.affichage = ''

#En cas d'erreur de valeur (domaine de fonction,...)
except ValueError:
   # Affichage de l'erreur sur l'écran de la calculatrice
   self.labels['resultat'].config(text="ERREUR DE VALEUR")
   # Remise à 0 du texte l'expression mathématique et de
l'affichage en vue du prochain calcul
   self.expression = ''
   self.affichage = ''

#En cas d'erreur de syntaxe
except SyntaxError:
   # Affichage de l'erreur sur l'écran de la calculatrice
   self.labels['resultat'].config(text = "ERREUR DE SYNTAXE")
   # Remise à 0 du texte l'expression mathématique et de
l'affichage en vue du prochain calcul
   self.expression = ''
   self.affichage = ''
```

Evidemment, ce n'est pour le moment qu'une fonction dans une version provisoire, car il restera à gérer l'historique comme expliqué ci-après.

★ Effacer le calcul

Lorsque l'utilisateur appuie sur "clear", il faut effacer le calcul en cours.

Pour cela, nous avons eu l'idée algorithmique suivante :

- Effacer le texte à l'écran de la calculatrice
- Remise à vide de l'expression mathématique en vue d'un prochain calcul

Traduit en Python, cela donne le résultat suivant :

```
def effacer(self):
   "Efface le calcul en cours"
   #Effacage du texte à l'écran de la calculatrice et remise à
vide de l'expression mathématique
   self.affichage = ''
   self.expression = ''
   #Suppression de l'affichage du résultat
   self.labels['resultat'].config(text=self.affichage)
```

La gestion de l'historique

L'ajout d'un historique n'a pas été très complexe après la réalisation des étapes précédentes. Nous avons décidé de procéder en deux étapes :

- sauvegarder les calculs et résultats dans un tableau
- afficher les 5 derniers calculs dans l'interface graphique

Pour la première étape, nous avons initialisé un tableau au sein du __init__() de la classe "Fenetre".

```
#Initialisation d'un tableau avec l'ensemble des calculs réalisés
self.list_historique = []
```

Puis dans notre fonction valider_calcul(), utilisée pour calculer les résultats des expressions rentrées, nous avons ajouté l'expression à calculer ainsi que le résultat au tableau représentant l'historique.

Afin d'améliorer l'expérience utilisateur par la suite lors de l'affichage de l'historique dans la fenêtre, nous avons décidé d'ajouter le calcul au début de l'historique, à l'index 0, à l'aide de la méthode insert comme suit (dans la méthode valider_calcul()) :

```
#Ajout du calcul au début de l'historique
self.list_historique.insert(0, self.labels['resultat']['text'])
```

Enfin, à chaque calcul effectué, nous souhaitons mettre à jour l'affichage de l'historique. Nous avons donc décidé d'appeler une fonction permettant de faire cela dans la méthode valider_calcul().

```
#Modification de l'historique
self.historique()
```

La seconde étape consiste en la création d'une fonction qui met à jour l'affichage de l'historique des calculs. Cette fonction présentera les 5 derniers calculs.

On définit une variable contenant le nombre de calculs à afficher pour l'historique :

```
#Variable contenant le nombre de calculs à afficher dans la
partie historique
self.nb_calcul = 5
```

Enfin, pour chaque élément dans la liste de l'historique, nous devons l'afficher dans l'interface graphique. C'est pourquoi nous avons choisi de faire une boucle for sur les 5

premiers éléments de l'historique (car on a ajouté les calculs au début du tableau à chaque fois) avec un appel de la fonction enumerate() pour récupérer un index qui nous servira par la suite.

Un bouton sera alors créé pour chacun des 5 derniers calculs de l'historique. Afin de pouvoir appeler la commande modifier_calcul() pour que l'utilisateur puisse recycler le résultat d'un précédent calcul, il nous faut extraire le résultat. Afin de l'extraire, la méthode split() semble la plus appropriée. Enfin, l'index créé précédemment nous permet de placer le bouton sur la grille.

D'où le code suivant pour la méthode historique() :

```
def historique(self):
   "Affiche l'historique des 5 derniers calculs"

   #Variable contenant le nombre de calculs à afficher dans la
partie historique
   self.nb_calcul = 5

   #Parcours des derniers éléments de l'historique pour les
afficher (uniquement en mode plein écran)
   for i, calcul in
enumerate(self.list_historique[:self.nb_calcul]):

       #Extraction du résultat du calcul
       self.valeur = str(calcul.split('=')[1])

       #Création d'un bouton contenant le calcul
       self.createButton(f"{calcul}", 2 + i, 5, commande =
partial(self.modifier_calcul, self.valeur, self.valeur),
colonnespan = 5, relief = FLAT)
```