

# Tk リファレンス

`tkinter` アプリケーションのルートウィンドウを管理する `Tk` クラスについての詳細なリファレンスです。

## 概要

`Tk` クラスは、`tkinter` アプリケーションのトップレベルウィンドウを作成するために使用されます。すべての `tkinter` ウィジェットは、このルートウィンドウまたはその子孫の中に配置される必要があります。

## 基本的な使用方法

### ウィンドウの作成と表示

```
import tkinter as tk

# ルートウィンドウの作成
app = tk.Tk()

# ウィンドウのタイトルを設定
app.title("シンプルなウィンドウ")

# ウィンドウのサイズを設定 (幅x高さ)
app.geometry("300x200")

# イベントループを開始
app.mainloop()
```

### クラスベースでの作成

```
import tkinter as tk

class SimpleApp(tk.Tk):
    def __init__(self):
        super().__init__()

        # ウィンドウの設定
        self.title("シンプルなウィンドウ（クラスベース）")
        self.geometry("300x200")

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

## 主要なメソッド

メソッド	説明
<code>title(string)</code>	ウィンドウのタイトルバーに表示されるテキストを設定します。
<code>geometry(geometry_string)</code>	ウィンドウのサイズと位置を設定します。例: "300x200+100+100" (幅300, 高さ200, x座標100, y座標100)
<code>minsize(width, height)</code>	ウィンドウの最小サイズを指定します。
<code>maxsize(width, height)</code>	ウィンドウの最大サイズを指定します。
<code>resizable(width, height)</code>	ウィンドウのサイズ変更の可否をブール値で設定します (水平方向、垂直方向)。
<code>mainloop()</code>	アプリケーションのイベントループを開始します。ユーザーの操作を待ち受け、ウィンドウが表示され続けるようにします。
<code>destroy()</code>	ウィンドウを破棄し、 <code>mainloop</code> を終了します。
<code>update()</code>	ウィンドウの表示を強制的に更新します。

## 実用的な例

### ウィンドウ位置の指定

`geometry` メソッドに "幅x高さ+X座標+Y座標" の形式で文字列を渡すことで、ウィンドウのサイズと表示位置を同時に指定できます。

```

import tkinter as tk

app = tk.Tk()
app.title("位置指定ウィンドウ")

# 幅 400, 高さ 300, 画面の (100, 200) の位置に表示
app.geometry("400x300+100+200")

app.mainloop()

```

## ウィンドウの中央表示

```

import tkinter as tk

app = tk.Tk()
app.title("中央表示ウィンドウ")

window_width = 400
window_height = 300

# 画面の解像度を取得
screen_width = app.winfo_screenwidth()
screen_height = app.winfo_screenheight()

# 中央に表示するための座標を計算
center_x = int(screen_width/2 - window_width / 2)
center_y = int(screen_height/2 - window_height / 2)

app.geometry(f'{window_width}x{window_height}+{center_x}+{center_y}')

app.mainloop()

```

## ウィンドウ終了の確認

```

import tkinter as tk
from tkinter import messagebox

def on_closing():
    if messagebox.askokcancel("終了", "本当に終了しますか？"):
        app.destroy()

app = tk.Tk()
app.title("終了確認")

# ウィンドウの閉じるボタンにカスタム関数をバインド
app.protocol("WM_DELETE_WINDOW", on_closing)

app.mainloop()

```

## クラスベースでの実装

小規模なスクリプトでは手続き的な書き方でも問題ありませんが、より複雑なアプリケーションでは、コードをクラスにまとめることで、構造化され、再利用しやすくなります。

`tk.Tk` や `tk.Frame` を継承してカスタムアプリケーションクラスを作成するのが一般的です。

- **`tk.Tk` を継承するケース:** アプリケーションの主となるウィンドウ（ルートウィンドウ）そのものをクラスとして定義する場合に適しています。シンプルで単一ウィンドウのアプリケーションに多く使われます。この場合、クラスのインスタンスがルートウィンドウ自身となります。
- **`tk.Frame` を継承するケース:** より複雑なUIで、ウィンドウの特定の部分（例えば、ツールバー、ステータスバー、メインコンテンツ領域など）を部品としてカプセル化したい場合に適しています。この方法では、まず `tk.Tk()` でルートウィンドウを作成し、その中に `tk.Frame` を継承したカスタムウィジェットのインスタンスを配置します。これにより、UI部品の再利用性が高まります。

以下は、主ウィンドウとして `tk.Tk` を継承する一般的な例です。

```

import tkinter as tk
from tkinter import messagebox

class App(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("クラスベースのアプリ")
        self.geometry("400x300")

        self.create_widgets()

    # ウィンドウの閉じるボタンの挙動を設定
    self.protocol("WM_DELETE_WINDOW", self.on_closing)

    def create_widgets(self):
        # ラベルの作成
        self.label = tk.Label(self, text="これはクラスベースのtkinterアプリケーションです。")
        self.label.pack(pady=20)

        # ボタンの作成
        self.greet_button = tk.Button(self, text="挨拶", command=self.say_hello)
        self.greet_button.pack()

        # 終了ボタン
        self.quit_button = tk.Button(self, text="終了", command=self.on_closing)
        self.quit_button.pack(pady=10)

    def say_hello(self):
        messagebox.showinfo("挨拶", "こんにちは！")

    def on_closing(self):
        if messagebox.askokcancel("終了確認", "本当にアプリケーションを終了しますか？"):
            self.destroy()

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

```

## ベストプラクティス

プラクティス	説明
インスタンス化	Tk のインスタンスは、アプリケーション全体で一つだけ作成するのが基本です。
ウィジェットの配置	Tk インスタンスを作成した後、他のウィジェットを作成して配置します。
mainloop の呼び出し	mainloop() は、すべてのウィジェットの作成と設定が終わった後、スクリプトの最後に呼び出します。
クラスベースの実装	中規模以上のアプリケーションでは、コードの再利用性とメンテナンス性を高めるために、tk.Tk や tk.Frame を継承したクラスとして実装することが推奨されます。

## 参考リンク

- [Python Docs - tkinter – Python interface to Tcl/Tk](#)
- [TkDocs Tutorial](#)

## Label リファレンス

テキストや画像を表示するための `Label` ウィジェットについての詳細なリファレンスです。

### 概要

`Label` ウィジェットは、ユーザーに情報を表示するための基本的なコントロールです。一行または複数行のテキスト、および画像を表示することができます。表示専用であり、ユーザーからの入力を受け付けることはありません。

### 基本的な使用方法

#### テキストラベルの作成

```
import tkinter as tk

app = tk.Tk()
app.title("Labelの例")
app.geometry("300x200")

# ラベルを作成し、ウィンドウに配置
label = tk.Label(app, text="こんにちは、tkinter!")
label.pack(pady=20) # pack() でウィジェットを配置

app.mainloop()
```

#### クラスベースでのラベル作成

```
import tkinter as tk

class LabelApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Labelの例（クラスベース）")
        self.geometry("300x200")

        self.create_widgets()

    def create_widgets(self):
        # ラベルを作成し、ウィンドウに配置
        self.label = tk.Label(self, text="こんにちは、tkinter!")
        self.label.pack(pady=20)

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

### 主要なオプション

`Label` ウィジェットは、生成時または `config()` メソッドで多くのオプションを設定できます。

- `text`: ラベルに表示するテキスト。
- `textvariable`: `tk.StringVar` などの変数を指定し、その変数の値が変更されるとラベルのテキストも自動的に更新されます。
- `image`: 表示する画像を `tk.PhotoImage` オブジェクトで指定します。
- `font`: フォントを指定します。タプル ("フォント名", サイズ, "スタイル") や文字列で指定できます。
- `fg` (または `foreground`): テキストの色を指定します。
- `bg` (または `background`): ラベルの背景色を指定します。
- `width`: ラベルの幅をテキスト単位で指定します。
- `height`: ラベルの高さをテキスト単位で指定します。
- `padx`, `pady`: ラベル内のテキストと境界線の間の水平・垂直方向の余白。
- `relief`: 境界線のスタイル (`flat`, `raised`, `sunken`, `groove`, `ridge`)。
- `borderwidth` (または `bd`): 境界線の幅。
- `anchor`: ラベル内でテキストを配置する位置 (`n`, `s`, `e`, `w`, `center` など)。
- `justify`: 複数行のテキストの行揃え (`left`, `center`, `right`)。
- `wraplength`: テキストがこの長さを超える場合に自動的に改行されるピクセル単位の幅。

## 実用的な例

### テキストの動的更新

`textvariable` を使用して、ボタンクリックでラベルのテキストを更新する例です。

```
import tkinter as tk
import random

def update_text():
    # StringVarの値を更新すると、ラベルの表示も変わる
    random_number = random.randint(1, 100)
    text_variable.set(f"ランダムな数字: {random_number}")

app = tk.Tk()
app.title("動的ラベル")
app.geometry("300x200")

# StringVarを作成
text_variable = tk.StringVar()
text_variable.set("ボタンを押してください")

label = tk.Label(
    app,
    textvariable=text_variable,
    font=("Helvetica", 14),
    pady=20
)
label.pack()

button = tk.Button(app, text="更新", command=update_text)
button.pack()

app.mainloop()
```

### 画像の表示

`PhotoImage` を使ってラベルに画像を表示します。注意: `PhotoImage` オブジェクトは、参照が失われるとガベージコレクションによって消去されてしまうため、インスタンス変数 (`label.image` など) に保持しておく必要があります。

```
import tkinter as tk

app = tk.Tk()
app.title("画像ラベル")

# このスクリプトと同じディレクトリに 'python_logo.png' があることを想定
# tkinterが標準でサポートしているのはGIFとPGM/PPM形式です。
# PNGやJPEGなどを扱うには、Pillowライブラリ(pip install Pillow)が必要です。
try:
    # Pillowを使ったPNGの読み込み
    from PIL import Image, ImageTk
    img = Image.open("python_logo.png") # Pillowで画像を開く
    photo_image = ImageTk.PhotoImage(img) # tkinterで使える形式に変換
except (ImportError, FileNotFoundError):
    # Pillowがない、またはファイルがない場合の代替
    # (tk.PhotoImageはPNGを直接読めないことが多い)
    # 代わりに標準の画像を使うか、エラーメッセージを表示
    try:
        photo_image = tk.PhotoImage(file="python_logo.gif") # GIFの場合
    except tk.TclError:
        photo_image = None
        print("画像ファイルが見つからなかったり、非対応の形式です。")

if photo_image:
    label = tk.Label(app, image=photo_image)
    label.image = photo_image # 参照を保持
    label.pack(pady=20)
else:
    label = tk.Label(app, text="画像を表示できませんでした")
    label.pack(pady=20)

app.mainloop()
```

### 参考リンク

- [Python Docs - `tkinter.Label`](#)
- [TkDocs - `Label`](#)

## Button リファレンス

ユーザーがクリックして操作を実行するための `Button` ウィジェットについての詳細なリファレンスです。

### 概要

`Button` ウィジェットは、ユーザーがクリックすることで特定の処理を実行するための基本的なコントロールです。テキストや画像を表示でき、マウスクリックやキーボード操作に応答します。

### 基本的な使用方法

#### シンプルなボタンの作成

```
import tkinter as tk

def button_clicked():
    print("ボタンがクリックされました！")

app = tk.Tk()
app.title("Buttonの例")
app.geometry("300x200")

button = tk.Button(app, text="クリックしてください", command=button_clicked)
button.pack(pady=20)

app.mainloop()
```

#### クラスベースでのボタン作成

```
import tkinter as tk

class ButtonApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Buttonの例（クラスベース）")
        self.geometry("300x200")

        self.create_widgets()

    def create_widgets(self):
        self.button = tk.Button(self, text="クリックしてください", command=self.button_clicked)
        self.button.pack(pady=20)

    def button_clicked(self):
        print("ボタンがクリックされました！")

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

### 主要なオプション

オプション	説明
<code>text</code>	ボタンに表示するテキスト。
<code>command</code>	ボタンがクリックされたときに実行される関数。
<code>image</code>	ボタンに表示する画像を <code>tk.PhotoImage</code> オブジェクトで指定。
<code>compound</code>	テキストと画像の配置方法 ( <code>top</code> , <code>bottom</code> , <code>left</code> , <code>right</code> , <code>center</code> )。
<code>font</code>	フォントを指定。タプル ("フォント名", サイズ, "スタイル") や文字列で指定。
<code>fg</code> (または <code>foreground</code> )	テキストの色。
<code>bg</code> (または <code>background</code> )	ボタンの背景色。
<code>activeforeground</code>	ボタンがアクティブ時のテキスト色。
<code>activebackground</code>	ボタンがアクティブ時の背景色。
<code>width</code>	ボタンの幅をテキスト単位で指定。
<code>height</code>	ボタンの高さをテキスト単位で指定。
<code>padx</code> , <code>pady</code>	ボタン内のテキストと境界線の間の水平・垂直方向の余白。
<code>relief</code>	境界線のスタイル ( <code>flat</code> , <code>raised</code> , <code>sunken</code> , <code>groove</code> , <code>ridge</code> )。

オプション	説明
<code>borderwidth</code> (または <code>bd</code> )	境界線の幅。
<code>state</code>	ボタンの状態 ( <code>normal</code> , <code>active</code> , <code>disabled</code> )。

## 実用的な例

### 複数のボタンを持つアプリケーション

```

import tkinter as tk
from tkinter import messagebox

class ButtonApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("ボタンアプリケーション")
        self.geometry("400x300")

        self.counter = 0

        self.create_widgets()

    def create_widgets(self):
        # カウンター表示用ラベル
        self.counter_label = tk.Label(self, text=f"カウント: {self.counter}", font=("Arial", 14))
        self.counter_label.pack(pady=10)

        # カウンターを増やすボタン
        self.increment_button = tk.Button(
            self,
            text="カウント+1",
            command=self.increment_counter,
            bg="lightblue",
            font=("Arial", 12)
        )
        self.increment_button.pack(pady=5)

        # カウンターをリセットするボタン
        self.reset_button = tk.Button(
            self,
            text="リセット",
            command=self.reset_counter,
            bg="orange",
            font=("Arial", 12)
        )
        self.reset_button.pack(pady=5)

        # 確認ダイアログを表示するボタン
        self.dialog_button = tk.Button(
            self,
            text="確認ダイアログ",
            command=self.show_dialog,
            bg="lightgreen",
            font=("Arial", 12)
        )
        self.dialog_button.pack(pady=5)

        # 無効化/有効化ボタン
        self.toggle_button = tk.Button(
            self,
            text="ボタンを無効化",
            command=self.toggle_buttons,
            bg="lightcoral",
            font=("Arial", 12)
        )
        self.toggle_button.pack(pady=5)

    def increment_counter(self):
        self.counter += 1
        self.counter_label.config(text=f"カウント: {self.counter}")

    def reset_counter(self):
        self.counter = 0
        self.counter_label.config(text=f"カウント: {self.counter}")

    def show_dialog(self):
        messagebox.showinfo("情報", f"現在のカウント: {self.counter}")

    def toggle_buttons(self):
        current_state = self.increment_button['state']
        if current_state == 'normal':
            # ボタンを無効化
            self.increment_button.config(state='disabled')
            self.reset_button.config(state='disabled')
            self.dialog_button.config(state='disabled')
            self.toggle_button.config(text="ボタンを有効化")
        else:
            # ボタンを有効化
            self.increment_button.config(state='normal')
            self.reset_button.config(state='normal')
            self.dialog_button.config(state='normal')
            self.toggle_button.config(text="ボタンを無効化")

if __name__ == "__main__":

```

```
app = ButtonApp()
app.mainloop()
```

## 画像付きボタン

```
import tkinter as tk
from tkinter import messagebox

def image_button_clicked():
    messagebox.showinfo("画像ボタン", "画像付きボタンがクリックされました!")

app = tk.Tk()
app.title("画像ボタンの例")
app.geometry("300x200")

try:
    # 注意: この例ではPillowライブラリが必要です (pip install Pillow)
    from PIL import Image, ImageTk

    # 画像ファイルを読み込み (存在しない場合のエラーハンドリング付き)
    try:
        image = Image.open("icon.png")
        image = image.resize((32, 32)) # サイズ調整
        photo = ImageTk.PhotoImage(image)

        # 画像付きボタン
        image_button = tk.Button(
            app,
            text="画像ボタン",
            image=photo,
            compound=tk.LEFT, # 画像をテキストの左に配置
            command=image_button_clicked
        )
        image_button.image = photo # 参照を保持
        image_button.pack(pady=20)

    except FileNotFoundError:
        # 画像ファイルが見つからない場合
        fallback_button = tk.Button(
            app,
            text="画像なしボタン",
            command=image_button_clicked
        )
        fallback_button.pack(pady=20)

    except ImportError:
        # Pillowがインストールされていない場合
        fallback_button = tk.Button(
            app,
            text="テキストのみボタン",
            command=image_button_clicked
        )
        fallback_button.pack(pady=20)

    app.mainloop()
```

## ベストプラクティス

プラクティス	説明
適切な <code>command</code> 関数	ボタンの <code>command</code> には引数を取らない関数を指定します。引数が必要な場合は <code>lambda</code> を使用するか、 <code>functools.partial</code> を利用します。
状態管理	ボタンの <code>state</code> オプション ( <code>normal</code> , <code>disabled</code> ) を適切に使用して、ユーザーの操作を制御します。
視覚的フィードバック	色やフォントを使用してボタンの用途を明確にし、ユーザビリティを向上させます。
画像の参照保持	画像を使用する場合は、 <code>PhotoImage</code> オブジェクトの参照をボタンのインスタンス変数に保持してガベージコレクションを防ぎます。

## 参考リンク

- [Python Docs - `tkinter.Button`](#)
- [TkDocs - `Button`](#)

# Entry リファレンス

一行のテキスト入力を受け付ける `Entry` ウィジェットについての詳細なリファレンスです。

## 概要

`Entry` ウィジェットは、ユーザーが一行のテキストを入力・編集するための基本的なコントロールです。文字列の入力、数値の入力、パスワードの入力などに使用されます。

## 基本的な使用方法

### シンプルなテキスト入力

```
import tkinter as tk

def get_text():
    text = entry.get()
    print(f"入力されたテキスト: {text}")

app = tk.Tk()
app.title("Entryの例")
app.geometry("400x200")

entry = tk.Entry(app, width=30)
entry.pack(pady=20)

button = tk.Button(app, text="テキストを取得", command=get_text)
button.pack()

app.mainloop()
```

### クラスベースでのテキスト入力

```
import tkinter as tk

class EntryApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Entryの例（クラスベース）")
        self.geometry("400x200")

        self.create_widgets()

    def create_widgets(self):
        self.entry = tk.Entry(self, width=30)
        self.entry.pack(pady=20)

        self.button = tk.Button(self, text="テキストを取得", command=self.get_text)
        self.button.pack()

    def get_text(self):
        text = self.entry.get()
        print(f"入力されたテキスト: {text}")

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

## 主要なオプション

オプション	説明
<code>textvariable</code>	<code>tk.StringVar</code> などの変数を指定し、その変数と Entry の内容を同期します。
<code>width</code>	Entry の幅を文字数で指定します。
<code>font</code>	フォントを指定。タプル（"フォント名", サイズ, "スタイル"）や文字列で指定。
<code>fg</code> (または <code>foreground</code> )	テキストの色。
<code>bg</code> (または <code>background</code> )	Entry の背景色。
<code>relief</code>	境界線のスタイル ( <code>flat</code> , <code>raised</code> , <code>sunken</code> , <code>groove</code> , <code>ridge</code> )。
<code>borderwidth</code> (または <code>bd</code> )	境界線の幅。
<code>state</code>	Entry の状態 ( <code>normal</code> , <code>readonly</code> , <code>disabled</code> )。
<code>show</code>	入力文字を隠す文字を指定（パスワード入力など）。例: <code>show="*"</code>

オプション	説明
<code>justify</code>	テキストの配置 ( <code>left</code> , <code>center</code> , <code>right</code> )。
<code>validate</code>	入力値の検証タイミング ( <code>none</code> , <code>focus</code> , <code>focusin</code> , <code>focusout</code> , <code>key</code> , <code>all</code> )。
<code>validatecommand</code>	検証時に実行される関数。

## 主要なメソッド

メソッド	説明
<code>get()</code>	Entry の現在のテキストを取得します。
<code>set(value)</code>	Entry のテキストを設定します。( <code>textvariable</code> 使用時)
<code>insert(index, string)</code>	指定位置にテキストを挿入します。
<code>delete(first, last=None)</code>	指定範囲のテキストを削除します。
<code>select_range(start, end)</code>	指定範囲のテキストを選択します。
<code>select_clear()</code>	テキストの選択を解除します。
<code>icursor(index)</code>	カーソルを指定位置に移動します。

## 実用的な例

### フォームアプリケーション

```

import tkinter as tk
from tkinter import messagebox

class FormApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("フォーム入力")
        self.geometry("400x300")

        self.create_widgets()

    def create_widgets(self):
        # 名前入力
        tk.Label(self, text="名前:", font=("Arial", 12)).pack(pady=(20, 5))
        self.name_var = tk.StringVar()
        self.name_entry = tk.Entry(self, textvariable=self.name_var, width=30, font=("Arial", 11))
        self.name_entry.pack()

        # 年齢入力
        tk.Label(self, text="年齢:", font=("Arial", 12)).pack(pady=(10, 5))
        self.age_var = tk.StringVar()
        self.age_entry = tk.Entry(self, textvariable=self.age_var, width=30, font=("Arial", 11))
        self.age_entry.pack()

        # パスワード入力
        tk.Label(self, text="パスワード:", font=("Arial", 12)).pack(pady=(10, 5))
        self.password_var = tk.StringVar()
        self.password_entry = tk.Entry(
            self,
            textvariable=self.password_var,
            width=30,
            font=("Arial", 11),
            show="*" # パスワードを隠す
        )
        self.password_entry.pack()

        # ボタン
        button_frame = tk.Frame(self)
        button_frame.pack(pady=20)

        submit_button = tk.Button(button_frame, text="送信", command=self.submit_form)
        submit_button.pack(side=tk.LEFT, padx=5)

        clear_button = tk.Button(button_frame, text="クリア", command=self.clear_form)
        clear_button.pack(side=tk.LEFT, padx=5)

        # 初期フォーカスを名前入力に設定
        self.name_entry.focus()

    def submit_form(self):
        name = self.name_var.get().strip()
        age = self.age_var.get().strip()
        password = self.password_var.get()

        if not name:
            messagebox.showerror("エラー", "名前を入力してください。")
            self.name_entry.focus()
            return

        if not age:
            messagebox.showerror("エラー", "年齢を入力してください。")
            self.age_entry.focus()
            return

        ...

```

```

try:
    age_int = int(age)
    if age_int < 0 or age_int > 150:
        raise ValueError
except ValueError:
    messagebox.showerror("エラー", "有効な年齢を入力してください。")
    self.age_entry.focus()
    return

if not password:
    messagebox.showerror("エラー", "パスワードを入力してください。")
    self.password_entry.focus()
    return

messagebox.showinfo("送信完了", f"名前: {name}\n年齢: {age}\nパスワードは設定されました。")

def clear_form(self):
    self.name_var.set("")
    self.age_var.set("")
    self.password_var.set("")
    self.name_entry.focus()

if __name__ == "__main__":
    app = FormApp()
    app.mainloop()

```

## 入力値のリアルタイム検証

```

import tkinter as tk

class ValidatedEntryApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("入力値検証")
        self.geometry("400x300")

        # 検証関数を登録
        self.validate_number = self.register(self.validate_number_input)

        self.create_widgets()

    def validate_number_input(self, value):
        """数値のみを許可する検証関数"""
        if value == "":
            return True # 空文字は許可
        try:
            float(value)
            return True
        except ValueError:
            return False

    def create_widgets(self):
        # 通常の Entry
        tk.Label(self, text="自由入力:", font=("Arial", 12)).pack(pady=(20, 5))
        self.free_entry = tk.Entry(self, width=30, font=("Arial", 11))
        self.free_entry.pack()

        # 数値のみの Entry
        tk.Label(self, text="数値のみ:", font=("Arial", 12)).pack(pady=(10, 5))
        self.number_entry = tk.Entry(
            self,
            width=30,
            font=("Arial", 11),
            validate='key', # キー入力時に検証
            validatecommand=(self.validate_number, '%P') # %P は新しい値
        )
        self.number_entry.pack()

        # 読み取り専用の Entry
        tk.Label(self, text="読み取り専用:", font=("Arial", 12)).pack(pady=(10, 5))
        self.readonly_var = tk.StringVar(value="変更できません")
        self.readonly_entry = tk.Entry(
            self,
            textvariable=self.readonly_var,
            width=30,
            font=("Arial", 11),
            state='readonly',
            bg='lightgray'
        )
        self.readonly_entry.pack()

        # 結果表示
        tk.Label(self, text="結果:", font=("Arial", 12)).pack(pady=(20, 5))
        self.result_text = tk.Text(self, width=40, height=6, font=("Arial", 10))
        self.result_text.pack()

        # 値を取得するボタン
        get_button = tk.Button(self, text="値を取得", command=self.get_values)
        get_button.pack(pady=10)

    def get_values(self):
        free_value = self.free_entry.get()
        number_value = self.number_entry.get()
        readonly_value = self.readonly_var.get()

        result = f"自由入力: {free_value}\n"
        result += f"数値入力: {number_value}\n"

```

```

result += f"読み取り専用: {readonly_value}\n"

self.result_text.delete(1.0, tk.END)
self.result_text.insert(1.0, result)

if __name__ == "__main__":
    app = ValidatedEntryApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
<code>textvariable</code> の活用	<code>tk.StringVar</code> を使用することで、Entry の値とアプリケーションの状態を簡単に同期できます。
入力値の検証	<code>validate</code> と <code>validatecommand</code> オプションを使用して、不正な入力を防ぎます。
フォーカス管理	<code>focus()</code> メソッドを使用して、適切なフィールドにフォーカスを設定します。
エラーハンドリング	ユーザー入力を処理する際は、適切なエラーハンドリングとユーザーフィードバックを提供します。
パスワード入力	機密情報の入力には <code>show</code> オプションを使用して文字を隠します。

## 参考リンク

- [Python Docs - `tkinter.Entry`](#)
- [TkDocs - Entry](#)

# Text リファレンス

複数行のテキスト入力・編集を行う `Text` ウィジェットについての詳細なリファレンスです。

## 概要

`Text` ウィジェットは、複数行のテキストの表示や編集を行うための強力なコントロールです。単純なテキストエディタから複雑な文書処理まで、様々な用途に使用できます。文字の装飾、検索機能、スクロール機能なども提供します。

## 基本的な使用方法

### シンプルなテキストエディタ

```
import tkinter as tk

def get_text():
    content = text_widget.get("1.0", tk.END)
    print(f"テキストの内容:\n{content}")

app = tk.Tk()
app.title("Textの例")
app.geometry("500x400")

text_widget = tk.Text(app, width=60, height=15)
text_widget.pack(pady=20)

button = tk.Button(app, text="テキストを取得", command=get_text)
button.pack()

app.mainloop()
```

### クラスベースでのテキストエディタ

```
import tkinter as tk

class TextApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Textの例（クラスベース）")
        self.geometry("500x400")

        self.create_widgets()

    def create_widgets(self):
        self.text_widget = tk.Text(self, width=60, height=15)
        self.text_widget.pack(pady=20)

        self.button = tk.Button(self, text="テキストを取得", command=self.get_text)
        self.button.pack()

    def get_text(self):
        content = self.text_widget.get("1.0", tk.END)
        print(f"テキストの内容:\n{content}")

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

## 主要なオプション

オプション	説明
<code>width</code>	Text ウィジェットの幅を文字数で指定します。
<code>height</code>	Text ウィジェットの高さを行数で指定します。
<code>font</code>	フォントを指定。タプル ("フォント名", サイズ, "スタイル") や文字列で指定。
<code>fg</code> (または <code>foreground</code> )	テキストの色。
<code>bg</code> (または <code>background</code> )	Text ウィジェットの背景色。
<code>wrap</code>	行の折り返し方法 ( <code>none</code> , <code>char</code> , <code>word</code> )。
<code>state</code>	Text ウィジェットの状態 ( <code>normal</code> , <code>disabled</code> )。
<code>relief</code>	境界線のスタイル ( <code>flat</code> , <code>raised</code> , <code>sunken</code> , <code>groove</code> , <code>ridge</code> )。
<code>borderwidth</code> (または <code>bd</code> )	境界線の幅。

オプション	説明
<code>selectbackground</code>	選択されたテキストの背景色。
<code>selectforeground</code>	選択されたテキストの前景色。
<code>insertbackground</code>	カーソルの色。
<code>undo</code>	アンドウ機能を有効にする ( <code>True</code> / <code>False</code> )。

## 主要なメソッド

メソッド	説明
<code>get(start, end=None)</code>	指定範囲のテキストを取得します。
<code>insert(index, text)</code>	指定位置にテキストを挿入します。
<code>delete(start, end=None)</code>	指定範囲のテキストを削除します。
<code>replace(start, end, text)</code>	指定範囲のテキストを置換します。
<code>see(index)</code>	指定位置が見えるようにスクロールします。
<code>search(pattern, start, end=None)</code>	パターンでテキストを検索します。
<code>mark_set(name, index)</code>	指定位置にマークを設定します。
<code>mark_unset(name)</code>	マークを削除します。
<code>tag_add(tag, start, end=None)</code>	指定範囲にタグを追加します。
<code>tag_delete(tag)</code>	タグを削除します。

## インデックスの指定方法

Text ウィジェットでは、テキストの位置を "行.列" の形式で指定します。

インデックス	説明
"1.0"	1行目の0列目（行の先頭）。
"2.5"	2行目の5列目。
<code>tk.END</code>	テキストの最後。
<code>tk.INSERT</code>	現在のカーソル位置。
<code>tk.SEL_FIRST</code>	選択範囲の開始位置。
<code>tk.SEL_LAST</code>	選択範囲の終了位置。

## 実用的な例

### スクロールバー付きテキストエディタ

```

import tkinter as tk
from tkinter import scrolledtext, messagebox, filedialog

class TextEditor(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("テキストエディタ")
        self.geometry("700x500")

        self.filename = None
        self.create_widgets()
        self.create_menu()

    def create_widgets(self):
        # スクロールバー付きテキストウィジェット
        self.text_area = scrolledtext.ScrolledText(
            self,
            wrap=tk.WORD,
            width=80,
            height=25,
            font=("Consolas", 11),
            undo=True # アンドウ機能を有効化
        )
        self.text_area.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # ステータスバー
        self.status_bar = tk.Label(
            self,
            text="準備完了",
            anchor=tk.W,
            relief=tk.SUNKEN,
            bg="lightgray"
        )
        self.status_bar.pack(side=tk.BOTTOM, fill=tk.X)

```

```

# カーソル位置の更新をバインド
self.text_area.bind('<KeyRelease>', self.update_cursor_position)
self.text_area.bind('<Button-1>', self.update_cursor_position)

def create_menu(self):
    menubar = tk.Menu(self)
    self.config(menu=menubar)

    # ファイルメニュー
    file_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="ファイル", menu=file_menu)
    file_menu.add_command(label="新規", command=self.new_file)
    file_menu.add_command(label="開く", command=self.open_file)
    file_menu.add_command(label="保存", command=self.save_file)
    file_menu.add_command(label="名前を付けて保存", command=self.save_as_file)
    file_menu.add_separator()
    file_menu.add_command(label="終了", command=self.quit)

    # 編集メニュー
    edit_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="編集", menu=edit_menu)
    edit_menu.add_command(label="元に戻す", command=lambda: self.text_area.edit_undo())
    edit_menu.add_command(label="やり直し", command=lambda: self.text_area.edit_redo())
    edit_menu.add_separator()
    edit_menu.add_command(label="切り取り", command=lambda: self.text_area.event_generate("<>"))
    edit_menu.add_command(label="コピー", command=lambda: self.text_area.event_generate("<>"))
    edit_menu.add_command(label="貼り付け", command=lambda: self.text_area.event_generate("<>"))
    edit_menu.add_command(label="すべて選択", command=lambda: self.text_area.tag_add(tk.SEL, "1.0", tk.END))

def new_file(self):
    if messagebox.askokcancel("新規ファイル", "現在の内容は失われます。続行しますか?"):
        self.text_area.delete("1.0", tk.END)
        self.filename = None
        self.title("テキストエディタ - 新規ファイル")

def open_file():
    filename = filedialog.askopenfilename(
        title="ファイルを開く",
        filetypes=[("テキストファイル", "*.txt"), ("すべてのファイル", "*.*")]
    )
    if filename:
        try:
            with open(filename, 'r', encoding='utf-8') as file:
                content = file.read()
            self.text_area.delete("1.0", tk.END)
            self.text_area.insert("1.0", content)
            self.filename = filename
            self.title(f"テキストエディタ - {filename}")
        except Exception as e:
            messagebox.showerror("エラー", f"ファイルを開けませんでした:\n{e}")

def save_file():
    if self.filename:
        try:
            content = self.text_area.get("1.0", tk.END)
            with open(self.filename, 'w', encoding='utf-8') as file:
                file.write(content)
            messagebox.showinfo("保存完了", "ファイルが保存されました。")
        except Exception as e:
            messagebox.showerror("エラー", f"ファイルを保存できませんでした:\n{e}")
    else:
        self.save_as_file()

def save_as_file():
    filename = filedialog.asksaveasfilename(
        title="名前を付けて保存",
        defaultextension=".txt",
        filetypes=[("テキストファイル", "*.txt"), ("すべてのファイル", "*.*")]
    )
    if filename:
        try:
            content = self.text_area.get("1.0", tk.END)
            with open(filename, 'w', encoding='utf-8') as file:
                file.write(content)
            self.filename = filename
            self.title(f"テキストエディタ - {filename}")
            messagebox.showinfo("保存完了", "ファイルが保存されました。")
        except Exception as e:
            messagebox.showerror("エラー", f"ファイルを保存できませんでした:\n{e}")

def update_cursor_position(self, event=None):
    cursor_position = self.text_area.index(tk.INSERT)
    line, column = cursor_position.split('.')
    self.status_bar.config(text=f"行: {line}, 列: {column}")

if __name__ == "__main__":
    app = TextEditor()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
スクロールバーの追加	長いテキストを扱う場合は、 <code>tkinter.scrolledtext.ScrolledText</code> を使用するか、独自にスクロールバーを追加します。

プラクティス	説明
インデックスの理解	Text ウィジェットのインデックス形式 "行.列" を正しく理解して使用します。
タグの活用	テキストの装飾や特別な動作には、タグ機能を活用します。
アンドゥ機能	ユーザビリティ向上のため、 <code>undo=True</code> オプションを設定してアンドゥ機能を有効にします。
イベントバインディング	キーボードやマウスイベントをバインドして、リアルタイムな操作を実現します。

## 参考リンク

- [Python Docs - tkinter.Text](#)
- [TkDocs - Text](#)

# Frame リファレンス

ウィジェットをグループ化し、レイアウトを整理するための `Frame` ウィジェットについての詳細なリファレンスです。

## 概要

`Frame` ウィジェットは、他のウィジェットを格納するためのコンテナです。ウィジェットをグループ化して整理したり、複雑なレイアウトを構築したりするために使用されます。Frame 自体は視覚的な要素を持ちませんが、境界線や背景色を設定して表示することも可能です。

## 基本的な使用方法

### シンプルなフレームの作成

```
import tkinter as tk

app = tk.Tk()
app.title("Frameの例")
app.geometry("400x300")

# メインフレーム
main_frame = tk.Frame(app, bg="lightblue", relief=tk.RAISED, bd=2)
main_frame.pack(padx=20, pady=20, fill=tk.BOTH, expand=True)

# フレーム内にウィジェットを配置
label = tk.Label(main_frame, text="フレーム内のラベル", bg="lightblue")
label.pack(pady=10)

button = tk.Button(main_frame, text="フレーム内のボタン")
button.pack(pady=5)

app.mainloop()
```

### クラスベースでのフレーム作成

```
import tkinter as tk

class FrameApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Frameの例（クラスベース）")
        self.geometry("400x300")

        self.create_widgets()

    def create_widgets(self):
        # メインフレーム
        self.main_frame = tk.Frame(self, bg="lightblue", relief=tk.RAISED, bd=2)
        self.main_frame.pack(padx=20, pady=20, fill=tk.BOTH, expand=True)

        # フレーム内にウィジェットを配置
        self.label = tk.Label(self.main_frame, text="フレーム内のラベル", bg="lightblue")
        self.label.pack(pady=10)

        self.button = tk.Button(self.main_frame, text="フレーム内のボタン")
        self.button.pack(pady=5)

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

## 主要なオプション

オプション	説明
<code>width</code>	フレームの幅をピクセル単位で指定します。
<code>height</code>	フレームの高さをピクセル単位で指定します。
<code>bg</code> (または <code>background</code> )	フレームの背景色。
<code>relief</code>	境界線のスタイル ( <code>flat</code> , <code>raised</code> , <code>sunken</code> , <code>groove</code> , <code>ridge</code> )。
<code>borderwidth</code> (または <code>bd</code> )	境界線の幅。
<code>padx</code> , <code>pady</code>	フレーム内のコンテンツと境界線の間の水平・垂直方向の余白。
<code>cursor</code>	フレーム上でのマウスカーソルの形状。

## レイアウトマネージャーとの組み合わせ

Frame は `pack()`, `grid()`, `place()` のすべてのレイアウトマネージャーと組み合わせて使用できます。

マネージャー	説明
<code>pack()</code>	上下左右に順次配置。シンプルなレイアウトに適している。
<code>grid()</code>	格子状の配置。表形式のレイアウトに適している。
<code>place()</code>	絶対座標での配置。精密な位置制御が可能。

## 実用的な例

### 複数のセクションを持つアプリケーション

```
import tkinter as tk
from tkinter import ttk

class SectionedApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("セクション分けアプリケーション")
        self.geometry("600x500")

        self.create_widgets()

    def create_widgets(self):
        # ヘッダーフレーム
        header_frame = tk.Frame(self, bg="darkblue", height=60)
        header_frame.pack(fill=tk.X)
        header_frame.pack_propagate(False) # 固定サイズを維持

        title_label = tk.Label(
            header_frame,
            text="アプリケーションタイトル",
            bg="darkblue",
            fg="white",
            font=("Arial", 16, "bold")
        )
        title_label.pack(expand=True)

        # メインコンテンツエリア
        content_frame = tk.Frame(self)
        content_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # 左サイドバー
        sidebar_frame = tk.Frame(content_frame, bg="lightgray", width=200, relief=tk.SUNKEN, bd=1)
        sidebar_frame.pack(side=tk.LEFT, fill=tk.Y, padx=(0, 10))
        sidebar_frame.pack_propagate(False)

        tk.Label(sidebar_frame, text="サイドバー", bg="lightgray", font=("Arial", 12, "bold")).pack(pady=10)

        # サイドバーのボタン
        for i in range(5):
            btn = tk.Button(sidebar_frame, text=f"メニュー {i+1}", command=lambda x=i: self.menu_clicked(x+1))
            btn.pack(fill=tk.X, padx=10, pady=2)

        # メインコンテンツエリア
        main_content_frame = tk.Frame(content_frame, bg="white", relief=tk.SUNKEN, bd=1)
        main_content_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        # メインコンテンツ
        self.content_label = tk.Label(
            main_content_frame,
            text="メインコンテンツエリア\nメニューを選択してください",
            bg="white",
            font=("Arial", 14),
            justify=tk.CENTER
        )
        self.content_label.pack(expand=True)

        # フッターフレーム
        footer_frame = tk.Frame(self, bg="gray", height=40)
        footer_frame.pack(fill=tk.X)
        footer_frame.pack_propagate(False)

        status_label = tk.Label(
            footer_frame,
            text="ステータス: 準備完了",
            bg="gray",
            fg="white"
        )
        status_label.pack(side=tk.LEFT, padx=10, expand=True, anchor=tk.W)

    def menu_clicked(self, menu_num):
        self.content_label.config(text=f"メニュー {menu_num} が選択されました\n\nここにコンテンツが表示されます")

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

## フォームレイアウト (grid使用)

```
import tkinter as tk
from tkinter import messagebox

class FormLayout(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("フォームレイアウト")
        self.geometry("500x400")

        self.create_widgets()

    def create_widgets(self):
        # メインフレーム
        main_frame = tk.Frame(self, padx=20, pady=20)
        main_frame.pack(fill=tk.BOTH, expand=True)

        # タイトル
        title_label = tk.Label(main_frame, text="ユーザー情報入力", font=("Arial", 16, "bold"))
        title_label.grid(row=0, column=0, columnspan=2, pady=(0, 20))

        # 個人情報セクション
        personal_frame = tk.LabelFrame(main_frame, text="個人情報", font=("Arial", 12, "bold"), padx=10, pady=10)
        personal_frame.grid(row=1, column=0, columnspan=2, sticky="ew", pady=(0, 10))

        # 個人情報フィールド
        tk.Label(personal_frame, text="氏名:").grid(row=0, column=0, sticky="e", padx=(0, 10), pady=5)
        self.name_entry = tk.Entry(personal_frame, width=30)
        self.name_entry.grid(row=0, column=1, pady=5)

        tk.Label(personal_frame, text="年齢:").grid(row=1, column=0, sticky="e", padx=(0, 10), pady=5)
        self.age_entry = tk.Entry(personal_frame, width=30)
        self.age_entry.grid(row=1, column=1, pady=5)

        tk.Label(personal_frame, text="メールアドレス:").grid(row=2, column=0, sticky="e", padx=(0, 10), pady=5)
        self.email_entry = tk.Entry(personal_frame, width=30)
        self.email_entry.grid(row=2, column=1, pady=5)

        # 連絡先セクション
        contact_frame = tk.LabelFrame(main_frame, text="連絡先", font=("Arial", 12, "bold"), padx=10, pady=10)
        contact_frame.grid(row=2, column=0, columnspan=2, sticky="ew", pady=(0, 10))

        # 連絡先フィールド
        tk.Label(contact_frame, text="電話番号:").grid(row=0, column=0, sticky="e", padx=(0, 10), pady=5)
        self.phone_entry = tk.Entry(contact_frame, width=30)
        self.phone_entry.grid(row=0, column=1, pady=5)

        tk.Label(contact_frame, text="住所:").grid(row=1, column=0, sticky="ne", padx=(0, 10), pady=5)
        self.address_text = tk.Text(contact_frame, width=30, height=3)
        self.address_text.grid(row=1, column=1, pady=5)

        # 設定セクション
        settings_frame = tk.LabelFrame(main_frame, text="設定", font=("Arial", 12, "bold"), padx=10, pady=10)
        settings_frame.grid(row=3, column=0, columnspan=2, sticky="ew", pady=(0, 20))

        # チェックボックス
        self.newsletter_var = tk.BooleanVar()
        newsletter_check = tk.Checkbutton(settings_frame, text="ニュースレターを受け取る", variable=self.newsletter_var)
        newsletter_check.grid(row=0, column=0, sticky="w", pady=2)

        self.notifications_var = tk.BooleanVar()
        notifications_check = tk.Checkbutton(settings_frame, text="通知を受け取る", variable=self.notifications_var)
        notifications_check.grid(row=1, column=0, sticky="w", pady=2)

        # ボタンフレーム
        button_frame = tk.Frame(main_frame)
        button_frame.grid(row=4, column=0, columnspan=2)

        submit_button = tk.Button(button_frame, text="送信", command=self.submit_form, bg="lightblue")
        submit_button.pack(side=tk.LEFT, padx=5)

        clear_button = tk.Button(button_frame, text="クリア", command=self.clear_form, bg="lightcoral")
        clear_button.pack(side=tk.LEFT, padx=5)

        # グリッドの重みを設定
        main_frame.columnconfigure(1, weight=1)

    def submit_form(self):
        name = self.name_entry.get()
        age = self.age_entry.get()
        email = self.email_entry.get()
        phone = self.phone_entry.get()
        address = self.address_text.get("1.0", tk.END).strip()
        newsletter = self.newsletter_var.get()
        notifications = self.notifications_var.get()

        if not name:
            messagebox.showerror("エラー", "氏名を入力してください")
            return

        result = f"氏名: {name}\n年齢: {age}\nメール: {email}\n電話: {phone}\n住所: {address}\n"
        result += f"ニュースレター: {'はい' if newsletter else 'いいえ'}\n"
        result += f"通知: {'はい' if notifications else 'いいえ'}"

        messagebox.showinfo("送信完了", result)
```

```

def clear_form(self):
    self.name_entry.delete(0, tk.END)
    self.age_entry.delete(0, tk.END)
    self.email_entry.delete(0, tk.END)
    self.phone_entry.delete(0, tk.END)
    self.address_text.delete("1.0", tk.END)
    self.newsletter_var.set(False)
    self.notifications_var.set(False)

if __name__ == "__main__":
    app = FormLayout()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
論理的なグループ化	関連するウィジエットをFrameでグループ化して、UIの構造を明確にします。
LabelFrameの活用	タイトル付きのセクションには LabelFrameを使用します。
pack_propagate() の理解	フレームの固定サイズを維持したい場合は pack_propagate(False)を使用します。
レイアウトマネージャーの統一	一つの親ウィジエット内では同一のレイアウトマネージャー (pack、grid、place) を使用します。
入れ子構造の活用	複雑なレイアウトは Frameを入れ子にして構築します。

## 参考リンク

- [Python Docs - tkinter.Frame](#)
- [TkDocs - Frame](#)

## Checkbutton リファレンス

オン/オフの選択を行う `Checkbutton` ウィジェットについての詳細なリファレンスです。

### 概要

`Checkbutton` ウィジェットは、ユーザーがオン（チェック）またはオフ（チェック解除）の状態を選択できるコントロールです。複数の選択肢から複数項目を選択可能な場合に使用されます。各チェックボタンは独立して操作でき、他のチェックボタンの状態に影響されません。

### 基本的な使用方法

#### シンプルなチェックボタン

```
import tkinter as tk

def show_state():
    state = check_var.get()
    print(f"チェック状態: {'オン' if state else 'オフ'}")

app = tk.Tk()
app.title("Checkbuttonの例")
app.geometry("300x200")

# BooleanVar でチェック状態を管理
check_var = tk.BooleanVar()

checkbutton = tk.Checkbutton(
    app,
    text="同意する",
    variable=check_var,
    command=show_state
)
checkbutton.pack(pady=20)

button = tk.Button(app, text="状態を確認", command=show_state)
button.pack()

app.mainloop()
```

#### クラスベースでのチェックボタン

```
import tkinter as tk

class CheckbuttonApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Checkbuttonの例（クラスベース）")
        self.geometry("300x200")

        self.create_widgets()

    def create_widgets(self):
        # BooleanVar でチェック状態を管理
        self.check_var = tk.BooleanVar()

        self.checkbutton = tk.Checkbutton(
            self,
            text="同意する",
            variable=self.check_var,
            command=self.show_state
        )
        self.checkbutton.pack(pady=20)

        self.button = tk.Button(self, text="状態を確認", command=self.show_state)
        self.button.pack()

    def show_state(self):
        state = self.check_var.get()
        print(f"チェック状態: {'オン' if state else 'オフ'}")

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

## 主要なオプション

オプション	説明
<code>text</code>	チェックボタンに表示するテキスト。
<code>variable</code>	チェック状態を管理する変数 ( <code>tk.BooleanVar</code> , <code>tk.IntVar</code> , <code>tk.StringVar</code> )。
<code>command</code>	チェック状態が変更されたときに実行される関数。
<code>onvalue</code>	チェックされたときの変数の値 (デフォルト: 1)。
<code>offvalue</code>	チェックが外されたときの変数の値 (デフォルト: 0)。
<code>font</code>	フォントを指定。タブル ("フォント名", サイズ, "スタイル") や文字列で指定。
<code>fg (または foreground)</code>	テキストの色。
<code>bg (または background)</code>	チェックボタンの背景色。
<code>activeforeground</code>	アクティブ時のテキスト色。
<code>activebackground</code>	アクティブ時の背景色。
<code>selectcolor</code>	チェックボックスの色。
<code>state</code>	チェックボタンの状態 ( <code>normal</code> , <code>active</code> , <code>disabled</code> )。
<code>anchor</code>	テキストの配置位置 ( <code>n</code> , <code>s</code> , <code>e</code> , <code>w</code> , <code>center</code> など)。
<code>justify</code>	複数行テキストの行揃え ( <code>left</code> , <code>center</code> , <code>right</code> )。

## 主要なメソッド

メソッド	説明
<code>select()</code>	チェックボタンをチェック状態にします。
<code>deselect()</code>	チェックボタンのチェックを外します。
<code>toggle()</code>	チェック状態を切り替えます。
<code>invoke()</code>	チェックボタンをクリックしたときと同じ動作を実行します。

## 実用的な例

### 設定オプション画面

```
import tkinter as tk
from tkinter import messagebox

class SettingsApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("設定オプション")
        self.geometry("400x350")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        title_label = tk.Label(self, text="アプリケーション設定", font=("Arial", 16, "bold"))
        title_label.pack(pady=10)

        # 一般設定セクション
        general_frame = tk.LabelFrame(self, text="一般設定", font=("Arial", 12, "bold"), padx=10, pady=10)
        general_frame.pack(fill="x", padx=20, pady=10)

        # 設定変数
        self.auto_save_var = tk.BooleanVar(value=True)
        self.startup_var = tk.BooleanVar()
        self.backup_var = tk.BooleanVar(value=True)

        # チェックボタン
        auto_save_check = tk.Checkbutton(
            general_frame,
            text="自動保存を有効にする",
            variable=self.auto_save_var,
            command=self.on_setting_changed
        )
        auto_save_check.pack(anchor="w", pady=2)

        startup_check = tk.Checkbutton(
            general_frame,
            text="Windowsスタートアップに追加",
            variable=self.startup_var,
            command=self.on_setting_changed
        )
        startup_check.pack(anchor="w", pady=2)

        backup_check = tk.Checkbutton(
            general_frame,
```

```

        text="自動バックアップを有効にする",
        variable=self.backup_var,
        command=self.on_setting_changed
    )
    backup_check.pack(anchor="w", pady=2)

    # 通知設定セクション
    notification_frame = tk.LabelFrame(self, text="通知設定", font=("Arial", 12, "bold"), padx=10, pady=10)
    notification_frame.pack(fill="x", padx=20, pady=10)

    # 通知設定変数
    self.email_notification_var = tk.BooleanVar()
    self.desktop_notification_var = tk.BooleanVar(value=True)
    self.sound_notification_var = tk.BooleanVar()

    email_notification_check = tk.Checkbutton(
        notification_frame,
        text="メール通知",
        variable=self.email_notification_var,
        command=self.on_setting_changed
    )
    email_notification_check.pack(anchor="w", pady=2)

    desktop_notification_check = tk.Checkbutton(
        notification_frame,
        text="デスクトップ通知",
        variable=self.desktop_notification_var,
        command=self.on_setting_changed
    )
    desktop_notification_check.pack(anchor="w", pady=2)

    sound_notification_check = tk.Checkbutton(
        notification_frame,
        text="音声通知",
        variable=self.sound_notification_var,
        command=self.on_setting_changed
    )
    sound_notification_check.pack(anchor="w", pady=2)

    # 詳細設定セクション
    advanced_frame = tk.LabelFrame(self, text="詳細設定", font=("Arial", 12, "bold"), padx=10, pady=10)
    advanced_frame.pack(fill="x", padx=20, pady=10)

    # 詳細設定変数
    self.debug_mode_var = tk.BooleanVar()
    self.telemetry_var = tk.BooleanVar()

    debug_mode_check = tk.Checkbutton(
        advanced_frame,
        text="デバッグモードを有効にする",
        variable=self.debug_mode_var,
        command=self.on_setting_changed
    )
    debug_mode_check.pack(anchor="w", pady=2)

    telemetry_check = tk.Checkbutton(
        advanced_frame,
        text="利用状況データの送信を許可",
        variable=self.telemetry_var,
        command=self.on_setting_changed
    )
    telemetry_check.pack(anchor="w", pady=2)

    # ボタンフレーム
    button_frame = tk.Frame(self)
    button_frame.pack(pady=20)

    save_button = tk.Button(button_frame, text="保存", command=self.save_settings, bg="lightblue")
    save_button.pack(side="left", padx=5)

    reset_button = tk.Button(button_frame, text="リセット", command=self.reset_settings, bg="lightcoral")
    reset_button.pack(side="left", padx=5)

    apply_button = tk.Button(button_frame, text="適用", command=self.apply_settings, bg="lightgreen")
    apply_button.pack(side="left", padx=5)

    def on_setting_changed(self):
        print("設定が変更されました")

    def save_settings(self):
        settings = self.get_current_settings()
        messagebox.showinfo("保存完了", f"設定が保存されました:\n\n{settings}")

    def reset_settings(self):
        # デフォルト値にリセット
        self.auto_save_var.set(True)
        self.startup_var.set(False)
        self.backup_var.set(True)
        self.email_notification_var.set(False)
        self.desktop_notification_var.set(True)
        self.sound_notification_var.set(False)
        self.debug_mode_var.set(False)
        self.telemetry_var.set(False)
        messagebox.showinfo("リセット完了", "設定がデフォルト値にリセットされました")

    def apply_settings(self):
        settings = self.get_current_settings()
        messagebox.showinfo("適用完了", f"設定が適用されました:\n\n{settings}")

    def get_current_settings(self):

```

```

settings = []
settings.append(f"自動保存: {'有効' if self.auto_save_var.get() else '無効'}")
settings.append(f"スタートアップ: {'有効' if self.startup_var.get() else '無効'}")
settings.append(f"自動バックアップ: {'有効' if self.backup_var.get() else '無効'}")
settings.append(f"メール通知: {'有効' if self.email_notification_var.get() else '無効'}")
settings.append(f"デスクトップ通知: {'有効' if self.desktop_notification_var.get() else '無効'}")
settings.append(f"音声通知: {'有効' if self.sound_notification_var.get() else '無効'}")
settings.append(f"デバッグモード: {'有効' if self.debug_mode_var.get() else '無効'}")
settings.append(f"テlemetry: {'有効' if self.telemetry_var.get() else '無効'}")
return "\n".join(settings)

```

```

if __name__ == "__main__":
    app = SettingsApp()
    app.mainloop()

```

## カスタム値を使ったチェックボタン

```

import tkinter as tk

class CustomValueApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("カスタム値チェックボタン")
        self.geometry("400x300")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        title_label = tk.Label(self, text="好きなプログラミング言語を選択", font=("Arial", 14, "bold"))
        title_label.pack(pady=10)

        # 言語選択用の変数 (StringVarを使用してカスタム値を設定)
        self.python_var = tk.StringVar()
        self.javascript_var = tk.StringVar()
        self.java_var = tk.StringVar()
        self.cpp_var = tk.StringVar()

        # Pythonチェックボタン
        python_check = tk.Checkbutton(
            self,
            text="Python",
            variable=self.python_var,
            onvalue="python_selected",
            offvalue="python_not_selected",
            command=self.update_display
        )
        python_check.pack(anchor="w", padx=50, pady=5)

        # JavaScriptチェックボタン
        javascript_check = tk.Checkbutton(
            self,
            text="JavaScript",
            variable=self.javascript_var,
            onvalue="js_selected",
            offvalue="js_not_selected",
            command=self.update_display
        )
        javascript_check.pack(anchor="w", padx=50, pady=5)

        # Javaチェックボタン
        java_check = tk.Checkbutton(
            self,
            text="Java",
            variable=self.java_var,
            onvalue="java_selected",
            offvalue="java_not_selected",
            command=self.update_display
        )
        java_check.pack(anchor="w", padx=50, pady=5)

        # C++チェックボタン
        cpp_check = tk.Checkbutton(
            self,
            text="C++",
            variable=self.cpp_var,
            onvalue="cpp_selected",
            offvalue="cpp_not_selected",
            command=self.update_display
        )
        cpp_check.pack(anchor="w", padx=50, pady=5)

        # 結果表示用テキストエリア
        self.result_text = tk.Text(self, width=50, height=8, font=("Consolas", 10))
        self.result_text.pack(pady=20)

        # 初期表示
        self.update_display()

    def update_display(self):
        # 結果をクリア
        self.result_text.delete(1.0, tk.END)

        # 現在の選択状況を表示
        result = "現在の選択状況:\n\n"

        python_status = self.python_var.get()

```

```

javascript_status = self.javascript_var.get()
java_status = self.java_var.get()
cpp_status = self.cpp_var.get()

result += f"Python: {python_status}\n"
result += f"JavaScript: {javascript_status}\n"
result += f"Java: {java_status}\n"
result += f"C++: {cpp_status}\n\n"

# 選択された言語をリストアップ
selected_languages = []
if "selected" in python_status:
    selected_languages.append("Python")
if "selected" in javascript_status:
    selected_languages.append("JavaScript")
if "selected" in java_status:
    selected_languages.append("Java")
if "selected" in cpp_status:
    selected_languages.append("C++")

if selected_languages:
    result += f"選択された言語: {' '.join(selected_languages)}\n"
else:
    result += "選択された言語: なし"

self.result_text.insert(1.0, result)

if __name__ == "__main__":
    app = CustomValueApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
適切な変数の使用	チェック状態を管理するために <code>tk.BooleanVar</code> を使用します。カスタム値が必要な場合は <code>tk.StringVar</code> や <code>tk.IntVar</code> も利用できます。
グループ化	関連するチェックボタンは <code>Frame</code> や <code>LabelFrame</code> でグループ化して整理します。
状態の初期化	変数の初期値を適切に設定して、期待される初期状態を明確にします。
コールバック関数	<code>command</code> オプションを使用してチェック状態の変更を検知し、適切な処理を実行します。
アクセシビリティ	<code>state</code> オプションを使用して、適切な状況でチェックボタンを無効化します。

## 参考リンク

- [Python Docs - tkinter.Checkbutton](#)
- [TkDocs - Checkbutton](#)

# Radiobutton リファレンス

複数の選択肢から一つを選択する `Radiobutton` ウィジェットについての詳細なリファレンスです。

## 概要

`Radiobutton` ウィジェットは、複数の選択肢の中から一つだけを選択するためのコントロールです。同じ変数を共有する複数のラジオボタンは相互排他的に動作し、一つのボタンが選択されると他のボタンは自動的に選択解除されます。

## 基本的な使用方法

### シンプルなラジオボタン

```
import tkinter as tk

def show_selection():
    selection = selected_option.get()
    print(f"選択された項目: {selection}")

app = tk.Tk()
app.title("Radiobuttonの例")
app.geometry("300x250")

# StringVar で選択状態を管理
selected_option = tk.StringVar(value="option1")

# ラジオボタンの作成
tk.Radiobutton(
    app,
    text="オプション 1",
    variable=selected_option,
    value="option1",
    command=show_selection
).pack(anchor="w", padx=20, pady=5)

tk.Radiobutton(
    app,
    text="オプション 2",
    variable=selected_option,
    value="option2",
    command=show_selection
).pack(anchor="w", padx=20, pady=5)

tk.Radiobutton(
    app,
    text="オプション 3",
    variable=selected_option,
    value="option3",
    command=show_selection
).pack(anchor="w", padx=20, pady=5)

button = tk.Button(app, text="選択を確認", command=show_selection)
button.pack(pady=20)

app.mainloop()
```

### クラスベースでのラジオボタン

```
import tkinter as tk

class RadiobuttonApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Radiobuttonの例（クラスベース）")
        self.geometry("300x250")

        self.create_widgets()

    def create_widgets(self):
        # StringVar で選択状態を管理
        self.selected_option = tk.StringVar(value="option1")

        # ラジオボタンの作成
        self.radio1 = tk.Radiobutton(
            self,
            text="オプション 1",
            variable=self.selected_option,
            value="option1",
            command=self.show_selection
        )
        self.radio1.pack(anchor="w", padx=20, pady=5)
```

```

self.radio2 = tk.Radiobutton(
    self,
    text="オプション 2",
    variable=self.selected_option,
    value="option2",
    command=self.show_selection
)
self.radio2.pack(anchor="w", padx=20, pady=5)

self.radio3 = tk.Radiobutton(
    self,
    text="オプション 3",
    variable=self.selected_option,
    value="option3",
    command=self.show_selection
)
self.radio3.pack(anchor="w", padx=20, pady=5)

self.button = tk.Button(self, text="選択を確認", command=self.show_selection)
self.button.pack(pady=20)

def show_selection(self):
    selection = self.selected_option.get()
    print(f"選択された項目: {selection}")

if __name__ == "__main__":
    app = RadiobuttonApp()
    app.mainloop()

```

## 主要なオプション

オプション	説明
<code>text</code>	ラジオボタンに表示するテキスト。
<code>variable</code>	選択状態を管理する変数 ( <code>tk.StringVar</code> , <code>tk.IntVar</code> など)。
<code>value</code>	このラジオボタンが選択されたときに変数に設定される値。
<code>command</code>	ラジオボタンが選択されたときに実行される関数。
<code>font</code>	フォントを指定。タプル ("フォント名", サイズ, "スタイル") や文字列で指定。
<code>fg (または foreground)</code>	テキストの色。
<code>bg (または background)</code>	ラジオボタンの背景色。
<code>activeforeground</code>	アクティブ時のテキスト色。
<code>activebackground</code>	アクティブ時の背景色。
<code>selectcolor</code>	ラジオボタンの色。
<code>state</code>	ラジオボタンの状態 ( <code>normal</code> , <code>active</code> , <code>disabled</code> )。
<code>anchor</code>	テキストの配置位置 ( <code>n</code> , <code>s</code> , <code>e</code> , <code>w</code> , <code>center</code> など)。
<code>justify</code>	複数行テキストの行揃え ( <code>left</code> , <code>center</code> , <code>right</code> )。
<code>indicatoron</code>	ラジオボタンのインジケータを表示するかどうか ( <code>True</code> / <code>False</code> )。

## 主要なメソッド

メソッド	説明
<code>select()</code>	このラジオボタンを選択状態にします。
<code>deselect()</code>	このラジオボタンの選択を解除します。
<code>invoke()</code>	ラジオボタンをクリックしたときと同じ動作を実行します。

## 実用的な例

### 設定選択画面

```

import tkinter as tk
from tkinter import messagebox

class ConfigurationApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("設定選択")
        self.geometry("500x700")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        title_label = tk.Label(self, text="アプリケーション設定", font=("Arial", 16, "bold"))
        title_label.pack(pady=10)

        # テーマ設定

```

```

theme_frame = tk.LabelFrame(self, text="テーマ選択", font=("Arial", 12, "bold"), padx=10, pady=10)
theme_frame.pack(fill="x", padx=20, pady=10)

self.theme_var = tk.StringVar(value="light")

themes = [
    ("ライトテーマ", "light"),
    ("ダークテーマ", "dark"),
    ("ハイコントラスト", "high_contrast"),
    ("システム設定に従う", "system")
]

for text, value in themes:
    tk.Radiobutton(
        theme_frame,
        text=text,
        variable=self.theme_var,
        value=value,
        command=self.on_theme_changed
    ).pack(anchor="w", pady=2)

# 言語設定
language_frame = tk.LabelFrame(self, text="言語選択", font=("Arial", 12, "bold"), padx=10, pady=10)
language_frame.pack(fill="x", padx=20, pady=10)

self.language_var = tk.StringVar(value="ja")

languages = [
    ("日本語", "ja"),
    ("English", "en"),
    ("Español", "es"),
    ("Français", "fr"),
    ("Deutsch", "de")
]

for text, value in languages:
    tk.Radiobutton(
        language_frame,
        text=text,
        variable=self.language_var,
        value=value,
        command=self.on_language_changed
    ).pack(anchor="w", pady=2)

# ファイルサイズ設定
filesize_frame = tk.LabelFrame(self, text="デフォルトファイルサイズ", font=("Arial", 12, "bold"), padx=10, pady=10)
filesize_frame.pack(fill="x", padx=20, pady=10)

self.filesize_var = tk.IntVar(value=1)

filesizes = [
    ("小 (1MB未満)", 1),
    ("中 (1-10MB)", 2),
    ("大 (10-100MB)", 3),
    ("特大 (100MB以上)", 4)
]

for text, value in filesizes:
    tk.Radiobutton(
        filesize_frame,
        text=text,
        variable=self.filesize_var,
        value=value,
        command=self.on_filesize_changed
    ).pack(anchor="w", pady=2)

# ボタンフレーム
button_frame = tk.Frame(self)
button_frame.pack(pady=20)

apply_button = tk.Button(button_frame, text="適用", command=self.apply_settings, bg="lightblue")
apply_button.pack(side="left", padx=5)

reset_button = tk.Button(button_frame, text="リセット", command=self.reset_settings, bg="lightcoral")
reset_button.pack(side="left", padx=5)

info_button = tk.Button(button_frame, text="現在の設定", command=self.show_current_settings, bg="lightgreen")
info_button.pack(side="left", padx=5)

def on_theme_changed(self):
    theme = self.theme_var.get()
    print(f"テーマが変更されました: {theme}")

def on_language_changed(self):
    language = self.language_var.get()
    print(f"言語が変更されました: {language}")

def on_filesize_changed(self):
    filesize = self.filesize_var.get()
    print(f"ファイルサイズ設定が変更されました: {filesize}")

def apply_settings(self):
    settings = self.get_current_settings()
    messagebox.showinfo("設定適用", f"以下の設定が適用されました:\n\n{settings}")

def reset_settings(self):
    self.theme_var.set("light")
    self.language_var.set("ja")
    self.filesize_var.set(1)
    messagebox.showinfo("リセット完了", "設定がデフォルト値にリセットされました")

```

```

def show_current_settings(self):
    settings = self.get_current_settings()
    messagebox.showinfo("現在の設定", settings)

def get_current_settings(self):
    theme_names = {
        "light": "ライトテーマ",
        "dark": "ダークテーマ",
        "high_contrast": "ハイコントラスト",
        "system": "システム設定に従う"
    }

    language_names = {
        "ja": "日本語",
        "en": "English",
        "es": "Español",
        "fr": "Français",
        "de": "Deutsch"
    }

    filesize_names = {
        1: "小 (1MB未満)",
        2: "中 (1-10MB)",
        3: "大 (10-100MB)",
        4: "特大 (100MB以上)"
    }

    theme = theme_names.get(self.theme_var.get(), "不明")
    language = language_names.get(self.language_var.get(), "不明")
    filesize = filesize_names.get(self.filesize_var.get(), "不明")

    return f"テーマ: {theme}\n言語: {language}\nファイルサイズ: {filesize}"

if __name__ == "__main__":
    app = ConfigurationApp()
    app.mainloop()

```

## インジケータなしのラジオボタン（ボタン風）

```

import tkinter as tk

class ButtonStyleRadioApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("ボタン風ラジオボタン")
        self.geometry("400x300")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        title_label = tk.Label(self, text="難易度を選択してください", font=("Arial", 16, "bold"))
        title_label.pack(pady=20)

        # 難易度選択
        self.difficulty_var = tk.StringVar(value="normal")

        difficulty_frame = tk.Frame(self)
        difficulty_frame.pack(pady=20)

        difficulties = [
            ("初心者", "beginner", "lightgreen"),
            ("普通", "normal", "lightblue"),
            ("上級者", "advanced", "orange"),
            ("エキスパート", "expert", "lightcoral")
        ]

        self.radio_buttons = []
        for i, (text, value, color) in enumerate(difficulties):
            rb = tk.Radiobutton(
                difficulty_frame,
                text=text,
                variable=self.difficulty_var,
                value=value,
                indicatoron=False, # インジケータを非表示にしてボタン風にする
                width=12,
                height=2,
                bg=color,
                font=("Arial", 12, "bold"),
                command=self.on_difficulty_changed
            )
            rb.grid(row=0, column=i, padx=5)
            self.radio_buttons.append(rb)

        # ゲームモード選択
        mode_label = tk.Label(self, text="ゲームモードを選択してください", font=("Arial", 14, "bold"))
        mode_label.pack(pady=(40, 10))

        self.mode_var = tk.StringVar(value="single")

        mode_frame = tk.Frame(self)
        mode_frame.pack()

        modes = [
            ("シングルプレイヤー", "single"),
            ("マルチプレイヤー", "multi"),
        ]

```

```

        ("協力プレイ", "coop"),
        ("対戦モード", "versus")
    ]

    for text, value in modes:
        tk.Radiobutton(
            mode_frame,
            text=text,
            variable=self.mode_var,
            value=value,
            font=("Arial", 11),
            command=self.on_mode_changed
        ).pack(anchor="w", pady=2)

    # 開始ボタン
    start_button = tk.Button(
        self,
        text="ゲーム開始",
        command=self.start_game,
        bg="darkgreen",
        fg="white",
        font=("Arial", 14, "bold"),
        height=2
    )
    start_button.pack(pady=30)

def on_difficulty_changed(self):
    difficulty = self.difficulty_var.get()
    print(f"難易度が選択されました: {difficulty}")

    # 選択されたボタンの見た目を変更
    for rb in self.radio_buttons:
        if rb['value'] == difficulty:
            rb.config(relief="sunken", bd=2)
        else:
            rb.config(relief="raised", bd=1)

def on_mode_changed(self):
    mode = self.mode_var.get()
    print(f"ゲームモードが選択されました: {mode}")

def start_game(self):
    difficulty = self.difficulty_var.get()
    mode = self.mode_var.get()

    difficulty_names = {
        "beginner": "初心者",
        "normal": "普通",
        "advanced": "上級者",
        "expert": "エキスパート"
    }

    mode_names = {
        "single": "シングルプレイヤー",
        "multi": "マルチプレイヤー",
        "coop": "協力プレイ",
        "versus": "対戦モード"
    }

    difficulty_text = difficulty_names.get(difficulty, difficulty)
    mode_text = mode_names.get(mode, mode)

    message = f"ゲームを開始します!\n\n難易度: {difficulty_text}\nモード: {mode_text}"

    # 簡単な確認ダイアログ風の表示
    result_window = tk.Toplevel(self)
    result_window.title("ゲーム開始")
    result_window.geometry("300x150")
    result_window.transient(self)
    result_window.grab_set()

    tk.Label(result_window, text=message, font=("Arial", 12)).pack(expand=True)
    tk.Button(result_window, text="OK", command=result_window.destroy).pack(pady=10)

if __name__ == "__main__":
    app = ButtonStyleRadioApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
変数の共有	同じグループのラジオボタンは同じ変数（ <code>tk.StringVar</code> など）を共有して、相互排他的な動作を実現します。
適切な値の設定	各ラジオボタンの <code>value</code> には識別しやすい一意の値を設定します。
初期値の設定	変数に初期値を設定して、デフォルトで選択されるオプションを明確にします。
グループ化	関連するラジオボタンは <code>Frame</code> や <code>LabelFrame</code> でグループ化して整理します。
カスタマイズ	<code>indicatoron=False</code> を使用してボタン風の見た目ににするなど、用途に応じてカスタマイズします。

## 参考リンク

- [Python Docs - `tkinter.Radiobutton`](#)

- [TkDocs - Radiobutton](#)

# Combobox リファレンス

ドロップダウンリストから項目を選択できる `Combobox` ウィジェットについての詳細なリファレンスです。

## 概要

`Combobox` ウィジェットは、複数の選択肢をドロップダウンリストとして表示し、ユーザーが一つの項目を選択できるコントロールです。Entry ウィジェットの機能も併せ持ち、リストにない値をユーザーが直接入力することも可能です。`tkinter.ttk` モジュールの一部として提供されます。

## 基本的な使用方法

### シンプルなコンボボックス

```
import tkinter as tk
from tkinter import ttk

def on_selection(event):
    selection = combobox.get()
    print(f"選択された項目: {selection}")

app = tk.Tk()
app.title("Comboboxの例")
app.geometry("300x200")

# コンボボックスの作成
values = ["りんご", "みかん", "バナナ", "ぶどう", "いちご"]
combobox = ttk.Combobox(app, values=values)
combobox.set("りんご") # 初期値を設定
combobox.bind("<<ComboboxSelected>>", on_selection)
combobox.pack(pady=20)

app.mainloop()
```

### クラスベースでのコンボボックス

```
import tkinter as tk
from tkinter import ttk

class ComboboxApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Comboboxの例（クラスベース）")
        self.geometry("300x200")

        self.create_widgets()

    def create_widgets(self):
        # コンボボックスの作成
        values = ["りんご", "みかん", "バナナ", "ぶどう", "いちご"]
        self.combobox = ttk.Combobox(self, values=values)
        self.combobox.set("りんご") # 初期値を設定
        self.combobox.bind("<<ComboboxSelected>>", self.on_selection)
        self.combobox.pack(pady=20)

    def on_selection(self, event):
        selection = self.combobox.get()
        print(f"選択された項目: {selection}")

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

## 主要なオプション

オプション	説明
<code>values</code>	ドロップダウンリストに表示される項目のタプルまたはリスト。
<code>textvariable</code>	<code>tk.StringVar</code> などの変数を指定し、選択値と同期します。
<code>state</code>	コンボボックスの状態 ( <code>normal</code> , <code>readonly</code> , <code>disabled</code> )。
<code>width</code>	コンボボックスの幅を文字数で指定します。
<code>font</code>	フォントを指定。
<code>justify</code>	テキストの配置 ( <code>left</code> , <code>center</code> , <code>right</code> )。

オプション	説明
<code>exportselection</code>	選択をクリップボードにエクスポートするかどうか。
<code>postcommand</code>	ドロップダウンが開かれる前に実行される関数。

## 主要なメソッド

メソッド	説明
<code>get()</code>	現在の値を取得します。
<code>set(value)</code>	値を設定します。
<code>current(index=None)</code>	現在の選択インデックスを取得または設定します。
<code>configure(values=...)</code>	valuesオプションを動的に変更します。

## イベント

イベント	説明
<code>&lt;&lt;ComboboxSelected&gt;&gt;</code>	ユーザーがドロップダウンから項目を選択したときに発生します。

## 実用的な例

### 動的な選択肢の変更

```

import tkinter as tk
from tkinter import ttk

class DynamicComboboxApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("動的コンボボックス")
        self.geometry("400x300")

        self.create_widgets()

    def create_widgets(self):
        # カテゴリ選択
        tk.Label(self, text="カテゴリを選択:", font=("Arial", 12)).pack(pady=10)

        self.category_var = tk.StringVar()
        self.category_combo = ttk.Combobox(
            self,
            textvariable=self.category_var,
            values=["果物", "野菜", "肉類", "魚類"],
            state="readonly"
        )
        self.category_combo.bind("<<ComboboxSelected>>", self.on_category_changed)
        self.category_combo.pack(pady=5)

        # 商品選択
        tk.Label(self, text="商品を選択:", font=("Arial", 12)).pack(pady=(20, 10))

        self.product_var = tk.StringVar()
        self.product_combo = ttk.Combobox(
            self,
            textvariable=self.product_var,
            state="disabled"
        )
        self.product_combo.bind("<<ComboboxSelected>>", self.on_product_selected)
        self.product_combo.pack(pady=5)

        # 結果表示
        self.result_label = tk.Label(self, text="", font=("Arial", 11), fg="blue")
        self.result_label.pack(pady=20)

        # 商品データ
        self.products = {
            "果物": ["りんご", "みかん", "バナナ", "ぶどう", "いちご"],
            "野菜": ["キャベツ", "人参", "玉ねぎ", "じゃがいも", "トマト"],
            "肉類": ["牛肉", "豚肉", "鶏肉", "ラム肉"],
            "魚類": ["サーモン", "マグロ", "アジ", "サバ", "イワシ"]
        }

    def on_category_changed(self, event):
        category = self.category_var.get()
        if category in self.products:
            # 商品選択肢を更新
            self.product_combo.configure(values=self.products[category])
            self.product_combo.set("") # 選択をリセット
            self.product_combo.configure(state="readonly")
            self.result_label.config(text="")

    def on_product_selected(self, event):
        category = self.category_var.get()
        product = self.product_var.get()
        self.result_label.config(text=f"選択: {category} - {product}")

```

```

if __name__ == "__main__":
    app = DynamicComboboxApp()
    app.mainloop()

```

## 検索可能なコンボボックス

```

import tkinter as tk
from tkinter import ttk

class SearchableComboboxApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("検索可能なコンボボックス")
        self.geometry("400x350")

        self.create_widgets()

    def create_widgets(self):
        tk.Label(self, text="国名を入力または選択:", font=("Arial", 12)).pack(pady=10)

        # 国名リスト
        self.countries = [
            "日本", "アメリカ", "イギリス", "フランス", "ドイツ", "イタリア",
            "スペイン", "カナダ", "オーストラリア", "ブラジル", "インド",
            "中国", "韓国", "タイ", "ベトナム", "シンガポール", "マレーシア"
        ]

        self.country_var = tk.StringVar()
        self.country_combo = ttk.Combobox(
            self,
            textvariable=self.country_var,
            values=self.countries,
            width=30
        )
        self.country_combo.pack(pady=5)

        # キー入力に応じて候補を絞り込む
        self.country_combo.bind('<KeyRelease>', self.on_key_release)
        self.country_combo.bind("<<ComboboxSelected>>", self.on_selection)

        # 結果表示エリア
        tk.Label(self, text="選択結果:", font=("Arial", 12)).pack(pady=(20, 5))

        self.result_text = tk.Text(self, width=40, height=10, font=("Arial", 10))
        self.result_text.pack(pady=5)

        # 情報表示ボタン
        info_button = tk.Button(self, text="国情情報を表示", command=self.show_country_info)
        info_button.pack(pady=10)

        # 国の情報 (サンプル)
        self.country_info = {
            "日本": "首都: 東京\n人口: 約1億2500万人\n通貨: 円",
            "アメリカ": "首都: ワシントンD.C.\n人口: 約3億3000万人\n通貨: ドル",
            "イギリス": "首都: ロンドン\n人口: 約6700万人\n通貨: ポンド",
            "フランス": "首都: パリ\n人口: 約6800万人\n通貨: ユーロ",
            "ドイツ": "首都: ベルリン\n人口: 約8300万人\n通貨: ユーロ"
        }

    def on_key_release(self, event):
        # 入力値に基づいて候補を絞り込む
        typed = self.country_var.get().lower()
        if typed == '':
            self.country_combo.configure(values=self.countries)
        else:
            filtered = [country for country in self.countries
                        if typed in country.lower()]
            self.country_combo.configure(values=filtered)

    def on_selection(self, event):
        selected = self.country_var.get()
        self.result_text.delete(1.0, tk.END)
        self.result_text.insert(1.0, f"選択された国: {selected}")

    def show_country_info(self):
        selected = self.country_var.get()
        if selected in self.country_info:
            info = self.country_info[selected]
            self.result_text.delete(1.0, tk.END)
            self.result_text.insert(1.0, f"国名: {selected}\n\n{info}")
        else:
            self.result_text.delete(1.0, tk.END)
            self.result_text.insert(1.0, f"'{selected}' の情報は登録されていません。")

    if __name__ == "__main__":
        app = SearchableComboboxApp()
        app.mainloop()

```

## ベストプラクティス

プラクティス	説明
適切な状態の設定	<code>readonly</code> 状態を使用して、リストからの選択のみを許可することができます。

プラクティス	説明
イベントハンドリング	<code>&lt;&lt;ComboboxSelected&gt;&gt;</code> イベントを使用して選択変更を検知します。
動的な選択肢	<code>configure(values=...)</code> を使用して選択肢を動的に変更できます。
初期値の設定	<code>set()</code> メソッドを使用して適切な初期値を設定します。
検索機能の実装	キー入力イベントを使用して検索機能を実装できます。

## 参考リンク

- [Python Docs - tkinter.ttk.Combobox](#)
- [TkDocs - Combobox](#)

# Listbox リファレンス

複数の項目を一覧表示し、選択を行う `Listbox` ウィジェットについての詳細なリファレンスです。

## 概要

`Listbox` ウィジェットは、項目の一覧を表示し、ユーザーが一つまたは複数の項目を選択できるコントロールです。スクロール機能も内蔵しており、多数の項目を効率的に表示できます。単一選択、複数選択、範囲選択などの選択モードをサポートします。

## 基本的な使用方法

### シンプルなリストボックス

```
import tkinter as tk

def on_selection(event):
    selection = listbox.curselection()
    if selection:
        index = selection[0]
        value = listbox.get(index)
        print(f"選択された項目: {value} (インデックス: {index})")

app = tk.Tk()
app.title("Listboxの例")
app.geometry("300x250")

# リストボックスの作成
listbox = tk.Listbox(app, height=6)
listbox.pack(pady=20)

# 項目を追加
items = ["りんご", "みかん", "バナナ", "ぶどう", "いちご", "メロン", "スイカ"]
for item in items:
    listbox.insert(tk.END, item)

# 選択イベントをバインド
listbox.bind("<<ListboxSelect>>", on_selection)

app.mainloop()
```

### クラスベースでのリストボックス

```
import tkinter as tk

class ListboxApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Listboxの例 (クラスベース) ")
        self.geometry("300x250")

        self.create_widgets()

    def create_widgets(self):
        # リストボックスの作成
        self.listbox = tk.Listbox(self, height=6)
        self.listbox.pack(pady=20)

        # 項目を追加
        items = ["りんご", "みかん", "バナナ", "ぶどう", "いちご", "メロン", "スイカ"]
        for item in items:
            self.listbox.insert(tk.END, item)

        # 選択イベントをバインド
        self.listbox.bind("<<ListboxSelect>>", self.on_selection)

    def on_selection(self, event):
        selection = self.listbox.curselection()
        if selection:
            index = selection[0]
            value = self.listbox.get(index)
            print(f"選択された項目: {value} (インデックス: {index})")

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

## 主要なオプション

オプション	説明
height	表示される行数。
width	幅を文字数で指定。
selectmode	選択モード ( <code>SINGLE</code> , <code>BROWSE</code> , <code>MULTIPLE</code> , <code>EXTENDED</code> )。
font	フォントを指定。
fg ( または foreground )	テキストの色。
bg ( または background )	背景色。
selectbackground	選択された項目の背景色。
selectforeground	選択された項目のテキスト色。
relief	境界線のスタイル ( <code>flat</code> , <code>raised</code> , <code>sunken</code> , <code>groove</code> , <code>ridge</code> )。
borderwidth ( または bd )	境界線の幅。
activestyle	アクティブ項目のスタイル ( <code>underline</code> , <code>dotbox</code> , <code>none</code> )。
listvariable	<code>tk.StringVar</code> でリストの内容を管理。

## 選択モード

モード	説明
<code>tk.SINGLE</code>	一度に一つの項目のみ選択可能。
<code>tk.BROWSE</code>	一つの項目のみ選択可能だが、ドラッグで選択を変更可能。
<code>tk.MULTIPLE</code>	複数の項目を個別に選択/解除可能。
<code>tk.EXTENDED</code>	複数選択可能で、Shift/Ctrl キーとの組み合わせで範囲選択可能。

## 主要なメソッド

メソッド	説明
<code>insert(index, *elements)</code>	指定位置に項目を挿入します。
<code>delete(first, last=None)</code>	指定範囲の項目を削除します。
<code>get(first, last=None)</code>	指定項目の値を取得します。
<code>curselection()</code>	現在選択されている項目のインデックスのタプルを返します。
<code>selection_set(first, last=None)</code>	指定項目を選択状態にします。
<code>selection_clear(first, last=None)</code>	指定項目の選択を解除します。
<code>size()</code>	リスト内の項目数を返します。
<code>see(index)</code>	指定項目が見えるようにスクロールします。
<code>activate(index)</code>	指定項目をアクティブにします。

## 実用的な例

### 複数選択とアクション付きリストボックス

```
import tkinter as tk
from tkinter import messagebox

class MultiSelectListboxApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("複数選択リストボックス")
        self.geometry("400x500")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        tk.Label(self, text="プログラミング言語を選択（複数可）:", font=("Arial", 12, "bold")).pack(pady=10)

        # リストボックス（複数選択モード）
        self.listbox = tk.Listbox(self, selectmode=tk.MULTIPLE, height=8, font=("Arial", 10))
        self.listbox.pack(pady=10)

        # 項目を追加
        languages = [
            "Python", "JavaScript", "Java", "C++", "C#", "Go", "Rust",
            "TypeScript", "PHP", "Ruby", "Swift", "Kotlin", "Dart"
        ]
```

```

]
for lang in languages:
    self.listbox.insert(tk.END, lang)

# ボタンフレーム
button_frame = tk.Frame(self)
button_frame.pack(pady=10)

# 各種ボタン
tk.Button(button_frame, text="選択項目を表示", command=self.show_selection).pack(side=tk.LEFT, padx=5)
tk.Button(button_frame, text="すべて選択", command=self.select_all).pack(side=tk.LEFT, padx=5)
tk.Button(button_frame, text="選択解除", command=self.clear_selection).pack(side=tk.LEFT, padx=5)

# 項目操作フレーム
operation_frame = tk.Frame(self)
operation_frame.pack(pady=10)

tk.Label(operation_frame, text="新しい言語:", font=("Arial", 10)).pack(side=tk.LEFT, padx=5)
self.new_lang_entry = tk.Entry(operation_frame, width=15)
self.new_lang_entry.pack(side=tk.LEFT, padx=5)

tk.Button(operation_frame, text="追加", command=self.add_language).pack(side=tk.LEFT, padx=5)
tk.Button(operation_frame, text="削除", command=self.remove_selected).pack(side=tk.LEFT, padx=5)

# 結果表示エリア
tk.Label(self, text="選択結果:", font=("Arial", 10, "bold")).pack(pady=(20, 5))
self.result_text = tk.Text(self, width=50, height=6, font=("Arial", 9))
self.result_text.pack(pady=5)

def show_selection(self):
    selection_indices = self.listbox.curselection()
    if selection_indices:
        selected_items = [self.listbox.get(i) for i in selection_indices]
        result = f"選択された言語 ({len(selected_items)}個):\n"
        result += "\n".join(f"- {item}" for item in selected_items)
    else:
        result = "何も選択されていません。"

    self.result_text.delete(1.0, tk.END)
    self.result_text.insert(1.0, result)

def select_all(self):
    self.listbox.selection_set(0, tk.END)
    self.show_selection()

def clear_selection(self):
    self.listbox.selection_clear(0, tk.END)
    self.result_text.delete(1.0, tk.END)

def add_language(self):
    new_lang = self.new_lang_entry.get().strip()
    if new_lang:
        # 重複チェック
        current_items = [self.listbox.get(i) for i in range(self.listbox.size())]
        if new_lang not in current_items:
            self.listbox.insert(tk.END, new_lang)
            self.new_lang_entry.delete(0, tk.END)
            messagebox.showinfo("追加完了", f"'{new_lang}' を追加しました。")
        else:
            messagebox.showwarning("重複エラー", f"'{new_lang}' は既に存在します。")
    else:
        messagebox.showwarning("入力エラー", "言語名を入力してください。")

def remove_selected(self):
    selection_indices = list(self.listbox.curselection())
    if selection_indices:
        # 逆順で削除 (インデックスのずれを防ぐため)
        for index in reversed(selection_indices):
            self.listbox.delete(index)
        self.result_text.delete(1.0, tk.END)
        messagebox.showinfo("削除完了", f"{len(selection_indices)}個の項目を削除しました。")
    else:
        messagebox.showwarning("選択エラー", "削除する項目を選択してください。")

if __name__ == "__main__":
    app = MultiSelectListboxApp()
    app.mainloop()

```

## スクロールバー付きリストボックス

```

import tkinter as tk

class ScrollableListboxApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("スクロール可能リストボックス")
        self.geometry("400x300")

        self.create_widgets()

    def create_widgets(self):
        # フレームを作成
        list_frame = tk.Frame(self)
        list_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=20)

        # リストボックスとスクロールバー

```

```

self.listBox = tk.Listbox(listFrame, selectmode=tk.EXTENDED)
scrollbar = tk.Scrollbar(listFrame, orient=tk.VERTICAL)

# スクロールバーとリストボックスを接続
self.listBox.config(yscrollcommand=scrollbar.set)
scrollbar.config(command=self.listBox.yview)

# 配置
self.listBox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

# 大量の項目を追加
for i in range(1, 101):
    self.listBox.insert(tk.END, f"項目 {i:03d}: サンプルデータ {i}")

# 操作ボタン
buttonFrame = tk.Frame(self)
buttonFrame.pack(pady=10)

tk.Button(buttonFrame, text="先頭へ", command=self.go_to_top).pack(side=tk.LEFT, padx=5)
tk.Button(buttonFrame, text="末尾へ", command=self.go_to_bottom).pack(side=tk.LEFT, padx=5)
tk.Button(buttonFrame, text="ランダム選択", command=self.random_select).pack(side=tk.LEFT, padx=5)

# 選択イベント
self.listBox.bind("<<ListboxSelect>>", self.on_selection)

# 結果表示
self.resultLabel = tk.Label(self, text="項目を選択してください", font=("Arial", 10))
self.resultLabel.pack(pady=5)

def on_selection(self, event):
    selection = self.listBox.curselection()
    if selection:
        if len(selection) == 1:
            index = selection[0]
            value = self.listBox.get(index)
            self.resultLabel.config(text=f"選択: {value}")
        else:
            self.resultLabel.config(text=f"{len(selection)}個の項目が選択されています")

def go_to_top(self):
    self.listBox.see(0)
    self.listBox.selection_clear(0, tk.END)
    self.listBox.selection_set(0)
    self.listBox.activate(0)

def go_to_bottom(self):
    last_index = self.listBox.size() - 1
    self.listBox.see(last_index)
    self.listBox.selection_clear(0, tk.END)
    self.listBox.selection_set(last_index)
    self.listBox.activate(last_index)

def random_select(self):
    import random
    size = self.listBox.size()
    if size > 0:
        random_index = random.randint(0, size - 1)
        self.listBox.see(random_index)
        self.listBox.selection_clear(0, tk.END)
        self.listBox.selection_set(random_index)
        self.listBox.activate(random_index)

if __name__ == "__main__":
    app = ScrollableListBoxApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
適切な選択モード	用途に応じて適切な <code>selectmode</code> を選択します。複数選択が必要な場合は <code>MULTIPLE</code> や <code>EXTENDED</code> を使用します。
スクロールバーの追加	多数の項目を扱う場合は、スクロールバーを追加してユーザビリティを向上させます。
選択状態の管理	<code>curselection()</code> を使用して現在の選択状態を適切に管理します。
項目の動的管理	<code>insert()</code> と <code>delete()</code> メソッドを使用して項目を動的に追加・削除します。
イベントハンドリング	<code>&lt;&lt;ListboxSelect&gt;&gt;</code> イベントを使用して選択変更を検知します。

## 参考リンク

- [Python Docs - tkinter.ListBox](#)
- [TkDocs - Listbox](#)

# Scale リファレンス

数値をスライダーで選択する `Scale` ウィジェットについての詳細なリファレンスです。

## 概要

`Scale` ウィジェットは、指定された範囲内の数値をスライダーで選択できるコントロールです。水平または垂直に配置でき、連続的または離散的な値の選択に使用されます。音量調整、明度調整、数値パラメータの設定などに適しています。

## 基本的な使用方法

### シンプルなスケール

```
import tkinter as tk

def on_scale_change(value):
    print(f"スライダーの値: {value}")

app = tk.Tk()
app.title("Scaleの例")
app.geometry("300x200")

# 水平スケールの作成
scale = tk.Scale(
    app,
    from_=0,
    to=100,
    orient=tk.HORIZONTAL,
    command=on_scale_change
)
scale.set(50) # 初期値を設定
scale.pack(pady=20)

app.mainloop()
```

### クラスベースでのスケール

```
import tkinter as tk

class ScaleApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Scaleの例（クラスベース）")
        self.geometry("300x200")

        self.create_widgets()

    def create_widgets(self):
        # 水平スケールの作成
        self.scale = tk.Scale(
            self,
            from_=0,
            to=100,
            orient=tk.HORIZONTAL,
            command=self.on_scale_change
        )
        self.scale.set(50) # 初期値を設定
        self.scale.pack(pady=20)

    def on_scale_change(self, value):
        print(f"スライダーの値: {value}")

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

## 主要なオプション

オプション	説明
<code>from_</code>	スケールの最小値。
<code>to</code>	スケールの最大値。
<code>orient</code>	スケールの向き ( <code>tk.HORIZONTAL</code> または <code>tk.VERTICAL</code> )。
<code>resolution</code>	スケールの分解能（最小変化量）。デフォルトは1。
<code>variable</code>	<code>tk.IntVar</code> や <code>tk.DoubleVar</code> で値を管理。

オプション	説明
<code>command</code>	値が変更されたときに実行される関数。
<code>length</code>	スケールの長さをピクセルで指定。
<code>width</code>	スケールの幅をピクセルで指定。
<code>label</code>	スケールの上（または左）に表示するラベルテキスト。
<code>showvalue</code>	現在の値を表示するかどうか ( <code>True</code> / <code>False</code> )。
<code>tickinterval</code>	目盛りの間隔。0の場合は目盛りを表示しない。
<code>font</code>	ラベルのフォント。
<code>fg</code> (または <code>foreground</code> )	テキストの色。
<code>bg</code> (または <code>background</code> )	背景色。
<code>relief</code>	境界線のスタイル。
<code>borderwidth</code> (または <code>bd</code> )	境界線の幅。

## 主要なメソッド

メソッド	説明
<code>get()</code>	現在の値を取得します。
<code>set(value)</code>	値を設定します。

## 実用的な例

### 色調整アプリケーション

```

import tkinter as tk

class ColorMixerApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("カラーミキサー")
        self.geometry("400x450")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        tk.Label(self, text="RGB カラーミキサー", font=("Arial", 16, "bold")).pack(pady=10)

        # 色表示用のキャンバス
        self.color_canvas = tk.Canvas(self, width=200, height=100, bg="black")
        self.color_canvas.pack(pady=10)

        # RGB値表示用のラベル
        self.rgb_label = tk.Label(self, text="RGB: (0, 0, 0)", font=("Arial", 12))
        self.rgb_label.pack(pady=5)

        # 16進値表示用のラベル
        self.hex_label = tk.Label(self, text="HEX: #000000", font=("Arial", 12))
        self.hex_label.pack(pady=5)

        # RGBスライダーのフレーム
        sliders_frame = tk.Frame(self)
        sliders_frame.pack(pady=20)

        # 赤 (Red) スライダー
        red_frame = tk.Frame(sliders_frame)
        red_frame.pack(pady=5)
        tk.Label(red_frame, text="Red:", width=6, fg="red", font=("Arial", 10, "bold")).pack(side=tk.LEFT)
        self.red_scale = tk.Scale(
            red_frame,
            from_=0,
            to=255,
            orient=tk.HORIZONTAL,
            length=200,
            command=self.update_color
        )
        self.red_scale.pack(side=tk.LEFT, padx=10)

        # 緑 (Green) スライダー
        green_frame = tk.Frame(sliders_frame)
        green_frame.pack(pady=5)
        tk.Label(green_frame, text="Green:", width=6, fg="green", font=("Arial", 10, "bold")).pack(side=tk.LEFT)
        self.green_scale = tk.Scale(
            green_frame,
            from_=0,
            to=255,
            orient=tk.HORIZONTAL,
            length=200,
            command=self.update_color
        )
        self.green_scale.pack(side=tk.LEFT, padx=10)

```

```

# 青(Blue)スライダー
blue_frame = tk.Frame(sliders_frame)
blue_frame.pack(pady=5)
tk.Label(blue_frame, text="Blue:", width=6, fg="blue", font=("Arial", 10, "bold")).pack(side=tk.LEFT)
self.blue_scale = tk.Scale(
    blue_frame,
    from_=0,
    to=255,
    orient=tk.HORIZONTAL,
    length=200,
    command=self.update_color
)
self.blue_scale.pack(side=tk.LEFT, padx=10)

# プリセットボタン
preset_frame = tk.Frame(self)
preset_frame.pack(pady=10)

presets = [
    ("赤", 255, 0, 0),
    ("緑", 0, 255, 0),
    ("青", 0, 0, 255),
    ("黄", 255, 255, 0),
    ("紫", 255, 0, 255),
    ("水色", 0, 255, 255),
    ("白", 255, 255, 255),
    ("黒", 0, 0, 0)
]
for i, (name, r, g, b) in enumerate(presets):
    if i % 4 == 0:
        row_frame = tk.Frame(preset_frame)
        row_frame.pack()

    btn = tk.Button(
        row_frame,
        text=name,
        width=8,
        command=lambda r=r, g=g, b=b: self.set_color(r, g, b)
    )
    btn.pack(side=tk.LEFT, padx=2, pady=2)

# 初期色を設定
self.update_color()

def update_color(self, event=None):
    red = self.red_scale.get()
    green = self.green_scale.get()
    blue = self.blue_scale.get()

    # RGB値を16進数に変換
    hex_color = f"#{red:02x}{green:02x}{blue:02x}"

    # キャンバスの色を更新
    self.color_canvas.configure(bg=hex_color)

    # ラベルを更新
    self.rgb_label.config(text=f"RGB: ({red}, {green}, {blue})")
    self.hex_label.config(text=f"HEX: {hex_color.upper()}")

def set_color(self, red, green, blue):
    self.red_scale.set(red)
    self.green_scale.set(green)
    self.blue_scale.set(blue)
    self.update_color()

if __name__ == "__main__":
    app = ColorMixerApp()
    app.mainloop()

```

## 計算機と設定パネル

```

import tkinter as tk
import math

class CalculatorApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("関数計算機")
        self.geometry("500x400")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        tk.Label(self, text="三角関数計算機", font=("Arial", 16, "bold")).pack(pady=10)

        # メインフレーム
        main_frame = tk.Frame(self)
        main_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=10)

        # 左側: コントロールパネル
        control_frame = tk.Frame(main_frame)
        control_frame.pack(side=tk.LEFT, fill=tk.Y, padx=(0, 20))

        # 角度スライダー (0~360度)

```

```

        tk.Label(control_frame, text="角度 (度)", font=("Arial", 12, "bold")).pack(pady=(0, 5))
        self.angle_scale = tk.Scale(
            control_frame,
            from_=0,
            to=360,
            orient=tk.VERTICAL,
            length=200,
            tickinterval=90,
            command=self.update_calculation
        )
        self.angle_scale.pack()

    # 振幅スライダー
    tk.Label(control_frame, text="振幅", font=("Arial", 12, "bold")).pack(pady=(20, 5))
    self.amplitude_scale = tk.Scale(
        control_frame,
        from_=0.1,
        to=3.0,
        resolution=0.1,
        orient=tk.VERTICAL,
        length=150,
        command=self.update_calculation
    )
    self.amplitude_scale.set(1.0)
    self.amplitude_scale.pack()

    # 右側: 結果表示
    result_frame = tk.Frame(main_frame)
    result_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

    # 結果表示エリア
    tk.Label(result_frame, text="計算結果", font=("Arial", 14, "bold")).pack(pady=(0, 10))

    self.result_text = tk.Text(result_frame, width=30, height=20, font=("Consolas", 10))
    self.result_text.pack(fill=tk.BOTH, expand=True)

    # リセットボタン
    reset_frame = tk.Frame(self)
    reset_frame.pack(pady=10)

    tk.Button(reset_frame, text="リセット", command=self.reset_values).pack(side=tk.LEFT, padx=5)
    tk.Button(reset_frame, text="ランダム設定", command=self.random_values).pack(side=tk.LEFT, padx=5)

    # 初期計算
    self.update_calculation()

def update_calculation(self, event=None):
    angle_deg = self.angle_scale.get()
    amplitude = self.amplitude_scale.get()

    # 度をラジアンに変換
    angle_rad = math.radians(angle_deg)

    # 三角関数の計算
    sin_val = amplitude * math.sin(angle_rad)
    cos_val = amplitude * math.cos(angle_rad)
    tan_val = amplitude * math.tan(angle_rad) if abs(math.cos(angle_rad)) > 1e-10 else float('inf')

    # 結果テキストを作成
    result = f"角度: {angle_deg}°\n"
    result += f"ラジアン: {angle_rad:.4f}\n"
    result += f"振幅: {amplitude}\n"
    result += f"{ '-' * 25}\n\n"
    result += f"sin({angle_deg}) × {amplitude} = {sin_val:.4f}\n"
    result += f"cos({angle_deg}) × {amplitude} = {cos_val:.4f}\n"

    if tan_val == float('inf'):
        result += f"tan({angle_deg}) × {amplitude} = undefined\n\n"
    else:
        result += f"tan({angle_deg}) × {amplitude} = {tan_val:.4f}\n\n"

    # 特殊角度の判定
    special_angles = {0: "0°", 30: "30°", 45: "45°", 60: "60°", 90: "90°",
                      120: "120°", 135: "135°", 150: "150°", 180: "180°",
                      210: "210°", 225: "225°", 240: "240°", 270: "270°",
                      300: "300°", 315: "315°", 330: "330°", 360: "360°"}

    if angle_deg in special_angles:
        result += f"* {special_angles[angle_deg]} は特殊角です\n\n"

    # 座標系での位置
    x = cos_val
    y = sin_val
    result += f"座標: ({x:.4f}, {y:.4f})\n"

    # 象限の判定
    if x > 0 and y >= 0:
        quadrant = "第1象限"
    elif x <= 0 and y > 0:
        quadrant = "第2象限"
    elif x < 0 and y <= 0:
        quadrant = "第3象限"
    elif x >= 0 and y < 0:
        quadrant = "第4象限"
    else:
        quadrant = "軸上"

    result += f"位置: {quadrant}"

    # テキストエリアを更新

```

```

    self.result_text.delete(1.0, tk.END)
    self.result_text.insert(1.0, result)

def reset_values(self):
    self.angle_scale.set(0)
    self.amplitude_scale.set(1.0)
    self.update_calculation()

def random_values(self):
    import random
    random_angle = random.randint(0, 360)
    random_amplitude = round(random.uniform(0.1, 3.0), 1)

    self.angle_scale.set(random_angle)
    self.amplitude_scale.set(random_amplitude)
    self.update_calculation()

if __name__ == "__main__":
    app = CalculatorApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
適切な範囲設定	from_ と to を用途に応じて適切に設定します。
分解能の調整	resolution オプションで適切な精度を設定します。小数が必要な場合は0.1などを指定します。
リアルタイム更新	command オプションを使用してスライダーの変更をリアルタイムで反映します。
変数との連携	variable オプションを使用してアプリケーションの状態と同期できます。
視覚的フィードバック	スライダーの変更に応じて視覚的な変化を提供してユーザビリティを向上させます。

## 参考リンク

- [Python Docs - tkinter.Scale](#)
- [TkDocs - Scale](#)

# Scrollbar リファレンス

スクロール可能なウィジェットをスクロールするための `Scrollbar` ウィジェットについての詳細なリファレンスです。

## 概要

`Scrollbar` ウィジェットは、`Text`、`Listbox`、`Canvas` などのスクロール可能なウィジェットと連携して、コンテンツのスクロール機能を提供するコントロールです。垂直または水平方向のスクロールをサポートし、大量のデータを効率的に表示できます。

## 基本的な使用方法

### Text ウィジェットとの組み合わせ

```
import tkinter as tk

app = tk.Tk()
app.title("Scrollbarの例")
app.geometry("400x300")

# メインフレームを作成し、gridで配置と伸縮の設定
main_frame = tk.Frame(app)
main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
main_frame.grid_rowconfigure(0, weight=1)
main_frame.grid_columnconfigure(0, weight=1)

# Text ウィジェットとScrollbarを作成
text_widget = tk.Text(main_frame, wrap=tk.NONE) # wrap=tk.NONEで横スクロールも有効に
v_scrollbar = tk.Scrollbar(main_frame, orient=tk.VERTICAL, command=text_widget.yview)
h_scrollbar = tk.Scrollbar(main_frame, orient=tk.HORIZONTAL, command=text_widget.xview)

# スクロールバーとText ウィジェットを接続
text_widget.config(yscrollcommand=v_scrollbar.set, xscrollcommand=h_scrollbar.set)

# gridでウィジェットを配置
text_widget.grid(row=0, column=0, sticky="nsew")
v_scrollbar.grid(row=0, column=1, sticky="ns")
h_scrollbar.grid(row=1, column=0, sticky="ew")

# サンプルテキストを追加
sample_text = ""
for i in range(1, 51):
    sample_text += f"行 {i}: これは非常に長いサンプルテキストです。ウィンドウの幅を超えることで、水平スクロールバーの必要性を示します。\\n"
text_widget.insert(1.0, sample_text)

app.mainloop()
```

### クラスベースでのスクロールバー

```
class ScrollbarApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Scrollbarの例（クラスベース）")
        self.geometry("450x450")

        self.create_widgets()

    def create_widgets(self):
        # メインフレームを作成し、gridで配置と伸縮の設定
        main_frame = tk.Frame(self)
        main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
        main_frame.grid_rowconfigure(0, weight=1)
        main_frame.grid_columnconfigure(0, weight=1)

        # Text ウィジェットとScrollbarを作成
        self.text_widget = tk.Text(main_frame, wrap=tk.NONE)
        self.v_scrollbar = tk.Scrollbar(main_frame, orient=tk.VERTICAL, command=self.text_widget.yview)
        self.h_scrollbar = tk.Scrollbar(main_frame, orient=tk.HORIZONTAL, command=self.text_widget.xview)

        # スクロールバーとText ウィジェットを接続
        self.text_widget.config(yscrollcommand=self.v_scrollbar.set, xscrollcommand=self.h_scrollbar.set)

        # gridでウィジェットを配置
        self.text_widget.grid(row=0, column=0, sticky="nsew")
        self.v_scrollbar.grid(row=0, column=1, sticky="ns")
        self.h_scrollbar.grid(row=1, column=0, sticky="ew")

        # サンプルテキストを追加
        sample_text = ""
        for i in range(1, 101):
            sample_text += f"行 {i}: これは非常に長いサンプルテキストです。ウィンドウの幅を超えることで、水平スクロールバーの必要性を示します。\\n"
        self.text_widget.insert(1.0, sample_text)

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

## 主要なオプション

オプション	説明
<code>orient</code>	スクロールバーの向き ( <code>tk.VERTICAL</code> または <code>tk.HORIZONTAL</code> )。
<code>command</code>	スクロール時に実行される関数。通常は対象ウィジェットの <code>yview</code> または <code>xview</code> メソッド。
<code>width</code>	スクロールバーの幅（垂直）または高さ（水平）をピクセルで指定。
<code>bg</code> (または <code>background</code> )	背景色。
<code>troughcolor</code>	スクロールバーの溝の色。
<code>relief</code>	境界線のスタイル ( <code>flat</code> , <code>raised</code> , <code>sunken</code> , <code>groove</code> , <code>ridge</code> )。
<code>borderwidth</code> (または <code>bd</code> )	境界線の幅。
<code>elementborderwidth</code>	スクロールバーの要素の境界線幅。
<code>jump</code>	スクロールバーをドラッグしたときの動作 ( <code>True</code> / <code>False</code> )。

## 主要なメソッド

メソッド	説明
<code>set(first, last)</code>	スクロールバーの位置を設定します。通常は対象ウィジェットから自動的に呼び出されます。
<code>get()</code>	現在のスクロール位置を取得します。

## 実用的な例

### 複数のスクロールバーを持つウィジェット

```

import tkinter as tk

class MultiScrollApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("複数スクロールバー")
        self.geometry("500x400")

        self.create_widgets()

    def create_widgets(self):
        # メインフレーム
        main_frame = tk.Frame(self)
        main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # Canvas用のフレーム
        canvas_frame = tk.Frame(main_frame)
        canvas_frame.pack(fill=tk.BOTH, expand=True)

        # Canvas ウィジェット
        self.canvas = tk.Canvas(canvas_frame, bg="white")

        # 垂直スクロールバー
        v_scrollbar = tk.Scrollbar(canvas_frame, orient=tk.VERTICAL, command=self.canvas.yview)
        self.canvas.configure(yscrollcommand=v_scrollbar.set)

        # 水平スクロールバー
        h_scrollbar = tk.Scrollbar(canvas_frame, orient=tk.HORIZONTAL, command=self.canvas.xview)
        self.canvas.configure(xscrollcommand=h_scrollbar.set)

        # スクロールバーとキャンバスを配置
        self.canvas.grid(row=0, column=0, sticky="nsew")
        v_scrollbar.grid(row=0, column=1, sticky="ns")
        h_scrollbar.grid(row=1, column=0, sticky="ew")

        # Grid の重みを設定
        canvas_frame.grid_rowconfigure(0, weight=1)
        canvas_frame.grid_columnconfigure(0, weight=1)

        # Canvas に大きなコンテンツを描画
        self.draw_content()

        # スクロール領域を設定
        self.canvas.configure(scrollregion=self.canvas.bbox("all"))

        # マウスホイールでのスクロールを有効化
        self.canvas.bind("<MouseWheel>", self.on_mouse_wheel)
        self.canvas.bind("<Button-4>", self.on_mouse_wheel)
        self.canvas.bind("<Button-5>", self.on_mouse_wheel)
        self.canvas.focus_set()

    def draw_content(self):
        # グリッド状にコンテンツを描画
        for i in range(50):
            for j in range(30):
                x1, y1 = j * 60 + 10, i * 40 + 10
                x2, y2 = x1 + 50, y1 + 30

                # 色をランダムに選択
                import random
                colors = ["lightblue", "lightgreen", "lightcoral", "lightyellow", "lightpink"]
                color = random.choice(colors)

                # 矩形を描画
                self.canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="black")

                # テキストを描画
                self.canvas.create_text((x1 + x2) / 2, (y1 + y2) / 2, text=".")


```

```

        self.canvas.create_text(x1 + 25, y1 + 15, text=f"{i},{j}", font=("Arial", 8))

def on_mouse_wheel(self, event):
    # マウスホイールでの垂直スクロール
    if event.delta:
        self.canvas.yview_scroll(int(-1 * (event.delta / 120)), "units")
    elif event.num == 4:
        self.canvas.yview_scroll(-1, "units")
    elif event.num == 5:
        self.canvas.yview_scroll(1, "units")

if __name__ == "__main__":
    app = MultiScrollApp()
    app.mainloop()

```

## Listbox とのスクロールバー連携

```

import tkinter as tk

class ScrollableListApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("スクロール可能リストボックス")
        self.geometry("400x350")

        self.create_widgets()

    def create_widgets(self):
        # タイトル
        tk.Label(self, text="大量データリスト", font=("Arial", 14, "bold")).pack(pady=10)

        # リスト操作ボタン
        button_frame = tk.Frame(self)
        button_frame.pack(pady=5)

        tk.Button(button_frame, text="先頭へ", command=self.go_to_top).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="末尾へ", command=self.go_to_bottom).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="中央へ", command=self.go_to_middle).pack(side=tk.LEFT, padx=5)

        # リストボックス用のフレーム
        list_frame = tk.Frame(self)
        list_frame.pack(fill=tk.BOTH, expand=True, padx=20, pady=10)

        # リストボックスとスクロールバー
        self.listbox = tk.Listbox(list_frame, font=("Arial", 10))
        scrollbar = tk.Scrollbar(list_frame, orient=tk.VERTICAL)

        # スクロールバーとリストボックスを接続
        self.listbox.config(yscrollcommand=scrollbar.set)
        scrollbar.config(command=self.listbox.yview)

        # 配置
        self.listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        # 大量のデータを追加
        self.populate_list()

        # 選択イベント
        self.listbox.bind("<<ListboxSelect>>", self.on_selection)

        # 情報表示
        self.info_label = tk.Label(self, text="項目を選択してください", font=("Arial", 10))
        self.info_label.pack(pady=5)

        # スクロール位置表示
        self.scroll_label = tk.Label(self, text="スクロール位置: 先頭", font=("Arial", 9), fg="gray")
        self.scroll_label.pack()

        # スクロール位置の監視
        self.monitor_scroll_position()

    def populate_list(self):
        # 1000個の項目を追加
        categories = ["ファイル", "ドキュメント", "画像", "音楽", "動画", "データ"]
        import random

        for i in range(1, 1001):
            category = random.choice(categories)
            size = random.randint(1, 999)
            unit = random.choice(["KB", "MB", "GB"])
            self.listbox.insert(tk.END, f"{i:04d}: {category}_{i:04d}.ext ({size}{unit})")

    def on_selection(self, event):
        selection = self.listbox.curselection()
        if selection:
            index = selection[0]
            value = self.listbox.get(index)
            self.info_label.config(text=f"選択: {value}")

    def go_to_top(self):
        self.listbox.see(0)
        self.listbox.selection_clear(0, tk.END)
        self.listbox.selection_set(0)
        self.listbox.activate(0)

    def go_to_bottom(self):
        last_index = self.listbox.size() - 1
        self.listbox.see(last_index)
        self.listbox.selection_clear(0, tk.END)
        self.listbox.selection_set(last_index)
        self.listbox.activate(last_index)

```

```

def go_to_middle(self):
    middle_index = self.listbox.size() // 2
    self.listbox.see(middle_index)
    self.listbox.selection_clear(0, tk.END)
    self.listbox.selection_set(middle_index)
    self.listbox.activate(middle_index)

def monitor_scroll_position(self):
    # スクロール位置を監視
    try:
        first_visible = self.listbox.nearest(0)
        total_items = self.listbox.size()
        percentage = int((first_visible / total_items) * 100) if total_items > 0 else 0

        self.scroll_label.config(text=f"スクロール位置: {percentage}% (項目 {first_visible + 1}/{total_items})")
    except:
        pass

    # 100ms後に再実行
    self.after(100, self.monitor_scroll_position)

if __name__ == "__main__":
    app = ScrollableListApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
適切な接続	<code>yscrollcommand</code> と <code>command</code> を適切に設定して、スクロールバーと対象ウィジェットを正しく接続します。
配置の工夫	<code>pack</code> または <code>grid</code> を使用してスクロールバーと対象ウィジェットを適切に配置します。
スクロール領域の設定	<code>Canvas</code> を使用する場合は、 <code>scrollregion</code> を適切に設定します。
マウスホイール対応	ユーザビリティ向上のため、マウスホイールでのスクロールを実装します。
双方向スクロール	必要に応じて垂直と水平の両方のスクロールバーを提供します。

## 対応ウィジェット

ウィジェット	垂直スクロール	水平スクロール	備考
<code>Text</code>	✓	✓	<code>yscrollcommand</code> , <code>xscrollcommand</code>
<code>Listbox</code>	✓	✓	<code>yscrollcommand</code> , <code>xscrollcommand</code>
<code>Canvas</code>	✓	✓	<code>yscrollcommand</code> , <code>xscrollcommand</code>
<code>Entry</code>	-	✓	<code>xscrollcommand</code> のみ

## 参考リンク

- [Python Docs - tkinter.Scrollbar](#)
- [TkDocs - Scrollbar](#)

# Menu リファレンス

メニューバーやコンテキストメニューを作成する `Menu` ウィジェットについての詳細なリファレンスです。

## 概要

`Menu` ウィジェットは、アプリケーションにメニューバー、プルダウンメニュー、ポップアップメニュー（コンテキストメニュー）を提供するためのコントロールです。階層構造のメニューを作成でき、キーボードショートカット、アクセラレータキー、チェックメニュー、ラジオメニューなどの機能をサポートします。

## 基本的な使用方法

### シンプルなメニューバー

```
import tkinter as tk
from tkinter import messagebox

def new_file():
    messagebox.showinfo("新規", "新しいファイルを作成します")

def open_file():
    messagebox.showinfo("開く", "ファイルを開きます")

def exit_app():
    app.quit()

app = tk.Tk()
app.title("Menuの例")
app.geometry("400x300")

# メニューを作成
menubar = tk.Menu(app)
app.config(menu=menubar)

# ファイルメニューを作成
file_menu = tk.Menu(menubar, tearoff=0)
menubar.add_cascade(label="ファイル", menu=file_menu)
file_menu.add_command(label="新規", command=new_file)
file_menu.add_command(label="開く", command=open_file)
file_menu.add_separator()
file_menu.add_command(label="終了", command=exit_app)

app.mainloop()
```

### クラスベースでのメニュー

```
import tkinter as tk
from tkinter import messagebox

class MenuApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Menuの例（クラスベース）")
        self.geometry("400x300")

        self.create_menu()

    def create_menu(self):
        # メニューバーを作成
        self.menubar = tk.Menu(self)
        self.config(menu=self.menubar)

        # ファイルメニューを作成
        file_menu = tk.Menu(self.menubar, tearoff=0)
        self.menubar.add_cascade(label="ファイル", menu=file_menu)
        file_menu.add_command(label="新規", command=self.new_file)
        file_menu.add_command(label="開く", command=self.open_file)
        file_menu.add_separator()
        file_menu.add_command(label="終了", command=self.quit)

        # 編集メニューを作成
        edit_menu = tk.Menu(self.menubar, tearoff=0)
        self.menubar.add_cascade(label="編集", menu=edit_menu)
        edit_menu.add_command(label="元に戻す", command=self.undo)
        edit_menu.add_command(label="やり直し", command=self.redo)

    def new_file(self):
        messagebox.showinfo("新規", "新しいファイルを作成します")

    def open_file(self):
        messagebox.showinfo("開く", "ファイルを開きます")
```

```

def undo(self):
    messagebox.showinfo("元に戻す", "操作を元に戻します")

def redo(self):
    messagebox.showinfo("やり直し", "操作をやり直します")

if __name__ == "__main__":
    app = MenuApp()
    app.mainloop()

```

## 主要なメソッド

メソッド	説明
add_command(label, command, ...)	コマンドメニュー項目を追加します。
add_cascade(label, menu, ...)	サブメニューを追加します。
add_checkbutton(label, variable, ...)	チェックボックス型のメニュー項目を追加します。
add_radiobutton(label, variable, value, ...)	ラジオボタン型のメニュー項目を追加します。
add_separator()	メニューに区切り線を追加します。
delete(index1, index2=None)	指定したメニュー項目を削除します。
insert_command(index, label, command, ...)	指定位置にコマンドメニューを挿入します。
entryconfig(index, **options)	メニュー項目の設定を変更します。

## 主要なオプション

### コマンドメニューのオプション

オプション	説明
label	メニュー項目に表示するテキスト。
command	メニュー項目が選択されたときに実行される関数。
accelerator	ショートカットキーの表示文字列（実際の機能は別途実装が必要）。
underline	アクセキーとして使用する文字のインデックス。
state	メニュー項目の状態（normal, active, disabled）。
font	フォント。
foreground	テキストの色。
background	背景色。

### その他のオプション

オプション	説明
tearoff	メニューをウィンドウから切り離し可能にするかどうか（0 または 1）。

## 実用的な例

### 完全なメニューシステム

```

import tkinter as tk
from tkinter import messagebox, filedialog

class FullMenuApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("完全なメニューシステム")
        self.geometry("600x400")

        # 状態変数
        self.word_wrap = tk.BooleanVar(value=True)
        self.view_mode = tk.StringVar(value="normal")

        self.create_widgets()
        self.create_menu()
        self.create_context_menu()

        # キーボードショートカットをバインド
        self.bind_shortcuts()

    def create_widgets(self):
        # メインテキストエリア
        self.text_area = tk.Text(self, wrap=tk.WORD, font=("Consolas", 11))
        self.text_area.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

```

```

# ステータスバー
self.status_bar = tk.Label(
    self,
    text="準備完了",
    relief=tk.SUNKEN,
    anchor=tk.W,
    bg="lightgray"
)
self.status_bar.pack(side=tk.BOTTOM, fill=tk.X)

def create_menu(self):
    # メニューバー
    self.menubar = tk.Menu(self)
    self.config(menu=self.menubar)

    # ファイルメニュー
    file_menu = tk.Menu(self.menubar, tearoff=0)
    self.menubar.add_cascade(label="ファイル", menu=file_menu)
    file_menu.add_command(label="新規", command=self.new_file, accelerator="Ctrl+N")
    file_menu.add_command(label="開く", command=self.open_file, accelerator="Ctrl+O")
    file_menu.add_command(label="保存", command=self.save_file, accelerator="Ctrl+S")
    file_menu.add_command(label="名前を付けて保存", command=self.save_as_file)
    file_menu.add_separator()
    file_menu.add_command(label="終了", command=self.quit, accelerator="Ctrl+Q")

    # 編集メニュー
    edit_menu = tk.Menu(self.menubar, tearoff=0)
    self.menubar.add_cascade(label="編集", menu=edit_menu)
    edit_menu.add_command(label="元に戻す", command=self.undo, accelerator="Ctrl+Z")
    edit_menu.add_command(label="やり直し", command=self.redo, accelerator="Ctrl+Y")
    edit_menu.add_separator()
    edit_menu.add_command(label="切り取り", command=self.cut, accelerator="Ctrl+X")
    edit_menu.add_command(label="コピー", command=self.copy, accelerator="Ctrl+C")
    edit_menu.add_command(label="貼り付け", command=self.paste, accelerator="Ctrl+V")
    edit_menu.add_separator()
    edit_menu.add_command(label="すべて選択", command=self.select_all, accelerator="Ctrl+A")
    edit_menu.add_command(label="検索", command=self.find, accelerator="Ctrl+F")

    # 表示メニュー
    view_menu = tk.Menu(self.menubar, tearoff=0)
    self.menubar.add_cascade(label="表示", menu=view_menu)

    # 折り返しのチェックメニュー
    view_menu.add_checkbutton(
        label="行の折り返し",
        variable=self.word_wrap,
        command=self.toggle_word_wrap
    )
    view_menu.add_separator()

    # 表示モードのラジオメニュー
    view_menu.add_radiobutton(
        label="通常表示",
        variable=self.view_mode,
        value="normal",
        command=self.change_view_mode
    )
    view_menu.add_radiobutton(
        label="フルスクリーン",
        variable=self.view_mode,
        value="fullscreen",
        command=self.change_view_mode
    )

    # ツールメニュー
    tools_menu = tk.Menu(self.menubar, tearoff=0)
    self.menubar.add_cascade(label="ツール", menu=tools_menu)
    tools_menu.add_command(label="文字数カウント", command=self.count_characters)
    tools_menu.add_command(label="大文字に変換", command=self.to_uppercase)
    tools_menu.add_command(label="小文字に変換", command=self.to_lowercase)

    # ヘルプメニュー
    help_menu = tk.Menu(self.menubar, tearoff=0)
    self.menubar.add_cascade(label="ヘルプ", menu=help_menu)
    help_menu.add_command(label="使い方", command=self.show_help)
    help_menu.add_command(label="バージョン情報", command=self.show_about)

def create_context_menu(self):
    # コンテキストメニュー (右クリックメニュー)
    self.context_menu = tk.Menu(self, tearoff=0)
    self.context_menu.add_command(label="切り取り", command=self.cut)
    self.context_menu.add_command(label="コピー", command=self.copy)
    self.context_menu.add_command(label="貼り付け", command=self.paste)
    self.context_menu.add_separator()
    self.context_menu.add_command(label="すべて選択", command=self.select_all)

    # テキストエリアに右クリックイベントをバインド
    self.text_area.bind("<Button-3>", self.show_context_menu) # Windows/Linux
    self.text_area.bind("<Button-2>", self.show_context_menu) # macOS

def bind_shortcuts(self):
    # キーボードショートカット
    self.bind("<Control-n>", lambda e: self.new_file())
    self.bind("<Control-o>", lambda e: self.open_file())
    self.bind("<Control-s>", lambda e: self.save_file())
    self.bind("<Control-q>", lambda e: self.quit())
    self.bind("<Control-z>", lambda e: self.undo())
    self.bind("<Control-y>", lambda e: self.redo())
    self.bind("<Control-x>", lambda e: self.cut())
    self.bind("<Control-c>", lambda e: self.copy())

```

```

    self.bind("<Control-v>", lambda e: self.paste())
    self.bind("<Control-a>", lambda e: self.select_all())
    self.bind("<Control-f>", lambda e: self.find())

    def show_context_menu(self, event):
        # コンテキストメニューを表示
        try:
            self.context_menu.tk_popup(event.x_root, event.y_root)
        finally:
            self.context_menu.grab_release()

    # ファイル操作
    def new_file(self):
        self.text_area.delete(1.0, tk.END)
        self.status_bar.config(text="新しいファイルを作成しました")

    def open_file(self):
        filename = filedialog.askopenfilename(
            title="ファイルを開く",
            filetypes=[("テキストファイル", "*.txt"), ("すべてのファイル", "*.*")]
        )
        if filename:
            try:
                with open(filename, 'r', encoding='utf-8') as file:
                    content = file.read()
                    self.text_area.delete(1.0, tk.END)
                    self.text_area.insert(1.0, content)
                    self.status_bar.config(text=f"ファイルを開きました: {filename}")
            except Exception as e:
                messagebox.showerror("エラー", f"ファイルを開けませんでした:\n{e}")

    def save_file(self):
        filename = filedialog.asksaveasfilename(
            title="ファイルを保存",
            defaultextension=".txt",
            filetypes=[("テキストファイル", "*.txt"), ("すべてのファイル", "*.*")]
        )
        if filename:
            try:
                content = self.text_area.get(1.0, tk.END)
                with open(filename, 'w', encoding='utf-8') as file:
                    file.write(content)
                    self.status_bar.config(text=f"ファイルを保存しました: {filename}")
            except Exception as e:
                messagebox.showerror("エラー", f"ファイルを保存できませんでした:\n{e}")

    def save_as_file(self):
        self.save_file()

    # 編集操作
    def undo(self):
        try:
            self.text_area.edit_undo()
            self.status_bar.config(text="操作を元に戻しました")
        except tk.TclError:
            self.status_bar.config(text="元に戻す操作がありません")

    def redo(self):
        try:
            self.text_area.edit_redo()
            self.status_bar.config(text="操作をやり直しました")
        except tk.TclError:
            self.status_bar.config(text="やり直す操作がありません")

    def cut(self):
        self.text_area.event_generate("<<Cut>>")
        self.status_bar.config(text="切り取りました")

    def copy(self):
        self.text_area.event_generate("<<Copy>>")
        self.status_bar.config(text="コピーしました")

    def paste(self):
        self.text_area.event_generate("<<Paste>>")
        self.status_bar.config(text="貼り付けました")

    def select_all(self):
        self.text_area.tag_add(tk.SEL, "1.0", tk.END)
        self.status_bar.config(text="すべて選択しました")

    def find(self):
        messagebox.showinfo("検索", "検索機能 (未実装)")

    # 表示操作
    def toggle_word_wrap(self):
        if self.word_wrap.get():
            self.text_area.config(wrap=tk.WORD)
            self.status_bar.config(text="行の折り返しを有効にしました")
        else:
            self.text_area.config(wrap=tk.NONE)
            self.status_bar.config(text="行の折り返しを無効にしました")

    def change_view_mode(self):
        mode = self.view_mode.get()
        if mode == "fullscreen":
            self.attributes('-fullscreen', True)
            self.status_bar.config(text="フルスクリーンモードに切り替えました")
        else:
            self.attributes('-fullscreen', False)
            self.status_bar.config(text="通常表示に切り替えました")

```

```

# ツール操作
def count_characters(self):
    content = self.text_area.get(1.0, tk.END)
    char_count = len(content) - 1 # 末尾の改行を除く
    word_count = len(content.split())
    line_count = content.count('\n')

    messagebox.showinfo(
        "文字数カウント",
        f"文字数: {char_count}\n単語数: {word_count}\n行数: {line_count}"
    )

def to_uppercase(self):
    try:
        selected_text = self.text_area.get(tk.SEL_FIRST, tk.SEL_LAST)
        self.text_area.delete(tk.SEL_FIRST, tk.SEL_LAST)
        self.text_area.insert(tk.INSERT, selected_text.upper())
        self.status_bar.config(text="大文字に変換しました")
    except tk.TclError:
        messagebox.showwarning("警告", "テキストを選択してください")

def to_lowercase(self):
    try:
        selected_text = self.text_area.get(tk.SEL_FIRST, tk.SEL_LAST)
        self.text_area.delete(tk.SEL_FIRST, tk.SEL_LAST)
        self.text_area.insert(tk.INSERT, selected_text.lower())
        self.status_bar.config(text="小文字に変換しました")
    except tk.TclError:
        messagebox.showwarning("警告", "テキストを選択してください")

# ヘルプ操作
def show_help(self):
    help_text = """
tkinter テキストエディタ

使い方:
- ファイルメニューから新規作成、開く、保存ができます
- 編集メニューから基本的な編集操作ができます
- 表示メニューで表示設定を変更できます
- ツールメニューで便利な機能を使用できます

キーボードショートカット:
- Ctrl+N: 新規
- Ctrl+O: 開く
- Ctrl+S: 保存
- Ctrl+Z: 元に戻す
- Ctrl+Y: やり直し
- Ctrl+X: 切り取り
- Ctrl+C: コピー
- Ctrl+V: 貼り付け
- Ctrl+A: すべて選択
"""

    messagebox.showinfo("使い方", help_text)

def show_about(self):
    messagebox.showinfo(
        "バージョン情報",
        "tkinter テキストエディタ\nバージョン 1.0\nPython tkinter で作成"
    )

if __name__ == "__main__":
    app = FullMenuApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
tearoff=0 の使用	<code>tearoff=0</code> を設定してメニューの切り離し機能を無効にします（現代的なUIでは一般的）。
適切な区切り	<code>add_separator()</code> を使用して関連するメニュー項目をグループ分けします。
キーボードショートカット	<code>accelerator</code> オプションでショートカットを表示し、実際のキーバインドも実装します。
コンテキストメニュー	右クリックメニューを提供してユーザビリティを向上させます。
状態管理	<code>state</code> オプションを使用してメニュー項目の有効/無効を適切に管理します。

## 参考リンク

- [Python Docs - tkinter.Menu](#)
- [TkDocs - Menu](#)

## Messagebox リファレンス

メッセージダイアログを表示する `tkinter.messagebox` モジュールについての詳細なリファレンスです。

### 概要

`tkinter.messagebox` モジュールは、ユーザーに情報を表示したり、確認を求めたり、警告を通知するためのダイアログボックスを提供します。インフォメーション、警告、エラー、質問など、様々な種類のメッセージダイアログを簡単に表示できます。

### 基本的な使用方法

#### シンプルなメッセージダイアログ

```
import tkinter as tk
from tkinter import messagebox

def show_info():
    messagebox.showinfo("情報", "これは情報メッセージです。")

def show_warning():
    messagebox.showwarning("警告", "これは警告メッセージです。")

def show_error():
    messagebox.showerror("エラー", "これはエラーメッセージです。")

app = tk.Tk()
app.title("MessageBoxの例")
app.geometry("300x200")

tk.Button(app, text="情報を表示", command=show_info).pack(pady=10)
tk.Button(app, text="警告を表示", command=show_warning).pack(pady=10)
tk.Button(app, text="エラーを表示", command=show_error).pack(pady=10)

app.mainloop()
```

#### クラスベースでのメッセージダイアログ

```
import tkinter as tk
from tkinter import messagebox

class MessageboxApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("MessageBoxの例（クラスベース）")
        self.geometry("300x350")

        self.create_widgets()

    def create_widgets(self):
        tk.Button(self, text="情報を表示", command=self.show_info).pack(pady=10)
        tk.Button(self, text="警告を表示", command=self.show_warning).pack(pady=10)
        tk.Button(self, text="エラーを表示", command=self.show_error).pack(pady=10)
        tk.Button(self, text="確認ダイアログ", command=self.show_question).pack(pady=10)

    def show_info(self):
        messagebox.showinfo("情報", "これは情報メッセージです。")

    def show_warning(self):
        messagebox.showwarning("警告", "これは警告メッセージです。")

    def show_error(self):
        messagebox.showerror("エラー", "これはエラーメッセージです。")

    def show_question(self):
        result = messagebox.askyesno("確認", "本当に実行しますか？")
        if result:
            messagebox.showinfo("結果", "「はい」が選択されました。")
        else:
            messagebox.showinfo("結果", "「いいえ」が選択されました。")

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

### 主要な関数

#### 情報表示系

関数	説明	戻り値
<code>showinfo(title, message, **options)</code>	情報アイコン付きのメッセージを表示します。	'ok'
<code>showwarning(title, message, **options)</code>	警告アイコン付きのメッセージを表示します。	'ok'
<code>showerror(title, message, **options)</code>	エラーアイコン付きのメッセージを表示します。	'ok'

## 質問・確認系

関数	説明	戻り値
<code>askquestion(title, message, **options)</code>	「はい」「いいえ」ボタン付きの質問ダイアログを表示します。	'yes' または 'no'
<code>askyesno(title, message, **options)</code>	「はい」「いいえ」ボタン付きの確認ダイアログを表示します。	True または False
<code>askokcancel(title, message, **options)</code>	「OK」「キャンセル」ボタン付きの確認ダイアログを表示します。	True または False
<code>askretrycancel(title, message, **options)</code>	「再試行」「キャンセル」ボタン付きのダイアログを表示します。	True または False
<code>askyesnocancel(title, message, **options)</code>	「はい」「いいえ」「キャンセル」ボタン付きのダイアログを表示します。	True, False, または None

## オプションパラメータ

パラメータ	説明
<code>parent</code>	親ウィンドウを指定します。ダイアログがモーダルになります。
<code>icon</code>	アイコンの種類を指定します ( <code>error, info, question, warning</code> )。
<code>type</code>	ボタンの種類を指定します ( <code>abortretryignore, ok, okcancel, retrycancel, yesno, yesnocancel</code> )。
<code>default</code>	デフォルトで選択されるボタンを指定します。

## 実用的な例

### ファイル操作の確認ダイアログ

```

import tkinter as tk
from tkinter import messagebox, filedialog
import os

class FileManagerApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("ファイルマネージャー")
        self.geometry("400x300")

        self.current_file = None
        self.create_widgets()

    def create_widgets(self):
        # ファイル操作ボタン
        button_frame = tk.Frame(self)
        button_frame.pack(pady=20)

        tk.Button(button_frame, text="新規作成", command=self.new_file, width=12).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="ファイルを開く", command=self.open_file, width=12).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="保存", command=self.save_file, width=12).pack(side=tk.LEFT, padx=5)

        # テキストエリア
        self.text_area = tk.Text(self, wrap=tk.WORD, font=("Consolas", 11))
        self.text_area.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # 危険な操作ボタン
        danger_frame = tk.Frame(self)
        danger_frame.pack(pady=10)

        tk.Button(danger_frame, text="全て削除", command=self.clear_all, bg="red", fg="white", width=12).pack(side=tk.LEFT, padx=5)
        tk.Button(danger_frame, text="ファイル削除", command=self.delete_file, bg="darkred", fg="white", width=12).pack(side=tk.LEFT, padx=5)
        tk.Button(danger_frame, text="アプリ終了", command=self.quit_app, width=12).pack(side=tk.LEFT, padx=5)

        # ステータス
        self.status_label = tk.Label(self, text="準備完了", relief=tk.SUNKEN, anchor=tk.W)
        self.status_label.pack(side=tk.BOTTOM, fill=tk.X)

    def new_file(self):
        if self.check_unsaved_changes():
            self.text_area.delete(1.0, tk.END)
            self.current_file = None
            self.status_label.config(text="新しいファイルを作成しました")
            messagebox.showinfo("成功", "新しいファイルを作成しました。")

    def open_file(self):
        if self.check_unsaved_changes():
            filename = filedialog.askopenfilename(
                title="ファイルを開く",
                filetypes=[("テキストファイル", "*.txt"), ("すべてのファイル", "*.*")]
            )
            if filename:
                try:
                    with open(filename, 'r', encoding='utf-8') as file:
                        content = file.read()
                    self.text_area.delete(1.0, tk.END)
                    self.text_area.insert(1.0, content)
                    self.current_file = filename
                    self.status_label.config(text=f"ファイルを開きました: {os.path.basename(filename)}")
                    messagebox.showinfo("成功", f"ファイルを開きました:\n{os.path.basename(filename)}")
                except Exception as e:
                    messagebox.showerror("エラー", f"ファイルを開けませんでした:\n{str(e)}")

    def save_file(self):
        if not self.current_file:
            filename = filedialog.asksaveasfilename(
                title="ファイルを保存",
                defaultextension=".txt",
                filetypes=[("テキストファイル", "*.txt"), ("すべてのファイル", "*.*")]
            )
            if not filename:
                return
            else:
                self.current_file = filename
                self.status_label.config(text=f"ファイルを保存しました: {os.path.basename(filename)}")
                messagebox.showinfo("成功", f"ファイルを保存しました:\n{os.path.basename(filename)}")
        else:
            with open(self.current_file, 'w', encoding='utf-8') as file:
                content = self.text_area.get(1.0, tk.END)
                file.write(content)
            self.status_label.config(text=f"ファイルを保存しました: {os.path.basename(self.current_file)}")
            messagebox.showinfo("成功", f"ファイルを保存しました:\n{os.path.basename(self.current_file)}")

```

```

        return
    self.current_file = filename

    try:
        content = self.text_area.get(1.0, tk.END)
        with open(self.current_file, 'w', encoding='utf-8') as file:
            file.write(content)
        self.status_label.config(text=f"ファイルを保存しました: {os.path.basename(self.current_file)}")
        messagebox.showinfo("成功", f"ファイルを保存しました:\n{os.path.basename(self.current_file)}")
    except Exception as e:
        messagebox.showerror("エラー", f"ファイルを保存できませんでした:\n{str(e)}")

def clear_all(self):
    # 複数段階の確認
    if messagebox.askokcancel("確認", "本当にすべてのテキストを削除しますか?"):
        if messagebox.askyesno("最終確認", "この操作は元に戻せません。本当に削除しますか?", icon="warning"):
            self.text_area.delete(1.0, tk.END)
            self.status_label.config(text="すべてのテキストを削除しました")
            messagebox.showinfo("完了", "すべてのテキストを削除しました。")

def delete_file(self):
    if not self.current_file:
        messagebox.showwarning("警告", "削除するファイルが選択されていません。")
        return

    filename = os.path.basename(self.current_file)

    # 危険な操作の確認
    result = messagebox.askyesnocancel(
        "ファイル削除",
        f"ファイル '{filename}' を完全に削除しますか?\n\nこの操作は元に戻せません。",
        icon="warning"
    )

    if result is True: # 「はい」が選択された
        try:
            os.remove(self.current_file)
            self.text_area.delete(1.0, tk.END)
            self.current_file = None
            self.status_label.config(text=f"ファイル '{filename}' を削除しました")
            messagebox.showinfo("削除完了", f"ファイル '{filename}' を削除しました。")
        except Exception as e:
            messagebox.showerror("エラー", f"ファイルを削除できませんでした:\n{str(e)}")
    elif result is False: # 「いいえ」が選択された
        messagebox.showinfo("キャンセル", "ファイル削除をキャンセルしました。")
    # result が None の場合（「キャンセル」）は何もしない

def quit_app(self):
    if self.check_unsaved_changes():
        if messagebox.askokcancel("終了確認", "アプリケーションを終了しますか?"):
            self.quit()

def check_unsaved_changes(self):
    # 実際の実装では、変更の有無をチェックする
    content = self.text_area.get(1.0, tk.END)
    if content.strip(): # 何かテキストがある場合
        result = messagebox.askyesnocancel(
            "未保存の変更",
            "変更が保存されていません。保存しますか?"
        )

        if result is True: # 保存する
            self.save_file()
            return True
        elif result is False: # 保存しない
            return True
        else: # キャンセル
            return False
    return True

if __name__ == "__main__":
    app = FileManagerApp()
    app.mainloop()

```

## ベストプラクティス

プラクティス	説明
適切なダイアログ選択	目的に応じて適切な種類のダイアログを選択します（情報表示、確認、エラー通知など）。
わかりやすいメッセージ	ユーザーが理解しやすい明確で簡潔なメッセージを作成します。
適切なタイトル	ダイアログの目的を明確に示すタイトルを設定します。
重要な操作の確認	削除や終了など、重要な操作には確認ダイアログを表示します。
エラーハンドリング	例外が発生した場合は、適切なエラーメッセージを表示します。

## 参考リンク

- [Python Docs - tkinter.messagebox](#)
- [TkDocs - Message Boxes](#)

# Layout Manager リファレンス

ウィジェットの配置を管理する `pack`, `grid`, `place` のレイアウトマネージャーについての詳細なリファレンスです。

## 概要

tkinter では、ウィジェットをウィンドウ内に配置するために3つのレイアウトマネージャーが提供されています。それぞれ異なる配置方法と用途があり、適切な選択により効率的で美しいUIを作成できます。

## Pack レイアウトマネージャー

### 概要

`pack` は最もシンプルなレイアウトマネージャーで、ウィジェットを一方向（上下または左右）に順番に配置します。

### 基本的な使用方法

```
import tkinter as tk

app = tk.Tk()
app.title("Pack レイアウトの例")
app.geometry("300x200")

# 上から順番に配置
tk.Label(app, text="上部ラベル", bg="lightblue").pack(side=tk.TOP)
tk.Label(app, text="下部ラベル", bg="lightgreen").pack(side=tk.BOTTOM)
tk.Label(app, text="左側ラベル", bg="lightcoral").pack(side=tk.LEFT)
tk.Label(app, text="右側ラベル", bg="lightyellow").pack(side=tk.RIGHT)

app.mainloop()
```

### クラスベースでのPack使用

```
import tkinter as tk

class PackApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Pack レイアウトの例（クラスベース）")
        self.geometry("400x300")

        self.create_widgets()

    def create_widgets(self):
        # ヘッダー
        header = tk.Label(self, text="ヘッダー", bg="darkblue", fg="white", height=2)
        header.pack(side=tk.TOP, fill=tk.X)

        # フッター
        footer = tk.Label(self, text="フッター", bg="darkgray", fg="white", height=2)
        footer.pack(side=tk.BOTTOM, fill=tk.X)

        # サイドバー
        sidebar = tk.Label(self, text="サイドバー", bg="lightgray", width=15)
        sidebar.pack(side=tk.LEFT, fill=tk.Y)

        # メインコンテンツ
        main_content = tk.Label(self, text="メインコンテンツ", bg="white")
        main_content.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

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

### Pack の主要オプション

オプション	説明
<code>side</code>	配置する方向 ( <code>tk.TOP</code> , <code>tk.BOTTOM</code> , <code>tk.LEFT</code> , <code>tk.RIGHT</code> )。デフォルトは <code>tk.TOP</code> 。
<code>fill</code>	ウィジェットがスペースを埋める方向 ( <code>tk.X</code> , <code>tk.Y</code> , <code>tk.BOTH</code> )。
<code>expand</code>	親ウィンドウのサイズ変更時にウィジェットが拡張するかどうか ( <code>True</code> / <code>False</code> )。
<code>padx</code> , <code>pady</code>	ウィジェット周辺の余白をピクセルで指定。
<code>ipadx</code> , <code>ipady</code>	ウィジェット内部の余白をピクセルで指定。

オプション	説明
<code>anchor</code>	ウィジェットの配置位置 ( <code>n</code> , <code>s</code> , <code>e</code> , <code>w</code> , <code>center</code> など)。

## Grid レイアウトマネージャー

### 概要

`grid` は表形式でウィジェットを配置するレイアウトマネージャーで、最も柔軟で強力な配置が可能です。

### 基本的な使用方法

```
import tkinter as tk

app = tk.Tk()
app.title("Grid レイアウトの例")
app.geometry("300x200")

# 2x2のグリッドに配置
tk.Label(app, text="(0,0)", bg="lightblue").grid(row=0, column=0)
tk.Label(app, text="(0,1)", bg="lightgreen").grid(row=0, column=1)
tk.Label(app, text="(1,0)", bg="lightcoral").grid(row=1, column=0)
tk.Label(app, text="(1,1)", bg="lightyellow").grid(row=1, column=1)

app.mainloop()
```

### クラスベースでのGrid使用

```
import tkinter as tk

class GridApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Grid レイアウトの例（クラスベース）")
        self.geometry("400x300")

        self.create_widgets()
        self.configure_grid()

    def create_widgets(self):
        # ヘッダー（複数列にわたって配置）
        tk.Label(self, text="アプリケーションタイトル", bg="darkblue", fg="white", font=("Arial", 14)).grid(
            row=0, column=0, columnspan=3, sticky="ew", pady=5
        )

        # ラベル
        tk.Label(self, text="名前:", font=("Arial", 10)).grid(row=1, column=0, sticky="e", padx=5, pady=5)
        tk.Label(self, text="メール:", font=("Arial", 10)).grid(row=2, column=0, sticky="e", padx=5, pady=5)
        tk.Label(self, text="メッセージ:", font=("Arial", 10)).grid(row=3, column=0, sticky="ne", padx=5, pady=5)

        # 入力フィールド
        self.name_entry = tk.Entry(self, width=30)
        self.name_entry.grid(row=1, column=1, columnspan=2, sticky="ew", padx=5, pady=5)

        self.email_entry = tk.Entry(self, width=30)
        self.email_entry.grid(row=2, column=1, columnspan=2, sticky="ew", padx=5, pady=5)

        self.message_text = tk.Text(self, width=30, height=5)
        self.message_text.grid(row=3, column=1, columnspan=2, sticky="nsew", padx=5, pady=5)

        # ボタン
        tk.Button(self, text="送信", bg="green", fg="white").grid(row=4, column=1, sticky="e", padx=5, pady=10)
        tk.Button(self, text="クリア", bg="red", fg="white").grid(row=4, column=2, sticky="w", padx=5, pady=10)

    def configure_grid(self):
        # 列の重みを設定（リサイズ時の動作）
        self.grid_columnconfigure(1, weight=1)
        self.grid_columnconfigure(2, weight=1)

        # 行の重みを設定
        self.grid_rowconfigure(3, weight=1)

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

### Grid の主要オプション

オプション	説明
<code>row</code> , <code>column</code>	ウィジェットを配置するグリッドの行・列番号 (0から開始)。
<code code="" rowspan<="">, <code>columnspan</code></code>	ウィジェットが占める行・列の数。
<code>sticky</code>	セル内でのウィジェットの配置 ( <code>n</code> , <code>s</code> , <code>e</code> , <code>w</code> の組み合わせ)。
<code>padx</code> , <code>pady</code>	ウィジェット周辺の余白をピクセルで指定。

オプション	説明
<code>ipadx , ipady</code>	ウィジェット内部の余白をピクセルで指定。

## Grid の重み設定

メソッド	説明
<code>grid_rowconfigure(index, weight=N)</code>	指定行の重みを設定。ウィンドウリサイズ時の拡張率を決定。
<code>grid_columnconfigure(index, weight=N)</code>	指定列の重みを設定。ウィンドウリサイズ時の拡張率を決定。

## Place レイアウトマネージャー

### 概要

`place` は絶対座標や相対座標でウィジェットを配置するレイアウトマネージャーです。

### 基本的な使用方法

```
import tkinter as tk

app = tk.Tk()
app.title("Place レイアウトの例")
app.geometry("400x300")

# 絶対座標で配置
tk.Label(app, text="絶対座標", bg="lightblue").place(x=50, y=50)

# 相対座標で配置
tk.Label(app, text="中央", bg="lightgreen").place(relx=0.5, rely=0.5, anchor=tk.CENTER)

# 右下に配置
tk.Label(app, text="右下", bg="lightcoral").place(relx=1.0, rely=1.0, anchor=tk.SE)

app.mainloop()
```

### クラスベースでのPlace使用

```
import tkinter as tk

class PlaceApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Place レイアウトの例 (クラスベース) ")
        self.geometry("500x400")

        self.create_widgets()

    def create_widgets(self):
        # 背景画像の代わりとしてキャンバス
        self.canvas = tk.Canvas(self, bg="lightsteelblue")
        self.canvas.place(x=0, y=0, relwidth=1, relheight=1)

        # タイトル (上部中央)
        title_label = tk.Label(self, text="Place レイアウト デモ", font=("Arial", 18, "bold"), bg="white")
        title_label.place(relx=0.5, y=20, anchor=tk.N)

        # ログインフォーム (中央)
        form_frame = tk.Frame(self, bg="white", relief=tk.RAISED, bd=2)
        form_frame.place(relx=0.5, rely=0.5, anchor=tk.CENTER, width=300, height=200)

        tk.Label(form_frame, text="ユーザー名:", bg="white").place(x=20, y=40)
        username_entry = tk.Entry(form_frame, width=25)
        username_entry.place(x=100, y=40)

        tk.Label(form_frame, text="パスワード:", bg="white").place(x=20, y=80)
        password_entry = tk.Entry(form_frame, width=25, show="*")
        password_entry.place(x=100, y=80)

        login_button = tk.Button(form_frame, text="ログイン", bg="blue", fg="white")
        login_button.place(x=100, y=120)

        cancel_button = tk.Button(form_frame, text="キャンセル")
        cancel_button.place(x=180, y=120)

        # ステータス (下部)
        status_label = tk.Label(self, text="ログインしてください", bg="yellow")
        status_label.place(relx=0.5, rely=1.0, anchor=tk.S, y=-10)

        # バージョン情報 (右下)
        version_label = tk.Label(self, text="v1.0", font=("Arial", 8), fg="gray")
        version_label.place(relx=1.0, rely=1.0, anchor=tk.SE, x=-5, y=-5)

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

## Place の主要オプション

オプション	説明
x, y	絶対座標での位置をピクセルで指定。
relx, rely	相対座標での位置を0.0～1.0の範囲で指定。
width, height	ウィジェットのサイズをピクセルで指定。
relwidth, relheight	ウィジェットのサイズを親ウィンドウに対する比率(0.0～1.0)で指定。
anchor	ウィジェットのアンカーポイント(n, s, e, w, centerなど)。

## 実用的な例

### 複合レイアウトアプリケーション

```
import tkinter as tk
from tkinter import ttk

class ComplexLayoutApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("複合レイアウト デモ")
        self.geometry("800x600")

        self.create_layout()

    def create_layout(self):
        # メインコンテナ (Grid使用)
        self.grid_rowconfigure(1, weight=1)
        self.grid_columnconfigure(1, weight=1)

        # ヘッダー (Pack使用)
        self.create_header()

        # サイドバー (Grid + Pack使用)
        self.create_sidebar()

        # メインコンテンツエリア (Grid + Place使用)
        self.create_main_content()

        # ステータスバー (Pack使用)
        self.create_status_bar()

    def create_header(self):
        header_frame = tk.Frame(self, bg="darkblue", height=60)
        header_frame.grid(row=0, column=0, columnspan=2, sticky="ew")
        header_frame.grid_propagate(False)

        # ヘッダー内でPackを使用
        tk.Label(header_frame, text="複合レイアウト アプリケーション",
                bg="darkblue", fg="white", font=("Arial", 16, "bold")).pack(side=tk.LEFT, padx=20, pady=15)

        # ヘッダーボタン
        button_frame = tk.Frame(header_frame, bg="darkblue")
        button_frame.pack(side=tk.RIGHT, padx=20, pady=15)

        tk.Button(button_frame, text="設定", width=8).pack(side=tk.LEFT, padx=2)
        tk.Button(button_frame, text="ヘルプ", width=8).pack(side=tk.LEFT, padx=2)
        tk.Button(button_frame, text="終了", width=8, bg="red", fg="white").pack(side=tk.LEFT, padx=2)

    def create_sidebar(self):
        sidebar_frame = tk.Frame(self, bg="lightgray", width=200)
        sidebar_frame.grid(row=1, column=0, sticky="ns")
        sidebar_frame.grid_propagate(False)

        # サイドバー内でPackを使用
        tk.Label(sidebar_frame, text="ナビゲーション", bg="gray", fg="white",
                font=("Arial", 12, "bold")).pack(fill=tk.X, pady=(0, 10))

        # メニュー項目
        menu_items = ["ダッシュボード", "データ", "レポート", "設定", "ユーザー管理"]

        for item in menu_items:
            btn = tk.Button(sidebar_frame, text=item, width=20, anchor=tk.W)
            btn.pack(fill=tk.X, padx=10, pady=2)

        # サイドバー下部
        tk.Label(sidebar_frame, text="ステータス", bg="lightgray").pack(side=tk.BOTTOM, pady=10)

    def create_main_content(self):
        main_frame = tk.Frame(self, bg="white")
        main_frame.grid(row=1, column=1, sticky="nsew")

        # メインコンテンツをさらにGridで分割
        main_frame.grid_rowconfigure(1, weight=1)
        main_frame.grid_columnconfigure(0, weight=1)
        main_frame.grid_columnconfigure(1, weight=1)

        # ツールバー (Pack使用)
        toolbar_frame = tk.Frame(main_frame, bg="lightsteelblue", height=40)
```

```

toolbar_frame.grid(row=0, column=0, columnspan=2, sticky="ew")
toolbar_frame.grid_propagate(False)

tk.Button(toolbar_frame, text="新規").pack(side=tk.LEFT, padx=5, pady=5)
tk.Button(toolbar_frame, text="開く").pack(side=tk.LEFT, padx=5, pady=5)
tk.Button(toolbar_frame, text="保存").pack(side=tk.LEFT, padx=5, pady=5)

# 左パネル (データ表示)
left_panel = tk.Frame(main_frame, bg="white", relief=tk.SUNKEN, bd=1)
left_panel.grid(row=1, column=0, sticky="nsew", padx=5, pady=5)

tk.Label(left_panel, text="データビュー", font=("Arial", 12, "bold")).pack(pady=10)

# Treeview for data display
tree = ttk.Treeview(left_panel, columns=("col1", "col2"), show="tree headings")
tree.heading("#0", text="ID")
tree.heading("col1", text="名前")
tree.heading("col2", text="値")

# サンプルデータ
for i in range(10):
    tree.insert("", "end", text=str(i), values=(f"項目{i}", f"値{i}"))

tree.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

# 右パネル (詳細表示)
right_panel = tk.Frame(main_frame, bg="white", relief=tk.SUNKEN, bd=1)
right_panel.grid(row=1, column=1, sticky="nsew", padx=5, pady=5)

tk.Label(right_panel, text="詳細ビュー", font=("Arial", 12, "bold")).pack(pady=10)

# 詳細フォーム (Gridを使用)
detail_frame = tk.Frame(right_panel, bg="white")
detail_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

for i, label in enumerate(["名前:", "タイプ:", "値:", "説明:"]):
    tk.Label(detail_frame, text=label, bg="white").grid(row=i, column=0, sticky="e", padx=5, pady=5)
    if label == "説明:":
        tk.Text(detail_frame, height=3, width=25).grid(row=i, column=1, padx=5, pady=5)
    else:
        tk.Entry(detail_frame, width=25).grid(row=i, column=1, padx=5, pady=5)

# ボタンフレーム (Pack使用)
button_frame = tk.Frame(right_panel, bg="white")
button_frame.pack(side=tk.BOTTOM, pady=10)

tk.Button(button_frame, text="更新", bg="green", fg="white").pack(side=tk.LEFT, padx=5)
tk.Button(button_frame, text="削除", bg="red", fg="white").pack(side=tk.LEFT, padx=5)
tk.Button(button_frame, text="リセット").pack(side=tk.LEFT, padx=5)

# フローティングボタン (Place使用)
floating_btn = tk.Button(main_frame, text="+", font=("Arial", 20, "bold"),
                        bg="blue", fg="white", width=3, height=1)
floating_btn.place(relx=1.0, rely=1.0, anchor=tk.SE, x=-20, y=-20)

def create_status_bar(self):
    status_frame = tk.Frame(self, bg="lightgray", height=25)
    status_frame.grid(row=2, column=0, columnspan=2, sticky="ew")
    status_frame.grid_propagate(False)

    # ステータス情報 (Pack使用)
    tk.Label(status_frame, text="準備完了", bg="lightgray").pack(side=tk.LEFT, padx=10)
    tk.Label(status_frame, text="行: 1, 列: 1", bg="lightgray").pack(side=tk.RIGHT, padx=10)

    # プログレスバー
    progress = ttk.Progressbar(status_frame, length=100, mode='indeterminate')
    progress.pack(side=tk.RIGHT, padx=10)

if __name__ == "__main__":
    app = ComplexLayoutApp()
    app.mainloop()

```

## レイアウトマネージャーの選択指針

用途	推薦レイアウト	理由
シンプルな一方向配置	Pack	実装が簡単で、ツールバーやボタンの配置に適している
フォーム・表形式	Grid	正確な位置制御が可能で、複雑なレイアウトに対応
重複配置・絶対位置指定	Place	絶対座標での配置や、ウィジェットの重ね合わせが可能
複合レイアウト	組み合わせ	各部分に最適なレイアウトマネージャーを使い分ける

## ベストプラクティス

プラクティス	説明
一つのコンテナに一つのマネージャー	同じ親ウィンドウ内では一つのレイアウトマネージャーのみを使用します。
Frameで分割	複雑なレイアウトはFrameで分けし、それぞれに適切なマネージャーを使用します。
Grid の重み設定	Gridを使用する場合は、 <code>grid_rowconfigure</code> と <code>grid_columnconfigure</code> で適切な重みを設定します。
レスポンシブデザイン	<code>fill</code> と <code>expand</code> オプションを適切に使用してウィンドウリサイズに対応します。

プラクティス	説明
一貫性の保持	同じアプリケーション内では一貫したレイアウト手法を使用します。

## 参考リンク

- [Python Docs - tkinter Layout Management](#)
- [TkDocs - Layout Management](#)

# tkinter.Canvas リファレンス

## 概要

Canvas ウィジェットは、グラフィックス描画のためのウィジェットです。線、矩形、楕円、テキスト、画像などを描画でき、イベント処理やアニメーションも可能です。ゲーム開発、データ可視化、図形エディタなどの用途に適しています。

## 基本的な使用方法

```
import tkinter as tk

root = tk.Tk()
canvas = tk.Canvas(root, width=400, height=300, bg='white')
canvas.pack()

# 図形を描画
canvas.create_line(0, 0, 400, 300, fill='red', width=3)
canvas.create_rectangle(50, 50, 150, 100, fill='blue', outline='black')
canvas.create_oval(200, 100, 300, 200, fill='yellow')

root.mainloop()
```

## 主要なオプション

オプション	説明	デフォルト値	例
width	キャンバスの幅	400	width=500
height	キャンバスの高さ	300	height=400
bg	背景色	SystemButtonFace	bg='white'
scrollregion	スクロール可能な領域	None	scrollregion=(0,0,800,600)
highlightthickness	フォーカス時の枠線の太さ	2	highlightthickness=0
relief	境界線のスタイル	'flat'	relief='sunken'
borderwidth	境界線の幅	2	borderwidth=1
cursor	マウスカーソルの形状	"	cursor='crosshair'

## 主要なメソッド

メソッド	説明	例
create_line(x1,y1,x2,y2,**options)	線を描画	create_line(0,0,100,100,fill='red')
create_rectangle(x1,y1,x2,y2,**options)	矩形を描画	create_rectangle(10,10,90,60,fill='blue')
create_oval(x1,y1,x2,y2,**options)	楕円を描画	create_oval(20,20,80,60,fill='green')
create_polygon(x1,y1,x2,y2,...,**options)	多角形を描画	create_polygon(10,10,50,5,90,20,fill='red')
create_text(x,y,**options)	テキストを描画	create_text(50,50,text='Hello',font=('Arial',12))
create_image(x,y,**options)	画像を描画	create_image(50,50,image=photo)
create_arc(x1,y1,x2,y2,**options)	弧を描画	create_arc(10,10,90,90,start=0,extent=90)
delete(item_id)	アイテムを削除	delete(item_id)
coords(item_id, x1,y1,x2,y2,...)	アイテムの座標を変更	coords(rect_id, 20,20,100,80)
itemconfig(item_id, **options)	アイテムの設定を変更	itemconfig(rect_id, fill='red')
move(item_id, dx, dy)	アイテムを移動	move(rect_id, 10, 5)
find_overlapping(x1,y1,x2,y2)	指定領域と重複するアイテムを検索	find_overlapping(0,0,50,50)
find_closest(x, y)	指定座標に最も近いアイテムを検索	find_closest(event.x, event.y)
bbox(item_id)	アイテムの境界ボックスを取得	bbox(rect_id)
tag_bind(tag, event, callback)	タグにイベントをバインド	tag_bind('clickable', '<Button-1>', on_click)

## 実用的な例

### 1. 基本的な図形描画

```
import tkinter as tk

class DrawingApp:
    def __init__(self, root):
        self.root = root
        self.root.title('图形描画アプリ')

    # キャンバス作成
    self.canvas = tk.Canvas(root, width=500, height=400, bg='white')
    self.canvas.pack(padx=10, pady=10)

    # 各種图形を描画
    self.draw_shapes()

    def draw_shapes(self):
        # 線
        self.canvas.create_line(50, 50, 200, 100, fill='red', width=3)

        # 矩形
        self.canvas.create_rectangle(50, 120, 150, 200,
                                    fill='lightblue', outline='blue', width=2)

        # 楕円
        self.canvas.create_oval(200, 120, 300, 200,
                               fill='lightgreen', outline='green', width=2)

        # 多角形 (三角形)
        self.canvas.create_polygon(350, 200, 400, 120, 450, 200,
                                  fill='lightyellow', outline='orange', width=2)

        # テキスト
        self.canvas.create_text(250, 50, text='图形描画の例',
                               font=('Arial', 16, 'bold'), fill='darkblue')

root = tk.Tk()
app = DrawingApp(root)
root.mainloop()
```

### 2. インタラクティブな描画アプリ

```
import tkinter as tk

class InteractiveCanvas:
    def __init__(self, root):
        self.root = root
        self.root.title('インタラクティブキャンバス')

        # コントロールフレーム
        control_frame = tk.Frame(root)
        control_frame.pack(pady=5)

        # 描画モード選択
        self.mode = tk.StringVar(value='rectangle')
        tk.Label(control_frame, text='描画モード:').pack(side=tk.LEFT)
        modes = [('矩形', 'rectangle'), ('椭円', 'oval'), ('線', 'line')]
        for text, mode in modes:
            tk.Radiobutton(control_frame, text=text, variable=self.mode,
                           value=mode).pack(side=tk.LEFT)

        # 色選択
        self.color = tk.StringVar(value='blue')
        tk.Label(control_frame, text='色:').pack(side=tk.LEFT, padx=(20,0))
        colors = [('青', 'blue'), ('赤', 'red'), ('緑', 'green')]
        for text, color in colors:
            tk.Radiobutton(control_frame, text=text, variable=self.color,
                           value=color).pack(side=tk.LEFT)

        # クリアボタン
        tk.Button(control_frame, text='クリア',
                  command=self.clear_canvas).pack(side=tk.LEFT, padx=(20,0))

        # キャンバス
        self.canvas = tk.Canvas(root, width=600, height=400, bg='white',
                               relief=tk.SUNKEN, borderwidth=2)
        self.canvas.pack(padx=10, pady=10)

        # イベントバインド
        self.canvas.bind('<Button-1>', self.start_draw)
        self.canvas.bind('<B1-Motion>', self.draw_motion)
        self.canvas.bind('<ButtonRelease-1>', self.end_draw)

        self.start_x = self.start_y = 0
        self.current_item = None

    def start_draw(self, event):
        self.start_x, self.start_y = event.x, event.y
        mode = self.mode.get()
        color = self.color.get()
```

```

if mode == 'rectangle':
    self.current_item = self.canvas.create_rectangle(
        self.start_x, self.start_y, self.start_x, self.start_y,
        outline=color, width=2)
elif mode == 'oval':
    self.current_item = self.canvas.create_oval(
        self.start_x, self.start_y, self.start_x, self.start_y,
        outline=color, width=2)
elif mode == 'line':
    self.current_item = self.canvas.create_line(
        self.start_x, self.start_y, self.start_x, self.start_y,
        fill=color, width=2)

def draw_motion(self, event):
    if self.current_item:
        self.canvas.coords(self.current_item,
                           self.start_x, self.start_y, event.x, event.y)

def end_draw(self, event):
    self.current_item = None

def clear_canvas(self):
    self.canvas.delete('all')

root = tk.Tk()
app = InteractiveCanvas(root)
root.mainloop()

```

### 3. アニメーション例

```

import tkinter as tk
import math

class AnimationDemo:
    def __init__(self, root):
        self.root = root
        self.root.title('アニメーションデモ')

        # キャンバス
        self.canvas = tk.Canvas(root, width=500, height=300, bg='black')
        self.canvas.pack(padx=10, pady=10)

        # アニメーション用のオブジェクト
        self.ball = self.canvas.create_oval(10, 10, 30, 30, fill='red')
        self.x = 250
        self.y = 150
        self.angle = 0

        # コントロール
        control_frame = tk.Frame(root)
        control_frame.pack()

        self.is_running = False
        self.start_button = tk.Button(control_frame, text='開始',
                                      command=self.toggle_animation)
        self.start_button.pack(side=tk.LEFT, padx=5)

        tk.Button(control_frame, text='リセット',
                  command=self.reset_animation).pack(side=tk.LEFT, padx=5)

        # アニメーション開始
        self.animate()

    def animate(self):
        if self.is_running:
            # 円運動
            radius = 100
            self.x = 250 + radius * math.cos(self.angle)
            self.y = 150 + radius * math.sin(self.angle)
            self.angle += 0.1

            # ボールの位置を更新
            self.canvas.coords(self.ball, self.x-10, self.y-10,
                               self.x+10, self.y+10)

        # 次のフレームをスケジュール
        self.root.after(50, self.animate)

    def toggle_animation(self):
        self.is_running = not self.is_running
        self.start_button.config(text='停止' if self.is_running else '開始')

    def reset_animation(self):
        self.is_running = False
        self.angle = 0
        self.x = 250
        self.y = 150
        self.canvas.coords(self.ball, self.x-10, self.y-10, self.x+10, self.y+10)
        self.start_button.config(text='開始')

root = tk.Tk()
app = AnimationDemo(root)
root.mainloop()

```

## 4. 画像表示とスクロール

```
import tkinter as tk
from tkinter import filedialog
from PIL import Image, ImageTk

class ImageViewer:
    def __init__(self, root):
        self.root = root
        self.root.title('画像ビューア')

        # メニュー
        menubar = tk.Menu(root)
        root.config(menu=menubar)

        file_menu = tk.Menu(menubar, tearoff=0)
        menubar.add_cascade(label='ファイル', menu=file_menu)
        file_menu.add_command(label='画像を開く', command=self.open_image)

    # スクロール付きキャンバス
    canvas_frame = tk.Frame(root)
    canvas_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

    self.canvas = tk.Canvas(canvas_frame, bg='gray90')

    # スクロールバー
    v_scrollbar = tk.Scrollbar(canvas_frame, orient=tk.VERTICAL,
                               command=self.canvas.yview)
    h_scrollbar = tk.Scrollbar(canvas_frame, orient=tk.HORIZONTAL,
                               command=self.canvas.xview)

    self.canvas.configure(yscrollcommand=v_scrollbar.set,
                         xscrollcommand=h_scrollbar.set)

    # レイアウト
    self.canvas.grid(row=0, column=0, sticky='nsew')
    v_scrollbar.grid(row=0, column=1, sticky='ns')
    h_scrollbar.grid(row=1, column=0, sticky='ew')

    canvas_frame.grid_rowconfigure(0, weight=1)
    canvas_frame.grid_columnconfigure(0, weight=1)

    self.image_id = None

    def open_image(self):
        file_path = filedialog.askopenfilename(
            filetypes=[('画像ファイル', '*.png *.jpg *.jpeg *.gif *.bmp')])

        if file_path:
            try:
                # 画像を読み込み
                image = Image.open(file_path)
                self.photo = ImageTk.PhotoImage(image)

                # 既存の画像を削除
                if self.image_id:
                    self.canvas.delete(self.image_id)

                # 新しい画像を表示
                self.image_id = self.canvas.create_image(0, 0, anchor=tk.NW,
                                                       image=self.photo)

                # スクロール領域を設定
                self.canvas.configure(scrollregion=self.canvas.bbox('all'))

            except Exception as e:
                tk.messagebox.showerror('エラー', f'画像を開けませんでした: {e}')

root = tk.Tk()
app = ImageViewer(root)
root.mainloop()
```

## ベストプラクティス

項目	説明	例
座標系の理解	左上がり(0,0)、右下に向かって増加	create_line(0, 0, 100, 100)
アイテムIDの管理	描画したアイテムのIDを保存して後で操作	self.rect_id = canvas.create_rectangle(...)
タグの活用	複数のアイテムをグループ化	create_rectangle(..., tags='group1')
イベント処理	マウスやキーボードイベントを適切に処理	canvas.bind('<Button-1>', self.on_click)
パフォーマンス	大量のアイテムを扱う場合は定期的にdeleteで削除	canvas.delete('temporary_items')
座標変換	ウィンドウサイズに応じた座標変換を実装	スケーリング関数を作成
アニメーション	after() メソッドを使用してスムーズなアニメーション	root.after(50, self.animate)
メモリ管理	画像リファレンスを保持してガベージコレクションを防ぐ	self.photo = ImageTk.PhotoImage(...)

## 描画オプション

### 共通オプション

オプション	説明	例
<code>fill</code>	塗りつぶし色	<code>fill='red'</code>
<code>outline</code>	輪郭線の色	<code>outline='black'</code>
<code>width</code>	線の太さ	<code>width=3</code>
<code>tags</code>	タグ (グループ化用)	<code>tags='group1'</code>
<code>state</code>	状態 (normal/disabled/hidden)	<code>state='disabled'</code>

### テキスト専用オプション

オプション	説明	例
<code>text</code>	表示テキスト	<code>text='Hello World'</code>
<code>font</code>	フォント	<code>font=('Arial', 12, 'bold')</code>
<code>anchor</code>	アンカー位置	<code>anchor='nw'</code>
<code>justify</code>	行揃え	<code>justify='center'</code>

### 参考リンク

- [Python tkinter Canvas公式ドキュメント](#)
- [tkinter Canvas チュートリアル](#)
- [Canvas座標系について](#)

# tkinter.Toplevel リファレンス

## 概要

Toplevelウィジェットは、メインウィンドウとは別の新しいウィンドウを作成するためのウィジェットです。ダイアログボックス、設定ウィンドウ、サブウィンドウなどの作成に使用され、モーダル（他の操作をブロック）または非モーダル（並行操作可能）として動作させることができます。

## 基本的な使用方法

```
import tkinter as tk

root = tk.Tk()
root.title("メインウィンドウ")

def open_sub_window():
    sub_window = tk.Toplevel(root) # 親ウィンドウを指定
    sub_window.title("サブウィンドウ")
    sub_window.geometry("300x200")

    tk.Label(sub_window, text="これはサブウィンドウです").pack(pady=20)
    tk.Button(sub_window, text="閉じる", command=sub_window.destroy).pack()

tk.Button(root, text="サブウィンドウを開く", command=open_sub_window).pack(pady=20)

root.mainloop()
```

## 親ウィンドウの指定について

### ベストプラクティス（推奨）

```
# 親ウィンドウを明示的に指定
sub_window = tk.Toplevel(parent)
```

### アンチパターン（非推奨）

```
# 親ウィンドウを指定しない
sub_window = tk.Toplevel()
```

## 親ウィンドウ指定の有無による違い

項目	親指定あり	親指定なし
ウィンドウ階層	親ウィンドウに属する	独立したトップレベルウィンドウ
最小化連動	親と連動して最小化	独立して最小化
タスクバー表示	親のグループに含まれる	個別にタスクバーに表示
親ウィンドウ終了時	自動的に閉じられる	残存する可能性がある（ゾンビプロセス）
Z-order管理	親より前面に表示	システム依存

## リスクレベル

- Critical:** 親ウィンドウ終了時にサブウィンドウが残存（メモリリーク、ゾンビプロセス）
- Major:** ユーザビリティ低下（タスクバーの混乱、ウィンドウ管理の困難）
- Minor:** 一貫性のないウィンドウ動作

## モーダル・非モーダルの使い分け

### モーダルダイアログ（他の操作をブロック）

使用場面： - ユーザーからの入力が必須の場合 - 設定変更の確認が必要な場合  
- エラーメッセージの表示 - ファイル保存前の未保存データ警告 - データ削除の確認

実装方法：

```

dialog = SomeDialog(parent)
dialog.transient(parent) # 親に関連付け
dialog.grab_set() # 他の操作をブロック
parent.wait_window(dialog) # ダイアログが閉じるまで待機

```

## 非モーダルウィンドウ（並行操作可能）

使用場面： - ツールパレット、プロパティパネル - ログ表示ウィンドウ - リアルタイム情報表示 - 補助的な操作ウィンドウ - マルチタスクが必要な作業

実装方法：

```

window = SomeWindow(parent)
window.transient(parent) # 親に関連付けるが、grab_setは呼ばない

```

## 主要なオプション

オプション	説明	デフォルト値	例
master	親ウィンドウ	None	master=root
class_	ウィンドウクラス名	'Toplevel'	class_='Dialog'
bg	背景色	SystemButtonFace	bg='white'
width	ウィンドウの幅	-	width=400
height	ウィンドウの高さ	-	height=300
relief	境界線のスタイル	'flat'	relief='raised'
borderwidth	境界線の幅	0	borderwidth=2

## 主要なメソッド

メソッド	説明	例
geometry(geometry)	ウィンドウサイズと位置を設定	geometry('400x300+100+50')
title(string)	ウィンドウタイトルを設定	title('設定ダイアログ')
grab_set()	モーダルにする（他の操作をブロック）	grab_set()
grab_release()	モーダルを解除	grab_release()
transient(master)	親ウィンドウに関連付け	transient(root)
focus_set()	フォーカスを設定	focus_set()
iconify()	ウィンドウを最小化	iconify()
deiconify()	最小化を解除	deiconify()
withdraw()	ウィンドウを非表示	withdraw()
destroy()	ウィンドウを閉じる	destroy()
protocol(name, func)	ウィンドウプロトコルを設定	protocol('WM_DELETE_WINDOW', on_close)

## 重複開防止のベストプラクティス

```

class MainApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.dialog = None # ダイアログ参照を保持

    def open_dialog(self):
        # 重複開防止
        if self.dialog and self.dialog.winfo_exists():
            self.dialog.lift() # 既存ダイアログを前面に
            return

        self.dialog = SomeDialog(self)

```

## 実用的な例

### 1. シンプルなモーダルダイアログ

```

import tkinter as tk
from tkinter import messagebox

```

```

class SimpleModalApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("モーダルダイアログテスト")
        self.geometry("300x200")
        self.current_dialog = None

        tk.Button(self, text="ダイアログを開く", command=self.open_dialog).pack(pady=20)
        tk.Button(self, text="テストボタン", command=self.test_click).pack()

    def open_dialog(self):
        if self.current_dialog and self.current_dialog.winfo_exists():
            self.current_dialog.lift()
            return

        dialog = SimpleDialog(self)
        self.current_dialog = dialog
        self.wait_window(dialog) # ここで処理が停止
        self.current_dialog = None

    def test_click(self):
        messagebox.showinfo("テスト", "ダイアログが開いている間はクリックできません")

class SimpleDialog(tk.Toplevel):
    def __init__(self, parent):
        super().__init__(parent)
        self.title("モーダルダイアログ")
        self.geometry("200x100")

        self.transient(parent) # 重要:親に関連付け
        self.grab_set() # 重要:他の操作をブロック

        tk.Label(self, text="モーダルダイアログです").pack(pady=10)
        tk.Button(self, text="閉じる", command=self.destroy).pack()

if __name__ == "__main__":
    app = SimpleModalApp()
    app.mainloop()

```

## 2. シンプルな非モーダルウィンドウ

```

import tkinter as tk

class SimpleNonModalApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("非モーダルウィンドウテスト")
        self.geometry("300x200")
        self.sub_window = None

        button_frame = tk.Frame(self)
        button_frame.pack(pady=20)

        tk.Button(button_frame, text="サブウィンドウを開く",
                  command=self.open_sub_window).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="閉じる",
                  command=self.close_sub_window).pack(side=tk.LEFT, padx=5)

        tk.Button(self, text="テストボタン", command=self.test_click).pack(pady=10)

    def open_sub_window(self):
        if self.sub_window and self.sub_window.winfo_exists():
            self.sub_window.lift()
            return

        self.sub_window = SubWindow(self)

    def close_sub_window(self):
        if self.sub_window and self.sub_window.winfo_exists():
            self.sub_window.destroy()
        self.sub_window = None

    def test_click(self):
        print("サブウィンドウが開いていてもクリックできます")

class SubWindow(tk.Toplevel):
    def __init__(self, parent):
        super().__init__(parent)
        self.parent = parent
        self.title("非モーダルウィンドウ")
        self.geometry("200x150")

        self.transient(parent) # 親に関連付け(grab_setは呼ばない)

        tk.Label(self, text="非モーダルウィンドウです").pack(pady=10)

        self.counter = 0
        self.label = tk.Label(self, text=f"カウンター: {self.counter}")
        self.label.pack()

        tk.Button(self, text="+1", command=self.increment).pack(pady=5)
        tk.Button(self, text="閉じる", command=self.on_close).pack()

        self.protocol("WM_DELETE_WINDOW", self.on_close)

    def increment(self):
        self.counter += 1

```

```
    self.label.config(text=f"カウンター: {self.counter}")

def on_close(self):
    self.parent.sub_window = None
    self.destroy()

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

## 注意点

- **親ウィンドウの指定:** 必ず親ウィンドウを指定してメモリリークを防ぐ
- **重複開防止:** 同じダイアログの重複開を適切に制御する
- **モーダルの適切な使用:** ユーザーの応答が必須の場合のみモーダルを使用
- **リソース管理:** 不要になったウィンドウは確実に `destroy()` する
- **フォーカス管理:** 適切な要素にフォーカスを設定してキーボード操作を改善

## t920\_keybinding.md - キーバインディング

`tkinter` では、キーボードショートカットやキーイベントをコントロールにバインドすることができます。これにより、特定のキーでボタンを押したような動作をさせたり、フォーカスを移動したりすることができます。

### 基本的なキーバインディング

#### bind() メソッド

すべての `tkinter` ウィジェットは `bind()` メソッドを持っており、キーイベントやマウスイベントを関数にバインドできます。

```
widget.bind('<KeyPattern>', callback_function)
```

#### キーパターンの書式

パターン	説明	例
<Key>	任意のキー	<Key>
<Return>	Enterキー	<Return>
<Escape>	Escapeキー	<Escape>
<Tab>	Tabキー	<Tab>
<BackSpace>	Backspaceキー	<BackSpace>
<Delete>	Deleteキー	<Delete>
<F1> - <F12>	ファンクションキー	<F1>, <F5>
<Up>, <Down>, <Left>, <Right>	矢印キー	<Up>
<Control-c>	Ctrl+C	<Control-c>
<Alt-f>	Alt+F	<Alt-f>
<Shift-Tab>	Shift+Tab	<Shift-Tab>
<Control-Alt-d>	Ctrl+Alt+D	<Control-Alt-d>

### ショートカットでボタンを実行

#### 基本例

```
import tkinter as tk

class App:
    def __init__(self, root):
        self.root = root

        # ボタンを作成
        self.button = tk.Button(root, text="クリックされました", command=self.button_click)
        self.button.pack(pady=10)

        # F1キーでボタンをトリガー
        root.bind('<F1>', lambda e: self.button_click())
        # Ctrl+Enterでもトリガー
        root.bind('<Control-Return>', lambda e: self.button_click())

    def button_click(self):
        print("ボタンがクリックされました!")

root = tk.Tk()
app = App(root)
root.mainloop()
```

#### invoke() メソッドを使用

```
# ボタンの invoke() メソッドを直接呼び出す
root.bind('<F1>', lambda e: self.button.invoke())
```

### フォーカス移動

#### focus\_set() でフォーカス移動

```

import tkinter as tk

class FocusApp:
    def __init__(self, root):
        self.root = root

        # 複数のエントリーウィジェット
        self.entry1 = tk.Entry(root)
        self.entry1.pack(pady=5)

        self.entry2 = tk.Entry(root)
        self.entry2.pack(pady=5)

        self.entry3 = tk.Entry(root)
        self.entry3.pack(pady=5)

        # キーバインディング
        root.bind('<F2>', lambda e: self.entry1.focus_set())
        root.bind('<F3>', lambda e: self.entry2.focus_set())
        root.bind('<F4>', lambda e: self.entry3.focus_set())

        # Tabキーで次のウィジェットへ (デフォルト動作を補完)
        root.bind('<Control-Tab>', self.focus_next)
        root.bind('<Control-Shift-Tab>', self.focus_prev)

    def focus_next(self, event):
        event.widget.tk_focusNext().focus_set()

    def focus_prev(self, event):
        event.widget.tk_focusPrev().focus_set()

```

## 高度なキーバインディング

### グローバルキーバインディング

```

import tkinter as tk

class GlobalKeyApp:
    def __init__(self, root):
        self.root = root

        # メニューバー
        menubar = tk.Menu(root)
        root.config(menu=menubar)

        file_menu = tk.Menu(menubar, tearoff=0)
        menubar.add_cascade(label="ファイル", menu=file_menu)
        file_menu.add_command(label="新規", command=self.new_file, accelerator="Ctrl+N")
        file_menu.add_command(label="開く", command=self.open_file, accelerator="Ctrl+O")
        file_menu.add_command(label="保存", command=self.save_file, accelerator="Ctrl+S")

        # テキストエリア
        self.text_area = tk.Text(root)
        self.text_area.pack(fill=tk.BOTH, expand=True)

        # グローバルキーバインディング
        self.bind_global_keys()

    def bind_global_keys(self):
        # ウィンドウ全体に対してバインド
        self.root.bind('<Control-n>', lambda e: self.new_file())
        self.root.bind('<Control-o>', lambda e: self.open_file())
        self.root.bind('<Control-s>', lambda e: self.save_file())
        self.root.bind('<Control-q>', lambda e: self.root.quit())

        # カスタムショートカット
        self.root.bind('<Control-d>', lambda e: self.duplicate_line())
        self.root.bind('<Control-slash>', lambda e: self.toggle_comment())

    def new_file(self):
        self.text_area.delete(1.0, tk.END)
        print("新しいファイル")

    def open_file(self):
        print("ファイルを開く")

    def save_file(self):
        print("ファイルを保存")

    def duplicate_line(self):
        # 現在の行を複製
        current_line = self.text_area.index(tk.INSERT).split('.')[0]
        line_content = self.text_area.get(f'{current_line}.0', f'{current_line}.end')
        self.text_area.insert(f'{current_line}.end', f'\n{line_content}')

    def toggle_comment(self):
        print("コメントの切り替え")

```

### 動的なキーバインディング

```

import tkinter as tk

```

```

class DynamicKeyApp:
    def __init__(self, root):
        self.root = root

        self.label = tk.Label(root, text="キーを押してください", font=("Arial", 14))
        self.label.pack(pady=20)

        self.binding_enabled = True

        # 動的バインディング
        self.setup_number_keys()

        # バインディングの切り替えボタン
        self.toggle_btn = tk.Button(root, text="バインディング切り替え",
                                    command=self.toggle_binding)
        self.toggle_btn.pack(pady=10)

    def setup_number_keys(self):
        # 数字キー 1-9 にバインド
        for i in range(1, 10):
            self.root.bind(f'<Key-{i}>', lambda e, num=i: self.number_pressed(num))

    def number_pressed(self, number):
        if self.binding_enabled:
            self.label.config(text=f"数字 {number} が押されました")

    def toggle_binding(self):
        self.binding_enabled = not self.binding_enabled
        status = "有効" if self.binding_enabled else "無効"
        self.toggle_btn.config(text=f"バインディング: {status}")

```

## コンテキスト固有のキーバインディング

### ウィジェット固有のバインディング

```

import tkinter as tk
from tkinter import ttk

class ContextKeyApp:
    def __init__(self, root):
        self.root = root

        # ノートブック (タブ)
        notebook = ttk.Notebook(root)
        notebook.pack(fill=tk.BOTH, expand=True)

        # テキストタブ
        text_frame = tk.Frame(notebook)
        notebook.add(text_frame, text="テキスト")

        self.text_widget = tk.Text(text_frame)
        self.text_widget.pack(fill=tk.BOTH, expand=True)

        # リストタブ
        list_frame = tk.Frame(notebook)
        notebook.add(list_frame, text="リスト")

        self.listbox = tk.Listbox(list_frame)
        self.listbox.pack(fill=tk.BOTH, expand=True)
        for i in range(20):
            self.listbox.insert(tk.END, f"アイテム {i+1}")

        # コンテキスト固有のバインディング
        self.setup_context_bindings()

    def setup_context_bindings(self):
        # テキストウィジェット固有
        self.text_widget.bind('<Control-d>', self.duplicate_text_line)
        self.text_widget.bind('<Control-k>', self.delete_text_line)

        # リストボックス固有
        self.listbox.bind('<Delete>', self.delete_list_item)
        self.listbox.bind('<Control-a>', self.select_all_items)

    def duplicate_text_line(self, event):
        widget = event.widget
        current_line = widget.index(tk.INSERT).split('.')[0]
        line_content = widget.get(f'{current_line}.0', f'{current_line}.end')
        widget.insert(f'{current_line}.end', f'\n{line_content}')

    def delete_text_line(self, event):
        widget = event.widget
        current_line = widget.index(tk.INSERT).split('.')[0]
        widget.delete(f'{current_line}.0', f'{int(current_line)+1}.0')

    def delete_list_item(self, event):
        selection = self.listbox.curselection()
        if selection:
            self.listbox.delete(selection[0])

    def select_all_items(self, event):
        self.listbox.select_set(0, tk.END)

```

## アクセラレータキー（メニュー用）

### accelerator オプション

```
import tkinter as tk

class AcceleratorApp:
    def __init__(self, root):
        self.root = root

        # メニューバー
        menubar = tk.Menu(root)
        root.config(menu=menubar)

        # ファイルメニュー
        file_menu = tk.Menu(menubar, tearoff=0)
        menubar.add_cascade(label="ファイル", menu=file_menu)

        # accelerator オプションでショートカット表示
        file_menu.add_command(label="新規", command=self.new_file, accelerator="Ctrl+N")
        file_menu.add_command(label="開く", command=self.open_file, accelerator="Ctrl+O")
        file_menu.add_command(label="保存", command=self.save_file, accelerator="Ctrl+S")
        file_menu.add_separator()
        file_menu.add_command(label="終了", command=root.quit, accelerator="Ctrl+Q")

        # 実際のキー/"インディング" (accelerator は表示のみ)
        root.bind('<Control-n>', lambda e: self.new_file())
        root.bind('<Control-o>', lambda e: self.open_file())
        root.bind('<Control-s>', lambda e: self.save_file())
        root.bind('<Control-q>', lambda e: root.quit())

        # underline オプションでアクセスキー
        edit_menu = tk.Menu(menubar, tearoff=0)
        menubar.add_cascade(label="編集", menu=edit_menu, underline=0) # Eキー
        edit_menu.add_command(label="コピー", command=self.copy, underline=0) # Cキー
        edit_menu.add_command(label="貼り付け", command=self.paste, underline=0) # 貼キー

    def new_file(self):
        print("新規ファイル")

    def open_file(self):
        print("ファイルを開く")

    def save_file(self):
        print("ファイルを保存")

    def copy(self):
        print("コピー")

    def paste(self):
        print("貼り付け")
```

## 実践的な応用例

### キーボードショートカット一覧表示

```
import tkinter as tk
from tkinter import ttk

class ShortcutDisplayApp:
    def __init__(self, root):
        self.root = root
        root.title("キーボードショートカット例")

        # ショートカット一覧
        self.shortcuts = {
            'F1': 'ヘルプを表示',
            'Ctrl+N': '新規作成',
            'Ctrl+O': 'ファイルを開く',
            'Ctrl+S': '保存',
            'Ctrl+Z': '元に戻す',
            'Ctrl+Y': 'やり直し',
            'Esc': 'キャンセル',
            'Enter': '実行'
        }

        # UI作成
        self.create_ui()
        self.setup_bindings()

    def create_ui(self):
        # メインフレーム
        main_frame = tk.Frame(self.root)
        main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # ショートカット一覧表示
        tree = ttk.Treeview(main_frame, columns=('action',), show='tree headings')
        tree.heading('#0', text='ショートカット')
        tree.heading('action', text='動作')

        for shortcut, action in self.shortcuts.items():
            tree.insert('', tk.END, text=shortcut, values=(action,))

    def setup_bindings(self):
        pass
```

```

tree.pack(fill=tk.BOTH, expand=True)

# ステータスラベル
self.status_label = tk.Label(main_frame, text="キーを押してください...", relief=tk.SUNKEN, anchor=tk.W)
self.status_label.pack(fill=tk.X, pady=(10, 0))

def setup_bindings(self):
    self.root.bind('<F1>', lambda e: self.show_action('ヘルプを表示'))
    self.root.bind('<Control-n>', lambda e: self.show_action('新規作成'))
    self.root.bind('<Control-o>', lambda e: self.show_action('ファイルを開く'))
    self.root.bind('<Control-s>', lambda e: self.show_action('保存'))
    self.root.bind('<Control-z>', lambda e: self.show_action('元に戻す'))
    self.root.bind('<Control-y>', lambda e: self.show_action('やり直し'))
    self.root.bind('<Escape>', lambda e: self.show_action('キャンセル'))
    self.root.bind('<Return>', lambda e: self.show_action('実行'))

def show_action(self, action):
    self.status_label.config(text=f"実行: {action}")
    self.root.after(2000, lambda: self.status_label.config(text="キーを押してください..."))

```

## まとめ

機能	説明	使用方法
基本バインディング	キーイベントを関数にバインド	widget.bind('<Key>', callback)
フォーカス移動	特定のウィジエットにフォーカスを設定	widget.focus_set()
グローバルバインディング	アプリケーション全体で有効なショートカット	root.bind('<Control-s>', callback)
コンテキストバインディング	特定のウィジエットでのみ有効	text_widget.bind('<Control-d>', callback)
アクセラレータ	メニューでのショートカット表示	accelerator="Ctrl+S"
アクセスキー	Altキーでのメニューアクセス	underline=0

## ベストプラクティス

1. **一貫性:** 標準的なショートカット (Ctrl+C, Ctrl+V等) を使用
2. **表示:** `accelerator` オプションでユーザーにショートカットを示す
3. **コンテキスト:** ウィジエット固有のショートカットは適切なスコープで定義
4. **無効化:** 必要に応じてバインディングの有効/無効を切り替え
5. **ドキュメント:** 利用可能なショートカットをユーザーに明示