Overview

This lesson explains what the Tkinter event loop is and why it is required for a GUI to stay alive, respond to the user, and update the screen.

Important Information

What the Event Loop Is

When you run a normal terminal program, Python starts at the top and runs each line once until the program ends. A GUI program is different because it must keep running while it waits for the user to do things like click, type, or resize a window.

In Tkinter, this "keep running and keep listening" behavior is handled by the **event loop**. After you build the window and widgets, you start the event loop using `root.mainloop()`. From that point on, Tkinter repeatedly checks for new events and handles them many times per second.

The Event Loop Does Three Big Jobs

The event loop is constantly cycling through three types of work:

1. Taking the next event from the event queue (like a click or key press)
2. Running the correct callback or event binding for that event
3. Processing screen updates (redrawing widgets when needed)

Because it loops so quickly, it feels like everything happens instantly.

Visualizing the Event Loop

The important idea is that your code briefly "steps out" of the loop when a callback runs, then returns control back to the event loop.

What "Blocking the Event Loop" Means

If your callback takes a long time (like a slow loop, a huge calculation, or waiting on the internet), the event loop cannot do its job. When that happens:

- the window won't respond to clicks
- the screen won't redraw
- the program looks frozen

This is called **blocking** the event loop.

A common beginner mistake is doing long work directly inside a button click function.

```
def on_click():
    # Bad idea: this blocks the event loop for a long time
    total = 0
    for i in range(50_000_000):
        total += i
```

The Most Important Rule

Event handlers must do their work quickly and return control to the event loop.

If something will take a long time, you should break it into small steps and let the event loop schedule those steps over time.

Doing Long Work One Step at a Time with `after`

Tkinter can create **timer events** using `after`. You give it a delay in milliseconds and a function to call later. That function should do a small piece of work and then schedule the next step.

```
from tkinter import *
from tkinter import ttk

root = Tk()

progress = ttk.Progressbar(root, maximum=20)
progress.grid(row=0, column=0, padx=10, pady=10)

step_number = 0

def do_step():
    global step_number

    step_number += 1
    progress["value"] = step_number

    if step_number < 20:
        root.after(100, do_step)  # schedule the next step in 100 ms

start_btn = ttk.Button(root, text="Start", command=do_step)
start_btn.grid(row=1, column=0, padx=10, pady=10)

root.mainloop()
```

This keeps the UI responsive because each step is short, and control returns to the event loop between steps.

Running Code When Tkinter Is Idle with `after_idle`

Sometimes you want something to run only when Tkinter has nothing urgent to do. `after_idle` requests a callback when the event queue is empty. Screen redraw work often happens during idle time.

```
def run_when_idle():
    print("Tkinter is idle right now.")

root.after_idle(run_when_idle)
```