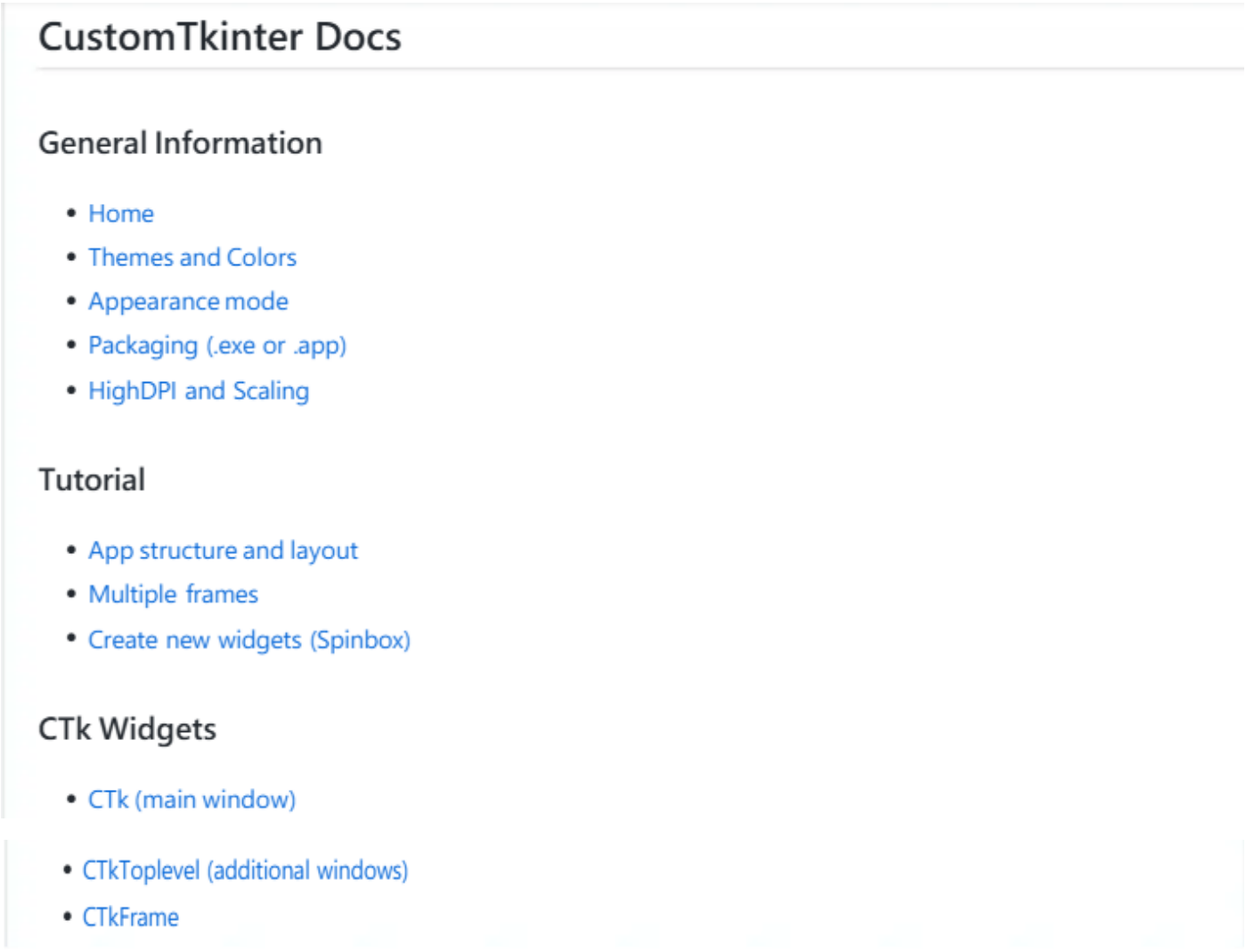
DOCS CUSTOMTKINTER

HOME

Il s'agit de la documentation officielle de CustomTkinter, où vous pouvez trouver des informations détaillées sur tous les widgets disponibles et quelques informations supplémentaires sur le fonctionnement des thèmes et du mode d'apparence. N'hésitez pas à poser des questions dans l'onglet Discussion lorsque quelque chose n'est pas clair, si vous avez besoin d'aide pour un projet ou si vous voulez montrer votre projet réalisé avec CustomTkinter. Si vous avez trouvé un bogue, veuillez le signaler dans l'onglet Problèmes et fournir le message d'erreur complet et un exemple de code minimal qui reproduit l'erreur. Si vous avez des identifiants sur des améliorations ou des demandes de nouveaux widgets, vous pouvez également les signaler dans l'onglet Problèmes, mais veuillez-vous informer sur les demandes précédentes pour éviter les doublons.



- CTkScrollableFrame
- CTkTabview
- CTkTextbox
- CTkScrollbar
- CTkButton
- CTkLabel
- CTkEntry
- CTkOptionMenu
- CTkComboBox
- CTkSegmentedButton
- CTkSwitch
- CTkCheckBox
- CTkRadioButton
- CTkSlider
- CTkProgressBar
- CTkInputDialog

Utility

- CTkFont
- CTkImage

COLORS

Toutes les couleurs des widgets CTk peuvent être personnalisées, les arguments appropriés peuvent être trouvés dans la documentation des widgets spécifiques. Notez que bg_color n'est la couleur derrière le widget que s'il a des coins arrondis. La couleur principale d'un widget est typique fg_color dans CustomTkinter :

Les couleurs peuvent être définies sur un seul nom de couleur (" red "), une seule chaîne de couleur hexadécimale(« #FF0000 ») ou une couleur de tuple pour une couleur de mode clair et une couleur de mode sombre (" red ","dark red "). La couleur actuelle sera alors automatiquement sélectionnée par le widget en fonction du mode d'apparence actuel. Ainsi, en utilisant la couleur du tuple, les widgets peuvent avoir des couleurs différentes en mode d'apparence clair et sombre. Si vous utilisez une seule couleur, cette couleur sera utilisée à la fois pour le mode d'apparence clair et sombre.

Example:

```
button = customtkinter.CTkButton(root_tk, fg_color="red")  # single color name

button = customtkinter.CTkButton(root_tk, fg_color="#FF0000")  # single hex string

button = customtkinter.CTkButton(root_tk, fg_color=("#DB3E39", "#821D1A"))  # tuple col
```

THEMES

Par défaut, toutes les couleurs sont définies par le thème de couleur. Actuellement, il y a trois thèmes disponibles : " bleu ", " bleu foncé " et " Vert ", où le « bleu » est le thème standard. Tous les thèmes prévoient des couleurs tuples pour un mode d'apparence clair et sombre. Vous pouvez définir le thème au début de votre programmation comme suit :

```
import customtkinter

customtkinter.set_default_color_theme("dark-blue")  # Themes: "blue" (standard), "green
```

CUSTOM THEMES

Un thème est décrit par un fichier .json comme cela : dark-blue.json. Vous pouvez également créer votre propre thème, afin de ne pas avoir à définir manuellement le style de chaque widget. Il suffit de copier le fichier .json ci-dessus et de modifier les valeurs. Ensuite, vous pouvez charger le nouveau thème en passant le chemin d'accès à votre fichier .json à la méthode .set_default_color_theme :

```
import customtkinter

customtkinter.set_default_color_theme("path/to/your/custom_theme.json")
```

APPEARANCE MODE

Le mode d'apparence décide quelle couleur sera choisie parmi les couleurs du tuple, si la couleur est spécifiée comme couleur du tuple. Vous pouvez définir le mode d'apparence à tout moment à l'aide de la commande suivante :

```
import customtkinter

customtkinter.set_appearance_mode("system")  # default value
customtkinter.set_appearance_mode("dark")
customtkinter.set_appearance_mode("light")
```

Si vous définissez le mode d'apparence sur « système », le mode actuel est lu à partir du système d'exploitation. Il s'adaptera également si le mode d'apparence du système d'exploitation change pendant la durée du programme. Notez que sous Linux, il sera toujours en « light » mode si défini sur « système », parce que le mode ne peut pas être lu à partir du système d'exploitation pour le moment, cela sera probablement mis en œuvre à l'avenir.

PACKAGING

Lorsque vous créez un .exe sur Windows avec pyinstaller, il y a deux choses que vous devez prendre en compte. Tout d'abord, vous ne pouvez pas utiliser le –onefile Option de pyInstaller, car la bibliothèque CustomTkinter comprend non seulement des fichiers .py, mais aussi des fichiers de données tels que .json et .otf. PyInstallern'est pas en mesure de les regrouper dans un seul fichier .exe, donc vous devez utiliser le –onedir option. Et deuxièmement, vous devez inclure le répertoire customtkinter manuellement avec le --add-data option de pyinstaller. Parce que pour une raison quelconque, pyinstaller n'inclut pas automatiquement les fichiers de données comme .json de la bibliothèque. Vous pouvez trouver l'emplacement d'installation de la bibliothèque customtkinter avec la commande suivante :

```
pip show customtkinter
```

A Location will be shown, for example: `c:\users\<user_name>\appdata\local\programs\python\python310\lib\site-packages`

Then add the library folder like this: `--add-data "C:/Users/<user_name>/AppData/Local/Programs/Python/Python310/Lib/site-packages/customtkinter;customtkinter/"`

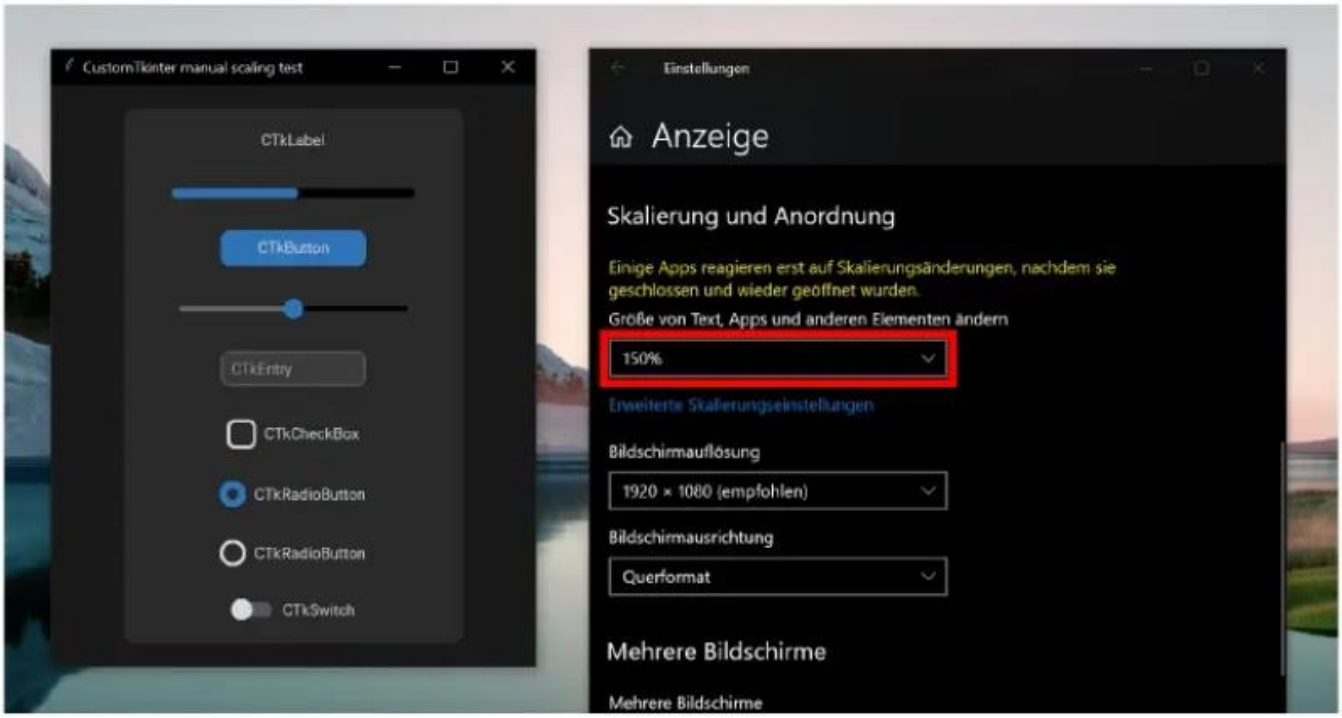
With Auto Py to Exe you would do it like this:

For the full command you get something like this:

```
pyinstaller --noconfirm --onedir --windowed --add-data "<CustomTkinter
Location>/customtkinter;customtkinter/"  "<Path to Python Script>"
```

SCALING

Par défaut, CustomTkinter prend en charge la mise à l'échelle HighDPI sur macOS et Windows. Sur macOS, la mise à l'échelle fonctionne automatiquement pour les fenêtres Tk. Sous Windows, l'application prend en charge la résolution DPI (windll.shcore.SetProcessDpiAwareness(2)) et le facteur d'échelle actuel de l'écran est détecté. Ensuite, chaque élément et les dimensions de la fenêtre sont mis à l'échelle par ce facteur par CustomTkinter.



Vous pouvez désactiver cette mise à l'échelle automatique comme suit :

```
customtkinter.deactivate_automatic_dpi_awareness()
```

Ensuite, la fenêtre sera floue sur Windows avec une valeur d'échelle de plus de 100 %.

CUSTOM SCALING

In addition to the automatically detected scaling factor, you can also set your own scalingfactors for the application like the following:

```
customtkinter.set_widget_scaling(float_value)  # widget dimensions and text size
customtkinter.set_window_scaling(float_value)  # window geometry dimensions
```

APP STRUCTURE AND LAYOUT

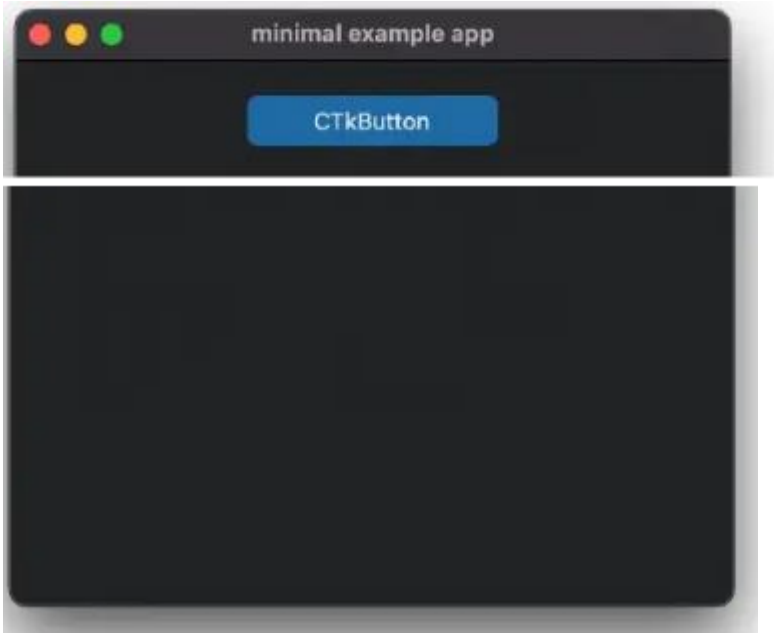
Recommandations

À moins que votre programme ne soit qu'un test, vous devez toujours structurer votre application en classes qui héritent de customtkinter. CTk pour la fenêtre principale et le personnaliser. CTkToplevel ou customtkinter. CTkFrame pour les fenêtres ou les cadres de niveau supérieur. Ce faisant, vous obtiendrez un code clair et lisible et une application extensible. Écrire beaucoup de code d'interface utilisateur sans classes et dans un seul fichier .py avec des variables globales comme c'est souvent le cas, est pénible à lire et un très mauvais style de codage ! La deuxième recommandation est de ne jamais utiliser le gestionnaire de géométrie .place(), mais d'utiliser .pack() pour les très petites applications ou .grid() dans tous les autres cas. Le problème avec .place() est que cela semble simple au début, lorsque vous placez simplement quelques widgets sur une fenêtre. Mais vous vous empêchez d'étendre votre application. Si vous souhaitez insérer un widget en haut, vous devez corriger les positions de tous les autres widgets ci-dessous. Avec grid ou pack, tous les widgets se déplacent et s'alignent

automatiquement avec un remplissage et une taille min/max donnés. En utilisant place, vous n'obtiendrez jamais un rembourrage cohérent et une fenêtre réactive.

Minimal onefile example app with .pack()

```
import customtkinter


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()

        self.title("minimal example app")
        self.minsize(400, 300)

        self.button = customtkinter.CTkButton(master=self, command=self.button_callback
        self.button.pack(padx=20, pady=20)

    def button_callback(self):
        print("button pressed")


if __name__ == "__main__":
    app = App()
    app.mainloop()
```



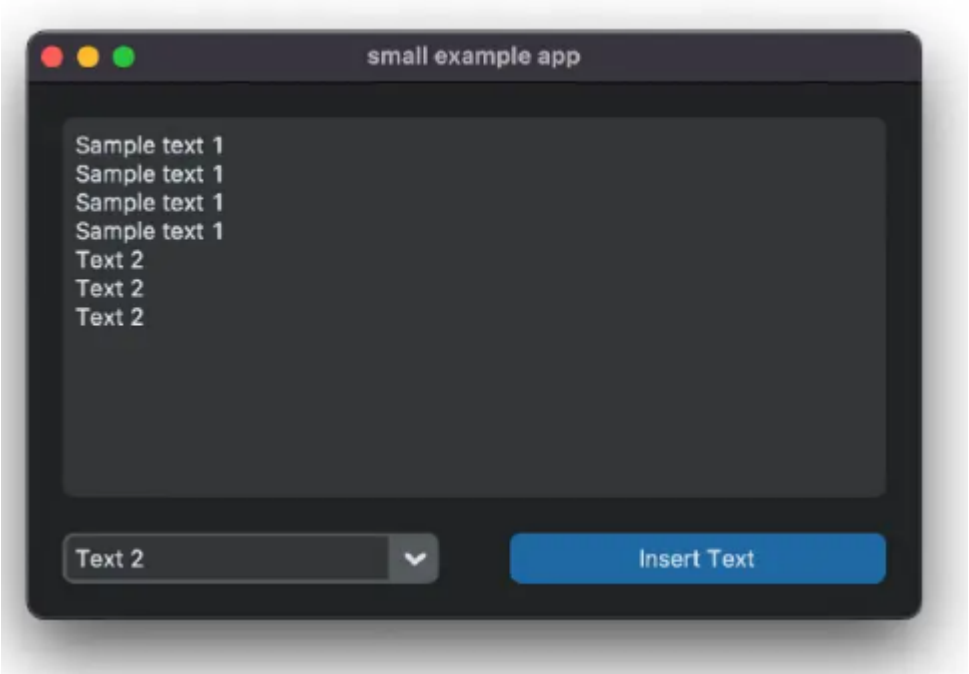
Small onefile example app with .grid()

Maintenant, nous utilisons le gestionnaire de géométrie de grille pour créer un système de grille 2x2. La première rangée prend un poids de 1, elle va donc s'étendre. La deuxième ligne a un poids par défaut de 0 et ne sera pas plus grande que nécessaire pour s'adapter aux widgets qu'elle contient. Les deux colonnes ont un poids égal de 1 afin qu'elles se développent de manière égale et remplissent la fenêtre. Le manuel a une portée de colonne de 2, de sorte qu'il remplira complètement la première ligne. En définissant sticky sur nsew , nous forçons le manuel à coller à tous les bords des cellules de la grille dans lesquelles il se trouve. Il s'étendra et remplira complètement la première rangée. Réglez donc le remplissage x à 20 des deux côtés et le remplissage y à (20, 0) afin qu'il n'y ait pas de remplissage en bas. (Cela viendra des boutons et de l'entrée dans la deuxième rangée)L'entrée et le bouton

obtiennent un rembourrage de 20 de tous les côtés et un collant de ew , de sorte qu'ils s'étendront horizontalement et rempliront toute la cellule de grille dans laquelle ils se trouvent.

```
import customtkinter


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()

        self.geometry("500x300")
        self.title("small example app")
        self.minsize(300, 200)

        # create 2x2 grid system
        self.grid_rowconfigure(0, weight=1)
        self.grid_columnconfigure((0, 1), weight=1)

        self.textbox = customtkinter.CTkTextbox(master=self)
```

```
        self.textbox.grid(row=0, column=0, columnspan=2, padx=20, pady=(20, 0), sticky=

        self.combobox = customtkinter.CTkComboBox(master=self, values=["Sample text 1",
        self.combobox.grid(row=1, column=0, padx=20, pady=20, sticky="ew")
        self.button = customtkinter.CTkButton(master=self, command=self.button_callback
        self.button.grid(row=1, column=1, padx=20, pady=20, sticky="ew")

    def button_callback(self):
        self.textbox.insert("insert", self.combobox.get() + "\n")


if __name__ == "__main__":
    app = App()
    app.mainloop()
```



APP STRUCTURE AND LAYOUT

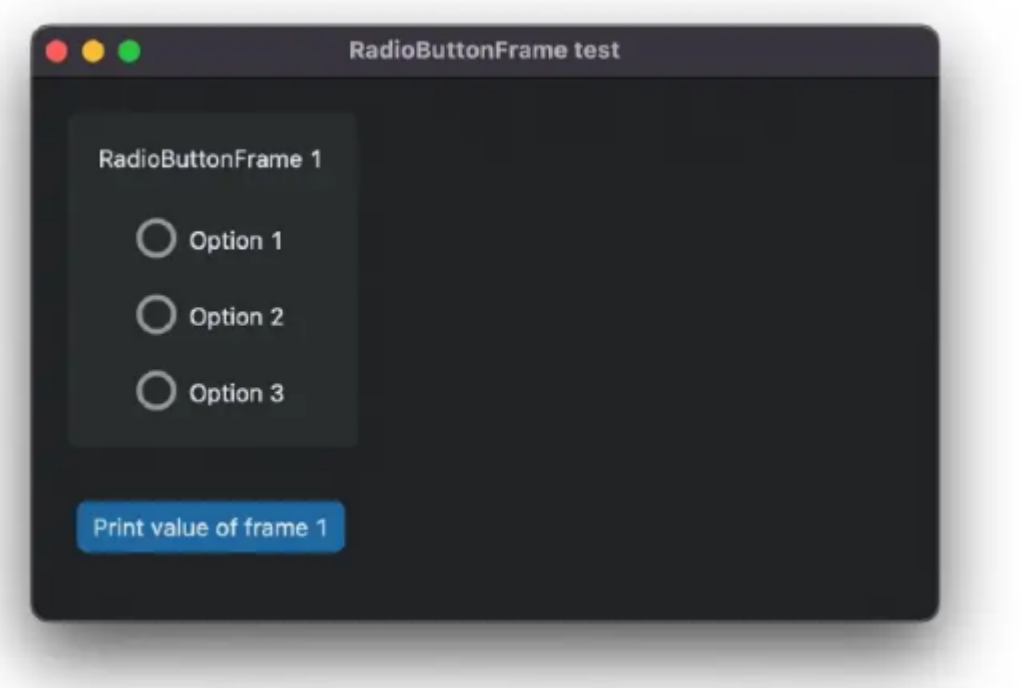
Les widgets qui vont ensemble ou qui créent ensemble une fonctionnalité spécifique doivent être placés dans un cadre, afin qu'ils puissent être séparés dans le code et qu'ils forment visiblement une unité dans l'interface utilisateur. Un cadre doit être créé en tant que classe distincte, qui peut également être placée dans un fichier ou un module séparé. Vous pouvez ensuite simplement créer une instance du cadre dans votre classe d'application principale et la placer sur la fenêtre. Un autre grand avantage est que vous pouvez facilement déplacer le cadre dans la fenêtre de l'application, car vous n'avez pas besoin de modifier les positions de tous les widgets à l'intérieur du cadre. Dans ce qui suit, j'ai créé un cadre qui implémente un groupe de boutons radio avec une étiquette en guise d'avance. La classe hérite de CTkFrame et sera étendue par des méthodes appelées get_value et set_value pour obtenir et définir la chaîne de boutons radio actuellement sélectionnée :

```
class RadioButtonFrame(customtkinter.CTkFrame):
    def __init__(self, *args, header_name="RadioButtonFrame", **kwargs):
        super().__init__(*args, **kwargs)

        self.header_name = header_name

        self.header = customtkinter.CTkLabel(self, text=self.header_name)
        self.header.grid(row=0, column=0, padx=10, pady=10)

        self.radio_button_var = customtkinter.StringVar(value="")

        self.radio_button_1 = customtkinter.CTkRadioButton(self, text="Option 1", value
        self.radio_button_1.grid(row=1, column=0, padx=10, pady=10)
        self.radio_button_2 = customtkinter.CTkRadioButton(self, text="Option 2", value
        self.radio_button_2.grid(row=2, column=0, padx=10, pady=10)
        self.radio_button_3 = customtkinter.CTkRadioButton(self, text="Option 3", value
        self.radio_button_3.grid(row=3, column=0, padx=10, pady=(10, 20))

    def get_value(self):
        """ returns selected value as a string, returns an empty string if nothing sele
        return self.radio_button_var.get()

    def set_value(self, selection):
        """ selects the corresponding radio button, selects nothing if no corresponding
        self.radio_button_var.set(selection)
```

Ce cadre peut maintenant être placé sur la fenêtre de l'application en créant une instance de celui-ci, comme vous le feriez pour un standard Cadre CTkK :

```
import customtkinter


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()

        self.geometry("500x300")
        self.title("RadioButtonFrame test")

        self.radio_button_frame_1 = RadioButtonFrame(self, header_name="RadioButtonFram
        self.radio_button_frame_1.grid(row=0, column=0, padx=20, pady=20)

        self.frame_1_button = customtkinter.CTkButton(self, text="Print value of frame
        self.frame_1_button.grid(row=1, column=0, padx=20, pady=10)

    def print_value_frame_1(self):
        print(f"Frame 1 value: {self.radio_button_frame_1.get_value()}")


if __name__ == "__main__":
    app = App()
    app.mainloop()
```

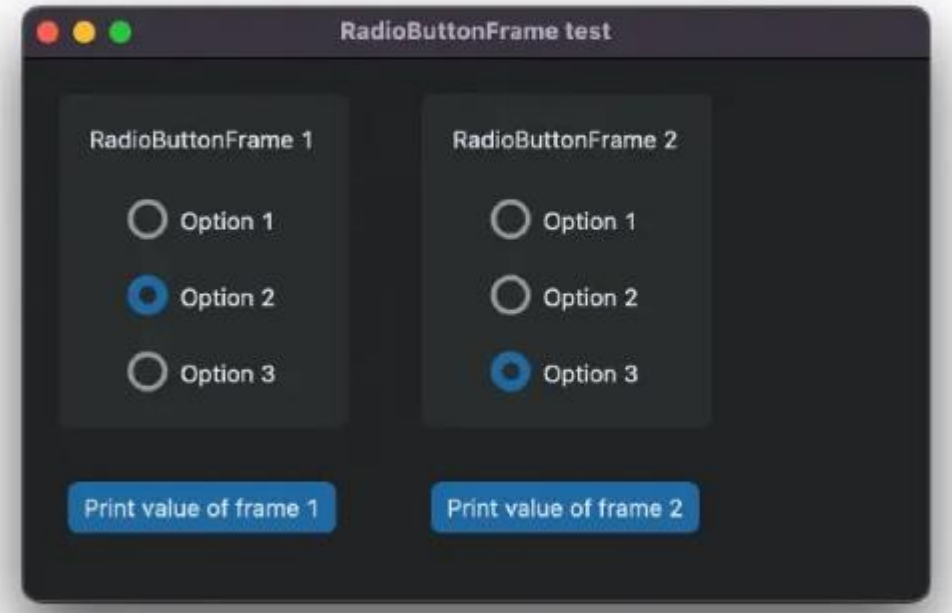
I also added a button a connected a command to print the current value of the radio buttonframe 1 to the terminal. The window will look like the following :



And because the radio button frame is structured as a class, there can easily be createdmultiple instances of it, like in the following example:

```
import customtkinter


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()

        self.geometry("500x300")
        self.title("RadioButtonFrame test")

        # create radio button frame 1 with print button
        self.radio_button_frame_1 = RadioButtonFrame(self, header_name="RadioButtonFram
        self.radio_button_frame_1.grid(row=0, column=0, padx=20, pady=20)

        self.frame_1_button = customtkinter.CTkButton(self, text="Print value of frame
        self.frame_1_button.grid(row=1, column=0, padx=20, pady=10)

        # create radio button frame 2 with print button
        self.radio_button_frame_2 = RadioButtonFrame(self, header_name="RadioButtonFram
        self.radio_button_frame_2.grid(row=0, column=1, padx=20, pady=20)

        self.frame_2_button = customtkinter.CTkButton(self, text="Print value of frame
        self.frame_2_button.grid(row=1, column=1, padx=20, pady=10)

    def print_value_frame_1(self):
        print(f"Frame 1 value: {self.radio_button_frame_1.get_value()}")

    def print_value_frame_2(self):
        print(f"Frame 2 value: {self.radio_button_frame_2.get_value()}")


if __name__ == "__main__":
    app = App()
```

```
app = App()
app.mainloop()
```



Of course you could also add another attribute to the RadioButtonFrame class, so that you cannot only customize the header_name, but also the values of the radio buttons.

CREATE NEW WIDGETS(SPINBOX)

If you want to create new widgets by yourself, because you need something that's notimplemented with Customtkinter, you can easily do this. In the following I created a Spinboxbased on float values.

First you create a new class, that inherits from customtkinter.CTkFrame :

```
import customtkinter

class WidgetName(customtkinter.CTkFrame):
    def __init__(self, *args,
                 width: int = 100,
                 height: int = 32,
                 **kwargs):
        super().__init__(*args, width=width, height=height, **kwargs)
```

By doing this, you already have a widget that has a default size given by the default values of the attributes width and height. And it also supports all attributes of a CTkFrame like fg_colorand corner_radius.

A simple implementation af a float based Spinbox could look like the following:

```python
class FloatSpinbox(customtkinter.CTkFrame):
    def __init__(self, *args,
                 width: int = 100,
                 height: int = 32,
                 step_size: Union[int, float] = 1,
                 command: Callable = None,
                 **kwargs):
        super().__init__(*args, width=width, height=height, **kwargs)

        self.step_size = step_size
        self.command = command

        self.configure(fg_color=("gray78", "gray28"))  # set frame color

        self.grid_columnconfigure((0, 2), weight=0)  # buttons don't expand
        self.grid_columnconfigure(1, weight=1)  # entry expands

        self.subtract_button = customtkinter.CTkButton(self, text="-", width=height-6,
                                                       command=self.subtract_button_cal
        self.subtract_button.grid(row=0, column=0, padx=(3, 0), pady=3)

        self.entry = customtkinter.CTkEntry(self, width=width-(2*height), height=height
        self.entry.grid(row=0, column=1, columnspan=1, padx=3, pady=3, sticky="ew")

        self.add_button = customtkinter.CTkButton(self, text="+", width=height-6, heigh
                                                  command=self.add_button_callback)
        self.add_button.grid(row=0, column=2, padx=(0, 3), pady=3)

        # default value
        self.entry.insert(0, "0.0")

    def add_button_callback(self):
        if self.command is not None:
            self.command()
        try:
            value = float(self.entry.get()) + self.step_size
            self.entry.delete(0, "end")
            self.entry.insert(0, value)
        except ValueError:
            return

    def subtract_button_callback(self):
        if self.command is not None:
            self.command()
        try:
            value = float(self.entry.get()) - self.step_size
            self.entry.delete(0, "end")
            self.entry.insert(0, value)
        except ValueError:
            return

    def get(self) -> Union[float, None]:
        try:
            return float(self.entry.get())
        except ValueError:
            return None
```

```
    def set(self, value: float):
        self.entry.delete(0, "end")
        self.entry.insert(0, str(float(value)))
```

Of course you could add a configure method or more attributes to configure the colors in theinit method.But with the above implementation you can use the spin box like the following:

```
app = customtkinter.CTk()

spinbox_1 = FloatSpinbox(app, width=150, step_size=3)
spinbox_1.pack(padx=20, pady=20)

spinbox_1.set(35)
print(spinbox_1.get())

app.mainloop()
```

CTK(Tkinter.Tk)

The CTk class creates the basis of any CustomTkinter program, it creates the main appwindow. During the runtime of a program there should only be one instance of these classwith a single call of the .mainloop() method, which starts the app. Additional windows arecreated using the CTkToplevel class.

Example Code:

Structure of a CustomTkinter program without any class (not recommended):

```
app = customtkinter.CTk()
app.geometry("600x500")
app.title("CTk example")

... program ...

app.mainloop()
```

Structure of a simple program with classes (recommended):

```
import customtkinter


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("600x500")
        self.title("CTk example")

        # add widgets to app
        self.button = customtkinter.CTkButton(self, command=self.button_click)
        self.button.grid(row=0, column=0, padx=20, pady=10)

    # add methods to app
    def button_click(self):
        print("button click")


app = App()
app.mainloop()
```

Arguments:

argument	value
fg_color	window background color, tuple: (light_color, dark_color) or single color

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

  ```
  app.configure(fg_color=new_fg_color)
  ```

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.

  ```
  fg_color = app.cget("fg_color")
  ...
  ```

- **.title(string)**

  Set title of window.

- **.geomtry(geometry_string)**

Set geometry and positions of the window like this: `"<width>x<height>"` or `"<width>x<height>+<x_pos>+<y_pos>"`

- **.minsize(width, height)**

  Set minimal window size.

- **.maxsize(width, height)**

  Set max window size.

- **.resizable(width, height)**

  Define, if width and/or height should be resizablee with bool values.

- **.after(milliseconds, command)**

  Execute command after milliseconds without blocking the main loop.

- **.withdraw()**

  Hide window and icon. Restore it with .deiconify().

- **.iconify()**

  Iconifies the window. Restore it with .deiconify().

- **.deiconify()**

  Deiconify the window.

- **.state()**

  Returns the window state: `'normal', 'iconic' or 'withdrawn'`

CTKTopLevel

The CTkToplevel class is used to create additional windows. For a CTkToplevel window, there isno additional call of .mainloop() needed, it opens right when it's created.

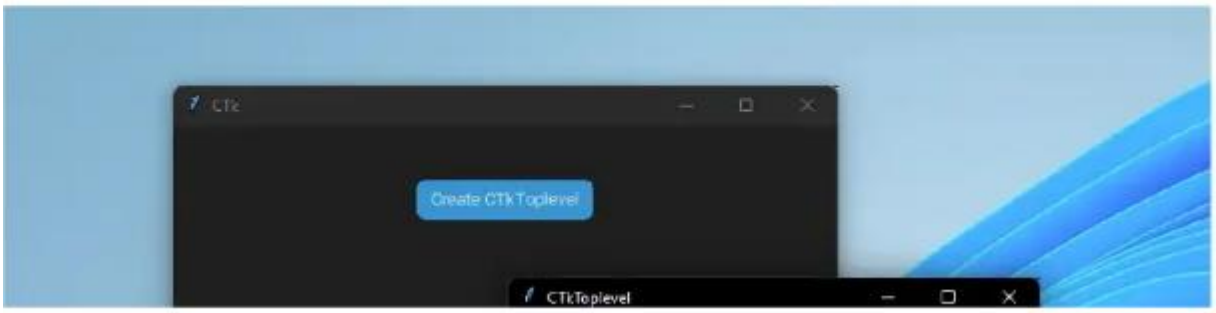
Example Code:

The following example shows how to create a toplevel window, which can be opened from themain app widnow. Before the toplevel window gets created, it is checked if the windowalready exists, to prevent opening the same window multiple times.

```
import customtkinter

class ToplevelWindow(customtkinter.CTkToplevel):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.geometry("400x300")

        self.label = customtkinter.CTkLabel(self, text="ToplevelWindow")
        self.label.pack(padx=20, pady=20)

class App(customtkinter.CTk):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.geometry("500x400")

        self.button_1 = customtkinter.CTkButton(self, text="open toplevel", command=sel
        self.button_1.pack(side="top", padx=20, pady=20)

        self.toplevel_window = None

    def open_toplevel(self):
        if self.toplevel_window is None or not self.toplevel_window.winfo_exists():
            self.toplevel_window = ToplevelWindow(self)  # create window if its None or
        else:
            self.toplevel_window.focus()  # if window exists focus it
```

```
if __name__ == "__main__":
    app = App()
    app.mainloop()
```

The example code results in the following windows:



Arguments:

argument	value
fg_color	window background color, tuple: (light_color, dark_color) or single color

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

```
toplevel.configure(fg_color=new_fg_color)
```

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.

```
fg_color = toplevel.cget("fg_color")
...
```

- **.title(string)**

  Set title of window.

- **.geomtry(geometry_string)**

Set geometry and positions of the window like this: `"<width>x<height>"` or `"<width>x<height>+<x_pos>+<y_pos>"`

- **.minsize(width, height)**

  Set minimal window size.

- **.maxsize(width, height)**

  Set max window size.

- **.resizable(width, height)**

  Define, if width and/or height should be resizablee with bool values.

- **.after(milliseconds, command)**

  Execute command after milliseconds without blocking the main loop.

- **.withdraw()**

  Hide window and icon. Restore it with .deiconify().

- **.iconify()**

  Iconifies the window. Restore it with .deiconify().

- **.deiconify()**

  Deiconify the window.

- **.state()**

  Returns the window state: `'normal', 'iconic' or 'withdrawn'`

CTKFrame

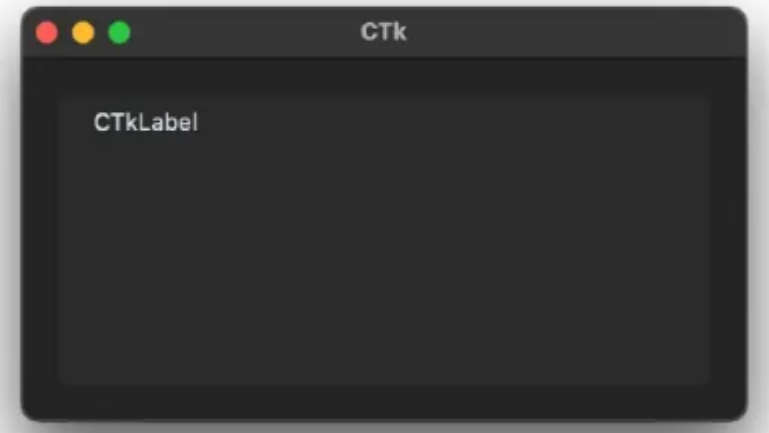
Example Code:

```
frame = customtkinter.CTkFrame(master=root_tk, width=200, height=200)
```

Frame structured into a class:

```
import customtkinter


class MyFrame(customtkinter.CTkFrame):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)

        # add widgets onto the frame...
        self.label = customtkinter.CTkLabel(self)
        self.label.grid(row=0, column=0, padx=20)


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("400x200")
        self.grid_rowconfigure(0, weight=1)  # configure grid system
        self.grid_columnconfigure(0, weight=1)

        self.my_frame = MyFrame(master=self)
        self.my_frame.grid(row=0, column=0, padx=20, pady=20, sticky="nsew")


app = App()
app.mainloop()
```

which results in:



Arguments:

argument	value
master	root, Frame or Toplevel
width	width in px
height	height in px
border_width	width of border in px
fg_color	foreground color, tuple: (light_color, dark_color) or single color or "transparent"
border_color	border color, tuple: (light_color, dark_color) or single color

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.

- **.bind(sequence=None, command=None, add=None)**

CTKSrollableFrame

Example Code:

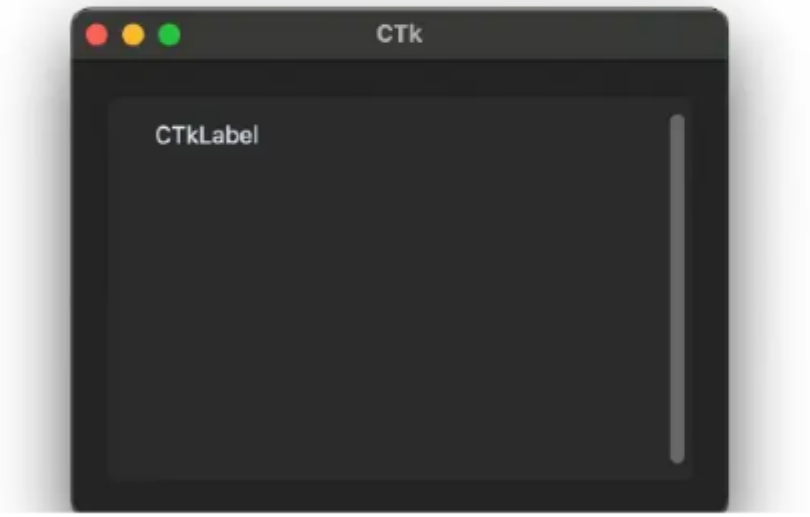
Standard scrollable frame with vertical orientation:

```
scrollable_frame = customtkinter.CTkScrollableFrame(master=root_tk, width=200, height=2
```

Scrollable frame structured into a class:

```
import customtkinter


class MyFrame(customtkinter.CTkScrollableFrame):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)

        # add widgets onto the frame...
        self.label = customtkinter.CTkLabel(self)
        self.label.grid(row=0, column=0, padx=20)


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()

        self.my_frame = MyFrame(master=self, width=300, height=200)
        self.my_frame.grid(row=0, column=0, padx=20, pady=20)


app = App()
app.mainloop()
```

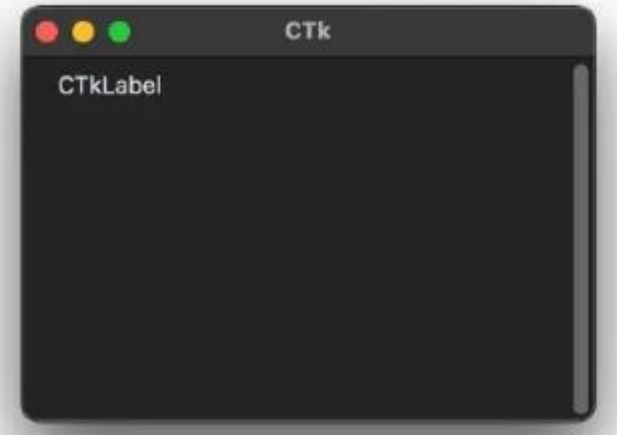
which results in:



The above example can be slightly modified, so that the scrollable frame completely fills the app window:

```python
class MyFrame(customtkinter.CTkScrollableFrame):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)

        # add widgets onto the frame...
        self.label = customtkinter.CTkLabel(self)
        self.label.grid(row=0, column=0, padx=20)


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()
        self.grid_rowconfigure(0, weight=1)
        self.grid_columnconfigure(0, weight=1)

        self.my_frame = MyFrame(master=self, width=300, height=200, corner_radius=0, fg
        self.my_frame.grid(row=0, column=0, sticky="nsew")


app = App()
app.mainloop()
```

which results in:



Arguments:

argument	value
master	root, Frame or Toplevel
width	width in px (inner frame dimensions)
height	height in px (inner frame dimensions)
corner_radius	corner radius in px
border_width	border width in px
fg_color	foreground color, tuple: (light_color, dark_color) or single color or "transparent"
border_color	border color, tuple: (light_color, dark_color) or single color
scrollbar_fg_color	scrollbar foreground color, tuple: (light_color, dark_color) or single color
scrollbar_button_color	scrollbar button color, tuple: (light_color, dark_color) or single color
scrollbar_button_hover_color	scrollbar button hover color, tuple: (light_color, dark_color) or single color
label_fg_color	label foreground color color, tuple: (light_color, dark_color) or single color
label_text_color	label text color, tuple: (light_color, dark_color) or single color
label_text	label text for label (title for frame)
label_font	font for label, tupel font or CTkFont
label_anchor	anchor for label, orientation of text, ("n", "ne", "e", "se", "s", "sw", "w", "nw", "center")
orientation	scrolling direction, "vertical" (default), "horizontal"

Methods:

- .configure(attribute=value, ...)

  All attributes can be configured and updated.

- .cget(attribute_name)

  Pass attribute name as string and get current value of attribute.

CTKTabView

The CTkTabview creates a tabview, similar to a notebook in tkinter. The tabs, which are createdwith .add("<tab-name>") are CTkFrames and can be used like CTkFrames. Any widgets can beplaced on them.

Example Code:

Example without using classes (not recommended):

```
tabview = customtkinter.CTkTabview(app)
tabview.pack(padx=20, pady=20)

tabview.add("tab 1")  # add tab at the end
tabview.add("tab 2")  # add tab at the end
tabview.set("tab 2")  # set currently visible tab

button_1 = customtkinter.Button(tabview.tab("tab 1"))
button_1.pack(padx=20, pady=20)
```

It's also possible to save tabs in extra variables:

```
tab_1 = tabview.add("tab 1")
tab_2 = tabview.add("tab 2")

button_1 = customtkinter.Button(tab_1)
```

Example with classes (recommended):

```
import customtkinter


class MyTabView(customtkinter.CTkTabview):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)

        # create tabs
        self.add("tab 1")
        self.add("tab 2")

        # add widgets on tabs
        self.label = customtkinter.CTkLabel(master=self.tab("tab 1"))
        self.label.grid(row=0, column=0, padx=20, pady=10)


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()

        self.tab_view = MyTabView(master=self)
```

```
        self.tab_view.grid(row=0, column=0, padx=20, pady=20)


app = App()
app.mainloop()
```

which results in:

Arguments:

argument	value
master	root, frame, top-level
width	width in px, tabs will be slightly smaller
height	height in px, tabs will be slightly smaller
corner_radius	corner radius in px
border_width	border width in px
fg_color	foreground color of the tabview itself and the tabs, tuple: (light_color, dark_color) or single color
border_color	border color, tuple: (light_color, dark_color) or single color

segmented_button_fg_color	foreground color of segmented button, tuple: (light_color, dark_color) or single color
segmented_button_selected_color	selected color of segmented button, tuple: (light_color, dark_color) or single color
segmented_button_selected_hover_color	selected hover color of segmented button, tuple: (light_color, dark_color) or single color
segmented_button_unselected_color	unselected color of segmented button, tuple: (light_color, dark_color) or single color
segmented_button_unselected_hover_color	unselected hover color of segmented button, tuple: (light_color, dark_color) or single color
text_color	text color of segmented button, tuple: (light_color, dark_color) or single color
text_color_disabled	text color of segmented buttons when widget is disabled, tuple: (light_color, dark_color) or single color

command	function will be called when segmented button is clicked
state	"normal" or "disabled"

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

- **.cget(attribute_name)**

  Get values of all attributes specified as string.

- **.tab(name)**

  Returns reference to tab with given name.

```
button = customtkinter.CTkButton(master=tabview.tab("tab name"))
```

- **.insert(index, name)**

  Insert tab with name at position of index, name must be unique.

- **.add(name)**

  Add tab with name at the end, name must be unique.

- **.delete(name)**

  Delete tab with name.

- **.set(name)**

  Set tab with name to be visible.

- **.get()**

  Get name of tab that's currently visible.

CTKTexBox

The CTkTextbox class creates a textbox, which is scrollable in vertical and horizontal direction(with wrap='none'). The Insert , get and delete methods are based on indices, which areexplained here:https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/text-index.html

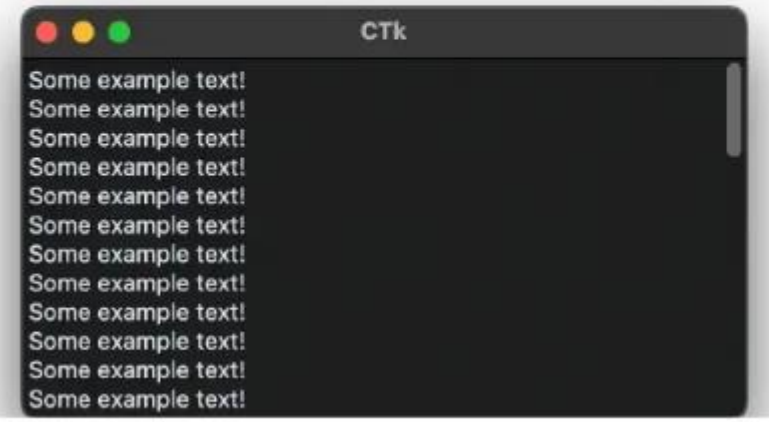
Example Code:

```
textbox = customtkinter.CTkTextbox(app)
textbox.grid(row=0, column=0)

textbox.insert("0.0", "new text to insert")  # insert at line 0 character 0
text = textbox.get("0.0", "end")  # get text from line 0 character 0 till the end
textbox.delete("0.0", "end")  # delete all text
textbox.configure(state="disabled")  # configure textbox to be read-only
```

Example program where the textbox completely fills the window:

```
import customtkinter


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()
        self.grid_rowconfigure(0, weight=1)  # configure grid system
        self.grid_columnconfigure(0, weight=1)

        self.textbox = customtkinter.CTkTextbox(master=self, width=400, corner_radius=0
        self.textbox.grid(row=0, column=0, sticky="nsew")
        self.textbox.insert("0.0", "Some example text!\n" * 50)


app = App()
app.mainloop()
```

which results in:



Arguments:

argument	value
master	root, frame, top-level
width	box width in px
height	box height in px
corner_radius	corner radius in px
border_width	border width in px
border_spacing	minimum space between text and widget border, default is 3. Set to 0 for the text to touch the widget border (if corner_radius=0)
fg_color	main widget color, tuple: (light_color, dark_color) or single color or "transparent"
border_color	border color, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
scrollbar_button_color	main color of scrollbar, tuple: (light_color, dark_color) or single color
scrollbar_button_hover_color	hover color of scrollbar, tuple: (light_color, dark_color) or single color
font	text font, tuple: (font_name, size)
activate_scrollbars	default is True, set to False to prevent scrollbars from appearing
state	"normal" (standard) or "disabled" (not clickable, read-only)
wrap	how to wrap text at end of line, default is 'char', other options are 'word' or 'none' for no wrapping at all and horizontal scrolling

and the following arguments of the tkinter.Text class:

```
"autoseparators", "cursor", "exportselection", "insertborderwidth", "insertofftime",
```

`"insertontime", "insertwidth", "maxundo", "padx", "pady", "selectborderwidth", "spacing1", "spacing2", "spacing3", "state", "tabs", "takefocus", "undo", "xscrollcommand", "yscrollcommand"`

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

  ```
  textbox.configure(state=..., text_color=..., ...)
  ```

- **.cget(attribute_name)**

  Get values of all attributes specified as string.

- **.bind(sequence=None, command=None, add=None)**

  Bind commands to events specified by sequence string.

- **.unbind(sequence, funcid=None)**

  Unbind command from sequence specified by funcid, which is returned by .bind().

- **.insert(index, text, tags=None)**

  Insert text at given index. Index for the tkinter.Text class is specified by 'line.character', 'end', 'insert' or other keywords described here: https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/text-index.html

- **.delete(self, index1, index2=None)**

  Delete the characters between index1 and index2 (not included).

- **.get(index1, index2=None)**

  Return the text from INDEX1 to INDEX2 (not included).

- **.focus_set()**

  Set focus to the text widget.

and nearly all other methods of tkinter.Text described here: https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/text-methods.html

CTKScrollbar

Example Code:

Default theme:

```
import customtkinter, tkinter

app = customtkinter.CTk()
app.grid_rowconfigure(0, weight=1)
app.grid_columnconfigure(0, weight=1)

# create scrollable textbox
tk_textbox = tkinter.Text(app, highlightthickness=0)
tk_textbox.grid(row=0, column=0, sticky="nsew")

# create CTk scrollbar
ctk_textbox_scrollbar = customtkinter.CTkScrollbar(app, command=tk_textbox.yview)
ctk_textbox_scrollbar.grid(row=0, column=1, sticky="ns")

# connect textbox scroll event to CTk scrollbar
tk_textbox.configure(yscrollcommand=ctk_textbox_scrollbar.set)

app.mainloop()
```

Arguments:

argument	value
master	root, tkinter.Frame or CTkFrame
command	function of scrollable widget to call when scrollbar is moved
width	button width in px
height	button height in px
corner_radius	corner radius in px
border_spacing	foreground space around the scrollbar in px (border)
fg_color	forground color, tuple: (light_color, dark_color) or single color or "transparent"
button_color	scrollbar color, tuple: (light_color, dark_color) or single color
button_hover_color	scrollbar hover color, tuple: (light_color, dark_color) or single color
minimum_pixel_length	minimum length of scrollbar in px
orientation	"vertical" (standard), "horizontal"

hover	hover effect (True/False)

Methods:

```
scrollbar = CTkScrollbar(master)
scrollbar.configure(...)
value_1, value_2 = scrollbar.get()
scrollbar.set(value_1, value_2)
```

CTKButton

Example Code:

Default theme:

```
def button_event():
    print("button pressed")

button = customtkinter.CTkButton(master=root_tk, text="CTkButton", command=button_event
button.pack(padx=20, pady=10)
```

Customized:

```
button = customtkinter.CTkButton(master=root_tk,
                                 width=120,
                                 height=32,
                                 border_width=0,
                                 corner_radius=8,
                                 text="CTkButton",
                                 command=button_event)
button.place(relx=0.5, rely=0.5, anchor=tkinter.CENTER)
```

Arguments:

argument	value
master	root, tkinter.Frame or CTkFrame
width	button width in px
height	button height in px
corner_radius	corner radius in px
border_width	button border width in px
border_spacing	spacing between text and image and button border in px, default is 2
fg_color	forground color, tuple: (light_color, dark_color) or single color or "transparent"
hover_color	hover color, tuple: (light_color, dark_color) or single color
border_color	border color, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
text_color_disabled	text color when disabled, tuple: (light_color, dark_color) or single color

text	string
font	button text font, tuple: (font_name, size), (set negative size value for size in pixels)
textvariable	tkinter.StringVar object to change text of button
image	put an image on the button, removes the text, must be class PhotoImage
state	"normal" (standard) or "disabled" (not clickable, darker color)
hover	enable/disable hover effect: True, False
command	callback function
compound	set image orientation if image and text are given ("top", "left", "bottom", "right")
anchor	alignment of text an image in button ("n", "ne", "e", "se", "s", "sw", "w", "nw", "center")

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

```
ctk_button.configure(border_width=new_borderwidth)
ctk_button.configure(image=new_image)
ctk_button.configure(text=new_text)
ctk_button.configure(command=callback_function)
...
```

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.

```
state = ctk_button.cget("state")
text = ctk_button.cget("text")
...
```

- **.invoke()**

  Calls command if button state is 'disabled'.

CTKLabel

textvariable	tkinter.StringVar object
text	string
width	label width in px
height	label height in px
corner_radius	corner radius in px
fg_color	foreground color, tuple: (light_color, dark_color) or single color or "transparent"
text_color	label text color, tuple: (light_color, dark_color) or single color
font	label text font, tuple: (font_name, size)
anchor	controls where the text is positioned if the widget has more space than the text needs, default is "center"
compound	control the postion of image relative to text, default is "center, other are: "top", "bottom", "left", "right"
	specifies how multiple lines of text will be aligned with respect to each
justify	other: "left" for flush left, "center" for centered (the default), or "right" for right-justified
padx	extra space added left and right of the text, default is 1
pady	extra space added above and belowof the text, default is 1

and other arguments of `tkinter.Label`

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

  ```
  ctk_label.configure(text=new_text)
  ...
  ```

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.

```
text = ctk_label.cget("text")
...
```

- .bind(sequence=None, command=None, add=None)

  Bind events to the label.

CTKEntry

Example Code:

Default theme:

```
entry = customtkinter.CTkEntry(master=root_tk, placeholder_text="CTkEntry")
entry.pack(padx=20, pady=10)
```

Customized:

```
entry = customtkinter.CTkEntry(master=root_tk,
                               placeholder_text="CTkEntry",
                               width=120,
                               height=25,
                               border_width=2,
                               corner_radius=10)
entry.place(relx=0.5, rely=0.5, anchor=tkinter.CENTER)
```

Arguments

argument	value
master	root, tkinter.Frame or CTkFrame
textvariable	tkinter.StringVar object
width	entry width in px
height	entry height in px
corner_radius	corner radius in px
fg_color	foreground color, tuple: (light_color, dark_color) or single color or "transparent"
text_color	entry text color, tuple: (light_color, dark_color) or single color
placeholder_text_color	tuple: (light_color, dark_color) or single color
placeholder_text	hint on the entry input (disappears when selected), default is None, don't works in combination with a textvariable
font	entry text font, tuple: (font_name, size)
state	"normal" (standard) or "disabled" (not clickable)

and the following arguments of the tkinter.Entry class:

```
"exportselection", "insertborderwidth", "insertofftime", "insertontime",
"insertwidth", "justify", "selectborderwidth", "show", "takefocus", "validate",
```

"validatecommand", "xscrollcommand"

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

```
ctk_entry.configure(state=new_state)
ctk_entry.configure(textvariable=textvariable)
...
```

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.

```
state = ctk_entry.cget("state")
...
```

- **.bind(sequence=None, command=None, add=None)**

  Bind command function to event specified by sequence.

- **.delete(first_index, last_index=None)**

  Deletes characters from the widget, starting with the one at index first_index, up to but not including the character at position last_index. If the second argument is omitted, only the single character at position first is deleted.

- **.insert(index, string)**

  Inserts string before the character at the given index.

- **.get()**

  Returns current string in the entry.

- **.focus()**

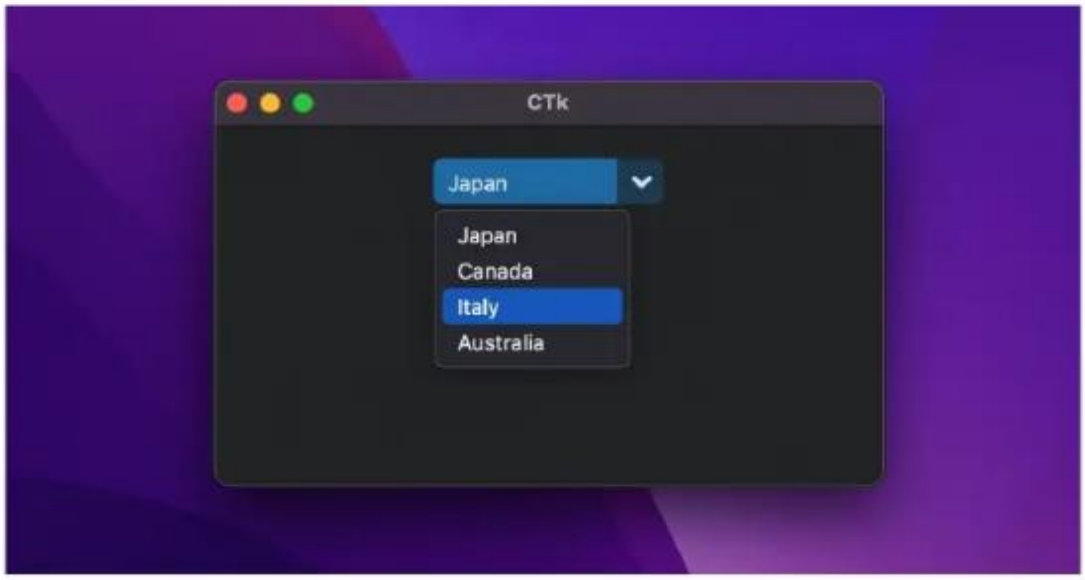
- **.focus_force()**

- **.index(index)**

- **.icursor(index)**

- .select_adjust(index)
- .select_from(index)
- .select_clear()
- .select_present()
- .select_range(start_index, end_index)
- .select_to(index)
- .xview(index)
- .xview_moveto(f)
- .xview_scroll(number, what)

CTKOptionMenu



Example Code:

Without variable:

```
def optionmenu_callback(choice):
    print("optionmenu dropdown clicked:", choice)

combobox = customtkinter.CTkOptionMenu(master=app,
                                       values=["option 1", "option 2"],
                                       command=optionmenu_callback)
combobox.pack(padx=20, pady=10)
combobox.set("option 2")  # set initial value
```

With variable:

```
optionmenu_var = customtkinter.StringVar(value="option 2")  # set initial value

def optionmenu_callback(choice):
    print("optionmenu dropdown clicked:", choice)

combobox = customtkinter.CTkOptionMenu(master=app,
                                       values=["option 1", "option 2"],
                                       command=optionmenu_callback,
                                       variable=optionmenu_var)
combobox.pack(padx=20, pady=10)
```

Arguments:

argument	value
master	root, frame, top-level

width	box width in px
height	box height in px
corner_radius	corner radius in px
fg_color	foreground (inside) color, tuple: (light_color, dark_color) or single color
button_color	right button color, tuple: (light_color, dark_color) or single color
button_hover_color	hover color, tuple: (light_color, dark_color) or single color
dropdown_fg_color	dropdown fg color, tuple: (light_color, dark_color) or single color
dropdown_hover_color	dropdown button hover color, tuple: (light_color, dark_color) or single color
dropdown_text_color	dropdown text color, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
text_color_disabled	text color when disabled, tuple: (light_color, dark_color) or single color
font	button text font, tuple: (font_name, size)
dropdown_font	button text font, tuple: (font_name, size)
hover	enable/disable hover effect: True, False
state	"normal" (standard) or "disabled" (not clickable, darker color)
command	function will be called when the dropdown is clicked manually
variable	StringVar to control or get the current text
values	list of strings with values that appear in the option menu dropdown
dynamic_resizing	enable/disable automatic resizing of optiomenu when text is too big to fit: True (standard), False
anchor	"n", "s", "e", "w", "center", orientation of the text inside the optionmenu, default is "w"

Methods:

- **.configure(attribute=value, ...)**

All attributes can be configured and updated.

```
ctk_combobox.configure(state=..., command=..., values=[...], variable=..., ...)
```

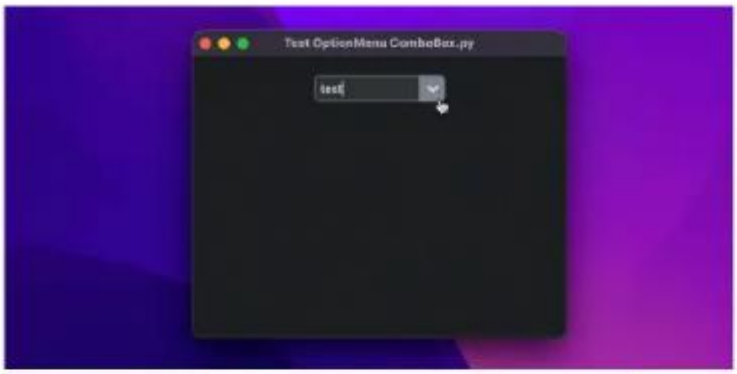
- .set(value)

  Set optionemnu to specific string value. Value don't has to be part of the values list.

- .get()

  Get current string value of optionmenu.

CTKComboBox



Example Code:

Without variable:

```
def combobox_callback(choice):
    print("combobox dropdown clicked:", choice)

combobox = customtkinter.CTkComboBox(master=app,
                                     values=["option 1", "option 2"],
                                     command=combobox_callback)
combobox.pack(padx=20, pady=10)
combobox.set("option 2")  # set initial value
```

With variable:

```
combobox_var = customtkinter.StringVar(value="option 2")  # set initial value

def combobox_callback(choice):
    print("combobox dropdown clicked:", choice)

combobox = customtkinter.CTkComboBox(master=app,
                                     values=["option 1", "option 2"],
                                     command=combobox_callback,
                                     variable=combobox_var)
combobox.pack(padx=20, pady=10)
```

Arguments:

argument	value
master	root, frame, top-level
width	box width in px
height	box height in px
corner_radius	corner radius in px
border_width	border width in px
fg_color	foreground (inside) color, tuple: (light_color, dark_color) or single color
border_color	border color, tuple: (light_color, dark_color) or single color
button_color	right button color, tuple: (light_color, dark_color) or single color
button_hover_color	hover color, tuple: (light_color, dark_color) or single color
dropdown_fg_color	dropdown fg color, tuple: (light_color, dark_color) or single color
dropdown_hover_color	dropdown button hover color, tuple: (light_color, dark_color) or single color
dropdown_text_color	dropdown text color, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
text_color_disabled	text color when disabled, tuple: (light_color, dark_color) or single color
font	button text font, tuple: (font_name, size)
dropdown_font	button text font, tuple: (font_name, size)
hover	enable/disable hover effect: True, False
state	"normal" (standard), "disabled" (not clickable, darker color), "readonly"
command	function will be called when the dropdown is clicked manually
variable	StringVar to control or get the current text
justify	"right", "left", "center", orientation of the text inside the entry, default is "left"

Methods:

- .configure(attribute=value, ...)

  All attributes can be configured and updated.

```
ctk_combobox.configure(state=..., command=..., values=[...], variable=..., ...)
```

- .set(value)

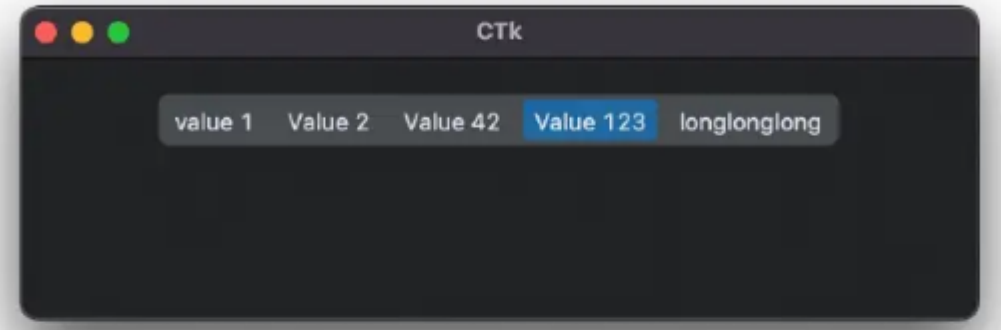
  Set combobox to specific string value. Value don't has to be part of the values list.

- .get()

  Get current string value of combobox entry.

CTKSegmentedButton

Widget will be available with version 5.0.0!



Example Code:

Without variable:

```
def segmented_button_callback(value):
    print("segmented button clicked:", value)

segemented_button = customtkinter.CTkSegmentedButton(master=app,
                                                     values=["Value 1", "Value 2", "Val
                                                     command=segmented_button_callback)
segemented_button.pack(padx=20, pady=10)
segemented_button.set("Value 1")  # set initial value
```

With variable:

```
segemented_button_var = customtkinter.StringVar(value="Value 1")  # set initial value

segemented_button = customtkinter.CTkSegmentedButton(master=app,
                                                     values=["Value 1", "Value 2", "Val
                                                     variable=segemented_button_var)
segemented_button.pack(padx=20, pady=10)
```

Arguments:

argument	value
master	root, frame, top-level
width	box width in px

height	box height in px
corner_radius	corner radius in px
border_width	space in px between buttons and the edges of the widget
fg_color	color around the buttons, tuple: (light_color, dark_color) or single color
selected_color	color of the selected button, tuple: (light_color, dark_color) or single color
selected_hover_color	hover color of selected button, tuple: (light_color, dark_color) or single color
unselected_color	color of the unselected buttons, tuple: (light_color, dark_color) or single color or "transparent"
unselected_hover_color	hover color of unselected buttons, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
text_color_disabled	text color when disabled, tuple: (light_color, dark_color) or single color
font	button text font, tuple: (font_name, size)
values	list of string values for the buttons, can't be empty
variable	StringVar to control the current selected value
state	"normal" (standard) or "disabled" (not clickable, darker color)
command	function will be called when the dropdown is clicked manually
dynamic_resizing	enable/disable automatic resizing when text is too big to fit: True (standard), False

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

```
segemented_button.configure(state=..., command=..., values=[...], variable=..., ...
```

- **.cget(attribute_name)**

  Get values of all attributes specified as string.

- **.set(value)**

  Set to specific value. If value is not in values list, no button will be selected.

- **.get()**

  Get current string value.

- **.insert(index, value)**

  Insert new value at given index into segmented button. Value will be also inserted into the values list.

- **.move(new_index, value)**

  Move existing value to new index position.

- **.delete(value)**

  Remove value from segmented button and values list. If value is currently selected, no button will be selected afterwards.

CTKSwitch

The switch should behave similar to the CTkCheckBox.

Example Code:

Default theme:

```
switch_var = customtkinter.StringVar(value="on")

def switch_event():
    print("switch toggled, current value:", switch_var.get())

switch_1 = customtkinter.CTkSwitch(master=root_tk, text="CTkSwitch", command=switch_eve
                                   variable=switch_var, onvalue="on", offvalue="off")
switch_1.pack(padx=20, pady=10)
```

Arguments:

argument	value
master	master widget
width	width of complete widget in px
height	height of complete widget in px
switch_width	width of switch in px
switch_height	height of switch in px
corner_radius	corner radius in px
border_width	box border width in px
fg_color	foreground (inside) color, tuple: (light_color, dark_color) or single color
border_color	border color, tuple: (light_color, dark_color) or single color or "transparent" default is "transparent"
progress_color	color of switch when enabled, tuple: (light_color, dark_color) or single color or "transparent"
button_color	color of button, tuple: (light_color, dark_color) or single color
button_hover_color	hover color of button, tuple: (light_color, dark_color) or single color
hover_color	hover color, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
text	string

textvariable	Tkinter StringVar to control the text
font	button text font, tuple: (font_name, size)
command	function will be called when the checkbox is clicked or .toggle() is called
variable	Tkinter variable to control or read checkbox state
onvalue	string or int for variable in checked state
offvalue	string or int for variable in unchecked state
state	"normal" (standard) or "disabled" (not clickable)

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

  ```
  ctk_switch.configure(fg_color=..., progress_color=..., command=..., text=..., ...)
  ```

- **.get()**

  Get current value of switch, 1 or 0 (checked or not checked).

- **.select()**

  Turn on switch (set value to 1), command will not be triggered.

- **.deselect()**

  Turn off switch (set value to 0), command will not be triggered.

- **.toggle()**

  Flip current value of switch, command will be triggered.

CTKCheckBox

Example Code:

Default theme:

```
check_var = tkinter.StringVar("on")

def checkbox_event():
    print("checkbox toggled, current value:", check_var.get())

checkbox = customtkinter.CTkCheckBox(master=root_tk, text="CTkCheckBox", command=checkb
                                     variable=check_var, onvalue="on", offvalue="off")
checkbox.pack(padx=20, pady=10)
```

Arguments:

argument	value
master	root, tkinter.Frame or CTkFrame
width	width of complete widget in px
height	height of complete widget in px
checkbox_width	width of checkbox in px
checkbox_height	height of checkbox in px
corner_radius	corner radius in px
border_width	box border width in px
fg_color	foreground (inside) color, tuple: (light_color, dark_color) or single color
border_color	border color, tuple: (light_color, dark_color) or single color
hover_color	hover color, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
text_color_disabled	text color when disabled, tuple: (light_color, dark_color) or single color
text	string
textvariable	Tkinter StringVar to control the text
font	button text font, tuple: (font_name, size)
hover	enable/disable hover effect: True, False
state	tkinter.NORMAL (standard) or tkinter.DISABLED (not clickable,

state	darker color)
command	function will be called when the checkbox is clicked
variable	Tkinter variable to control or read checkbox state
onvalue	string or int for variable in checked state
offvalue	string or int for variable in unchecked state

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

  ```
  ctk_checkbox.configure(fg_color=..., command=..., text=..., ...)
  ```

- **.get()**

  Get current value, 1 or 0 (checked or not checked).

- **.select()**

  Turn on checkbox (set value to 1), command will not be triggered.

- **.select()**

  Turn on checkbox (set value to 1), command will not be triggered.

- **.deselect()**

  Turn off checkbox (set value to 0), command will not be triggered.

- **.toggle()**

  Flip current value, command will be triggered.

CTKRadioButton

This CTkRadioButton works similar to the `tkinter.Radiobutton`.

Example Code:

Default theme:

```
radio_var = tkinter.IntVar(0)

def radiobutton_event():
    print("radiobutton toggled, current value:", radio_var.get())

radiobutton_1 = customtkinter.CTkRadioButton(master=root_tk, text="CTkRadioButton 1",
                                             command=radiobutton_event, variable= radio
radiobutton_2 = customtkinter.CTkRadioButton(master=root_tk, text="CTkRadioButton 2",
                                             command=radiobutton_event, variable= radio

radiobutton_1.pack(padx=20, pady=10)
radiobutton_2.pack(padx=20, pady=10)
```

Arguments:

argument	value
master	root, tkinter.Frame or CTkFrame
width	width of complete widget in px
height	height of complete widget in px
radiobutton_width	width of radiobutton in px
radiobutton_height	height of radiobutton in px
corner_radius	corner radius in px
border_width_unchecked	border width in unchecked state in px
border_width_checked	border width in checked state in px
fg_color	foreground (inside) color, tuple: (light_color, dark_color) or single color
border_color	border color, tuple: (light_color, dark_color) or single color
hover_color	hover color, tuple: (light_color, dark_color) or single color
text_color	text color, tuple: (light_color, dark_color) or single color
text_color_disabled	text color when disabled, tuple: (light_color, dark_color) or single color

text	string
textvariable	Tkinter StringVar to control the text
font	button text font, tuple: (font_name, size)
hover	enable/disable hover effect: True, False
state	tkinter.NORMAL (standard) or tkinter.DISABLED (not clickable, darker color)
command	function will be called when the checkbox is clicked
variable	Tkinter variable to control or read checkbox state
value	string or int value for variable when RadioButton is clicked

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

```
ctk_radiobutton.configure(fg_color=..., command=..., text=..., ...)
```

- **.select()**

  Select radio button (set value to 1), command will not be triggered.

- **.deselect()**

  Deselect radio button (set value to 0), command will not be triggered.

- **.invoke()**

  Same action as if user would click the radio button, command will be triggered.

CTKSlide

Example Code:

Default theme:

```
def slider_event(value):
    print(value)

slider = customtkinter.CTkSlider(master=root_tk, from_=0, to=100, command=slider_event)
slider.place(relx=0.5, rely=0.5, anchor=tkinter.CENTER)
```

Arguments:

argument	value
master	root, tkinter.Frame or CTkFrame
command	callback function, receives slider value as argument
variable	tkinter.IntVar or tkinter.DoubleVar object
width	slider width in px
height	slider height in px
border_width	space around the slider rail in px
from_	lower slider value
to	upper slider value
number_of_steps	number of steps in which the slider can be positioned
fg_color	foreground color, tuple: (light_color, dark_color) or single color
progress_color	tuple: (light_color, dark_color) or single color or "transparent", color of the slider line before the button
border_color	slider border color, tuple: (light_color, dark_color) or single color or "transparent", default is "transparent"
button_color	color of the slider button, tuple: (light_color, dark_color) or single color
button_hover_color	hover color, tuple: (light_color, dark_color) or single color
orientation	"horizontal" (standard) or "vertical"
state	"normal" or "disabled" (not clickable)
hover	bool, enable/disable hover effect, default is True

Methods:

- .configure(attribute=value, ...)

  All attributes can be configured and updated.

```
ctk_slider.configure(fg_color=..., progress_color=..., button_color=..., ...)
```

- .set(value)

  Set slider to specific float value (range 0-1).

- .get()

  Get current value of slider (range 0-1).

```
value = ctk_slider.get()
```

Pub ×

CTKProgressBar

Example Code:

Default theme:

```
progressbar = customtkinter.CTkProgressBar(master=root_tk)
progressbar.pack(padx=20, pady=10)
```

Arguments:

argument	value
master	root, tkinter.Frame or CTkFrame
width	slider width in px
height	slider height in px
border_width	border width in px
corner_radius	corner_radius in px
fg_color	foreground color, tuple: (light_color, dark_color) or single color
border_color	slider border color, tuple: (light_color, dark_color) or single color
progress_color	progress color, tuple: (light_color, dark_color) or single color
orientation	"horizontal" (default) or "vertical"
mode	"determinate" for linear progress (default), "indeterminate" for unknown progress
determinate_speed	speed for automatic progress in determinate mode started by .start(), default is 1
indeterminate_speed	speed for automatic progress in indeterminate mode started by .start(), default is 1

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

  ```
  ctk_progressbar.configure(fg_color=..., progress_color=..., ...)
  ```

- **.set(value)**

Set progress bar to specific value (range 0 to 1).

- **.get()**

  Get current value of progress bar.

```
value = ctk_progressbar.get()
```

- **.start()**

  Start automatic progress. Speed is set by `indeterminate_speed` and `determinate_speed` attributes, default is 1. If mode is set to "indeterminate", the progress bar will oscillate, otherwise it will fill up and then repeat.

- **.stop()**

  Stop automatic progress.

- **.step()**

  Do single step manually otherwise done by .start() and .stop() automatic loop. Step size is set by determinate_speed and indeterminate_speed attributes.

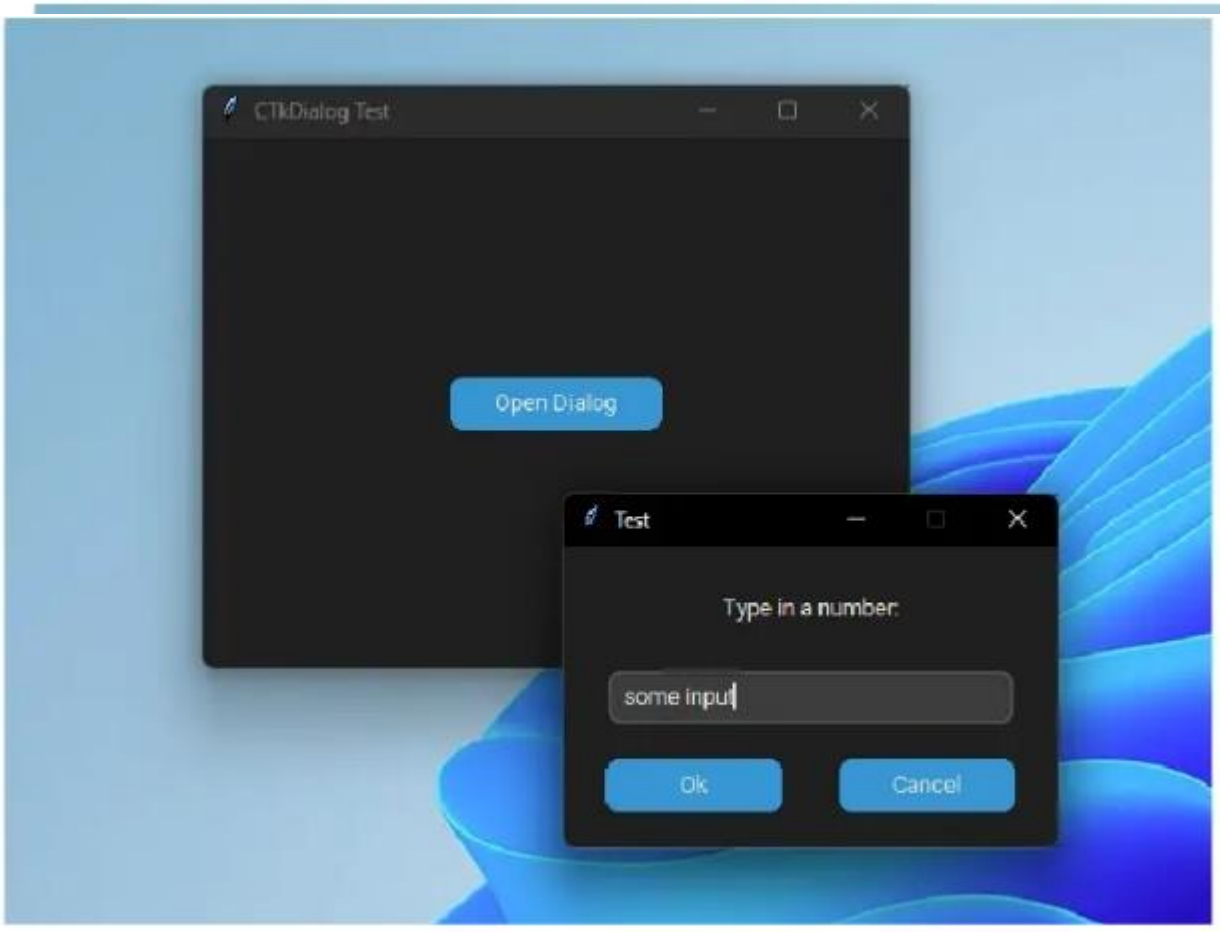
CTKInputDialog

This is a simple dialog to input a string or number.

Example Code:

```
import customtkinter
import tkinter

customtkinter.set_appearance_mode("dark")

app = customtkinter.CTk()
app.geometry("400x300")


def button_click_event():
    dialog = customtkinter.CTkInputDialog(text="Type in a number:", title="Test")
    print("Number:", dialog.get_input())


button = customtkinter.CTkButton(app, text="Open Dialog", command=button_click_event)
button.place(relx=0.5, rely=0.5, anchor=tkinter.CENTER)

app.mainloop()
```

This example code results in the following window and dialog:



Arguments:

argument	value
title	string for the dialog title
text	text for the dialog itself
fg_color	window color, tuple: (light_color, dark_color) or single color
button_fg_color	color of buttons, tuple: (light_color, dark_color) or single color
button_hover_color	hover color of buttons, tuple: (light_color, dark_color) or single color
button_text_color	text color of buttons, tuple: (light_color, dark_color) or single color
entry_fg_color	color of entry, tuple: (light_color, dark_color) or single color
entry_border_color	border color of entry, tuple: (light_color, dark_color) or single color
entry_text_color	text color of entry, tuple: (light_color, dark_color) or single color

Methods:

```
input_value = dialog.get_input()  # opens the dialog
```

CTKFont

Widget will be available with version 5.0.0!

There are two methods in CustomTkinter to set a font. The first one is by creating a tuple of the form:

```
button = customtkinter.CTkButton(app, font=("<family name>", <size in px>, "<optional k
```

This font can not be configured afterwards. The optional keywords can be normal/bold, roman/italic, underline and overstrike.

The better way is to create a CTkFont object, which can be modified afterwards and can be used for multiple widgets:

```
button = customtkinter.CTkButton(app, font=customtkinter.CTkFont(family="<family name>"

button.cget("font").configure(size=new_size)  # configure font afterwards
```

All widgets get a CTkFont object by default, so configuring the font of a widget is possible by default.

A font object can also be applied to multiple widgets, and if it gets configured, the changes get passed to all widgets using this font:

```
my_font = customtkinter.CTkFont(family="<family name>", size=<size in px>, <optional ke

button_1 = customtkinter.CTkButton(app, font=my_font)
button_2 = customtkinter.CTkButton(app, font=my_font)

my_font.configure(family="new name")  # changes apply to button_1 and button_2
```

Arguments:

argument	value
family	The font family name as a string.
size	The font height as an integer in pixel.
weight	'bold' for boldface, 'normal' for regular weight.
slant	'italic' for italic, 'roman' for unslanted.
underline	True for underlined text, False for normal.
overstrike	True for overstruck text, False for normal.

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and get updated.

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.

- **.measure(text)**

  Pass this method a string, and it will return the number of pixels of width that string will take in the font.

- **.metrics(option)**

  If you call this method with no arguments, it returns a dictionary of all the font metrics. You can retrieve the value of just one metric by passing its name as an argument. Metrics include:

  ascent: Number of pixels of height between the baseline and the top of the highest ascender.

  descent: Number of pixels of height between the baseline and the bottom of the lowest ascender.

Pub

descent: Number of pixels of height between the baseline and the bottom of the lowest ascender.

fixed: This value is 0 for a variable-width font and 1 for a monospaced font.

linespace: Number of pixels of height total. This is the leading of type set solid in the given font.

CTKImage

The CTkImage is not a widget itself, but a container for up to two PIL Image objects for lightand dark mode. There's also a size tuple which describes the width and height of the imageindependent of scaling. Therefore it's important that the PIL Image's are in a higher resolutionthan the given size tuple, so that the image is not blurry if rendered on a 4K monitor with 2xscaling. So that the image is displayed in sharp resolution on a 2x scaled monitor, the givenOIL Image's must have at least double the resolution than the requested size.

Create a CTIImage object:

Create a CTkImage object:

```
from PIL import Image

my_image = customtkinter.CTkImage(light_image=Image.open("<path to light mode image>"),
                                  dark_image=Image.open("<path to dark mode image>"),
                                  size=(30, 30))

button = customtkinter.CTkButton(app, image=my_image)
```

Arguments:

argument	value
light_image	PIL Image object for light mode
dark_image	PIL Image object for dark mode
size	tuple (,) for rendering size independent of scaling

If only `light_image` or only `dark_image` is given, the existing one will be used for both light and dark mode.

Methods:

- **.configure(attribute=value, ...)**

  All attributes can be configured and updated.

- **.cget(attribute_name)**

  Pass attribute name as string and get current value of attribute.