Tkinter objektorientiert

Ein Fenster mit einem Label erstellen:

```
from tkinter import *

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

        self.title("Tkinter Window")
        self.geometry("800x600")

        # widgets:
        self.my_label = Label(
            text="Some label",
            font=("Carlito", 42)
            )
        self.my_label.pack(pady=20)

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

1 Erläuterungen

```
1 import tkinter as tk
```

importiert das ganze Modul unter dem Alias tk, sodass man alles mit tk. aufruft (z.B. tk.Tk()), was den Code klar und konfliktfrei hält. Allen Tkinter-Funktionen und -Klassen muss `tk.` vorangestellt werden (z.B. tk.Button, tk.Label).

```
1
2 from tkinter import *
```

importiert alle Methoden und Klassen direkt in den Namensraum, was Tipparbeit spart, aber bei umfangreicheren Projekten zu Namenskonflikten führen kann und den Code unübersichtlicher macht (z.B. Tk() ohne Präfix)

```
1     class App(Tk):
2         def __init__(self):
3             super().__init__()
```

- In Zeile 1 wird eine neue Klasse mit dem Namen App erstellt, sie erbt von der Klasse Tk

Tk()-Objekt: Wenn man in Python Tk() aufruft (oft in Kombination mit import tkinter as tk), erstellt man das Hauptfenster (die Wurzel-Instanz) der Anwendung, auf dem alle anderen Widgets (Buttons, Labels, Textfelder) platziert werden.

- In Zeile 2 wird ein Konstruktor erstellt. Das ist eine Methode, die immer dann automatisch aufgerufen wird, wenn von der Klasse App eine Instanz, also ein konkretes Objekt, erstellt wird. Der Konstruktor bestimmt, wie das App-Fenster aussehen soll und welche Komponenten - Widgets - es enthalten soll.

def: Leitet die Definition einer Funktion oder Methode ein.

init: Der spezielle Name der Initialisierungsmethode. Doppel-Unterstriche am Anfang und Ende kennzeichnen spezielle Methoden in Python (sogenannte “Dunder-Methoden”).

self: Der erste Parameter jeder Instanzmethode einer Klasse. Er ist eine Referenz auf die spezifische Instanz (das Objekt), für das die Methode gerade ausgeführt wird, und wird verwendet, um Attribute wie self.name = “Wert” zu setzen.

- Zeile 3 ruft die init-Methode, also wie oben, den Konstruktor, aber den der direkten Elternklasse auf, um deren Initialisierungscode auszuführen, bevor die untergeordnete Klasse eigene Attribute setzt.

`super()`: Eine eingebaute Funktion, die ein temporäres Objekt zurückgibt, das Methoden der Elternklasse (Superklasse) repräsentiert. `super().__init__()`: Dies bedeutet also: “Nimm das Objekt der Elternklasse (via super()) und rufe dessen init-Methode auf”.

```
1         self.title("Tkinter Window")
2         self.geometry("800x600")
```

Hier werden Eigenschaften des App-Fensters festgelegt: Die Methode `title()` legt fest, wie der Fenstertitel aussiet. Sie erhält als Parameter einen (beliebigen) Text, der immer in doppelten Anführungszeichen stehen muss. In gleicher Weise bestimmt Zeile 2 die Größe des App-Fensters mit Hilfe der Methode `geometry()` und den Parametern für Höhe und Breite (in Pixeln).

Beiden Methoden ist `self`vorangestellt. `self` signalisiert, dass es sich um eine Instanzmethode handelt, also um eine Methode die nur in Verbindung mit einer Instanz, einem konkresten Objekt der Klasse App aufgerufen werden kann. `self` repräsentiert das spezifische Fensterobjekt, das Sie gerade bearbeiten.

```
1         self.my_label = Label(text="Some label", font=("Carlito", 42))
```

Diese Zeile erstellt ein Label, also eine einfache Textzeile.

Hier zeigt `self` an, dass es sich bei `my_label` Instanzvariable handelt, die ein konkretes Objekt der Klasse Label mit den als Parameter definierten Eigenschaften speichert.

Instanzvariable bedeutet, dass jede Instanz (jedes spezifische Objekt) der übergeordneten Klasse (z. B. App) ihre eigene, einzigartige Version dieser Variable speichert.

Label() ist der Aufruf, der eine neue Instanz (ein neues Objekt) der Klasse Label erstellt. In der objektorientierten Programmierung nennt man dies einen Konstruktor. Wenn Sie Label(…) aufrufen, führt Python intern die spezielle Methode init dieser Klasse aus. Diese Methode "konstruiert" das Objekt im Speicher und initialisiert es mit den von Ihnen bereitgestellten Parametern (wie text="Some label" und font=("Carlito", 42)). Aus der Perspektive des Aufrufs (Label(…)) sieht es syntaktisch genauso aus wie ein normaler Funktionsaufruf. Sie übergeben Argumente und erhalten ein Rückgabeergebnis (das neue Label-Objekt). Label (ohne die Klammern) ist der Name der Klasse (die Blaupause oder Schablone) für alle Tkinter-Labels.

```
        self.my_label.pack(pady=20)
```

Damit das Label im Fenster angezeigt wird, muss es noch dem Fenster hinzugefügt werden. Dies erfolgt mit der Methode `pack()`. Sie wird auf der Instanzvariablen `my_label` aufgerufen und ist für das Layout-Management zuständig. Die Methode kann Parameter entgegennehmen, die die Position des Labels im Fenster konkretisieren. `pady=20` legt einen externen Abstand (Padding) von 20 Pixeln auf der y-Achse, oberhalb und unterhalb des Labels fest, sowohl zum Rand des Fensters als auch zu anderen Widgets, die eventuell darüber oder darunter liegen.

```
app = App()
```

Ähnlich wie oben beim Erstellen des Labels wird hier eine Instanz, ein konkretes Objekt der Klasse App mit dem Konstruktor `App()` erstellt und in der Variablen app gespeichert.

app repräsentiert das Objekt, das in der Variable app gespeichert ist, die Hauptinstanz, das Hauptfenster der gesamten Anwendung.

```
app.mainloop()
```

Die Methode `mainloop()` startet die Ereignisschleife (event loop) von Tkinter.

Blockiert die Ausführung: Sobald mainloop() aufgerufen wird, stoppt die weitere Ausführung des Python-Skripts an dieser Stelle, bis das Fenster geschlossen wird.

Lauscht auf Ereignisse: In dieser Schleife wartet Tkinter ununterbrochen auf Benutzereingaben (Mausklicks, Tastatureingaben), Systemereignisse (Fenstergröße ändern, Fenster minimieren) und Aktualisierungen der Benutzeroberfläche. Verarbeitet Ereignisse: Wenn ein Ereignis auftritt, leitet die Schleife es an das entsprechende Widget und die dazugehörigen Funktionen (Callbacks oder Event-Handler) weiter, die Sie im Code definiert haben (z.B. eine Funktion, die aufgerufen wird, wenn ein Button geklickt wird).

Hält das Fenster geöffnet: Ohne diese Schleife würde Ihr Python-Skript einfach durchlaufen, alle Widgets erstellen und sofort beenden. Das Fenster würde also sofort wieder verschwinden. Zusammenfassung: app.mainloop() ist die Methode, die sicherstellt, dass das Fenster geöffnet bleibt und auf Interaktion des Benutzers reagieren kann. Es ist das Herzstück jeder Tkinter-Anwendung.

2 Reihenfolge der Ausführung:

Vorbereitung (Code wird ausgeführt): - Das Skript startet, - die Klasse App wird definiert, - das Hauptfenster wird erstellt (app = App()), - alle Widgets (self.my_label) werden erstellt und positioniert (pack()).

Bis hierhin läuft alles sequentiell durch.

Ereignisschleife startet und blockiert (Hier stoppt die sequentielle Ausführung):

- Die Zeile app.mainloop() wird erreicht.
- Das Programm hält hier an und übergibt die Kontrolle an Tkinter, um auf Benutzerinteraktionen zu warten.

Nach mainloop() (Code wird angehalten): Jeglicher Python-Code, der direkt unter app.mainloop() steht, wird erst dann ausgeführt, wenn das Fenster vom Benutzer geschlossen wird und die mainloop()-Methode beendet ist.

mainloop() stoppt nicht die Erstellung der Fenster und Widgets – diese sind ja bereits erstellt worden, bevor mainloop() aufgerufen wurde. Es stoppt die sofortige Ausführung des restlichen Skripts und beginnt mit dem Warten auf Ereignisse.